OLIMPÍADA INFORMÁTICA ARGENTINA

Cuaderno de actividades 2021

Niveles 1-2-3

Categoría Programación

Organiza:

UNSAM

Auspicia y financia:

Ministerio de Educación
Presidencia de la Nación

Bienvenidxs!!!

El presente manual está dirigido a docentes y alumnxs del nivel medio que deseen participar en los certámenes de programación organizados por la Olimpíada Informática Argentina.

Nuestro equipo técnico pedagógico a preparado, clasificado y ordenado material de capacitación y entrenamiento necesario para poder participar en las distintas instancias de la categoría programación.

El manual es un conjunto de documentos y herramientas que permiten alcanzar los siguientes objetivos, según el nivel de conocimientos del alumno:

- Aprender desde cero los rudimentos de un lenguaje visual.
- Resolver desafíos didácticos básicos con un lenguaje visual.
- Aprender desde cero los rudimentos de los lenguajes de programación utilizados en las competencias OIA.
- Aprender técnicas de resolución de problemas.

Los documentos:

- ✓ OIA-Wiki
- ✓ Solucionario

Las herramientas:

- ✓ OIA-Juez
- ✓ OIA-Foro

Esperamos que nuestra propuesta estimule el interés de los estudiantes en programar y participar en las actividades del programa OIA.

El equipo OIA

Índice

Introducción

¿Qué es una computadora?

Una computadora es un dispositivo electrónico [No es esencial que sea electrónico, pero todas las computadoras desde aproximadamente 1960 lo son. Las computadoras utilizadas en la segunda guerra mundial eran fundamentalmente electromecánicas] utilizado para procesar información y obtener resultados .

Los datos e información se pueden introducir en una computadora como entrada , y a continuación se procesan para producir una salida .

Los componentes físicos que constituyen la computadora forman el hardware .

Un conjunto de instrucciones que hacen funcionar a la computadora se denomina un programa . Se denomina programador a una persona que escribe programas.

El software es el conjunto de todos los programas de una computadora.

Organización de una computadora

Los componentes de una computadora pueden dividirse de la siguiente manera:

- Dispositivos de entrada. Son los que permiten introducir datos en la compu, que irán a parar a la memoria (principal o externa). Algunos ejemplos son:
    - Teclados
    - Lectores de códigos de barras (utilizados por computadoras en supermercados)
    - Lápices ópticos
    - Joysticks
    - Mouse
    - Scanner
    - Micrófonos
    - Placas de red
- Dispositivos de salida. Son los que permiten representar resultados del procesamiento de los datos. Algunos ejemplos son:
    - Pantalla
    - Impresoras
    - Parlantes
    - Placas de red
    - Motores eléctricos en las articulaciones de un robot
- CPU (procesador). Dirige y controla todo el procesamiento y movimiento de la información . Se considera "el cerebro" de una computadora, por analogía con el cerebro humano.
- Memoria principal (RAM):
    - Permite almacenar información y datos utilizados por la computadora en todos sus cálculos y procesamiento de datos .
    - Está compuesta de muchísimas celdas o unidades básicas de información (Típicamente bytes, compuestos por 8 bits, dígitos binarios).
    - Los datos en memoria RAM son temporarios : se pierden cuando la computadora se apaga.

- Memoria externa:
    - Discos rígidos, disquetes, memorias SD, pendrives USB, cintas magnéticas.
    - Permite almacenar y recuperar información desde un medio de almacenamiento permanente (no se pierde al apagar la computadora).
    - Es en ella donde se guardan todos los archivos , que son unidades independientes con datos en memoria externa, guardados en una carpeta bajo un nombre.

El software (los programas)

Una computadora típica tiene, incluso antes de que la comencemos a utilizar, ya instalados muchos programas fundamentales para su funcionamiento, que forman el software de sistema .

Uno de los programas más importantes del software de sistema es el sistema operativo , que realiza tareas generales de control y coordinación entre los distintos programas , permitiendo a todos usar la misma computadora y organizando las distintas operaciones que puede querer llevar a cabo el usuario (Ejemplos de sistemas operativos son Microsoft Windows [https://es.wikipedia.org/wiki/Microsoft_Windows], GNU/Linux [https://es.wikipedia.org/wiki/GNU/Linux], Mac OS X [https://es.wikipedia.org/wiki/Mac_OS_X], Android [https://es.wikipedia.org/wiki/Android]). También se encarga de cargar y poner en marcha los programas cuando el usuario quiere ejecutarlos.

Los programas que realizan tareas concretas que interesan al usuario (navegador de internet, sistema de contabilidad de una empresa, grabador de efectos de guitarra de un estudio musical, programas para realizar cálculos científicos, videojuegos, etc) se denominan programas de aplicación .

Para crear un programa de aplicación como los mencionados, un programador debe escribir las correspondientes instrucciones que indican a la computadora cómo operar , y estas se escriben en algún Lenguaje de programación (C,C++,Pascal, Java, C#, Haskell, Python, Javascript, Smalltalk, Go, Scala, y muchos, muchos otros).

La computadora solamente entiende las instrucciones en su propio lenguaje de máquina , que está compuesto solamente de ceros y unos y por lo tanto un humano no lo puede leer ni escribir fácilmente de manera directa (Esto es fácil de verificar abriendo un archivo ejecutable con un editor de texto). Para ello existen programas traductores (Los compiladores e intérpretes) que se encargan de traducir las instrucciones en el lenguaje de programación (que son las que entienden y usan los humanos) al lenguaje de máquina (ceros y unos). Un archivo ejecutable contiene un programa escrito en el lenguaje de máquina, y por eso puede ser ejecutado directamente por la computadora.

Los lenguajes de programación

Efectivamente, los programas son una "receta" o "lista de instrucciones" de qué hacer, que será seguida por la computadora. Al texto que describe estos programas, escrito en un cierto lenguaje de programación, se lo denomina código fuente .

Así como los mismos textos pueden escribirse en los distintos idiomas del mundo, y un mismo texto puede verse de manera muy diferente en cada uno de ellos; cada lenguaje de programación tiene sus reglas particulares sobre cómo se deben escribir las órdenes para la computadora.

El siguiente es un fragmento de programa en el lenguaje de programación C++ a modo de ejemplo:

```
int sumaDeLosDigitos(int numero)
{
    int sumaParcial = 0;
    while (numero > 0)
    {
        sumaParcial += numero % 10; // Suma la ultima cifra del numero al resultado intermedio "sumaParcial"
        numero /= 10; // Le borra la ultima cifra al numero
    }
    return sumaParcial;
}
```

Este es un ejemplo de programa sencillo que indica a la computadora cómo calcular la suma de los dígitos de un número dado cualquiera. Más adelante podremos entender bien los detalles de su funcionamiento, pero es importante ir destacando esta característica esencial de los programas, que es su generalidad : Al escribir un programa, le estamos enseñando a la computadora cómo procesar cualquier posible dato de entrada, de una manera general, y deberemos tener eso en cuenta al programar. El

ejemplo anterior por ejemplo explica cómo obtener la suma de los dígitos de un número cualquiera , que en el código fuente se ha denominado "numero".

La resolución de problemas con computadora

El proceso de resolución de un problema con una computadora conduce a la escritura de un programa y a su ejecución en la misma.

Aunque el proceso de diseñar programas es esencialmente un proceso creativo, se pueden considerar una serie de pasos o fases comunes a ser seguidos por todos los programadores.

- Análisis del problema [Entender qué tenemos que hacer, pensar, pensar, analizarlo]
- Diseño del algoritmo [Diseñar un método para resolverlo, una idea, tener claro el "cómo" es que se resuelve]
- Codificación ("codeo", "codear", "escribir el programa") [Escribir un programa que le explique ese "cómo" del paso anterior a la computadora]
- Compilación y ejecución [Generar un archivo ejecutable y ejecutarlo en la computadora para obtener los resultados]
- Verificación [Contrastar los resultados contra lo esperado en distintos casos de tests]
- Depuración [Entender y corregir todos los errores en el programa que vayamos encontrando a raíz del paso anterior]
- Documentación [Dejar explicado con claridad para futuros programadores y usuarios, qué hace, cómo funciona, y cómo se usa el programa creado] (Esto es por ejemplo el editorial de topcoder/codeforces, un buen tutorial, manual de usuario, gráficos y diagramas explicativos, etc).

Notar que no necesariamente se siguen estrictamente en ese orden (en proyectos grandes, nunca se hacen en orden sino que se hace "todo en paralelo todo el tiempo"), y en pequeños programas (como todos los que haremos, y todos los que se hacen en competencias de programación) es común que no se noten o incluso que falten algunos pasos, pero "la idea del proceso es esa, y siempre está presente".

Mientras más tiempo se invierta en análisis y diseño del algoritmo, en general menos tiempo se invertirá en los demás pasos: conviene pensar y entender muy bien qué queremos hacer, antes de mandarse a escribir código en la computadora.

Un algoritmo es un método para resolver un problema mediante una serie de pasos precisos, definidos y finitos.

1. Preciso : Está bien claro qué pasos hay que hacer, y en qué orden. No hay ambigüedad.
2. Definido : Llega al mismo resultado cada vez que se lo ejecuta [Si no se cumple esta propiedad, se habla de algoritmos no deterministas]
3. Finito : Tiene fin, el método termina siempre luego de realizar suficientes pasos, y no puede continuar "eternamente".

O sea que un algoritmo es, en definitiva, una receta super recontra precisa . Como para que la entienda un robot super literal sin inteligencia. O sea, una computadora .

Ejercicio: Escritura de algoritmo

Escribir un algoritmo para que Agus pueda calcular la suma de los números desde 1 hasta N (un número cualquiera) usando la calculadora. Es decir, con N=3, al seguir el algoritmo Agus debería obtener 6 (Porque 1 + 2 + 3 = 6). Sin embargo con N = 5 se debería obtener 15 al finalizar el proceso.

Conocimientos de Agus:

- Agus sabe contar hasta N (conoce el siguiente de un número, y sabe cuándo llega a N)
- Agus sabe escribir un número en la calculadora.

Recuerde que para que sea un algoritmo, las órdenes deben ser bien precisas y exactas, y no pueden quedar libradas al "sentido común" de Agus (ya que Agus opera como un robot mecánico que sigue todo lo que le pidan literalmente).

Posibles soluciones (Escritura de algoritmo)

El siguiente es un posible algoritmo para darle a Agus (los pasos deben seguirse en orden uno a continuación de otro, a menos que se indique explícitamente lo contrario):

1. Encender la calculadora con el botón “ON”.
2. Comenzar con “1” como el número actual, e introducirlo en la calculadora.
3. Si el número actual ya es N, saltar al paso 8.
4. Avanzar el número actual al siguiente.
5. Presionar el botón “+” de la calculadora.
6. Introducir el número actual en la calculadora.
7. Volver al paso 3.
8. Presionar el botón “=” de la calculadora.
9. En este paso, tenemos en el visor de la calculadora el resultado deseado.
10. Apagar la calculadora con el botón “OFF”.

Otra manera muy similar de escribir este algoritmo es la siguiente:

- Encender la calculadora con el botón “ON”.
- Comenzar con “1” como el número actual, e introducirlo en la calculadora.
- Repetir lo siguiente mientras que el número actual no sea N:
    - Avanzar el número actual al siguiente.
    - Presionar el botón “+” de la calculadora.
    - Introducir el número actual en la calculadora.
- Presionar el botón “=” de la calculadora.
- En este paso, tenemos en el visor de la calculadora el resultado deseado.
- Apagar la calculadora con el botón “OFF”.

Si bien ambas descripciones son en este caso completamente equivalentes, esperamos que coincida en que la segunda es más clara. Preferiremos siempre descripciones que se basen en “procesos” y “repeticiones”, antes que en “números de línea” y “saltos” entre ellos, ya que resultan más fáciles de entender y modificar.

Notar que las instrucciones son bien precisas y específicas, y no dan lugar a cuestionamientos. ¿Son también precisas las instrucciones que propuso antes de ver estos ejemplos?

Notar también que estas instrucciones, al ser tan precisas, pueden ser inadecuadas para modelos de calculadora levemente distintos. Esto es normal en programación: una computadora procesa las instrucciones dadas literalmente y no tiene la “capacidad de adaptación” o “sentido común” de un ser humano a la hora de seguir lo indicado en un programa. Por eso ante cambios relativamente menores (como usar otro modelo de calculadora), es común tener que modificar el programa igualmente, porque la computadora no logra adaptarse por sí sola (a menos por supuesto, que mediante un programa... ¡Ya se le haya indicado con total precisión cómo debe hacer para adaptarse por sí sola ante un cambio! Eso es extremadamente difícil y es la tarea que llevan a cabo los investigadores en inteligencia artificial).

Análisis del problema

Cuando vayamos a resolver un problema en computadora lo primero que tenemos que lograr tener en claro es:

- ¿Qué entradas se requieren (tipo y cantidad)?
- ¿Cuál es la salida deseada (tipo y cantidad)?
- ¿Qué método produce la salida deseada?

Por ejemplo, si queremos hacer un programa que sume dos números enteros positivos, la entrada serán dos números enteros positivos A y B. La salida deseada será un único número entero positivo, X, que será el resultado X=A+B y por lo tanto se obtiene realizando una operación de suma entre A y B.

Diseño del algoritmo

- Enfoque “top-down ”: Sabemos (iremos aprendiendo cómo durante el curso) resolver problemas pequeños y sencillos. Para encarar problemas grandes entonces la idea será irlos descomponiendo en tareas más simples y luego juntar todos los pasos adecuadamente.
- Ejemplo: Calcular pago mensual neto de un trabajador conociendo horas trabajadas, tarifa horaria y tasa de impuestos. Lo descomponemos en 3:
    - Leer datos: Horas, tarifa, tasa

    - Calcular el sueldo neto usando los datos
    - Mostrar el sueldo neto por la salida en el formato deseado
- Si aprendemos entonces a “leer datos”, a “hacer cuentitas”, y a “escribir a la salida”, podremos juntar todo eso en un solo programa para resolver el problema original.

Hello World

Primer programa de Ejemplo

Una vez que tenemos listo todo el ambiente (Geany bien instalado y compilador g++ listo para que Geany lo use), ¡Estamos en condiciones de crear programas con Geany!

A modo de ejemplo, veamos un primer programa de ejemplo que llamaremos "HolaMundo". En Geany, crear un archivo nuevo llamado "HolaMundo.cpp" (Es importante que todos nuestros archivos de C++ utilicen la extensión ".cpp", que es la extensión de C++ más común. Si no utilizamos una extensión .cpp, Geany no entenderá que el archivo en cuestión está escrito en el lenguaje C++, y no lo compilará correctamente. Esto es porque Geany permite utilizar también otros lenguajes).

Una vez creado este archivo, copiar (o tipear) el siguiente texto como contenido del mismo. Este texto es un programa escrito en el lenguaje de programación C++. Más adelante estudiaremos bien qué significa cada parte del programa.

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hola mundo!" << endl;
    return 0;
}
```

Una vez escrito y guardado (Archivo -> Guardar, o Ctrl + S) un programa en Geany, deberemos compilarlo antes de poderlo ejecutar. Para compilar, en Geany utilizamos la tecla F9 (O bien el menú Construir -> Construir).

La ventana de Geany está dividida en dos partes: En la superior podemos escribir el código fuente (texto del programa), y en la inferior Geany nos muestra el resultado de la compilación , que puede ser exitoso o tener errores. Por ejemplo, si compilamos el programa anterior, Geany debería mostrar un resultado exitoso similar al siguiente:

```
g++  -Wshadow -Wall  -Wextra -D_GLIBCXX_DEBUG -o "HolaMundo" "HolaMundo.cpp" (in directory: /home/santo/Documentos/C++)
Compilation finished successfully.
```

La parte de "Compilation finished successfully" nos indica que el resultado de la compilación fue exitosa. Cuando acabamos de compilar exitosamente un programa en Geany, podemos ejecutarlo con la tecla F5 (O bien con el menú Construir -> Ejecutar).

Al ejecutar el programa, la computadora se encarga de realizar todas las instrucciones que el programa le da, una tras otra, y veremos en la pantalla el resultado de la ejecución (si lo hay). En este ejemplo, la única instrucción que da el programa es la de imprimir en la pantalla una línea con el mensaje "Hola mundo!", y por lo tanto eso será lo que veremos al ejecutar con F5 este programa. Por ejemplo, podríamos ver una pantalla similar a la siguiente:

```
Hola mundo!


------------------
(program exited with code: 0)
Press return to continue
```

Ejemplo de programa con un error de compilación

Que los programas nos compilen sin errores como en el caso anterior es algo muy raro. Casi siempre, tenemos pequeños errores en el código fuente, de los cuales el compilador nos irá avisando, y que vamos corrigiendo a medida que los vemos.

Por ejemplo, imaginemos que hubiéramos tipeado el siguiente programa incorrecto:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hola mundo!" << endl
    return 0;
}
```

Este es igual al HolaMundo.cpp de la sección anterior, pero nos hemos olvidado un ";". En este caso, si compilamos este programa con F9, el mensaje que muestra Geany en la parte inferior es:

```
g++  -Wshadow -Wall  -Wextra -D_GLIBCXX_DEBUG -o "HolaMundo" "HolaMundo.cpp" (in directory: /home/santo/Documentos/C++)
HolaMundo.cpp: In function ‘int main()’:
HolaMundo.cpp:8:5: error: expected ‘;’ before ‘return’
     return 0;
     ^
Compilation failed.
```

La parte de `HolaMundo.cpp:8:5` nos indica que se detectó algún error en el archivo HolaMundo.cpp, en la línea 8, en la columna 5. A continuación se describe el mensaje de error, que en este caso es `error: expected ‘;’ before ‘return’`. Esto significa que el compilador estaba esperando un ";" de terminación de la instrucción, pero sin embargo se encontró con el "return" de la línea siguiente, y como la aparición repentina de ese return le resulta inválida, nos informa de este error.

Si se compila este programa, Geany subrayará de rojo automáticamente la línea donde el compilador informa del error, para facilitar encontrar el mismo y así corregir más fácil el programa. Además, si hacemos clic en el error mismo en la parte inferior de los mensajes (Es decir, en la línea en rojo que dice `HolaMundo.cpp:8:5: error: expected ‘;’ before ‘return’`), Geany llevará la vista directamente hasta la línea del error y nos la indicará con una flecha amarilla en el lado izquierdo. Esto es especialmente útil en programas largos con por ejemplo 100 líneas, de manera que podamos saltar directamente al error en lugar de buscarlo a mano por todo el archivo.

Algo interesante para notar en este ejemplo es que el compilador nos dice el lugar del archivo donde descubrió que hay un error, pero el error en sí puede estar en otro lugar. En nuestro caso, el compilador recién descubre del error (la falta del ";") en la línea 8, pero el ";" en sí mismo debería estar en la línea anterior, con lo cual podríamos decir que el error está en la línea 7 en lugar de en la línea 8. Tener esto en mente es útil a la hora de corregir los programas para que compilen correctamente.

Descuidos comunes

Hay que tener cuidado de los siguientes descuidos muy comunes:

- Olvidarse de usar F9 para compilar el programa, y en cambio usar directamente F5 para ejecutarlo. Si hacemos esto, Geany ejecutará la última versión que hayamos compilado, que puede no ser igual a la que tenemos a la vista y así generarnos confusión. Si nunca hemos compilado nada, se verá un mensaje de error al ejecutar.
- Usar F9 para compilar el programa, y luego usar F5 para ejecutar el programa, pero el programa no compiló exitosamente. Si hacemos eso, Geany ejecutará la última versión que compiló exitosamente. Como esa no es la que tenemos escrita en el archivo (que es la que no ha compilado exitosamente), nuevamente esto puede causarnos confusión.

Análisis del programa Hola Mundo

Entendamos ahora el contenido del programa, línea por línea. Para algunas de las líneas, no necesitaremos entender bien a fondo los motivos por los cuáles son necesarias, y en realidad lo que haremos en la práctica será copiarlas siempre igual en todos nuestros programas. De cualquier manera, comentamos a continuación todo el programa, para que podamos hacernos al menos una idea de qué hace cada cosa.

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hola mundo!" << endl;
    return 0;
}
```

La primera línea, `#include <iostream>`, es una directiva `#include`. Estas directivas se utilizan para indicarle al compilador que vamos a querer utilizar partes de una biblioteca estándar existente. Se le llama biblioteca a un conjunto de elementos que ya están programados, y que podemos entonces utilizar en nuestro programa ahorrándonos así la necesidad de programarlos nosotros. En este caso particular, la línea indica que queremos utilizar la biblioteca iostream. Al hacerlo, a continuación tendremos disponibles todos los elementos de la biblioteca iostream. En nuestro programa en particular, los elementos de esta biblioteca que utilizaremos más adelante serán "cout" y "endl". Se puede comprobar que si quitamos esta línea del programa, el compilador generará un error en la línea que contiene a "cout" y "endl", ya que al no haber incluido la biblioteca no podemos usar estos elementos.

La línea `using namespace std;` la utilizaremos textualmente en todos nuestros programas. Los elementos de las bibliotecas estándar de C++ están todos contenidos en lo que se llama un namespace, en concreto en el namespace std (Del inglés, "standard"). Si no incluyéramos esta línea, tendríamos que poner `std::` delante de cualquier elemento de las bibliotecas estándar, lo cual nos resulta incómodo. Con esta línea indicamos que queremos usar todas las cosas de std "directamente", sin ponerles `std::` delante.

Se puede comprobar por ejemplo, que si eliminamos esta línea, el compilador genera un error en la línea que contiene a cout y endl. Sin embargo, si cambiamos la línea que los contiene por `std::cout << "Hola mundo!" << std::endl;` (colocando el `std::` delante de cout y endl), el programa pasa a compilar correctamente. Es por esto que utilizamos siempre la línea `using namespace std;`, para así evitarnos muchos `std::` en el programa.

Luego viene la función main() . Más adelante aprenderemos lo que es una función, y entonces entenderemos que main es una función como cualquier otra. Por ahora, lo importante es saber que con `int main()` comienza la parte del programa donde colocaremos todas las instrucciones que la computadora ejecutará secuencialmente, en el orden que indiquemos. Estas instrucciones deben siempre ir rodeadas entre llaves `{` y `}`. En general, las llaves `{` y `}` son utilizadas siempre en C++ para encerrar bloques de instrucciones, que se ejecutarán en secuencia , una atrás de otra, en el orden indicado.

Por lo tanto, en todos nuestros programas tendremos una parte de la siguiente forma:

```
int main()
{
    //...INSTRUCCIONES...
}
```

Donde en la parte que hemos anotado con `...INSTRUCCIONES...`, colocaremos la lista de instrucciones que indicarán a la computadora qué debe hacer, en orden.En nuestro caso, esta lista tiene solamente dos instrucciones.

Notar que todas las instrucciones terminan en ";" : En C++, el símbolo ";" es el marcador de fin de instrucción . Es por eso que si lo omitimos, el compilador piensa que la instrucción continúa en la línea siguiente. Esto está de hecho permitido: es posible partir una instrucción entre varias líneas, y no es obligatorio colocar una sola instrucción por línea, aunque esto es lo más recomendado.

La primera instrucción de nuestro programa es `cout << "Hola mundo!" << endl;`. Esta instrucción le ordena a la computadora que muestre el mensaje `"Hola mundo!"` por la pantalla. `"Hola mundo!"` es lo que llamamos una cadena de texto, es decir, una porción de texto normal para escribir en pantalla, que siempre se escriben en C++ encerrados entre comillas `"`. `cout` es un elemento de la biblioteca iostream, al cual podemos enviar cadenas de texto para que las muestre por la pantalla. La forma de enviar cadenas a cout es con el símbolo `<<`, que visualmente se asemeja a una especie de flecha que apunta hacia cout.

Por eso `cout << "Hola mundo!"` le envía el mensaje "Hola mundo!" a cout para que lo imprima. Estos envíos con `<<` se pueden encadenar en una misma instrucción, y por ejemplo podríamos haber hecho con idénticos resultados `cout << "Hola" << " " << "mundo!" << endl;`. Notar en este caso la cadena `" "`, que contiene únicamente un espacio en blanco.

Finalmente, `endl` es un elemento especial que se le puede enviar a cout para indicar que se salte a la línea siguiente. En este caso, como escribimos una única línea en la pantalla, quizás no se note la importancia de saltar a la línea siguiente. Sugerimos probar para esto ejecutar dos copias de esta instrucción, y probar las distintas combinaciones de poner y sacar el endl para observar mejor sus efectos.

La última instrucción del programa es `return 0`. En todos nuestros programas colocaremos esta instrucción al final del main. Lo que hace esta instrucción es indicar que se debe terminar la función main (que terminará todo el programa) con el código de finalización 0 (que es un código que significa que no hubo errores y todo salió bien).

Jugando con el Hola Mundo

En la sección anterior, vimos un ejemplo de programa que llamamos `HolaMundo.cpp`. Reproducimos este programa a continuación:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hola mundo!" << endl;
    return 0;
}
```

En esta sección, experimentaremos un poco con posibles cambios al mismo, para ver el comportamiento que resulta de los cambios.

Comentarios

Un cambio muy sencillo que podemos hacer a cualquier programa es el de agregar comentarios .

Un comentario es una aclaración en forma de texto, para ser leída por un humano, y que el compilador ignora por completo . De esta forma, el comportamiento de nuestros programas no cambia en lo más mínimo por agregar y sacar comentarios. Los comentarios son simplemente una forma de ordenarse a la hora de programar , o bien de clarificar cosas que podrían no entenderse cuando otra persona lea el código . Es decir, se utilizan para aumentar la claridad del programa y hacerlo más fácil de leer.

La forma más común de comentarios son los comentarios de una línea, y se obtienen utilizando `. Cuando se coloca` en alguna línea, todo lo que sigue en esa misma línea es completamente ignorado por el compilador, que lo considerará un comentario.

Otra forma de realizar comentarios menos común es a través de los comentarios multilínea , que comienzan con `/*` y finalizan con `*/`.

Veamos un ejemplo del `HolaMundo.cpp` con comentarios agregados:

```
#include <iostream>

using namespace std; // Esto es solo por comodidad, nos sirve para no poner std:: todo el tiempo.

int main()
{
    /* Esto es un comentario de multiples lineas.
     * Notar que por belleza estetica, estamos comenzando todas las lineas con
     * un "*", para que parezca un 'margen' o 'borde' desde el cual escribir.
     * Como todo esto esta encerrado entre los extremos de un comentario multilinea,
     * da igual cualquier cosa que escribamos, y el compilador lo ignorara por completo.

int main ()
{
     return 5;
}

     Notar que lo anterior, aunque parezca codigo fuente c++, no se ejecuta nunca: Seguimos
```

```
     estando adentro del comentario.

     */
    cout << "Hola mundo!" << endl; // Esta linea si la tendra en cuenta el compilador, salvo por este comentario
    return 0;
}
```

Si compilamos (F9) y ejecutamos (F5) este programa modificado, veremos que hace lo mismo que el anterior: Los comentarios no afectan en nada.

Escribir muchas líneas

En `HolaMundo.cpp`, damos a la computadora la instrucción de imprimir una línea con el texto `Hola mundo!`. Pero podríamos perfectamente escribir una secuencia de órdenes, escribiendo muchas cosas:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hola mundo!" << endl;
    cout << "Estamos comenzando a realizar programas propios. WOW!" << endl;
    cout << endl; // Dejamos una linea en blanco: Pasamos a la siguiente directamente
    cout << "Esto va a salir en ";
    cout << "una";
    cout << " sola linea " << "sin" << " importar el";
    cout << " hecho de que le vamos tirando el texto en partes." << endl;
    cout << "Solamente cuando usamos endl para cambiar de linea," << endl;
    cout << "cout pasa a la linea siguiente" << endl;
    return 0;
}
```

Notar que en este caso, terminamos todas las instrucciones con el correspondiente `;`. Como las instrucciones son ejecutadas desde arriba hacia abajo, y de izquierda a derecha, (en el mismo orden en que se leen los textos en español), cuando compilemos y ejecutemos el programa anterior obtendremos lo siguiente como resultado:

```
Hola mundo!
Estamos comenzando a realizar programas propios. WOW!

Esto va a salir en una sola linea sin importar el hecho de que le vamos tirando el texto en partes.
Solamente cuando usamos endl para cambiar de linea,
cout pasa a la linea siguiente
```

Es posible que al ejecutar esto con Geany, en la ventana parezca que la línea larga (que comienza con "Esto va a salir") se vea partida en dos líneas. Pero esto es por una cuestión de que la ventana que utiliza Geany para la visualización no es muy ancha: el texto que escribe el programa tiene todo eso en una sola línea. Esto lo podemos verificar si seleccionamos el texto que ha producido el programa, hacemos clic derecho -> copiar, y luego lo pegamos en un editor de texto separado (como el mismo Geany): Veremos que allí el programa ha producido un texto que efectivamente, solamente cambia de línea al utilizar los endl.

Programa que saluda

Hasta ahora, nunca hemos hecho un programa interactivo: Nuestros programas solamente escriben un texto fijo a la pantalla sin importar lo que hagamos. Veamos ahora un programa que saluda:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    cout << "Cual es tu nombre?" << endl;
    string nombre;
    cin >> nombre;
    cout << "Buenos dias " << nombre << "!" << endl;
```

```
    return 0;
}
```

Este es un ejemplo muy sencillo, pero aquí hemos incorporado varios elementos nuevos que explicaremos brevemente (serán explicados mejor en la sección siguiente).

En primer lugar, hemos añadido la línea `#include <string>`. Esto es necesario porque queremos utilizar `string`, que es parte de la biblioteca de igual nombre. `string` se utiliza para que los programas puedan manipular textos.

La línea `string nombre;` le avisa a la computadora que usaremos una variable , a la cual hemos llamado nombre (puesto que la usaremos para el nombre del usuario que vamos a saludar). Las variables son un concepto fundamental de la programación imperativa: Lo veremos mucho mejor en la sección siguiente, pero por ahora, basta con saber que la intención de esta línea es reservar espacio para un texto que vamos a recibir del usuario (en concreto: su nombre).

La línea `cin >> nombre;` es novedosa: Utiliza el elemento nuevo `cin`, que es parte junto con `cout` y `endl` de la biblioteca `iostream`. Así como `cout` es un elemento al cual podemos enviar datos que queremos que se escriban en la pantalla, `cin` trabaja como entrada , y de él podemos recibir datos que se ingresan al programa (normalmente, a través del teclado, pero es posible cambiarlo y que por ejemplo `cin` obtenga los datos desde un archivo). La forma de escribir esto es igual que para `cout`, pero con "las flechas" invertidas, pues ahora recibimos los datos desde `cin`. Además, en lugar de indicar cuál es el dato que enviamos, indicamos dónde guardarlo : Eso es lo que hacemos al indicar la variable `nombre` usada anteriormente.

Finalmente, la línea `cout << "Buenos dias " << nombre << "!" << endl;` imprime por pantalla el saludo final. Notar que como estamos enviando `nombre` a `cout`, que es algo que recibimos previamente a través de `cin`, la cadena que se imprime no está fija, sino que depende de lo que haya ingresado el usuario. En otras palabras, este programa presenta un saludo interactivo según el nombre del usuario, como podemos verificar si lo ejecutamos.

Se sugiere fuertemente experimentar haciendo cambios a este programa. Veremos más cuidadosamente los principios en funcionamiento detrás de instrucciones como esta, en la sección siguiente.

Cuentas sencillas

Es sabido que la computadora tiene una gran capacidad de cálculo . Podemos ponerla a hacer cuentas, y las resolverá de manera prácticamente instantánea. Veamos un ejemplo que suma dos números recibidos por pantalla:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Ingrese los numeros " << endl;
    int a,b;
    cin >> a >> b;
    cout << "La suma de los dos numeros es " << a + b << endl;
    return 0;
}
```

Este ejemplo es similar al anterior pero se usa `int` en lugar de `string`: `int` se utiliza para indicar que trabajaremos con números enteros . Veremos más en detalle qué significa esto en la sección siguiente.

Al ejecutar este programa, si ingresamos por ejemplo 23 y 48, podemos obtener una salida como la siguiente:

```
Ingrese los numeros
23 48
La suma de los dos numeros es 71
```

Observamos que este programa es capaz de sumar los números que le indicamos, sin importar cuáles sean . Es decir, le hemos enseñado a la computadora, a través de nuestro programa, cómo interactuar para escribir la suma de dos números dados. ¿Qué ocurrirá si usamos `a * b` en lugar de 'a + b'? ¿Y si usamos a - b? ¿Y si usamos a / b? ¡Experimente!

¿Se anima a hacer un programa que lea dos números, como el anterior, pero luego los escriba en el orden inverso al que fueron ingresados? Lo dejamos como desafío. Por ejemplo si al programa se ingresa `12 34`, se espera que escriba a continuación `34 12`.

Así como podemos utilizar `+`, `-`, `*` y `/` para realizar operaciones aritméticas fundamentales, es posible utilizar los paréntesis `(` y `)` para poder escribir cosas como (1+2) * 3 (que tiene por resultado 9) en lugar de 1 + 2 * 3 (que tiene como resultado 7, por las reglas de separar en términos). Con eso podríamos escribir cualquier operación aritmética y usar a la computadora como calculadora en programas sencillos.

Por ejemplo, el siguiente programa:

```
#include <iostream>

using namespace std;

int main()
{
    cout << 23+12 << endl;
    cout << ((23*11-32)*5 + 70) / 6 << endl;
    cout << (((5*5 - 4*4 - 3*3))) << endl;
    cout << 5*5 - 4*4 - 3*3 << endl;
    cout << (5*5 - 4*4 - 3*3) * (5*5 - 4*4 - 3*3) << endl;
    return 0;
}
```

Imprimiría por pantalla lo siguiente:

```
35
195
0
0
0
```

Notar que en C++ es válido utilizar "paréntesis redundantes", como los de la tercera cuenta. Lo que no está permitido es tener paréntesis inválidos, o que no se correspondan entre sí. Por ejemplo, una expresión como `(2 + 4(` generará un error de compilación, pues el segundo paréntesis que abre debería estar cerrando. Similarmente, algo como `(2 + 4` sería una expresión incompleta, pues falta cerrar el paréntesis, y esto también producirá un error de compilación.

Variables, valores, expresiones y tipos

Valores

En programación, un valor es un dato en concreto. La computadora manipula valores permanentemente, transformándolo unos en otros mientras realiza sus cálculos y procesamientos.

Así por ejemplo, el número `32` es un valor. El `48` es otro valor. Cuando ingresamos datos a la computadora a través del teclado, lo que estamos ingresando son valores.

No solamente los números en sí son valores: como hemos dicho, cualquier dato que pueda ser manipulado por la computadora constituye un valor en concreto. Si está leyendo esto en una computadora, entonces quedará claro que las computadoras manipulan textos. Por lo tanto, un texto como `"Patoruzito"` es también un valor, así como lo son los textos `"Enero"`, `"Febrero"`, y `"Esta oracion autorreferencial es un dato."`.

Podemos observar que no todos los datos son "similares": `12`, `32`, `48` y `50` son todos números, mientras que en cambio `"Lunes"`, `"Martes"` y `"Miércoles"` son todos textos. Esto nos lleva a la siguiente noción fundamental, que es la de tipo.

Tipos

Un tipo de dato (o más brevemente, un tipo) es un conjunto de valores que son "similares", en cuanto a que pueden ser tratados de la misma manera por la computadora. En los ejemplos anteriores ya vimos dos tipos de datos bien distintos: Los números enteros, como `42` y `10`, y las cadenas de texto, como `"Domingo"` y `"Viernes"`. Cada tipo tiene asociadas operaciones potencialmente diferentes: los números los podemos restar, y por ejemplo `42 - 10` da por resultado `32`. Sin embargo... ¿Qué significaría una operación como `"Domingo" - "Viernes"`? ¿Qué significa en general restar textos?

Vemos entonces que las operaciones válidas para hacer en un programa, dependen del tipo de los valores que estemos manipulando.

En C++, los tipos tienen nombres particulares, y estos nombres se usan en los programas. Ya estuvimos usando en programas anteriores los tipos de datos más comunes para representar números y cadenas de texto:

- `int`: El tipo int es uno de los más comunes. Se utiliza para representar números enteros. Así, los textos como `"Seminario"` no pueden ser valores de tipo int. El tipo int representa números enteros, ya sean positivos, negativos, o cero, pero no se puede usar para números arbitrariamente grandes: Únicamente permite números desde -2147483648 (que es 2 a la 31 negativo) hasta 2147483647 (que es 2 a la 31 menos 1), inclusive. Si las cuentas intermedias que realicemos en nuestro programa generan números por fuera de este rango, se producirán resultados erróneos. A eso lo llamamos overflow, y es un peligro con el que siempre se tiene que tener cuidado cuando los números con los que trabajamos puedan volverse grandes.
- `string`: El tipo string se utiliza para representar cadenas de texto. Como hemos explicado antes, no podremos entonces restar dos valores de tipo string. Para utilizar `string` hay que agregar la línea `#include <string>` al comienzo del programa.
- `char`: Este es un tipo de datos que nunca usamos antes. Un valor de tipo `char` representa un caracter, es decir, una letra particular de un texto. Así como anotamos los valores de tipo string entre comillas dobles `"`, anotamos los valores de tipo char entre comillas simples. Por ejemplo, `'a'`, `'b'`, `'A'`, `')'`, `'+'`, `' '` son todos valores distintos de tipo char. Es decir, cada letra en particular de un valor de tipo string, será un valor de tipo char.

Expresiones

Las expresiones son porciones de código fuente en las cuales se aplican operaciones a valores básicos, para denotar así valores más complejos que son el resultado de las operaciones. Por ejemplo `(1+2)*4` es una expresión, que denota el valor 12 (ya que indica que se deben sumar los valores 1 y 2, lo cual daría 3, y luego eso multiplicarlo por 4). Lo que en matemática es "una cuenta", en programación se dice que es una expresión.

Notar que como en programación hay muchos más valores que solamente números (por ejemplo, nosotros vimos ya que hay cadenas de texto), una expresión en programación es más general que "una cuentita" en el sentido matemático. En particular, según el tipo que tengan los valores con los que estemos trabajando, serán válidas expresiones que usen algunas operaciones y no otras.

En cualquier expresión, siempre se pueden utilizar paréntesis para indicar el orden en que se realizan las operaciones.

Expresiones aritméticas

Las expresiones aritméticas son las más sencillas, y son las que se obtienen directamente utilizando los operadores aritméticos básicos: suma, resta, producto y división (+,-,* y / respectivamente), además de los paréntesis. Estas son las que se corresponden directamente con cuentas matemáticas.

Así, `(1 + 5) * 3` es una expresión aritmética. Otra más larga es `( ( (1+5)*2)/3)*8`. Si `x` es un `int`, entonces `x+1` es una expresión que denota al entero que le sigue a `x`, y `2*x` es una expresión que indica el doble de `x`.

Además de estas 4 operaciones aritméticas básicas, es común también utilizar la operación `%`, que se llama "módulo", y sirve para obtener el resto de la división. De esta forma, la expresión `9 / 2` da por resultado `4` (ya que cuando trabajamos con enteros, como trabajaremos casi siempre, la operación da el resultado de hacer la división entera), y la operación `9 % 2` da por resultado `1`, pues al dividir 9 por 2 se obtiene un cociente de 4, y un resto de 1. Similarmente, `14 % 5` da por resultado 4, mientras que `14 / 5` da por resultado 2.

Expresiones con strings

El tipo `string` de C++ es un tipo que soporta varias operaciones útiles. Mencionamos a continuación 3 de las más comunes.

Si `s` es un string, podemos consultar la longitud de `s` con el operador especial `.size()`. Por ejemplo, si `s` fuera el string `"Limon"`, que tiene 5 letras, la expresión `s.size()` tendría por resultado 5.

Otro operador interesante es la llamada concatenación , que en C++ se indica con el operador `+` (el mismo que la suma normal, pero al operar con strings tiene otro significado), y no es otra cosa que "pegar" las dos cadenas, una a continuación de la otra, en el orden indicado . Así, si `a` es una cadena "Abra" y `b` es una cadena `"Cadabra`, entonces la expresión `a+b` denotará el string `"AbraCadabra"`, mientras que `b+a` denotará el `CadabraAbra`.

Un detalle de C++ es que cuando escribimos directamente una cadena entre comillas en el código, como por ejemplo `"Abra"`, el tipo de ese valor no es exactamente `string`, sino que es otro tipo más complicado, del cual no hablaremos pues escapa a este curso introductorio. Este tipo no funciona con los dos operadores mencionados, de manera tal que `"Abra".size()` y `"Abra" + "Cadabra"` no funcionarán en C++. Esto se puede resolver encerrando a las cadenas entre comillas con `string(...)` (lo cual le indicará al compilador, que queremos que esos valores sean strings que se pueden sumar y operar). En los ejemplos anteriores, `string("Abra").size()` y `string("Abra") + string("Cadabra")` funcionarán sin problemas.

El último operador que nos interesa sobre strings es el de acceso a un caracter . Si tenemos un string `s`, podemos obtener su primera letra (que será de tipo char) haciendo `s[0]`. La segunda será `s[1]`, la tercera `s[2]`, y así siguiendo, donde notar que se comienza a contar desde cero. Así por ejemplo, la expresión `"Cadabra"[3]` denota un char con la cuarta letra de `"Cadabra"`, es decir, `'a'` . Similarmente `"Cadabra"[2]` tendría el valor `'d'` . Este operador funciona sin problemas sin necesidad de usar `string(...)` sobre las constantes.

Notar que las expresiones pueden ser combinadas libremente, de manera que `"Pepe"[1+2]` denota un caracter con una e, y por ejemplo `(string("Juan") + string("Perez") ).size()` denota al int `9`

Expresiones con char

La operación fundamental entre caracteres es la aritmética. Esto puede parecer ilógico en principio, pues no queda claro qué significa por ejemplo sumar una 'a' con una 'b'.

Las letras, y todos los demás caracteres en la computadora, están ordenados con cierto criterio. En el caso de C++ en las computadoras usuales, para los valores que más comúnmente utilizamos, el orden de los caracteres está dado por el llamado código ASCII [https://es.wikipedia.org/wiki/ASCII].

De esta forma, si `c` denota un cierto caracter, `c+1` denota al caracter siguiente en este ordenamiento. Similarmente, `c-1` denota al anterior. Una propiedad útil de este ordenamiento es que las letras mayúsculas están todas juntas y en el orden del alfabeto inglés (sin la eñe). Similarmente con las letras minúsculas. Esto quiere decir que por ejemplo, `'a' + 3` es una expresión que tendrá por resultado `'d'` , y `'Q' - 2` es una expresión que denota al caracter `'O'` . Notar que mayúsculas y minúsculas son caracteres distintos.

Similarmente, podemos obtener la distancia en el abecedario entre dos letras haciendo su resta: `'e' - 'a'` dará por resultado un int, en este caso 4 (pues desde la a hay que avanzar 4 letras para llegar a la e). Notar que si mezclamos mayúsculas con minúsculas, en estos casos tendremos resultados incorrectos, pues en la tabla ASCII mencionada las mayúsculas y minúsculas tienen códigos diferentes (no son el mismo caracter).

Otra propiedad útil del ordenamiento ASCII es que los dígitos del `'0'` al `'9'` están en orden y todos juntos en el ordenamiento; y por lo tanto, si sabemos que `x` es un `char` que corresponde a un dígito, con la expresión `x - '0'` podemos obtener el valor numérico del caracter, directamente como un número entero.

Variables

Hemos visto hasta ahora expresiones, que operan con valores directamente escritos en el programa. Por ejemplo, una expresión como `1 + 2` opera directamente con el `1` y el `2` allí escritos, obteniendo `3`. Este tipo de expresiones no permiten al programa "adaptarse" a los datos: Siempre se opera igual.

Para que un programa pueda trabajar con datos arbitrarios, surge el concepto clave de variable .

Una variable es un espacio de memoria de la computadora, donde se almacena un valor . Podemos pensarlo como una caja , en la que guardamos el dato que nos interesa. Una variable tiene un nombre , que nos permite referirnos a ella. Es decir, la caja donde se guarda el dato tiene un nombre, con el cual podemos hablar tanto de la caja, como del valor que se guarda en ella.

Una variable en C++ puede guardar únicamente datos de un cierto tipo , que es el tipo con el cual se declara la variable. Esto es lo que hacemos en líneas como `int x;` en los programas que ya vimos: Esa línea declara que utilizaremos una variable de nombre "x", y en la cual podremos guardar valores de tipo int . Cuando declaramos `string nombre;`, estamos indicando a la computadora que utilizaremos una caja, a la cual nos referiremos como "nombre", y en la cual guardaremos datos de tipo `string`.

Ahora podemos entender el efecto de líneas como `cin >> x`: esta instrucción le ordena a la computadora que lea un dato ingresado por el usuario, y lo guarde en la variable x. Una propiedad central de las variables es que son una caja que guarda un solo valor , y ese valor guardado puede cambiar con el tiempo . Por lo tanto, cuando se almacena un nuevo valor en la caja, se pierde todo lo que hubiera allí antes .

Así por ejemplo, si tenemos el siguiente fragmento de programa:

```
int x;
cin >> x;  // Lectura 1
cout << x + 1 << endl; // Escritura 1
cin >> x;  // Lectura 2
cout << x + 1 << endl; // Escritura 2
```

Supongamos para el siguiente ejemplo, que el usuario va a introducir los números 7 y 23.

Al ejecutar la "Lectura 1", la computadora leerá un número ingresado por el usuario, que guardará en la caja `x`. Cualquier dato que hubiera antes en x se pierde , y quedará a partir de ahora guardado en `x` el valor 7 que introduce el usuario. Luego, la computadora ejecuta la Escritura 1, en la cual se usa la expresión `x + 1`: Como en la caja x hay guardado un 7 , esta expresión dará por resultado 7 + 1 = 8, y por lo tanto se mostrará por pantalla el valor 8.

Luego de esto, se ejecuta la Lectura 2, y entonces se almacena en `x` lo que el usuario ingrese. Como ingresa un valor de 23, en `x` queda guardado el valor 23. El viejo valor de x (que era 7) se pierde al almacenar en `x` un valor nuevo. Como consecuencia de

esto, cuando a continuación de esto se ejecuta la Escritura 2, se imprimirá por pantalla un `24`: Notar que las instrucciones de Escritura 1 y Escritura 2 son idénticas, pero sin embargo producen resultados distintos, porque el valor almacenado en x cambió entre medio de ambas . Esto nos muestra que el orden de ejecución de las distintas operaciones es fundamental.

Una variable puede utilizarse directamente en una expresión, como hemos hecho con `x+1` en el ejemplo. Cuando esto se hace, la computadora utiliza en los cálculos y operaciones el valor que se encuentre almacenado en dicha variable en ese momento . Como este valor puede ir cambiando, como vimos, y depender de lo que haya ingresado el usuario, esto permite a un programa ser flexible y operar con cualquier dato que el usuario ingrese, mientras lo hayamos guardado adecuadamente en alguna variable para poder procesarlo luego.

El operador de asignación

Una operación fundamental en C++ viene dada por el operador de asignación . Con este nombre designamos al `=`: este es un operador que se utiliza para almacenar un valor en una variable . En nuestra analogía de cajas, este operador no significa más que una orden de meter un valor en una caja (Decimos que le asignamos un valor a una variable).

Cuando en C++ escribimos `x = 2 + 5`, estamos ordenando a la computadora que guarde el resultado de la expresión que está en el lado derecho del `=` (en este caso, un 7) en la caja que se indica en el lado izquierdo del `=`. Notar que `=` no es una igualdad en el sentido que se le da en matemática, sino que es una operación : `=` ordena a la computadora que almacene el resultado de una expresión (lado derecho de `=`), en una variable (lado izquierdo de `=`).

En particular, `a = b` y `b = a` significan cosas muy distintas , cosa que no ocurre en matemática: La primera ordena a la computadora guardar en la variable `a`, el valor que ahora se encuentre en la variable `b`. Mientras que la segunda ordena guardar en la variable `b`, el valor que ahora se encuentre en la variable `a`.

Así, si por ejemplo tenemos que `a` guarda un 5, y que `b` guarda un 10, luego de ejecutar `a = b`, ambas variables contendrán un 10, mientras que si se ejecutara `b = a`, ambas quedarían con un valor 5.

Por ejemplo, el siguiente fragmento de programa:

```
int x = 25;
int y = 10;
int z = x+y;
cout << x << " " << y << " " <<  z << endl;
x = 15;
cout << x << " " << y << " " <<  z << endl;
z = x+y;
cout << x << " " << y << " " <<  z << endl;
```

Mostraría por pantalla lo siguiente:

```
25 10 35
15 10 35
15 10 25
```

En este ejemplo se muestra una nueva posibilidad en la sintaxis de C++, que es la de usar el operador de asignación (es decir, el `=`) en la misma línea donde se declara la variable . Esto es extremadamente útil para darle un valor inicial a una variable, guardando en el mismo momento en que creamos la caja, un valor inicial en la misma. Esto se llama inicializar la variable, y es una muy buena costumbre, para asegurarnos de no olvidar guardar un valor en la variable antes de utilizarla: Si utilizamos una variable en la cual nunca hemos guardado un valor, no se sabe con qué valor "comienza": Podría pasar cualquier cosa.

Operaciones de entrada / salida

Ya hemos estado trabajando con operaciones de entrada salida, mediante `cin` y `cout`: Resumimos aquí brevemente su uso con variables.

Cuando queremos recibir datos del usuario, debemos almacenarlos en alguna variable. Para eso utilizamos `cin`, que permite hacer justamente eso. Por ejemplo el siguiente fragmento de código:

```
int x;
cin >> x;
int y,z;
```

```
cin >> y >> z;
cin >> x >> y >> z;
```

Muestra un ejemplo de 3 lecturas de datos realizadas con `cin`: En la primera, se recibe un número entero y se guarda en la variable `x`. En la segunda, se reciben dos enteros más, y se guardan (¡En este orden!) en las variables `y` y `z`. Notar que es posible en una misma instrucción leer varias variables, utilizando para eso el operador `>>` varias veces. Finalmente, en la tercera y última lectura con `cin` se leen tres variables: `x`, `y` y `z`. Notar que en este caso, la tercera lectura borra los valores que se hayan leído en las primeras dos lecturas, ya que los nuevos valores son almacenados en las mismas variables donde se habían leído los valores anteriores, que se pierden al ser reemplazados por los nuevos.

Cuando queremos enviar datos a la salida, utilizamos `cout`. En este caso, `cout` se encarga de escribir en la pantalla el resultado de cualquier expresión que le indiquemos. Además, tenemos el elemento especial `endl`, que sirve para indicar a `cout` que pase a escribir a la siguiente línea. Al elemento `endl` se lo suele denominar el fin de línea

Por ejemplo, el siguiente fragmento de código:

```
cout << "Mensaje1" << endl;
cout << "En la segunda linea " << "podemos enviar un texto en partes" << endl;
cout << "Sin endl, no se salta de linea.";
cout << "Por lo tanto esto se pega a la anterior.";
cout << endl; // Se puede enviar solamente el endl en una instruccion, para forzar el salto de linea.
cout << (1+2+3+4) << " " << (1+3)*10 << endl; // Se pueden enviar expresiones para que se escriban
int x = 42;
cout << x+10 << endl; // Las variables se pueden usar en cualquier expresion
```

Produciría por pantalla el siguiente resultado:

```
Mensaje1
En la segunda linea podemos enviar un texto en partes
Sin endl, no se salta de linea.Por lo tanto esto se pega a la anterior.
10 40
52
```

Const

A veces queremos tener variables que guarden un valor que nunca se va a modificar. Esto se hace por claridad y facilidad del programa: Por ejemplo, imaginemos que estamos haciendo un programa propio que calcula cuánto trabajo le toca hacer a cada uno de los integrantes de un grupo. Supongamos que nuestro grupo tiene 4 personas. Podríamos simplemente poner `4` en el código, cada vez que necesitemos utilizar la cantidad de personas. Sin embargo, esto no deja claro que ese `4` es la cantidad de personas, y no otra cosa que también sea 4 "de casualidad". Similarmente, si un día quisiéramos cambiar el programa para que trabaje con 5 personas... ¡Tendríamos que ir por todo el programa buscando todos los `4`, para ver cuáles se refieren a la cantidad de personas, y esos cambiarlos por un 5!

En lugar de usar el valor directamente, entonces, conviene guardarlo en una variable al comienzo:

```
int CANTIDAD_PERSONAS = 4;
```

Y luego utilizarlo cada vez que necesitemos hablar de la cantidad de personas en el programa: De esta forma, para cambiar la cantidad más adelante hay que cambiar un solo lugar, y el programa queda mucho más claro.

Como esta variable no la vamos a modificar , es muy recomendado declararla con `const`: `const` es una palabra especial que indica al compilador que no vamos a modificar nunca esta variable: Queremos que sea en realidad una constante , y no una variable que realmente puede cambiar de valor. De esta forma si por accidente la modificamos, el compilador nos avisará.

La declaración se hace simplemente agregando `const` al comienzo:

```
const int CANTIDAD_PERSONAS = 4;
```

Es una convención común (de muchas posibles) escribir las constantes en mayúsculas.

Ámbito de una variable

No se puede utilizar una variable en cualquier lugar del programa, sino que cada variable tiene un ámbito , y solo puede utilizarse dentro del mismo.

El ámbito de una variable es el bloque (conjunto de instrucciones delimitadas con llaves `{` y `}`) que contiene la declaración de la variable. Más adelante veremos instrucciones como `if`, `while` y `for` que contienen bloques de instrucciones, y una variable declarada dentro de uno de estos bloques, no puede utilizarse fuera de los mismos.

Estructuras de control selectivas

Hasta ahora, la lista de instrucciones de nuestro programa que se ejecutarán está fija: Esto es, siempre especificamos una lista de instrucciones, y cada una de ellas es ejecutada en orden, de arriba hacia abajo.

Hay veces en las que solamente queremos llevar a cabo una acción determinada a veces , cuando se cumple una cierta condición particular que hace que queramos llevar a cabo la tarea. Veremos en esta lección cómo se puede lograr esto en C++.

La instrucción if

La instrucción `if` sirve para instruir a la computadora a que lleve a cabo un determinado conjunto de instrucciones, únicamente cuando se cumpla una condición específica .

Versión común

La sintaxis (forma de escritura) de la instrucción if es la siguiente:

```
if (condicion)
{
    instruccion1;
    instruccion2;
    //...
    instruccionFinal;
}
```

Notar que los paréntesis alrededor de la condición son obligatorios . De manera similar a lo que ocurre en main, donde escribimos todas las instrucciones que queremos que se ejecuten, las llaves `{` y `}` delimitan un bloque de instrucciones, que únicamente se ejecutarán cuando se cumpla la condición. Si la condición no se cumple, se saltearán todas las instrucciones encerradas entre llaves .

Por ejemplo, el siguiente programa puede utilizarse para leer un número, y escribir en pantalla un mensaje de acuerdo a su signo:

```
#include <iostream>

using namespace std;

int main()
{
    int x;
    cin >> x;
    if (x > 0)
    {
        cout << "El numero " << x << " es positivo." << endl;
    }
    if (x < 0)
    {
        cout << "El numero " << x << " es negativo." << endl;
    }
    return 0;
}
```

Notar algo muy importante, que es que indentamos (colocamos espacios a la izquierda de) las instrucciones del if que encerramos entre llaves: Esto lo hacemos por claridad, para que sea mucho más fácil leer los programas. Cada vez que ponemos un bloque de instrucciones entre llaves, es conveniente que todas las instrucciones contenidas estén más a la derecha visualmente

que las instrucciones que rodean al bloque, para que sea más fácil entender a simple vista dónde comienza y termina el bloque de instrucciones del if. Notar que al compilador no le importan los espacios, y únicamente se basa en las llaves para decidir hasta dónde llega el if. Pero es importante para poder leer y entender más fácil el código, utilizar prolijamente los espacios.

Vemos aquí nuestro primer ejemplo de condición : `x > 0` es una condición que puede colocarse en un if (entre paréntesis), y que se cumple justamente cuando la expresión `x` es mayor que cero. Como `x` es directamente una variable con el valor que leímos, esto se cumplirá cuando el número ingresado por el usuario sea mayor que cero. En dicho caso (¡y solo en dicho caso!) el programa ejecutará las instrucciones entre llaves que siguen al if. En este caso, es una única instrucción que muestra un mensaje indicando que el número es positivo.

Similarmente, luego de verificar la primera condición y (quizás) ejecutar lo indicado entre llaves, se llega en el programa al segundo if: En este se verifica la condición `x < 0`, y por lo tanto las instrucciones entre llaves que siguen a este if únicamente serán ejecutadas cuando el número ingresado sea negativo.

Por ejemplo, si ingresamos el número 4 veremos en pantalla al ejecutar la siguiente interacción:

```
4
El numero 4 es positivo.


------------------
(program exited with code: 0)
Press return to continue
```

En cambio, si ingresamos un valor de -2, al ejecutar veremos en pantalla lo siguiente:

```
-2
El numero -2 es negativo.


------------------
(program exited with code: 0)
Press return to continue
```

Si ingresamos un valor 0, veremos lo siguiente:

```
0


------------------
(program exited with code: 0)
Press return to continue
```

¡Vemos en este caso que el programa no imprime ningún mensaje! ¿Por qué ocurre esto?

La computadora ejecuta las instrucciones indicadas una por una en orden. En particular, cuando llega a cada if, verifica la condición correspondiente para ver si se cumple, y solo ejecuta la instrucción entre llaves (de impresión del mensaje) cuando la condición se cumple. Por lo tanto si este programa no imprime nada, debería ser porque cuando se ingresa el valor 0, ninguna de las dos condiciones se cumple, y no se ejecuta ninguno de los ifs.

En efecto, cero es un número especial, el único entero que no es ni positivo ni negativo, y por lo tanto `0 < 0` y `0 > 0` son ambas falsas . Podemos entonces agregar este caso como un nuevo if en el programa:

```
#include <iostream>

using namespace std;

int main()
{
    int x;
    cin >> x;
    if (x > 0)
    {
        cout << "El numero " << x << " es positivo." << endl;
    }
    if (x < 0)
    {
        cout << "El numero " << x << " es negativo." << endl;
```

```
    }
    if (x == 0)
    {
        cout << "El numero ingresado es cero." << endl;
    }
    return 0;
}
```

Si ejecutamos ahora los mismos ejemplos de antes, obtendremos los mismos resultados en los primeros dos casos: pero al ingresar el valor cero, ahora observaremos lo siguiente:

```
0
El numero ingresado es cero.


------------------
(program exited with code: 0)
Press return to continue
```

Notar que en nuestra tercera condición, hemos utilizado por primera vez el operador ==: Este operador no tiene nada que ver con el =: Recordemos que el operador = se utiliza para la asignación de variables ("meter valores en cajas"). El == en cambio se utiliza para expresar condiciones, y funciona como la igualdad matemática.

El == es un ejemplo de los que se denominan operadores de comparación. Los veremos en más detalle muy pronto.

Veamos un segundo ejemplo de programa, que muestre un mensaje de acuerdo a la paridad del número ingresado:

```
int main()
{
    int x;
    cin >> x;
    if (x % 2 == 0)
    {
        cout << "El numero " << x << " es par." << endl;
    }
    if (x % 2 == 1)
    {
        cout << "El numero " << x << " es impar." << endl;
    }
    return 0;
}
```

Recordemos que % es el operador de "resto de la división", también llamado módulo. Para saber si un número es par o impar, debemos considerar el resto de la división por 2. Cuando el resto sea 0, el número es par, y cuando sea 1, es impar. Esas son las condiciones que verificamos en los ifs de este programa, y en cada caso, imprimimos un mensaje apropiado para la condición que se cumple.

El else

En el último ejemplo, verificábamos si un número era par o impar, e imprimíamos un mensaje de acuerdo a la paridad. Notemos que en este caso tenemos dos opciones excluyentes: O bien el número es par, y entonces solamente se imprime el mensaje para el caso par, o bien esto no ocurre(porque el número es impar), y entonces solamente se imprime el mensaje para el caso impar.

Esta situación, en la cual hay una cierta condición, y se debe ejecutar un conjunto de instrucciones cuando se cumple la condición, y otro conjunto cuando no se cumple, es muy común. Para ello existe una parte opcional de la instrucción if, que hasta ahora no hemos utilizado, y es la sección `else`.

Es posible escribir lo siguiente:

```
if (condicion)
{
   instruccionA1;
   instruccionA2;
   instruccionA3;
   ...
}
else
```

```
{
   instruccionB1;
   instruccionB2;
   instruccionB3;
   ...
}
```

En este caso, el primer bloque de instrucciones (instruccionA1, instruccionA2, etc) corresponde al "if normal", y se ejecutará cuando la condición indicada entre paréntesis ocurra. En cambio, el segundo bloque de instrucciones (instruccionB1, instruccionB2, etc) se pone a continuación de `else`, y se ejecuta cuando la condición del if no ocurre. De esta forma, uno solo de los bloques de instrucciones será ejecutado, dependiendo de si la condición del if vale o no.

A continuación mostramos el ejemplo anterior de par o impar, reescrito utilizando la instrucción else:

```
int main()
{
    int x;
    cin >> x;
    if (x % 2 == 0)
    {
        cout << "El numero " << x << " es par." << endl;
    }
    else
    {
        cout << "El numero " << x << " es impar." << endl;
    }
    return 0;
}
```

En este caso, en lugar de utilizar un segundo `if` con la condición (x % 2 == 1), como esta condición es la que ocurre exactamente cuando no ocurre la primera, podemos simplemente extender el primer `if` con un `else`, para indicar qué hacer cuando el número no es par (en cuyo caso, será impar).

Operadores de comparación

Para expresar las condiciones anteriores, hemos utilizado los operadores <, >, e ==. Estos operadores se utilizan para expresar condiciones, mediante la comparación de otros dos valores. Así, `x > 10` expresa la condición de que el valor almacenado en `x` debe ser mayor que 10. Similarmente, `x + y == z` expresa la condición de que el valor almacenado en `x`, más el valor almacenado en `y`, debe ser igual al valor almacenado en `z`.

Estos operadores se llaman operadores de comparación. A continuación mostramos los más importantes operadores de comparación, junto a un texto que indica su significado:

```
== "Igual a"
!= "Distinto de"
<  "Menor a"
>  "Mayor a"
<= "Menor o igual a"
>= "Mayor o igual a"
```

Cada uno de estos operadores puede utilizarte para comparar los valores de dos expresiones, obteniéndose así una condición que puede utilizarse en un if. Recordar siempre que el operador de comparación "==", y el operador de asignación "=" son completamente diferentes, y mezclarlos puede llevar a errores en el comportamiento del programa.

Ya hemos usado estos operadores con valores de tipo int: en dicho caso, la comparación se realiza por valor numérico. También es posible utilizar los operadores de comparación con variables string (textos): En este caso, las palabras se ordenan en el orden del diccionario. El nombre técnico para el orden del diccionario (Donde primero van las palabras con A, luego con B, luego con C, etc) es "orden lexicográfico". Por ejemplo, tendremos que "vaca" > "sopa", y "abcd" < "bcda". Notar sin embargo que como ya mencionamos, a cada caracter (char) corresponde un valor numérico ASCII, y las mayúsculas y minúsculas tienen valores distintos. Como C++ ordena las variables de tipo string usando estos códigos, cadenas que mezclen mayúsculas y minúsculas no se ordenarán según el orden normal del diccionario, ya que en ASCII todas las mayúsculas vienen antes que todas las minúsculas. Así por ejemplo, si bien "burro" > "agua", tenemos que "Burro" < "agua", pues la 'B' viene antes que la 'a', que viene antes que la 'b'.

Versión con una única instrucción

Hemos visto que la sintaxis (escritura) completa del `if` tiene la siguiente forma:

```
if (condicion)
{
    instrucciones;
}
else
{
    instrucciones;
}
```

Además, ya hemos mencionado que la parte del `else`, para especificar qué hacer cuando no se cumple la condición, es opcional . Una opción adicional que existe en C++ en el caso del `if` (o el `else`), es la de no utilizar las llaves cuando el bloque de instrucciones que delimitan contiene una sola instrucción. En este caso, si no usamos llaves, C++ asumirá que la primera instrucción que sigue a continuación conforma el bloque completo.

Veamos algunos ejemplos:

```
// Escritura completa
if (x > 0)
{
    cout << "El numero es positivo" << endl;
}
cout << "Fin del programa" << endl;

// Escritura sin llaves
if (x > 0)
    cout << "El numero es positivo" << endl;
cout << "Fin del programa" << endl;
```

Los dos ejemplos anteriores son equivalentes, ya que hay una sola instrucción entre llaves. Notemos en cambio que los siguientes dos ejemplos no son equivalentes:

```
// Escritura completa
if (x > 0)
{
    cout << "El numero ";
    cout << "es positivo" << endl;
}
cout << "Fin del programa" << endl;

// Escritura sin llaves:
// CAMBIA EL SIGNIFICADO!
// Las llaves anteriores NO SE PUEDEN OMITIR.
if (x > 0)
    cout << "El numero ";
    cout << "es positivo" << endl;
cout << "Fin del programa" << endl;

// La version anterior sin llaves equivale a esto:
if (x > 0)
{
    cout << "El numero ";
}
cout << "es positivo" << endl;
cout << "Fin del programa" << endl;
```

En el ejemplo anterior, vemos que omitir las llaves cuando el bloque de instrucciones que queremos ejecutar tiene más de una instrucción es un error. Solamente es posible omitir las llaves, cuando el bloque tiene una única instrucción . Ante la duda , o posibilidad de confusión, es mejor dejar las llaves aunque se utilice una única instrucción, para evitar problemas.

Existe un peligro más cuando omitimos las llaves, y este peligro es el resultado de que en C++ la parte `else` sea opcional. Supongamos un código como el siguiente:

```
// Ejemplo 1 (resulta correcto)
if (x > 0)
    if (x % 2 == 0)
        cout << "Positivo par" << endl;
    else
        cout << "Positivo impar" << endl;
```

```
// Ejemplo 2 (resulta incorrecto)

if (x > 0)
    if (x % 2 == 0)
        cout << "Positivo par" << endl;
else
    cout << "Negativo" << endl;
```

Notemos que, más allá de los mensajes que se van a mostrar, la única diferencia entre el primer y el segundo ejemplo es la indentación (cantidad de espacios a la izquierda) del `else`. Pero al compilador no le importa la cantidad de espacios . Por lo tanto, ambos casos son interpretados de idéntica manera. En este caso... ¿Corresponde para C++ el `else` al primer `if`, como querríamos en el ejemplo2, o corresponde al segundo, como querríamos en el ejemplo 1?

La regla que sigue C++, que determina cómo resolver esta confusión cuando no se usan llaves en el `if`, es que un `else` en el código corresponde al `if` inmediatamente anterior. Es decir, el compilador interpreta la situación como querríamos en el ejemplo 1, con lo cual el ejemplo 2 nos resulta incorrecto. Para escribir lo que querríamos en el ejemplo 2, es necesario sí o sí utilizar llaves.

Veamos a continuación entonces cómo escribir los ejemplos con la escritura completa con llaves:

```
// Ejemplo 1 (con llaves, correcto)
if (x > 0)
{
    if (x % 2 == 0)
    {
        cout << "Positivo par" << endl;
    }
    else
    {
        cout << "Positivo impar" << endl;
    }
}

// Ejemplo 2 (con llaves, correcto)

if (x > 0)
{
    if (x % 2 == 0)
    {
        cout << "Positivo par" << endl;
    }
}
else
{
    cout << "Negativo" << endl;
}
```

Notar que ahora en el ejemplo 2, al tener todas las llaves, el compilador no puede confundirse y emparejar el `else` con el `if` interno, ya que ese `if` se encuentra dentro del bloque de instrucciones entre llaves, y por lo tanto no puede corresponderse con un `else` que está por fuera de las llaves.

Ejercicios

- Escribir un programa que lea un numero, y nos diga si es múltiplo de 3 o no. [http://juez.oia.unsam.edu.ar/#/task/multi_tres/statement]
- Escribir un programa que lea dos palabras, y nos diga cuál viene primero en el diccionario. [http://juez.oia.unsam.edu.ar/#/task/cual_va_primero/statement]
- Escribir un programa que lea un número de año, y nos diga si es bisiesto. [http://juez.oia.unsam.edu.ar/#/task/es_bisiesto/statement]

Operadores lógicos

A veces, queremos expresar condiciones compuestas , en base a otras condiciones más simples. Los operadores lógicos sirven para obtener tales condiciones. A continuación mostramos los operadores lógicos más comunes, junto con su nombre:

```
&& "and"
|| "or"
!  "not"
```

El operador `&&`, denominado "and", se utiliza para formar una condición compuesta en la cual se exige que otras dos condiciones se cumplan al mismo tiempo . Por ejemplo, `x > 0 && x % 2 == 0` expresa la condición de que x sea un número par positivo, indicando que deben cumplirse tanto `x > 0` como `x % 2 == 0`. `x > 0 && x < 0`, por ejemplo, denota una condición imposible de cumplir, ya que se pide que x sea positivo y negativo.

El operador `||`, denominado "or", se utiliza para formar una condición compuesta en la cual se exige que al menos una de otras dos condiciones se cumpla. Así por ejemplo, cero será el único número que no cumple la condición `x > 0 || x < 0`: En esta condición se pide que x sea positivo o negativo. El único número que no cumple ninguna de ellas es el cero. Otro ejemplo es `x % 2 == 0 || x % 3 == 0`, en el cual se pide que x sea múltiplo de 2 o de 3. Además, si x resulta ser múltiplo de ambos (como por ejemplo, 12), la condición igualmente se cumple, pues con una sola que se cumpla basta, y si se cumplen las dos "mejor".

Finalmente, el operador ! se usa para invertir una condición dada, que generalmente deberá encerrarse entre paréntesis. Por ejemplo, `!(x < 0)` es completamente equivalente a `(x >= 0)`. Por otro lado, `!(x % 2 == 0 || x % 3 == 0)`, por ejemplo, estaría negando la condición anterior de que el número x sea múltiplo de 2 o de 3, y por lo tanto esta condición solamente será cierta cuando el número no sea múltiplo ni de 2 ni de 3.

Utilizando estos operadores lógicos, podemos muchas veces resumir largas cadenas de ifs, en una sola condición compuesta más clara. Veamos por ejemplo el siguiente ejemplo para decidir si un año es bisiesto:

```
#include <iostream>

using namespace std;

int main()
{
    int year;
    cin >> year;
    if (year % 400 == 0)
        cout << "Es bisiesto." << endl;
    else
    {
        // En este caso, no es multiplo de 400
        if (year % 100 == 0)
            cout << "No es bisiesto " << endl;
        else
        {
            // En este caso, no es multiplo de 100
            if (year % 4 == 0)
                cout << "Es bisiesto." << endl;
            else
                cout << "No es bisiesto." << endl;
        }
    }
    return 0;
}
```

Si bien es correcto, este código puede resumirse en el siguiente mucho más claro, utilizando expresiones lógicas:

```
#include <iostream>

using namespace std;

int main()
{
    int year;
    cin >> year;
    if (year % 400 == 0 || (year % 4 == 0 && year % 100 != 0))
        cout << "Es bisiesto." << endl;
    else
        cout << "No es bisiesto " << endl;
    return 0;
}
```

Notar que encerramos entre paréntesis la condición `(year % 4 == 0 && year % 100 != 0)`. Siempre es conveniente utilizar paréntesis para indicar el orden de las operaciones lógicas, cuando encadenamos una mezcla de operaciones `||` y `&&`.

El tipo bool

Ya hemos visto los tipos `int`, `string` y `char`. Veremos ahora un tipo de datos adicional: El tipo `bool`.

Un `bool` representa el resultado de analizar si una condición es cierta o falsa. Por lo tanto, un valor `bool` representa un sí o un no. En C++, el sí se escribe `true` (del inglés "verdadero") y el no se escribe `false` (del inglés, "falso").

Ahora podemos entender que todos los operadores de comparación que vimos, así como todos los operadores lógicos, operan con expresiones y producen resultados de tipo `bool`: `true` cuando la condición analizada es cierta, y `false` cuando la condición analizada no lo es.

Por ejemplo, en el caso de los operadores de comparación, `1 < 2` da por resultado `true`, que es un valor de tipo `bool`. `1 == 2` es otro valor de tipo `bool`, que será `false`. Como con cualquier otro tipo de datos, podemos declarar variables `bool`:

```
bool cond1 = 1 == 2;
bool cond2 = 1 < 2;
if (cond1)
    cout << "Esto no se ejecuta" << endl;
if (cond2)
    cout << "Esto si se ejecuta" << endl;
if (cond1 || cond2)
    cout << "Esto si se ejecuta" << endl;
if (cond1 && cond2)
    cout << "Esto no se ejecuta" << endl;
```

En este ejemplo, vemos que las operaciones lógicas `||` y `&&` operan con 2 valores de tipo bool: `||` da por resultado `true` cuando al menos uno de los operandos lo es, y `&&` devuelve `true` solamente cuando ambos operandos lo son.

Además, en este ejemplo vemos que lo que hemos llamado `condición`, y que se debe colocar entre paréntesis en el `if`, en realidad puede ser cualquier expresión de tipo bool. Esto permite guardar valores bool intermedios en variables, y usarlos libremente en expresiones compuestas mediante operadores lógicos y de comparación.

Más ejercicios

- Escribir un programa que lea tres números, y los vuelva a imprimir pero ordenados de menor a mayor. [http://juez.oia.unsam.edu.ar/#/task/tri_sort/statement]
- Escribir un programa que lea una hora en formato de 24 horas (exactamente 5 caracteres) y la imprima en forma AM / PM. [http://juez.oia.unsam.edu.ar/#/task/la_hora/statement] Ejemplos:

```
23:12 imprime 11:12 PM
10:15 imprime 10:15 AM
12:15 imprime 12:15 PM
00:15 imprime 12:15 AM
```

- Leer dos numeros de hasta 2 digitos, e imprimir una "cuenta" del producto. [http://juez.oia.unsam.edu.ar/#/task/productis/statement] La cuenta debe estar bien alineada a derecha. Ejemplos:

```
Si vienen 12 y 5,
   12
 x  5
 ----
   60
Si vienen 10 y 10,
   10
 x 10
 ----
  100
Si vienen 50 y 60,
   50
 x 60
 ----
 3000
Si vienen 0 y 99,
    0
 x 99
 ----
    0
```

Estructuras de control repetitivas

Hasta ahora, la cantidad de instrucciones de nuestro programa que se ejecutarán está acotada : Esto es, siempre especificamos una lista de instrucciones, y cada una se ejecutará como mucho una vez (y algunas podrían no ejecutarse, si utilizamos las estructuras de control selectivas).

Sin embargo, si queremos realizar una tarea una única vez, generalmente podríamos realizarla manualmente y listo: nos interesa utilizar la computadora para automatizar tareas repetitivas , en las que haya que repetir cómputos mecánicamente una y otra vez, hasta llegar a un resultado. Veremos en esta lección cómo se puede lograr esto en C++.

La instrucción while

La instrucción `while` sirve para instruir a la computadora a que lleve a cabo un determinado conjunto de instrucciones, mientras se cumpla una condición específica .

La sintaxis (forma de escritura) de esta instrucción es idéntica al `if` común (sin `else`), pero utilizando en su lugar la palabra `while`:

```
while (condicion)
{
    instruccion1;
    instruccion2;
    //...
    instruccionFinal;
}
```

Al igual que ocurría con el `if`, los paréntesis alrededor de la condición son obligatorios, y las instrucciones del `while` se encuentran en un bloque encerrado entre llaves . También es posible omitir las llaves si el `while` contiene una sola instrucción, como en el caso del `if`.

Cuando la computadora se topa con un `while`, se determina si la condición indicada en el `while` es cierta: si no lo es, se saltean todas las instrucciones del `while`, exactamente igual que ocurre con el `if`. La diferencia es que, si la condición ocurre, entonces el bloque de instrucciones se ejecuta por completo como ocurría con el `if`, pero luego de eso se vuelve a revisar la condición desde el comienzo nuevamente : la computadora continuará repitiendo el bloque de instrucciones entre llaves una y otra vez, hasta que llegue un punto en que la condición no ocurre. En otras palabras, la computadora continuará ejecutando las instrucciones mientras que la condición sea verdadera.

Veamos un primer ejemplo sencillo de esto, en el siguiente programa:

```
#include <iostream>

using namespace std;

int main()
{
    int x;
    cin >> x;
    while (x > 0)
    {
        cout << "Recibido " << x << ", que es positivo." << endl;
        cin >> x;
    }
    cout << "El numero " << x << " no es positivo!" << endl;
```

```
    return 0;
}
```

Si ejecutamos es programa, por cada número que ingresemos, el programa nos mostrará por pantalla un mensaje con dicho número, y continuará haciendo esto mientras que el número que ingresamos sea positivo . Esto es porque en cada paso, se examina la condición `x>0`, para ver si es positivo el valor almacenado en `x`, y mientras que lo sea se continúa ejecutando el cuerpo del `while`: es decir, se muestra el mensaje y se lee en `x` un nuevo valor ingresado por el usuario.

Recomendamos al lector ejecutar este programa para ver el efecto que tiene, y analizar cómo siguiendo las instrucciones mecánicamente de la forma que hemos explicado, se explica perfectamente el comportamiento de la computadora.

Imaginemos ahora otro ejemplo: supongamos que deseamos mostrar por pantalla todos los números desde `1` hasta `N`, una por línea, siendo `N` un cierto valor que el usuario pueda ingresar. Como la cantidad de cosas que tenemos que escribir no está prefijada, sino que depende de lo que ingrese el usuario, tendremos que utilizar necesariamente alguna estructura repetitiva, para poder así imprimir muchas veces.

Veamos cómo puede llevarse a cabo esta tarea utilizando while. Para eso, deberemos encontrar una condición que nos permita saber cuándo seguir escribiendo, y cuándo parar. En estos casos puede ser útil preguntarse lo siguiente: ¿Cómo realizaríamos esta tarea, si tuviéramos que realizarla manualmente, de manera mecánica? Si tuviéramos que escribir por ejemplo, los números del 1 al 1000, lo que haríamos sería ir contando : escribimos el 1, luego pasamos al siguiente número , que es el 2, y lo escribimos, luego pasamos al siguiente , que es el 3, y lo escribimos, y así seguiríamos mientras no hayamos escrito todos los números que queríamos . ¿Cómo podemos saber si ya escribimos todos los números? Como solamente queremos escribir números hasta N, queremos seguir escribiendo mientras que el número a escribir a continuación sea menor o igual que N .

Teniendo esto en cuenta, podemos escribir el siguiente programa para mostrar todos los números desde 1 hasta un cierto entero positivo que ingrese el usuario:

```
#include <iostream>

using namespace std;

int main()
{
    int N;
    cin >> N;
    int x = 1;
    while (x <= N)
    {
        cout << x << endl;
        x++; // Es lo mismo que x = x + 1 en C++: incrementa el valor de x en 1
    }
    return 0;
}
```

Notar que necesitamos utilizar una variable `x`, que sirve para saber cuál es el numero actual, que vamos a escribir en cada paso. Con esta variable vamos contando los números que pasan, comenzando desde el 1, y aumentando en cada paso mientras que sea `x <= N`. Se suele decir que `x` es un contador .

Este mecanismo es muy común, y es el que utilizamos cuando tenemos que repetir ciertas operaciones sobre todo un rango de números: utilizamos una variable como contador, de forma tal manera que en el cuerpo del while, realizamos las operaciones utilizando el valor del contador (en nuestro ejemplo, `x`). Y luego de cada paso, cambiamos el contador al siguiente valor , permitiendo que en la siguiente iteración se procese el siguiente número.

La instrucción for

El patrón del ejemplo anterior es muy común: tenemos una inicialización justo antes del `while`, donde guardamos en el contador un valor inicial adecuado: `int x = 1;` en el ejemplo. Luego, tenemos el `while` con una condición que indica cuándo hay que seguir procesando: `x <= N` en el ejemplo. Finalmente, para pasar al siguiente número, al final de cada paso aumentamos `x` al siguiente valor: usamos en el ejemplo la instrucción `x++`. Notar que este patrón es independiente de las operaciones que vayamos a realizar sobre los valores de x : en cualquier caso en el que queramos recorrer todos los números de 1 hasta N para hacer algo con ellos, tendremos un código muy similar, con esas 3 partes.

Este patrón en 3 partes (inicialización, while con condición, e incremento del contador al final del while) es tan común que existe una instrucción que lo resume para facilitar la lectura de los programas: es la instrucción `for`.

La sintaxis del `for` es:

```
for(inicializacion; condicion; incremento)
{
    // Cuerpo de instrucciones a realizar
}
```

Es decir, las 3 partes del patrón anterior se ponen todas juntas, entre paréntesis y separadas por punto y coma, en el momento de declarar el for. Un for como el anterior es equivalente a:

```
{
  inicializacion;
  while(condicion)
  {
      // Cuerpo de instrucciones a realizar
      incremento;
  }
}
```

De esta forma, el ejemplo anterior generalmente se escribiría utilizando un for, de la siguiente manera:

```
#include <iostream>

using namespace std;

int main()
{
    int N;
    cin >> N;
    for(int x = 1; x <= N; x++)
        cout << x << endl;
    return 0;
}
```

Notar que podemos omitir las llaves en este último ejemplo porque el cuerpo del for contiene una única instrucción: el incremento `x++` ya no se pone en el cuerpo al utilizar el for. Generalmente, la escritura con for es más clara porque separara la parte de la iteración , utilizada para recorrer los valores de x que nos interesan , del cuerpo principal donde realizamos las operaciones que nos interesan sobre cada valor de x.

Además notemos que es válido declarar la variable `x` en la parte de inicialización del for: El ámbito de dicha variable es todo el contenido del for (tanto el encabezado de 3 partes, como el cuerpo de instrucciones). La variable `x` declarada en la inicialización del for no puede utilizarse fuera del for. Si se vuelve a realizar otro for similar, se estaría utilizando una variable x diferente .

Ejercicios

1. Escribir un programa que lea un número `N`, y escriba en pantalla un solo número: la suma de todos los números desde 1 hasta `N`.
2. Escribir un programa que lea un número `N`, y escriba en pantalla un solo número: la suma de todos los números desde 1 hasta `N` que sean múltiplos de 3.
3. Escribir un programa que lea un número `N`, y escriba en pantalla un solo número: la suma de todos los números desde 1 hasta `N` que sean múltiplos de 3 pero no de 5.
4. Escribir un programa que lea un número `N`, y escriba en pantalla un solo número: el producto de todos los números desde 1 hasta `N` (a este número se lo conoce como N factorial , y se escribe N!).
5. Escribir un programa que lea un número `N`, y escriba en pantalla un solo número: la suma todos los números desde 1 hasta `N`, pero elevados al cuadrado (por ejemplo, para N=3 la respuesta es 14=1+4+9).

Soluciones a los ejercicios

1. El siguiente código muestra un ejemplo de solución:

   ```
   #include <iostream>

   using namespace std;
   ```

```
int main()
{
    int N, suma = 0;
    cin >> N;
    for(int x = 1; x <= N; x++)
        suma = suma + x;
    cout << suma << endl;
    return 0;
}
```

Se puede observar que utilizamos una variable auxiliar `suma`, que comienza en `0`, y en cada paso lo que hacemos sumarle el número actual. De esta forma, como en cada paso el valor de `suma` es aumentado en el número correspondiente, al final del proceso tendrá la suma de todos los números, y por eso escribimos su valor al final. La instrucción `suma = suma + x` justamente aumenta el valor de suma, porque lo que allí se indica es que se guarde en la variable suma, el valor que hay ahora en la variable suma, más el valor de x. Esta instrucción también puede abreviarse en C++ como `suma += x` (similarmente, existen operadores `-=` para restar, `*=` para multiplicar, etc).

2. Este ejemplo es muy parecido al anterior: basta agregarle un if para que solamente se ejecute la operación de suma, cuando el número actual es múltiplo de 3.

```
#include <iostream>

using namespace std;

int main()
{
    int N, suma = 0;
    cin >> N;
    for(int x = 1; x <= N; x++)
    if (x % 3 == 0)
        suma += x;
    cout << suma << endl;
    return 0;
}
```

3.
```
#include <iostream>

using namespace std;

int main()
{
    int N, suma = 0;
    cin >> N;
    for(int x = 1; x <= N; x++)
    if (x % 3 == 0 && x % 5 != 0)
        suma += x;
    cout << suma << endl;
    return 0;
}
```

4. Este ejercicio es casi igual al primero, pero en lugar de sumar los números, queremos multiplicarlos. Utilizaremos `*` entonces en lugar de `+` para la operación.

```
#include <iostream>

using namespace std;

int main()
{
    int N, producto = 1;
    cin >> N;
    for(int x = 1; x <= N; x++)
        producto *= x;
    cout << producto << endl;
    return 0;
}
```

Un detalle importante es que al calcular un producto, inicializamos la variable auxiliar en uno , y no en cero como hacíamos en el caso de la suma. Notar que como le vamos multiplicando cada vez más números, si comenzara en cero,

quedaría en cero para siempre. Los factoriales son números que crecen muy rápidamente: si ejecutamos el programa, veremos que ya con valores de `N` mayores que 12 obtenemos resultados demasiado grandes para el tipo de datos int, con lo cual veremos resultados erróneos, y a veces incluso negativos.

5.
```
#include <iostream>

using namespace std;

int main()
{
    int N, suma = 0;
    cin >> N;
    for(int x = 1; x <= N; x++)
        suma += x*x;
    cout << suma << endl;
    return 0;
}
```

La instrucción do-while

La última estructura de control repetitiva es la instrucción do-while, que es la menos utilizada de todas. Esta instrucción funciona igual que `while`, pero con la única salvedad de que la condición se verifica luego de cada paso, en lugar de antes de cada paso. Por lo tanto, do-while siempre realiza al menos un paso, mientras que con un while, si la condición es falsa al comenzar, no se realiza ningún paso.

```
do
{
   // cuerpo de instrucciones a repetir
} while (condicion);
```

Ejercicios

- Escribir un programa que lea un numero N, y luego imprima la suma de los números pares, menos la suma de los impares, para los primeros N naturales [http://juez.oia.unsam.edu.ar/#/task/pares_impares/statement]. Por ejemplo, para N=3 sería 2 - (1+3), para N=6 sería (2+4+6)-(1+3+5), para N=1 sería -1.
- Escribir un programa que lea una palabra, y la imprima encerrada en un cuadrado de asteriscos [http://juez.oia.unsam.edu.ar/#/task/pollo_pan/statement]. Ejemplos:

```
Si lee "pollo" imprime:
 *******
 *pollo*
 *******
Si lee "pan" imprime:
 *****
 *pan*
 *****
```

- Escribir un programa que lea N números, y visualice el máximo, el mínimo y la suma [http://juez.oia.unsam.edu.ar/#/task/max_min_sum/statement]. El valor de N se solicita al comenzar.
- Escribir un programa que lea una palabra, y la imprima al revés (leída de derecha a izquierda) [http://juez.oia.unsam.edu.ar/#/task/string_reverser/statement].
- Un número es perfecto, cuando la suma de sus divisores es igual al mismo número. Crear un programa que dado un N, busque y encuentre todos los números perfectos hasta N [http://juez.oia.unsam.edu.ar/#/task/busca_perfectos/statement].
- Un número es primo, cuando no tiene divisores que no sean 1 o el mismo número. Hacer un programa que imprima los primos hasta N, para N dado por la entrada [http://juez.oia.unsam.edu.ar/#/task/busca_primos/statement].

Vectores

Motivación

Supongamos que queremos crear programas capaces de realizar las siguientes tareas:

1. Dada una secuencia de números, determinar si tiene repetidos.
2. Dada una secuencia de números, decidir si es "capicúa".
3. Dada una secuencia de números, imprimirla al revés.

Recomendamos fuertemente al lector que piense primero cómo haría para resolver estas consignas con las herramientas que ya hemos enseñado. Probablemente le resulte muy difícil, y es bueno notar esa dificultad.

A continuación veremos una nueva herramienta muy poderosa que podemos agregar a nuestros programas. Sin ella, resulta imposible (o al menos mucho más difícil) resolver cualquiera de los problemas planteados en esta lección.

El tipo vector

Así como `int` es un tipo de datos que usamos para guardar y manipular números enteros, `string` es un tipo de datos que usamos para guardar textos, `char` el que usamos para guardar letras individuales y `bool` el que usamos para guardar valores trueo falsę aprenderemos ahora un nuevo tipo que sirve para guardar listas de valores : es el tipo vector.

Para poderlo utilizar, hay que poner `#include <vector>` al comienzo del programa, exactamente igual que ocurría con el tipo `string`. Una particularidad del tipo vector es que es lo que se denomina un tipo paramétrico : Esto lo que significa es que no hay un único tipo de vector, sino que depende del tipo de elementos que tenga la lista .

Así, podemos imaginar por ejemplo las lista `{"goma", "espuma", "goma", "eva"}`, que es una lista de textos, o sea una lista de string , de longitud 4. O podemos imaginar también la lista `{2,3,5,7,11}`, que es una lista de enteros, o sea una lista de int de longitud 5. En el primer caso, tendremos entonces un vector de string , que se escribe `vector<string>`, y en el segundo un vector de int , que se escribe `vector<int>`.

Por ejemplo, es válido en un programa [1] declarar vectores de la forma mostrada:

```
vector<string> v1 = {"goma", "espuma", "goma", "eva"};
vector<string> v2 = {"a", "b", "c"};
vector<int> v3 = {2,3,5,7,11};
```

Como con los otros tipos, es posible utilizar el operador de asignación para copiar toda una lista de una variable a otra: `v1 = v2;` sería una instrucción válida en el caso anterior, que copia todo el contenido de `v2` y lo guarda en `v1` (se perdería por lo tanto completamente la lista que había en `v1`, es decir, `{"goma", "espuma", "goma", "eva"}`). En cambio, sería inválido hacer `v2 = v3;`, y tal instrucción generaría un error al compilar: `vector<int>` y `vector<string>` son ambos vectors, pero de distinto tipo. Una lista de enteros y una lista de textos no son intercambiables .

Ya hemos mencionado que lo que distingue a un tipo de los demás son las operaciones que pueden realizarse con los valores de ese tipo. A continuación, mostramos las principales operaciones que pueden realizarse con los vectors (que, recordemos, se usan para guardar listas , secuencias o vectores de valores):

- Si `v` es un vector, `v.size()` indica su tamaño (cantidad de elementos de la lista). Esto se parece a lo que ya vimos para string.

- Podemos indicar un valor de tipo vector directamente dando su lista de elementos entre llaves: `{1,2,3}` o `{5}`, o incluso es posible usar `{}` para indicar un vector vacío.
- Podemos crear un vector de una cierta longitud directamente al declararlo, indicando la longitud entre paréntesis: `vector<int> v(1000)` crea un vector de tamaño 1000, y `vector<int> v(N)` crea un vector de tamaño `N`, donde N es por ejemplo una variable de tipo int. Si queremos indicar un valor inicial específico para todas las posiciones del vector, utilizamos para eso un segundo número: `vector<int> v(1000,5)` declara una nueva variable v de tipo vector, que tendrá 1000 elementos, y todos ellos serán un 5. Similarmente, `vector<int> v(5, 2);` es lo mismo que `vector<int> v = {2,2,2,2,2};`
- De manera similar a lo que pasaba con los strings, podemos referirnos a un elemento puntual de un vector utilizando corchetes y la posición que nos interesa, comenzando desde cero . Por ejemplo, si v es un `vector<int>`, `v[0]` es el primer elemento de la lista, `v[9]` sería el décimo, y `v[v.size()-1]` sería el último.
- Los corchetes pueden usarse tanto para leer como para escribir los valores del vector: `cout << v[1]` mostraría por pantalla el segundo elemento del vector, mientras que `v[2] = 10` cambia el tercer elemento del vector, de manera que ahora tenga un 10. Otro uso muy común es leer de la entrada: `cin >> v[i]` lee de la entrada y lo guarda en la posición `i` del vector `v`, donde `i` generalmente será una variable contadora que indica la posición donde vamos a guardar.

La gran conveniencia que aporta el vector que no teníamos antes es la posibilidad de crear un número arbitrario de variables . Creando un vector de tamaño 1000 tenemos a nuestra disposición 1000 variables (`v[0]`, `v[1]`, `v[2]`, ... , `v[998]`, `v[999]`), sin tener que escribir 1000 líneas de código diferentes.

Otra ventaja importante de usar vector es a la hora de procesar una secuencia que viene en la entrada del programa: si no usamos vector, no podremos almacenar toda la secuencia a la vez, y entonces en una sola pasada a medida que vamos leyendo cada valor, tendremos que realizar todas las operaciones que nos interesen. Al tener vector, podemos primero leer toda la secuencia y guardarla en el vector, y luego recorrerla todas las veces que nos sea cómodo, realizando los cómputos que queramos.

Soluciones a los problemas planteados anteriormente

En estos tres ejemplos, así como en los cinco ejercicios que siguen, siempre que el programa reciba secuencias, supondremos que primero se lee un entero `N`, con la cantidad de elementos, y luego se leen los elementos de la secuencia en sí.

1. ```
   #include <iostream>
   #include <vector>

   using namespace std;

   int main()
   {
       int N;
       cin >> N;
       vector<int> v(N);
       for (int i = 0; i < N; i++)
           cin >> v[i];
       bool huboRepeticion = false;
       for (int i = 0; i < N; i++)
           for (int j = 0; j < i; j++)
               if (v[j] == v[i])
                   huboRepeticion = true;

       if (huboRepeticion)
           cout << "La secuencia tiene repetidos" << endl;
       else
           cout << "Todos los elementos de la secuencia son distintos" << endl;
       return 0;
   }
   ```

2. ```
   #include <iostream>
   #include <vector>

   using namespace std;

   int main()
   {
       int N;
       cin >> N;
       vector<int> v(N);
   ```

```
    for (int i = 0; i < N; i++)
        cin >> v[i];
    bool esCapicua = true;
    for (int i = 0; i < N; i++)
        if (v[i] != v[N-1-i])
            esCapicua = false;

    if (esCapicua)
        cout << "La secuencia es capicua" << endl;
    else
        cout << "La secuencia NO es capicua" << endl;
    return 0;
}
```

3. ```
   #include <iostream>
   #include <vector>

   using namespace std;

   int main()
   {
       int N;
       cin >> N;
       vector<int> v(N);
       for (int i = 0; i < N; i++)
           cin >> v[i];
       for (int i = N-1; i >=0; i--)
       {
           if (i != N-1) // No ponemos el espacio al principio de la linea
               cout << " ";
           cout << v[i];
       }
       cout << endl;
       return 0;
   }
   ```

Ejercicios

1. Dada una secuencia de números, determinar cuántas veces aparece cada uno de los números del 1 al 10 en la secuencia [http://juez.oia.unsam.edu.ar/#/task/cuenta_numeros/statement].
2. Dada una secuencia de números, determinar cuál es el número que más veces aparece en la secuencia, así como cuántas veces aparece [http://juez.oia.unsam.edu.ar/#/task/mas_aparece/statement].
3. Dada una secuencia de números distintos, determinar cuántos pares de números hay cuya suma sea un múltiplo de 10 [http://juez.oia.unsam.edu.ar/#/task/cuenta_pares/statement].
4. Dada una secuencia de números distintos, determinar cuántos pares de números hay cuya suma sea un número primo [http://juez.oia.unsam.edu.ar/#/task/cuenta_primos/statement].
5. Dada una secuencia de números distintos, determinar cuántas ternas (combinaciones de tres) de estos números forman un triángulo [http://juez.oia.unsam.edu.ar/#/task/cuenta_triangulos/statement] (considerando que los números son las longitudes de los lados). Por ejemplo, 2 4 5 son longitudes que forman un triángulo, pero 1 2 5 no (si lo intentamos, primero dibujamos el lado de 5, y luego los lados de 1 y 2 entre los dos son demasiado cortos y "no alcanzan" a cerrar un triángulo junto con el lado de 5).

Modificando el tamaño de un vector

Cuando sabemos la cantidad de elementos que tiene o va a tener nuestra lista, en general lo mejor es crear directamente un vector de ese tamaño. Sin embargo, a veces queremos cambiar el tamaño de una lista: los motivos más comunes son para poder agregar o sacar elementos a una lista existente.

Para esto tenemos en C++ tres operaciones útiles de vector:

- Si v es un vector, con `v.resize(nuevoTam)` podemos cambiar su tamaño al nuevo valor, que está indicado por `nuevoTam`. Si este valor es más chico que el tamaño actual de `v`, los elementos sobrantes del final se pierden.
- Con `v.push_back(e)`, agregamos el elemento `e` al final de toda la lista. Por ejemplo si `v` es un `vector<int>`, `v.push_back(15)` le agrega un 15 al final. El tamaño de un vector aumenta en 1 cuando se hace push_back.
   ```

- Con `v.pop_back()` podemos borrar el último elemento de la lista. El tamaño se reduce en 1 al hacer pop_back. Es por lo tanto una forma más práctica de hacer `v.resize(v.size()-1)`.

Ejercicio

1. Se debe leer una secuencia de números positivos que viene de la entrada, pero no sabemos su longitud: la secuencia termina cuando viene un cero, que indica que ya no hay que leer más. El programa debe determinar si la secuencia dada tiene repetidos o no [http://juez.oia.unsam.edu.ar/#/task/busca_repetidos/statement].

Sobre el uso de ".size()" en comparaciones

Si compilamos con todos los warnings activados, podremos observar que el compilador genera una advertencia cuando utilizamos `.size()` en una comparación con enteros. Por ejemplo en un simple recorrido:

```
for (int i = 0; i < v.size(); i++)
    cout << v[i] << endl;
```

El compilador generará un warningle que estamos comparando `.size()` contra enteros (en este caso, contra `i` en la parte `i < v.size()`). Esto es porque `.size()` en realidad no genera un `int`, sino que genera un `unsigned int`, que es siempre no negativo y tiene ciertas diferencias que pueden generar errores fácilmente, por lo que recomendamos usar siempre `int` y olvidarse de `unsigned int`.

Si bien en este ejemplo es relativamente inofensivo, ignorar estas advertencias del compilador puede llevar a graves errores en otros casos (por ejemplo, si quisiéramos ignorar el último elemento, y cambiáramos la comparación `i < v.size()` por `i < v.size() - 1`, nuestro programa podría colgarse o fallar de manera imprevista). La solución práctica a esto es siempre que usemos `.size()` en una comparación, rodear a `v.size()` de `int(...)`, para indicarle al compilador que queremos que `v.size()` sea un `int` "normal". El ejemplo quedaría:

```
for (int i = 0; i < int(v.size()); i++)
    cout << v[i] << endl;
```

Esta regla práctica logra dos cosas: elimina la advertencia del compilador, y evita los posibles errores, fallos o problemas que podríamos tener al mezclar `int` con `unsigned int`.

Otra forma de recorrer un vector

Otra forma de recorrer un vector [2] es utilizando lo que se suele denominar foreach. Es una forma más conveniente de escribir la iteración que ya vimos.

En lugar de hacer por ejemplo:

```
for (int i=0; i<int(v.size()); i++)
    cout << v[i] << endl;
```

Podríamos equivalente utilizar el siguiente código:

```
for (int x : v)
    cout << x << endl;
```

Al escribir `for (int x : v)`, directamente `x` va tomando todos los valores `v[i]` del ejemplo anterior, es decir, "la iteración se hace sola", lo cual es mucho más cómodo cuando simplemente queremos procesar una vez cada elemento en orden. Cuando queremos trabajar con varios elementos a la vez, generalmente será más cómodo usar la "iteración clásica" que vimos antes (pues queremos tener a mano la variable `i` que indica la posición actual).

Matrices

Llamamos matriz a un rectángulo de valores (generalmente números). En programación, las matrices son muy muy comunes. Por ejemplo, una matriz podría ser:

```
1 2 5 8
9 3 1 5
```

```
30 5 3 4
```

Generalmente las matrices se usan para representar datos interesantes de la realidad: por ejemplo, los números podrían indicar la altura de un terreno en cada posición, teniendo así una especie de mapa físico del mismo.

Hemos visto que podemos usar `vector` para representar listas de valores, en particular, listas de números. Una manera posible de trabajar con matrices en computación es considerarlas como listas de filas : En efecto, si miramos la primera fila del ejemplo anterior, no es más que una lista de 4 números: `{1,2,5,8}`. Por lo tanto, esa primera fila podríamos guardarla en un `vector<int>` como los que ya hemos usado:

```
vector<int> fila1 = {1,2,5,8};
```

Y lo mismo ocurre con las demás filas: cada una es una lista de 4 elementos:

```
vector<int> fila2 = {9,3,1,5};
vector<int> fila3 = {30,5,3,4};
```

¿Cómo podríamos representar la matriz entera? Para describirla, basta dar la lista de sus 3 filas... Por lo tanto, la matriz será un `vector` (lista), y cada uno de los valores de dicha lista será a su vez, otra lista (de números: una fila). Nuestra matriz será entonces un `vector<vector<int>>`: Una lista (por eso el primer `vector`), tal que cada elemento de la lista es a su vez una lista (por eso cada elemento es `vector<int>`).

El código para guardar la matriz completa queda entonces:

```
vector<int> fila1 = {1,2,5,8};
vector<int> fila2 = {9,3,1,5};
vector<int> fila3 = {30,5,3,4};
vector<vector<int>> matriz = {fila1, fila2, fila3};
```

O incluso, podríamos haber escrito todas las listas directamente sin usar variables intermedias:

```
vector<vector<int>> matriz = {{1,2,5,8}, {9,3,1,5}, {30,5,3,4}};
```

Es común en estos casos usar enters y espacios para mayor claridad:

```
vector<vector<int>> matriz = { {1,2,5,8},
                               {9,3,1,5},
                               {30,5,3,4}
                             };
```

¿Cómo podríamos acceder, por ejemplo, al 8 que está guardado en la matriz? Recordando que nuestra matriz es una lista de filas , primero tenemos que ver en qué fila está: El 8 está en la primera fila, que contando desde 0 es la fila 0. Por lo tanto, `matriz[0]` (que es el primer elemento de la lista de filas) será la primera fila de la matriz: Un `vector<int>` que será `{1,2,5,8}`. De esta lista, el 8 es el elemento 3 (contando desde 0). Por lo tanto, `matriz[0][3] == 8`.

Podemos cambiar el 8 por un 100 haciendo `matriz[0][3] = 100`. Similarmente, `matriz[1][0] == 9` y `matriz[1][1] == 3`.

Hemos visto una manera de crear una matriz pequeña manualmente. ¿Cómo podríamos crear una matriz de `N` filas y `M` columnas? La siguiente sería una manera basada en las ideas que ya vimos:

```
vector<vector<int>> matriz(N);
for (int i = 0; i < N; i++)
    matriz[i].resize(M);
```

Primero creamos `matriz`, un vector de `N` elementos, pues la matriz tendrá `N` filas. Luego recorremos las `N` filas (por eso hacemos un `for`, con `i` variando de `0` hasta `N-1`), y a cada una de ellas le cambiamos el tamaño a `M`, con el método `resize` que ya vimos antes.

Una forma más práctica de lograr esto (aunque más avanzada de entender) es directamente usar la siguiente línea:

```
vector<vector<int>> matriz(N, vector<int>(M));
```

`vector<int>(M)` es una expresión que denota directamente un vector de tamaño `M`. Eso es lo que queremos que sean cada uno de los N elementos de la matriz (que es una lista de filas). Cuando queríamos crear un vector de `N` elementos, que todos tengan un cierto valor inicial, indicábamos dos valores entre paréntesis: primero la cantidad, y en segundo lugar el valor. En este caso, queremos que nuestra matriz tenga `N` filas, y que cada una de ellas sea una lista de tamaño M . Por eso indicamos

`vector<int>(M)` como segundo valor entre paréntesis luego de la coma: es el valor que queremos que tome cada elemento de la lista de filas.

Con esta técnica de usar un vector de vectores para guardar matrices, ya podemos trabajar con rectángulos de valores (sean letras, números, palabras, etc). Además, al aprender vector y este tipo de técnicas, por primera vez tenemos a nuestra disposición infinitos tipos de datos: Antes de conocer vector, solamente conocíamos 4 tipos: `int`, `string`, `char` y `bool`. Ahora que conocemos `vector`, no tenemos 5 tipos sino infinitos: Pues tenemos `vector<int>`, `vector<string>`, pero también `vector<vector<int>>`, y `vector<vector<vector<int>>>` (que sería una lista de matrices de enteros, o lo que es lo mismo, una lista de listas de listas de enteros...) y así podríamos seguir infinitamente. Hemos mostrado el ejemplo de las matrices (o las listas de listas) que son por mucho el caso más común.

Ejercicios

En estos ejercicios, suponer que primero se da una línea con un N y un M, que indican la cantidad de filas y columnas respectivamente de la matriz, y luego N líneas con M valores cada una, indicando el contenido de cada fila.

1. Dada una matriz de números, imprimir las sumas de sus filas y sus columnas [http://juez.oia.unsam.edu.ar/#/task/suma_filas/statement] (N + M valores en total).
2. Dada una matriz de números, imprimir la matriz traspuesta [http://juez.oia.unsam.edu.ar/#/task/matriz_traspuesta/statement], es decir, aquella que tiene en la fila i, columna j, lo que la original tenía en la fila j, columna i.
3. Dado un rectángulo de números enteros (podrían ser negativos), ¿Cuál es la máxima suma posible de un subrectángulo de mayor suma [http://juez.oia.unsam.edu.ar/#/task/max_sum_rect/statement]? Un subrectángulo se forma tomando los elementos de un conjunto de filas y columnas todas contiguas , sin dejar "agujeros". Notar que el subrectángulo no puede ser vacío.
4. Dado un rectángulo de letras y una lista de palabras, ¿Cuáles de las palabras aparecen en esta sopa de letras [http://juez.oia.unsam.edu.ar/#/task/sopa_de_letras/statement]? Las palabras pueden aparecer en horizontal (tanto hacia la derecha como hacia la izquierda) o en vertical (tanto hacia arriba como hacia abajo). No se cuentan las posibles apariciones en diagonal.

1) , 2)
En C++11

Funciones

Motivación

Veremos en esta lección el concepto de función. Empezaremos tratando de entender cuál es el problema o dificultad que las funciones nos pueden ayudar a aliviar, es decir, empezaremos dándonos una mínima idea de la respuesta a la pregunta de: "Funciones, ¿para qué?".

Si bien es posible resolver cualquier problema sin usar funciones, hay 3 excelentes motivos para utilizar funciones:

1. Para ordenar mejor el código fuente, y hacerlo más fácil de leer y entender .
2. Para facilitar un enfoque de programación top-down , en el cuál vamos descomponiendo el problema en partes, y luego nos concentramos en cómo resolver cada parte.
3. Para evitar tener código repetido , que es algo que debemos evitar a toda costa si queremos programar mejor y con menos chances de errores.

Ordenar el código fuente

Supongamos que tenemos el siguiente código (que iría dentro de `main` en un programa):

```
int a,b;
cout << "Ingrese el rango de numeros a explorar" << endl;
cin >> a >> b;
int sumaPrimosAlCuadrado = 0;
for (int i=a; i<= b; i++)
if (i >= 2)
{
    bool esPrimo = true;
    for (int d = 2; d*d <= i; d++)
    if (i % d == 0)
        esPrimo = false;
    if (esPrimo)
        sumaPrimosAlCuadrado += i*i;
}
cout << "La primera suma pedida es:" << sumaPrimosAlCuadrado << endl;
int sumaMultiplosEspeciales = 0;
for (int i=a; i<= b; i++)
{
    if ((i % 3 == 0 || i % 10 == 0) && i % 30 != 0)
        sumaMultiplosEspeciales += i;
}
cout << "La segunda suma pedida es:" << sumaMultiplosEspeciales << endl;
```

Es muy difícil de entender qué hace este código a simple vista, y el motivo principal es que hace muchas cosas mezcladas todas a la vez . Veremos que una función permite separar una porción de programa definida, y luego usarla cuando la necesitemos. Es bueno comparar el tener que leer y entender el programa anterior, con leer algo como lo siguiente:

```
int a,b;
leerRangoAExplorar(a,b);
mostrarResultados(sumaDePrimosAlCuadradoEnRango(a,b), sumaDeMultiplosEspecialesEnRango(a,b));
```

Aquí, `leerRangoAExplorar(a,b);` corresponderá al siguiente fragmento del código anterior:

```
cout << "Ingrese el rango de numeros a explorar" << endl;
cin >> a >> b;
```

Mientras que `sumaDePrimosAlCuadradoEnRango(a,b)` representa:

```
int sumaPrimosAlCuadrado = 0;
for (int i=a; i<= b; i++)
if (i >= 2)
{
    bool esPrimo = true;
    for (int d = 2; d*d <= i; d++)
    if (i % d == 0)
        esPrimo = false;
    if (esPrimo)
        sumaPrimosAlCuadrado += i*i;
}
```

`sumaDeMultiplosEspecialesEnRango(a,b)` corresponde al:

```
int sumaMultiplosEspeciales = 0;
for (int i=a; i<= b; i++)
{
    if ((i % 3 == 0 || i % 10 == 0) && i % 30 != 0)
        sumaMultiplosEspeciales += i;
}
```

Y el `mostrarResultados` corresponde al:

```
cout << "La primera suma pedida es:" << sumaPrimosAlCuadrado << endl;
cout << "La segunda suma pedida es:" << sumaMultiplosEspeciales << endl;
```

Separando estas operaciones que son independientes entre sí, y dándoles un nombre claro, el código es mucho más fácil de entender, ya que podemos analizar cada parte independientemente por un lado, y por otro lado, su combinación para formar el programa completo. `leerRangoAExplorar` , `sumaDePrimosAlCuadradoEnRango`, `sumaDeMultiplosEspecialesEnRango` y `mostrarResultados` son ejemplos de funciones

Facilitar un enfoque top-down

Esta ventaja está estrechamente relacionada con la anterior. Supongamos que nos dieran la siguiente consigna:

"Crear un programa que lea dos números `a` y `b`, que indican un rango de números (inclusive), y calcule y muestre en la pantalla dos valores: La suma de los cuadrados de todos los números primos entre a y b, y además, la suma de todos los números entre a y b que son múltiplos de 3 y de 10, pero no de 30."

El programa anterior (el que tenía todo junto) sería un ejemplo de resolución de esta tarea. Ahora bien, escribir y pensar todo ese código junto en un solo paso, a partir de esta descripción, es muy complicado. Pero podríamos planear descomponer este problema en tareas más chicas, y escribir nuestra solución suponiendo que tenemos resueltas esas tareas .

Por ejemplo, en este caso podríamos identificar que tenemos que hacer 4 cosas:

1. Leer a y b de la entrada
2. Calcular la suma de los cuadrados de los primos pedidos
3. Calcular la suma de los números que son "múltiplos especiales" (explicado en la consigna)
4. Escribir los resultados a la salida

Identificamos entonces que para 1), necesitaremos dos variables `a`,`b` en las cuales guardaremos los números leídos. Podemos denominar `leerRangoAExplorar(a,b);` al proceso de leer esas variables: Sin preocuparnos por ahora sobre cómo lo haremos . Eso lo dejamos para después.

Para realizar los cálculos de 2) y 3), necesitaremos los valores a y b , que obtuvimos en 1). Llamaremos (de vuelta, sin preocuparnos todavía por cómo lograremos hacer los cálculos) `sumaDePrimosAlCuadradoEnRango(a,b)` al resultado de hacer los cálculos que indica el paso 2), y llamaremos `sumaDeMultiplosEspecialesEnRango(a,b)` al resultado del paso 3).

Finalmente, llamaremos `mostrarResultados` al proceso del paso 4), que usa los resultados obtenidos en los pasos 2 y 3 . Con estas ideas, podemos planear un `esqueleto` de nuestro programa, que quedaría más o menos así:

```
int a,b;
leerRangoAExplorar(a,b);
mostrarResultados(sumaDePrimosAlCuadradoEnRango(a,b), sumaDeMultiplosEspecialesEnRango(a,b));
```

Ahora que tenemos el esqueleto del programa listo, podemos concentrarnos en detalle en ver cómo resolvemos cada una de esas 4 partes. Usar funciones nos permitirá escribir este esqueleto directamente en el main, y luego aclarar en funciones separadas cómo realizar cada una de las 4 partes.

Evitar código repetido

Llamamos código repetido a un conjunto de operaciones completamente análogas, que aparece repetido en el programa más de una vez. Esto es algo que queremos evitar a toda costa porque es muy propenso a errores.

Una ejemplo podría ser, si tenemos un programa que tiene que analizar si un número dado es primo, luego realizar un montón de cálculos y operaciones, y al final del programa antes de terminar debe analizar si otro número es primo. Si bien se analizan dos números distintos, la serie de operaciones que hacemos es la misma en los dos casos, y solamente cambia el número (o la variable donde está guardado). Esto nos llevaría a tener dos fragmentos del programa (por ejemplo, dos `for` con cálculos) esencialmente iguales, uno para cada número. Esto lleva fácilmente a errores, porque si en algún momento queremos cambiar algo de este código (o corregir un error encontrado) tenemos que cambiarlo en los dos lugares, y es muy fácil olvidarse de cambiar uno, o equivocarse en uno de los cambios.

Mediante funciones, podremos escribir el código para determinar si un número es primo una sola vez, y luego reutilizar ese código todas las veces que queramos, sin necesidad de escribir todo el código de nuevo.

La idea de función

Una función en C++[1] será un fragmento de programa bien definido, que puede utilizarse como parte de otros programas o funciones.

La sintaxis para definir una función es la siguiente:

```
tipo_de_la_respuesta nombreDeLaFuncion(lista_de_parametros)
{
    // Cuerpo de la funcion, con las instrucciones correspondientes a la misma
    return resultado; // Con return se termina la función y se indica el resultado final
}
```

Esto debe escribirse antes del main, y no adentro. Eso es porque main es una función como las demás, y no se permite en C++ escribir una función dentro de otra. La particularidad que tiene la función `main` es que es allí donde comienza a ejecutarse el programa: la computadora comienza a leer instrucciones por el `main`.

Una función, al ser un fragmento de programa, puede realizar cálculos y tareas, y puede eventualmente producir un resultado. Ese resultado se llama el valor de retorno de la función, y se dice que la función lo devuelve al usar la instrucción return. En el código anterior, la parte de `tipo_de_la_respuesta` se usa para indicar el tipo que tendrá el resultado de la función.

Veamos un ejemplo de función con `lista_de_parametros` vacía, lo cual es válido en C++:

```
int leerUnNumeroYElevarloAlCuadrado()
{
    int x;
    cin >> x;
    return x*x;
}
```

Este código corresponde a un fragmento de programa, que lee con `cin` un número `x`, y devuelve con `return` el valor `x*x`, es decir el número al cuadrado: Si se lee 3, se devuelve 9, si se lee 5 se devuelve 25, si se lee -2 se devuelve 4, etc. Cuando se ejecuta una instrucción `return`, se devuelve el valor indicado y la función termina inmediatamente, sin importar que pudiera haber más instrucciones además del `return`.

Como lo que devuelve es un entero, hemos colocado `int` justo al comienzo, antes del nombre de la función. Los paréntesis luego del nombre de la función son obligatorios siempre que escribamos una función, incluso cuando dentro de ellos no pongamos nada, como en el ejemplo.

Si en nuestro programa colocamos este código antes del `main`, podremos utilizar esta función en el programa principal: para ello basta con escribir la instrucción `leerUnNumeroYElevarloAlCuadrado();`, y eso automáticamente ejecutará todo el código correspondiente a esa función. Nuevamente, al utilizar (también denominado llamar o invocar) una función, los paréntesis son obligatorios.

Veamos a continuación un ejemplo de programa completo que usa esa función:

```
#include <iostream>

using namespace std;

int leerUnNumeroYElevarloAlCuadrado()
{
    int x;
    cin >> x;
    return x*x;
}

int main()
{
    int a = leerUnNumeroYElevarloAlCuadrado();
    int b = leerUnNumeroYElevarloAlCuadrado();
    int c = leerUnNumeroYElevarloAlCuadrado();
    cout << "El gran total es: " << a+b+c << endl;
}
```

Este programa lee tres números, y al final muestra el gran total: La suma de los cuadrados de los números leídos. Notar que la operación de leer un número con `cin` y elevarlo al cuadrado se realiza 3 veces, porque 3 veces escribimos `leerUnNumeroYElevarloAlCuadrado()` en el programa principal: Pero una sola vez tuvimos que escribir las instrucciones completas para realizar esa tarea, al definir la función antes del main. Luego podemos usarla libremente como si fuera una operación más.

De este programa podemos destacar que cuando tenemos una función que devuelve un resultado, al llamar a la función podemos directamente escribir la llamada y usar el resultado en una expresión, como si fuera directamente el valor . Es decir, cuando ponemos `leerUnNumeroYElevarloAlCuadrado()`, podemos pensar para nuestro razonamiento que eso se va a reemplazar directamente por el resultado final de los cálculos de la función.

Así, si al ejecutar el programa anterior ingresáramos los valores 3, 1, y 10, sería como si en el main se ejecutase lo siguiente:

```
int main()
{
    int a = 9;
    int b = 1;
    int c = 100;
    cout << "El gran total es: " << a+b+c << endl;
}
```

Obteniendo el resultado 110. A modo de ejemplo, damos una versión distinta del programa que calcula la suma del triple de cada número al cuadrado, para que quede claro que las llamadas a funciones se pueden usar en el medio de expresiones más complejas si así nos conviene (damos solo el main, pues la función es igual que antes):

```
int main()
{
    int a = 3 * leerUnNumeroYElevarloAlCuadrado();
    int b = 3 * leerUnNumeroYElevarloAlCuadrado();
    int c = 3 * leerUnNumeroYElevarloAlCuadrado();
    cout << "El gran total es: " << a+b+c << endl;
}
```

Incluso sería válido (aunque es más difícil de leer) escribir el programa original con todas las llamadas en la misma línea:

```
int main()
{
    cout << "El gran total es: " << leerUnNumeroYElevarloAlCuadrado() +
                                    leerUnNumeroYElevarloAlCuadrado() +
                                    leerUnNumeroYElevarloAlCuadrado() << endl;
}
```

Este último ejemplo muestra una característica importante de las funciones: Cada vez que se escribe una llamada a función en el código, se ejecutan las instrucciones de la función : Si se escribe 3 veces, se ejecutan tres veces. En nuestro ejemplo, eso quiere decir que el fragmento anterior no lee un número y luego lo "triplica" al sumarlo consigo mismo 3 veces, sino que lee 3 números distintos, porque lee uno nuevo en cada llamada.

Si queremos reutilizar el valor que se obtuvo al ejecutar una función sin volver a ejecutarla, conviene guardar el resultado en una variable , como hicimos en los primeros ejemplos, donde teníamos algo como `int a = leerUnNumeroYElevarloAlCuadrado();`.

Parámetros

No siempre queremos que una función haga exactamente lo mismo cada vez que se usa. A veces, queremos que haga casi lo mismo, pero cambiando algún dato entre usos. Por ejemplo, podríamos querer una función que eleve un número al cuadrado, es decir, que permita calcular `x*x` si ya tenemos un entero `x`. Así, cuando usamos la función con 3, queremos que devuelva `3*3 == 9`, y cuando la usamos con -4 queremos que devuelva `(-4)*(-4) == 16`.

En el ejemplo anterior la función no hace siempre lo mismo, porque a veces hace 3*3 y a veces (-4)*(-4), pero más allá del número que vamos a elevar, las operaciones que hace la función son siempre las mismas, y solo cambia este dato inicial. A ese dato que cambia , lo llamamos en programación un parámetro de la función. Una función puede tener 1 o más parámetros, o incluso cero: Las funciones que vimos antes tenían cero parámetros. La función de elevar al cuadrado tendría un único parámetro: El número entero que vamos a querer elevar.

Los parámetros que tendrá una función se indican al escribir su código, entre paréntesis, luego del nombre: se debe dar una lista separada por comas, en la cual se indique el tipo y el nombre de cada parámetro. Es por eso que en los ejemplos anteriores necesitamos siempre un par de paréntesis `()` vacíos: las funciones que estábamos utilizando tenían cero parámetros.

Veamos a continuación el ejemplo de la función para elevar un entero al cuadrado:

```
int alCuadrado(int x)
{
    return x*x;
}
```

Esta función es muy simple, pues lo único que hace es devolver `x*x`. Notemos que `x` es el único parámetro de esta función: entre paréntesis hemos indicado su tipo, `int`, y le hemos dado el nombre `x`, lo cual permite usar este parámetro en el código de la función .

Cuando una función tiene parámetros, al usarla hay que aclarar qué valores tomar para esos parámetros. En nuestro ejemplo de función que eleva al cuadrado un número, no podemos en el programa simplemente usar `alCuadrado()` como en los ejemplos anteriores: ¿Cuál sería el resultado si hiciéramos eso? ¿Qué número se estaría elevando al cuadrado? La función no puede saberlo si no se lo indicamos. Por este motivo, cuando se usa una función con parámetros hay que indicar entre paréntesis los valores que queremos usar para esos parámetros, en el mismo orden en que se dieron al escribir el código de la función.

Veamos un ejemplo completo de programa que usa la función `alCuadrado`:

```
#include <iostream>

using namespace std;

int alCuadrado(int x)
{
    return x*x;
}

int main()
{
    int num;
    cout << "Ingrese un numero" << endl;
    cin >> num;
    cout << "El numero " << num << " al cuadrado es " << alCuadrado(num) << endl;
    cout << "De paso, te cuento que 7 al cuadrado es " << alCuadrado(2+5) << endl;
    return 0;
}
```

El ejemplo muestra que la llamada a una función (es decir, usarla) cuenta como una "operación", igual que la suma o la resta, y puede usarse en expresiones más complicadas (por ejemplo como parte de una cuenta y operaciones aritméticas).

Funciones con varios parámetros

Una función puede necesitar varios datos para realizar su tarea. De ser así, se trabaja de idéntica manera pero separando los parámetros con comas, y dándolos siempre en el mismo orden. Por ejemplo, a continuación se muestra un ejemplo de código que usa una función que pega dos palabras usando un guión entre ellas:

```
#include <iostream>

using namespace std;

string pegarConGuion(string palabra1, string palabra2)
{
    return palabra1 + "-" + palabra2;
}

int main()
{
    string a,b;
    cout << "Ingrese dos palabras" << endl;
    cin >> a >> b;
    cout << pegarConGuion(a,b) << endl;
    cout << pegarConGuion("super", "sonico") << endl;
    return 0;
}
```

Alentamos al lector a que lea este código y prediga qué es lo que va a mostrar por pantalla, y luego verifique que efectivamente así sea.

Procedimientos

Veremos ahora ejemplos de funciones que no devuelven nada , lo que a veces (sobre todo en otros lenguajes) se denomina procedimiento o subrutina (En C++, lo usual es llamar a todos función, aunque no devuelvan nada).

¿Por qué podríamos querer que una función no devuelva nada? Porque podría importarnos solamente lo que la función hace , es decir, las instrucciones que ejecuta, sin necesidad de que nos devuelva un valor final. Un ejemplo sencillo de esto podría ser alguna función que escriba en pantalla:

```
void mostrarConEspacios(string palabra)
{
    for (int i=0;i<int(palabra.size());i++)
    {
        if (i > 0)
            cout << " ";
        cout << palabra[i];
    }
}
```

Esta función recibe una palabra, y la escribe en pantalla pero con un espacio insertado entre cada par de letras consecutivas. En este caso, no hay nada que devolver: Al que llama no le importa obtener ningún resultado, sino simplemente lograr que se escriba a pantalla lo que queremos. Una vez que la función hace lo que queríamos (en este caso, imprimir a pantalla), el que la usa no espera ningún resultado adicional. Es por eso que por ejemplo no pusimos return en esta función, pues no hay nada que devolver.

Cuando una función no devuelve nada, se indica `void` como su tipo de retorno. `void` significa "vacío" en inglés, y es un tipo muy especial que se usa en el valor de retorno de las funciones para indicar que no devuelven nada.

En una función que devuelve `void` (que es lo mismo que decir que no devuelve nada), no es posible utilizar `return x;`, pues no se puede devolver nada. Sin embargo, está permitido utilizar `return;` a secas, sin indicar ningún valor de retorno: Esto tiene el efecto de terminar inmediatamente la ejecución de la función , en el momento en que se ejecuta el `return`.

Pasaje por copia y por referencia

Supongamos que escribimos la siguiente función, con la idea de que aumente en uno la variable indicada:

```
void incrementar(int x)
{
    x = x + 1;
}
```

Con lo que vimos hasta ahora, podría parecer que esta función hará lo que queremos. Podemos intentar utilizarla en un programa:

```
#include <iostream>

using namespace std;

void incrementar(int x)
{
    x = x + 1;
}

int main()
{
    int x = 20;
    cout << x << endl;
    incrementar(x);
    cout << x << endl;
    incrementar(x);
    cout << x << endl;
    return 0;
}
```

Querríamos que este programa muestre 20, 21 y 22, pues utiliza nuestra función para ir aumentando la variable. Sin embargo si lo ejecutamos, veremos por pantalla 20, 20 y 20: No se ha incrementado nada. ¿Por qué resulta ser así?

Para ayudar a entender esto agregaremos al programa un par de instrucciones con `cout` dentro de la función, para ver si se incrementa o no.

```
#include <iostream>

using namespace std;

void incrementar(int x)
{
    cout << "Recibo x con " << x << endl;
    x = x + 1;
    cout << "Al terminar x tiene " << x << endl;
}

int main()
{
    int x = 20;
    cout << x << endl;
    incrementar(x);
    cout << x << endl;
    incrementar(x);
    cout << x << endl;
    return 0;
}
```

Este programa muestra por pantalla:

```
20
Recibo x con 20
Al terminar x tiene 21
20
Recibo x con 20
Al terminar x tiene 21
20
```

Podemos ver que dentro de la función se está produciendo el incremento, como queremos, pues se recibe 20 y luego se tiene 21. Pero este cambio no se observa fuera de la función : La llamada a la función parece no estar teniendo ningún efecto sobre el `x` del main.

El motivo por el que esto ocurre es que cuando se ejecuta una función, al comenzar se hace una copia de los parámetros: En la primera llamada a incrementar, El `x` del main vale 20. Pero la función no trabaja con el x del main : trabaja todo el tiempo con una copia de ese `x`. Así, la función tiene su propia copia del parámetro `x`, que al comenzar toma el valor 20 que tenía la variable con la cual fue llamada.

Lo que observamos es que la función incrementa esta copia, y dentro de la función siempre estamos usando la copia, pero al terminar la función esta copia deja de existir y la variable original del `main` queda inalterada sin cambios. Si bien en muchos casos esto es exactamente lo que queremos, para evitar que una función "nos cambie accidentalmente" nuestras variables, en este ejemplo queremos intencionalmente cambiar una variable del main. Es decir, en este ejemplo queremos intencionalmente que no se haga ninguna copia para la función, sino que se trabaje directamente con el dato original , para que todos los cambios que haga la función se hagan sobre el original.

La manera de hacer esto en C++ es agregando un ampersand `&` justo antes del nombre del parámetro en el momento en que escribimos el código de la función: dicho ampersand indica a C++ que no queremos trabajar con una copia, sino con la variable original directamente.

Nuestro ejemplo quedaría entonces (solamente le agregamos un `&` en el parámetro `x`):

```
#include <iostream>

using namespace std;

void incrementar(int &x)
{
    cout << "Recibo x con " << x << endl;
    x = x + 1;
    cout << "Al terminar x tiene " << x << endl;
}

int main()
{
    int x = 20;
    cout << x << endl;
    incrementar(x);
    cout << x << endl;
    incrementar(x);
    cout << x << endl;
    return 0;
}
```

Que produce el resultado esperado:

```
20
Recibo x con 20
Al terminar x tiene 21
21
Recibo x con 21
Al terminar x tiene 22
22
```

Notar que el ampersand se debe agregar en la declaración (donde escribimos el código) de la función, y no en la invocación (donde la usamos / llamamos).

A la forma normal de pasar los parámetros, que ocurre cuando no especificamos ningún ampersand, se la llama pasar por copia o pasar por valor, pues en la función directamente se copia el valor de los parámetros y se trabaja siempre con las copias, sin cambiar los originales.

En cambio, cuando se usa el ampersand, se dice que ese parámetro se pasa por referencia o por variable, pues en todo momento se hace referencia a la variable original, y nunca existe una copia diferente para la función. Cualquier cambio que haga la función, impactará en la variable original.

La mayoría de la veces pasaremos los parámetros por valor, y solo cuando tengamos alguna razón específica para hacerlo los pasaremos por referencia. Uno de los ejemplos más comunes es el que acabamos de ver, donde queremos modificar una variable del main como parte de las operaciones de la función.

Procedimientos + Pasaje por referencia

El uso de procedimientos es bastante usual cuando se utilizan parámetros pasados por referencia. Esto es porque uno de los usos principales de pasar un parámetro por referencia es para que la función lo pueda modificar, y entonces, si queremos que la función modifique el parámetro que le pasamos, es posible (pero no necesariamente cierto) que no necesitemos ningún resultado adicional, sino que solamente nos importe esta modificación.

Un ejemplo podría ser una función que recibe un `vector<int>` y duplica todos sus elementos:

```
#include <iostream>
#include <vector>

using namespace std;

void duplicarElementos(vector<int> &v)
{
    for (int i=0;i<int(v.size());i++)
        v[i] *= 2;
}

int main()
{
    vector<int> w = {1,2,3,8};
    duplicarElementos(w);
    for (int x : w)
        cout << x << endl;
    return 0;
}
```

Que muestra por pantalla:

```
2
4
6
16
```

Notar que si cambiamos la línea `void duplicarElementos(vector<int> &v)` por `void duplicarElementos(vector<int> v)` (quitando el `&`), el programa ya no hará lo qué queremos. ¿Qué mostrará? ¿Y por qué es así?

Variables locales y globales

Todas las variables que hemos utilizado hasta ahora han sido siempre variables locales. Esto significa que fueron definidas dentro de una función (`main` es también una función: por lo tanto, las variables que hemos definido en `main` son también variables locales, de la función main).

Una variable local solamente puede utilizarse dentro de la función en que fue definida, y no desde otras funciones (para eso existe la idea de usar los parámetros, para pasar información útil a una función).

Existe otro tipo de variables que pueden accederse desde cualquier lugar del programa. Estas variables se denominan variables globales. Basta declararlas en el programa directamente fuera de cualquier función para obtener una variable global. Desde ese punto del programa en adelante, se podrá utilizar esa variable, que está "compartida" entre todas las funciones del programa.

En general, utilizar demasiadas variables globales en lugar de parámetros es considerado una mala práctica en C++, y hacerlo puede llevar fácilmente a tener programas difíciles de leer. Se utilizan principalmente en competencias de programación, como forma práctica de tener accesibles datos importantes que son utilizados a lo largo de todo el programa, y evitar así tener que pasar los mismos parámetros todo el tiempo entre funciones. Existen otras técnicas para lograr esto mismo, pero son más avanzadas (por ejemplo, usar struct/class y métodos, y/o enfoques de programación orientada a objetos).

Además, existe un peligro adicional a tener en cuenta las variables globales, que es la posibilidad de ocultar una variable global con una variable local. Esto ocurre cuando tenemos una variable local con el mismo nombre que una variable global: Estas dos serán variables distintas, pero al tener el mismo nombre, no es posible utilizar ambas a la vez. Dentro de la función, solamente será posible utilizar la variable local. Por eso se dice que la variable local oculta a la global.

Por ejemplo, el siguiente código:

```
int mivar = 32;
```

```
int foo()
{
   int mivar = 12;
   mivar++;
   return 2*mivar;
}

int main()
{
    cout << foo() << endl;
    cout << mivar << endl;
    return 0;
}
```

Mostrará 26 y 32. El valor de la variable global `mivar` nunca es modificado, ya que dentro de `foo` se trabaja con la variable local del mismo nombre, que oculta a la variable global correspondiente. Por estos motivos, en general es muy mala idea usar el mismo nombre para una variable global y una local, pues podemos tener problemas y usar la variable que no queríamos sin darnos cuenta.

En inglés esto se denomina shadowSi se activa la opción `-Wshadow`, el compilador nos advierte si ocultamos una variable de esta forma.

Observaciones adicionales

- En las funciones se puede llamar (utilizar) otras funciones: esto está totalmente permitido:

```
int multiplicar(int a, int b)
{
    return a*b;
}

int elevar(int base, int exponente)
{
    int resultado = 1;
    for (int i=0;i<exponente;i++)
        resultado = multiplicar(resultado, base);
    return resultado;
}
```

- Variables con el mismo nombre pero definidas en funciones distintas, representan variables diferentes (como el `x` del main y el `x` de la función en el ejemplo anterior de `incrementar`)

Ejemplos de implementación de funciones

- Nuestro primer ejemplo es la función `main`, que ya venimos usando en todos nuestros programas: Es una función que devuelve un `int`, que usa la computadora para saber si hubo errores. Por convención, se debe devolver cero si todo salió bien, y por eso es buena costumbre terminar todos los programas con `return 0`. La función `main` es importante porque tiene la característica especial de que allí comienzan a ejecutarse todos nuestros programas, aunque tengan otras funciones.
- Como ejemplo de pensamiento top-down, supongamos que debemos realizar un programa que lea una secuencia de números, y luego calcule y muestre por pantalla la suma, el máximo y el mínimo de todos estos números. Podemos programar primero que anda el main de la siguiente manera:

  ```
  int main()
  {
      vector<int> v;
      v = leerNumeros();
      imprimirResultados(suma(v), maximo(v), minimo(v));
      return 0;
  }
  ```

  Donde nos hemos ordenado y hemos logrado descomponer el problema entero en tareas más pequeñas. Luego podríamos agregarle las funciones que faltan al programa, para completarlo, dejando inalterado el mismo main que ya escribimos:

```
vector<int> leerNumeros()
{
    // instrucciones...
}

int suma(vector<int> v)
{
    // instrucciones...
}

int maximo(vector<int> v)
{
    // instrucciones...
}

int minimo(vector<int> v)
{
    // instrucciones...
}

void imprimirResultados(int laSuma,int elMaximo, int elMinimo)
{
    // instrucciones...
}
```

- El siguiente es un ejemplo con funciones que calculan áreas de figuras geométricas, que muestra como podemos reutilizar ciertas funciones dentro de otras:

```
int areaParalelogramo(int base, int altura)
{
    return base * altura;
}
int areaCuadrado(int lado)
{
    return areaParalelogramo(lado, lado);
}
int areaTriangulo(int base, int altura) // Trabaja con enteros: Redondea hacia abajo
{
    return areaParalelogramo(base, altura) / 2;
}
```

- El siguiente es un ejemplo de función que recibe dos variables enteras, e intercambia sus valores. ¡Notar el uso del ampersand!

```
void intercambiar(int &variable1, int &variable2)
{
    int auxiliar = variable1; // Es necesario un auxiliar: ¿Por qué?
    variable1 = variable2;
    variable2 = auxiliar;
}
```

  Similarmente, el siguiente ejemplo permite "rotar" los valores de tres variables dadas: Es decir, transforma [a,b,c] en [b,c,a]:

```
void rotar3(int &a, int &b, int &c)
{
   int auxiliar = a; // Nuevamente, ¿Por qué es necesario el auxiliar?
   a = b;
   b = c;
   c = auxiliar;
}
```

Algunas funciones predefinidas

En C++, existen algunas funciones predefinidas que ya existen, y conocerlas puede simplificarnos la tarea de programar ya que nos ahorramos tener que escribirlas nosotros mismos. Mencionamos algunas a continuación:

- `max`: Dados dos números, devuelve el máximo. Por ejemplo `max(2,9) == 9` y `max(5,3) == 5`. Similarmente tenemos `min` para el mínimo.

- `swap`: Intercambia los valores de las dos variables que se le indica. Por ejemplo si `x` tiene un `3`, y `q[i]` tiene un `8`, luego de hacer `swap(x,q[i])` quedará `q[i]` con un 3 y `x` con un 8.
- `abs`: Devuelve el valor absoluto (módulo) de un entero. Por ejemplo `abs(-3) == 3`, `abs(0) == 0` y `abs(15) == 15`.

Todas estas funciones requieren utilizar `#include <algorithm>` para tenerlas disponibles.

Algunos errores comunes

- Pasar a la función una cantidad de parámetros diferente de las que la función necesita, o con el tipo incorrecto. Por ejemplo si tenemos la función

  ```
  int mayor(int num1, int num2)
  {
      if (num1 > num2)
          return num1;
      else
          return num2;
  }
  ```

  serían incorrectas las siguientes llamadas:

  ```
  mayor(k,m,n) // Pasa 3 parámetros, pero la función toma solamente 2
  mayor(23, "miliwatt") // Pasa 2 parámetros, pero el segundo es una cadena y debería ser un int
  ```

- Diseñar una función con la idea de que modifique uno de sus parámetros, pero trabajar con una copia por no utilizar el ampersand `&`.
- Intentar utilizar un parámetro (con su nombre) fuera de una función: Los parámetros solamente están definidos dentro de la función, y no tiene sentido utilizarlos fuera de ella (son variables locales).
- Utilizar una función que todavía no se definió. Se debe programar el código de una función, antes de utilizarla.

Ejercicios

Recomendamos revisar los ejercicios pasados del curso, y volver a programarlos aprovechando la idea de funciones: ¿En cuáles es apropiado utilizar funciones para estructurar mejor el programa?

Por ejemplo, en los ejercicios en los que se hablaba de números primos, se podría escribir una función que toma un `int N`, y devuelve un `bool` indicando si es primo. O por ejemplo, escribir una función `sumaDeDivisores` puede ser útil para escribir de forma más fácil y clara programas que buscan números perfectos.

Otros ejercicios:

- Escribir una función `string escribirEnBase(int numero, int base)`, que tome un número y una base (Entre 2 y 16 inclusive) y devuelva una cadena con la escritura de ese número en la base indicada.
- Escribir una función `int leerNumeroEnBase(string escritura, int base)`, que tome la escritura de un cierto número en la base indicada (Entre 2 y 16 inclusive) y devuelva el número en cuestión.

Puede ver aquí cómo realizar cambios de base.

Los #define

En C++, además de las funciones que ya vimos, existe un mecanismo relacionado para evitar repetir código, y es la directiva `#define`.

Mediante un `#define`, es posible definir lo que se denomina una macroque es una regla para reemplazar textualmente un fragmento de código en el programa . Por ejemplo, supongamos que colocamos en cualquier línea de un programa, el siguiente `#define`:

```
#define declaraEInicializaEnUnValor(nombreVariable, valor) int nombreVariable = valor
```

Luego de haber escrito este `#define`, si en cualquier lugar del programa aparece un `declaraEInicializaEnUnValor(x, 33)`, se reemplazará textualmente en ese mismo lugar por un `int x = 33`. Por ejemplo,el siguiente sería un programa válido:

```
#include <iostream>

using namespace std;

#define declaraEInicializaEnUnValor(nombreVariable, valor) int nombreVariable = valor

int main()
{
    declaraEInicializaEnUnValor(x, 40);
    declaraEInicializaEnUnValor(y, 100);
    cout << x + y << endl;
    return 0;
}
```

Que muestra por pantalla 140. Notar que luego de los reemplazos, el programa es absolutamente equivalente a si se hubiera escrito directamente:

```
#include <iostream>

using namespace std;

int main()
{
    int x = 40;
    int y = 100;
    cout << x + y << endl;
    return 0;
}
```

En general, siempre que podamos conviene utilizar funciones en lugar de `#defines`, y dejaremos los `#defines` únicamente para los casos en que no podamos hacer lo mismo con una función. Por ejemplo, en el ejemplo anterior creamos una macro que permite declarar una variable y empezar a usarla directamente, que es algo que no podríamos haber hecho con una función. A diferencia de una función, una macro hace un reemplazo totalmente mecánico de los valores indicados entre paréntesis, exactamente igual que si se hiciera copy paste mecánicamente en el código.

Algunos ejemplos de macros muy útiles

Especialmente en competencias de programación, es muy común tener un fragmento de código como el siguiente:

```
for (int numero = 0; numero < valorMaximo; numero++)
    // instrucciones
```

Donde recorremos todos los números desde `0` hasta `valorMaximo - 1` inclusive. Notar que al escribir ese fragmento tenemos que escribir la variable `numero` tres veces , lo cual aumenta las chances de equivocarnos (especialmente, si hacemos copy + paste de otro for similar, ya que entonces es muy fácil olvidarnos de cambiar alguna de las 3 apariciones).

Otra variante muy similar sería cuando queremos recorrer todos los índices de un cierto vector:

```
for (int var = 0; var < int(vector.size()); var++)
    // instrucciones
```

En este caso se agrega la conversión con `int()`, que debe usarse al comparar con `.size()`.

Para programar más fácilmente y con menor chances de errores este tipo de for comunes, podemos utilizar un `#define` para definir una forma compacta de indicar los elementos importantes que cambian de caso en caso, y que el resto se reemplace mecánicamente siempre igual:

```
#define forn(i,n) for(int i=0;i<int(n);i++)
```

En este caso, indicamos en el `#define` el nombre de la variable que vamos a declarar, y el valor tope hasta el cual vamos a iterar (iteraremos desde `0` hasta `n-1` inclusive). Esto es lo único que cambia en estos ejemplos, y el resto es siempre igual.

De esta forma, una vez que tenemos este #define el primer for que mostramos nos quedaría simplemente:

```
forn(numero, valorMaximo)
    // instrucciones
```

Y el segundo quedaría:

```
forn(var, vector.size())
    // instrucciones
```

Vemos que ahora solamente hace falta especificar una vez el nombre de la variable, y todo lo demás es copiado automáticamente en forma mecánica por el `#define`.

Se puede consultar aquí otros ejemplos de macros más avanzadas, muy útiles para programación competitiva, además del `forn` ya mostrado.

Por qué es mejor usar funciones

Hemos mencionado que siempre que podamos hacer algo con funciones en lugar de `#define`, es conveniente hacerlo con funciones. Esto es porque las funciones son "más seguras" de utilizar. Veamos un ejemplo para ver por qué esto es así.

Supongamos que queremos tener un fragmento de código para elevar un número al cuadrado. La forma más simple de hacerlo es multiplicar al número con sí mismo, ya que multiplicar es una operación atómica disponible. Podemos entonces pensar en hacer un `#define` para ello:

```
#define cuadrado(x) x*x
```

De esta forma, cuando escribamos `cuadrado(2)`, por ejemplo, se reemplaza por `2*2`, que es el número al cuadrado como queremos. Sin embargo, el `#define` que acabamos de definir es muy peligroso de utilizar: imaginemos por ejemplo que lo usamos en la siguiente línea:

```
cout << cuadrado(2+3) << endl;
```

Esperamos obtener 25... pero la salida de este programa produce 11. ¿Por qué ha ocurrido esto?

El motivo es que, como ya hemos mencionado, los `#define` realizan un reemplazo mecánico y textual de los elementos que les indicamos, igual que si hiciéramos copy paste, sin entender el significado de su contenido. Por lo tanto, como el `#define` indica que debemos cambiar `cuadrado(x)` por `x*x`, tenemos que fijarnos quién es `x`, copiarlo dos veces y colocar un asterisco en el medio, textualmente, pues eso es lo que hacen los `#define`. En nuestro ejemplo, `x` es `2+3`, pues se ha escrito `cuadrado(2+3)`: Entonces, el fragmento `cuadrado(2+3)` es reemplazado por `2+3*2+3`, ya que el `#define` reemplaza textualmente cada copia de `x` por el texto indicado. Esta expresión, al no tener paréntesis, da por resultado `2+6+3=11`, pero nosotros queríamos realizar `(2+3)*(2+3)=25`.

Esto podría resolverse si utilizamos paréntesis en todos lados :

```
#define cuadrado(x) ((x)*(x))
```

Este `#define` funciona correctamente, pero ahora quedó bastante más feo y difícil de leer. Una función como

```
int cuadrado(int x)
{
    return x*x;
}
```

nos hubiera evitado todos estos problemas, pues en las funciones no se hace un reemplazo textual mecánico del código , sino que se calcula el valor indicado antes de comenzar a ejecutar la función . En el ejemplo con la función cuadrado, se calcularía el valor `2+3=5` antes de iniciar la función, de forma que cuando se comienza a ejecutar la función cuadrado, ya se tiene `x=5`. En este sentido, las funciones son más inteligentes que el #define .

En resumen, utilizaremos `#define` solamente cuando nos permite hacer algo que con una función no podríamos hacer. Por ejemplo, escribir algún tipo de `for` común, o resumir alguna instrucción que declare una variable, es algo que no podríamos reemplazar fácilmente por una función. En cambio, un simple cálculo como elevar al cuadrado sí es algo que podríamos hacer fácilmente con una función, y entonces generalmente conviene hacerlo así para evitar posibles errores, y no tener que llenar todo de paréntesis.

---

1)

Si bien tiene ciertas similitudes con una función matemática, la palabra función en C++ y en la matemática significan cosas bien diferentes

Structs

Motivación

Supongamos que tenemos que guardar los datos de los alumnos de toda una escuela, para poder trabajar con ellos en el programa. Específicamente, por cada alumno tenemos:

- Nombre
- Apellido
- Dia de nacimiento
- Mes de nacimiento
- Año de nacimiento
- Año de escolaridad que cursa
- División (Una sola letra: 'A', 'B', 'C', etc)
- Teléfono
- Dirección

¿Cómo podríamos hacer para almacenar todos estos datos, por cada alumno?

Con las herramientas que vimos hasta el momento, la manera más natural es usar `vector`, para poder tener listas de datos. Concentrémonos por ejemplo en los nombres: Si solamente quisiéramos guardar los nombres de los alumnos, sería fácil, pues tendríamos un `vector<string> nombres;` donde guardaremos todos los nombres y listo.

¿Qué pasa si queremos guardar tanto nombre como apellido? Podemos tener dos vectors diferentes:

```
vector<string> nombres;
vector<string> apellidos;
```

Así, si por ejemplo hiciéramos `nombres = {“Andrea”, “Pablo”, “Marta”}` y `apellidos = {“Aluvia”, “Poncho”, “Muller”}`, tendríamos guardados los datos de 3 alumnos: Andrea Aluvia, Pablo Poncho y Marta Muller.

Notemos que de esta forma, tenemos que asegurarnos de guardar los apellidos y los nombres en el mismo orden , pues sino, no podemos recuperar qué nombre iba con qué apellido. Así, todos los vector deben estar coordinados de forma que el primer dato en todos corresponde al “alumno 1”, el segundo en todos corresponde al “alumno 2” y así siguiendo.

¿Y si tuviéramos que agregar los demás datos? Tendríamos que tener muchos vectors:

```
vector<string> nombres;
vector<string> apellidos;
vector<int> diasNacimiento;
vector<int> mesesNacimiento;
vector<int> annosNacimiento;
vector<int> annosEscolaridad;
vector<char> divisiones;
vector<string> telefonos;
vector<string> direcciones;
```

Si bien este mecanismo de mantener vectores paralelos funciona (y es usual utilizarlo por ejemplo en competencias de programación cuando hay pocos campos de información), en C++ existe una manera mejor de resolver esta situación, y son justamente los Struct, que estudiaremos en esta sección.

Definición de un Struct

Un structes en C++ un tipo de datos compuesto : Es decir, será un tipo de datos que se forma a partir de otros tipos más básicos .

Por ejemplo, una esquina en una ciudad viene dada generalmente indicando las dos calles que allí se cruzan. Podríamos tener así un tipo para las esquinas, que podríamos escribir así:

```
struct Esquina
{
    string calle1;
    string calle2;
};
```

o también de forma completamente equivalente:

```
struct Esquina
{
    string calle1, calle2;
};
```

Notar que las declaraciones de los struct terminan con un `;`, a diferencia de por ejemplo las funciones, que no llevan ese `;`.

De esta forma, acabamos de crear un nuevo tipo de datos, llamado `Esquina`, y que se forma con dos componentes: calle1, de tipo string, y calle2, también de tipo string.

¿Cómo utilizamos este tipo? Podemos declarar variables de tipo Esquina exactamente igual que declaramos las de otros tipos como int o string:

```
Esquina e;
Esquina e2;
e.calle1 = "Cordoba";
e.calle2 = "Medrano";
e2 = e; // Ahora la variable e2 contiene la misma esquina que la variable e
```

El ejemplo muestra que podemos utilizar `.` para acceder a los componentes individuales de un struct. Cada uno de estos componentes funciona como una variable independiente, del tipo correspondiente, y el struct lo que hace es funcionar como una sola "gran variable" que las une a todas. También en el ejemplo vemos que es válido utilizar el operador de asignación para copiar structs, igual que copiábamos variables de tipos básicos como `int`. Esto lo que hará es copiar cada componente.

De manera similar a lo que ocurre con los vectors, es posible[1)] indicar un valor para todo el struct en lugar trabajar con componentes de a una, mediante el uso de llaves para dar los valores en el orden del struct. El ejemplo anterior por lo tanto se podría reescribir como:

```
Esquina e;
Esquina e2;
e = {"Cordoba","Medrano"};
e2 = e; // Ahora la variable e2 contiene la misma esquina que la variable e
```

O directamente en la declaracion:

```
Esquina e = {"Cordoba","Medrano"};
Esquina e2;
e2 = e; // Ahora la variable e2 contiene la misma esquina que la variable e
```

Así, para representar la escuela anterior, podríamos tener un struct con la información del alumno:

```
struct Alumno
{
    string nombre;
    string apellido;
    int diaNacimiento;
    int mesNacimiento;
    int annoNacimiento;
    int annoEscolaridad;
    char division;
    string telefono;
```

```
    string direccion;
};
```

Y tener un solo vector , que guardará directamente Alumnos:

```
vector<Alumno> alumnos;
```

Como un struct define un nuevo tipo que puede usarse igual que los ya existentes, las funciones pueden recibir parámetros de tipo struct:

```
string nombreCompleto(Alumno a)
{
    return a.nombre + " " + a.apellido;
}
```

1)

En C++11

Más tipos de datos básicos de C++

Tipos enteros

En C++ existen múltiples tipos de datos enteros . El más común de ellos es el tipo `int`, que permite almacenar números entre -2^{31} y $2^{31}-1$, inclusive. Si alguna operación da por resultado números fuera de este rango de valores, obtendremos al trabajar con int resultados incorrectos.

Similarmente, existen en C++ otros tipos de datos enteros fundamentales, pero de distinto tamaño : Los `int` tienen un tamaño de 32 bits (dígitos binarios), o 4 bytes, y eso además de definir el espacio de memoria RAM que ocupa cada variable de tipo `int`, limita el rango de valores que estos pueden representar.

En C++ existen otras variables enteras de diversos tamaños:

- `char`: Entero de 8 bits, entre -2^7 y 2^7-1 inclusive, es decir, entre -128 y 127 inclusive. Ya lo hemos utilizado cuando trabajamos con caracteres, pues un caracter en C++ se representa directamente mediante un número, que es su código ASCII [https://en.wikipedia.org/wiki/ASCII].
- `short`: Entero de 16 bits, entre $-2^{15} = -32768$ y $2^{15}-1 = 32767$ inclusive.
- `int`: Entero de 32 bits, entre $-2^{31} = -2147483648$ y $2^{31}-1 = 2147483647$ inclusive.
- `long long`: Entero de 64 bits, entre $-2^{63} = -9223372036854775808$ y $2^{63}-1 = 9223372036854775807$ inclusive.

Para referencia, las máximas potencias de 10 que entran en el rango de cada tipo son respectivamente:

- `char`: Hasta $100 = 10^2$
- `short`: Hasta $10.000 = 10^4$
- `int`: Hasta $1.000.000.000 = 10^9$
- `long long`: Hasta $1.000.000.000.000.000.000 = 10^{18}$

Todos estos tipos se usan de la misma manera que `int`, y solo cambia la cantidad de memoria que utilizan estas variables y el rango de valores posibles. Por ejemplo es perfectamente válido lo siguiente:

```
short x = 32;
int y = 1000;
long long z = x + y;
```

En las operaciones aritméticas, como regla general el tamaño del resultado de una operación es el máximo tamaño de sus operandos, es decir que si sumamos `short` e `int`, obtendremos `int`, y si sumamos `int` con `long long` obtendremos `long long`. Esto es independiente del resultado de la operación, y solo depende de los tipos involucrados.

Tipos enteros sin signo

Todos los anteriores permitían representar números negativos y positivos. Si bien casi nunca los usaremos, existen versiones de los anteriores en las cuales solamente se permiten números no negativos : estas se obtienen agregando `unsigned` (del inglés: “sin signo”) al comienzo del tipo correspondiente. Así podemos obtener los siguientes tipos:

- `unsigned char`: Entero de 8 bits, entre 0 y $2^8 - 1 = 255$ inclusive (Es por esto que en The Legend of Zelda [https://es.wikipedia.org/wiki/The_Legend_of_Zelda_(videojuego)], el máximo número de "Rupies" que se pueden tener es exactamente 255. Estas primeras consolas de videojuegos eran de 8 bits).
- `unsigned short`: Entero de 16 bits, entre 0 y $2^{16} - 1 = 65535$ inclusive.
- `unsigned int` (equivalente a `unsigned` directamente sin aclarar `int`): Entero de 32 bits, entre 0 y $2^{32} - 1 = 4294967295$ inclusive.
- `unsigned long long`: Entero de 64 bits, entre 0 y $2^{64} - 1 = 18446744073709551615$ inclusive.

Es decir, las versiones `unsigned` ocupan la misma memoria que sus correspondientes con signo, no permiten negativos, y a cambio llegan hasta números aproximadamente el doble de grandes como límite superior. La tabla de potencias de diez máximas representables es igual que antes, excepto que en `unsigned long long` ahora entra el número 10^{19}, mientras que en `long long` solo se puede representar hasta 10^{18}.

Tipo de un literal entero

Cuando escribimos un número directamente en el código, como por ejemplo `x = y + 33;`, cabe preguntarse: ¿De qué tamaño es ese 33? Esto es importante para las cuentas intermedias, pues ese tipo define el tamaño del resultado. Por ejemplo, si hacemos `long long x = y + 1000000000;`, donde `y` es de tipo `int`, el resultado "matemático" de la cuenta siempre entrará en el rango de `long long`, pero... ¿Es para la computadora el resultado de la cuenta un `long long`?

La respuesta a esta última pregunta es que no: Los literales enteros son de tipo `int`, automáticamente, a menos que les pongamos un `LL` (indicador de `long long`) al final, en cuyo caso serán `long long`.

Así, en el ejemplo de `long long x = y + 1000000000;`, `y` es de tipo `int`, y el `1000000000` se considera de tipo `int`, por lo tanto el resultado de la suma será de tamaño `int`, y si este resultado se va del rango posible de valores de `int`, quedará en `x` un valor erróneo, incluso cuando el valor verdadero hubiera podido entrar en dicha variable.

Si en cambio hacemos `long long x = y + 1000000000LL;`, no tendremos este problema pues el resultado de la cuenta será un `long long`, al serlo `1000000000LL`.

Similar problema tendremos si hacemos `long long x = y + z`, siendo tanto `y` como `z` variables de tipo `int`. Podemos solucionarlo convirtiendo una de ellas a `long long`: `long long x = y + (long long)(z)`, de forma análoga a lo que hacíamos con el `LL` para los literales enteros.

El caso más común donde esto ocurre es en la expresión: `1 << i` (que es común si se trabaja con operaciones de bits con números de 64 bits): El resultado de esta operación será de tipo `int`, que no es lo que queremos si estamos trabajando valores de 64 bits. `1LL << i` resuelve por completo este problema.

Reglas prácticas para el manejo de tipos enteros

En general, mezclar distintos tamaños y tipos puede llevar a confusiones y errores. Las "reglas prácticas" más comunes a seguir son las siguientes:

- Usar `int`, a menos que sean necesarios números que no entren en `int`. En tal caso, usar `long long` para todas las variables involucradas en estos cálculos con números grandes.
- En las constantes enteras, usar el sufijo `LL` cuando aparecen en una cuenta con números de tamaño `long long`.
- Utilizar los tipos `char` o `short` únicamente si es absolutamente necesario ahorrar memoria.
- Utilizar otros tipos (como por ejemplo los unsigned) solamente en casos excepcionales donde no veamos ninguna buena alternativa.

Observaciones sobre el tamaño de los distintos tipos

Hemos mencionado aquí los tamaños para el caso del compilador gcc para PC, que es probablemente el más utilizado en competencias de programación. Sin embargo, el lenguaje C++ tiene la particularidad de que no garantiza el tamaño exacto de los tipos enteros , que pueden variar entre distintos compiladores del lenguaje.

Por ejemplo, existen compiladores para 64 bits en los cuales `int` es un tipo de 64 bits como `long long`. Si usamos compiladores "antiguos" para el sistema operativo DOS (Como el clásico Turbo C++ [https://en.wikipedia.org/wiki/Turbo_C%2B%2B]), `int` será más chico y tendrá solamente 16 bits, ya que ese era el tamaño normal de los números con los que las computadoras podían trabajar eficientemente.

Tipos de punto flotante

Siempre hemos trabajado con números enteros, que en computación es el caso más común de todos (y especialmente, en competencias de programación como la IOI y la OIA).

Sin embargo, a veces es necesario trabajar con números decimales fraccionarios, especialmente al realizar cómputo científico, simulaciones o videojuegos, donde aparecen cálculos de física y química. Daremos aquí una introducción completamente básica, y aquí se puede ver en cambio una descripción más completa y avanzada de la aritmética de punto flotante.

Existen en C++ fundamentalmente 3 tipos de punto flotante disponibles. El más común, que es el recomendado y que utilizaremos casi siempre, es double . El tipo `double` utiliza 8 bytes (64 bits) para representar un número con coma. La precisión de double es de 15 dígitos decimales, que suelen ser más que suficientes para todos los cálculos que nos interesan.

El tipo se utiliza en cuentas igual que hicimos con los enteros, pero se pueden usar números con coma (que en C++ se escriben siempre con `.`, y nunca con `,`):

```
double x = 0.2;
double y = x * 5.3;
double z = y / (x-0.72);
cout << z << " " << x << endl;
```

Este programa muestra por pantalla `-2.03846 0.2`. Notemos que a diferencia de lo que ocurre con enteros, la división `/` se realiza automáticamente con coma cuando estamos trabajando con `doubles`.

Es muy común querer mostrar una cierta cantidad fija de decimales, y no que esto lo decida el programa arbitrariamente como ocurrió en el ejemplo. Esto se puede hacer incluyendo `#include <iomanip>`, y agregando una línea al programa antes de utilizar `cout` para mostrar los números:

```
double x = 0.2;
double y = x * 5.3;
double z = y / (x-0.72);
cout << fixed << setprecision(7);
cout << z << " " << x << endl;
```

Cambiando el `7` por otro número, elegimos cuántos decimales queremos mostrar. El ejemplo anterior genera la siguiente salida por pantalla:

```
-2.0384615385 0.2000000000
```

Una característica de los números de punto flotante es que no dan resultados exactos . Si bien tienen una precisión de 15 dígitos y normalmente los resultados son muy buenos generalmente, incluso para operaciones muy sencillas, no se puede asumir que los resultados que dan serán exactos , sino que debemos considerar siempre que tienen un pequeño error.

Esto queda claro con el siguiente ejemplo:

```
cout << fixed << setprecision(3);
for (double x = 0.1; x < 1.0; x += 0.1)
    cout << x << endl;
```

Que muestra por pantalla:

```
0.100
0.200
0.300
0.400
0.500
0.600
0.700
0.800
```

```
0.900
1.000
```

Que no es lo que esperamos si suponemos que las operaciones son exactas : El último número que vemos es 1, que no debería haberse mostrado pues pedimos seguir solo si `x < 1`.

Esto ocurre porque a diferencia de lo que pasa con enteros, las cuentas tienen pequeños errores, y en realidad el último número no es 1.000 sino que es un número apenas más pequeño, que cumple `x < 1`. Podemos ver esto si aumentamos la precisión: mostrando 20 cifras decimales vemos

```
0.10000000000000000555
0.20000000000000001110
0.30000000000000004441
0.40000000000000002220
0.50000000000000000000
0.59999999999999997780
0.69999999999999995559
0.79999999999999993339
0.89999999999999991118
0.99999999999999988898
```

Vemos que, si bien los errores relativos son extremadamente pequeños (aparecen recién en la cifra 15), siempre están ahí , así que no podemos tratar a los números como absolutamente exactos en nuestro programa.

Otros tipos de punto flotante

Además de `double`, existen otros tipos de punto flotante en C++ que se usan de idéntica manera:

- `float`: Es un número de punto flotante que ocupa solamente 32 bits. Tiene mucha menos precisión que `double`, por lo cual lo recomendamos solamente cuando estemos ante una emergencia y sea absolutamente necesario ahorrar memoria o acelerar un poquito el tiempo de cálculo de la computadora.
- `long double`: Es un número de punto flotante de 80 bits, con más precisión que double (unas 18 cifras). En general no es necesario (aunque en algunos casos podría serlo), y es más lento y ocupa más memoria.

Archivos

Hasta ahora, siempre hemos leido mediante `cin`, y escrito mediante `cout`. Sin embargo, a veces es cómodo utilizar uno o más (todos los que queramos) archivos de texto de donde sacar los datos, y similarmente escribir a uno o más archivos los resultados de nuestro programa.

En C++, utilizar archivos es muy fácil, ya que la forma de hacerlo es prácticamente igual a lo que ya venimos haciendo con cin y cout.

Para utilizar archivos y hacer funcionar los ejemplos de esta sección, tendremos que incluir `#include <fstream>` en nuestros programas.

Archivos de entrada

Es posible crear una variable que represente un archivo de entrada , del cual el programa leerá datos . Estas variables se utilizan de exactamente igual manera que `cin`, y se declaran como en el siguiente programa de ejemplo:

```
#include <fstream>

using namespace std;

int main()
{
    ifstream archivo1("nombre1.txt");
    ifstream archivo2("nombre2.in");
    int x,y,z;
    archivo1 >> x>> y >> z;
    string a; int b;
    archivo2 >> a >> b;
    return 0;
}
```

En el ejemplo se ve que podemos leer datos contenidos en los archivos exactamente igual que hacíamos con cin, pero indicando la variable del archivo correspondiente. En lugar de tener que ser tipeados por el usuario, el programa recibe los datos contenidos en los archivos correspondientes. Las variables son de tipo ifstream (Del inglés, "Input File Stream"), y al declararlas se indica entre paréntesis el nombre del archivo del cuál se leerá utilizando esa variable. Dicho archivo debe existir y contener los datos deseados, al momento de ejecutar el programa.

En este ejemplo se usan dos archivos distintos: El contenido de "nombre1.txt" se guarda en x,y,z (que tendrán 3 enteros) y el de "nombre2.in" se guarda en a,b (Un string y un entero).

Archivos de salida

Los archivos de salida se pueden utilizar de manera completamente análoga a los de entrada, pero operando como si fueran `cout`, y su tipo será ofstream (Del inglés "Output File Stream").

```
#include <fstream>

using namespace std;

int main()
{
```

```
    ofstream archivo1("nombre1.txt");
    ofstream archivo2("nombre2.out");
    int x = 32,y = 10;
    archivo1 << x << " " << y << " " << -33 << endl;
    string a = "pepe"; int b = 100;
    archivo2 << a << endl << b << endl;
    return 0;
}
```

Luego de ejecutar este programa, “nombre1.txt” contendrá lo siguiente:

```
32 10 -33
```

y “nombre2.out” contendrá lo siguiente:

```
pepe
100
```

Ejemplos de aplicación

Los problemas de la Olimpíada Informática Argentina anteriores al 2014 utilizaban entrada y salida mediante archivos. Se puede practicar utilizar archivos de entrada y salida por ejemplo con este problema [http://juez.oia.unsam.edu.ar/#/task/mensajes/statement] o con este otro [http://juez.oia.unsam.edu.ar/#/task/maraton/statement].

Idea

Si queremos el máximo común divisor (mcd) entre A y B (digamos que $A \geq B$), podemos pensar esto:

Si un número d divide a A y a B, entonces divide a la diferencia. Esto se ve ya que si $A = n * d$ y $B = m * d$, entonces $(A - B) = (n - m) * d$, que como n y m son enteros es múltiplo de d.

Entonces, si todos los divisores comunes de A y B dividen a la diferencia, en particular el más grande lo hará $\Rightarrow mcd(A, B) = mcd(A, A - B)$.

Ahora, qué pasa si queremos usar esto para calcular $mcd(10000, 1)$? Diríamos "bueno, eso es igual a $mcd(9999, 1)$, que es igual a $mcd(9998, 1), \ldots$. Pero eso es bastante lento. Algo que podemos pensar es que al principio, cuando tenemos $mcd(10000, 1)$, ya sabemos cuántas veces vamos a restar el 1 al número más grande. Tantas veces como pueda mientras siga siendo no negativo.

Entonces si $A = k * B + r$, con r no negativo, le vamos a restar k veces el número B al otro número, y luego de eso nos va a quedar r en vez de A, junto con B que no lo modificamos.

Les suena la expresión $A = k * B + r$? El valor de r viene a ser el resto de A en la división por B.

Este algoritmo de ir calculando los restos y calculando el mcd de números cada vez más chicos se conoce como algoritmo de Euclides.

Código

Un código pequeño pero muy efectivo a la hora de buscar un máximo común divisor es el siguiente:

```
int mcd(int a, int b){
    if(b==0){
        return a;
    }else{
        return mcd(b, a%b);
    }
}
```

Complejidad

Esto es importante para asegurarnos que este método conviene más que ir mirando los divisores de uno de los dos y ver si divide al otro.

Si tenemos los números A y B, con $A > B$, cuánto se achica el número A al reemplazarlo por $r = A\%B$?

- Si $B > A/2$, entonces $A = B + r$ con $r < A/2$
- Si $B \leq A/2$, entonces como el resto r de dividir por B es siempre menor que $B \Rightarrow r \leq B/2$

Entonces el número más grande será a lo sumo la mitad del más chico. Luego de un paso más, el número que no se modificó, B, será a lo sumo la mitad del otro, que es a lo sumo la mitad de B, por lo que B se reemplaza por a lo sumo $B/4$.

De este razonamiento se puede ver que la complejidad es del orden del logaritmo del número más chicos de los iniciales.

Comentario

Una propiedad válida para todo par de enteros A, B es que $A * B = mcd(A, B) * mcm(A, B)$, donde mcm denota el mínimo común múltiplo. Entonces, una manera de calcular el mcd de dos enteros de manera rápida es hacer simplemente $mcd(A, B) = A * B / mcd(A, B)$.

Enunciado

Calcular el resto de a^b en la división por m. Acá, el número b puede ser muy grande, y sin embargo la respuesta no, ya que calculamos el resto en la división por m.

Idea

La idea es aprovechar el hecho de que $a^b = (a^2)^{b/2}$. Vamos a implemente una función recursiva que eleve al cuadrado la base (haciendo simplemente la cuentita) y divida por dos el exponente. El único cuidado que hay que tener es que si b es impar, al hacer $b/2$ estamos agarrando la parte entera de esa división, entonces tenemos que multiplicar por un a más.

Código

```
int my_pow(int a, int b, int m){
    if(b==0){
        return 1;
    }else{
        if(b%2==0){
            return my_pow(a*a%m, b/2, m)%m;
        }else{
            return (a*my_pow(a*a%m, b/2, m))%m;
        }
    }
}
```

Si m es grande, tanto como para que $a * a$ se pueda pasar del valor máximo de int, se recomienda usar long long int en esta función, o tener sumo cuidado (se pueden hacer cosas como 1LL*..., es decir, convertir a long long en el momento de un producto y luego al hacer $\%m$ el número vuelve a entrar en int).

Búsqueda lineal y binaria

Las técnicas de búsqueda lineal y binaria son las dos técnicas más fundamentales que existen en computación para encontrar un elemento particular de entre un conjunto de posibles opciones.

Empezaremos con un problema de motivación, pero finalmente veremos una manera de pensar que nos permitirá aplicar la búsqueda binaria utilizando siempre el mismo código, en otras situaciones muy distintas pero manteniendo siempre la misma idea fundamental.

Problema motivador original

Dado un arreglo ordenado de menor a mayor, sin repetidos, y un número particular que se desea buscar , determinar si el número deseado está en el arreglo o no.

Búsqueda lineal

Este algoritmo es muy fácil de programar, y su idea es muy simple e intuitiva. Se recorrerán todas las posiciones del arreglo, barriendo todos los candidatos posibles uno por uno. En cada uno, realizamos la verificación necesaria : Si el elemento es igual al número deseado, entonces sabemos que el número está, y podemos finalizar la ejecución. Si en cambio, luego de haber recorrido todas las posiciones, no hemos encontrado en ninguna un valor que coincida con el número buscado, entonces este no está.

El código correspondiente a este ejemplo podría ser:

```
bool estaEnArreglo(int numeroDeseado, const vector<int> &elementosOrdenados){
    for(int x : elementosOrdenados){
        if(x == numeroDeseado){
            return true;
        }
    }
    return false;
}
```

Ventajas:

- Fácil de entender y programar
- Funciona en cualquier arreglo (ordenado o no) , con tan solo tener un operador de igualdad.

Desventajas

- Ineficiente: complejidad de peor caso $\Theta(N)$

La búsqueda lineal se utiliza al menos una vez en casi todos los problemas: no necesariamente aquello que estamos "buscando" es un número que se encuentra cargado en un arreglo, sino que en muchos problemas tenemos muchos "candidatos" posibles (que pueden ser casillas, argumentos a una función matemática, o cualquier otra lista de cosas), y por ejemplo queremos descubrir "el mejor" de todos ellos[1], para lo cual lo que hacemos es recorrer todos y guardarnos en un acumulador el mejor hasta el momento.

Búsqueda binaria

En el ejemplo que acabamos de ver, nunca hemos aprovechado que el arreglo está ordenado. Ese mismo código visto funciona perfecto con cualquier arreglo, ordenado o no, pero en peor caso recorre los n elementos del arreglo.

El algoritmo de búsqueda binaria se basa en aprovechar que el arreglo está ordenado, para poder descartar muchas posiciones de la búsqueda rápidamente, con una sola observación.

Si bien el objetivo final que hemos planteado en este caso es únicamente decidir si el elemento está o no está, teniendo así una salida de tipo booleano ("sí" o "no"), al considerar el problema como un problema de búsqueda, lo que queremos es encontrar el elemento, y al encontrarlo, lo encontraremos necesariamente en alguna posición del arreglo, es decir, un índice en el mismo. Por ejemplo en el arreglo $[3, 7, 15, 19]$, el 3 aparece en la posición 0, el 7 aparece en la posición 1 y el 19 en la posición 3.

En términos de posiciones, la búsqueda lineal que vimos antes se ocupa de probar todas las posiciones una por una, desde la 0 hasta la $n - 1$ inclusive. Esto es necesario si buscamos en un arreglo desordenado. Ahora bien, supongamos que tenemos el siguiente arreglo ordenado: $[2, 10, 20, 50, 100, 150, 210, 1000, 1005, 2000]$

Supongamos que el número buscado es el 1. Podríamos pensar en comenzar por la primera posición e ir probando todas en orden, ya que así funciona la búsqueda lineal. Pero si observamos con cuidado, al examinar la primera posición del arreglo encontraríamos un 2, y se tiene que $1 < 2$: Estamos encontrando en la primera posición un elemento que es mayor que el que buscamos. Pero si el arreglo está ordenado, entonces todos los números que siguen también serán mayores: Al estar en orden, los de más a la derecha son más grandes, y si el 2 ya "se pasó" del número buscado, todos los siguientes también "se pasarán", necesariamente.

Lo anterior nos permitiría concluir luego de examinar una única posición, que el 1 no se encuentra en el arreglo. Ya podemos ver la ganancia que produce tener el arreglo ordenado para este problema: podemos descartar muchas posiciones con una única pregunta. Podemos resumir la observación que acabamos de descubrir para nuestro problema así: Si alguna posición del arreglo se pasa, todas las siguientes se pasan, en donde "se pasa" significa que el número allí guardado es mayor que el buscado, y por lo tanto el buscado tiene que estar sí o sí a su izquierda.

Algo diferente hubiera ocurrido, sin embargo, si hubiéramos buscado el número 2018. Como $2 < 2018$, al examinar el primer elemento del arreglo estaríamos encontrando algo menor a lo buscado, y como los números crecen hacia la derecha, concluimos que el 2018 deberá aparecer más adelante si está. En este caso, con nuestra pregunta solamente habríamos descartado al primer 2, y no estaríamos ganando mucho como ocurría antes. ¿Qué pasaría si en cambio decidimos examinar la última posición del arreglo? En este caso, encontraríamos un 2000, y como $2000 < 2018$, tenemos que el 2000 es todavía más chico que el número buscado. Como el arreglo está ordenado, a la izquierda del 2000 los números son aún más pequeños todavía, y por lo tanto todos seguirán siendo menores que el número buscado. En este caso, examinando solamente el número del final, habríamos descartado todos y concluido que el 2018 no se encuentra en el arreglo.

Podemos de manera similar a lo que habíamos hecho antes, resumir la observación que hicimos de la siguiente manera: Si alguna posición del arreglo se queda chica, todas las anteriores se quedan chicas, en donde "se queda chica" significa que el número allí guardado es menor que el buscado, y por lo tanto el buscado tiene que estar sí o sí a la derecha.

Hemos descubierto dos nociones distintas muy útiles, la de "pasarse" (porque las de la derecha también se pasan) y la de "quedarse chica" (porque las de la izquierda también se quedan chicas), pero en realidad son ideas opuestas, así que por comodidad nos conviene hablar de una sola. Usaremos la de pasarse.

Entonces, vamos a imaginarnos que una posición i del arreglo se pasa cuando el número allí guardado es mayor que el buscado: `arreglo[i] > buscado`. Si una posición i se pasa, el número no puede estar en ninguna posición j que sea $j \geq i$.

Lo que ya hemos observado entonces es que si una posición i no se pasa (es decir, `arreglo[i] <= buscado`), como las que están a la izquierda tienen valores más chicos, esas tampoco se pasan. Y si una posición no se pasa, entonces el número buscado está allí o más a la derecha, es decir, no puede estar en ninguna posición j que sea $j < i$.

La idea del método de búsqueda binaria es ir examinando posiciones, para ver si se pasan o no se pasan. En base a eso, podemos usar lo que sabemos para descartar muchas posiciones a la vez, y consultar solamente por las que quedan por descubrir. Surge con esta idea una pregunta natural: ¿Qué posición nos conviene examinar primero?

Vimos dos ejemplos en los cuales examinamos un elemento en un extremo del arreglo. En ambos casos ocurría lo mismo (pero con las direcciones izquierda y derecha intercambiadas): Podíamos tener suerte y resolver todo el problema en una sola pregunta, pero en el peor caso, la comparación realizada solamente nos permitía descartar este elemento extremo, teniendo que explorar aún todo el resto del arreglo. ¿Qué pasaría entonces si en lugar de consultar por un elemento en un extremo, consultamos por un

elemento en la mitad del arreglo?[2]. En este caso, si esa posición se pasa, ya no hace falta examinar la mitad derecha, pues también se pasa. Si en cambio, esa posición central que comenzamos examinando no se pasa, entonces ya no hace falta examinar ninguna posición en la mitad izquierda del arreglo, pues tampoco se pasará. Si justo el elemento que encontramos es el que buscábamos, podríamos para la búsqueda inmediatamente si así lo deseamos.

Si continuamos de esa manera, siempre consultando un elemento central del subarreglo de elementos "desconocidos" (aquellas posiciones que aún no sabemos si se pasan o no se pasan), en cada paso descartamos la mitad de los elementos restantes. De esta manera, el total de elementos examinados será únicamente $\lceil \lg N \rceil$, que es muchísimo mejor que las N consultas de la búsqueda lineal.

A continuación se puede ver un código de ejemplo que muestra esta idea. Mantendremos todo el tiempo dos posiciones especiales: la más grande conocida que no se pasa, y la más chica conocida que se pasa. Observemos que estas dos posiciones resumen toda la información que hemos conseguido hasta el momento con nuestra búsqueda[3].

```
bool estaEnArreglo(int numeroBuscado, const vector<int> &arreglo)
{
    // Guardamos siempre el mas grande que **sabemos que no se pasa**,
    //                    y el mas chico  que **sabemos que se pasa**
    // Inicialmente, no sabemos nada de ningun elemento en el arreglo,
    // asi que les ponemos los valores extremos -1 y N que estan "justo afuera".
    // Es como si el arreglo ordenado tuviera un -INF en la posicion -1, y un
    // +INF en la posicion N.
    int noSePasa = -1, sePasa = int(arreglo.size());
    while (sePasa - noSePasa > 1) // Si quedan posiciones "desconocidas", seguimos buscando
    {
        int medio = (sePasa + noSePasa)/2;
        if (arreglo[medio] == numeroBuscado)
            return true;
        else if (arreglo[medio] > numeroBuscado)
            sePasa = medio;
        else
            noSePasa = medio;
    }
    return false;
}
```

Un esquema más general

Hemos presentado la versión anterior de la búsqueda binaria primero, porque es una manera natural de desarrollar el tema de la forma en que deseamos mostrarla sobre el ejemplo de búsqueda en el arreglo ordenado. Este ejemplo es, por lejos, el ejemplo más común con el que se enseña y se muestra por primera vez la idea de búsqueda binaria[4], y por eso lo hemos utilizado.

Sin embargo, no es un ejemplo del caso más simple ni del más común de todos, cuando se utiliza búsqueda binaria en competencias de programación. El ejemplo anterior puede verse como un caso particular de la más compleja búsqueda binaria separadora, en que se separa en tres partes ("los menores al buscado", "los iguales al buscado", "los mayores al buscado") y además se decide cortar la separación prematuramente no bien sabemos que el "rango de iguales" es no vacío.

Veremos a continuación un esquema de la búsqueda binaria más común y más simple, que es conveniente conocer y utilizar siempre que sea posible, para minimizar la chance de cometer errores en la búsqueda binaria.

Motivación

El ejemplo sencillo que tomaremos como motivación será el de encontrar una pseudo-raíz-cuadrada de un número positivo dado N. Definimos la pseudo-raíz-cuadrada de un número positivo N, como el número positivo x tal que $x^2 + x = N$[5].

Así, la raíz cuadrada de 4 es exactamente 2, pero su pseudo-raíz-cuadrada es aproximadamente 1.5615, pues $1.5615^2 + 1.5615 \approx 4$.

Específicamente, nos mantendremos en enteros y nos alcanzará con encontrar la parte entera de la pseudo-raíz-cuadrada: Por ejemplo si nos dieran 4 como entrada, la respuesta es 1 porque es la parte entera de 1.5615.

Si bien es posible utilizar observaciones matemáticas para encontrar eficientemente la respuesta (asumiendo que podemos calcular raíces cuadradas normales), si cambiáramos un poco la función (es decir la "cuenta" que estamos haciendo) habría que cambiar completamente el método. Veremos una solución con búsqueda binaria que sirve para lograr esto mismo sin asumir casi nada sobre "la cuenta" que hacemos.

La receta

Lo primero que tenemos que hacer si queremos aplicar búsqueda binaria, es identificar una propiedad binaria. ¿Qué es una propiedad binaria? Ya vimos un ejemplo antes: la noción de que una posición "se pasa", era una propiedad binaria: hasta un cierto punto no se cumple, pero a partir de la primera posición donde se cumple, de ahí en adelante se cumple siempre.

Es decir, una propiedad binaria nos clasifica los números enteros en dos partes: Los que no cumplen la propiedad, y los que la cumplen, de manera que todos los que cumplen vienen después que todos los que no cumplen. Otra manera de decir lo mismo más resumidamente es que, si x cumple la propiedad, $x + 1$ también.

Por ejemplo, "ser par" no es propiedad binaria, porque los pares y los impares están todos "mezclados" en el orden de los enteros, no están todos los pares "de un lado" y todos los impares "del otro". En cambio, "ser positivo" es una propiedad binaria.

Solamente podremos aplicar búsqueda binaria a propiedades binarias. Un error común es intentar usar búsqueda binaria en una propiedad que no lo sea. De lo que se encarga la receta de búsqueda binaria que veremos, es de encontrar los valores a y b "extremos" de la propiedad: es decir, aquellos tales que $b = a + 1$, a no cumpla la propiedad, y b si la cumpla. Podemos pensar que entre a y b justamente estará el "corte" de la propiedad: a es el más grande que no cumple, mientras que b es el más chico que sí la cumple.

Para todos los problemas que se resuelven con una búsqueda binaria normal, puede plantearse una propiedad binaria, de tal manera que sabiendo a y b podamos hacer con ellos lo que corresponda según nuestro problema. Y es muy útil pensar los problemas de esta forma, porque entonces la misma búsqueda binaria clara y prolija podemos utilizarla en todos nuestros problemas, siempre idéntica, reduciendo así muchísimo la posibilidad de tener errores al programarla.

La forma en que programamos la búsqueda binaria para encontrar a y b es muy similar a lo que ya hicimos para el ejemplo previo, pues nuestra propiedad binaria en ese caso era "ser un índice i tal que $arreglo[i] > x$", siendo x el número buscado: a siempre será un número que no cumpla, y b siempre será un número que sí cumpla. Esta es la característica fundamental de la receta, y facilita mucho recordarla y programarla sin errores:

```
int a = numeroQueNoCumpla; // Depende del problema, antes fue -1
int b = numeroQueSiCumpla; // Depende del problema, antes fue N
while (b-a > 1) // Mientras no encontramos el "corte" de la propiedad:
{
    int c = (a+b)/2; // Tomamos un elemento en el medio para analizar
    if (c cumple la propiedad)
       b = c; // Mantenemos la regla: b es siempre uno que cumple
    else
       a = c; // Mantenemos la regla: a es siempre uno que no cumple
}
// Termino la busqueda binaria y nuestros resultados listos para usar son:
//  a : Tiene el mas grande que no cumple
//  b : Tiene el mas chico que si cumple
```

Una propiedad muy útil de esta receta es que, como los elementos que se examinan son los del centro del rango, sabemos que durante la ejecución solamente se preguntará en el "if" que verifica la propiedad, por valores de c que se encuentren estrictamente entre los a y b iniciales. En el ejemplo que vimos antes, como $a = -1$ y $b = N$, eso nos garantizaba que solamente se iba a consultar la propiedad con valores de c en el rango $[0, N)$, que son los valores válidos para acceder al arreglo y entonces no había problemas de acceso fuera de rango.

El ejemplo motivador con la receta

¿Cómo podemos aplicar esta idea al ejemplo de calcular la pseudo-raíz-cuadrada? La cuenta $x^2 + x$ justamente parte a los números en dos: Es posible que $x^2 + x > N$, o que no sea así. Si ignoramos los números negativos, que no importan para nuestro problema, como la pseudo-raíz-cuadrada exacta cumple $x^2 + x$ pero necesitamos su parte entera, tenemos que redondear hacia abajo, y por lo tanto nuestro resultado r será menor o igual, teniendo $r^2 + r \leq N$.

Es decir, seguro que el r que buscamos es un número tal que $r^2 + r \leq N$. Además, podemos observar que justamente el r que nos dará la respuesta es el r más grande que verifique esta desigualdad. Si tomamos entonces como propiedad binaria de un número x, que sea $x \geq 0$ y que $x^2 + x > N$, el último número que no cumpla esto será justamente la raíz (porque el primero en cumplirlo es el primero que "se pasa" estrictamente). En otras palabras, si usamos la receta con la propiedad, al terminar el valor a será justamente la pseudo-raiz-cuadrada de N.

Para poder usar la receta, solamente nos falta determinar valores iniciales que cumplan y que no cumplan la propiedad . Es algo con lo que hay que tener cuidado porque cambia de problema en problema, y es un error común usar por error usar valores erróneos en la inicialización, lo que arruina toda nuestra búsqueda binaria.

En este caso, un valor inicial que podemos usar para a es 0: Como el número N de entrada es positivo y por definición de la pseudo-raiz-cuadrada, la parte entera buscada nunca es negativa, así que 0 nunca cumple la propiedad, ya que $0^2 + 0 = 0 \leq N$. Para b hay que razonar un poco más, pero no tanto: Como para un número entero positivo x siempre es $x^2 > 0$, tenemos que siempre $x^2 + x > x$, y por lo tanto N es un valor que cumple (es decir, el mismo N siempre se pasa de su pseudo-raiz-cuadrada), así que podemos comenzar con $b = N$[6].

Con esto en mente, copiamos la receta aplicando la propiedad de nuestro caso y nos queda:

```
typedef long long tint;
tint pseudoRaizCuadrada(tint N)
{
    assert(N > 0);
    // La propiedad es: x >= 0 && x*x+x>N
    tint a = 0; // NO cumple la propiedad
    tint b = N; // SI cumple la propiedad
    while (b-a>1)
    {
       tint c = (a+b)/2;
       if (c*c+c > N) // c cumple la propiedad??
          b = c; // Siempre b es uno que SI cumple
       else
          a = c; // Siempre a es uno que NO cumple
    }
    return a; // Justamente, la respuesta es el ultimo que no cumple nuestra propiedad
}
```

El ejemplo usa la función assert simplemente por claridad, ya que estamos asumiendo que $N > 0$. Notar que no hizo falta verificar `c >= 0`, pues sabemos que el a inicial es 0, y entonces todos los c que se examinarán en la búsqueda binaria serán mayores, o sea positivos.

El ejemplo original, pero ahora con la receta

Volvamos al ejemplo de programar una función que responda eficientemente si un arreglo ordenado de números contiene un valor buscado.

Para aplicar la receta, tenemos que pensar una propiedad binaria, que nos ayude a resolver el problema. La propiedad lo que hace siempre es "partir" los números en dos, los que cumplen y los que no, y nos devuelve dónde está ese "corte". Así que necesitamos una propiedad de manera tal que el lugar donde corte, nos sirva para saber si el número está o no está.

Justamente como el arreglo está ordenado, sabemos que si el `numeroDeseado` aparece, todos los siguientes son mayores o iguales. En cambio, todos los anteriores son estrictamente menores [7]. Es decir, si el número está presente en el arreglo, está en el primer índice i tal que $arreglo[i] \geq numeroDeseado$. Con lo cual si tomamos como propiedad "$x \geq 0$ y también $arreglo[x] \geq numeroDeseado$, o bien $x \geq N$"[8], el primer valor que cumple será la posición donde encontraremos al elemento.

Podemos verificar que la propiedad elegida es binaria: para esto basta ver que si x la cumple, entonces $x + 1$ también. Si x la cumplía por haberse pasado del arreglo, $x + 1$ también. Sino, x era la posición de un elemento que ya era mayor o igual que N, así que al consiedar $x + 1$, como el arreglo está ordenado , tendremos un número que no es más chico, y por lo tanto también será mayor o igual que N.

Al ser una propiedad binaria, podemos usar búsqueda binaria para encontrar los extremos (el último que no cumple, y el primero que cumple), y luego simplemente verificamos al final si el elemento buscado efectivamente está allí, en la posición donde debería

estar (que es la primera que cumple).

```
bool estaEnArreglo(int numeroDeseado, vector<int> elementosOrdenados)
{
    const int N = int(elementosOrdenados.size());
    int low = -1; // Aca guardamos hasta donde sabemos que son menores
    int high = N; // Aca guardamos desde donde son mayores o iguales

    while(high-low>1)
    {
        int mid = (high+low)/2;
        if(elementosOrdenados[mid]>=numeroDeseado)
            high = mid; // Siempre high es uno que SI cumple
        else
            low = mid;  // Siempre  low es uno que NO cumple
    }
    // Como desde high sabemos que son todos mayores o iguales, si la posicion high es mayor ya esta,
    // desde aca son todos mayores y el numero no esta. Entonces si esta, esta en la posicion high.
    return high<N && elementosOrdenados[high]==numeroDeseado; // Pensar por que pedimos high<N
}
```

Esto que hemos hecho de encontrar "la primera posición mayor o igual que un cierto valor dado" es lo que en C++ se llama una consulta de lower_bound, y podrían utilizarse funciones ya programadas para eso (que por dentro, utilizan una búsqueda binaria como la que estamos enseñando).

A modo de ejemplo, esta misma solución utilizando el `lower_bound` ya existente en C++ sería:

```
bool estaEnArreglo(int numeroDeseado, vector<int> elementosOrdenados)
{
    // lower_bound devuelve un *iterador*, que apunta a la posicion que antes llamamos "high"
    auto it = lower_bound(elementosOrdenados.begin(), elementosOrdenados.end(), numeroDeseado);
    // Lo que antes era high<N, ahora es que el iterador no sea el .end()
    return it!=elementosOrdenados.end() && *it==numeroDeseado;
}
```

También se podría haber utilizado la función `binary_search` ya existente en C++. Conocer las funciones existentes muy útil para programar más rápido y con menos errores, pero no reemplaza saber programar nuestra búsqueda binaria, ya que no siempre nos sirve el resultado exacto de las funciones estándar, o a veces necesitamos utilizar las ideas de los algoritmos pero modificadas para nuestro caso particular.

Posibles bugs

- Setear high en $size - 1$, ó low en 0
- Hacer while(low<=high)
- Hacer $low = mid + 1$, ó $low = mid - 1$ en vez de $low = mid$ (lo mismo al mover el valor de high)

Para evitar estos bugs, si bien vale memorizarse la función, puede ser que se nos olvide algo, o nos confundamos. Está bueno pensar bien antes de empezar a programar la búsqueda, en qué índice estoy seguro de que se cumple la propiedad del if y en qué índice estoy seguro de que no. Además, qué pasa si se cumple en todos los elementos o en ninguno (cuáles terminan siendo los valores de low y high). En este caso por ejemplo, tuvimos que tener cuidado de verificar que sea `high < N` al final de la función, para evitar acceder a un elemento fuera de rango si resulta ser `high == N`, lo cual ocurre cuando el elemento buscado es mayor que todos los del arreglo.

Resumen de búsqueda binaria

Ventajas:

- Eficiente: complejidad de peor caso $\Theta(\lg N)$

Desventajas:

- Un poco más complicada de programar que la búsqueda lineal.
- Solamente funciona cuando tenemos datos "ordenados" de alguna forma:
    - Ya sea un arreglo ordenado, sobre el que buscamos valores
    - O más en general, la propiedad que estemos estudiando debe ser binaria

Sobre búsqueda binaria como una manera de invertir funciones monótonas

Supongamos que tenemos una función, ya sea una "cuenta" como en matemáticas del estilo $x^2 + 3x$, o simplemente una función como en programación: un mecanismo (o código) a partir del cual podemos obtener un valor $f(x)$ a partir de cada x [9].

El problema "directo" consiste en calcular $f(x)$, a partir de x, o sea computar la función. Supongamos que ese podemos resolverlo, sea porque la f es una simple cuenta de matemáticas, o porque ya la tenemos programada. El problema inverso consiste en, a partir de un valor y dado, encontrar un cierto x para que sea $y = f(x)$. Es decir, encontrar el x sabiendo el resultado que debería dar la f. En este sentido, hablamos de que queremos invertir la función.

Muchas veces, ocurrirá que la función es monótona. Por ejemplo, puede ser creciente, en cuyo caso si $x \leq y$, tendremos también $f(x) \leq f(y)$ [10].

Cuando tenemos una función monótona, podemos siempre usar búsqueda binaria para invertirla: Para esto, basta considerar la propiedad $P(x) \equiv f(x) \geq y$, donde y es el resultado deseado de f. Como la f es creciente, esta propiedad es binaria: Una vez que un x la cumple, tomar x más grandes solamente agranda el valor de f, y por lo tanto se seguirá cumpliendo para siempre. Por lo tanto, podemos aplicar la técnica para encontrar el primer x que satisface $f(x) \geq y$ (por lo que sabremos que $f(x-1) < y$).

Esto no es más que otro ejemplo del lower_bound ya mencionado antes. La clave es que, si hay algún valor de x donde $f(x) = y$, entonces el primero en cumplir la propiedad anterior debe ser el primero de ellos.

Exactamente esta misma idea utilizamos antes para calcular la pseudo-raíz-cuadrada: Allí estábamos utilizando búsqueda binaria para invertir la función f dada por $f(x) = x^2 + x$. Notar que esta es una función monótona: no hubiéramos podido aplicar la técnica directamente para invertir, por ejemplo, la función $x^4 - 5x^3 + 10x^2 - 30x$, ya que incluso mirando solamente los x positivos, esta función decrece primero pero crece luego.

Como comentario simpático pero no muy útil, todas las búsquedas binarias podemos pensarlas como un caso particular de invertir funciones: Estamos invirtiendo la función f tal que $f(x) = 1$ si x cumple la propiedad, y $f(x) = 0$ si x no la cumple.

Sobre búsqueda binaria con punto flotante

Si bien pueden aparecer en competencias para secundarios, los usos de búsqueda binaria con punto flotante ocurren principalmente en competencias de programación para universitarios.

Cuando trabajamos con una propiedad binaria pero en los números reales, no podemos hablar de un "último" que no cumple, ni de un "primero" que cumple, ya que en la recta numérica estos forman un continuo, y los números no tienen un anterior ni un siguiente.

Podemos sin embargo hablar de un punto de corte de la propiedad: Un cierto punto clave $c \in \mathbb{R}$, tal que ninguno de los menores a c cumple la propiedad, y todos los mayores que c cumplen la propiedad. El propio c podría cumplir o no la propiedad, pero eso no será importante, porque de cualquier manera el método de búsqueda binaria no calcula el valor de c en forma exacta.

Lo que podemos hacer es utilizar la misma receta que antes, pero utilizando números de punto flotante para las variables a, b, c. En lugar de terminar como antes cuando a y b eran "consecutivos", la búsqueda terminará cuando a y b estén "suficientemente cerca". En ese momento, sabemos que el punto de corte verdadero c está en algún lugar entre a y b, con lo cual lo tenemos determinado aproximadamente.

```
const double EPS = 1e-9;
double a = algoQueNoCumpla;
double b = algoQueSiCumpla;
// EPS es un double que indica cuanto error toleramos:
// Mientras menos error, mas pasos tarda.
while (b-a > EPS)
{
   double c = (a+b)/2.0;
   if (c cumple)
       b = c;
   else
```

```
        a = c;
}
// El punto de corte verdadero esta entre a y b: tomamos un valor aproximado.
double corte = (a+b)/2.0;
```

Sobre truquito anti-overflow

En el paso de la búsqueda binaria, todos los ejemplos anteriores utilizaron por claridad `c=(a+b)/2`. Esta cuenta puede dar overflow si `(a+b)` puede ser muy grande, incluso cuando a y b estuvieran ambos dentro del rango de un entero de 32 o 64 bits.

Un truco que puede usarse en tales casos extremos es cambiar la cuenta por su equivalente `c=a+(b-a)/2`, que si estamos trabajando con números no negativos, no tendrá overflow, ya que la resta es menor que el `b`, que ya está entrando (suponemos) en nuestras variables. Si utilizamos números positivos y negativos en la búsqueda binaria (es menos común pero puede ocurrir), entonces dependiendo de los números es posible que este truquito no gane nada.

De cualquier manera, en la mayoría de los problemas nos evitamos estos inconvenientes de overflow utilizando `long long`, y no es necesario recurrir a medidas extremas (porque los números o bien "son mucho más chicos que el máximo" o bien son "mucho más grandes que el máximo", de forma que este truquito no afecte).

Máximos y mínimos en funciones unimodales

Es posible utilizar búsqueda binaria para encontrar máximos y mínimos en funciones unimodales: esto se explica en este artículo.

Material adicional

Charla de Facundo Gutiérrez dada en el Nacional de OIA de 2017 : Charla Busqueda binaria [http://foro.oia.unsam.edu.ar/uploads/default/original/1X/5d40fd2f2d1dae5f44c376845ae820dfa99e5157.pdf]

1)
El significado exacto de "el mejor" varía, lógicamente, de problema en problema, pero este esquema de búsqueda se mantiene siempre igual.
2)
Si el arreglo tiene una cantidad par de elementos, hay dos elementos centrales, y podríamos tomar cualquiera de ellos
3)
A menos que encontremos el elemento en sí, pero en ese caso esta implementación particular corta inmediatamente la búsqueda
4)
La analogía de "buscar una palabra en el diccionario" es exactamente equivalente, pues el diccionario es el arreglo y la palabra es el número buscado.
5)
Notemos que la raíz cuadrada normal se obtiene usando $x^2 = N$
6)
También se podría haber tomado $N + 1$, o $N + 27$, o $3N$: Lo importante es asegurarse de empezar eligiendo algún valor que cumpla la propiedad
7)
Si consideramos la primera aparición del número, en caso de que aparezca varias veces
8)
Considerar que todos los valores a partir de N, es decir que ya "se salieron" del arreglo, cumplen la propiedad, es como pensar que el arreglo ordenado "continúa para siempre" con valores "+INF"
9)
En muchos problemas, la función a la cual le aplicamos esta técnica se calcula ejecutando a su vez algún otro algoritmo, como podría ser un BFS, un algoritmo goloso, un algoritmo de programación dinámica, etc
10)
Si fuera decreciente, basta hacer lo mismo pero invirtiendo las comparaciones.

Introducción

Es una técnica que consiste en resolver problemas de forma recursiva partiendo un problema en uno o más problemas más chicos.

Identificamos dos partes:

- Divide: Las llamadas recursivas a problemas más chicos
- Conquer: Formación de la solución al problema original

Ejemplo: Merge Sort

En el algoritmo de merge sort utilizamos la técnica de Divide & Conquer.

- Divide: Llamadas a sort de las dos mitades del arreglo
- Conquer: Construcción del ordenamiento a partir de las dos mitades, etapa de merge

Complejidad en D&C

- Ad-Hoc / Árbol de Ejecución
- Método de sustitución
- Teorema maestro

Ejemplos de algoritmos/problemas con D&C

- Solución $O(nlogn)$ a maximum subarray
- Karatsuba y Strassen

Continuar leyendo

- D&C sobre árboles
- Programación Dinámica

Grafos

Clase PAP 2017 Melanie https://git.exactas.uba.ar/ltaravilse/pap-alumnos/blob/master/clases/clase03-grafos/grafos.pdf [https://git.exactas.uba.ar/ltaravilse/pap-alumnos/blob/master/clases/clase03-grafos/grafos.pdf]

Un grafo es un conjunto de objetos , llamados vértices o nodos, unidos mediantes líneas llamadas aristas. Se los usa para representar muchas cosas, por ejemplo en arquitectura, los nodos pueden representar habitaciones o espacios, y las aristas pueden significar que los espacios unidos comparten una pared.

También se los usa para analizar redes, donde los nodos son computadores y una arista representa que las computadoras que une están conectadas.

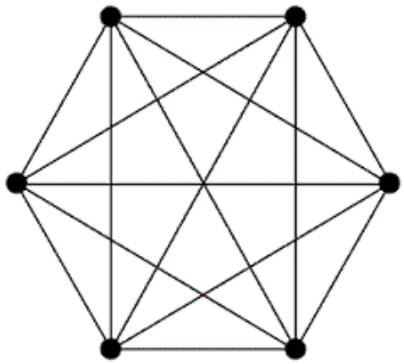
Se los usa para entender y analizar muchíisimas cosas en la vida real, y hay muchos problemas de programación que ilustran situaciones posiblemente reales para los cuales usaremos grafos.

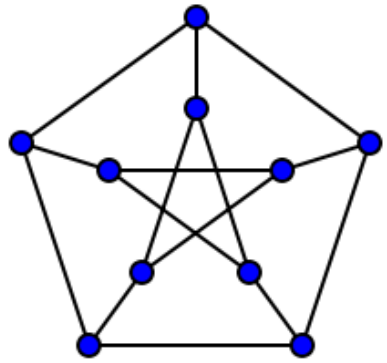
Llamaremos n a la cantidad de nodos y m a la cantidad de aristas.

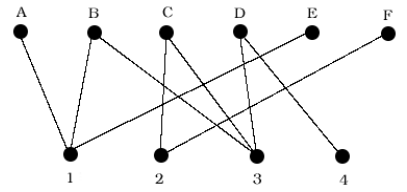
Idea, intuición, dibujitos, ejemplos

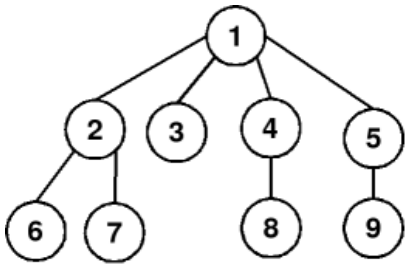
Los grafos sirven para representar relaciones entre distintas "cosas". Esas "cosas", nuestros nodos, pueden ser cualquier cosa, comidas, personas, ciudades... y hasta puede haber algunos nodos que representen un tipo de cosa, y otros nodos que representen otro tipo, por ejemplo nodos que representen personas, y otros que representen comidas, y que la relación sea "a la persona A le gusta la comida B". Las relaciones las representamos con aristas. Hay veces que además de saber que dos nodos están relacionados, nos importa cómo. Las aristas pueden tener un número que indique su distancia por ejemplo, indicando qué tan lejos están esos nodos. También podrían estar orientadas, es decir que no sea sólo una línea, sino una flecha. Esto podría indicar por ejemplo que desde la esquina A se puede llegar a la B a través de una arista, que sería la calle, pero si es de una sola mano no podemos ir a través de ella de B a A, entonces indicamos ese tipo de relación con una flecha.

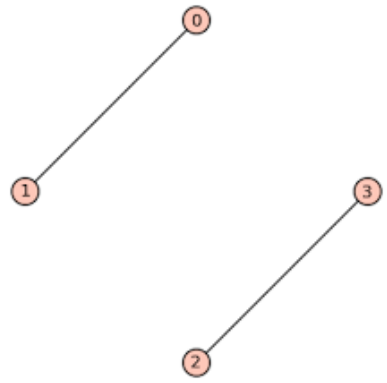
Estos son algunos ejemplos de grafos:











La definición formal dice que un grafo G es un par ordenado (V, E) donde V es un conjunto de nodos, y E es un conjunto de aristas o arcos que unen esos nodos.

Representación en la computadora

Algo muy importante es cómo guardar un grafo en la computadora. En una hoja es fácil, lo dibujamos y listo. Pero y en la compu?

Existen varias maneras de guardar la información de un grafo en la computadora. Las que vamos a ver a continuación son las más populares:

Matriz de adyacencia

La matriz de adyacencia es una matriz de n x n, donde n es la cantidad de nodos, que en la posición j de la fila i, guarda la información de la arista que conecta al nodo i con el nodo j. Es importante notar que deberemos saber cuál es el nodo i, y cuál el nodo j, es decir, debemos asignar números a nuestros nodos para manejarlos mejor.

Entonces, si las aristas del grafo no tienen información, y lo importante es si la arista existe (si el nodo i está unido al j), podemos guardar un 1 si la arista existe, y un 0. Por ejemplo así:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

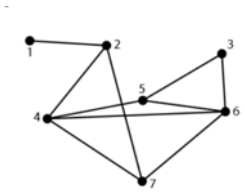
Si las aristas tuvieran longitud supongamos positiva, en vez de unos y ceros, podríamos guardar el valor de la longitud cuando la arista exista, y un 0 si no existe. Si las aristas pueden tomar por ejemplo cualquier valor entre -1000 y 1000, basta con asignar por ejemplo el número 1001 cuando la arista no existe, y así con un if saber si existe o no (dependiendo de si vale 1001 o no).

Se implementa con un vector de 2 dimensiones, $vector < vector < int >> matriz(n, vector < int > (n))$ que en matriz[i][j] hay un 1 si el nodo i está unido al j, y un 0 si no (en el primer caso). También podría ser de $boolean$ en vez de int y guardar true/false.

Ventaja : Permite saber si hay o no (o cuánto mide) una arista entre dos nodos en $O(1)$. Desventaja : La complejidad espacial y temporal para almacenar la matriz es $O(n^2)$.

Listas de adyacencia

Para cada nodo, vamos a guardar una lista con los nodos unidos a él. Si las aristas tuvieran longitud, podemos guardar en vez de los números de nodos, pares de enteros, que indiquen el nodo unido y la longitud de la arista que los une.



$L_1 : 2$
$L_2 : 1 \to 4 \to 7$
$L_3 : 5 \to 6$
$L_4 : 2 \to 5 \to 6 \to 7$
$L_5 : 3 \to 4 \to 6$
$L_6 : 3 \to 4 \to 5 \to 7$
$L_7 : 2 \to 4 \to 6$

Se implementa también como un vector de 2 dimensiones, $vector < vector < int >> grafo(n)$, pero de manera tal que en el vector $grafo[i]$ están todos los nodos que están unidos al nodo i a través de aristas.

Ventaja importante : La complejidad espacial es $O(n + m)$! Tenemos n listas, pero en ellas aparecen exactamente todos los nodos entre los cuales hay aristas. Con la matriz de adyacencia también guardábamos información si no había arista, ahora si no hay arista simplemente no aparecen en la lista. Desventaja : Saber si exista una arista entre i y j implica recorrer toda la lista de i y ver si está j, cuya complejidad es $O(n)$.

¿Por qué es más común utilizar las listas de adyacencia?

Porque lo que más vamos a querer hacer es recorrer un grafo , entonces más que saber si existe una arista entre dos, querríamos "movernos a través de ella", y para eso, desde un nodo, queremos encontrar todas las aristas con un extremo en él, entonces mirar todos los nodos (matriz de adyacencia) es más caro que mirar únicamente los que sabemos que está unidos a él. Para ver más sobre recorrer grafos y problemas típicos, pueden ver las técnicas típicas que son BFS y DFS.

No obstante, es bueno mencionar que las matrices de adyacencia funcionan y siempre podrían utilizarse, aunque el programa podría ser potencialmente muy lento (cuando n^2 sea mucho mayor que m). Si la complejidad no es un inconveniente, suele ser más cómodo usar matriz de adyacencia para todo.

Grafo implícito

Una última representación que vale la pena mencionar es la de grafo "implícito". En este caso, en realidad lo que hacemos es no representar al grafo directamente en memoria. Es decir, no guardamos todos los nodos y todas las aristas, sino que "sabemos" cuáles son, por las características del grafo. Esto depende de cada problema, pero cuando el grafo es algo en particular, suele ser lo más cómodo, sobre todo cuando el grafo es algo que nosotros como programadores decidimos, y no algo que se recibe directamente desde la entrada.

Por ejemplo, supongamos que vamos a trabajar con un grafo que tiene como nodos los números del 1 al 100. Además, sabemos que el grafo tiene una propiedad clave: dos nodos solamente son vecinos, cuando la diferencia entre sus números es 2, 3 o 7. Si

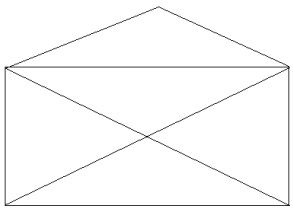
bien podríamos armar toooda la matriz de adyacencia, o toooda la lista de adyacencia de este grafo, en este caso es más fácil directamente no crear esos datos, y directamente usar "la forma del grafo" para saber qué aristas hay.

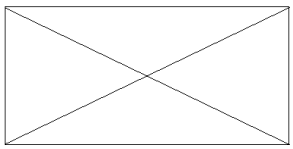
Por ejemplo, la matriz de adyacencia se usa accediendo a posiciones (i, j) para ver si hay arista entre i y j. En nuestro ejemplo, en lugar de usar una matriz, para ver si hay arista entre i y j podemos simplemente hacer la diferencia `abs(j-i)`, y ver si es 2, 3 o 7 para decidir allí mismo si hay arista. Esto podría programarse incluso en una función, `hayArista(int i,int j)`, que funcionaría como si fuera la matriz de adyacencia, pero sin tener que armarla.

Similarmente, el uso típico de las listas de adyacencia es recorrer los vecinos de un cierto nodo. En nuestro ejemplo, los vecinos del nodo x solamente pueden ser $x-2, x-3, x-7, x+2, x+3$ o $x+7$. Basta entonces verificar cuáles de esos números están entre 1 y 100, y tendremos todos los vecinos de x, sin necesidad de armarse todas las listas de adyacencia completas de antemano (aunque podríamos). En casos como este hay incluso trucos comunes para programar más fácilmente estos "movimientos".

Algún ejemplito de problema

Una situación típica que involucra fuertemente teoría de grafos es la que se conoce como "a ver si podés dibujar la figura sin levantar el lápiz ni pasar dos veces por la misma línea". Alguien te da por ejemplo alguna de estas figuras:





Lo que podemos plantear es un grafo en el cual los segmentos rectos son aristas, y sus extremos son nodos.

Entonces, ¿qué tiene que pasar para que podamos dibujar la figura sin levantar el lápiz? Pensemos que empezamos en uno de nuestros nodos. Entonces vamos a ir recorriendo arista por arista, yendo siempre a nodos adyacentes, hasta pasar por todas. Ahora, si el nodo en el que empezamos terminamos, qué significa? Significa que desde ahí, empezamos "saliendo" y terminamos "entrando". Quizás en el medio volvimos a pasar por este nodo, pero si volvimos a pasar, necesariamente fue entrando y saliendo, ya que al no poder levantar el lápiz, no podemos entrar y después recorrer una arista que no contenga al nodo. Entonces, básicamente, recorrimos tantas arista desde el nodo, como hacia él. Lo que podemos deducir de acá, es que la cantidad de aristas que tienen al nodo inicial como extremo, debe ser par. Y qué pasa con todos los otros nodos? Bueno, al no empezar ni terminar en ellos, también entramos y salimos la misma cantidad de veces, por lo que también son extremos de una cantidad par de aristas.

Si en cambio, no terminamos en el mismo nodo con el que empezamos, entonces ambos nodos de inicio y final, deben ser extremos de una cantidad impar de aristas.

Y estas son las únicas dos situaciones posibles, ya que si no empezamos desde un nodo sino desde el medio de una arista, y vamos desde ahí hacia un nodo, luego vamos a tener que hacer la otra parte de la arista y obviamente si podemos, podemos también empezar desde uno de estos nodos y terminar el mismo con el mismo camino.

Entonces, sabemos que es condición necesaria que haya 2 ó 0 nodos que estén en una cantidad impar de aristas. Puede verse, pensando en esto de "salir" y "entrar" del nodo que es una condición suficiente. Esto de "a cuántas aristas pertenece un nodo" se llama el grado de un nodo, y puede verse junto con otras definiciones acá.

Entonces, si vamos a tener un grafo de n nodos, numerados del 0 al $n-1$, y m aristas, que vendrán dadas mediante dos enteros u, v representando a los nodos que une cada arista, podemos verificar si la figura que el grafo representa puede dibujarse

sin levantar el lápiz con este código:

```
#include<bits/stdc++.h>

using namespace std;

int main(){

    int n, m;
    cin>>n>>m;
    vector< vector<int> > grafo(n);
    for(int i=0; i<m; i++){
        int u, v;
        cin>>u>>v;
        grafo[u].push_back(v);
        grafo[v].push_back(u);
    }
    int impares=0;
    for(int i=0; i<n; i++){
        if(grafo[i].size()%2==1){
            impares++;
        }
    }
    if(impares==0 || impares==2){
        cout<<"Se puede dibujar"<<endl;
    }else{
        cout<<"No se puede dibujar"<<endl;
    }
}
```

Dijkstra

Enunciado

Dado un grafo con aristas con longitudes positivas, y un nodo de "comienzo" (al cual llamaremos source por la definición en inglés, o S para resumir), encontrar el camino más corto desde éste nodo hasta todos los demás.

Comentario: la idea de nombrar a algunas cosas según su significado en inglés es que en internet hay muuuucha más documentación y textos para leer sobre esto (y en general, de cualquier cosa) en inglés. Entonces está bueno ir acostumbrándose a algunas definiciones en inglés por si quieren googlear algo.

Idea

La idea es arrancar yendo desde el nodo source al nodo más cercano. Supongamos que el nodo más cercano es el nodo A y está a distancia 4. Puede haber un camino más corto para, desde el source, llegar hasta A?

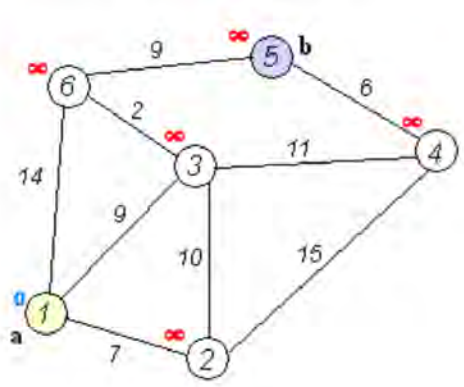
No, porque si lo hubiera, por ejemplo si el camino fuera S->B->A, significa que hay otro nodo (el B) que está a menos distancia de A, pero por definición sabemos que eso no pasa.

También vamos a guardar la información de "hasta este momento, la distancia más corta desde el source hasta el nodo X es D". Entonces si hay una arista desde S hasta B de longitud 7, guardamos ese número.

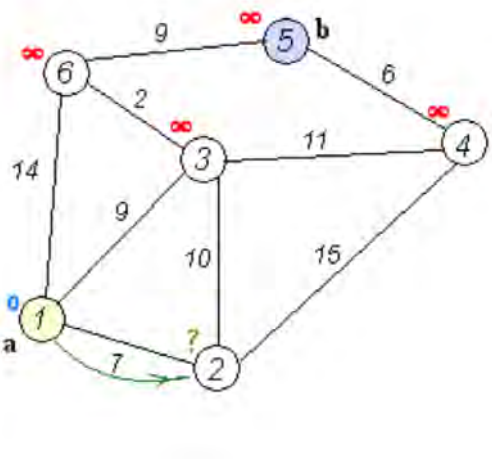
Una vez que guardamos esa información para todos los vecinos de S, agarramos al nodo más cercano hasta S de los que todavía no vimos, en nuestro caso A.

Miramos los vecinos que no visitamos (o sea, a S no lo vamos a mirar), y nos fijamos si la distancia desde S hasta el nodo actual sumado a la longitud de la arista AV siendo V un vecino, es menor que la distancia que teníamos hasta el momento desde S hasta V. Si es más chico, actualizamos el valor como esa suma.

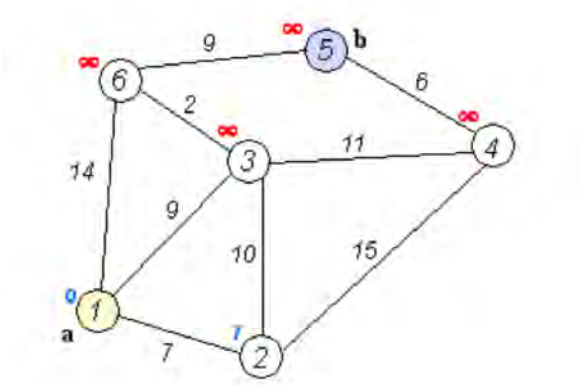
Tenemos este grafo, y queremos ir del nodo $S = a$ (nodo 0, que empieza con distancia 0 y el resto con infinito) hasta el nodo $b = 5$ (imágenes tomadas de esta clase [https://git.exactas.uba.ar/ltaravilse/pap-alumnos/blob/master/clases/clase03-grafos/grafos.pdf] de Melanie Sclar)



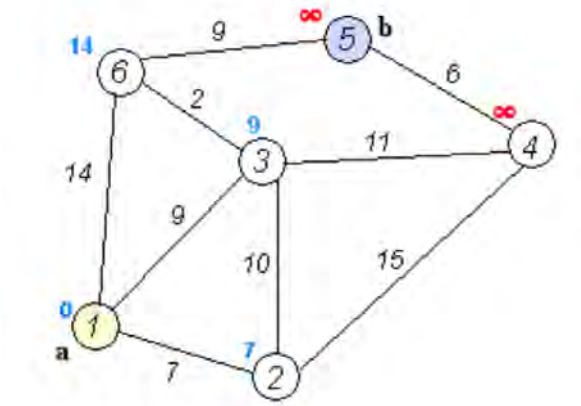
Sacamos al nodo 1 de la priority_queue, que es el nodo más cercano, y vemos qué pasa con los vecinos



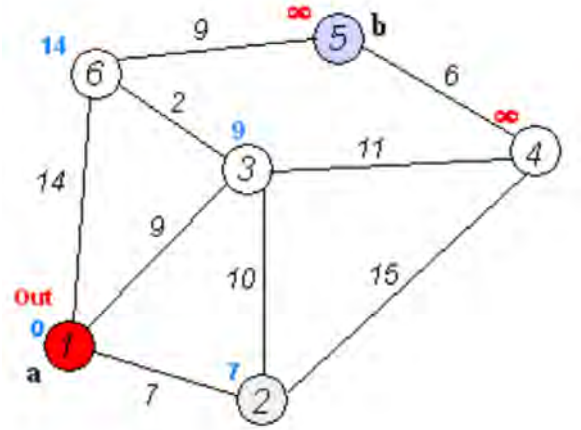
Ir al nodo 2 con distancia $0 + 7$ es mejor (más corto) que con la distancia que teníamos, infinito, entonces actualizamos.



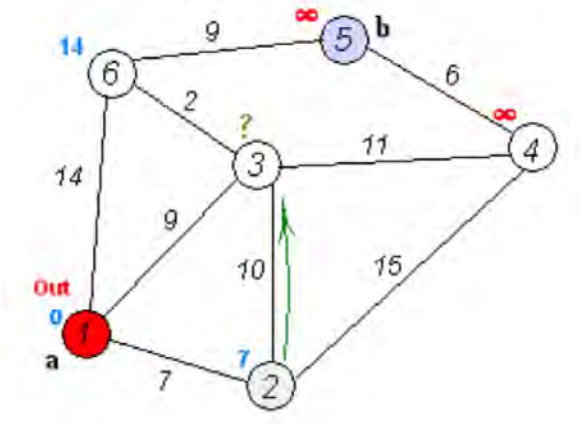
Lo mismo con los nodos 3 y 4, vamos a actualizar con la distancia 0 más la longitud de la arista.



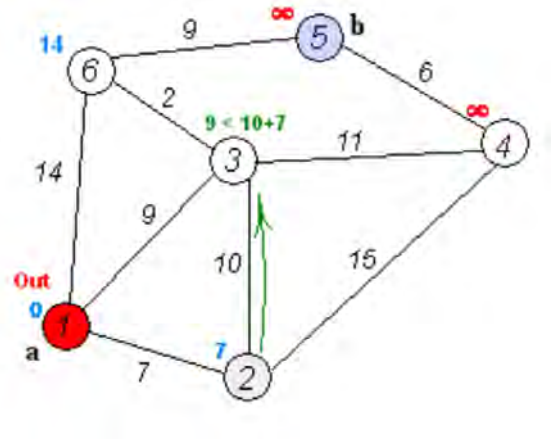
Terminamos de ver los vecinos del nodo 1, lo marcamos como visitado.



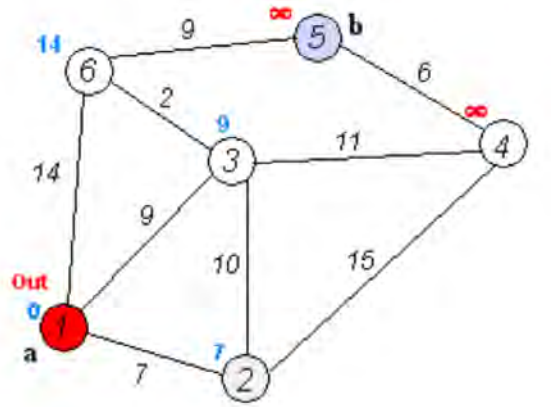
Agarramos al nodo más cercano de los no visitados que tenemos, el 2, y miramos sus vecinos no visitados, a los que todavía podríamos encontrar un camino mejor



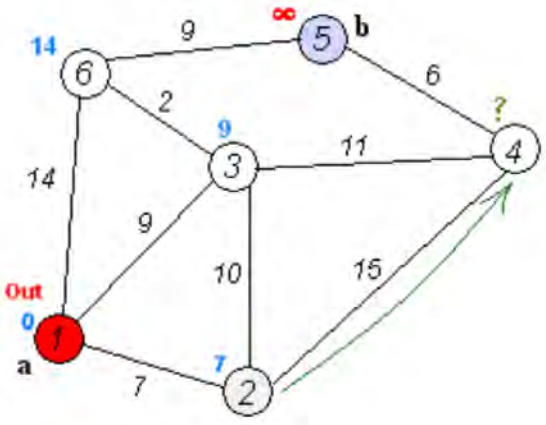
El camino que tenemos hasta el nodo 3 (vecino del 2), es peor que yendo desde el 2 con el camino que tenemos hasta el 2?



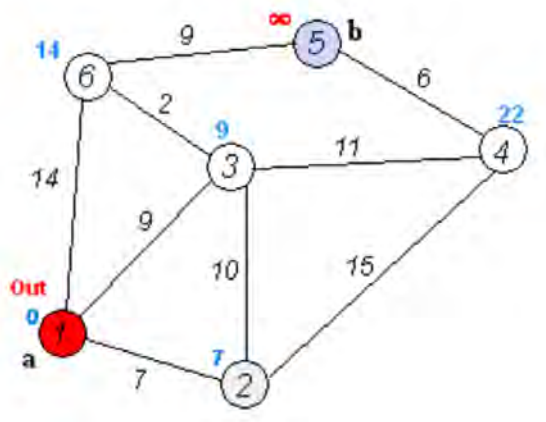
No, entonces, no actualizamos la distancia



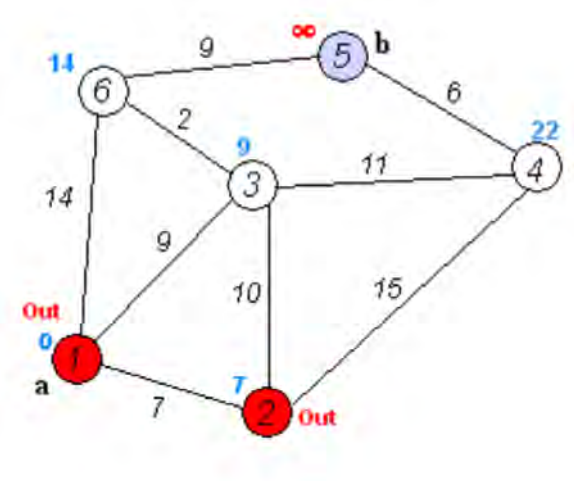
La distancia que tenemos hasta el nodo 4, es peor que yendo desde el nodo 2?



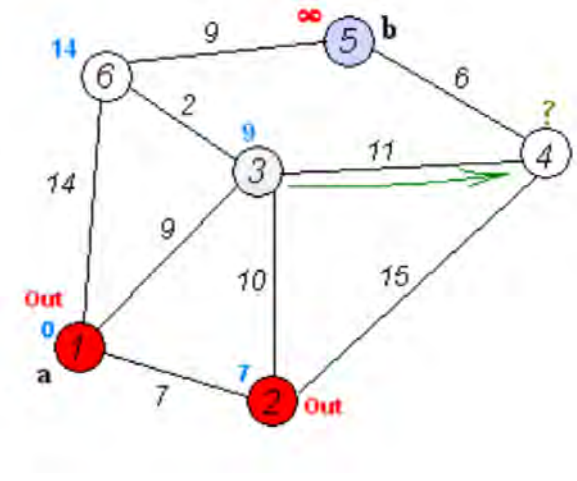
Sí, entonces actualizamos



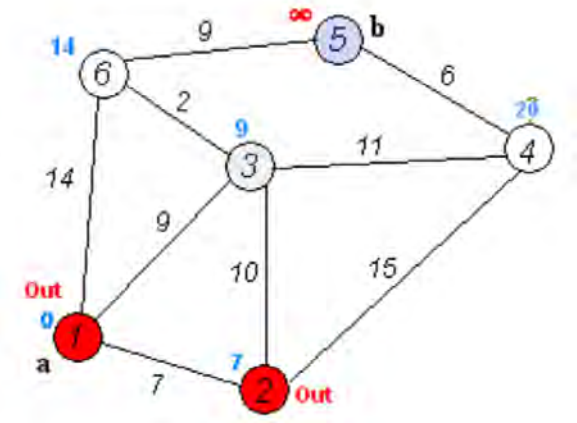
Ya vimos todos los vecinos del 2 sin visitar, lo marcamos como visitado y nos olvidamos de este nodo



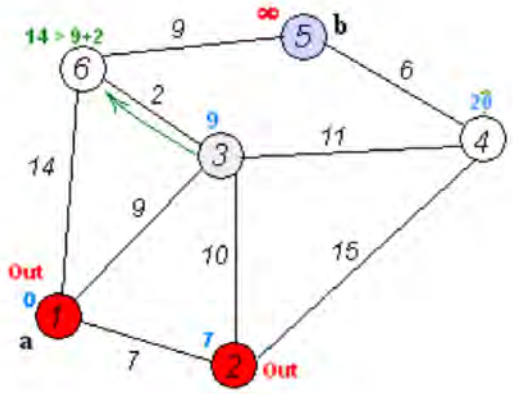
Agarramos al nodo 3, el más cercano de los no visitados hasta acá



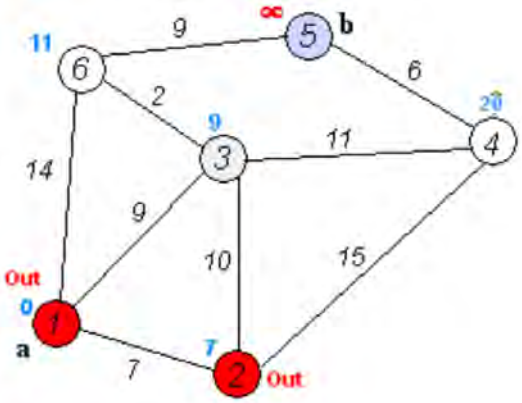
La distancia que tenemos hasta el nodo 4, es peor que ir desde el 3 con la distancia desde el source hasta él, 9? Sí, entonces actualizamos



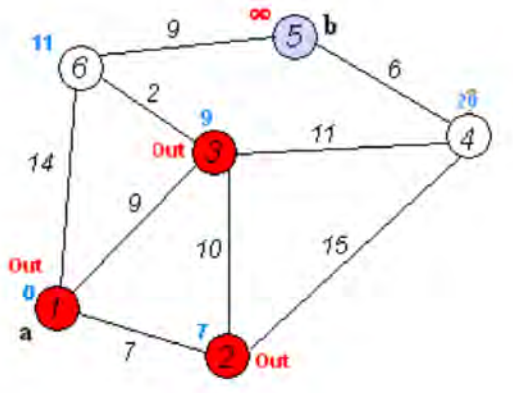
La distancia que tenemos hasta el nodo 6, 14, es peor que ir desde el nodo 3 hasta el 4?



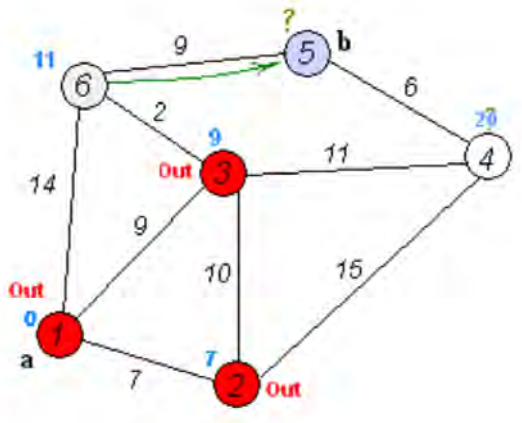
Sí, entonces actualizamos la distancia al nodo 6



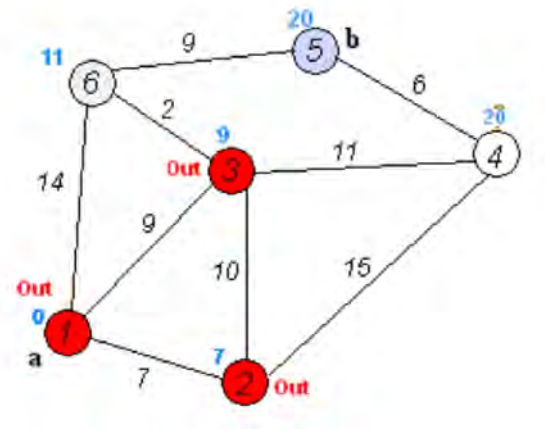
Ya vimos todos los vecinos del nodo 3, entonces lo marcamos como visitado



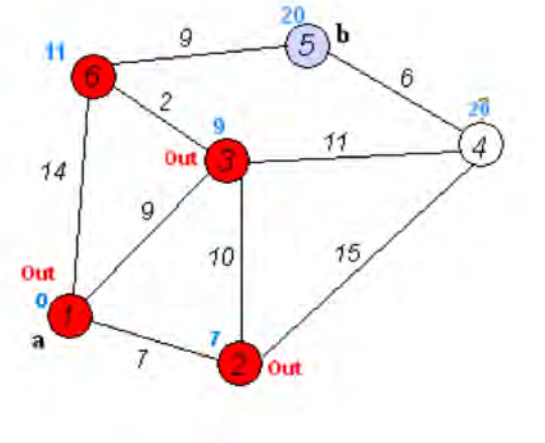
Agarramos al nodo más cercano que tenemos, el 6, y miramos sus vecinos no visitados



La distancia al nodo 5 que tenemos guardada es peor que yendo desde el 6? Sí, entonces actualizamos.



Ya miramos todos los vecinos del 6 sin visitar, lo marcamos como visitado.



Luego, podemos agarrar a cualquiera de los nodos 4 ó 5 (la distancia que tenemos es igual, 20), intentar ir al otro, ver que la distancia es mayor, no actualizar, y terminar el algoritmo.

Más simple pero más lento (O(V^2))

dijkstra-lento.cpp

```
// Tenemos la matriz de vecinos vector< vector<int> > matriz que en matriz[x][y] guarda cuanto mide esa arista, -1 si no existe

int dijkstraLento(int s, int dest){
    int n = matriz.size();
    const int INF = 100000000;
    vector<int> dist(n, INF);
    vector<int> prev(n, -1);
    vector<bool> vis(n, false);

    queue<int> q;
    q.push(s);
    dist[s]=0;
    forn(paso, n){
        int nodoMasCercano=-1;
        forn(i, n){
            if(!vis[i]){
                if(nodoMasCercano==-1 || dist[nodoMasCercano]>dist[i]){
                    nodoMasCercano=i;
                }
            }
        }
        vis[nodoMasCercano]=true;
        forn(i, n){
            int val = dist[nodoMasCercano]+matriz[nodoMasCercano][i];
            if(val<dist[i]){
                dist[i]=val;
                prev[i]=nodoMasCercano;
            }
        }
    }
    // Imprimimos el camino, inverso. Para imprimirlo bien vamos guardando en un vector y lo imprimimos al revés
    int estoy = dest;
    cout<<estoy<<endl;
    while(estoy!=s){
        estoy=prev[estoy];
        cout<<estoy<<endl;
    }
    return dist[dest];
}
```

Implementación un poquito más compleja pero mucho más eficiente (O(E*log(V)))

Ahora, en vez de tener la matriz de dimensiones n x n, vamos tener un grafo que guarde las aristas de cada nodo con sus longitudes. Si no habría que recorrer todos los nodos desde cada uno para ver los vecinos y la complejidad se nos iría al caso anterior.

dijkstra.cpp

```
// Tenemos un vector<vector<pair<int,int> > > grafo donde los elementos del vector i son del estilo (distancia, vecino)

int dijkstra(int s, int dest){
    int n = grafo.size();
    const int INF = 100000000;
    vector<int> dist(n, INF);
    vector<int> prev(n, -1);
    vector<bool> vis(n, false);

    priority_queue< pair<int,int> , vector<pair<int,int> >, greater<pair<int,int> > > q;
    q.push(make_pair(0, s));
    dist[s]=0;
    while(!q.empty()){
        int nodoMasCercano=q.top().second;
        q.pop();
        if(!vis[nodoMasCercano]){
                        vis[nodoMasCercano]=true;
                        forn(i, grafo[nodoMasCercano].size()){
                                int vecino = grafo[nodoMasCercano][i].second;
                                int longitud = grafo[nodoMasCercano][i].first; // si guardaramos (vecino, distancia) esto sería al revés first y second
                                int val = dist[nodoMasCercano]+longitud;
                                if(val<dist[vecino]){
                                        dist[vecino]=val;
                                        prev[vecino]=nodoMasCercano;
                                        q.push(make_pair(val, vecino));
                                }
                        }
                }
    }
    // Imprimimos el camino, inverso. Para imprimirlo bien vamos guardando en un vector y lo imprimimos al revés
    int estoy = dest;
    cout<<estoy<<endl;
    while(estoy!=s){
        estoy=prev[estoy];
        cout<<estoy<<endl;
    }
    return dist[dest];
}
```

Requisitos previos

Grafos

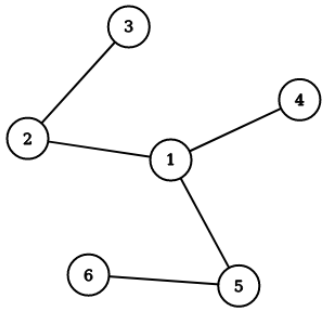
Árboles

Siempre que hablemos de árboles vamos a estar hablando en grafos conexos . Para que un grafo sea un árbol necesita ser conexo, entonces lo daremos por sentado.

Un árbol es un tipo de grafo particular. Una de las particularidades que tiene es que no tiene ciclos. Las siguientes características son equivalente, es decir, si un grafo (conexo) cumple una, cumple las demás:

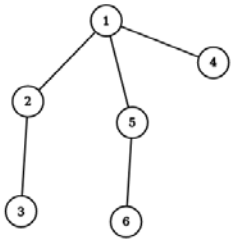
- $E = V - 1$ (la cantidad de aristas es uno menos que la cantidad de nodos)
- No tiene ciclos
- Al agregar una arista entre dos nodos no adyacentes, aparece un ciclo
- Al sacar cualquier arista existente, el grafo se desconecta
- Para cada par de nodos existe un único camino simple (que no repite vértices)

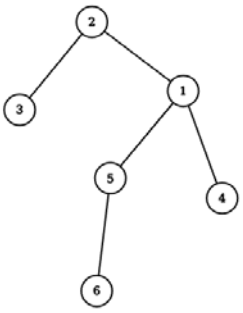
Dejo un grafo de este tipo para tener en la cabeza al seguir leyendo:



Algo interesante que podemos hacer en todos los árboles es establecer un nodo raíz . Si establecemos un nodo como raíz, por ejemplo si elegimos como raíz al nodo 1 ó 2 en el árbol de recién, convendría (y se suele) dibujarlos así respectivamente:

Es decir, se ubica la raíz arriba de todo, y luego los nodos se dibujan más abajo, en la medida en que están más lejos de la raíz.





Para un árbol con raíz, tienen sentido las siguiente definiciones:

Altura o Profundidad de un nodo : Es la longitud del camino entre el nodo y la raíz. En el árbol de raíz 1, las profundidades de los nodos 1, 5, 3 son 0, 1, 2 respectivamente.

Hojas : Son los nodos desde los cuales es imposible ir a otro nodo más alejado de la raíz sin "subir" (acercarnos a la raíz). En el árbol de raíz 1, las hojas son los nodos 3, 4 y 6.

Padre de un nodo : Es el nodo que está inmediatamente arriba de él, es decir que es adyacente al nodo, y está en el camino entre el nodo y la raíz. La raíz no tiene padre. En el árbol de raíz 1, el padre de 6 es 5 y el de 5 es 1.

Hijos de un nodo : Son los nodos adyacentes a él que tienen altura mayor a él. Las hojas no tienen hijos. En el árbol de raíz 1, el único de 2 es 3, y los hijos de 1 son 2, 5 y 4.

Subárbol de un nodo dado : Es el árbol que queda formado por el nodo dado, sus hijos, los hijos de sus hijos, y así siguiendo hasta considerar todos los nodos posibles. La raíz del subárbol es el nodo dado. El subárbol de una hoja es simplemente ese mismo nodo. En el árbol de raíz 1, el subárbol del nodo 5 son sólo los nodos 5 y 6 con la arista que los une.

Desdenciente de un nodo x: es un nodo y que forma parte del subárbol de x. A veces se suele considerar a un nodo descendiente de sí mismo.

Ancestro de un nodo x: es un nodo y tal que x es descendiente de y. A veces se suele considerar a un nodo ancestro de sí mismo.

Ancestro común de un conjunto de nodos : Es un nodo que es ancestro de todos los nodos dados. Es decir, todos los nodos del conjunto forman parte del subárbol del nodo que llamamos ancestro común. La raíz es siempre un ancestro común de cualquier conjunto de nodos. En general es interesante conocer el ancestro común más bajo de dos nodos, que se obtiene tomando, de todos los ancestros comunes, aquel que sea el "más bajo", o sea el que está a mayor profundidad.

Cómo recorrer árboles

Bueno, una manera de recorrerlo es simplemente con BFS o DFS, marcando los visitados y todo igual a como hacíamos antes, ya que eso funciona para cualquier grafo.

Otra manera de recorrerlo es desde la raíz hacia abajo, con una función recursiva que se mueva a todos los vecinos del nodo excepto a su padre, que será un parámetro de la función. Entonces el código quedaría:

```
void recorrer(int nodo, int padre){
    for(int i=0; i<grafo[nodo].size(); i++){
        if(grafo[nodo][i]!=padre){
            // Codigo aca o despues de llamar a la funcion, que haga algo
            recorrer(grafo[nodo][i], nodo); // Ahora el padre del nodo al que vamos es el nodo actual
        }
```

```
    }
}

// Para empezar en la raíz sin que se evite ningún nodo, es común hacer:
recorrer(raiz, -1);
```

Solucionario de la Olimpíada Informática Argentina 2018

Autores

Brian Bokser

Román Castellarin

Sebastián Cherny

Agustín Santiago Gutiérrez

Facundo Martín Gutiérrez

Carlos Miguel Soto

Ariel Zylber

Índice general

Capítulo 1

Introducción

Todos los enunciados de los problemas se encuentran disponibles en: http://www.oia.unsam.edu.ar/problemas-categoria-programacion

Además, se pueden realizar envíos de soluciones para los problemas en el juez online de la olimpíada http://juez.oia.unsam.edu.ar. Se puede entrar directamente la página de un problema particular accediendo a http://juez.oia.unsam.edu.ar/#/task/PROBLEMA/statement, reemplazando el texto `PROBLEMA` por el "código de problema" correspondiente (el nombre corto: `electromovil`, `solitario`, `gps`, etc).

Capítulo 2

Selectivo para la IOI

2.1. Día 1

2.1.1. Problema 1: Auto Eléctrico [electromovil]

http://juez.oia.unsam.edu.ar/#/task/electromovil/statement

Al comenzar nuestro viaje tenemos una cierta autonomía en nuestra batería y nuestro objetivo consiste en alcanzar la última estación de servicio (en la cual podemos asumir que cargamos nuestra batería por simplicidad).

Un primer enfoque posible para resolver el problema utiliza ***programación dinámica*** de la siguiente forma. Llamemos $f(j)$ a la menor cantidad de veces que debemos cambiar la batería para alcanzar la j-ésima estación (incluyendo el cambio de batería en la estación j). Sabemos que $f(0) = 0$ al comenzar nuestro recorrido.

Veamos qué ocurre con la j-ésima estación. Para esto puede sernos útil asumir que ya conocemos el resultado de $f(i)$ para todo $i < j$. A la estación j tuvimos que llegar habiendo cambiado la batería en alguna estación anterior. De todas estas posibiles estaciones anteriores nos interesan solo aquellas cuya *autonomía es suficiente para alcanzar a la estación* j (de no ser así necesariamente tendrá que pasar por una estación por la que sí alcance su autonomía, cuyo caso estamos contemplando).

Concluimos que $f(j) = \min_{i<j}\{f(i)+1\} = \min_{i<j} \overbrace{\{f(i)\}}^{\text{estación que vengo}} + \overbrace{1}^{\text{cambio la batería}}$
Teniendo esta recursión, aplicando programación dinámica (para más información en el tema se puede ver http://wiki.oia.unsam.edu.ar/algoritmos-oia/programacion-dinamica), obtenemos una solución de complejidad $\mathcal{O}(E^2)$, pues por cada estación debemos consultar todas sus anteriores.

Concretamente, si queremos calcular $f(i)$ debemos analizar la situación de todos los $0 \leq i < j$. En particular, solo nos interesan aquellas ubicaciones con `ubicacion[i] + autonomia[i]` $\geq$ `ubicacion[j]`, de todas ellas buscamos algún i_j que minimice $f(i)$. Finalmente $f(j) = f(i_j) + 1$. A continuación podemos ver un ejemplo en la iteración de $j = 4$.

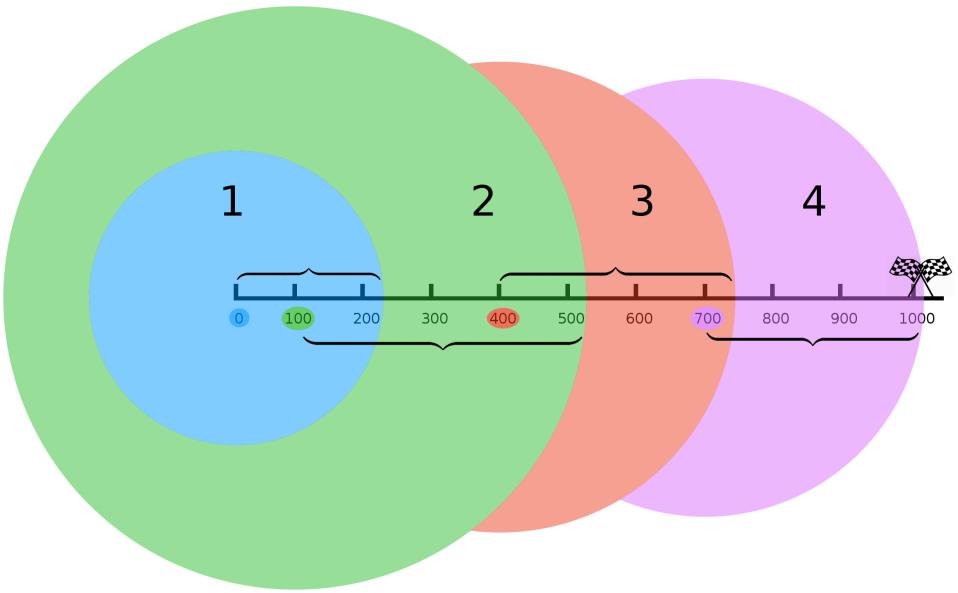
j	0	1	2	3	4	5	6	7	8	9	10
ubicacion[j]	0	100	200	300	400	500	600	700	800	900	1000
autonomia[j]	225	415	150	225	350	125	275	330	225	315	280
$f(j)$	0	1	1	2	2	2	3	3	4	4	4
padre[j]	-1	0	0	1	1	1	4	4	6	6	6

Para finalizar el problema con esta idea, resta ver cómo recalcular el camino. Esto puede hacerse con una idea relativamente estándar que consiste en *almacenar en un arreglo auxiliar dónde se realizó el mínimo* de $f(i)$ dentro de los $i < j$ al resolver el subproblema i. Corresponde a lo que anteriormente llamamos i_j, y en el ejemplo anterior está almacenado en `padre[j] = i`$_j$. A continuación se muestra un pseudocódigo que refleja este enfoque. Se asume que `ubicacion` y `autonomia` son dos arreglos de tamaño $E + 1$ incluyendo el punto de partida.

Algorithm 1 Solución 1 al Problema 1 de Selectivo 2018 Día 1 - electromóvil

```
1: function SORTUCHO(UBICACION: ARR. DE ENTEROS, AUTONOMIA: ARR. DE ENTEROS)(ARR. de ENTEROS)
2:     f ← [∞, ...,∞]                                  ▷ Arreglo de E+1 lugares
3:     padre ← [-1, ...,-1]                            ▷ Arreglo de E+1 lugares
4:     f[0] ← 0
5:     for j = 1 ... E do                              ▷ Recorremos en orden creciente
6:         for i = 0 ... j-1 do
7:             if ubicacion[i] + autonomia[i] ≥ ubicacion[j] then
8:                 if f[i] < f[j] then
9:                     f[j] ← f[i]+1
10:                    padre[j] ← i
11:    solucion ← []
12:    actual ← E
13:    while padre[actual] ≠ -1 do ▷ Usamos E ≥ 2 (y 0 no va en la solución)
14:        solucion.agregar_al_frente(ubicacion[actual])
15:        actual ← padre[actual]
16:    return solucion
```

Dado que en las cotas del enunciado tenemos que $E \leq 1{,}000{,}000$, no esperamos obtener el puntaje total por este problema con este enfoque. Antes de introducir el siguiente enfoque veamos qué ocurre con las estaciones vistas en orden. En un principio podremos llegar a cualquier estación con `ubicacion[j]` $\leq$ `autonomia[0]`. De todas ellas, ¿hay alguna que sea *mejor que otra* o que las *domine* en algún sentido? Sí. De todas las alcanzables, nos conviene tomar aquella que maximice $autonomia[j] + ubicacion[j]$, dado que es la que nos permite llegar más lejos con la misma cantidad de cambios. Hecha esta elección, la situación procede análogamente como se muestra en la figura, y se detalla en el siguiente pseudocódigo.



Algorithm 2 Solución 2 al Problema 1 de Selectivo 2018 Día 1 - electromóvil

```
1: function ELECTROMOVIL(UBICACION: ARR. DE ENTEROS, AUTONOMIA: ARR. DE ENTEROS)(ARR. de ENTEROS)
2:     desde ← ubicacion[0]
3:     hasta_actual ← ubicacion[0]
4:     hasta_proximo ← ubicacion[0]+autonomia[0]
5:     solucion ← []
6:     for j = 1 ... E do                          ▷ Recorremos en orden creciente
7:         if ubicacion[j] > hasta_proximo then    ▷ Imposible de alcanzar
8:             solucion ← []
9:             break
10:        else if ubicacion[j] > hasta_actual then    ▷ Cambio de batería
11:            solucion.agregar_al_final(desde)
12:            hasta_actual ← hasta_proximo
13:        if ubicacion[j]+autonomia[j] > hasta_proximo then    ▷ Siempre vemos si podemos mejorar la mayor distancia que podemos alcanzar
14:            desde ← ubicacion[j]
15:            hasta_proximo ← ubicacion[j]+autonomia[j]
16:    if solucion ≠ [] then
17:        solucion.sacar_del_frente()    ▷ Habíamos agregado ubicacion[0]
18:        solucion.agregar_al_final(ubicacion[E])
19:    return solucion
```

La complejidad de este algoritmo (que aprovecha que la entrada ya está ordenada) es lineal: $\mathcal{O}(E)$.

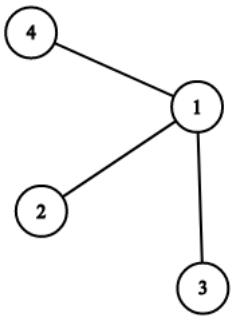
2.1.2. Problema 2: Contando subredes [subredes]

http://juez.oia.unsam.edu.ar/#/task/subredes/statement

Este problema es interesante ya que fue el primer problema "Output-Only" en la OIA. En IOI este tipo de problemas son comunes: el participante tiene disponibles en su computadora para utilizar **todos los casos de prueba**, y solamente envía al juez online los archivos de salida con las soluciones, sin importar el código, lenguaje o herramientas que se hayan usado para generar esas salidas.

El problema en sí consiste en contar la cantidad de subgrafos (subredes) de un grafo (red) G, isomorfos a otro grafo (subred) H dado. En este caso, había 10 archivos de entrada, cada uno correspondía a un H distinto, y cada uno valía 10 puntos. La idea era ver los archivos de entrada para saber cuáles eran estos grafos, y escribir así soluciones específicas para ellos (aunque varias están relacionadas entre sí).

2.1.2.1. $K_{1,3}$



Cada aparición de esta subred determina en forma precisa cuál es el nodo de tipo "1" utilizado para esa aparición, pues es el único de grado 3 en la subred (esta idea de nodos "distinguibles" o "indistinguibles" es fundamental para evitar contar varias veces lo mismo). Por lo tanto, podemos contar todas las apariciones sumando por cada nodo v de la red completa, la cantidad de veces que aparece la subred teniendo a v como el nodo "1" de la figura (el de grado 3).

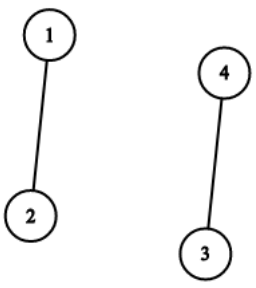
Una vez fijado que v representa al nodo de grado 3, se puede observar que para armar la subred simplemente hay que elegir 3 vecinos distintos de v. Hay exactamente una subred posible por cada elección de 3 vecinos. Como v tiene $d(v)$ vecinos, sumando todo la respuesta para este caso es directamente:

$$\sum_{v \in V} \binom{d(v)}{3} = \sum_{v \in V} \frac{d(v)(d(v)-1)(d(v)-2)}{6}$$

Donde es importante notar que si el grado es menor que 3, el combinatorio correspondiente debería dar 0, pues no hay subred posible para ese v.

Implementativamente, solo hay que calcular los grados en el grafo, lo que se puede hacer en tiempo lineal, y luego hacer la cuenta.

2.1.2.2. $K_2 \cup K_2$



Para fijar una subred en este caso, hay que elegir dos aristas que no se toquen.

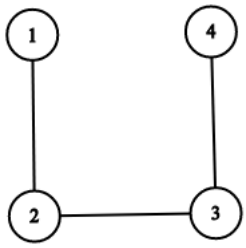
Para contar cuántas posibilidades hay, podemos iterar todas las aristas, y por cada una de ellas, sumar la cantidad de aristas que no la tocan.

Si una arista e une dos vértices de grados d_1 y d_2, entonces toca $(d_1-1)+(d_2-1)$ aristas diferentes (pues a los grados hay que restar uno, correspondiente a la propia arista e). Por lo tanto del total m de aristas, considerando que no están disponibles e ni sus "aristas vecinas", quedan $m-(d_1-1)-(d_2-1)-1 = m-d_1-d_2+1$ aristas disponibles para usar.

Sumando este número para todas las aristas habremos contado todas las subredes posibles, pero habremos contado a cada una dos veces: Una vez al pararnos en una de sus aristas, y otra vez al pararnos en la otra. Por esta razón, la respuesta final será la mitad de esa suma.

De forma similar al caso anterior, para resolver este caso basta con calcular los grados de los nodos en tiempo lineal, y luego recorrer las aristas sumando para hacer la cuenta (y dividir por dos al final), por lo que ambos códigos son casi iguales cambiando levemente la cuenta. Completando ambos se obtenían 20 puntos.

2.1.2.3. P_3

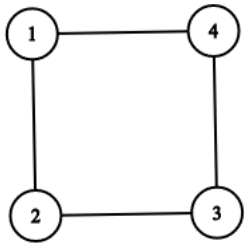


Para contar estas subredes, una posibilidad es notar que la subred determina en forma precisa la arista "central", entre los nodos 2 y 3. Podemos iterar las aristas sumando cuántas subredes de esta forma tienen a la arista e como central.

Fijada la arista e que une los nodos u y v, para completar la subred hay que elegir un nodo vecino de u y uno vecino de v **diferentes entre sí** (pues elegir el mismo formaría un triángulo, que no se corresponde con esta subred). La cantidad de formas de elegir los vecinos en total sería $d_1 \cdot d_2$, pero a eso debemos entonces restar la cantidad de vecinos en común entre u y v.

Si tenemos el grafo representado, por ejemplo, con matriz de adyacencia, es sencillo calcular la intersección entre los vecinos de u y v en tiempo $O(n)$: Basta probar todos los nodos x distintos de u y v, y ver si existen ambas aristas, la x–u y la x–v. Como esto se hace para cada arista, el tiempo total resulta $O(nm)$.

2.1.2.4. C_4



La idea que explicaremos para contar este subgrafo es bastante potente, y de hecho puede utilizarse para resolver también el caso anterior, aunque consideramos que el método ya explicado es más simple de entender y deducir.

La idea es precomputar una tabla de $n \times n$, que para cada par de nodos u, v, indique la cantidad de nodos x (que no sean ni u ni v) tales que existen las aristas x–u y x–v.

El algoritmo para precomputarla es muy simple: Se inicializa la tabla en cero, y luego se recorren todos los nodos. Al procesar x, se toman todos los pares de vecinos distintos u y v, y se suma 1 a la matriz en la posición (o posiciones) correspondiente al par u, v.

Es evidente que este algoritmo para computar la matriz tiene una complejidad de peor caso $O(n^3)$, pues tiene 3 for anidados que recorren hasta n como máximo. Pero de hecho si analizamos con cuidado, podemos demostrar que es $O(nm)$: Esto porque la cantidad total de veces que se iteran los fors interiores será proporcional a la suma total de todos los números en la matriz resultado (ya que en cada paso se incrementa alguno). Pero cada incremento "involucra" dos aristas: x–u y x–v. Y una arista en particular puede estar involucrada en un máximo de $2(n-2)$ incrementos en total, y como hay m aristas, esto hace que el total de incrementos máximo sea $O(nm)$.

Otra forma de demostrar lo mismo es contando la cantidad de pares:

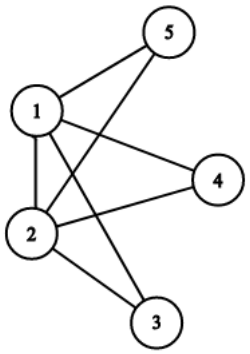
$$\sum_{v \in V} \binom{d(v)}{2} = \sum_{v \in V} \frac{d(v)(d(v)-1)}{2} \leq \sum_{v \in V} \frac{d(v)n}{2} = \frac{n}{2} \sum_{v \in V} d(v) = \frac{n}{2} \cdot 2m = nm$$

Finalmente, una vez que tenemos esa tabla precomputada, podemos observar que si fijamos un par de vértices (u, v) para ser una diagonal del cuadrado, lo que necesitamos para formar el cuadrado son dos vértices distintos que tengan aristas a

u y a v. Pero justamente, la matriz precomputada nos dice esto en la fila u columna v. Así que por cada par de vértices (u, v), si la matriz dice que existen t vértices con la propiedad, debemos sumar $\frac{t(t-1)}{2}$, pues se deben elegir dos de ellos.

Al número final se lo debe dividir por 2, pues como hay dos diagonales en cada cuadrado, los estamos contando doble a todos.

2.1.2.5. $(K_3 \cup K_1 \cup K_1)^c$

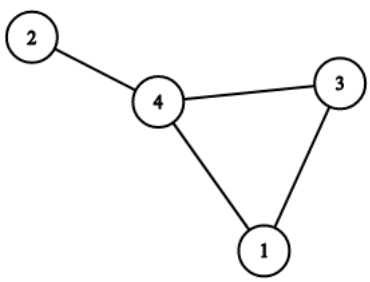


Este caso también puede resolverse con la misma matriz precomputada explicada para el subgrafo anterior: aquí podemos iterar cada para arista (u, v), y contar cuántos subgrafos hay en donde (u, v) corresponden a los nodos 1 y 2 del dibujo. Notar que estos nodos están destacados en la subred, pues son los únicos de grado 4, así que estaremos contando cada subred exactamente una vez.

De forma similar al caso anterior, podemos ver que lo que necesitamos ahora no son 2 sino 3 vértices distintos que tengan aristas tanto a u como a v. El código entonces es idéntico al del caso anterior, pero recorriendo únicamente aristas sumando $\frac{t(t-1)(t-2)}{6}$, y sin la división por 2 al final.

Otra forma posible de resolver este caso es notar que el nodo 1 se conecta a todos los demás, así que entre sus vecinos lo que buscamos es el grafo $K_{1,3}$ del caso 1. Esta misma idea se explica en más detalle para los dos grafos que siguen.

2.1.2.6. $(P_2 \cup K_1)^c$

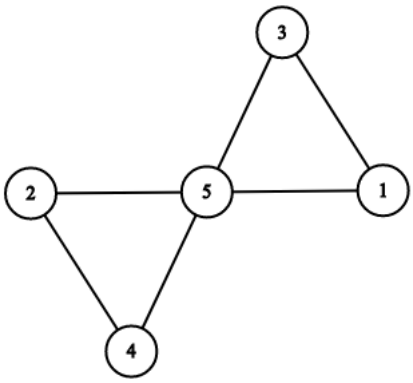


Veamos dos formas posibles de resolver este caso:

La primera se basa nuevamente en la matriz precomputada anterior. Iteramos con dos fors fijando un primero nodo u, que tomará el rol del nodo 4, y un segundo nodo v, que tomará el rol del nodo 1. Para determinar la red una vez fijados u y v, solamente falta elegir qué nodos cumplirán el rol de 3 y de 2. Como el nodo que cumpla el rol de 3 deberá ser vecino de u y v, tenemos en la matriz en la fila u columna v ya computado la cantidad de valores posibles para el nodo 3. Y finalmente, la cantidad de elecciones para el nodo 2 será simplemente $d(u) - 2$, pues en el conteo debemos excluir a los vecinos 1 y 3. Sumando todo esto y dividiendo por 2 (pues 1 y 3 son simétricos así que los contaríamos doble) obtenemos la respuesta.

La segunda se basa en notar que el nodo 4 es muy especial en la subred: se conecta a TODOS los demás. Si lo fijamos en un nodo v, podemos contar entre los vecinos de v en G en busca de una arista, y un nodo que no toque la arista. Si entre los vecinos de v existen t aristas, en total debemos sumar al procesar el nodo v un total de $t(d(v) - 2)$ subredes. Por cada nodo v, se puede calcular la cantidad t de aristas entre sus vecinos en tiempo $O(d(v)^2)$ probando todos los pares de vecinos. Como ya vimos antes, al sumar los cuadrados de los grados de un grafo el resultado es $O(nm)$, así que esta otra solución tiene la misma complejidad que la anterior.

2.1.2.7. $(C_4 \cup K_1)^c$



Podemos aplicar la misma idea de la segunda solución al caso anterior: el nodo 5 es muy particular pues se conecta con todos los demás de la subred. Iteramos entonces todos los posibles nodos v, contando en cada caso la cantidad de subredes que lo utilizan como nodo 5. Para esto, basta con elegir 2 aristas disjuntas entre sus vecinos: pero ya vimos en el caso 2 como puede contarse eso en tiempo lineal, así que lo aplicamos al grafo de vecinos de cada nodo, sumando. Como antes, la complejidad final resulta $O(nm)$

2.1.2.8. K_{10}

Los últimos archivos contienen subredes computacionalmente difíciles. Resolviendo todos los anteriores, ya tendríamos 70 puntos, y lo que sigue es únicamente para obtener los 30 restantes.

El problema general de contar cuántas veces aparece un subgrafo arbitrario dentro de otro es NP-hard, y por lo tanto nadie conoce un algoritmo general polinomial, y los expertos creen que no existe ninguno. Es por esto que estos últimos archivos contienen grafos más pequeños.

La solución para todos ellos es utilizar un buen algoritmo de backtracking, con las podas suficientes para que se pueda ejecutar en el tiempo de la prueba.

Para el caso de prueba en el cual la subred es un grafo completo de 10 nodos, hay que aprovechar la particularidad de esta red para obtener un backtracking eficiente (y probablemente, un poco más simple en su implementación). En este caso, el orden de los 10 nodos elegidos para la subred no es importante, solo importa contar cuántos subconjuntos de 10 nodos existen, tales que todos ellos estén conectados entre sí. Los backtracking generales que distinguen nodos, como los que describiremos en la sección siguiente, tienen un tiempo de ejecución proporcional a la cantidad de *automorfismos* de la subred: es decir, la cantidad de reordenamientos posibles de los vértices, de modo que se vuelve a obtener la misma subred. En el caso particular del grafo completo, todos los ordenamientos valen, así que esos algoritmos de backtracking serían 10! veces más lentos que el que proponemos aquí para este caso (más de tres millones de veces más lentos).

El algoritmo de backtracking propuesto para este caso es ir nodo por nodo, eligiendo si incluirlo o no en el conjunto de nodos de la subred. El estado de la recursión serían dos índices (i, k), que por ejemplo pueden iniciar en $(0, 10)$: i indica cuál es el siguiente nodo a considerar, y k indica cuántos nodos nos faltan elegir en la subred. Si elegimos el nodo i, se pasaría a $(i + 1, k - 1)$ y sino a $(i + 1, k)$.

Las podas fundamentales que proponemos aplicar para que esto funcione mucho más rápido son tres:

- Cuando el grafo de los $n-i$ nodos restantes es completo, se puede simplemente sumar $\binom{n-i}{k}$ a la respuesta, ya que cualquier conjunto posible de k nodos para elegir sirve. Esta situación puede detectarse eficientemente sin aumentar la complejidad asintótica del backtracking, manteniendo un contador de aristas durante el procesamiento del backtracking, pues si $2m' = n'(n' - 1)$ entonces el grafo es completo (donde $n' = n - i$ es la cantidad de nodos restantes, y

similarmente m' son las aristas restantes).

Para mantener eficientemente el conteo de aristas m', se puede mantener el grado de cada nodo en el grafo de los $n - i$ nodos restantes, y entonces $2m'$ será la suma de esos grados.

- Si entre los nodos restantes hay alguno cuyo grado restante es menor que $k-1$, estamos seguros de que no sirve para el subgrafo buscado, y podemos borrarlo. Esto altera el grado de sus vecinos, reduciéndolo en 1, y por lo tanto podríamos eliminar varios nodos en cascada en cada paso mediante este procedimiento.
- Intentar ordenar los nodos que se van examinando por grado. Esto es porque al elegir un nodo para la subred, se descartan a todos los que no son vecinos, así que un nodo con poco grado descarta a casi todos y por lo tanto reduce mucho el espacio de búsqueda.

Notar que en la transición del backtracking, es fundamental marcar como borrados (y en particular, actualizar los grados) aquellos nodos que no se conectan a alguno de los ya elegidos, de modo de no tener que verificarlo en cada paso, y reduciendo enormemente el espacio de búsqueda restante.

Esta forma de enumerar las subredes de 10 nodos es más que suficientemente eficiente para el problema en cuestión, terminando en uno o dos minutos. Otros backtrackings con menos podas podrían utilizarse para igualmente obtener los 10 puntos de este caso, pero tardan más en ejecutarse.

2.1.2.9. Grafos aleatorios

Los últimos dos casos (20 puntos) tienen grafos complicados aleatorios / arbitrarios. Para estos grafos no podemos aplicar el backtracking del paso anterior, pues los nodos no son todos equivalentes, y es importante tener en cuenta qué nodo del grafo representa qué nodo de la subred. Estos son los casos más difíciles de resolver de todo el problema.

El estado fundamental a mantener en este caso es más complejo, pues no bastan dos índices, sino que hay que saber el **conjunto** de nodos de la subred que falta asignar (no solamente su cantidad), y los nodos del grafo disponibles a los que pueden ser asignados.

Al igual que comentamos en el caso anterior, examinar los nodos en orden creciente de grado reduce notoriamente el espacio de búsqueda, pues los nodos con poco grado son heurísticamente los que más restringen los subsiguientes posibles nodos que elijamos (específicamente, generan mayor *cantidad* de restricciones).

Alternativamente, una idea adicional permite aumentar muchísimo la eficiencia: se pueden mantener listas o conjuntos de nodos, **uno por cada uno** de los 10 nodos de la subred. Cada uno de estos conjuntos indica **todos** los nodos del grafo que todavía es posible que se correspondan con el nodo de la subred asociado a ese conjunto.

Inicialmente, estos conjuntos contienen todos los nodos del grafo, pero cada elección que se realice filtra las listas. Si alguna se vacía, se puede podar, pues no ha quedado ningún nodo posible para "cumplir el rol" de ese nodo de la subred. En cada paso, se elige la lista más chica, y se prueban todas esas posibilidades para el nodo que tomará ese rol, filtrando las listas correspondientes a vecinos en la subred, eliminando aquellos nodos que no sean vecinos del nodo elegido en el grafo.

Como antes, un backtracking que use todas estas ideas de podas ejecuta sumamente rápido para los casos de prueba, pero no es estrictamente necesario realizar exactamente esta solución: un backtracking diferente, o uno similar con menos podas, puede obtener también la solución en el tiempo de prueba, aunque posiblemente demore más tiempo en ejecutarse.

Un detalle final es que si hay nodos indistinguibles, el algoritmo anterior (y casi cualquier backtracking razonable) cuenta cada subred múltiples veces (exactamente una vez por cada automorfismo de la subred), pues cuenta todas las formas diferentes de hacer corresponder nodos de la subred con nodos del grafo.

La solución es dividir por la cantidad de posibilidades existentes. En el caso particular de los casos de prueba, hay solo 2 automorfismos, así que basta con poner una división por dos en la respuesta final. Como la subred tiene solamente 10 nodos, la cantidad de automorfismos puede contarse fácilmente en $O(n!)$ utilizando por ejemplo el siguiente fragmento de código:

```
automorfismos = 0;
int permu[10];
for (int i=0;i<10;i++)
    permu[i] = i;
do
{
    bool ok = true;
    for(int i=0;i<10;i++)
    for(int j=0;j<10;j++)
        ok &= subred[i][j] == subred[permu[i]][permu[j]];
    automorfismos += ok;
} while (next_permutation(permu, permu+10));
```

2.1.3. Problema 3: Cartas [solitario]

http://juez.oia.unsam.edu.ar/#/task/solitario/statement

Conviene separar este problemas en dos partes principales: la de descubrir la *posición* en el mazo inicial, de la carta que queda al final, y luego la de *identificar* qué carta hay en esa posición.

2.1.3.1. Posición

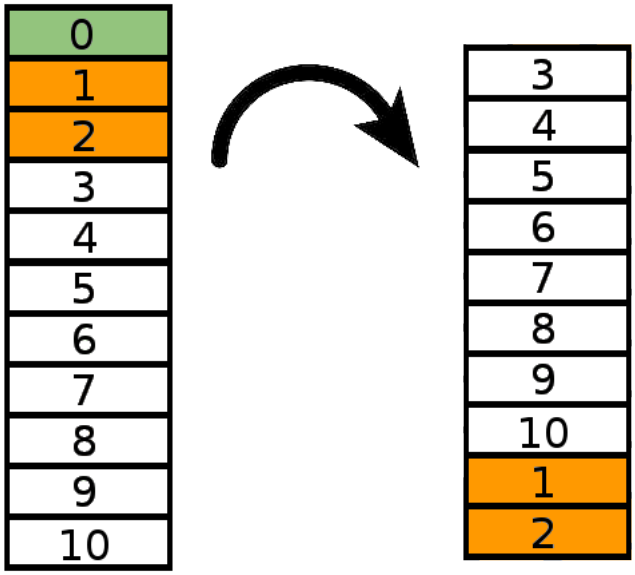
Supongamos que tenemos un mazo de N cartas numerado, de manera que la carta de arriba del mazo es la carta 0, la siguiente es la carta 1, y así hasta la carta de abajo de todo, que será la carta $N-1$.

Nuestro objetivo será computar $f(K, N)$, el número de carta que queda al final si realizamos el procedimiento indicado.

Esta función f puede calcularse por **simulación**: Podríamos por ejemplo tener un arrego que represente el mazo, con una carta por cada posición en el arreglo, y realizar todas las operaciones indicadas hasta que quede un arreglo de tamaño 1 con la carta final. Si se utiliza una cola de dos puntas eficiente (por ejemplo utilizando la estructura de datos `deque` en C++), esto puede realizarse con complejidad $O(NK)$ (ya que el procedimiento tiene K pasos por cada carta que se elimina, y hay N cartas).

Veamos a continuación como calcular f de un modo más "matemático", sin simular directamente el procedimiento.

Si pensamos en el primer paso, por ejemplo con $K = 2$ y $N = 11$ tenemos una situación como la que sigue:



La carta superior, que siempre es la 0, se indica en verde, y esta carta es **descartada** completamente del mazo.

Por otro lado, las siguientes $K = 2$ cartas, que estarán numeradas desde 1 hasta K, se indican en naranja: estas cartas pasan a la parte de abajo del mazo, como resultado de repetir K veces la acción de "pasar la carta de arriba a la parte de abajo".

El procedimiento anterior tiene una salvedad: Si fuera $K \geq N$, no sería cierto que simplemente se pasan hacia abajo las cartas de 1 a K, ya que al dar una vuelta entera al mazo se volverían a pasar por las mismas cartas. La observación aquí es que realizar la acción de pasar carta $N - 1$ veces seguidas no tienen ningún efecto (porque al haber pasado por todas las $N - 1$ cartas restantes, el mazo vuelve a quedar exactamente como estaba). Podemos entonces quedarnos con el resto $K' = \texttt{K\% (N-1)}$, que será un número entre 0 y $N - 2$ de cartas naranjas que sí darán una situación como la del dibujo de arriba.

La clave de esto es que luego del primer paso, **tenemos un mazo más chico**: Este mazo tiene únicamente $N - 1$ cartas. Por lo tanto, $f(K, N - 1)$ indicaría por definición cuál es la posición de la carta de este segundo mazo que sobrevive como resultado del procedimiento. Esto da lugar a una solución recursiva: interpretando esta posición $f(K, N - 1)$ dentro de este segundo mazo, según esa posición caiga en la parte blanca o en la parte naranja, podremos calcular finalmente cuál es el número de carta sobreviviente en todo el proceso de las N cartas, incluyendo el primer paso también.

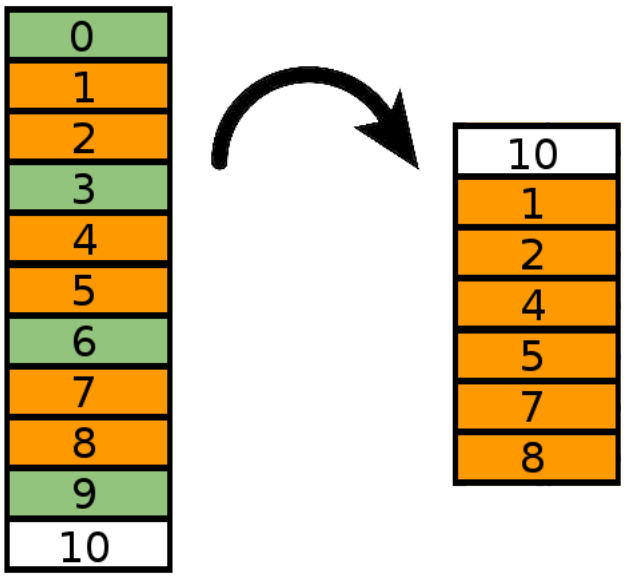
Esto da lugar a la siguiente función recursiva:

$$f(K,N) = \begin{cases} 0 & \text{si } N = 1 \\ \text{Ponemos: } K' = K \bmod (N-1) & \\ K' + 1 + f(K, N-1) & \text{si } f(K,N-1) < N - K' \\ K' + 1 + f(K, N-1) - N & \text{sino} \end{cases}$$

Notar la diferencia entre K y K'.

Como hay N números entre 1 y N, y para cada uno de esos valores la función llama al anterior, en total se calculan N valores de la función y la complejidad de este método es $O(N)$.

Podemos aún mejorarlo para obtener la solución esperada de 100 puntos: Observemos que si en lugar de imaginarnos únicamente el primer paso, nos imaginamos 2 pasos adicionales, tendríamos una situación como la que sigue:



Tenemos que todas las cartas cuya posición es múltiplo de $K+1$ son cartas verdes que son **descartadas** del mazo. De esta manera, con este método estaremos descartando $T = 1 + \left\lfloor \frac{N-1}{(K+1)} \right\rfloor$ cartas a la vez, todo en un único análisis de varios pasos seguidos. Notar que si $K = 0$ estaremos descartando todas las cartas y nos queda un mazo vacío, así que lo correcto será separar ese caso para los cálculos (aprovechando que $f(0,N) = N-1$).

La posición final de la carta en el mazo resultante de la derecha podemos

obtenerla, como antes, llamando a $f(K, N-T)$, y solo resta implementar la lógica para determinar cuál es el número de carta en el mazo original de la carta que está en esa posición del mazo de la derecha. Para esto, observemos que el mazo de la derecha se compone primero de una sección de cartas blancas, formadas por todas las que sobraron al final, que son exactamente $S = (N-1) \bmod (K+1)$. Y luego vienen las cartas naranja, que internamente se encuentran divididas en bloques de tamaño exactamente K: el primero tiene las cartas 1 a K, el segundo las cartas $1+K+1$ a $K+K+1$, y en general el i-ésimo (contando desde 0) tendrá las cartas $1+i(K+1)$ hasta $K+i(K+1)$.

Todo el análisis nuevo anterior será para el caso en que $K < N-1$, ya que sino se dará toda la vuelta al mazo, y entonces solo podremos sacar de a una carta exactamente como hacíamos antes.

$$g(K,N) = \begin{cases} 0 & \text{si } N = 1 \\ N-1 & \text{si } K = 0 \\ f(K,N) & \text{si } K \geq N-1 \\ \text{Ponemos: } S = (N-1) \bmod (K+1) & \\ \text{Ponemos: } T = 1 + \left\lfloor \frac{N-1}{(K+1)} \right\rfloor & \\ N - S + g(K, N-T) & \text{si } g(K,N-T) < S \\ \left\lfloor \frac{g(K,N-T)-S}{K} \right\rfloor + 1 + g(K,N-T) - S & \text{sino} \end{cases}$$

Notar que esta nueva función g más eficiente utiliza la anterior f cuando el valor de N se vuelve suficientemente pequeño, ya que si el K es grande en relación al N y se da una vuelta entera al mazo en cada paso, el cálculo como está en g no es correcto.

Se puede probar que la complejidad de este nuevo método es $O(K \ln N)$, muchísimo más eficiente para valores grandes de N gracias a que se descartan muchísimas cartas en cada paso.

2.1.3.2. Identificación de la carta

Esta es la parte más fácil de las dos en que se divide el problema. Como el mazo es copia de muchos mazos pequeños pero todos iguales, solo importa la posición de la carta final dentro de su mazo pequeño correspondiente.

Sabemos que la carta que sobrevive es $f(K,N) = g(K,N)$ que calculamos en la parte anterior. Pero ese número de carta es **desde la parte superior del mazo**. Los montones se arman de abajo hacia arriba, así que es más cómodo en realidad

saber la ubicación desde abajo (pero para todos los cálculos anteriores, sí era mucho más cómodo contar desde arriba). Entonces tenemos $p = N - 1 - f(K, N)$ que será la posición de la carta pero contada desde la parte de abajo del mazo.

Dado este p, como las cartas 0 a $M - 1$ (contando desde abajo) forman la primera copia del mazo, las M a $2M - 1$ forman la segunda, y así siguiendo, lo único que importa es p mod M, ya que eso indica la posición de la carta que sobrevive dentro de su correspondiente copia del mazo pequeño.

La respuesta final por lo tanto será `c[p%M]`, donde c es el arreglo donde se indican las M cartas que forman el mazo pequeño que es copiado muchas veces para obtener las N cartas totales.

2.2. Día 2

2.2.1. Problema 1: GPS anticongestión [gps]

http://juez.oia.unsam.edu.ar/#/task/gps/statement

El problema nos pide encontrar las longitudes de los k caminos más cortos entre dos nodos fijos en un grafo dirigido con pesos.

Un primer enfoque posible es comenzar con $k = 1$. ¿Cómo encontramos la distancia más corta entre dos nodos en un grafo con pesos? Para resolver este caso, empleamos el algoritmo de *Dijkstra*, que se explica en el siguiente link: http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/dijkstra.

En resumen, el algoritmo de Dijkstra se puede pensar como "explorar todos los caminos y ver cuál es más corto", pero con una optimizacion que evita mirar todos los (posiblemente infinitos) caminos. Esta optimizacion consiste en explorar, por cada nodo del grafo, sólo *uno* de los caminos que pasa por ese nodo. Para que el algoritmo siga siendo correcto, además, este camino que exploramos debe ser el *camino óptimo* hasta ese nodo, es decir, el que tiene menor longitud.

Implementativamente, esto se resuelve manteniendo un conjunto de "candidatos" a caminos óptimos. Si en cada paso exploramos el candidato más corto que termina en un nodo todavía no explorado, es fácil ver que ese candidato a camino óptimo era, en realidad, el camino óptimo. Una implementación del algoritmo sería así:

Algorithm 3 Implementación de Dijkstra

```
1: function DIJKSTRA(GRAFO ARR. DE ARISTAS, NODO_INICIAL ENTERO, NODO_DESTINO ENTERO)(ENTERO)
2:     visitado ← [No, ..., No]
3:     candidatos ← {}
4:     camino_inicial.longitud ← 0
5:     camino_inicial.nodo ← nodo_inicial
6:     candidatos.añadir_nuevo_camino(camino_inicial)
7:     distancia_a_nodo_final ← ∞
8:     while no candidatos.vacío() do
9:         longitud, nodo ← candidatos.camino_más_corto()
10:        candidatos.borrar_camino_más_corto()
11:        if no visitado[nodo] then
12:            visitado[nodo] ← Sí
13:            if nodo es nodo_destino then
14:                distancia_a_nodo_final ← longitud
15:            for peso, hijo en grafo[nodo] do
16:                camino.longitud ← peso+longitud
17:                camino.nodo ← hijo
18:                candidatos.añadir_nuevo_camino(camino)
19:    return distancia_a_nodo_final
```

Implementando este algoritmo sin más, podemos obtener `15` puntos de la primer subtarea ($k = 1$), pero para resolver el problema completo vamos a tener que generalizar el algoritmo de Dijkstra.

Como dijimos, la optimización en la que se basa Dijkstra es explorar solamente el camino óptimo que pasa por cada nodo. Pero al tratar de hacer esto en nuestro problema estamos descartando información importante acerca de las demás formas de llegar a ese nodo. Por esto, debemos explorar, por cada nodo, los k caminos más cortos que pasan por el mismo.

Implementativamente, esto es sólo una pequeña modificación sobre el dijstra común y corriente. En lugar tener un arreglo que nos dice si un nodo ya fue explorado por el algoritmo, debemos mantener un arreglo que nos diga cuántas veces fue explorado cada nodo, y seguimos explorando caminos por cada nodo hasta que este haya sido explorado k veces.

Además, para devolver la respuesta final, debemos anotar la longitud de cada camino que exploramos que termina en el nodo final. La implementación podría quedar así:

Algorithm 4 Dijkstra modificado, que resuelve el problema

```
1: function DIJKSTRA_MODIFICADO(GRAFO ARR. DE ARISTAS, NODO_INICIAL ENTERO, NODO_DESTINO ENTERO, K ENTERO)(ARR. de ENTEROS)
2:     veces_visitado ← [0, ..., 0]
3:     candidatos ← {}
4:     camino_inicial.longitud ← 0
5:     camino_inicial.nodo ← nodo_inicial
6:     candidatos.añadir_nuevo_camino(camino_inicial)
7:     distancias_a_nodo_final ← []
8:     while no candidatos.vacío() do
9:         longitud, nodo ← candidatos.camino_más_corto()
10:        candidatos.borrar_camino_más_corto()
11:        if visitado[nodo] <k then
12:            veces_visitado[nodo] ← veces_visitado[nodo]+1
13:            if nodo es nodo_destino then
14:                distancias_a_nodo_final.poner_al_final(longitud)
15:            for peso, hijo en grafo[nodo] do
16:                camino.longitud ← peso+longitud
17:                camino.nodo ← hijo
18:                candidatos.añadir_nuevo_camino(camino)
19:    return distancias_a_nodo_final
```

La complejidad de este algoritmo es $\mathcal{O}(Mk \cdot \log Mk)$, ya que cada nodo se revisa k veces, y por cada una de estas veces se revisan, entre todos los nodos, un total de M aristas, y cada una de estas conlleva a una operación de "añadir nuevo camino", que toma $\mathcal{O}(\log$ `candidatos.size()`), y `candidatos.size()` está acotado a su vez por la cantidad de inserciones (que es, como ya dijimos, $\mathcal{O}(Mk)$)

Esta complejidad es suficiente para resolver el problema sin restricciones.

2.2.2. Problema 2: Secuenciando el ADN [secuenciando]

http://juez.oia.unsam.edu.ar/#/task/secuenciando/statement

El problema nos pide descubrir una secuencia secreta de la que solo sabemos la longitud N y las posibles letras que la componen. Para ello podemos hacer preguntas sobre si una secuencia es subsecuencia de la secreta a lo que nos responderán sí o no. El objetivo es descubrir la secuencia secreta usando pocas preguntas.

Ls primera idea de la solución es darnos cuenta que podemos saber la cantidad de veces que aparece una letra con pocas preguntas. Por ejemplo, si preguntamos por la secuencia *aaaaaaaaaa* de 10 letras a, la respuesta es positiva si la secuencia secreta tiene al menos 10 letras a y negativa si hay menos de 10. Esto nos dice que podemos hacer una busqueda binaria para saber cuantas a aparecen en la secuencia secreta.

Si queremos saber si hay al menos k letras a, preguntamos por la secuencia de k letras a consecutivas. Y esto funciona para cualquiera de las letras de la palabra. Por lo tanto en $log(N)$ preguntas podemos descubrir la cantidad de veces que aparece una letra.

Con esta idea podemos resolver la primera subtarea, ya que $log(N)$ es a lo sumo 10. Por lo que con 10 preguntas podemos averiguar la cantidad de a que hay y sabemos que el resto de las letras tienen que ser c y como todas las a vienen antes que las c, queda determinada la secuencia secreta. Por ejemplo, si sabemos que la longitud de la secuencia secreta es 10 y con las preguntas averiguamos que hay 4 letras a, las otras 6 tienen que ser c y la secuencia es *aaaacccccc*.

Para el resto de las subtareas se complica porque si bien podemos saber cuantas veces aparece cada letra, tenemos que descubrir como se intercalan las distintas letras de la secuencia. La segunda idea es que si sabemos cuantas letras a y cuantas b hay, por ejemplo, con algunas preguntas más podemos averiguar como se intercalan. La idea será ir averiguando cuantas a aparecen antes de todas las b y cuantas hay entre cada dos b que aparecen. Supongamos que hay 4 letras a y 6 letras b, y la palabra secreta es *abbaabbabb*, la estrategia para averiguarla será la siguiente:

- Preguntamos por la secuencia *abbbbbb*, una letra a y todas las b. Nos dicen que sí es porque hay una a que aparece antes que todas las b.
- Preguntamos por la secuencia *aabbbbbb*, agregando una a al principio. Nos dicen que no porque hay sólo una letra a antes que todas las b.
- Preguntamos por la secuencia *ababbbbb*, como nos dijeron que no, sacamos la a y la ponemos despues de la primera b. Nos dicen que no porque la próxima a aparece despues de la segunda b.
- Preguntamos por la secuencia *abbabbbb*, ahora nos responden que sí.
- Preguntamos por la secuencia *abbaabbbb*, agregando una a despues de la última que pusimos. Nos vuelven a decir que sí.
- Y así siguiendo, cada vez que nos responden que sí, agregamos una a más a la derecha de la última a que pusimos y cada vez que nos dicen que no sacamos la última a y la movemos después de la próxima b.

Notar que no hace falta preguntar cuantas a aparecen después de la última b porque sabemos cuantas a hay en total y por lo tanto al final tienen que aparecer todas las a que faltaron. Además, si ya ubicamos todas las a, no hace falta seguir preguntando,

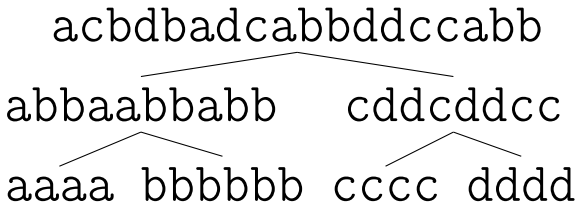
sabemos que todas las que siguen son b. Si la a aparece n_a veces y la b aparece n_b, el proceso termina:

- O bien cuando ubicamos todas las a. Notar que ubicamos una a cada vez que la respuesta es sí. Es decir cuando tenemos n_a respuestas afirmativas.
- O bien cuando ya averiguamos las ubicaciones de todas las a que aparecen antes de la última b. Como avanzamos una b cada vez que nos responden que no, esto quiere decir que nos dieron n_b respuestas negativas.

Esto quiere decir que como mucho hacemos $n_a + n_b - 1$ preguntas, ya que si hicimos esa cantidad de preguntas o bien tuvimos n_a respuestas afirmativas o n_b respuestas negativas.

Con esto podemos resolver las siguientes dos subtareas. Con $log(N)$ preguntas (a lo sumo 10) averiguamos cuantas g aparecen y con esto deducimos cuantas t aparecen. Luego, si la g aparece n_g veces y la t aparece n_t veces, con $n_g + n_t - 1$ preguntas más descubrimos la secuencia secreta. Notar que $n_g + n_t - 1 = N - 1$, por lo que hacemos a lo sumo 999 preguntas más.

Las siguientes subtareas pueden resolverse con esta misma técnica, pero utilizándola para "fusionar" distintas palabras parcialmente encontradas. En el caso anterior, usábamos la búsqueda binaria para tener como palabras iniciales *aaaa* y *bbbbbb*, y luego con la técnica de ir intercalando las a en diferentes posiciones sucesivas, podíamos fusionarlas con un costo total n_a+n_b-1, obteniendo la secuencia *abbaabbabb*.



Si la palabra original en cambio hubiera sido *acbbacabbccabb*, hubiéramos obtenido la misma palabra anterior al fusionar las a y las b, pero además sabríamos que la palabra tiene 4 letras c. Con lo cual, podemos hacer un paso más análogo al anterior de ir probando donde insertar la siguiente letra, para fusionar *abbaabbabb* con *cccc*. Como antes, una fusión entre dos palabras de longitudes l_1 y l_2 cuesta $l_1 + l_2 - 1$.

Si vamos fusionando todas las letras de a dos en forma de árbol binario balanceado, al estilo del algoritmo de Mergesort, se puede corroborar que el total

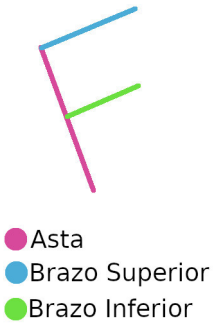
final de preguntas a realizar durante las fusiones no excede nunca $N \lg N$ preguntas (ya que al igual que en mergesort: hay $\lg N$ niveles de fusiones, y el costo total en cada nivel es $O(N)$). Esto es suficiente para resolver el problema.

Una estrategia aún mejor en casos con cantidades de letras desbalanceadas es fusionar siempre las dos palabras más cortas (lo cual tiene relación con los códigos de Huffman), pero no era necesaria esta optimización para resolver el problema (y en el peor caso, que ocurre cuando todas las letras aparecen la misma cantidad de veces, no se gana nada comparado a la solución balanceada anterior).

2.2.3. Problema 3: Buscando la F [buscandof]

http://juez.oia.unsam.edu.ar/#/task/buscandof/statement

El problema nos pide, dadas las coordenadas de ciertos puntos en el plano, hallar la máxima cantidad de estos puntos que pueden pertenecer a una efe.

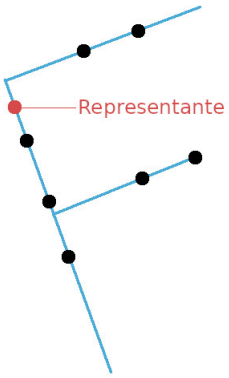


Para este problema, una posible solución consiste en iterar sobre todas las posibles efes, pero el problema es que hay infinitas de estas, por lo que tenemos que hallar una forma de quedarnos solo con "las que nos interesan".

Para esto, hay que caracterizar de alguna forma todas las efes que vayan a ser candidatos a ser efes óptimas.

Podemos comenzar, por ejemplo, con las efes que tienen dos puntos en su asta. De estos puntos, podemos tomarnos el punto que está más cerca del brazo superior de la efe, y utilizarlo como "representante" de la efe, junto con la dirección del asta.

Para seguir con esta idea, supongamos que nos dicen este punto "representante" de la efe, y la dirección en la que va su asta. ¿Podemos encontrar cuál de todas las efes que tienen esas caracteríaticas es la óptima?

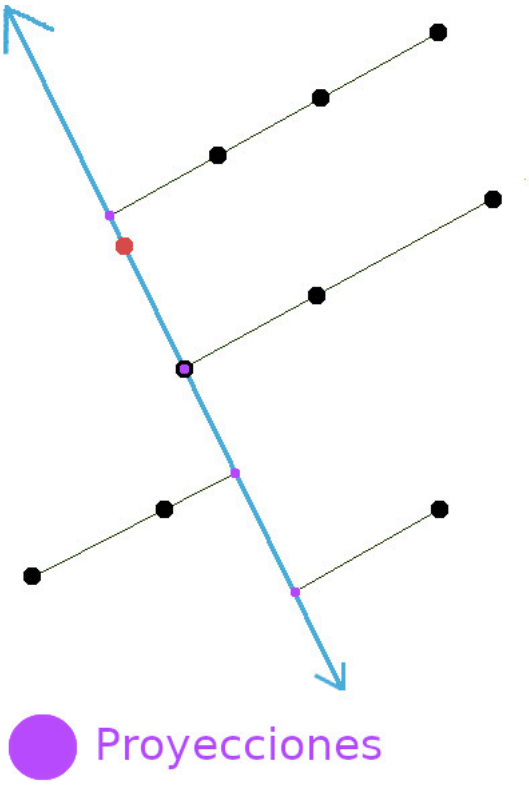


Inmediatamente podemos descartar todos los puntos del lado "incorrecto" (del lado para el que no va la efe), y también los puntos en la recta del asta que están más allá que nuestro representante, ya que, por como definimos representante, este es el que está más cerca del brazo superior.

Para optimizar qué puntos elegimos de los que nos

quedan, es necesario una observación clave: si sabemos
la dirección del asta de la efe, para que dos puntos estén en el mismo brazo de la misma, *deben tener la misma proyección hacia el asta.* (Ver nota al final).

Podemos entonces agrupar los puntos en base a su proyección, y de todos esos grupos, vamos a querer los que tengan más puntos. Pero no podemos simplemente elegir los dos que tengan más: para que nuestra efe sea válida, nuestro representante no puede "sobresalir" de la efe, es decir, tiene que estar antes de la proyección de uno de los brazos de la efe.



Teniendo en cuenta esto, si sabemos todas las proyecciones hacia el asta, y sabemos cuantos puntos comparten cada proyección, encontrar la cantidad óptima de puntos es fácil: para el brazo que tiene que estar por sobre el representante, nos tomamos la proyección que más puntos tenga que esté arriba del representante, y para el otro brazo nos tomamos la que tiene más puntos de las que sobraron.

Todo este proceso (encontrar la efe óptima dado el par (representante, dirección)), se puede hacer en tiempo $\mathcal{O}(n)$.

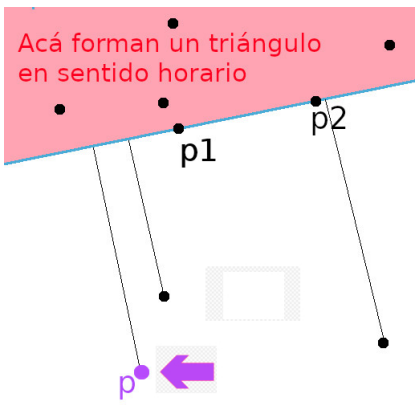
Ahora el problema reside en encontrar todos los posibles pares (representante, dirección) para probar. Una primera aproximación puede ser tratar de ver todas las efes que tengan al menos dos puntos en su asta: si sabemos su representante, sabemos que la dirección de su asta es en dirección de algún otro de los puntos del input. Podemos probar entonces hacer que cada uno de los n puntos sea un representante, y por cada uno de esos, nos tomamos las $n-1$ direcciones posibles (a cada uno de los otros puntos). Con esto iteraríamos un total de $\mathcal{O}(n^2)$ pares (representante, dirección), y cada uno cuesta $\mathcal{O}(n)$ procesarlo, por los que la complejidad queda $\mathcal{O}(n^3)$.

Ahora solo nos queda ver las efes con solo un (o cero) puntos en su asta. Notemos que podemos asumir, sin pérdida de generalidad, que toda efe tiene algún

punto en su asta, ya que si tuviese cero, podríamos "correr" el asta hasta que toque uno de los puntos de los brazos, y ahí tendría uno.

Vamos a hacer lo siguiente: por cada par de puntos p_1, p_2, vamos a construirnos un nuevo candidato (representante, dirección), y con estos vamos a recorrer todas las efes con un solo punto en su asta.

Dados los puntos p_1, p_2, vamos a tomarnos el punto p que tenga proyección sobre la recta $\overline{p_1, p_2}$ lo más hacia el lado de p_1 posible, pero tal que p_2, p_1 y p formen un triángulo en sentido antihorario. Luego vamos a recorrer las efes con el par (p, dirección perpendicular a $\overline{p_1 p_2}$).



Intuitivamente, estamos tratando de obtener la efe óptima que contenga a p_1 y a p_2 en uno de sus brazos, por lo tanto la dirección de su asta tiene que ser la perpendicular a $\overline{p_1 p_2}$.

Además, nos tomamos el punto "más hacia el lado de p_1", ya que esto nos da más espacio para poner más puntos en los brazos: Si teníamos una efe que era óptima, y podemos correr su asta hasta otro punto hacia el lado opuesto de los brazos, entonces seguirá siendo óptima.

La condición de que el triángulo sea antihorario no es más que decir "que p esté del lado correcto de la recta $\overline{p_1 p_2}$", como para que cuando construyamos la efe, el punto p no sobresalga por encima de la misma.

En total volvimos a tener $\mathcal{O}(n^2)$ pares (representante, dirección), y cada uno (igual que antes), nos cuesta $\mathcal{O}(n)$ tiempo procesarlo, por lo que esta segunda parte del algoritmo es también $\mathcal{O}(n^3)$.

Nota sobre proyecciones: aunque calcular proyecciones explícitamente es algo que se puede hacer, en este problema (como sólo queremos ver si dos puntos tienen la misma proyección), es más fácil utilizar el *producto interno*.

Dado un vector v, el producto interno de un punto con ese vector es muy muy fácil de calcular dadas las coordenadas, y nos dice básicamente "cuánto para allá" está el punto (donde "allá" es la dirección del vector v). Utilizando esto, que dos puntos tengan la misma proyección sobre la recta que estamos mirando es equivalente a que

tenga el mismo producto interno con el vector dirección del asta.

Si hacemos esto, una propiedad intereseante del producto interno es que se mantiene en enteros, lo que nos facilita el proceso de "agrupar de a proyecciones", ya que no tenemos que lidiar con las imprecisiones de los `double`s ni con fracciones.

Capítulo 3

Certamen Jurisdiccional

3.1. Nivel 1

3.1.1. Problema 1: Comprando rabanitos [rabanitos]

http://juez.oia.unsam.edu.ar/#/task/rabanitos/statement

Este problema consiste en una correcta implementación de la descripción del enunciado. Vamos a recibir dos números, que llamaremos `precio_rabanin` y `precio_rabanon`. Con estos dos precios tenemos 3 casos posibles que debemos distinguir:

1. Si `precio_rabanin` $<$ `precio_rabanon`, entonces debemos devolver como respuesta `"RABANIN"`.

2. Si `precio_rabanin` $>$ `precio_rabanon`, entonces debemos devolver como respuesta `"RABANON"`.

3. Si `precio_rabanin` $=$ `precio_rabanon`, entonces debemos devolver como respuesta `"DA IGUAL"`.

En todos los casos *no debemos imprimir las comillas* (o sea, debemos imprimir una cadena de texto normalmente), y debemos tener el cuidado de *imprimir todas las letras en mayúscula*. Un pseudocódigo que resuelve el problema se ve de la siguiente forma.

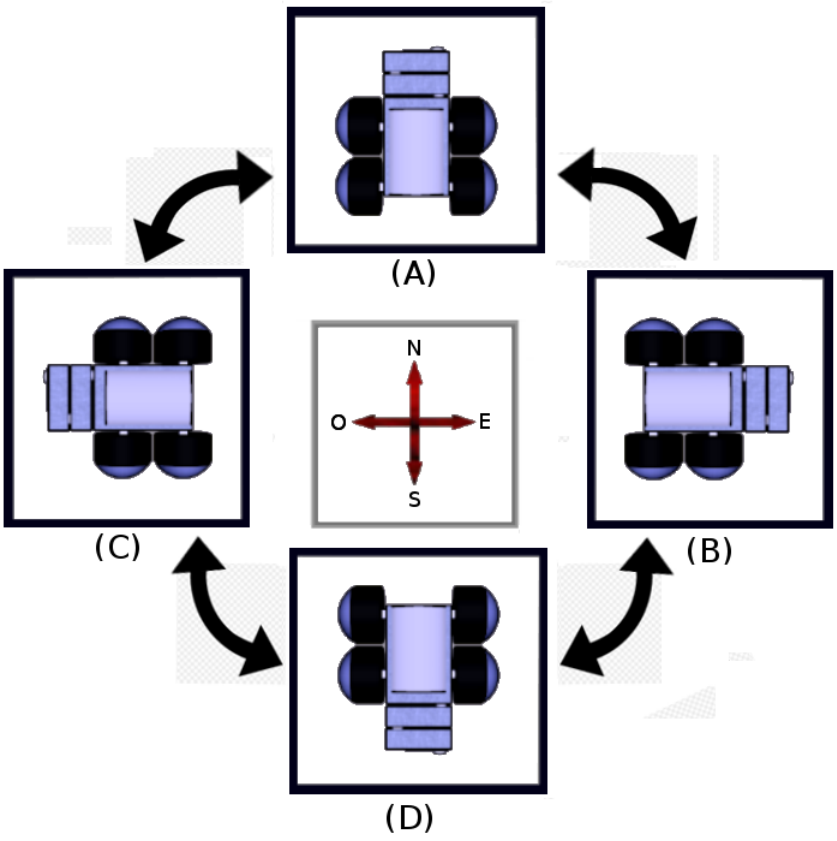
Algorithm 5 Solución al Problema 1 de Nivel 1 Jurisdiccional 2018 - rabanitos

```
1: function RABANITOS(PRECIO_RABANIN: ENTERO, PRECIO_RABANON: ENTERO)(PALABRA)
2:     if precio_rabanin < precio_rabanon then
3:         return RABANIN
4:     else if precio_rabanin > precio_rabanon then
5:         return RABANON
6:     else
7:         return DA IGUAL
```

3.1.2. Problema 2: Controlando al robot [robotito]

http://juez.oia.unsam.edu.ar/#/task/robotito/statement

Originalmente tenemos un robot que mira hacia el norte. En la entrada nos dan una serie de comandos que indica si el robot gira en modo *horario* o *antihorario*. A continuación ilustramos la situación. El robot comienza mirando hacia el norte como en la figura **(A)**. En caso realizar un giro *horario* al comenzar (notado con la letra `'H'` en la entrada), alcanzaremos la situación de la figura **(B)**. En caso realizar un giro *antihorario* al comenzar (notado con la letra `'A'` en la entrada), alcanzaremos la situación de la figura **(C)**.



El robot aplicará la serie de comandos *secuencialmente en el orden que viene en la entrada*. Debemos responder *cuántos movimientos ocurren* la primera vez que el robot queda mirando hacia el sur (la situación **(D)** en la figura), en caso de que esto nunca ocurra debemos responder -1.

Hay *solo* 4 *situaciones o estados distintos*. Con este esquema en mente, surgen algunas observaciones que pueden hacerse y que facilitarán una correcta implementación del problema. Concretamente:

- En caso de realizar un giro *horario* y uno *antihorario* es lo mismo que no haber girado (volvemos a la situación original).
- Solo hay dos formas de llegar a la situación en la que miramos al sur (o bien viniendo del oeste o bien viniendo del este).
- La **diferencia entre giros horarios y antihorarios no puede ser mayor a 2**, pues de ser mayor a 2 necesariamente tuvimos que haber pasado por la dirección sur.

Veamos entonces cómo resolver el problema. Hay muchas formas de implementarlo. Siguiendo el hilo de esta explicación quizá la más natural sea tener dos contadores, `horario` y `antihorario` que cuenten la cantidad de giros que hicimos de cada tipo. En todo momento sabemos que $|\texttt{horario} - \textit{antihorario}| \leq 2$, y si es igual a 2 quiere decir que estamos mirando hacia el sur. Un pseudocódigo posible es el siguiente, se asume que `instrucciones` es una cadena de texto de largo `N`.

Algorithm 6 Solución al Problema 2 de Nivel 1 Jurisdiccional 2018 - robotito

```
1:  function ROBOTITO(INSTRUCCIONES: PALABRA) (ENTERO)
2:      horario ← 0
3:      antihorario ← 0
4:      total_instrucciones ← 0
5:      for i = 0 ... N-1 do
6:          total_instrucciones ← total_instrucciones + 1
7:          if instrucciones[i] == 'H' then
8:              horario ← horario + 1
9:          else          ▷ No hace falta ver si es 'A' porque solo hay dos opciones
10:             antihorario ← antihorario + 1
11:         if |horario - antihorario| == 2 then
12:             break
13:     if |horario - antihorario| == 2 then
14:         return total_instrucciones
15:     else
16:         return -1
```

3.1.3. Problema 3: Contando números escalonados [escalonados]

http://juez.oia.unsam.edu.ar/#/task/escalonados/statement

Lo que pide el problema es básicamente recorrer todos los números del 10 al N, y contar cuántos de ellos son escalonados. Algo que nos va a servir en este ejercicio, es *separarlo en dos partes*: Primero, pensar y programar la manera en que vamos a **chequear si un número dado es escalonado o no**. Luego, cuando ya tengamos esa parte resuelta, simplemente vamos a **recorrer los números**, y sumar uno a un contador que tendrá la respuesta al final si el número en cuestión es escalonado (para lo cual utilizamos el código ya escrito).

Ahora, ¿cómo podemos hacer un código que nos diga si un número es escalonado o no? Podemos pensar cómo haríamos nosotros "a mano": Vamos viendo dígito a dígito, empezando por el segundo, si es más grande que el anterior. Si alguno pasa que no, entonces no lo es. Si se cumple con todos los dígitos, entonces sí.

¿Cómo obtenemos el primer dígito? Bueno, no es tan fácil, hay que dividir al número por la potencia de 10 más grande que es menor al número (donde la división es la división entera, como cuando trabajamos con enteros en la compu). Por ejemplo, para el número 3780, si lo dividimos por 1000 (que es la potencia de 10 más grande que no se pasa), entonces obtenemos 3 como resultado, que es efectivamente el primer dígito. Pero para esto tenemos que calcular la potencia más grande de 10 que no se pasa.

No es tan difícil, y puede quedar como ejercicio pensarlo de esta manera, pero pensemos algo un poco distinto: ¿Qué pasa si en vez de ir de izquierda a derecha, vamos de *derecha a izquierda*? En vez de ver si el dígito que viene es mayor, ahora *debemos ver que sea menor* para que el número sea escalonado.

¿Y cómo obtenemos el último dígito de un número? Simplemente tomando el resto del número en la división por 10. Y cuando obtenemos el último dígito (y chequeamos lo que tenemos que chequear), para descartar este último dígito podemos tomar la división entera del número por 10, y listo, eliminamos el último dígito.

Entonces con esta idea tenemos lo necesario para hacer un código sencillo, que mira de derecha a izquierda los dígitos y compara el que está mirando con el anterior:

Algorithm 7 Cómo saber si un número es escalonado

```
1: function esEscalonado(x : ENTERO) (BOOLEANO)
2:     respuesta ← true
3:     ultimoDigitoVisto ← x % 10                    ▷ Tomamos módulo 10
4:     x ← x / 10                                    ▷ División entera, o sea ⌊x/10⌋
5:     while x > 0 do
6:         digitoActual ← x % 10
7:         if ultimoDigitoVisto ≤ digitoActual then
8:             respuesta ← false
9:         x ← x / 10
10:        ultimoDigitoVisto ← digitoActual
11:    return respuesta
```

Ahora que tenemos esa función, simplemente recorremos los números pedidos, y chequeamos si se cumple la condición en cada uno de ellos:

Algorithm 8 Solución Problema 3 Nivel 1 - Jurisdiccional - escalonados

```
1: function escalonados(N : ENTERO)(ENTERO)
2:     contador ← 0
3:     for i = 10 ... N do
4:         if esEscalonado(i) then
5:             contador ← contador + 1
6:     return contador
```

3.1.4. Problema 4: Lanzamiento de aceitunas [olivares]

http://juez.oia.unsam.edu.ar/#/task/olivares/statement

Este problema es casi idéntico al problema `olivares2`, utilizado en nivel 3 (ver sección 3.3.2).

La única diferencia es que esta versión nivel 1 tiene cotas más bajas, y por lo tanto permite soluciones menos eficientes: por ejemplo, soluciones cuadráticas utilizando ordenamiento por burbujeo son suficientes para resolver esta versión.

Para un análisis detallado del problema, ver la versión en 3.3.2.

3.2. Nivel 2

3.2.1. Problema 1: Controlando al robot (compartido con nivel 1) [robotito]

http://juez.oia.unsam.edu.ar/#/task/robotito/statement

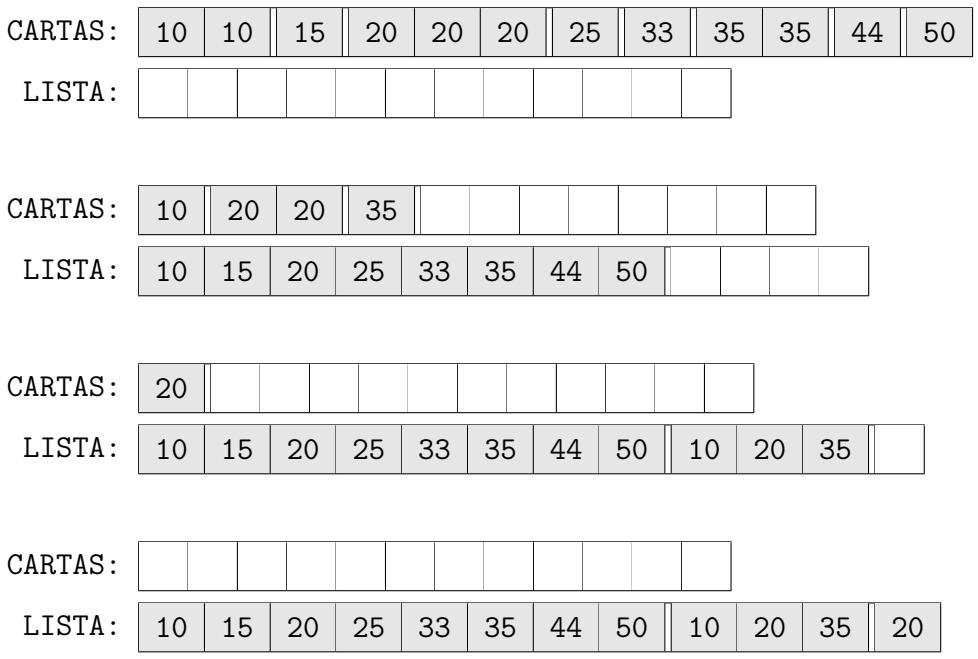
Este problema es exactamente igual al problema 2 del nivel 1 (ver 3.1.2).

3.2.2. Problema 2: Jugando al sortucho [sortucho]

http://juez.oia.unsam.edu.ar/#/task/sortucho/statement

En el problema hay tarjetas con un número en cada una. Primeramente se agrupan las cartas con un mismo número y se descartan la *mitad de las tarjetas redondeando hacia abajo*. Por lo tanto, si en un grupo de cartas con un mismo número había x cartas, para lo que resta del problema tendremos $\lceil \frac{x}{2} \rceil$ cartas con dicho número.

Una vez hecha esta simplificación, se toman los grupos de cartas y se ordenan por el número de tarjeta de *menor a mayor*. Finalmente, se genera una lista de números tomando las tarjetas restantes, sacando una carta de cada grupo (en orden de menor a mayor), y poniéndola al final de la lista. Este proceso se repite hasta que no queden cartas. En el ejemplo del enunciado, al comenzar este último paso se procede de la siguiente forma.



Un primer intento de solución podría consistir en simplemente simular el proceso que se describe en el enunciado. Si originalmente tenemos un arreglo `histograma` tal que `histograma[x]` nos dice cuántas tarjetas tienen el número x, lo primero que hacemos es descartar la mitad en cada grupo. Es decir, reemplazamos `histograma[x]` $\leftarrow \lceil \frac{\texttt{histograma[x]}}{2} \rceil$.

Veamos entonces cómo interpretar una entrada de N tarjetas para generar el arreglo `histograma`. Las líneas en azul marcan una variante que además devuelve el máximo valor almacenado en `histograma`. Aquí asumimos que `histograma` comienza inicializado con ceros y con un tamaño superior al máximo valor de una tarjeta (que llamamos `MAX_NUM`), se vería de la siguiente forma.

Algorithm 9 Análisis de Entrada - Prob. 2 de Nivel 2 Jurisdiccional 2018 - sortucho

```
1: function SORTUCHO_ENTRADA(TARJETAS: ARREGLO DE ENTEROS)(ENTERO)
2:     for i = 0 ... N-1 do
3:         histograma[tarjetas[i]] ← histograma[tarjetas[i]] + 1
4:     M ← −∞
5:     for x = 1 ... MAX_NUM do
6:         histograma[x] ← ⌈histograma[x]/2 ⌉
7:         M ← máx(M, histograma[x])
8:     return M
```

Recorriendo `histograma` en ***orden creciente***, debemos agregar un número a la lista por cada x con `histograma[x]` $\geq \mathbf{1}$. Finalmente deberíamos repetir este proceso con la salvedad de agregar un número a la lista por cada x con `histograma[x]` $\geq \mathbf{2}$, y así hasta el máximo valor que tenga almacenado `histograma`, que llamaremos $M = \max_{x \geq 0}$ `histograma[x]`, y que vimos cómo calcular más arriba (en azul).

Algorithm 10 Solución 1 al Problema 2 de Nivel 2 Jurisdiccional 2018 - sortucho

```
1: function SORTUCHO(TARJETAS: ARR. DE ENTEROS)( ARR. de ENTEROS)
2:     M ← sortucho_entrada(tarjetas)
3:     solucion ← []
4:     nivel ← 1
5:     for pasada = 1 ... M do
6:         for x = 1 ... MAX_NUM do
7:             if histograma[x] ≥ nivel then
8:                 solucion.agregar_al_final(x)
9:         nivel ← nivel + 1
10:    return solucion
```

Esta solución tal cual como está, si bien es correcta, es un tanto ineficiente. ¿En qué casos podríamos tener problemas?. Analicemos en detalle cuántas operaciones se realizan en cada caso. Al manipular la entrada en el primer algoritmo realizamos $\mathcal{O}(N + MAX\_NUM)$. En el segundo algoritmo por cada pasada de la primera iteración recorremos la totalidad de los MAX_NUM números, haciendo $\mathcal{O}(N \cdot MAX\_NUM)$ operaciones. Por lo tanto si en la entrada nos vienen 1,000,000 tarjetas todas iguales con el número 1,000,000 este algoritmo estaría haciendo alrededor de 10^{12} operaciones, lo cual es demasiado alto.

Como vimos, este algoritmo realiza tantas *pasadas* como el número de tarjeta que aparezca la mayor cantidad de veces. Ahora ¿Cuántos números pueden aparecer más de k veces a la vez? A lo sumo $\lfloor \frac{N}{k} \rfloor$ (de haber más nos pasamos de N números en total), por lo tanto recorrer todos los números en cada pasada es lo que lo hace ineficiente. Para ser concretos, en la primera pasada puede haber N números distintos, en la segunda $\lfloor \frac{N}{2} \rfloor$, en la tercera $\lfloor \frac{N}{3} \rfloor$.

Utilizando que $\sum_{k=1}^{N} \frac{N}{k} = N \cdot \sum_{k=1}^{N} \frac{1}{k} \sim \mathcal{O}(N \cdot \lg N)$, podríamos pensar que un algoritmo que en cada pasada solamente mira las tarjetas de las cuales todavía quedan números por pasar a la lista realiza alrededor de $N \cdot \lg N$ operaciones, pero implementado correctamente realiza solo $\mathcal{O}(N)$ operaciones, ¿por qué?. Si realizamos un *análisis amortizado* de la situación podemos ver que estamos realizando solo $\mathcal{O}(1)$ operaciones por cada tarjeta de la entrada, ¡pues estamos viendo cada tarjeta una sola vez!.

Ahora solo nos queda resolver el problema sabiendo que al mirar solamente las tarjetas que restan por poner en la lista no estaremos haciendo muchas operaciones. La idea clave consiste en **asignar a cada tarjeta la pasada en la cual será agregada a la lista**. Un posible algoritmo que realiza esta idea es el siguiente:

Algorithm 11 Solución 2 al Problema 2 de Nivel 2 Jurisdiccional 2018 - sortucho

```
1:  function SORTUCHO(TARJETAS: ARR. DE ENTEROS)(ARR. de ENTEROS)
2:      M ← sortucho_entrada(tarjetas)
3:      solucion_pares ← []                  ▷ Lista de pares {first,second}
4:      for x = 1 ... MAX_NUM do
5:          for nivel = 1 ... histograma[x] do
6:              solucion_pares.agregar_al_final({nivel,x})
7:      solucion_pares.ordenar()          ▷ Los pares se comparan lugar a lugar
8:      solucion ← []
9:      for {nivel,x} ∈ solucion_pares do
10:         solucion.agregar_al_final(x)
11:     return solucion
```

El problema de esta solución es que si bien solo recorremos $\mathcal{O}(N)$ tarjetas, al ordenar estamos realizando $\mathcal{O}(N \lg N)$ operaciones. Afortunadamente hay muchas formas de implementar la idea que vimos, veamos algunas más eficientes.

Una forma más eficiente que la anterior podría consistir en tener *dos listas*, la primera que comienza con todas las tarjetas distintas y la segunda vacía. Al realizar la primera pasada se agregan a la segunda lista de números solamente las cartas que tienen `histograma[x]` ≥ 2. Al finalizar la pasada se vacía la primera lista y se intercambian los roles entre ambas listas. Luego, basta repetir este procedimiento

(aumentando por `histograma[x]` $\geq 3, 4, \ldots$) hasta finalizar con ambas listas vacías. En cada paso estamos asegurándonos de recorrer en la próxima pasada solo las tarjetas que todavía faltan colocar.

Como dijimos, la idea es *asignar a cada tarjeta la pasada en la que será colocada* en la lista. Entonces, otra forma de implementarla consiste en tener una lista por cada pasada que contenga los números que serán colocados en esa pasada, para finalmente volcarlos en la lista `solucion`. Lo importante es que estos dos últimos enfoques realizan solo $\mathcal{O}(1)$ operaciones por cada número, obteniendo una complejidad de $\mathcal{O}(N)$ para resolver el problema. Un posible pseudocódigo de este último enfoque consiste en:

Algorithm 12 Solución 3 al Problema 2 de Nivel 2 Jurisdiccional 2018 - sortucho

```
1:  function SORTUCHO(TARJETAS: ARR. DE ENTEROS)(ARR. de ENTEROS)
2:      M ← sortucho_entrada(tarjetas)
3:      solucion ← []
4:      pasada_num ← [[], ..., []]          ▷ Lista de listas con MAX_NUM lugares
5:      for x = 1 ... MAX_NUM do            ▷ Recorremos en orden creciente
6:          for nivel = 1 ... histograma[x] do
7:              pasada_num[nivel].agregar_al_final(x)
8:      for nivel = 1 ... M do
9:          for x ∈ pasada_num[nivel] do
10:             solucion.agregar_al_final(x)
11:     return solucion
```

3.2.3. Problema 3: Canción [cancion]

http://juez.oia.unsam.edu.ar/#/task/cancion/statement

El problema nos pide encontar el "estribillo" de una string s dada en la input, que se define como *la mayor substring que se repite (al menos) dos veces, sin solaparse*

Para resolver este problema vamos a utilizar una estructura denominada *trie*, más precisamente, el *suffix trie* de nuestra string s.

Se puede leer más acerca de la estructura en http://wiki.oia.unsam.edu.ar/algoritmos-oia/trie, pero un trie es básicamente una estructura que guarda un conjunto de strings en forma de árbol: cada nodo del trie representa un *prefijo común* de algunas de las strings del conjunto (posiblemente una sola); es decir, el trie *fusiona los prefijos comunes* de las strings que están en el conjunto, haciéndolos un sólo nodo.

La estructura de trie es tal que si un nodo representa el prefijo p, luego su

padre es el representante del prefijo que se obtiene sacándole la última letra a p. Así, el único prefijo sin padre es el "prefijo vacío", y cumple la función de raíz del trie.

La estructura de suffix trie surge de poner en un trie todos los sufijos de una string dada (en nuestro caso, s). ¿Por qué nos sirve esto? Pues porque todas las substrings de s son el prefijo de alguno de los sufijos de s, por lo que los nodos de nuestro suffix trie se van a corresponder con los substrings de s.

Entonces, ahora logramos tener una estructura en forma de árbol que contiene todas las substrings de la string s, pero necesitamos más. Nos gustaría saber, para cada nodo del trie (que representa una substring):

1. cuál es su longitud (porque queremos la más larga), y
2. si se repite dos veces sin solaparse.

Para calcular si una cadena se repite sin solapar, es necesaria una observación clave. Dada una substring t, para ver si aparece dos veces sin solaparse, "conviene" tomar su primera y última aparición en s. Esto es así ya que estas son las dos apariciones de t que van a estar más lejos entre sí, por lo que si estas se solapan, todas las demás también se solaparán.

Además, si sabemos en qué índice de s comienzan la primera y la última aparición de una substring t, podemos calcular si estas se solapan o no: Como ya sabemos el tamaño de la substring t (por 1), basta con ver si la substring t cabe entre los dos índices, es decir, si la distancia entre la primera aparición y la última aparición es al menos la longitud de t.

Hay dos opciones para calcular estos tres valores (longitud, índice de primera aparición, índice de última aparición) en cada nodo del trie:

3.2.3.1. A) Guardar la información mientras se construye el trie

En este problema, como vamos a insertar todos los substring en el trie, utilizaremos dos fors para fijar primero la posición inicial del substring (i), y, por cada una de esas posibles posiciones iniciales, en un segundo for recorremos todas las posibles posiciones finales (j). La clave es que al recorrer así, cada substring que examinamos tiene exactamente un caracter más que la anterior, y entonces no vamos a insertarla en el trie desde cero, sino que simplemente avanzamos un nodo desde la posición donde quedamos tras procesar el substring anterior. Esto permite construir el suffix trie y a la vez recorrer explícitamente todos los substrings en forma sencilla, pero con complejidad cuadrática y no cúbica.

De esta manera, cuando un substring aparece muchas veces, al ir creando el trie recorremos su nodo varias veces, una por cada aparición. Y en el momento en que pasamos por ese nodo, conocemos (gracias a los índices i,j) el tamaño y la ubicación del substring en cuestión. Podemos entonces aprovechar para allí mismo anotar esta información en el correspondiente nodo. Más precisamente:

- Al pasar por un nodo, **siempre anotamos** su longitud, que será $j - i$, y **siempre pisamos** el valor almacenado de su última aparición con i (si barremos i crecientemente, el último valor con el que pisaremos será la última aparición).
- Al pasar por un nodo **por primera vez**, anotamos su primera aparición en i (si barremos i crecientemente, la primera vez que encontramos un nodo será en la primera aparición de ese substring).

3.2.3.2. B) Construir el trie, y luego recuperar la información del trie

La longitud de cada substring (nodo en el trie) es la más simple de las dos cosas a calcular. Dada una substring t de s, los antecesores del nodo de t en el árbol serán los prefijos sucesivos de t en orden decreciente de longitud, hasta llegar a la raíz que es el prefijo de longitud 0. Por esto, la longitud de t corresponderá con la prefundidad de t en el árbol, que se puede calcular haciando un `dfs` sobre el árbol.

Para calcular para cada substring su índice de primera y última aparición, hay que adentrarnos en la estructura del suffix trie. Cuando construimos el trie, algunos nodos se marcan como *nodos terminales*. Estos son los nodos que representan no cualquier substring de s, sino específicamente un sufijo (notar que las hojas siempre son nodos terminales, pero no necesariamente viceversa).

Todos los nodos terminales en el subárbol del nodo de una substring t representan todos los sufijos de s de los cuales t es prefijo, es decir, todas las apariciones de t en s.

Sabiendo esto, podemos guardar en cada nodo terminal el índice donde comienza su sufijo correspondiente (será simplemente n menos su longitud, pues todos los sufijos terminan donde termina todo el string). Luego, por cada substring t, si miramos el mínimo de esos índices de inicio para todos los nodos terminales en el subárbol del nodo correspondiente a t, obtenemos el índice de la primer aparición de t como substring de s.

Esta operación se puede realizar fácilmente con una dp sobre el árbol: en cada nodo queremos el mínimo sobre todos sus hijos. Además, análogamente se puede

obtener el *último* índice de aparición (tomando el máximo en vez del mínimo).

3.2.3.3. Solución final

Entonces, para cada substring t ya tenemos su índice de primer y última aparición, así como también su longitud. Ahora sólo basta con iterar sobre todas estas substrings, chequear que la primera y última aparición no se solapen, y tomar de todas estas la que tiene mayor longitud (Y en caso de no haber, imprimir `NO HAY`).

En términos de complejidad, esta solución es $\mathcal{O}(n^2)$ (n es la longitud de s), ya que insertar una string en un trie toma tiempo proporcional a su longitud, y estamos insertando todos los sufijos de la string s, que tienen suma de longitudes:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \mathcal{O}(n^2)$$

Por lo que, en particular, el trie tendrá también $\mathcal{O}(n^2)$ nodos.

Además, la complejidad de eventuales dfs para calcular la profundidad en el árbol, así como las dps para calcular apariciones, siempre es lineal en el tamaño del árbol, que es $\mathcal{O}(n^2)$.

Finalmente, iterar sobre todo el árbol para computar la substring de mayor longitud que se repite sin solaparse es de nuevo lineal en el tamaño del árbol, es decir $\mathcal{O}(n^2)$.

Luego la complejidad total del algoritmo es $\mathcal{O}(n^2)$, que con $n = 2000$ es suficiente.

3.2.4. Problema 4: Nuevas Autopistas [nautopistas]

http://juez.oia.unsam.edu.ar/#/task/nautopistas/statement

Este problema, nos sugiere fuertemente que lo modelemos mediante grafos, al hablar de ciudades conectadas mediante autopistas que conectan ciudades. Recordemos que el objetivo en este problema es construir autopistas, de forma tal que el costo total sume un numero específico. Simplificamos la explicación al hablar de un solo número en lugar de una lista, pero al final veremos que dá lo mismo. Veamos como traducir cada parte del problema a una propiedad del grafo.

Tenemos N ciudades y N autopistas distintas que podemos construir. Cada una de las autopistas tiene un costo para construirse, y conecta en forma directa dos ciudades diferentes. Nuestro grafo tiene N nodos conectados mediante N aristas.

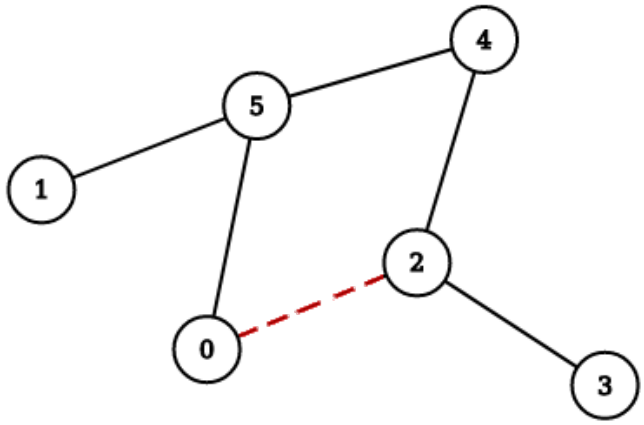
Cada arista conecta dos nodos distintos mediante un costo. Es un ¨grafo pesado¨

Es posible llegar desde cualquier ciudad a cualquier otra utilizando solo autopistas de la lista de posibles. Esto signficia que el grafo es conexo.

Queremos ver si podemos eliminar algunas autopistas de la lista, tal que las restantes sigan conectando todas las ciudades, y que además el costo total de construcción sea exactamente un cierto número F que le gusta al ministro. Entonces queremos tener un subgrafo del original que siga siendo conexo pero que además tenga costo total F. [1]

Ahora que tenemos este modelo de nuestro problema, podemos pensar propiedades que conocemos de la teoría de grafos. Sabemos que un Grafo de N nodos y N-1 aristas conexo es un árbol. ¿Qué ocurre con un grafo de N nodos y N aristas conexo? ¿Cómo es el grafo original? Podemos demostrar que tal grafo tiene **exactamente un ciclo**. Esto se debe a que podemos formarlo agregando una arista (u, v) a su árbol generador mínimo. Luego, como ya había un camino P entre u y v por ser conexo el árbol, P + (u, v) forma un ciclo. Y es el único pues el árbol no tenía ciclos.

Además tenemos que tener en cuenta que un grafo de N-2 aristas (donde N sigue siendo la cantidad de nodos) no es conexo. Demostración: Supongamos que tenemos un grafo de N-2 aristas conexo, al agregarle una arista cualquier, como sigue siendo conexo y tiene N-1 aristas, es un árbol. Pero como le agregamos una arista a un grafo conexo, tiene un ciclo. ¡Absurdo!



Concluimos entonces que a nuestro grafo original de N aristas le vamos a sacar

[1] Esto se relaciona con el problema de Árbol Generador Mínimo. Se puede leer más al respecto en http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/arbol-generador

0 ó 1 arista, de forma tal que se mantenga conexo.

Es fácil verificar si sacando 0 aristas logramos nuestro cometido. Simplemente chequeamos si la suma total de costos de autopistas (de acá en adelante T) es igual al número F.

Si el costo F es mayor a T, es imposible cumplirlo. Por último, si F es menor que T, el problema está en verificar si es posible lograrlo utilizando 1 arista.

Entonces buscamos una arista para sacar del grafo con las siguientes restricciones:

- Ser parte del ciclo.

- Tener costo T - F (que sabemos es positivo).

Tiene que ser parte del único ciclo, porque de lo contrario dejaríamos disconexo al grafo. Y tiene que tener costo $T - F$ para que al removerla nos quede un árbol de costo F. Así que vamos a tener que encontrar todas las aristas del ciclo, y verificar si alguna de ellas tiene costo exactamente $T - F$. Para eso vamos a usar el algoritmo de DFS: http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/dfs.

Durante la ejecución del DFS, recorremos el grafo de forma tal de apilar los nodos que vemos, hasta terminar de recorrer todas las ramas de sus hijos. Si en algún punto de ese proceso nos encontramos un nodo que ya está en la pila (nodo x), significa que recorriendo los hijos de x, ¡Volvimos a encontrar a x! Eso es exactamente un ciclo. Y el ciclo está formado por todas las aristas que están en la pila entre las dos apariciones del nodo x.

Algorithm 13 Solución al Problema 4 de Nivel 2 Jurisdiccional 2018 - autopistas

```
1: function AUTOPISTAS(G: ADYLISTA, NUMEROS: LISTA[ENTERO])(BOOLEANO)
2:     total ← sumarAristas(G)
3:     if total ∈ numeros then
4:         return true
5:     ciclo : Lista[Arista] ← encontrarCiclo(G)
6:     for arista : ciclo do
7:         if total − arista.costo ∈ numeros then
8:             return true
9: function SUMARARISTAS(G: ADYLISTA)(ENTERO)
10:    total ← 0
11:    for nodo : G do
12:        for arista : G[nodo] do
13:            total ← total + arista.costo
14:    return total
15: function ENCONTRARCICLO(G: ADYLISTA)(LISTA[ARISTA])
16:    ciclo ← []
17:    aristasEnStack ← []
18:    dfs(0, G, aristasEnStack, ciclo)
19:    return ciclo
20: function DFS(NODO: ENTERO, G: ADYLISTA, VISITADO: ARREGLO[BOOLEANO], ENSTACK: ARREGLO[BOOLEANO], ARISTASENSTACK: LISTA[ARISTA], CICLO: LISTA[ARISTA])(VOID)
21:    for arista: G[nodo] do
22:        visitado[nodo] ← true
23:        enStack[nodo] ← true
24:        if ¬visitado[arista.hasta] then
25:            aristasEnStack ← aristasEnStack + {arista.hasta}
26:            dfs(arista.hasta, G, aristasEnStack, ciclo)
27:            aristasEnStack ← aristasEnStack.pop()
28:        else if enStack[arista.hasta then
29:            ciclo ← aristasEnStack
30:        enStack[nodo] ← false
```

3.3. Nivel 3

3.3.1. Problema 1: Revisando el boletín [boletin]

http://juez.oia.unsam.edu.ar/#/task/boletin/statement

Lo primero a resolver, es cómo hacer para ingresar varios números. Bueno, en realidad, no es "varios", porque para ingresar 10 números podríamos copiar y pegar 10 veces la línea para ingresar uno y listo (aunque *no es recomendable hacerlo*).

Cuando la cantidad total es variable, hay que hacer algo un poco más complejo.

Lo que hacemos es hacer un **for**, para que la computadora entre al "for" unas N veces, y adentro de cada ciclo, se ingrese un número. Hay que tener cuidado con esto porque, como el N es variable, no podemos declarar N variables y hacer "`ingresar var1`", "`ingresar var2`", ..., "`ingresar varN`", **¡porque no sabemos a priori cuánto vale** N**!**.

Entonces, vamos a guardar los datos ingresados en la misma variable. Para no perder la información de los números ingresados, vamos a necesitar usar el valor de alguna manera. Lo usual es almacenar el valor en un conjunto (`vector` en C++, `List` en Java), para al final de los N pasos tener todos los valores guardados. Pero en este caso no va a hacer falta, veamos por qué.

Para **calcular la suma** de un conjunto de números, podemos inicializar una variable en 0, e ir sumando uno por uno los números. En este ejericio, esto podemos hacerlo de dos maneras: O bien primero guardamos todos los números que nos dan, para luego recorrer esa lista e ir sumando de uno, o a medida que vamos leyendo los números, podemos ir sumando sus valores a la variable que inicializamos en 0.

Para **calcular el promedio**, dado que el promedio es la suma total dividido la cantidad de números sumados, una vez que tenemos la suma, y habiendo guardado el número incial N, simplemente debemos hacer la división $\lfloor \frac{SUMA}{N} \rfloor =$ `suma/N`.

Para **saber si aprueba o no**, además del promedio (que vimos antes cómo calcular), debemos guardar el último número ingresado. Si guardamos todos los números en una lista esto es fácil, accedemos a la última posición de la lista y nos fijamos si es más grande que 7 o no. Aquí tendremos que usar un **if** con dos condiciones, la del promedio y la de la última nota. Para eso usamos "`&&`" que es un "Y", es decir que queremos que el promedio sea mayor o igual a 7 *y además*, que la última nota sea mayor o igual a 7.

También podemos hacerlo con un **if** adentro del otro, aunque va a ser más molesto porque vamos a tener que escribir dos **else**, uno para el primer *if*, ya que si la primera condición no se cumple, queremos ir al *else* para imprimir que "NO" aprueba, pero además, si pasa la primera condición pero no la segunda, también queremos imprimir "NO", entonces debemos hacer otro comando *else*.

Si no tenemos la lista de los números, como al ingresar datos lo hacemos uno por uno, en la variable que sea (por ejemplo, si guardamos cada número en una misma variable "x"), nos quedó justo en esa variable el último valor ingresado al final.

Para **saber cuál es la mejor nota**, la manera de hacerlo es la siguiente: Inicializar una variable con el valor -1 (ya que todas las notas son entre 0 y 10), y a

medida que nos ingresan los números, nos fijamos, si el valor ingresado es mayor al que tenemos (de ser así hemos encontrado una nota mejor a la veníamos guardando), entonces actualizamos el valor con el ingresado recién. Al final, tendremos en esta variable la mejor nota de todas. Por eso empezamos con un -1, porque por más que sean todos ceros, ya la primera es mayor a lo que empezamos guardando, y entonces vamos a actualizar a nuestra variable con esta primera nota.

A continuación se muestra un código en C++ que además de resolver el problema sirve de ejemplo sobre cómo usar el **for** para ingresar N números.

```
#include<iostream>

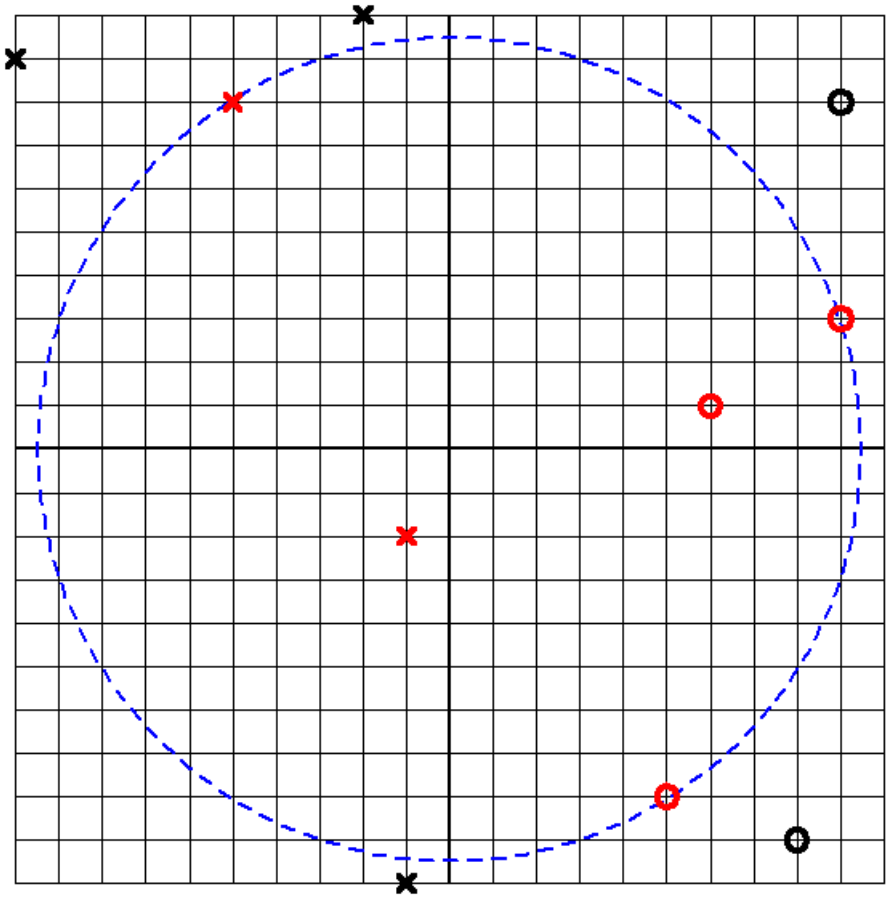
using namespace std;

int main(){
    int N;
    int suma=0, mejorNota=-1;
    int dato;
    cin>>N;
    for(int i=0; i<N; i++){
        cin>>dato; // Guardamos el valor que se ingresa en la variable dato.
                   // Esta variable va a ir tomando todos los valores ingresados,
                  // y al final contendrá el valor del último número.
        suma = suma + dato;
        if(dato > mejorNota){
            mejorNota = dato;
        }
    }
    int promedio = suma/N;
    cout<<suma<<endl;
    cout<<promedio<<endl;
    if(promedio >= 7 && dato >= 7){
        cout<<"SI"<<endl;
    }else{
        cout<<"NO"<<endl;
    }
    cout<<mejorNota<<endl;
}
```

3.3.2. Problema 2: Lanzamiento de aceitunas [olivares2]

http://juez.oia.unsam.edu.ar/#/task/olivares2/statement

En el problema se nos pide encontrar el *valor del radio* (en realidad de su cuadrado) que maximiza la diferencia entre los tiros del jugador estrella del Oli y su archirival que caen adentro de la cancha. Para un cierto tiro realizado en (x, y) nos va a interesar saber para qué valores de R^2 el tiro estará adentro o afuera de la cancha, ¿Cómo podemos hacerlo?¿Cuál es el menor valor de R^2 para que el tiro esté adentro?.



El menor valor posible de R para que el tiro esté adentro está dado por la distancia del punto (x, y) al centro de la cancha (de esa forma la circunferencia que delimita la cancha pasa exactamente por el punto (x, y)). Esta distancia la podemos calcular por pitágoras, pues corresponde a la hipotenusa del triángulo recto con base x y altura y. Concluimos que R satisface la ecuación $R^2 = x^2 + y^2$. Para radios mayores, el tiro también caerá adentro de la cancha.

Por lo tanto fijado el valor de R^2, un lanzamiento en (x, y) se lo considera adentro si $x^2 + y^2 \leq R^2$ (notar que incluimos el igual, pues el enunciado dice que

línea es adentro). De cada punto solo nos interesa el valor de $x^2 + y^2$, primero que nada vamos a generar el arreglo `distancias` que almacenará el valor de x^2+y^2 para los lanzamientos del Oli y $-x^2 - y^2$ para los lanzamientos de su rival.

Algorithm 14 Entrada al Problema 3 de Nivel 2 Jurisdiccional 2018

```
1: function OLIVARES_ENTRADA(coordX: ARR. DE ENTEROS, coordY: ARR. DE ENTEROS)( ARR. de ENTEROS)
2:     distancias ← []
3:     for i = 0 ... N-1 do
4:         radio_cuadrado ← coordX[i]*coordX[i] + coordY[i]*coordY[i]
5:         if coordX[i] < 0 then
6:             radio_cuadrado ← -radio_cuadrado
7:         distancias.agregar_al_final(radio_cuadrado)
8:     return distancias
```

Una primera idea consiste en iterar todos los radios posibles, luego para cada radio fijo se pueden calcular cuántos tiros del lanzador del Oli y su archirival están adentro de la cancha. Finalmente bastaría con calcular la diferencia, y tomar aquel menor radio que maximice la diferencia buscada.

Un pseudocódigo que implemente esta idea es el siguiente. Aquí `MAX_RADIO` refiere al máximo valor posible de $x^2 + y^2$ dado por las cotas del enunciado (lo cual tendrá distinta relevancia dependiendo de la subtarea). La cantidad de operaciones que realiza este algoritmo será del orden de $\mathcal{O}(N \cdot MAX\_RADIO)$.

Una optimización que podemos hacer de manera relativamente sencilla radica en la observación de que *no hace falta probar todos los radios posibles*, probando solamente los distintos valores de $x^2 + y^2$ alcanza (pues en los valores intermedios no varía la cantidad de lanzamientos adentro o afuera).

En este último caso obtendríamos una complejidad temporal de $\mathcal{O}(N^2)$, y el único cambio corresponde a la línea 5 (debemos reemplazarla por el comentario en azul), y las líneas 9 y 11, donde debemos tomar módulo en `r_cuad` para tener en cuenta que podrían ser lanzamientos del rival.

Algorithm 15 Solución 1 al Problema 3 de Nivel 2 Jurisdiccional 2018 - olivares

```
1: function OLIVARES(COORDX: ARR. DE ENTEROS, COORDY: ARR. DE ENTEROS)(ENTERO)
2:     distancias ← olivares_entrada(coordX,coordY)
3:     max_dif ← 0  ▷ Siempre podemos obtener una diferencia de 0 con radio 0
4:     r_cuad_respuesta ← 0
5:     for r_cuad = 0 ... MAX_RADIO do          ▷ for r_cuad ∈ distancias do
6:         oli ← 0
7:         rival ← 0
8:         for i = 0 ...  N-1 do
9:             if distancia[i] > 0  and distancia[i] ≤ |r_cuad| then
10:                oli ← oli + 1
11:            else if distancia[i] < 0  and -distancia[i] ≤ |r_cuad| then
12:                rival ← rival + 1
13:        if  oli - rival > max_dif then
14:            max_dif ← oli - rival
15:            r_cuad_respuesta ← r_cuad
16:        else if oli-rival == max_dif and r_cuad < r_cuad_respuesta  then
17:            r_cuad_respuesta ← r_cuad          ▷ Nos piden el menor radio posible
18:    return r_cuad_respuesta
```

En el caso de nivel 1, esta última variante del algoritmo visto alcanza para obtener la totalidad de los puntos (teniendo cuidado con el *overflow* utilizando `long long` en C++ por ejemplo). Pero en el caso de nivel 3 tenemos que $N \leq 100{,}000$ y un algoritmo de complejidad $\mathcal{O}(N^2)$ no terminará de correr en el tiempo deseado.

Veamos entonces cómo resolver el problema en este último caso. La idea será aprovechar que si **vamos viendo los lanzamientos en orden** (por distancia al centro), entonces solamente se van agregando lanzamientos y en todo momento podemos ir manteniendo la diferencia entre ambos jugadores.

El mayor cuidado a la hora de implementar esta idea es cuando hay *varios lanzamientos con la misma distancia al centro, debemos incluir a todos a la vez.* Una correcta implementación de esta idea tiene una complejidad de $\mathcal{O}(N \lg N)$, dado que se requiere ordenar `distancias` por valor absoluto.

Algorithm 16 Solución 2 al Problema 3 de Nivel 2 Jurisdiccional 2018 - olivares

```
1: function OLIVARES(COORDX: ARR. DE ENTEROS, COORDY: ARR. DE ENTEROS)(ENTERO)
2:     distancias ← olivares_entrada(coordX,coordY)
3:     distancias.ordenar()        ▷ En orden creciente por valor absoluto
4:     max_dif ← 0  ▷ Siempre podemos obtener una diferencia de 0 con radio 0
5:     r_cuad_respuesta ← 0
6:     oli ← 0
7:     rival ← 0
8:     for i = 0 ... N-1 do                    ▷ Notar que las visitamos en orden
9:         if distancia[i] > 0  then
10:            oli ← oli + 1
11:        else if distancia[i] < 0  then
12:            rival ← rival + 1
13:        if i == N-1 or |distancia[i]| ≠ |distancia[i+1]| then ▷ Si falta
   agregar lanzamientos con la misma distancia o bien ya terminamos...
14:            if  oli - rival > max_dif then        ▷ Si mejora la respuesta...
15:                max_dif ← oli - rival
16:                r_cuad_respuesta ← r_cuad
17:    return r_cuad_respuesta
```

Esta es solo una forma de implementar esta última idea. Otra opción posible (para agregar a todos los lanzamientos que están a la misma distancia del centro a la vez) es tener asignado para cada distancia cuántos lanzamientos hay del lanzador del oli y cuántos de su rival (con `map` en C++ por ejemplo).

3.3.3. Problema 3: Bolñitsy góroda [hospitales]

http://juez.oia.unsam.edu.ar/#/task/hospitales/statement

Breve abstracción: dado un grafo conexo, simple y ponderado; donde algunos vértices están marcados como especiales (hospitales), responder consultas de la forma x, z con tres valores y, d, c: el hospital y más cercano a x, la distancia óptima del camino $x \rightsquigarrow y \rightsquigarrow z$, y la cantidad de formas de hacer dicho camino.

Vista la gran cantidad de consultas que nos puedan hacer, es preferible tener precomputado para cada sitio su hospital más cercano, junto a la distancia y cantidad de maneras de realizar un camino óptimo desde un sitio a cualquier otro vértice.

Este problema es clásico en grafos: *hallar la distancia más corta para todo par de vértices*. En este enunciado, pedimos también contar la cantidad de caminos mínimos.

Existe una solución conocida que utiliza *programación dinámica*, llamada **Algoritmo de Floyd-Warshall**. Se puede leer más sobre este algoritmo en http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/floyd-warshall.

`dp(i, j, k)`: distancia más corta de i a j, utilizando como vértices intermedios $\{0..k\}$

Para $k = -1$, no podemos usar vértices intermedios, por lo que la expresión es simplemente:

$$dp(i, j, -1) = \begin{cases} d_{ij}, & \text{si hay una arista con peso } d_{ij} \text{ entre } i \text{ y } j \\ \infty, & \text{en caso contrario} \end{cases}$$

Para $k \geq 0$, o bien un camino óptimo utiliza el vértice k o bien no lo hace, de donde obtenemos:

$$dp(i, j, k) = \text{mín} \begin{cases} dp(i, k, k-1) + dp(k, j, k-1) & \text{(pasa por k)} \\ dp(i, j, k-1) & \text{(no pasa por k)} \end{cases}$$

Observación 1: Notemos que en el primer caso del mín, restamos uno al tercer argumento porque un camino óptimo no pasa dos veces por el mismo vértice (k).

Observación 2: Podemos calcular esta DP en orden creciente de k, para utilizar sólo $O(N^2)$ en memoria, en lugar de $O(N^3)$.

Una vez calculados y almacenados estos valores en un array `D[i][j]`, `D[i][j] = dp(i, j, N-1)`, podemos rápidamente iterar sobre los pares y hallar para cada sitio, el hospital más cercano.

Siguiendo una lógica similar, podemos contar la cantidad de caminos óptimos:

$$dp2(i, j, -1) = \begin{cases} 1, & \text{si hay una arista entre } i \text{ y } j \\ 0, & \text{en caso contrario} \end{cases}$$

$$dp2(i, j, k) = f(i, j, k) + g(i, j, k)$$

Donde f indica la cantidad de caminos óptimos que pasan por k, y g la cantidad de caminos óptimos que no lo hacen.

$$\text{f}(i, j, k) = \begin{cases} dp2(i, k, k-1) \times dp2(k, j-1, k-1) & \text{si un camino óptimo pasa por k} \\ 0 & \text{en caso contrario} \end{cases}$$

Se puede verificar si un camino óptimo pasa por k o no simplemente verificando la igualdad:

$$dp(i, j, k) == dp(i, k, k-1) + dp(k, j, k-1)$$

¡ya que un camino óptimo se compone de subcaminos óptimos!

Análogamente,

$$\mathrm{g}(i, j, k) = \begin{cases} dp2(i, j, k-1) & \text{si un camino óptimo no pasa por k} \\ 0 & \text{en caso contrario} \end{cases}$$

Condición que puede verificarse con:

$$dp(i, j, k) == dp(i, j, k-1)$$

Esto nos da una solución con complejidad:

$O(N^3 + M + Q)$ en tiempo,
$O(N^2)$ en memoria,

Que cumple ampliamente los límites del problema.

3.3.4. Problema 4: Tateti Zero [tatetizero]

http://juez.oia.unsam.edu.ar/#/task/tatetizero/statement

Este es un problema bastante complejo. Primero veamos cómo resolver la subtarea. Decía que en un subconjunto de casos de prueba, habría *una sola casilla vacía*. Pensemos en resolver únicamente este problema: Hay un tablero de 3×3, con cuatro cruces y cuatro círculos ubicados de manera que no hay ningún tatetí. Como empieza el círculo, sabemos que *la novena jugada la hará quien juegue círculos*. Entonces, podemos simplemente **colocar un círculo en la casilla vacía, y ver si hay tatetí**.

¿Cómo hacemos para saber si en un tablero hay alguien que ganó? Simplemente chequeando en todas las posibles direcciones, si las 3 casillas consecutivas tienen la misma letra.

Para esto entonces, tenemos que decidir cómo guardar el tablero, para luego poder acceder/consultar el valor de una casilla. Una manera es una lista de lista de caracteres. Donde la primera lista de listas contiene los 3 caracteres de la primera

fila, en forma de lista. (Donde una lista es un `vector` o `array` en C++, una `List` en Java, etcétera.)

Otra un poco más sencilla es una lista de strings, donde el primer string es la primera fila, el segundo la segunda, y el último string contiene la última fila. Algo aún más sencillo es simplemente guardar 3 strings, siempre recordando cuál string es cada fila para no perdernos y modificar el tablero.

Vamos a trabajar con 3 strings, que llaramos $F1$, $F2$, $F3$ simplemente para hacer sencilla la notación. Si guardaran una lista de strings, $F1$ sería el primer elemento de esa lista.

Notemos con $F1_1$ a la primera celda de la primera fila (y respectivamente cambiando los números para las otras celdas de otras filas). Entonces para **chequear si hay tatetí**, simplemente tenemos que chequear si $F1_1, F1_2, F1_3$ son iguales, lo mismo para la columna $F1_1, F2_1, F3_1$, y las diagonales, como por ejemplo $F1_1, F2_2, F3_3$. Esto en todas filas/columnas/diagonales. Entonces, si hiciéramos una función llamada por ejemplo `hayTateti` que recibe los tres strings, ya sabemos cómo responder `True` o `False`. Si en alguna dirección de las mencionadas hay, devolvemos `True`, y si no `False`.

Es muy importante tomarse el debido tiempo para pensar esta función, no equivocarse en números de índices para las casillas, si es necesario poner un comentario para cada dirección (por ejemplo en el ´if´ de la primera fila, escribir como comentario ´chequeo tatetí fila 1´ o algo así).

Si esta función está mal, va a fallar todo y va a ser difícil descubrir por qué.

También es importante *poner el código en una función*, ya que, para el problema completo, lo vamos a usar muchas veces, y además aunque sea sólo para la primera parte, queda mucho más prolijo porque son un montón de "ifs".

Entonces la primera parte queda completa: Recibimos las tres filas, buscamos (con un for o simplemente viendo cada una de las 9 casillas) el punto (`'.'`), que sabemos que va a haber uno solo, reemplazamos ese caracter por una `'O'`, y llamamos a la función `hayTateti`. Si hay, le ponemos una `'G'` a esa celda. Si no, una `'E'` ya que sabemos que como no había tatetí al principio, no podrá haber tatetí de las `'X'` si sólo agregamos una `'O'`. Imprimimos lo que haga falta y listo.

Algo importante a tener en cuenta, es que si el tablero fuera por ejemplo de 10×10, entonces buscar una casilla con un puntito se hace prácticamente imposible si queremos mirarlas de a una. Para eso vamos a necesitar un **for**. Y si guardamos las filas en 10 variables de tipo `string` distintas, vamos a necesitar un **for** por cada

fila y quedaría todo súper engorroso. Por eso es que convendría muchísimo guardar una lista de strings: simplemente hacemos un **for** para pasar por cada elemento de la lista (es decir, cada fila), y adentro de ese otro **for** para ver cada celda, y con un **if** vemos si el valor es un punto o no.

Ahora que vimos la subtarea, y que sabemos cómo chequear si en un tablero hay tatetí, veamos cómo resolver el problema en su totalidad.

Lo que queremos hacer es simular un juego **óptimo**, es decir, donde cada jugador hace lo mejor para ganar, asumiendo que la otra persona hará lo mismo. Entonces, lo que podríamos programar es una función que nos diga la jugada óptima dado el tablero actual, y a quién le toca. ¿Cómo podemos pensar a la función?

Supongamos que le toca a A, y queremos responder "si A juega en la celda c, ¿cómo sería el resultado cuando le toque a mi oponente?". Para responder a esa pregunta, podemos pensar en modificar la celda c como si A jugara ahí, y luego pensar "qué es lo mejor que le puede pasar a B con este nuevo tablero". Y para responder a esta pregunta, podemos **llamar a esta misma función, pero ahora pensando que es el turno de** B. Si esta función nos dice "con este tablero, B ganaría", entonces A seguro que no va a querer jugar ahí. Pero si la función nos dice "con este tablero, B perdería", entonces A sí va a querer jugar ahí y ganar (ya que la función nos dice lo mejor que le puede pasar a B, en este caso no podría evitar perder).

Esta es una función **recursiva**, que se llama a sí misma. ¿Cómo funciona esta función? Lo que hará será recorrer todas las celdas vacías, ubicar la letra que corresponda en esta celda, y llamar a la función con el nuevo tablero, indicando que le toca al otro jugador. Si alguna celda nos da que nuestro contrincante perdería, entonces jugando ahí ganamos. Ahora si la función nos da que la otra ganaría, entonces ahí perdemos. Y si nos da que hay empate, entonces jugando ahí hay empate.

Se puede leer de recursión acá: http://wiki.oia.unsam.edu.ar/algoritmos-oia/recursion , y un poco más de utilidad para este problema acá: http://wiki.oia.unsam.edu.ar/algoritmos-oia/backtracking

Ahora, como toda función recursiva, necesita una **situación en la que se deje de llamar a sí misma**, ya que de no ser así se quedaría colgada haciendo infinitas llamadas a sí misma. ¿En qué situación podemos dejar de llamarla? Pensemos un segundo.

¡Cuando el tablero está lleno! Si se llama a la función con el tablero lleno, como no hay casilla vacía para llenar, ya sabremos seguro el resultado de la partida. Pero

hay una situación más donde sabemos el resultado antes de llenar el tablero ...

¡Cuando el partido terminó antes, porque hay tatetí!. Y acá es cuando vamos a usar la función `hayTateti` que vimos antes. Si a la función le pasamos un tablero indicando que le toca a X, y hay tatetí de O, entonces la función devolverá ´pierde´. Y si no hay tateti, buscará las casillas vacías y hará de vuelta la recursión. Si el tablero está lleno, y no hay tateti, la respuesta será ´empata´.

Entonces veamos un pseudocódigo de esta función, que quizás ilustre un poco mejor la situación. En el pseudocódigo, `tablero` será una lista de 3 strings, y `turno` será una letra, `X` u `O` según a quién le toque.

Algorithm 17 Mejor resultado dado un tateti y a quién le toca

```
1: function mejorResultado(tablero : ARREGLO DE PALABRAS, turno : LETRA ) (PALABRA)
2:     if ultimoDigitoVisto ≤ digitoActual then
3:         return "pierde"
4:     puedoGanar ← False
5:     puedoEmpatar ← False
6:     hayAlgunaVacia ← False
7:     oponente ← 'X'
8:     if turno == 'X' then
9:         oponente ← 'O'
10:    for i = 0, 1, 2 do
11:        for j = 0, 1, 2 do
12:            if tablero[i][j] == '.' then
13:                hayAlgunaVacia ← True
14:                tablero[i][j] ← turno
15:                resultado ← mejorResultado(tablero, oponente)
16:                if resultado == "pierde" then ▷ Jugando en (i,j) el rival pierde
17:                    puedoGanar ← True
18:                else if resultado == "empata" then
19:                    puedoEmpatar ← True
20:                tablero[i][j] ← '.'     ▷ Como modificamos el tablero, para seguir probando otras casillas, volvemos a su estado inicial
21:    if hayAlgunaVacia == False then
22:        return "empata"
23:    if puedoGanar == True then
24:        return "gana"
25:    if puedoEmpatar == True then
26:        return "empata"
27:    return "pierde"
```

Entonces, ahora que tenemos programada esa función, una vez que recibimos `tablero`, lo que tendríamos que hacer es:

1. Ver a quién le toca. Para esto, con un par de **for** contamos cuántas `'X'` y `'O'` hay. Si hay igual cantidad le toca a `'O'` y si hay una `'O'` más, le toca a `'X'`.

2. Construir un tablero aparte, donde iremos poniendo `'G'`, `'E'` o `'P'` en las vacías, pero *no modificar el tablero real del juego*.

3. Recorrer todas las celdas. Si encontramos un punto (`'.'`), lo reemplazamos por la letra que corresponda según lo que hicimos en el ítem 1, y averiguamos el resultado llamando a la misma función, pero con el turno de la otra persona. Si esa función nos devuelve `"pierde"`, entonces sabemos que jugando acá vamos a ganar. Si nos devuelve `"empata"`, sabemos que jugando acá empatamos, y si devuelve `"gana"`, sabemos que jugando acá perdemos.

4. Mientras recorremos las celdas y vamos averiguando si corresponde poner una `'G'`, `'E'` o `'P'`, colocamos estas letras en nuestra copia del tablero.

5. Si en algún lado pusimos una `'G'`, la situación es ganadora. Si en algún lado pusimos una `'E'`, la situación es empatadora. Y si fueron todas `'P'`, es perdedora. Para saber esto podemos poner un contador que cuenta la cantidad de `'G'` y `'E'` que ponemos, o simplemente una variable booleana que nos diga si colocamos una `'G'` o no (y lo mismo para `'E'`).

6. Imprimimos la situación seguida de nuestro tablero paralelo con las letras correspondientes.

Capítulo 4

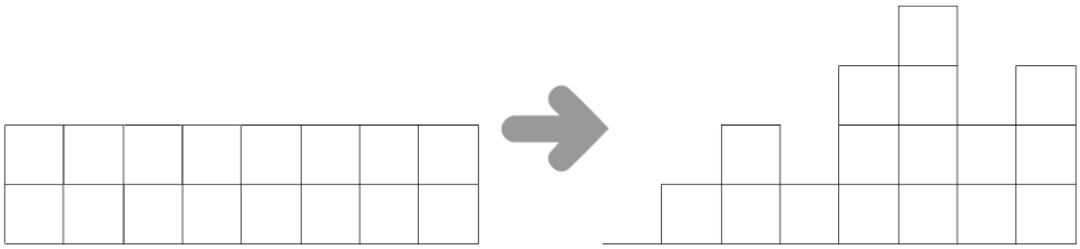
Certamen Nacional

4.1. Nivel 1

4.1.1. Problema 1: Ordenando la habitación [zapatos]

http://juez.oia.unsam.edu.ar/#/task/zapatos/statement

El enunciado nos dice que originalmente había P pilas de cajas de zapatos, y que todas las pilas tenían la misma cantidad de cajas. La idea clave es que al mover cajas de zapatos de una pila a la otra, **la cantidad total de cajas no varía** (la caja que sale de una pila, se agrega en otra).

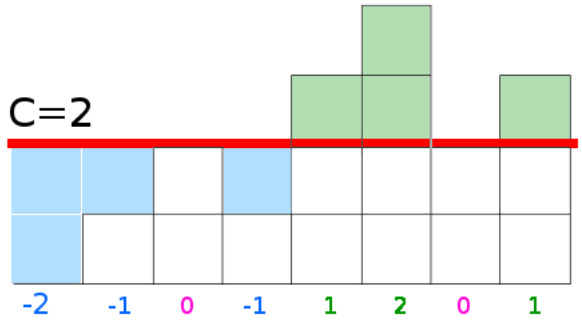


Como originalmente todas las pilas tenían la misma cantidad de cajas de zapatos, llamando C a esa cantidad, sabemos que en total habrá $C \cdot P$ cajas de zapatos (C cajas en cada una de las P pilas). Por otro lado, esta cantidad total la podemos obtener sumando la cantidad que hay actualmente en cada pila. Si llamamos $SUMA$ a la cantidad total de cajas de zapatos que vienen en la entrada, tenemos la siguiente relación:

$$C \cdot P = SUMA \Rightarrow C = \frac{SUMA}{P}$$

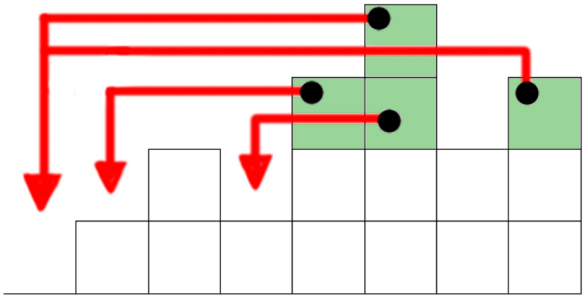
Donde tanto $SUMA$ como P son conocidos o podemos calcularlos a partir de los datos que vienen en el enunciado.

Finalmente, para cada pila podemos calcular cuántas cajas de zapatos le faltan o le sobran respecto de la cantidad que tenían originalmente (que es C). Llamando $cajas_i$ a la cantidad de cajas que tiene actualmente la pila i, podemos concluir que $cajas_i - C$ será un número positivo si le sobran cajas a la i-ésima pila, y será un número negativo si le faltan cajas a dicha pila. Llamemos S a la *cantidad total de cajas de zapatos que sobran.*



Para volver a cada pila a la cantidad original de cajas de zapatos, al menos debemos mover todas las que sobran, es decir S. Si cada una de estas cajas la ponemos en algún lugar donde falten, habremos hecho exactamente esa mínima cantidad de movimientos necesaria (y por lo tanto lo habremos realizado de manera óptima).

Para calcular S se puede sumar $cajas_i - C$ cuando este número sea positivo. En el ejemplo la respuesta es 4 y corresponde a sumar las cajas de zapatos que sobran en cada pila donde sobran cajas (la cantidad de cajas verdes en la imagen). Notar que la cantidad de cajas que sobran es igual a la cantidad de cajas que faltan en todo momento (si habría más de una que de la otra, la cantidad total de cajas habría variado).



Algorithm 18 Solución al Problema 1 de Nivel 1 Nacional 2018 - zapatos

```
1: function ZAPATOS(P : ENTERO, CAJAS : ARREGLO DE ENTEROS)(ENTERO)
2:     SUMA ← 0
3:     for i = 0 ... P-1 do
4:         SUMA ← SUMA + cajas[i]
5:     C ← SUMA/P
6:     cajas_a_mover ← 0
7:     for i = 0 ... P-1 do
8:         if cajas[i]-C > 0 then
9:             cajas_a_mover ← cajas_a_mover + (cajas[i]-C)
10:    return cajas_a_mover
```

4.1.2. Problema 2: Procesador de textos [pluralizador]

http://juez.oia.unsam.edu.ar/#/task/pluralizador/statement

En este problema se nos da un conjunto de sustantivos en singular y nuestra tarea es pasarlos al plural *siguiendo las reglas del enunciado para calcular el plural de una palabra*. Estas reglas dicen:

1. Si el sustantivo **termina en vocal** se agrega **"s"**.
2. Si el sustantivo **termina en "s" o "x"** entonces **queda igual**.
3. Si el sustantivo **termina en z** entonces **se cambia la "z" por "ces"**.
4. Si el sustantivo **termina en otra consonante** entonces **se agrega "es"**

Además, nos piden que almacenemos en un arreglo cuántas veces fue utilizada cada regla. Por lo tanto, el problema en cuestión pide solamente implementar correctamente lo que está descrito en el enunciado. Una forma posible de hacerlo se muestra en el siguiente pseudocódigo.

Se asume que el arreglo `cantidadesPorRegla` comienza inicializado con 0, y que podemos preguntar directamente si un cierto elemento pertenece a un contenedor (si tuviéramos que hacerlo nosotros tendríamos que iterar el contenedor, no debería ser una dificultad mayor). Además utilizaremos el símbolo `+`, para *concatenar* dos palabras.

Una observación del enunciado es que en este problema no hace falta devolver nada, sino que *al terminar de ejectuarse, las variables* (en este caso arreglos) *que vienen en la entrada deben modificarse para almacenar la respuesta.*

Algorithm 19 Solución al Problema 2 de Nivel 1 Nacional 2018 - pluralizador

```
1: function PLURALIZADOR(N : ENTERO, PALABRAS : ARREGLO DE PALABRAS,
   CANTIDADESPORREGLA : ARREGLO DE ENTEROS)
2:     vocales ← ['a','e','i','o','u'] ▷ Solo vienen palabras en minúscula
3:     for i = 0 ... N-1 do                                  ▷ Para cada palabra:
4:         largo ← palabras[i].largo()
5:         if palabra[i][largo-1] ∈ vocales then                    ▷ Regla 1
6:             palabra[i] ← palabra[i] + 's'
7:             cantidadPorRegla[0] ← cantidadPorRegla[0]+1
8:         else if palabra[i][largo-1] ∈ ['s','x'] then             ▷ Regla 2
9:             cantidadPorRegla[1] ← cantidadPorRegla[1]+1
10:        else if palabra[i][largo-1] == 'z' then                  ▷ Regla 3
11:            palabra[i][largo-1] ← 'c'
12:            palabra[i] ← palabra[i] + 'es'
13:            cantidadPorRegla[2] ← cantidadPorRegla[2]+1
14:        else                                                     ▷ Regla 4
15:            palabra[i] ← palabra[i] + 'es'
16:            cantidadPorRegla[3] ← cantidadPorRegla[3]+1
```

4.1.3. Problema 3: Armando cartas numerológicas [numerologo]

http://juez.oia.unsam.edu.ar/#/task/numerologo/statement

El enunciado describe un proceso que se le debe realizar a un número para obtener el siguiente en una *secuencia de buena suerte*. Concretamente, para generar el siguiente número se realiza lo siguiente:

1. Se factoriza el número en factores primos.
2. Se ordenan los factores (con repeticiones) de menor a mayor.
3. Se concatenan todos los números para formar uno más grande.

Si sabemos realizar correctamente cada paso, entonces el problema resulta relativamente sencillo, dado un número en la entrada debemos generar el siguiente hasta alcanzar un número primo u obtener un número mayor que 10,000.

¿Cómo sabemos que el proceso termina?¿Por qué no puede ciclar entre dos (o más números) que no son primos? Una forma sencilla es implementar el proceso y probar todos los casos posibles en la computadora. Otra forma es ver que si el número no es primo, entonces el siguiente número obtenido por el proceso es estrictamente mayor que el original.

La idea detrás de que la secuencia es **estrictamente creciente** cuando el número N dado no es primo, podemos escribirlo como $N = A \cdot B$, ambos A y B mayores o iguales que 2.

Llamemos c a la cantidad de cifras de B, entonces la concatenación de A y B está dada por el número $A \cdot 10^c + B$. Notemos que $10^c > B$ (pues c es la cantidad de cifras). Finalmente queremos probar que $A \cdot 10^c + B > A \cdot B \iff 10^c \cdot A > A(B-1) \overset{\text{porque } A>0}{\iff} 10^c > B - 1$, lo cual ya probamos que ocurre.

Finalmente, si tenemos la factorización de un número $N = p_1 \cdot p_2 \cdot \cdots \cdot p_k$, con $p_i \leq p_{i+1}$ y k la cantidad de factores primos que tiene el número (contando repetido). Aplicando el argumento secuencialmente con los primeros dos números, conluimos que $N = p_1 \cdot p_2 \ldots p_k < (p_1 p_2) \cdot p_3 \ldots p_k < (p_1 p_2 p_3) \ldots p_k < \ldots < \underbrace{(p_1 p_2 \ldots p_k)}_{\text{siguiente en la secuencia}}$, que es lo que queríamos probar.

Veamos entonces cómo implementar cada una de las operaciones. Para **factorizar** un número, podemos simplemente iterar por todos los números menores y chequear si lo dividen o no. En caso de que lo divida, debemos *agregar todas las apariciones de ese divisor primo*. Otra opción es usar la *criba de eratóstenes* pero dadas las cotas del enunciado realmente no hace falta. En caso de estar interesados en esto último, recomendamos leer el post en la wiki http://wiki.oia.unsam.edu.ar/algoritmos-oia/enteros/criba-de-eratostenes.

Para **ordenar** podemos utilizar cualquier algoritmo de ordenamiento que conozcamos (la cantidad de números a ordenar es la cantidad de factores primos de un número, que es $\mathcal{O}(\lg N)$). En muchos lenguajes de programación ya hay implementado un algoritmo de ordenamiento eficiente (`sort` en C++ por ejemplo). Veamos cómo utilizar todo esto para calcular la lista ordenada de factores de un número como así también la cantidad de factores primos distintos que tiene.

Algorithm 20 Factores ordenados de N y su cantidad de factores primos distintos

```
1:  function FACTORIZAR(N: ℕ, listaFactores: 𝔸ℝ. DE ℕ)(ℕ)
2:      factores_distintos ← 0
3:      divisor_primo ← 2
4:      while divisor_primo ≤ N do  ▷ Iterando en orden nos ahorramos ordenar
5:          if (N % divisor_primo) == 0 then
6:              factores_distintos ← factores_distintos + 1
7:              while (N % divisor_primo) == 0 do ▷ Contar todas las apariciones
8:                  lista_factores.agregar_al_final(divisor_primo)
9:                  N ← N/divisor_primo
10:         divisor_primo ← divisor_primo + 1
11:     return factores_distintos
```

Restaría ver cómo **concatenar** dos números, pero en la demostración de que la secuencia es creciente vimos una opción para hacer eso. Si tenemos dos números A y B, si calculamos c la cantidad de cifras de B, entonces la concatenación $(AB) = 10^c \cdot A + B$. Auxiliarmente, veamos cómo calcular 10^c para un número B cualquiera.

Algorithm 21 Dado un número B calcula 10^c, con c la cantidad de cifras de B

```
1: function POT10CIFRAS(B : ℕ)(ℕ)
2:     respuesta ← 1
3:     while B > 0 do
4:         respuesta ← respuesta * 10
5:         B ← ⌊B/10⌋
6:     return respuesta
```

Luego basta con repetir la operación a todos los números de la lista. En el siguiente algoritmo `lista_factores` es un arreglo de largo `K`.

Algorithm 22 Concatenar los números de una lista

```
1: function CONCATENAR(LISTAFACTORES: 𝔸ℝ. DE ℕ𝕊)(ℕ)
2:     respuesta ← 0
3:     for i = 0 ... K-1 do
4:         respuesta ← respuesta * pot_10_cifras(lista_factores[i])
5:         respuesta ← respuesta + lista_factores[i]
6:     return respuesta
```

Juntando todo esto, un pseudocódigo del algoritmo que resuelve el problema se encuentra a continuación. En el mismo asumimos que `secuencia` es un contenedor que comienza vacío.

Algorithm 23 Solución al Problema 3 de Nivel 1 Nacional 2018 - numerologo

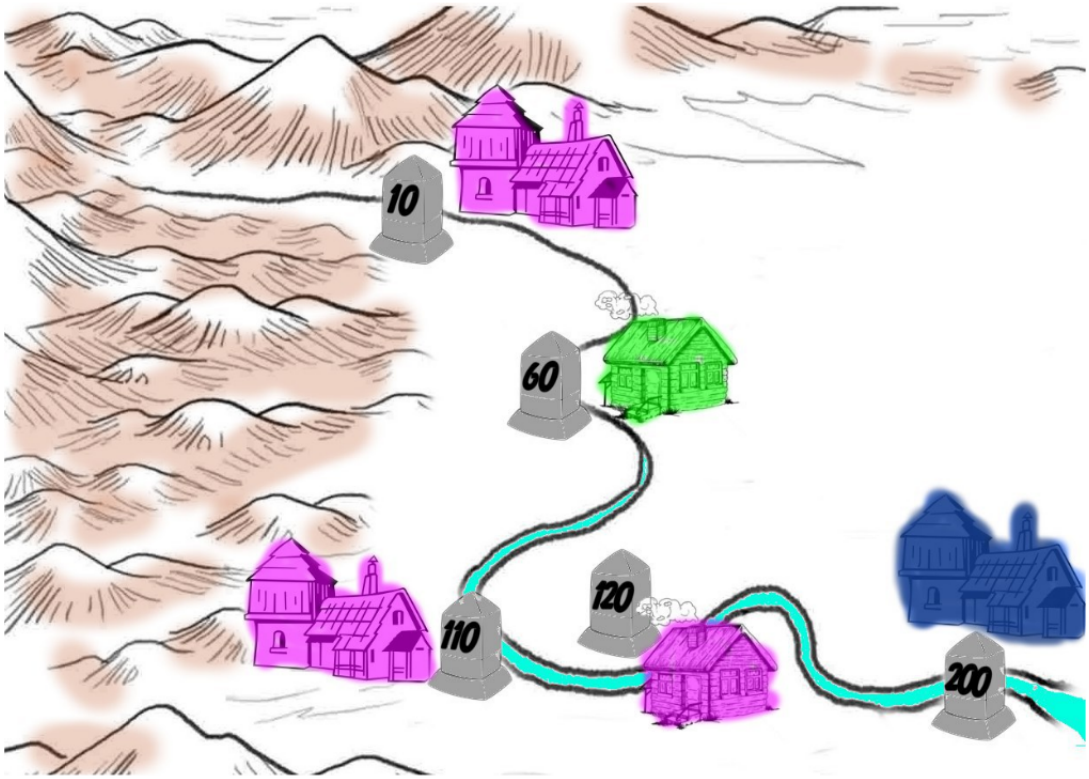
```
1: function NUMEROLOGO(N: ℕ, SECUENCIA: 𝔸ℝ. DE ℕ𝕊)(ℕ)
2:     lista_factores ← []
3:     factores_distintos ← factorizar(N,lista_factores)
4:     actual ← N
5:     while actual ≤ 10000 do
6:         secuencia.agregar_al_final(actual)
7:         actual ← concatenar(lista_factores)
8:         lista_factores ← []
9:         auxiliar ← factorizar(actual,lista_factores)
10:    return factores_distintos
```

4.2. Nivel 2

4.2.1. Problema 1: Dividiendo pueblos [pueblitos]

http://juez.oia.unsam.edu.ar/#/task/pueblitos/statement

De todas las asignaciones válidas de los pueblos a los hijos, se busca una que **maximice la mínima distancia entre dos pueblos vecinos asignados a hijos distintos**. Notemos que la longitud de estos intervalos corresponde a la diferencia entre la ubicación de dos pueblos consecutivos. La idea clave para resolver este problema radica en que si se tiene una configuración donde todos los pueblos de un mismo hijo no son consecutivos, entonces **se puede obtener otra configuración donde todos los pueblos de todos los hijos sean consecutivos sin que la respuesta empeore**. Quizá la imagen del enunciado no nos ayuda a ver esto, pero puede verse que existe una configuración que no empeora la situación intercambiando la asignación de los primeros dos pueblos.



Veamos cómo probar esto. Si tenemos un hijo del rey, que llamaremos A, con pueblos no consecutivos, entonces hay dos rangos *no vacíos* $\mathcal{A}_1 = [l_1, r_1]$, y $\mathcal{A}_2 = [l_2, r_2]$ donde todos los pueblos en ambos rangos corresponden al hijo A. Sin pérdida de generalidad podemos asumir que $r_1 < l_2$ (de no ser así intercambiamos

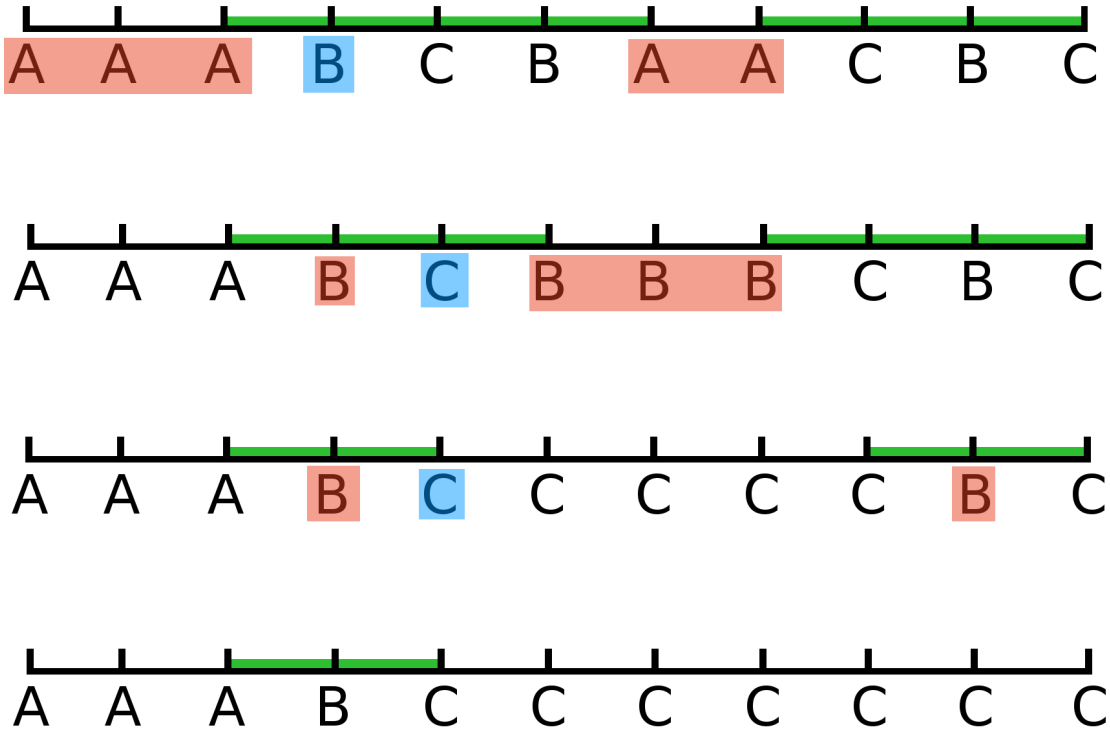
los roles de $\mathcal{A}_1$ con $\mathcal{A}_2$) y que los rangos son *maximales* (fueron tomados lo más grande posibles, no se pueden extender ni a izquierda ni a derecha con pueblos del hijo A).

Además de estos dos rangos, necesariamente debe haber otro hijo del rey, que llamaremos B, entre r_1 y l_2, pues de no ser así $\mathcal{A}_1$ y $\mathcal{A}_2$ estarían pegados y los hipotéticos pueblos de A serían consecutivos. Consideremos entonces la configuración que toma todos los pueblos igual que antes, salvo a los que están comprendidos en $\mathcal{A}_2$ y ahora se los asigna al hijo B. Veamos que la situación es factible y que no empeoró.

Para ver que es factible solo hay que ver dos cosas. Primeramente que *cada pueblo está siendo entregado a un hijo*, lo cual ocurre trivialmente (todos los pueblos tienen la misma asignación que antes salvo los de $\mathcal{A}_2$ a los cuales les asignamos otro hijo, de todas formas, todo pueblo tiene a un hijo asignado). Y por último tenemos que ver que *cada hijo recibe al menos un pueblo*. Esto ocurre gracias a que la configuración anterior era válida, como al único hijo al que le sacamos peublos es a A, el único que podría no cumplir esto es A. Sin embargo, todavía tiene al menos todos los pueblos de $\mathcal{A}_1$, por lo que la asignación sigue siendo válida.

Nos resta ver que la respuesta no empeoró. Solo nos interesan las vecindades entre pueblos asignados a hijos distintos. Observemos que los únicos pueblos que cambiaron son los de $\mathcal{A}_2$. Con esto en mente, notemos que entre los pueblos consecutivos interiores a $\mathcal{A}_2$ la situación no cambió, pues siguen siendo pueblos asignados a un mismo hijo (solo que ahora es B en lugar de A). Por lo tanto, las únicas vecindades que cambiaron ocurren inmediatamente a la izquierda de l_2 e inmediatamente a la derecha de r_2. Como tomamos $\mathcal{A}_2$ de forma maximal, sabíamos que estos pueblos inmediatamente aledaños son distintos de A, por lo tanto antes estaban siendo considerados en la respuesta, y por ende no hemos empeorado la situación (en todo caso, si algún vecino aledaño era B podríamos haber mejorado la respuesta).

Veamos esto con un esquema. Marcamos con verde los intervalos que son considerados en la respuesta (formalmente, la respuesta será la máxima longitud de estos intervalos), notemos que los intervalos verdes que quedan al final son siempre un subconjunto de los originales por lo que explicamos. Los pueblos marcados en naranja representan a $\mathcal{A}_1$ y $\mathcal{A}_2$ en cada paso, y el pueblo marcado en azul es uno cualquiera intermedio que es el que le será asignado a $\mathcal{A}_2$ en el paso siguiente.



Algo importante que hay que notar es que **en ningún momento hace falta implementar en la computadora** todo esto. Todo lo que hicimos fue un razonamiento de "lápiz y papel" que nos ayudará en la resolución del problema. Más aún en una instancia de competencia, con suficiente confianza en la intuición propia (lo cual es un arma de doble filo) este razonamiento podría haber sido utilizado sin una demostración formal.

Utilizando que ahora podemos asumir que los pueblos asignados a un mismo hijo son consecutivos es fácil ver que *la cantidad de intervalos distintos hijos en los extremos serán exactamente* $K-1$ (pues esa es la cantidad de veces que pasamos de un hijo a otro, y todos deben ser utilizados). Nuestro objetivo es *maximizar la mínima distancia de uno de estos intervalos* que tienen distintos hijos en sus extremos, ¿cómo deberíamos tomarlos entonces? ¡Lo más grandes posibles!

Por lo tanto, la solución al problema consiste en tomar todas las diferencias entre pueblos consecutivos, y encontrar una asignación factible que tenga distintos hijos solamente en los $K-1$ intervalos más grandes. Más aún, la máxima mínima distancia buscada corresponde a la $(K-1)$-ésima distancia entre consecutivos más grande. Veamos un pseudocódigo de cómo calcular este valor y generar una asignación factible. Se asume que `ubicacion` es un arreglo de tamaño `N` que viene ordenado (como indica el enunciado). La respuesta estará en el arreglo `asignacion` que tiene tamaño N.

Algorithm 24 Solución al Problema 1 de Nivel 2 Nacional 2018 - pueblitos

```
1: function PUEBLITOS(K : ENTERO, UBICACION : ARREGLO DE ENTEROS)(ENTERO)
2:     diferencias ← []                          ▷ Arreglo de tamaño N-1
3:     for i = 0 ... N-2 do
4:         diferencias[i] ← ubicacion[i+1] - ubicacion[i]
5:     diferencias.ordenar()      ▷ En orden decreciente (mayor a menor)
6:     hijo ← 1
7:     dist_respuesta ← diferencias[K-2]     ▷ Notar que indexamos desde 0
8:     for i = 0 ... N-1 do
9:         asignacion[i] ← 1          ▷ Asignamos todo al hijo 1 para empezar
10:    for i = 1 ... N-1 do
11:        if hijo < K and ubicacion[i]-ubicacion[i-1] ≥ dist_respuesta then
12:            hijo ← hijo + 1
13:        asignacion[i] ← hijo
14:    return dist_respuesta
```

La complejidad temporal de este algoritmo es $\mathcal{O}(N \lg N)$, pues debemos ordenar un arreglo de tamaño N. El resto del algoritmo es $\mathcal{O}(N)$, ya que solo hacemos iteraciones lineales sobre arreglos de tamaño $\mathcal{O}(N)$.

4.2.2. Problema 2: Vigilando la ciudad [vigilantes]

http://juez.oia.unsam.edu.ar/#/task/vigilantes/statement

4.2.2.1. Subtarea $x_i \in \{1, 2\}$

En este caso muy especial, hay solamente dos calles norte-sur que contienen a todos los vigilantes. Podemos observar que en tal situación, como máximo son necesarios 3 envíos de señales, así que podemos analizar la situación de cada vigilante directamente:

- Aquellos que están en la misma x o la misma y que el jefe, los marcamos a distancia 1.
- Si en la misma y que el jefe existió otro vigilante (que por el ítem anterior, estaba a distancia 1), entonces podemos cubrir ambas calles sin problema, y ya podemos marcar todos los vigilantes que quedan con distancia 2.
- Si no se da la situación del paso anterior, pero hay algún vigilante de los que están en la misma x que el jefe, que comparte la y con alguno de los que falta, podemos marcar con $d = 2$ a todos aquellos vigilantes que están en la otra x

pero tienen un y compartido con otro vigilante, y con $d = 3$ a todos los demás restantes.

- Si tampoco se da la situación anterior, entonces todos los vigilantes que quedaron sin marcar quedan con el valor -1, pues no se puede llegar a ellos.

Lo anterior puede implementarse eficientemente con algunos fors + ifs, y chequeos de pertenencia a conjuntos (para verificar si hay alguien con la misma y).

4.2.2.2. Subtareas de tamaño pequeño

Este es un problema clásico de camino mínimo, y puede modelarse naturalmente con grafos.

El modelo más natural es poner un nodo por cada vigilante, y una arista entre dos de ellos exactamente cuando los correspondientes vigilantes están en la misma fila o en la misma columna. Luego, el dato que pide calcular el enunciado no es más que la distancia desde un nodo origen (el que corresponde al jefe de todos los vigilantes) y todos los demás nodos.

Esto puede calcularse utilizando el algoritmo de BFS. Se puede leer sobre este algoritmo en la wiki: http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/bfs.

Por cada nodo, para poder ejecutar el algoritmo de BFS, lo importante es poder iterar sus vecinos. Es decir, por cada vigilante debemos poder computar aquellos que ve. Hay dos mecanismos básicos, y cuál conviene dependerá de los tamaños de la subtarea:

- Iterar cada uno de los N vigilantes, y verificar si tiene la misma coordenada x o la misma coordenada y que el vigilante actual. Tiempo $O(N)$

- Iterar todas las ubicaciones del mapa en la misma x que el vigilante actual, para ver en cada una si hay vigilante o no. Luego lo mismo con las y. Esto toma tiempo $O(x_{max} + y_{max})$, y además es necesario llenar inicialmente una matriz de tamaño $x_{max} \times y_{max}$ que indica el número de vigilante que hay en cada esquina, si lo hay.

Utilizando estas ideas se consiguen 71 puntos en este problema.

4.2.2.3. Solución completa

Para obtener una solución completa de 100 puntos, es necesario realizar el BFS más eficientemente, sin que sea necesario volver a explorar todos los vecinos de cada nodo. Esto es así porque, por ejemplo, si todos los vigilantes están en una misma calle, se ven todos entre sí, y por lo tanto es inevitable un tiempo de orden N^2 si se iteran todas las vecindades explícitamente.

Hay varias maneras de implementar un BFS más eficiente para este caso especial. La que proponemos consiste en dar vuelta los roles: en el grafo anterior, los vigilantes son nodos, y las calles indican las aristas. Lo que haremos ahora será que **las calles sean los nodos**, y que **los vigilantes sean las aristas**. La arista de un vigilante une siempre exactamente dos calles: la vertical y la horizontal que corresponden a la esquina del vigilante.

La pregunta a responder ahora es la distancia a la cual **cada arista** del grafo está de una cierta arista inicial. Como los caminos de nodos y de aristas son básicamente equivalentes, lo que podemos hacer es lanzar un BFS sobre este grafo, con dos nodos iniciales a distancia 0: Los extremos de la arista que corresponde al jefe de vigilantes.

Al terminar, la distancia a cada arista (u, v) sera $1 + min(d_u, d_v)$, excepto la arista del jefe de vigilantes que tendrá distancia 0 por ser la inicial (y no 1 como indicaría la cuenta).

Esta solución da un algoritmo de tiempo lineal, pues el tamaño del grafo (es decir, la cantidad de nodos y aristas) es proporcional a la cantidad de vigilantes.

4.2.3. Problema 3: Viaje de egresados [egresados]

http://juez.oia.unsam.edu.ar/#/task/egresados/statement

Este era indudablemente el problema más difícil del certamen nivel 2. No se esperaba que ningún participante apuntara a 100 puntos en este problema durante la prueba, sino solamente a puntajes parciales.

La clave para resolver este problema es utilizar búsqueda binaria en la respuesta: Es decir, haremos búsqueda binaria en el tiempo T permitido entre una habitación de estudiante y su coordinador más cercano. Para cada valor de T, nuestro algoritmo deberá poder indicar si es posible ubicar los C coordinadores de forma tal que todos los estudiantes puedan ser alcanzados por alguno de los coordinadores en un tiempo máximo T.

Se pueden leer ejemplos y explicaciones sobre esta técnica de búsqueda binaria

sobre la respuesta en la web de OIA: http://www.oia.unsam.edu.ar/charlas/

4.2.3.1. Subtarea $P = 1$

El caso particular en el que tenemos un único piso daba 16 puntos y es muchísimo más simple. En estos casos el descanso puede reemplazarse por una habitación no reservada, pues lo único que lo diferencia de un espacio vacío normal es la capacidad de cambiar de piso allí, por lo cual si no hay otros pisos podemos pensar que tenemos nada más letras N y S.

Para este caso podemos dar un **algoritmo goloso** que es completamente válido: Si consideramos la primera habitación reservada (es decir la primera S), deberá ser cubierta por alguno de los coordinadores que utilicemos, es decir, o bien debe tener un coordinador en esa misma habitación, o bien el coordinador más cercano debe estar a una distancia máxima T. Pero como todas las demás habitaciones están más a la derecha, lo que conviene es colocar un coordinador en la habitación más a la derecha que todavía pueda cubrir esta habitación, ya que cualquier otra elección cubre un subconjunto de las habitaciones cubiertas por esa opción de más a la derecha posible.

El algoritmo goloso consiste en repetir este paso hasta cubrir todas las habitaciones: En cada paso, se toma la primera habitación sin cubrir (la primera S de la cadena que aún no fue cubierta), y de todas las habitaciones (incluyendo esa misma), se elige la de más a la derecha entre todas las que cubren esa habitación (que podría ser esa misma habitación, ya sea porque es la única que queda por cubrir, o porque todas las demás que quedan están demasiado lejos, a más de T pasos).

Como este algoritmo goloso cubrirá el piso utilizando la mínima cantidad posibles de coordinadores, basta verificar si la cantidad usada fue a lo sumo C: si se requirieron más de C coordinadores, T no era un valor posible así que la respuesta tendrá que ser un T mayor, mientras que si utilizaron C o menos coordinadores, es posible obtener una separación T y la respuesta final será T o menos. Continuando con la búsqueda binaria que utiliza el algoritmo goloso en cada paso, obtenemos la respuesta.

4.2.3.2. Subtarea $P = 2$

Para esta subtarea, tenemos la complicación de que al haber dos pisos, es posible que un coordinador de un piso cubra habitaciones que están en el otro.

Podemos primero calcular el T óptimo cuando asumimos que esto no ocurre: Es decir, haciendo búsqueda binaria en T como antes, podemos computar para cada

piso en forma independiente la mínima cantidad de coordinadores que necesita, digamos que son C_1 y C_2: y entonces, si resulta $C_1 + C_2 \leq C$, se puede con ese T, y sino, no se puede. Este método nos dará el T óptimo en el caso de que los coordinadores no crucen de piso.

Falta considerar el caso en que los coordinadores cruzan de piso. Una observación clave para simplificar el estudio es que no tiene sentido que un coordinador pase del piso 1 al 2, y que también un coordinador pase del 2 al 1. Si así fuera, podemos observar que para alguno de los dos pisos, una cierta habitación está siendo cubierta por un coordinador del otro piso, cuando también podría ser cubierta en igual o menor tiempo por un coordinador del mismo piso. De manera similar, podemos asumir que existe una única habitación de la cual sale un coordinador que puede cruzar al otro piso: ya que si hubiera más de una, solamente la más cercana al descanso es relevante, pues la otra tarda más en llegar al otro piso.

Teniendo esto en cuenta, como hay un máximo de 2000 habitaciones, podemos realizar fuerza bruta probando las 2000 posibilidades de habitación que contiene al coordinador que puede ir al otro piso. Lo que hacemos entonces es probar para cada habitación, cuál es la mejor solución posible que usa esa habitación para poner allí un coordinador, y tal que ese coordinador es el único que vamos a considerar que puede cruzar al otro piso. Por las observaciones anteriores, considerando todas estas posibilidades y tomando la mejor, obtenemos la respuesta óptima.

Para esto marcamos esa habitación y todas las que alcanza como ya cubiertas, **incluso aquellas que alcanza en el otro piso**. Luego, en cada piso, utilizamos exactamente el mismo goloso del caso $P = 1$, con la salvedad de que como vimos, ya iniciamos con un coordinador ubicado y ciertas habitaciones cubiertas, pero el método sigue siendo el mismo: Tomamos siempre la primera habitación (la más a la izquierda) del piso que aún no fue cubierta, y la cubrimos poniendo el coordinador lo más a la derecha posible.

Notar que en cuanto a código fuente, la implementación de esta idea es muy poco código adicional además del que ya se tenía del caso $P = 1$, pues basta con un for que itere las 2000 habitaciones, y el código para marcar y procesar las que esta habitación especial preelegida ya cubre.

4.2.3.3. Solución completa

Utilizando el algoritmo explicado anteriormente, se obtienen 40 puntos. Se explica a continuación un algoritmo de programación dinámica que puede resolver eficientemente todos los casos. Como siempre, utilizaremos búsqueda binaria en la respuesta, de modo que el problema que nos enfocamos en resolver es calcular

la mínima cantidad de coordinadores tal que todos quedan vigilados a distancia máxima T.

La idea principal es utilizar en cada piso el método goloso, pero ahora solamente desde las puntas hacia el descanso. Es decir, como antes, tomamos el primer S no cubierto, y lo cubrimos con el coordinador más a la derecha posible, y repetimos, pero solo mientras que esta habitación S **no sea alcanzable desde el descanso**[1]. En cuanto esta habitación se encuentra a menos de T del descanso, dejamos pendiente su cubrimiento con el goloso. Lo mismo realizamos tomando el último S no cubierto, y lo cubrimos con el coordinador más a la izquierda posible, siempre y cuando la habitación se encuentre a T o más unidades del descanso. Es decir, realizamos el goloso desde las dos puntas, aproximándose al descanso.

Este mismo proceso podemos completarlo en cada piso nuevo que procesemos. De esta forma, en cada piso, las habitaciones que posiblemente quedaron sin cubrir están a una distancia máxima $T-1$ del descanso. Esto significa que, si para cubrir una de estas habitaciones se utiliza un coordinador de ese mismo lado del descanso en ese mismo piso, basta con un coordinador para cubrir todas las que haya, y además ese coordinador conviene ubicarlo en la habitación más cercana al descanso que haya de ese lado, pues eso maximiza la cobertura a otros pisos y al otro lado del descanso en el mismo piso.

En otras palabras, luego de los coordinadores que puso el goloso en cada piso, a lo sumo habrá que agregar 1 o 2 por piso, que deberán estar sí o sí en las habitaciones más cercanas al descanso de cada lado. Luego en cada piso nos quedan solo 4 posibilidades para elegir: Poner 0 coordinadores adicionales, poner uno a izquierda, uno a derecha, o poner los dos. Con nuestro algoritmo de programación dinámica consideramos las 4 opciones y tomamos la mejor.

Esto se puede ver como un problema de camino mínimo: para construir nuestra solución iremos eligiendo en cada piso una de 4 opciones, cada una de las cuales nos llevará a un estado diferente, y el costo de cada movimiento viene dado por la cantidad de coordinadores que estamos utilizando según la opción que elegimos (es decir, las "aristas" de los movimientos tienen pesos 0, 1 o 2). Queremos llegar al final a un estado donde hayamos cubierto todas las habitaciones S del hotel, con costo mínimo. Se podría utilizar el algoritmo de Dijkstra, pero como el proceso no tiene ciclos (pues siempre pasamos al piso siguiente), se puede utilizar directamente el método de programación dinámica.

[1] En realidad, que su distancia al descanso sea T o más, pues si es exactamente T, como no hay coordinador posible justo en el descanso, necesariamente la cubre un coordinador que no cruza el descanso.

El estado que se debe guardar es un par (i, k) donde i indica el piso actual que vamos a procesar, y k es un entero que puede ser positivo o negativo: Un valor positivo de k significa que de los pisos anteriores, gracias a un coordinador que ya pusimos hay un tiempo remanente k desde el descanso, que puede servirnos para cubrir habitaciones en este piso y posteriores. Un valor negativo de k indica que en los pisos anteriores quedaron cosas sin cubrir, y por lo tanto para llegar a alcanzarlas estamos obligados a poner un coordinador a un tiempo máximo $-k$ del descanso, ya sea en este piso o posteriores. Un valor $k = 0$ equivale a no tener restricciones, y es la situación inicial: no podemos usar coordinadores anteriores, pero tampoco quedó nada pendiente de cubrir en pisos anteriores.

Una vez que sabemos todos los coordinadores que se ubican en este piso, podemos calcular su aporte para pisos futuros y ver si cumplen la deuda pendiente de pisos anteriores (Recordar que con el método de programación dinámica vamos a probar las 4 opciones posibles en cada piso). Si el más cercano de los coordinadores está a una cierta distancia d del descanso, como los coordinadores pueden viajar un tiempo T, este coordinador llega al descanso con un sobrante de tiempo $s = T - d$ (si $d \geq T$ podemos ignorarlo y tomar $s = 0$, pues este coordinador no puede aportar nada yendo hasta el descanso).

Si $k < 0$ y $-k \leq s$, se cubre lo que se necesitaba de pisos anteriores y ahora el sobrante es simplemente $k_{nuevo} = s$, mientras que sino, sigue siendo necesario cubrir lo anterior y el coordinador de este piso no sirve, por lo que seguiremos teniendo el mismo $k_{nuevo} = k$. Si en cambio $k \geq 0$, nos quedamos como sobrante para futuros pisos con $k_{nuevo} = \text{máx}(k, s)$.

Finalmente, al pasar al piso siguiente hay que actualizar este sobrante k_{nuevo} restando el tiempo t_e que insume un piso de escalera. Si k_{nuevo} era negativo, al restar sigue siendo negativo (la deuda de cubrir una habitación anterior no desaparece), pero si era no negativo, no puede volverse negativo sino que lo dejamos en 0 (pues si el sobrante ya no nos sirve porque estamos lejos, igualmente no se convierte en deuda).

La complejidad de este método, adicional al costo lineal del algoritmo goloso, es $O(PT_{max})$, proporcional a los estados recorridos con programación dinámica. Como el tiempo máximo posible T_{max} está acotado por $1000 + P \cdot t_e + 1000 \leq 3000$, esta complejidad es suficiente para resolver el problema eficientemente en el tiempo provisto.

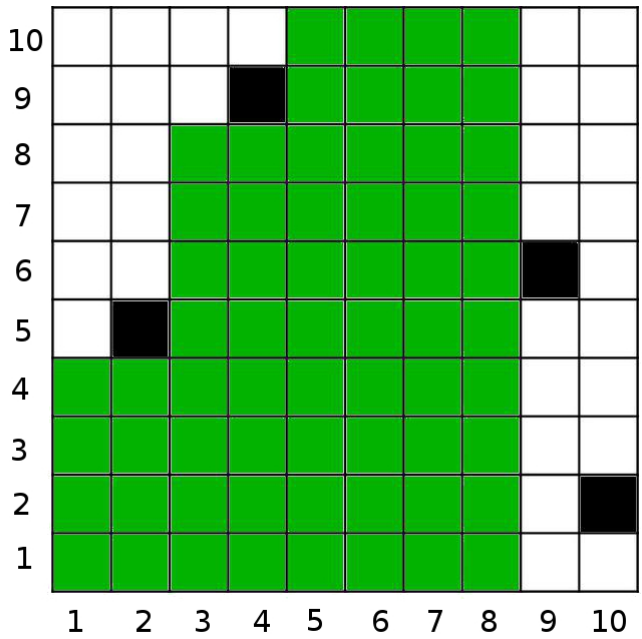
4.3. Nivel 3

4.3.1. Problema 1: El Genio de la Lámpara [genio]

http://juez.oia.unsam.edu.ar/#/task/genio/statement

El problema puede resumirse de la siguiente forma. Se tiene una grilla de $N \times M$, con A casilleros donde hay artefactos malditos. Se debe elegir *un número* $K \leq M$, y *un arreglo creciente de* K *posiciones* llamado `H` que corresponde al hechizo del genio. Este hechizo tiene en el $j - simo$ lugar a la altura que se elige para cada columna con índices desde 1 hasta K.

El hechizo abarca a todas las posiciones (i, j), con $1 \leq j \leq K$ y $1 \leq i \leq$ `H[j]`. No cualquier hechizo que cumpla estas condiciones será válido. Además, para que un hechizo sea válida se pide que entre todas las posiciones que abarca, no exista un artefacto maldito. En el siguiente ejemplo, se muestra en verde a las casillas que abarca el hechizo y en negro a los artefactos malditos, con valores de $K = 8$ y un hechizo `H` $= [4, 4, 8, 8, 10, 10, 10, 10]$.



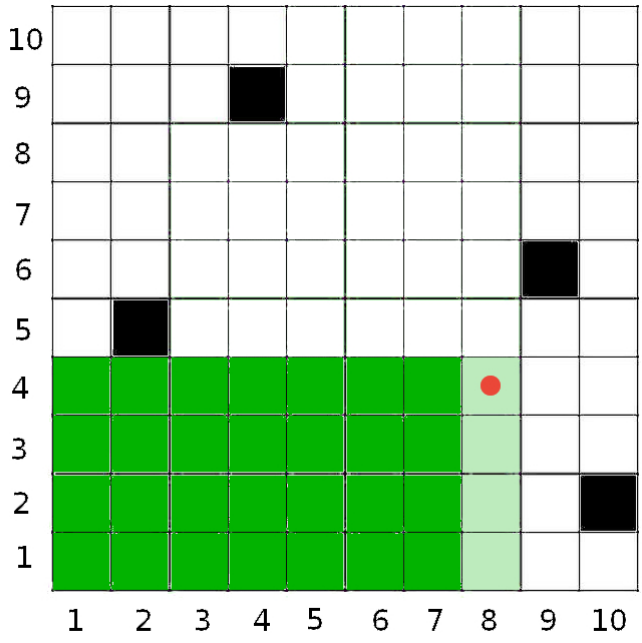
Naturalmente, el problema nos pide encontrar la *mayor cantidad de casillas que puede abarcar un hechizo válido*. Veamos una primera idea para calcular esto:

Consideremos $f(i, j)$ a la mayor cantidad de casillas que abarca un hechizo que tiene como $K = j$ y `H[j] = i` (es decir que la casilla superior derecha del hechizo se alcanza en (i, j)). Claramente $f(i, j) = -\infty$ (inválido) si en alguna casilla

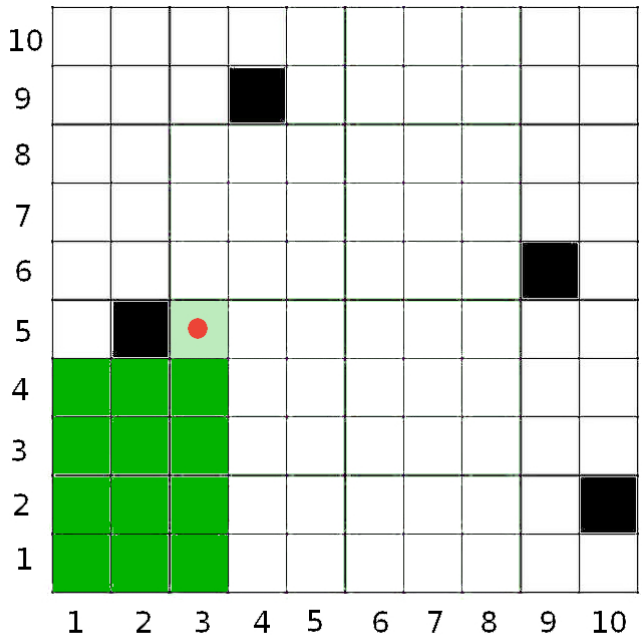
de esa columna j existe un artefacto maldito en alguna fila r con $1 \leq r \leq i$, pues cualquier hechizo con `H[j] = i` **sería inválido**. Si tenemos que pensar gráficamente a las casillas donde $f(i,j) = -\infty$, podemos partir desde cada artefacto maldito, todas las casillas que estén por encima (pero en la misma columna), tendrán como $f(i,j) = -\infty$.

Ahora debemos calcular $f(i,j)$ en las casillas restantes. Naturalmente, la solución al problema será el mayor valor de $f(i,j)$ para algún (i,j) en el tablero. Podemos calcular $f(i,j)$ a partir de los valores de $f(i,j)$ en casillas vecinas. Concretamente, tenemos dos opciones para extender a una solución existente.

Si utilizamos una solución con casilla superior derecha en la casilla inmediatamente a la izquierda de (i,j), es decir $(i,j-1)$, podemos obtener una solución agregando toda la columna j hasta la fila i, obteniendo una solución que abarca $f(i,j-1)+i$ casillas. En la siguiente figura se ejemplifica este caso con $i = 4$ y $j = 8$. En verde oscuro marcamos a las casillas abarcadas por la solución brindada por $f(i,j-1)$, y con verde claro las i que se agregan a la nueva solución.



Otra opción es extender una solución utilizando que conocemos la solución de la casilla que está inmediatamente por debajo. En este caso, simplemente se agrega una casilla a la solución, como se muestra en la siguiente figura con $i = 5$ y $j = 3$.



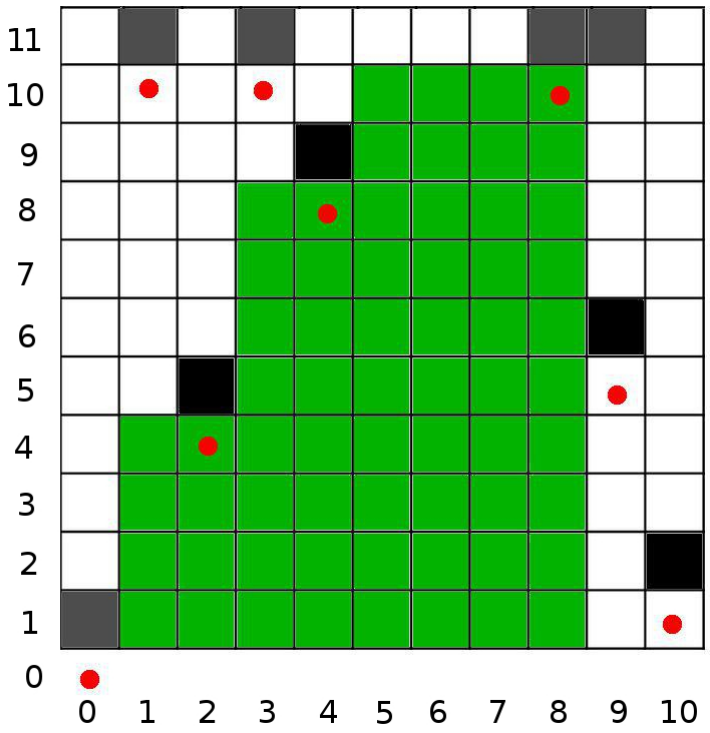
Esta recursión, que si se quiere escribir explícitamente sería de la forma $f(i,j) = \text{máx}\{f(i-1,j)+1, f(i,j-1)+i\}$, nos da una solución posible al problema con complejidad $\mathcal{O}(N \cdot M)$. Una forma de implementarla sería recorrer la grilla de forma *bottom-up* (de forma que al querer calcular $f(i,j)$ ya estén calculados los valores de $f(i,j-1)$ y $f(i-1,j)$). Veamos si podemos mejorar esto.

La siguiente idea será analizar si toda elección de (i,j) tiene sentido. Si logramos descartar algunas soluciones (porque encontramos otras que *dominan* a esta solución), no hace falta que las consideremos. En este espíritu analicemos los siguientes casos motivados por lo visto en las transiciones anteriores:

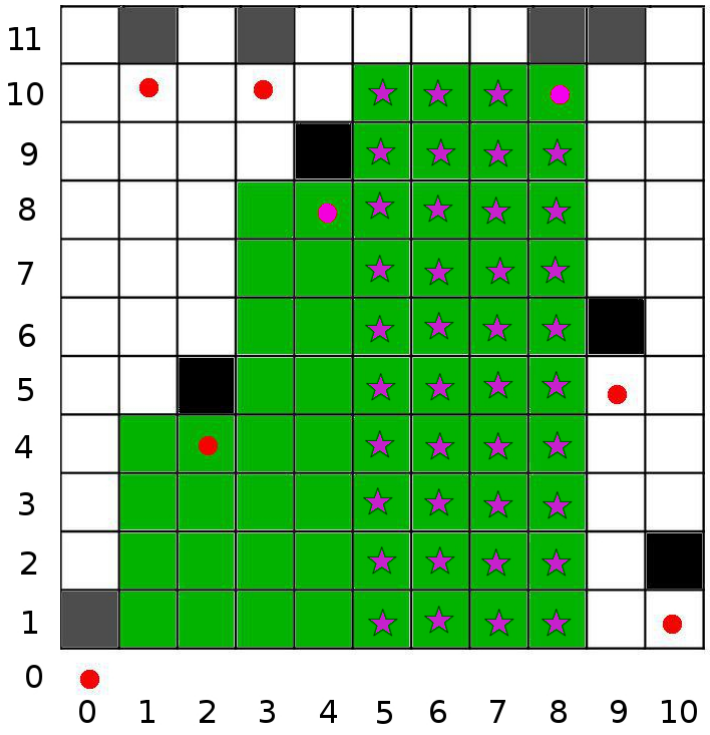
- ¿Podrá ser solución al problema un hechizo con casilla superior derecha en (i,j) si sabemos que la casilla $(i+1,j)$ está libre? No, pues podemos tomar el mismo hechizo que tiene la solución en (i,j), y aumentar la última coordenada del hechizo en una unidad, obteniendo una solución que sigue siendo válida (pues H sigue siendo creciente, y solo cambiamos una casilla), y alcanza un mayor valor.
- ¿Podrá ser solución al problema un hechizo con casilla superior derecha en (i,j) si sabemos que todas las casillas de la forma $(r,j+1)$ con $1 \leq r \leq i$ están libres? No, pues podemos extender el hechizo de manera similar, obteniendo una mejor solución.

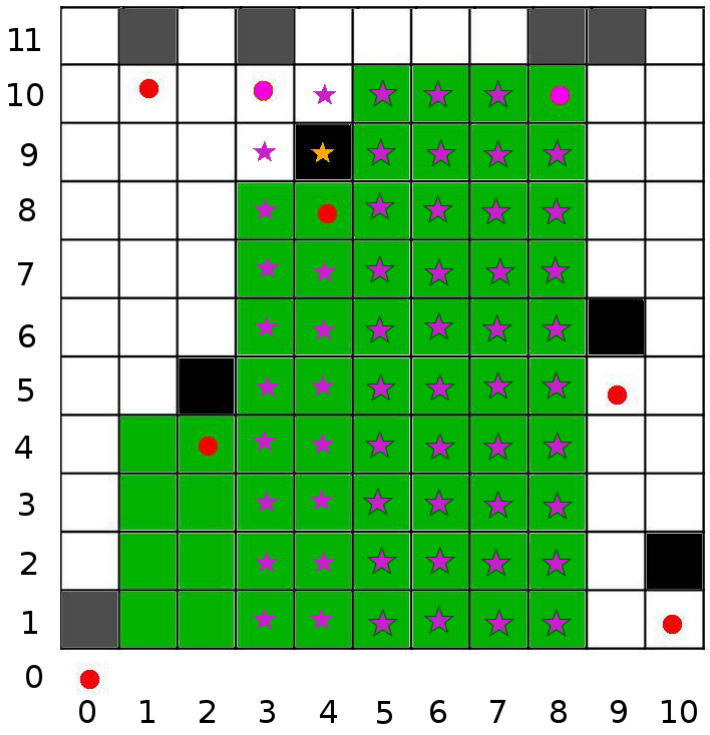
Por lo tanto, por cada artefacto maldito, tendremos a lo sumo dos soluciones

candidatas a analizar. Una, la que tiene su casilla superior derecha en la casilla que está inmediatamente por debajo (no permitiendo que se extienda en vertical su última columna), y otra en la casilla de la columna anterior a la del artefacto y en la fila más alta que tenga todas sus casillas con fila menor libres. En la figura se muestran los candidatos marcados con un círculo rojo. Además de los candidatos se agregó en la figura a una fila y una columna extra en los bordes, junto con un *candidato centinela* que puede ser de ayuda en la implementación (para evitar casos bordes).



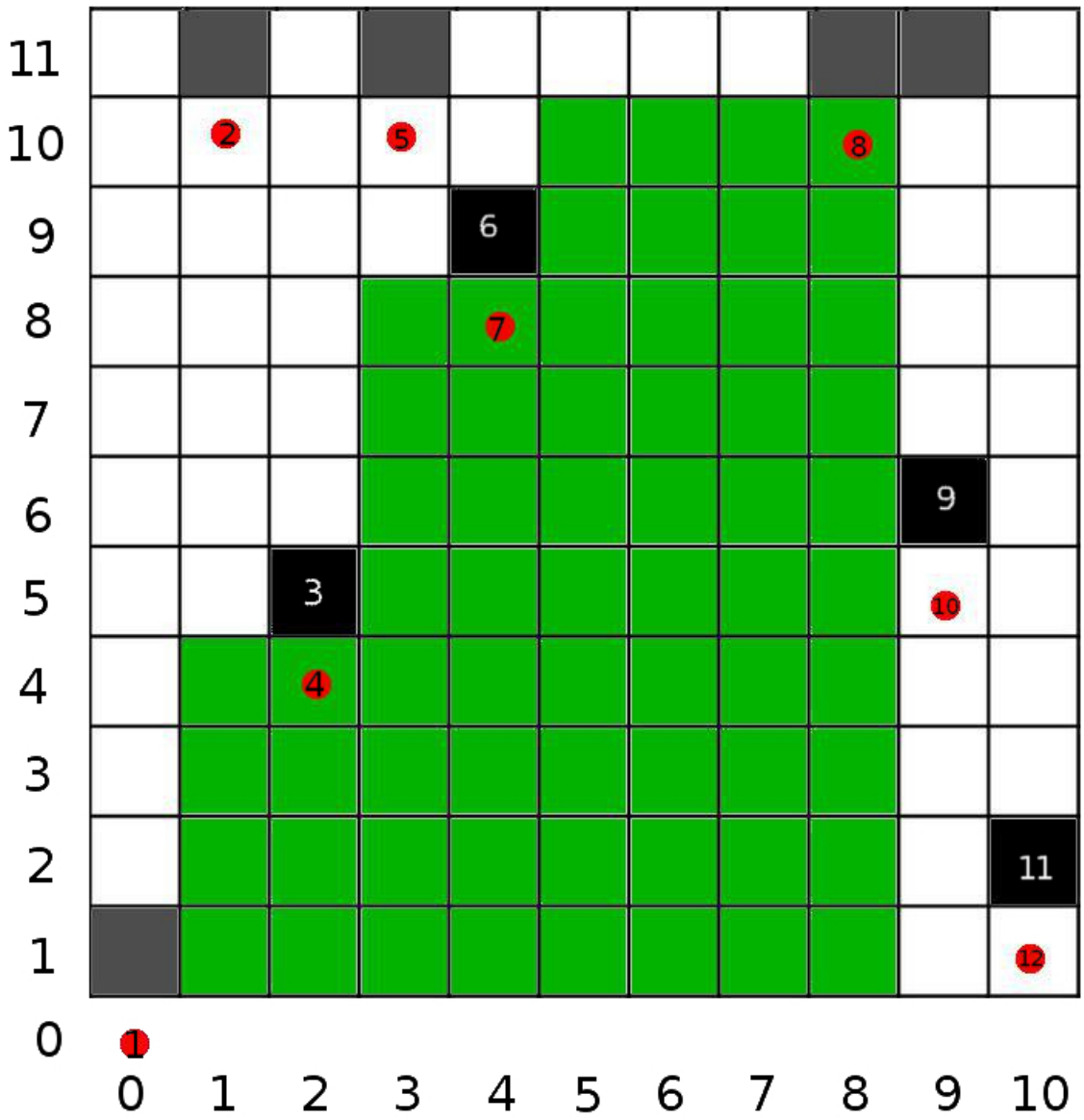
Supongamos entonces que tenemos ordenados a los candidatos (puntos rojos del ejemplo) crecientes en las columnas, y en caso de empate en las columnas, decrecientes en las filas (es decir, de izquierda a derecha y de arriba a abajo). Si estamos en el k-ésimo candidato (notar que es minúscula, no confundir con K el largo del hechizo), situado en la casilla (i_k, j_k), entonces una opción es iterar entre todos los candidatos anteriores y verificar si es factible unir ambas soluciones. Para ello, al analizar si podemos utilizar la solución de un candidato q con $q < k$, necesariamente deberá ocurrir que $i_q < i_k$ (el hechizo debe ser creciente) y $j_q < j_k$ (pensar por qué no puede haber dos candidatos en la misma columna). Además deberá estar libre de artefactos malditos el rectángulo con esquinas opuestas (i_k, j_k) y $(1, j_q + 1)$, como se muestra en las siguientes figuras (en la primera se puede, mientras que en la segunda no es válido).





Según cómo hagamos esta verificación, podremos resolver distintas subtareas. Una forma relativamente sencilla y eficiente resulta de utilizar una técnica que suele denominarse *sweep line* (barrido de línea). Cabe destacar, que este nombre es el de la técnica más general y muchas veces se dirá que un problema utiliza esta técnica ignorando las partes propiamente del problema.

Lo que vamos a hacer es aprovechar el orden que le dimos a los candidatos (cabe aclarar, el orden propuesto anteriormente no fue arbitrario) y vamos a agregar en el orden de los candidatos a los artefactos malditos. De esta forma, vamos a recorrer tanto a candidatos como a artefactos malditos en dicho orden, tal y como se muestra en la figura. Además, vamos a mantener una pila auxiliar mientras recorremos cuyo tope guarde *el candidato más a la derecha que es factible ubicar antes del candidato actual*.



Al encontrarnos con un candidato en el recorrido vamos a asignarle como candidato anterior al tope de la pila, y luego lo vamos a agregar como nuevo tope a la pila. Al encontrarnos con un artefacto maldito vamos a retirar candidatos de la pila, hasta que el tope de la pila esté en una fila menor a la del artefacto maldito.

De esta forma nos aseguramos que la región buscada entre el candidato en cuestión y el tope de la fila está libre, pues, de haber un artefacto en ese rectángulo hay dos opciones, según si está en una fila mayor a la del tope de la pila o si está en una fila menor o igual.

En el primer caso, tendríamos que deberíamos haber apilado al candidato generado por el artefacto maldito en el rectángulo, contradiciendo quién es nuestro tope de la pila actual. En el segundo caso, la presencia del artefacto hace que deberíamos haber desapilado al tope de la pila en cuestión. Esto garantiza que el rectángulo está libre hasta la columna donde está el candidato en cuestión, pero por cómo generamos los candidatos, sabemos que debajo del candidato está toda la columna libre.

Más aún, puede verse que por cómo se generan los candidatos a partir de los artefactos malditos, no puede haber dos candidatos válidos anteriores para un candidato, por lo tanto con este método en realidad estamos obteniendo al único candidato que puede venir anteriormente.

De esta forma, calculando el área de los rectángulos en cuestión entre candidatos, podemos calcular $f(i, j)$ para los candidatos (cuya cantidad es menor o igual a $2A$), lo cual concluye una solución eficiente al problema.

4.3.2. Problema 2: Creando un emporio [emporio]

http://juez.oia.unsam.edu.ar/#/task/emporio/statement

Para pensar este problema, conviene plantearlo en términos de grafos: pondremos un nodo por cada uno de los P puntos de interés, y una arista por cada una de las R rutas que los interconectan.

Lo primero que debemos entender para poder resolver este problema es el **costo**[2] que produce cada arista. La arista corresponde a una ruta, y cada ruta tiene tres atributos:

- T, la cantidad de personas que la transitan diariamente.
- D, la cantidad de dinero que cada una de esas personas gastaría.
- M, el costo de manteniemiento fijo, que no depende de la cantidad de personas.

Si en esta ruta se instala el paseo, entonces, en total el empresario tendrá un costo M por su instalación, pero a este costo hay que restarle lo que recupera como

[2]Podríamos trabajar con el beneficio y tener todos los signos al revés, es igual.

ganancia de lo que compran las personas: gastarán en total $T \cdot D$, por lo cual el costo final es $c = M - T \cdot D$. Si este costo es negativo, la ruta es un buen lugar para instalar un paseo comercial y de hecho genera ganancia neta para el empresario.

Por esta razón, **siempre conviene crear paseos comerciales en todas las rutas con** $c < 0$. Para ver que esto es cierto, podemos suponer que tuviéramos una solución en la cual en una de estas rutas no se creó paseo comercial. Al incorporar un paseo comercial en esa ruta, como $c < 0$ el costo total baja (equivale a decir que la ganancia neta es mayor), y si los P puntos de interés estaban conectados por paseos en la solución anterior, al agregar una ruta con paseo siguen conectados. Con lo cual, si en una solución no incorporamos el paseo, podemos obtener otra solución válida y mejor incorporándolo. Es por esto que todas las rutas con $c < 0$ van a tener paseo, y debemos incorporarlas.

De las que quedan, la intuición es que como tienen $c \geq 0$ generan un costo y no querríamos crear paseo allí, pero deberemos construir algunos para tener un emporio válido, es decir, para que todos los puntos de interés estén interconectados mediante paseos comerciales.

Este problema es muy similar al de árbol generador mínimo: http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/arbol-generador

En efecto, si imaginamos que cada componente conexa que se forma al mirar las aristas con $c < 0$ fuera un único nodo, y las aristas restantes con $c \geq 0$ unen estos nodos, como debemos interconectarlos con costo mínimo y para eso no vamos a querer usar ciclos (pues al no quedar aristas negativas, los ciclos nunca benefician), lo que necesitamos es un AGM de estos nodos correspondientes a las componentes.

Alternativamente, una implementación más sencilla es aprovechar que el algoritmo de Kruskal para AGM ya va formando "componentes" en su ejecución (a saber, tiene las componentes representadas por la estructura de union-find: http://wiki.oia.unsam.edu.ar/algoritmos-oia/estructuras/union-find). Lo que podemos hacer es correr el algoritmo de Kruskal, aprovechando que como ordena las aristas por c, primero recorrerá todas las aristas con $c < 0$, que son las que queremos agregar sí o sí. La única salvedad que hay que hacer entonces es que, a diferencia del algoritmo de Kruskal común, para las aristas con $c < 0$ no vamos a verificar si forman ciclo antes de agregarlas, sino que siempre las agregaremos al conjunto solución directamente.

4.3.3. Problema 3: Respondiendo pedidos de Radiotaxi [radiotaxi]

http://juez.oia.unsam.edu.ar/#/task/radiotaxi/statement

En este problema se presentan T taxis en ubicaciones (x_i, y_i) de la grilla, y lo mismo ocurre con C clientes que se encuentran ubicados también sobre la grilla.

El tiempo de viaje en este problema entre dos puntos de la grilla (calles de la ciudad) viene dado por la llamada **distancia manhattan**. Esta es la distancia que es necesario recorrer en una grilla cuadriculada para viajar entre dos intersecciones: Así, la distancia entre los puntos $(0, 1)$ y $(2, 2)$ es de 3 cuadras, a pesar de que físicamente su distancia en línea recta sería de $\sqrt{5} = 2{,}236$.

Más precisamente, si tenemos un punto (x_1, y_1) y un punto (x_2, y_2), su distancia manhattan es $|x_1 - x_2| + |y_1 - y_2|$, es decir, "la diferencia de las x más la diferencia de las y" (o dicho de otra manera, la cantidad de cuadras norte-sur que hay que recorrer, más la cantidad de cuadras este-oeste). En este problema, esta distancia es importante pues nos indica el tiempo de viaje desde la ubicación de un taxista, hasta la ubicación de un cliente que desee levantar.

Como hay suficientes taxis para levantar a todos los clientes (pues se garantiza que $C \leq T$), debemos encontrar para cada cliente qué taxi lo levantará. Si los tiempos correspondientes (calculables con la distancia manhattan, una vez elegida la asignación que nos dice qué taxi va con cada cliente), son $t_1, t_2, \cdots, t_C$, **el más grande de ellos** corresponderá al último cliente en subir a un taxi, y por lo tanto es lo que nos interesa para este problema. Es decir, queremos encontrar la asignación que **minimice** $\max(t_1, t_2, \cdots, t_C)$.

4.3.3.1. Subtarea $T \leq 10$

Para esta subtarea, es posible iterar exhaustivamente todas las asignaciones posibles, hacer la cuenta con cada una, y tomar la mejor (ya que hay a lo sumo $10! = 10 \cdot 9 \cdot 8 \cdots 2 \cdot 1 = 3628800$ asignaciones como máximo.

4.3.3.2. Subtarea $C = 1$

En este caso, simplemente podemos iterar todos los T taxis para calcular el tiempo que cada uno toma (calculando la distancia manhattan hasta el único cliente), y tomar el mínimo de todos como la respuesta. Tiempo: $O(T)$

4.3.3.3. Subtarea $C = 2$

Podemos hacer una pequeña variación sobre la idea anterior: como solamente hay dos clientes, para cada uno de ellos hacemos lo mismo que en la subtarea anterior, revisando para todos los taxis el tiempo al cliente y seleccionando al mejor.

El único caso para considerar es aquel en que el mejor taxi sea el mismo para ambos clientes, ya que en ese caso no se puede asignar a ambos. En ese caso, conviene asignarlo a alguno de los 2, y como son solo 2 clientes, probamos ambas posibilidades:

- Si lo asignamos al primer cliente, queda un problema con un solo cliente y un taxi menos, y lo resolvemos como la subtarea anterior.
- Lo mismo ocurrirá si decidíamos usar el mejor taxi para el segundo cliente.
- Probamos ambas posibilidades, y nos quedamos con la que haya dado un tiempo menor.

La complejidad de vuelta es $O(T)$

4.3.3.4. Subtarea de calle única y $C = T$

En este caso particular, todos los taxis y clientes están sobre una misma calle, por lo cual tenemos el problema en una sola dimensión. En efecto, la parte $|y_2 - y_1|$ de la distancia Manhattan será siempre 0, y la distancia será simplemente la diferencia de x.

Para este caso particular en que hay la misma cantidad de taxis que de clientes, se puede utilizar un algoritmo goloso especial que resuelve el problema: **ordenar** todos los clientes y taxis por coordenada x, y luego emparejar el primer cliente con el primer taxi, el segundo cliente con el segundo taxi, y así siguiendo.

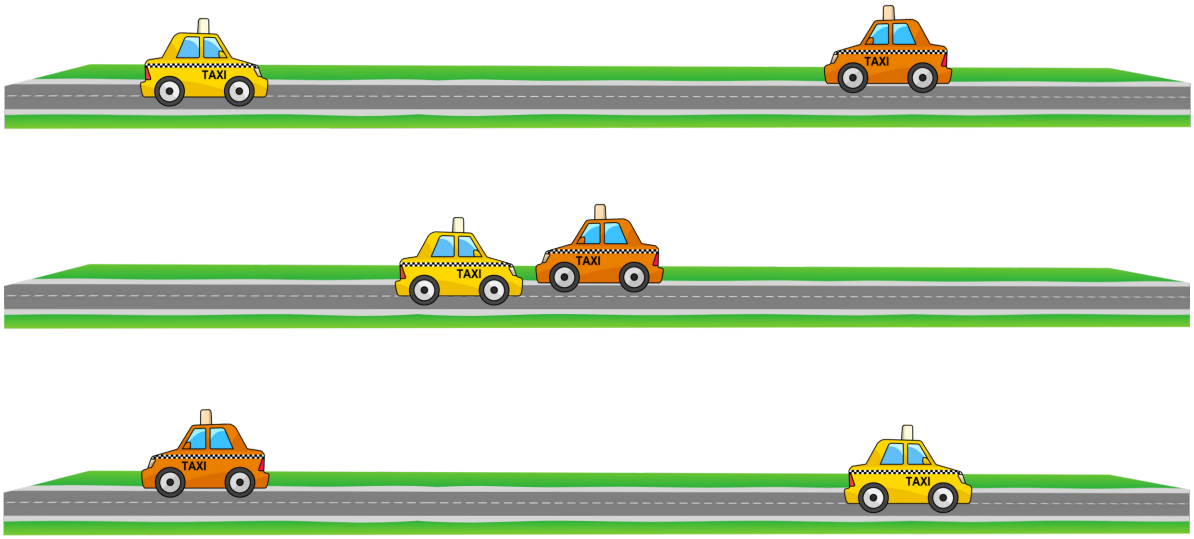
La razón por la que esto es así es que, si no los emparejáramos de esta forma, tendríamos una solución en la que hay *cruces*: dos clientes en posiciones $c_1 < c_2$ y dos taxis en posiciones $t_1 < t_2$, pero de tal forma que c_1 es levantado por t_2 y c_2 es levantado por t_1.

En este caso en que tenemos el cruce, los tiempos correspondientes a estos dos clientes son $|c_1 - t_2|$ y $|c_2 - t_1|$. Si los emparejásemos de la otra forma (es decir, en orden), de modo que c_1 fuera levantado por t_1 y c_2 por t_2, los tiempos serían $|c_1 - t_1|$ y $|c_2 - t_2|$.

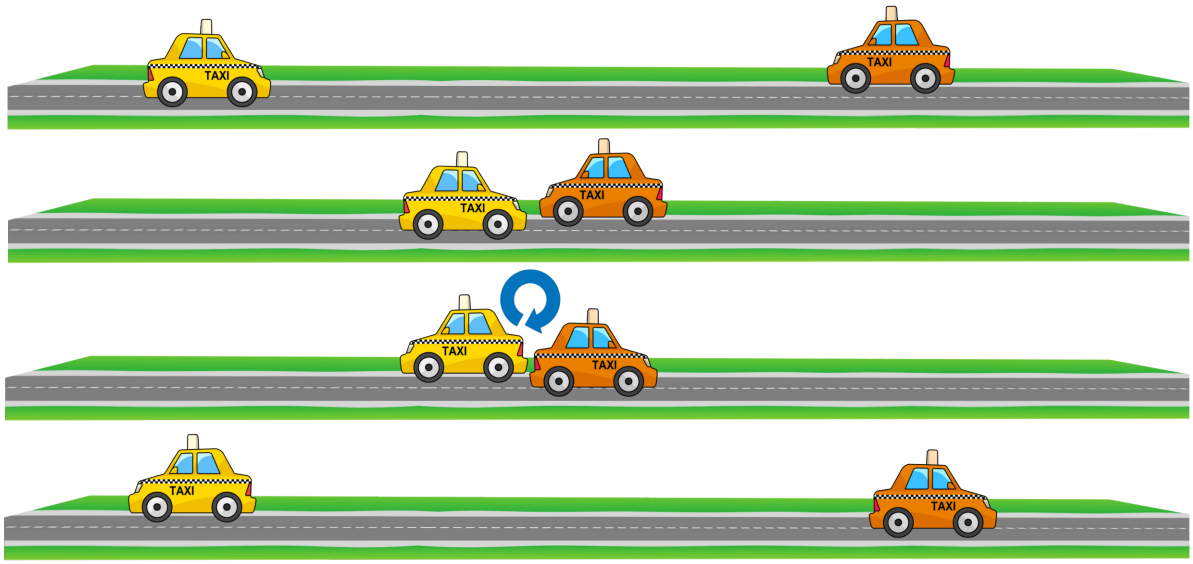
Se puede comprobar en este caso, gracias al orden, que siempre resulta $\max(|c_1 - t_1|, |c_2 - t_2|) \leq \max(|c_1 - t_2|, |c_2 - t_1|)$. Por esta razón, **dar vuelta un**

cruce no empeora la solución. Con lo cual, si empezando en un emparejamiento cualquiera, damos vuelta cruces hasta que ya no haya más, no tendremos un emparejamiento peor: es decir, cualquier emparejamiento es peor o igual al ordenado. Por esta razón, el ordenado (que es el único sin cruces) es el mejor.

Para razonar por qué dar vuelta un cruce no empeora, observemos que si tenemos una situación de cruce, e imaginamos el viaje de los taxis hasta el cliente: inicialmente, $t_1 < t_2$, pero como al final t_1 termina donde lo espera c_2, y t_2 termina donde lo espera c_1, los taxis se cruzan en algún momento del camino:



y por lo tanto en ese instante de tiempo, se encuentran en el mismo lugar. Por lo tanto, en ese momento ambos taxis son indistinguibles, y para lo que queda de trayecto, da igual a fines de tiempos que cambien de plan y t_1 prosiga hasta c_1 y t_2 hasta c_2:



Por lo tanto, es posible que t_1 busque a c_1 y t_2 busque a c_2 sin tener un peor tiempo total, como afirmamos antes.

4.3.3.5. Subtarea de calle única

Solución con programación dinámica:

De la observación que hicimos en la explicación de la subtarea anterior, podemos ver que no es conveniente que dos taxis se crucen en su camino. Es decir, una vez que decidimos **cuáles** son los C taxis que van a levantar a los clientes, la asignación de qué taxi usar para qué cliente queda ya determinada como en la sección anterior: el primer taxi irá con el primer cliente, el segundo con el segundo, y así siguiendo (cuando los ordenamos por aparición en la calle).

Esto da lugar a una solución de programación dinámica con complejidad $C(T-C)$. Planteamos $dp(t, c)$, con $t \geq c$, como el mínimo tiempo para que los primeros c clientes estén subidos a un taxi, si se los asigna de forma óptima a algunos de los primeros t taxis. La solución al problema será entonces $dp(T, C)$.

La clave el que como el taxi de más a la derecha va con el cliente de más a la derecha, hay solo dos opciones en cada paso: o bien no utilizar el taxi de más a la derecha, o bien utilizarlo, y entonces sabemos que va a buscar al cliente de más a la derecha.

Si no utilizamos el taxi de más a la derecha, la mejor solución posible es $dp(t-1, c)$. Y si lo usamos, para asignar a los $c-1$ clientes restantes lo mejor posible será $dp(t-1, c-1)$, pero hay que tener en cuenta que como el taxi t busca al cliente c, el tiempo final será $max(dp(t-1, c-1), dist(t, c))$

Entonces, la recursión queda:

$$dp(t,c) = \begin{cases} \text{mín}(dp(t-1,c), \text{máx}(dp(t-1,c-1), dist(t,c))) & \text{si } t \geq c > 0 \\ +\infty & \text{si } t < c \\ 0 & \text{si } c = 0 \end{cases}$$

El segundo caso ya que cuando $t < c$, no alcanza la cantidad de taxis para los clientes, y entonces es imposible cumplir lo pedido, así que indicamos el peor valor posible de tiempo, para que la fórmula recursiva prefiera siempre la otra opción posible cuando existe.

Solución con búsqueda binaria y algoritmo goloso:

La solución eficiente esperada para este problema se basa en utilizar búsqueda binaria en la respuesta. Es decir, iremos probando con búsqueda binaria distintos valores posibles X del tiempo máximo permitido, y en cada paso tendremos que ver si es posible asignar de modo que ningún cliente tarde más de X en subir a su taxi. Si esto es posible, en la búsqueda binaria pasamos a probar valores más chicos, y si no es posible, valores más grandes, hasta encontrar el valor extremo que da la respuesta.

Para saber si con un cierto valor X es posible, lo que nos estamos preguntando es si es posible asignar a cada cliente un taxi que no esté a más de X unidades de distancia. Es decir, si el taxi está en la posición t y el cliente en la posición c de la calle, tiene que ser $|t - c| \leq X$. O sea que por cada cliente, tenemos un **intervalo** de taxis posibles que pueden levantarlo. Se puede demostrar que en este caso, conviene usar el primer taxi posible (el que tenga menor coordenada x) para levantar al primer cliente, y así siguiendo, el menor taxi posible (entre los restantes) para levantar al segundo cliente, hasta que se hayan cubierto todos los clientes o quede un cliente sin taxis disponibles a distancia aceptable. Este método goloso determina correctamente en este caso si es posible levantar a todos los clientes sin pasarse del tiempo permitido X.

4.3.3.6. Caso general

En el caso general, podemos hacer búsqueda binaria en la respuesta, de manera exactamente idéntica a lo que planteamos en la subtarea anterior, pero ahora habrá que cambiar la verificación de factibilidad de un cierto valor solución al tener la grilla 2D en lugar de 1D.

Lo que podemos hacer es, una vez fijada la distancia manhattan máxima

permitida, calcular las distancias entre los distintos pares de clientes y taxis, y solo quedarnos con las parejas cuya distancia no supere la máxima.

Con todas estas parejas podemos formar un grafo bipartito[3], que tenga clientes por un lado y taxis por otro. Cada pareja válida corresponde a una arista entre el cliente y el taxi asociados. Como un matching asigna clientes a taxis, lo que queremos saber es si existe un matching que asigne todos los clientes, o sea un matching de tamaño C. Para esto podemos utilizar un algoritmo de matching máximo bipartito, y verificar si es C: Si el tamaño de un matching máximo bipartito es C, es posible asignar a todos los clientes. Si el tamaño es menor, no es posible.

Finalmente, para que esto tenga una complejidad final razonable y alcance el tiempo de ejecución permitido, hay que aplicar una optimización que nos permite reducir el número de taxis en todos los casos:

Como el máximo de taxis es 100.000 pero el de clientes es 100, en los casos grandes hay muchísimos más taxis que clientes. Entonces quedarán muchísimos taxis sin usar.

En efecto, cada cliente querría usar su mejor taxi (el que tenga más cerca). La única razón para que no lo use, es que ya esté asignado. En ese caso, querría usar su segundo mejor taxi. La única razón para no hacerlo sería que también esté ya asignado. En ese caso, querría usar su tercer mejor taxi. En general, cada cliente quiere utilizar el taxi más cercano a él, que no sea uno de los que utilizan los demás clientes, ya que si esto no ocurriera la solución sería mejorable (o al menos, el valor de ese cliente, sin afectar a ningún otro).

Pero como solo hay $C - 1$ otros clientes además de él, en el análisis anterior podemos ver que un cliente como mucho utilizará el taxi número C, es decir que por cada cliente basta quedarnos con sus mejores C taxis, y esos son los únicos candidatos posibles. De esta manera, con un cómputo inicial de costo $O(CT \lg(CT))$ podemos reducir la cantidad de taxis a C^2 como máximo. Más aún, como por cada cliente tenemos identificados C candidatos, estos son los únicos que debemos considerar a la hora de colocar aristas en el grafo para analizar la factibilidad de una solución. Por lo tanto, la cantidad de aristas del grafo bipartito será $O(C^2)$, y entonces el costo de encontrar un matching máximo será $O(C^3)$, que es suficientemente bueno como para poder resolver este problema.

[3]Se puede leer en la wiki sobre grafos bipartitos y matching máximo: http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/grafos-bipartitos/maximo-matching-bipartito

OIA-Juez

Juez online oficial de la Olimpíada Informática Argentina.

http://juez.oia.unsam.edu.ar/

El OIA Juez (OIAJ) es un sistema de evaluación automática en línea que permite el entrenamiento de los alumnos en programación, mediante el envío de soluciones en C, C++, Pascal o Java.

¿Por dónde comienzo si no tengo mucha experiencia en la programación competitiva?

Dentro del archivo de enunciados, es posible buscar problemas por categoría. Estas categorías generalmente se corresponden con las técnicas que podrían utilizarse para resolver los problemas, y existe también una categoría principiante, con los problemas del sitio más adecuados para los que están comenzando.

OIA-Foro

http://foro.oia.unsam.edu.ar/

OIA-Foro permite el libre intercambio de información y material, así como consultas y discusiones entre docentes, entrenadores, alumnos, organizadores, y en general, todos aquellos interesados en la comunidad de OIA.

OIAX

Introducción
OIAX es una imagen de VirtualBox de un Linux basado en Xubuntu (xubuntu.org) que contiene el software necesario para las competencias de OIA. Al ser una imagen de máquina virtual, se puede ejecutar el OIAX sobre cualquier sistema operativo con solo tener instalado VirtualBox (https://www.virtualbox.org/) o cualquier programa de máquinas virtuales compatible.

Para comenzar simplemente se debe iniciar el VDI con VirtualBox o equivalente. Al iniciar llegará a una pantalla similar a la que se ve debajo, ya ingresado como el usuario ”oia”. Esa es la misma configuración que encontrará durante la competencia.

En esta VM al igual que en la competencia, se debe utilizar el usuario **oia**, con contraseña **oia**.

Luego de ingresar, el entorno del sistema estará listo, con los entornos y editores, como se describe más adelante.

Software instalado

El ambiente gráfico es XFCE dado que la imagen está basada en Xubuntu. El ambiente XFCE es muy similar a GNOME, pero con menos accesorios, lo que lo hace más liviano.

Además de los programas del entorno usuales, se encuentran instalados los siguientes programas:

Compiladores

* ”fpc”: FreePascal Compiler (version 3.0.0)
* ”g++” (version 5.4.0)

Debuggers:
* gdb
* ddd

Editores/IDEs:

* Geany, similar a KDevelop, pero más simple.
Sobre Geany: ****Recomendamos utilizar este editor por su simplicidad y buen funcionamiento**.**
* gedit
* vim
* gvim
* CodeBlocks

Manejadores de archivos:
* Thunar, parte de XFCE

Navegadores:
* Mozilla Firefox

Descarga

Se pueden consultar instrucciones para levantar la máquina virtual en http://wiki.oia.unsam.edu.ar/curso-cpp/ambiente/oiax

Actualmente se puede descargar la imagen vía HTTP.

Versión actual:

* oiax versión 4.0: http://www.oia.unsam.edu.ar/oia-programacion/oiax/

Versión anterior (un poco más liviana: muy similar a la versión actual)

* oiax3.0-nacional2013.zip: