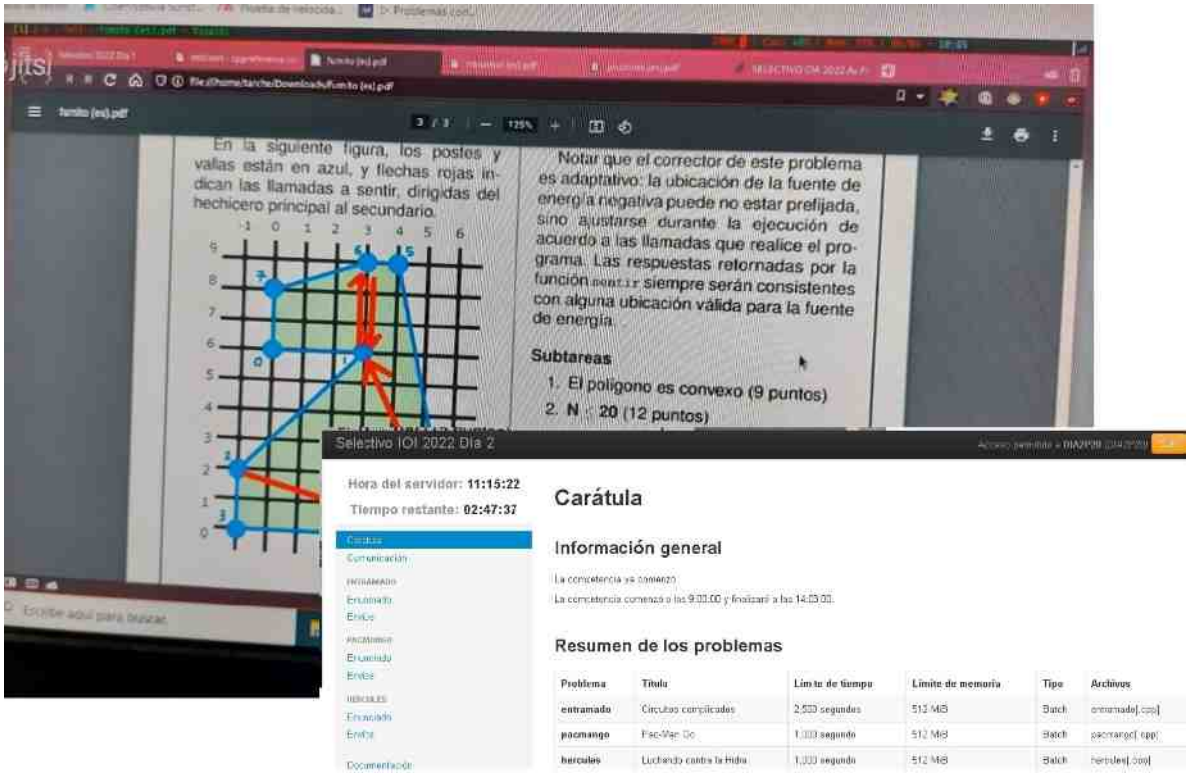


OLIMPÍADA INFORMÁTICA ARGENTINA

Cuaderno de actividades 2022

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 true o false, 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 warning de 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 shadow: Si 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 macro, que 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 $mcd(9998, 1)$, ... 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 sí 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 $\mathbb{R}$, 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 $O(n \log n)$ 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.

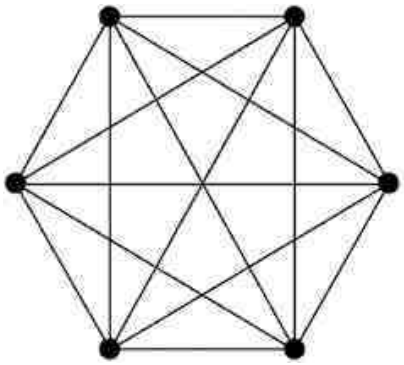
Se los usa para entender y analizar muchíisimas cosas en la vida real, y hay muchos problemas de programación que ilustran situaciones posiblemente reales para los cuales usaremos grafos.

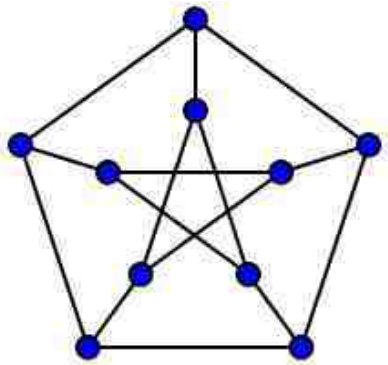
Llamaremos n a la cantidad de nodos y m a la cantidad de aristas.

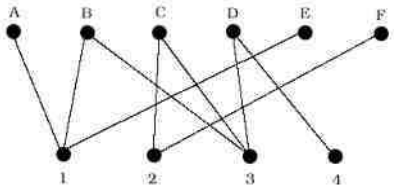
Idea, intuición, dibujitos, ejemplos

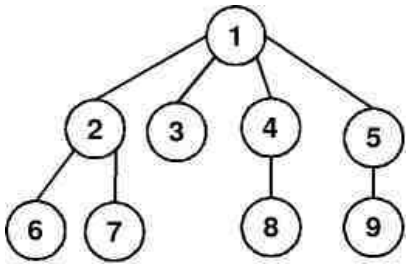
Los grafos sirven para representar relaciones entre distintas "cosas". Esas "cosas", nuestros nodos, pueden ser cualquier cosa, comidas, personas, ciudades... y hasta puede haber algunos nodos que representen un tipo de cosa, y otros nodos que representen otro tipo, por ejemplo nodos que representen 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.

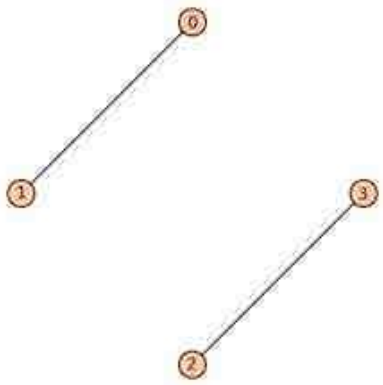
Estos son algunos ejemplos de grafos:











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:

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.

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

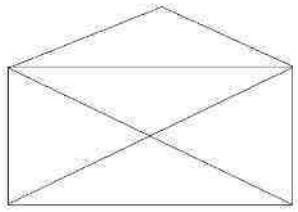
bien podríamos armar toooda la matriz de adyacencia, o toooda la lista de adyacencia de este grafo, en este caso es más fácil directamente no crear esos datos, y directamente usar "la forma del grafo" para saber qué aristas hay.

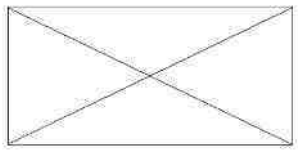
Por ejemplo, la matriz de adyacencia se usa accediendo a posiciones (i, j) para ver si hay arista entre i y j. En nuestro ejemplo, en lugar de usar una matriz, para ver si hay arista entre i y j podemos simplemente hacer la diferencia `abs(j-i)`, y ver si es 2, 3 o 7 para decidir allí mismo si hay arista. 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;
    }
}
```

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

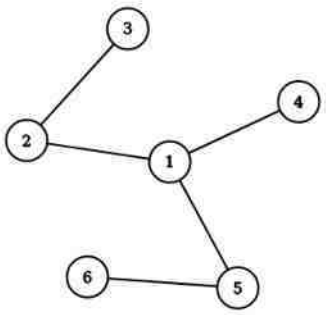
Árboles

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);
```

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: fila, tiranic, ppt, etc).

Capítulo 2

Selectivo para la IOI

2.1. Día 1

2.1.1. Problema 1: Armando la fila [fila]

http://juez.oia.unsam.edu.ar/#/task/fila/statement

En este problema, nos dan una fila de personas y la fecha de nacimiento de cada una de ellas. Se define el *nivel de enojo* de la persona en la posición *i* como la máxima diferencia de posiciones que tiene con una persona delante de él que además es más joven. En el ejemplo que viene en el enunciado, los niveles de enojo serían:

i	1	2	3	4	5	6	7	8
← CAJA	18/7/71	8/4/57	28/8/82	17/11/66	7/9/75	16/12/94	9/5/88	17/9/83
enojo[i]	0	1	0	3	2	0	1	2

Luego, el máximo nivel de enojo es 3 (celda en el índice 4. Lo enoja la persona más cercana a la caja con índice 1 que está a 3 posiciones de distancia). Para aquellas personas enojadas, debemos retornar *sus índices*, ordenados *en orden decreciente de enojo* (mayor a menor), y en caso de empate *por menor índice* (cercanía a la caja). Lo cual nos da: 4 5 8 2 7

Ahora que entendimos lo que nos pide el problema. Veamos cómo resolverlo de manera eficiente. ¿Por qué de manera eficiente? Porque por enunciado, la cantidad de personas que hay en la fila podría ser un número cercano a 300.000, así que una implementación directa de lo que pide el enunciado tardaría demasiado. Pues si para cada persona, recorremos la fila de izquierda a derecha hasta la primera persona que enoja a dicha persona (si es que existe alguna), estaríamos haciendo en total una

cantidad de operaciones de orden cuadrático en la cantidad de personas que vienen en la fila (lo cual es demasiado para casos con 300.000 personas).

Una forma de resolver el problema de manera eficiente es la siguiente. Al igual que antes, **para cada *i* buscaremos al más cercano en la caja que es más joven que la persona en *i***. Pero...¿Cómo podemos acelerar esta búsqueda? La idea será utilizar ***búsqueda binaria***.

Llamemos F al arreglo con las *fechas de nacimiento* ordenados por proximidad a la caja que viene en la entrada. Además vamos a definir a un arreglo auxiliar M tal que M[i] guarda *la fecha de nacimiento de la persona más joven dentro de las primeras i personas*. En nuestro ejemplo:

i	1	2	3	4	5	6	7	8
F[i]	18/7/71	8/4/57	28/8/82	17/11/66	7/9/75	16/12/94	9/5/88	17/9/83
M[i]	18/7/71	18/7/71	28/8/82	28/8/82	28/8/82	16/12/94	16/12/94	16/12/94

Para calcular el arreglo M,podemos usar que M[0] = F[0], y que para $i > 0$ vale que M[i] = max(M[i-1],F[i]) (notar que si una fecha de nacimiento es mayor, la persona es más joven).

Finalmente si nos centramos en la *i*-ésima persona, podemos encontrar a la persona más próxima a la caja que es más joven realizando una búsqueda binaria de la siguiente manera (vamos a suponer en una primera instancia que existe alguna persona más joven dentro de las anteriores, luego vamos a ver que en realidad no hace falta)

- Tomamos a = 0 y b = i como límites en la búsqueda binaria. (Indexamos desde 1.Como invariante a lo largo de la búsqueda tenemos que: en [1..a] son todos más viejos que *i*, y en [1..b] hay alguno más joven que *i*).
- En cada paso de la búsqueda binaria: (mientras b-a > 1)
  - Tomamos j = $b \frac{a+b}{2} c$
  - Si F[i] < M[j] (hay alguien más joven que *i* en las primeras j personas), entonces b = j. De no ser así, a = j.
- enojo[i] = (i-b)

Notemos que al finalizar la búsqueda binaria, b = a+1, y por cómo planteamos el problema sabemos que en [1..a] son todos más viejos que *i*, y en [1.. a+1] hay alguno más joven que *i*. Por lo tanto, **la persona con índice** b **es la persona más próxima a la caja que es más joven que** *i* .

{z} b

Además, **si no existe una persona másjoven que** *i* en las personas anteriores, entonces el algoritmo finalizará con b = i y a = b-1, por lo tanto enojo[i] = i-b = 0, y la persona no resulta enojada, que es lo que queremos.

Como la **búsqueda binaria reduce el campo de búsqueda a la mitad en cada paso** ,al cabo de $O(\lg(n))$ pasos finalizará la búsqueda para cada *i*. (*n* denota la cantidad de personas). Por lo tanto, como hay *n* personas,esta solución tiene una complejidad de $O(n \lg(n))$

Algo que podemos notar es que para cada *i*, la búsqueda binaria no hace falta hacerla sobre todas las personas de la fila, sino **solamente aquellas que están pintadas de gris en la tabla anterior** (pues las que no están pintadas están *dominadas* por la más próxima a su izquierda pintada de gris, como explicaremos más adelante).

Veamos ahora otra forma de resolver el problema. La idea principal en la que va a rondar esta solución es ***invertir la pregunta***. En vez de encontrar para cada persona quién lo enoja (si es que alguien lo enoja), lo que vamos a hacer es *para cada persona ver a quiénes enoja*.

Cada persona enoja a todos los que sean más viejos que él y estén a su derecha . Esto es clave, ¿por qué?, porque aparece la idea de que una **persona sea dominada por otra** en el siguiente sentido: Si una persona tiene a otra persona más joven a su izquierda, entonces la primera persona no enoja a nadie (o si enoja a alguien, podemos afirmar que existe otra que lo enoja a ese alguien con nivel de enojo más alto, que es lo que en realidad nos interesa).

¿Cómo podemos usar esta idea para resolver el problema (eficientemente)?

Vamos a tener *dos copias de la fila*, **una con las *fechas de nacimiento* tal cual como vienen, *ordenadas por el número que llevan en la fila*** (el de más a la izquierda es el próximo en ser atendido), que llamaremos F al igual que antes. En **la otra nos guardaremos *los índices* de las personas en la fila, *ordenados por fecha de nacimiento* en orden decreciente** (por ende la primera persona es la más joven y la última la más vieja) que llamaremos E. En el ejemplo de la entrada tenemos:

i	1	2	3	4	5	6	7	8
F[i]	18/7/71	8/4/57	28/8/82	17/11/66	7/9/75	16/12/94	9/5/88	17/9/83
E[i]	6	7	8	3	5	1	4	2

En E nos guardamos qué posición ocupa cada persona en la fila original.Por ejemplo: E[2] = 7, porque la segunda persona más joven ocupa la séptima posición en la fila.

Ahora vamos a responder para cada persona a quiénes enoja. Para ello **vamos a tener dos punteros/índices** , uno lo llamamos *i*, que comienza en *i* = 1, y va a iterar sobre F (la que está ordenada por *proximidad a la caja*). El otro lo llamamos *j*, que comienza al final de E en *j* = *n*, y va a ir retrocediendo.

Para familiarizarnos, pensemos lo siguiente.La persona que está en *i* = 1 en la fila, ¿a quiénes enoja?... Como vimos, enoja a *todas las personas más viejas que están a su derecha*,como es la primera, en particular enojará a todas las que sean más viejas que él.

Para simular esto podemos **achicar** *j* **mientras la persona en** E[*j*] **sea más vieja que la que está en** *i***, y notar que todas ellas estarán enojadas** (todas estarán enojadas porque estamos hablando de la primera persona en la fila, sino estarán enojadas solo aquellas que estén a su derecha, como ya vimos).

Para saber si la persona en E[*j*] es más vieja que la persona en *i*, simplemente debemos saber si nació antes, o sea, si vale que: F [E[*j*]] < F[*i*]. Finalmente, de ser así, el nivel de enojo de cada una de las personas más viejas al ir achicando *j*, corresponde a E[*j*] − *i* (la diferencia de posiciones en la fila), con el cuidado de tenerlo solo en cuenta si es que este número es positivo (si el número es negativo, significa que E[*j*] < *i*, o sea que a pesar de que la persona en E[*j*] es más vieja que la persona en *i*, pero está situada a su izquierda, y por lo tanto la persona en *i* no la enoja).

Una vez que hicimos eso, la persona en la posición *i* no va a enojar a nadie más, entonces avanzamos *i* y repetimos la disminución del *j*. Notemos que si este nuevo *i* es más viejo que alguno anterior, entonces no va a enojar a nadie (por la idea de personas dominadas que mencionamos antes).

En cada paso, **o bien aumenta** *i* **o disminuye** *j***, lo cual puede pasar a lo sumo** *n* **veces cada una** por lo tanto, nos queda que esta solución tiene una complejidad de $\underbrace{O(n \lg n)}_{\text{Ordenar}} + \underbrace{O(n)}_{\text{'Two Pointer}} = O(n \lg n)$

2.1.2. Problema 2: Antártida [antartida]

http://juez.oia.unsam.edu.ar/#/task/antartida/statement

El problema nos da un conjunto de intervalos sobre un círculo y nos pregunta cual es la longitud máxima que podemos cubrir con algunos de los intervalos si no podemos elegir dos intervalos que se superpongan.

Para simplificar la resolución, comenzaremos explicando un problema más fácil. Supongamos que en lugar de tener los intervalos sobre un círculo, los tenemos sobre una recta y nos piden los mismo, la mayor longitud de recta que podemos cubrir sin elegir dos intervalos que se superpongan.

Lo primero que haremos será ordenar los intervalos por extremo derecho. De esta manera nos quedan los intervalos (a_1, b_1), (a_2, b_2), $\ldots$, (a_n, b_n) con los b_i ordenados de menor a mayor.

La idea será usar programación dinámica, vamos a calcular para cada i, cual es la mayor longitud que podemos cubrir sin cubrir nada a la derecha del punto b_i. Esto es lo mismo que decir usando solo algunos de los primeros i intervalos. Llamemos $maxhasta_i$ a estos valores.

Para calcular $maxhasta_i$, lo que hacemos es elegir qué hacemos con el i-ésimo intervalo, es decir, decidir si lo usamos en nuestro cubrimiento. Recordemos que queremos elmáximo cubrimiento usando algunos de los primeros i intervalos. Tenemos dos casos:

- Si decidimos no usar al i-ésimo intervalo en el cubrimiento, entonces nuestro cubrimiento usa algunos de los primeros $i - 1$ intervalos. Pero esto es por definición $maxhasta_{i-1}$.
- Si decidimos usar al i-ésimo intervalo, el resto de los intervalos que usemos pueden terminar a lo sumo en a_i, para no solaparse con el i-ésimo. Si miramos cuáles son los intervalos que terminan a lo sumo en a_i, como los intervalos están ordenados por punto de finalización, son los primeros j intervalos donde j es el índice más grande tal que $b_j < a_i$. Luego, el mejor cubrimiento lo obtenemos usando lo mejor que podemos hasta el j-ésimo intervalo y agregando lo que ganamos por usar el i-ésimo. En total cubrimos $maxhasta_j + (b_i - a_i)$. Notar que $b_i - a_i$ es la longitud del i-ésimo intervalo. Si no hay ningún intervalo que termine antes de a_i, no podemos elegir ningún otro sin que se solape con el i-ésimo, por lo que lo único que podemos hacer cubre $b_i - a_i$.

Con esto podemos concluir que el mejor cubrimiento hasta el i-ésimo es lo mejor de

estas dos alternativas. Concluímos que podemos obtener *maxhasta* por medio de la siguiente fórmula:

$$maxhasta_i = m\acute{a}x(maxhasta_{i-1}, maxhasta_j + (b_i - a_i))$$

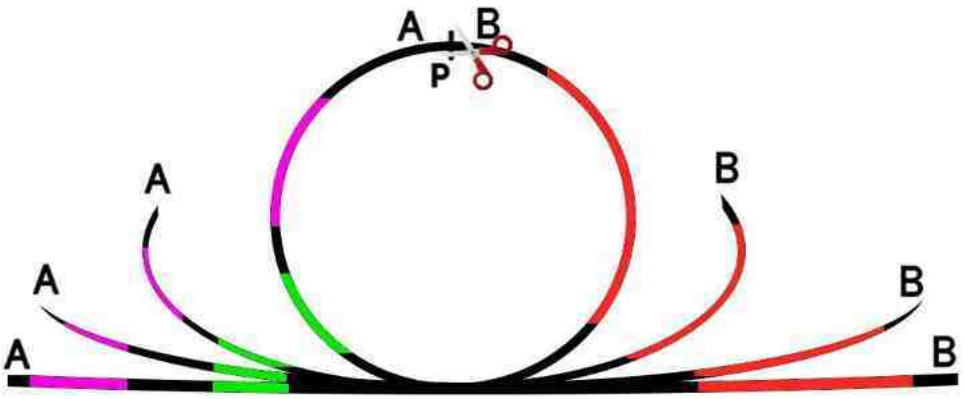
Donde j es el mayor índice tal que $b_j < a_i$. (Para solucionar el problema cuando no hay ninguno podemos definir $b_0 = -\acute{\imath}nf$ y $maxhasta_0 = 0$)

Con esto podemos calcular todos los $maxhasta_i$ en orden creciente y la solución a nuestro problema es $maxhasta_n$. Para encontrar el j correspondiente a cada i, lo más intuitivo es para cada i empezar con $j = 0$ e ir incrementándolo hasta que deje de valer $b_{j+1} < a_i$. El problema de esto es que calcular el j correspondiente a un i tardaría $O(n)$ y si lo hacemos para todos los i el algoritmo queda $O(n^2)$ que no alcanza para obtener el puntaje máximo. Para solucionar esto podemos notar que se puede hacer una búsqueda binaria para encontrar este j en $O(\lg n)$ por lo que hacerlo para todos los i queda $O(n \lg n)$ que es suficiente para obtener el máximo puntaje.

Presentamos un pseudocódigo final del problema para el caso de los intervalos sobre una recta:

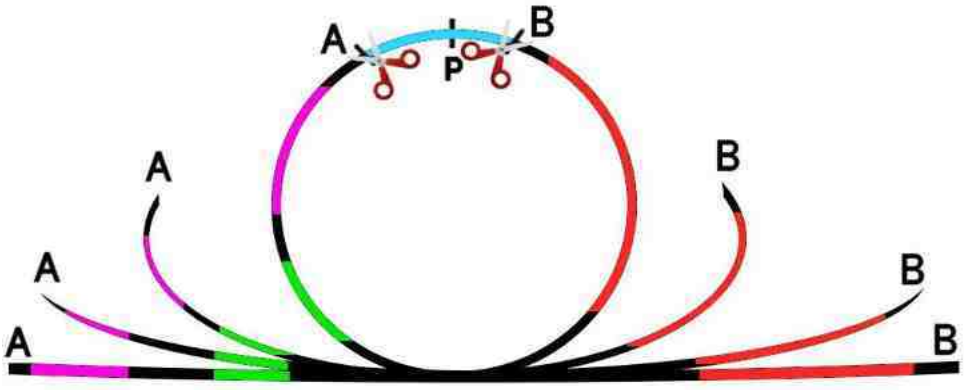
```
 1: procedure MaximoCubrimientoIntervalos(intervalos)
 2:     OrdenarPorFin(intervalos)
 3:     maxhasta_0 ← 0
 4:     N ← size(intervalos)
 5:     for i ← 1 . . . N do
 6:         j ← calcularJ (intervalos, i)
 7:         maxhasta_i = max(maxhasta_{i−1}, maxhasta_j + longitud(intervalos[i]))
 8:     return maxhasta_N
 9: procedure CalcularJ(intervalos, i)
10:     top ← i, bot ← 0, mid ← (top + bot)/2
11:     while top − bot > 1 do
12:         if intervalos[i].begin >= intervalos[mid].end then
13:             bot = mid
14:         else
15:             top = mid
16:     return bot
```

Sin embargo, el problema original era sobre un círculo y no sobre una recta. La idea para resolverlo será reducirlo al caso de la recta. Para esto notemos que podemos elegir un punto sobre el círculo, cortar el círculo por ese punto, estirar el circulo cortado para que quede una línea y usar el algoritmo que vimos más arriba.



El problema surge al elegir por dónde cortar el círculo. El enunciado del problema asegura que hay un punto *P* del cículo que está cubierto por a lo sumo *k* intervalos. Si encontramos ese punto, existirán *k* + 1 posibilidades:

- O bien *P* está cubierto por uno de esos *k* intervalos en el cubrimiento óptimo. En cuyo caso, el resto de los intervalos que usemos caen en el pedazo de círculo de afuera del intervalo que cubre a *P* , y "estirando" ese pedazo podemos pensar que nos queda una recta. La solución entonces resulta ser la suma entre la solución al problema restante en la recta (que no tiene los intervalos que pasen por la región cortada), y la longitud del intervalo que contiene a *P* que estamos considerando.



- O bien *P* no está cubierto en el cubrimiento óptimo. En este caso podemos cortar al círculo por *P* y resolver el problema en el caso recta directamente, ya que al cortar por *P* , no estaremos cortando ningún intervalo de la solución óptima.

Lo que podemos hacer es probar estas *k* + 1 posibilidades,y quedarnos con la que resulte en la mejor solución. La complejidad sería $O(kn \lg n)$ y como *k* es chico, es suficiente.

Nos queda ver cómo encontramos P . Para eso podemos empezar en un punto Q cualquiera del círculo (el del meridiano de 0 grados de longitud, por ejemplo) y contar en $O(n)$ cuantos intervalos cubren a Q. A partir de ahí vamos moviendo el Q por el círculo (en sentido horario, por ejemplo) hasta que esté cubierto por a lo sumo k intervalos. Cada vez que Q pasa por un punto de fin de un intervalo, restamos uno a la cantidad de intervalos que cubren a Q y cada vez que pasamos por un punto de inicio sumamos uno a esta cantidad. Antes de terminar de dar una vuelta entera al círculo deberíamos encontrar el P que nos sirve. Todo este procedimiento se puede implementar en $O(n \lg n)$ si ordenamos los puntos de inicio y fin de los intervalos según su grado y así obtenemos en $O(1)$ cuál es el próximo punto de inicio o fin por el que pasará Q.

2.1.3. Problema 3: Encontrando el Tiranic [tiranic]

http://juez.oia.unsam.edu.ar/#/task/tiranic/statement

En este problema, sabemos que existe una grilla de $m \times n$, dentro de la cual está escondido elbarco que se busca.Este barco es un rectángulo de $a \times b$ casillas ($1 \leq a \leq m$, $1 \leq b \leq n$), ubicado en algún lugar de la grilla. No conocemos ni la ubicación del barco en la grilla, ni los valores de a y b, sino solo las dimensiones de la grilla completa, m y n.

Una primera observación clave es que lo importante es lograr hacer una medición con radar que encuentre el barco (es decir, que **no devuelva** -1). Una vez hecha una medición así, con solamente 3 mediciones adicionales podemos identificar completamente la ubicación exacta y tamaños del barco.

Más precisamente, supongamos como ejemplo que hacemos una medición hacia la derecha que encuentra al barco a 3 casillas de distancia del radar. Esto ya nos indica dónde está “el borde izquierdo” del rectángulo que corresponde al barco. Preguntando por esa misma fila pero hacia la izquierda, obtendremos el borde derecho.

Para obtener los otros dos bordes, podemos similarmente usar el radar hacia arriba y hacia abajo, en la columna número 4. Sabemos que esa columna se interseca con el rectángulo del barco, porque justamente la primera medición hacia la derecha que indicó distancia 3 nos garantiza que en esa fila en la columna 4 hay una casilla del barco. En general si en la primera medición el radar encuentra el barco a distancia d, podemos preguntar por ambas direcciones de la columna $d + 1$ para obtener los bordes superior e inferior del rectángulo.

Dicho todo lo anterior, la parte más difícil del problema pasa a ser cómo

asegurarse de hacer una medición que encuentre al barco con pocos usos del radar (ya que luego hemos visto que se hacen 3 usos más y con eso se termina de identificar todo).

El problema incluye en su enunciado la siguiente garantía importantísima:

Por las descripciones históricas disponibles, se sabe que el área del Tiranic (medida en casillas) es ***mayor o igual que 100****.*

Es decir, sabemos que $ab \geq 100$. La clave del problema consiste en aprovechar este conocimiento sobre a y b para reducir el número de preguntas que se realizan en peor caso.

A continuación se describen varias soluciones posibles,en orden creciente de dificultad y puntaje que podrían obtener:

- Como por las cotas del enunciado $nm \leq 10^6$, se tendrá $\min(n, m) \leq 10^3$ (porque si n y m fueran los dos más que 1000, su producto ya se pasaría de 10^6).

  Preguntando por completo ordenadamente desde el borde más chico de los dos (por ejemplo si el mínimo fuera n, preguntaríamos hacia abajo por las columnas 1, 2, 3, 4, ...), en a lo sumo 1000 preguntas habremos descubierto una esquina del barco. En este caso como hemos encontrado de hecho una esquina, ya tenemos 2 bordes identificados, y bastan dos preguntas más en la fila y columna de esta esquina para determinar completamente las dimensiones del rectángulo. Este método utiliza a lo sumo $\min(n, m) + 2$ preguntas, y con esto se obtienen 5 puntos.

  Probablemente esta sea la solución más simple posible que obtiene puntaje positivo, y se observa que no utiliza nunca el conocimiento que tenemos de que $ab \geq 100$.

- Podemos obtener mejores soluciones usando $ab \geq 100$. Una muy buena observación es que necesariamente tiene que ser $\max(a, b) \geq 10$, es decir que existe una dimensión en la que el barco mide al menos 10 (Esto es así porque si las dos fueran 9 o menos, el área sería como máximo 81 y no llega a 100).

  Con eso en mente, podemos preguntar **cada 10 filas y columnas**, en lugar de preguntar por todas las filas/columnas como en el caso anterior. Por ejemplo, preguntaríamos en las filas $10, 20, 30, \cdots$ e igualmente con las columnas, preguntando en todas las numeradas con un múltiplo de 10.

  Alguna de todas esas mediciones debe necesariamente encontrar el barco, pues de lo contrario tendría ambas medidas menores a 10 (en la “grilla” que

forman esas mediciones,quedan “espacios blancos”de 9×9, donde el barco no entra). Una vez que tocamos el barco, ya vimos que 3 preguntas más son suficientes para terminar de determinarlo. De esta forma se hacen a lo sumo $\frac{n}{10} + \frac{m}{10} + 3$ mediciones, lo cual es en peor caso 100003, para tableros con dimensiones muy desiguales.Pero si combinamos esta estrategia con la estrategia sencilla anterior (eligiendo la que menos preguntas vaya a utilizar según nuestro análisis), 335 preguntas son siempre suficientes. Con esto se obtienen 25 puntos.

- La estrategia anterior puede ser refinada: como vimos, al preguntar cada 10 filas / columnas, el mapa queda separado en espacios blancos de 9×9 donde el barco no cabe, pues $9 \cdot 9 = 81 < 100$.

  Ahora bien, no es necesario que estos espacios sean cuadrados de 9×9, sino que lo único importante es que su área sea justamente menor que 100, para que el barco no quepa en ellos. Dejando espacios de 9×11 también nos aseguraremos tocar al barco en algún momento, porque los espacios quedan de área $99 < 100$. Esto corresponde a seguir preguntando cada 10 en una dimensión, pero a preguntar **cada 12** en la otra, obteniendo un valor de $\frac{n}{10} + \frac{m}{12} + 3$ mediciones (asumiendo $n \leq m$, y sino los intercambiamos).

  Con esta mejora, 306 mediciones siempre bastan, y esto alcanza para obtener 50 puntos.

- Por último, si somos todavía más flexibles con los tamaños de esta “grilla de barridos de radar”, la estrategia anterior puede generalizarse a cualesquiera valores p, q tales que $p \cdot q < 100$ (La estrategia de 25 puntos era el caso $p = q = 9$, y el caso anterior es $p = 9$, $q = 11$). Para tales p y q, preguntando cada $\left\lfloor \frac{p}{2} \right\rfloor + 1$ en una dimensión y cada $q + 1$ en la otra, tendremos una solución con $\frac{n}{p+1} + \left\lfloor \frac{m}{q+1} \right\rfloor + 3$ mediciones.

  El valor óptimo de p y q dependerá de los valores exactos de m y n, pero como $1 \leq p, q \leq 100$, podemos sencillamente evaluar todas las posibilidades de p y q con dos *for*, y tomar aquella combinación para la cual la cuenta anterior sea mínima. Luego estos p y q determinan la estrategia, que consiste en preguntar cada $p + 1$ filas y cada $q + 1$ columnas (o viceversa).

  Si bien para tableros cuadrados se puede comprobar que la cantidad de operaciones es igual que en la solución anterior (porque el óptimo resulta ser $p = 9$, $q = 11$ en esos casos), esta generalización a valores de p y q bien elegidos mejora mucho la estrategia en tableros rectangulares con dimensiones muy desiguales.

Como en el peor de los casos esta estrategia realiza 186 preguntas (que corresponde al caso $m = n$), ya es suficiente para garantizar los 100 puntos.

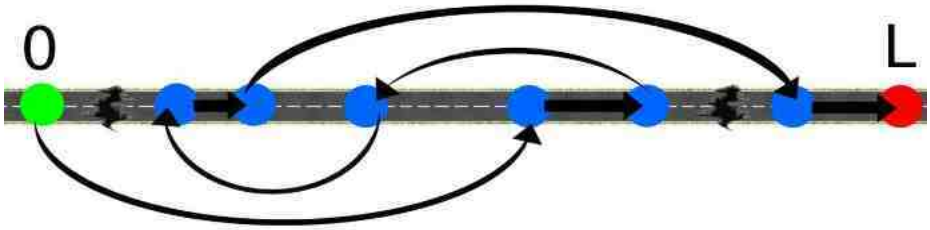
2.2. Día 2

2.2.1. Problema 1: Un largo camino a casa. . . [desvios]

http://juez.oia.unsam.edu.ar/#/task/desvios/statement

Primero que nada, representaremos elproblema con grafos y luego veremos cómo resolverlo. Los nodos serán los puntos de inicio y fin de un camino vecinal, sumado al punto de inicio y de fin del camino (el *km* 0 y el *km L*).

Podemos notar que un camino vecinal puede comenzar en el principio de la carretera y un camino puede terminar en el punto donde empieza otro. Más aún, puede ser que sea conveniente tomar un camino vecinalque retroceda para evitar un obstáculo. Veamos un ejemplo:



Como en algunos casos puede ser conveniente usar el camino vecinal para retroceder, deberemos representar el problema como un grafo dirigido para permitir utilizarlos en ambos sentidos. La carretera principal solo puede recorrerse en un sentido.

Además, ya podemos calcular el costo de recorrer cada segmento delcamino con las siguientes observaciones:

- Un segmento de longitud x de la carretera principal se recorre en tiempo x, pues el enunciado dice que la unidad de tiempo corresponde al tiempo que se demora en recorrer 1*km* de carretera. Además, hay que restarle eltiempo perdido por cada obstáculo en ese segmento del camino.
- El tiempo que demora cada camino vecinal es recibido en la entrada y no precisa cálculos adicionales.

Dicho todo esto, lo que queremos es encontrar el camino de costo mínimo que nos lleve desde el punto de inicio de la carretera al punto de fin. Para ello,

existen varios algoritmos de caminos mínimos (ver wiki) y según cuál elijamos será la velocidad de nuestra solución.

Para hacer una correcta elección de algoritmo siempre tenemos que analizar cuántos nodos y cuántas aristas tiene nuestro grafo. Como los nodos son los extremos de los caminos vecinales más el nodo que representa al *km* 0 y al *km* L, tendremos a lo sumo $2D+2$ nodos (ya discutimos que algunos de estos nodos pueden corresponder al mismo punto del camino, y por esto decimos *a lo sumo* $2D + 2$ y no *exactamente*).

$$V \leq 2D + 2 \qquad \text{donde V es la cantidad de nodos}$$

Además, tenemos $2D$ aristas que son los caminos vecinales en ambos sentidos y $V - 1$ aristas que corresponden a los segmentos de carretera. Así,

$$E = 2D + V - 1 \leq 2D + 2D + 2 - 1 = 4D + 1 \qquad \text{donde E es la cantidad de aristas}$$

Como hay pocas aristas proporcionalmente a la cantidad de nodos que tenemos, un algoritmo muy conveniente es el de Dijkstra (ver referencias en la wiki [1]) en su versión $O(E \log V)$. Esto significa a grosso modo que la cantidad de operaciones que realiza la computadora al ejecutar el algoritmo es proporcional a cuanto valga $E \log V$. Como el valor máximo de D es 500000, esto significa en el peor caso demorará aproximadamente $(4 \cdot 500000 + 1) \log(2 \cdot 500000 + 2) \approx 28000000$ operaciones.

Un último detalle es asegurarse de poder calcular los costos de las aristas rápidamente. Esto no es un problema mayor, ya que sabiendo los extremos de cada camino vecinal y que por lo aclarado en el enunciado no hay obstáculos en estos puntos es tan solo sumar las demoras de los obstáculos en cada segmento de carretera delimitado por dos puntos del camino. Para hacer esto podemos tener un arreglo con un elemento por cada segmento de carretera e ir recorriendo linealmente la lista de obstáculos, que vienen ordenados por ubicación.

2.2.2. Problema 2: Planificando la temporada [alquiler]

http://juez.oia.unsam.edu.ar/#/task/alquiler/statement

Este problema se puede resolver de dos maneras.

[1] http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/dijkstra
http://www.oia.unsam.edu.ar/charlas/ (Camino mínimo y AGM)

La primera que daremos como oficial es un **algoritmo goloso** (greedy en inglés). En este tipo de algoritmos a la hora de tomar una decisión, se opta por aquella que resulta mejor en cada momento, sin tener en cuenta qué podría pasar en el futuro como consecuencia. Esto no siempre es conveniente, por ejemplo en una ruta, quizás elegimos ir por una calle en la que vemos menos autos pero que hace que en el futuro vayamos a una congestión.

La manera de utilizar un algoritmo goloso para resolver el problema en este caso, es ordenar a las reservas. Pero, ¿con qué criterio debemos ordenarlas? Podría ser por duración, por fecha de inicio, por nombre... La manera de ordenarlos es por **fecha de finalizacion de la reserva** En general, elegir un criterio por el cuál ordenar los objetos de nuestro problema, y a cada paso tomar el primero disponible según dicho criterio, es un patrón común a muchos algoritmos golosos.

Una vez que están ordenadas,¿qué conviene elegir? Lo que va a convenir es alquilarle a la familia que termina primero. Cada vez que tenemos la localidad vacía, elegiremos de las reservas que están disponibles aquella que termine antes (siuna reserva debía empezar algún día del pasado obviamente no la consideramos).

¿Por qué funciona esto? Si en una solución óptima, en algún momento no le alquilamos a la que termina primero (de las que podemos, claro está), entonces podemos cambiar a esta familia por la familia que sí terminaba primero, y seguiremos teniendo la misma cantidad de confirmados. Como ambas familias podían confirmarse, y además la que estamos metiendo termina antes que la que sacamos, las familias confirmadas que vienen después tendrán la localidad libre. Podemos hacer esto para cada familia confirmada que no era la que terminaba antes, y llegamos a una solución generada por el algoritmo goloso que tiene la misma cantidad de familias confirmadas que una solución óptima.

Otra solución para este problema utiliza **programación dinámica** . Similarmente al caso anterior vamos a ordenar las reservas, ¿pero con qué cirterio? Podemos ordenar por fecha de llegada, de salida, por mail...

Pensemosen la siguiente idea: Supongamos que se le quiere alquilar el departamento a la persona *X* que se quiere alojar entre los días *A* y *B*. Si decidimos alquilarle, ¿cómo debemos seguir? ¿qué problema hay que resolver?. Vamos a tener que resolver un subproblema similar al original, pero *si vamos a empezar a alquilar el departamento desde eldía B*

Supongamos entonces,que vamos a ver sinos conviene alquilarle a *X* y que ya sabemos cuál es la respuesta buscada si empezáramos a alquilar el departamento a partir de cualquier día mayor o igual a *B*. Entonces sabemos que alquilándole

a *X*, la mayor cantidad de alquileres que podemos tener empezando a alquilar en cualquier día *d* entre *A* y *B* será *mejorEmpezandoElDiaB* + 1.

Y así vamos a actualizar nuestras respuestas, yendo hacia atrás en los días (así tenemos calculadas las respuestas a los subproblemas que necesitamos). Entonces, ¿cómo queremos ordenar a los candidatos? Por *fecha de llegada al departamento*, ya que si vamos a mirar la respuesta parcial de *lo mejor empezando en el día D*, queremos tener la respuesta habiendo analizado qué pasa si les alquilamos a todos los que llegarían a partir del día *D* en adelante.

¿Cómo implementamos esto? Para empezar, ordenamos nuestro vector *reservas* mediante una función que ordene según la fecha de llegada.

Luego, vamos recorriendo las reservas desde la última hasta la primera, y guardando en un vector mejorDesde, que en mejorDesde[i] guarda la mejor respuesta que podríamos conseguir si descartáramos a todas las reservas de índice menor a *i* (en nuestro vector ordenado).

Iterando de esta manera, ¿cuál será el valor de mejorDesde[i-1]? Para calcular ese valor, tenemos que pensar que o bien le alquilaremos el departamento a la persona *i*, o no. Si no se lo alquilamos, entonces tendremos el valor de mejorDesde[i], ya que si excluimos a todos los de índice menor a $i - 1$, y también a $i - 1$, estamos excluyendo a todos los índices menores a *i*. Y si le alquilamos a la persona $i - 1$, hay que analizar quién sería la primera persona a la que le podríamos alquilar (porque podrían solaparse los períodos de alquiler). Para saber quién sería esta primera persona, debemos buscar a la persona con menor fecha de llegada, pero que llega después (o el mismo día) a la salida de $i - 1$. Como tenemos a las reservas ordenadas por fecha de llegada, la podemos encontrar haciendo una *búsqueda binaria*. Una vez encontrado el índice de esta persona, llamémoslo *j*, simplemente tendremos que *mejorDesde*[*i*] será mejorDesde[j]+1.

En consecuencia, sabemos que mejorDesde[n]=1 ya que es el último, y mejorDesde[i-1] = max(mejorDesde[i], mejorDesde[j]+1), siendo *j* el índice explicado previamente.

Una vez completado el vector mejorDesde, la respuesta estará en mejorDesde[0], ya que para alquilar no queremos excluir a nadie.

Para concluir el problema, solo resta reconstruir la solución, es decir, queremos los mails de las personas a las que hay que confirmar la reserva. Para eso, cuando estamos calculando mejorDesde[i-1], en vez de simplemente guardar el valor del máximo, podemos poner un if para saber si conviene alquilarlo, o no. Recordemos que al analizar la reserva $i - 1$, va a convenir alquilarla siempre que

mejorDesde[j]+1 >mejorDesde[i]. Luego, si efectivamente conviene alquilarlo, guardamos esa información en algún otro vector.

Finalmente, para reconstruir la solución basta con recorrer desde el primero hasta el final todos los índices y cuando estamos en uno que conviene alquilar (con índice *i*), lo alquilamos y saltamos al *j* correspondiente, el primero con fecha de llegada mayor o igual a la fecha de salida del *i*.

2.2.3. Problema 3: Elucidar entre caballeros y escuderos [elucidar]

http://juez.oia.unsam.edu.ar/#/task/elucidar/statement

En este problema, recibimos como entrada las afirmaciones de cada una de las *N* personas, y tenemos que contar la cantidad de escenarios coherentes con esas afirmaciones.Un escenario es justamente una posible asignación a cada una de las *N* personas, diciendo si es caballero o escudero.Así, con 3 personas, “A y B son escuderos, pero C es caballero” sería un escenario diferente a “A es escudero, pero B y C son caballeros”.

Si observamos las subtareas, este problema tiene la particularidad de “separarse”en dos situaciones principales:Hay subtareas con *N* muy grande, pero en las cuales sabemos que todas las afirmaciones son **atómicas** , y por otro lado hay subtareas con pocas personas (*N* pequeño) pero sin limitación. Encararemos cada una por separado.

2.2.3.1. Subtareas con afirmaciones atómicas

El problema es mucho más simple de analizar en el caso en que sabemos que todas las afirmaciones son atómicas. En este caso, cada afirmación será un único número, positivo o negativo, según la persona esté diciendo que cierta otra persona es escudero,o que es caballero. Analicemos un poco las posibilidades, a ver a qué llegamos.

Supongamos entonces que afirmaciones[i][0] (que tiene el número con la afirmación que hizo la persona *i*, ya que al ser atómica es un solo número y nada más) contenga:

- **Caso 1** : El número (positivo) +*j*. En este caso, tenemos que la persona *i* está diciendo que la persona *j* es caballero (que son los que nunca mienten). Analicemos las 4 posibilidades:

- Ambos caballeros: En este caso, la situación es perfectamente correcta, ya que como *j* es caballero, *i* está diciendo la verdad, y eso es justamente lo que hacen los caballeros.
- Ambos escuderos: En este caso, también tenemos una situación correcta: como *j* es escudero, lo que está diciendo *i* es falso (porque dijo que *j* era caballero) pero es normal que *i* mienta, porque también es escudero.
- *i* caballero, *j* escudero: ¡Esto es imposible! Si analizamos como en los anteriores casos,como *j* es escudero tenemos que *i* mintió, y entonces *i* no puede ser caballero.
- *i* escudero, *j* caballero: ¡Tampoco es posible!La razón es similar: ahora resulta que *i* dijo la verdad, pero entonces no puede ser escudero.

- **Caso 2** : El número (negativo) $-j$. En este caso, tenemos que la persona *i* está diciendo que la persona *j* es escudero (que son los que siempre mienten). Analizando las 4 posibilidades igual que hicimos en el caso 1, llegamos a lo siguiente:
  - Ambos caballeros: Esto es imposible, pues de ser así *i* debería haber dicho la verdad, y dijo que *j* es escudero.
  - Ambos escuderos: Este caso también es imposible: como *j* es escudero, *i* dijo la verdad, pero al ser escudero debería mentir.
  - *i* caballero, *j* escudero: Este caso es perfectamente posible, pues al ser *j* escudero, *i* dijo la verdad, como corresponde al ser caballero.
  - *i* escudero, *j* caballero: Este caso también es posible, pues al ser *j* caballero resulta que *i* mintió, y eso es lo que hacen los escuderos.

Hecho todo el análisis anterior, podemos resumir el resultado obtenido en lo siguiente:

- Caso afirmaciones[i][0] == +j: O bien ambos son caballeros, o bien ambos son escuderos.
- Caso afirmaciones[i][0] == -j: Uno de los dos es caballero, y el otro es escudero (pero no sabemos cuál es cuál).

Supongamos que tuviéramos una variable por cada persona: $v_1, v_2, \cdots, v_n$, de modo tal que $v_i = 0$ si la persona *i* es escudero, y $v_i = 1$ si es caballero.

Lo que hemos descubierto es que cada una de las *n* afirmaciones nos da una relación entre dos personas, que puede ser o de igualdad, o de diferentes:

cuando afirmaciones[i][0] == +j , sabemos que $v_i = v_j$; mientras que cuando afirmaciones[i][0] == -j, lo que sabemos es que $v_i \neq v_j$. El problema ahora es contar la cantidad de posibles combinaciones de cómo asignar 0 y 1 para las variables, que cumplan todas estas *n* restricciones de pares iguales y pares diferentes.

Para esto algo que podemos pensar primero que nada es, ¿Siempre será posible realizar la asignación? Si no se puede, ¿Por qué no se puede? ¿Cuáles son los posibles obstáculos a la asignación?

Los ejemplos más simples que hay son:

- Una relación de la forma $v_i \neq v_i$, que indica que una variable es distinta de sí misma: en la historia original del problema, esto corresponde a alguien que diga “yo soy escudero”: si eso ocurre, la asignación es imposible porque no puede ser ni escudero ni caballero.

- Dos relaciones opuestas: Si tenemos que $v_i = v_j$ y también que $v_i \neq v_j$, evidentemente no se puede cumplir ambas al mismo tiempo.

Los dos ejemplos anteriores tienen relaciones de pares “diferentes”. En el primer ejemplo de hecho,solamente se tiene una relación de diferente,y ninguna relación de igualdad. ¿Habrá alguna entrada posible que use solamente igualdades, y tal que no haya ninguna asignación?

Resulta que no: para que la asignación sea imposible, es necesario tener sío sí relaciones de “diferentes”. Esto es porque si solamente hay relaciones de igualdad, podemos poner a todas las variables en 0 (o todas en 1), y entonces como van a ser todas iguales, se cumplen automáticamente todas las relaciones de igualdad.

¿Hay otra situación en la cual no haya ninguna asignación válida, además de las que ya mencionamos?

La respuesta a esta pregunta es que sí: Consideremos por ejemplo afirmaciones $v_1 \neq v_2$, $v_2 \neq v_3$, $v_3 \neq v_1$. No es posible resolver esto, ya que como solo hay dos valores, al ser v_1 y v_3 opuestos a v_2 tienen que ser iguales,pero nos dicen que son diferentes.

La situación anterior se produce siempre que tenemos un **ciclo de longitud impar** entre las afirmaciones de $\neq$.Es decir, $v_1 \neq v_2$, $v_2 \neq v_3$, $v_3 \neq v_4$, $v_4 \neq v_5$, $v_5 \neq v_1$ tampoco tiene solución, por las mismas razones, mientras que $v_1 \neq v_2$, $v_2 \neq v_3$, $v_3 \neq v_4$, $v_4 \neq v_1$ sí tiene soluciones (por ejemplo: $v_1 = v_3 = 0$ y $v_2 = v_4 = 1$).

Además, podríamos tener una situación como $v_1 \neq v_2$, $v_2 = v_3$, $v_3 \neq v_4$, $v_4 = v_5$, $v_5 \neq v_6$, $v_6 = v_1$ que tampoco es posible: aquí también tenemos un ciclo impar de relaciones de $\neq$, pero este ciclo no es entre "variables" sino entre grupos de variables que sabemos iguales: Es decir, podemos pensar que las relaciones de $=$ **agrupan** i y j, forzándolas a tener el mismo valor, de modo que pueden considerarse ambas para todo fin como si fuera una única variable. Lo mismo si hubiera un grupo de más variables unidas,por ejemplo en $v_1 = v_3$, $v_1 = v_2$, $v_2 = v_4$, las variables 1, 2, 3 y 4 valen todas lo mismo y por lo tanto se comportan como si fueran una sola variable en todos los lugares en que cualquiera de ellas aparece.

Vale la pena mencionar que uno de los primeros casos que mencionamos,el de $v_i \neq v_i$, puede considerarse un ciclo de longitud 1, y por lo tanto impar, así que teniendo eso en cuenta no sería un caso diferente a los demás.

Teniendo todo esto en cuenta, podemos pensar entonces en **unir** las variables relacionadas por un $=$, para identificar los grupos de variables equivalentes. Esto puede realizarse eficientemente mediante DFS o BFS (componentes conexas, donde las aristas son las relaciones de igualdad) o mediante una estructura de Union-Find.

Luego, podemos considerar estos grupos y sus relaciones de $\neq$: Si estas relaciones contienen algún ciclo impar (incluyendo una relación de un conjunto en si mismo, que corresponde en el problema original a una relación de $\neq$ entre dos variables unidas en un mismo grupo de equivalentes), el problema será imposible.

Por otra parte, si no hay ciclos impares, en cada componente conexa de estas relaciones de $\neq$,tendremos solamente 2 opciones posibles:Esto porque una vez que elegimos el valor de una variable, el de todas las demás queda forzado automáticamente, ya que siguiendo las relaciones de igualdad y de $\neq$ van quedando determinados los valores de todas las demás variables. Además estos valores determinados son válidos y no se contradicen, ya que si por dos caminos diferentes pudiéramos llegar a una misma variable asignando 1 por un lado y 0 por el otro, entonces por un camino se habrían usado una cantidad par de $\neq$, y en la otra una cantidad impar, así que uniendo ambos tendríamos un ciclo impar de $\neq$,cosa que ya verificamos que no ocurra.

Por todo esto, la respuesta final será 0, si hay algún ciclo impar como los que describimos, o 2^C sino, donde C es la cantidad de componentes conexas.

Para verificar que no haya ciclo impar entre las relaciones de $\neq$ se puede utilizar también DFS o BFS[2].

[2] http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/bfs

Finalmente, se podría hacer una solución más particular aprovechando que lo que tenemos es un grafo funcional [3]. Estos grafos se caracterizan por tener componentes débilmente conexas formadas por un único ciclo dirigido, y "árboles" colgando de ese ciclo, con las aristas apuntando hacia el ciclo. La función asociada al grafo sería $f(i) = j$ cuando la persona i hizo una afirmación sobre la persona j. Como estas aristas son de dos tipos que ya vimos ($6=$ y $=$), basta encontrar los ciclos y recorrerlos para ver si alguno tiene una cantidad impar de $6=$. Si no hay ciclos impares, entonces la respuesta es 2^C, donde C es la cantidad de ciclos (notar que podemos tener ciclos que sean bucles, es decir, una arista de un nodo a sí mismo).

2.2.3.2. Subtareas con afirmaciones generales

En estas subtareas tenemos que la cantidad de personas N es $N < 30$. Se puede demostrar que contar la cantidad de asignaciones válidas de caballeros y escuderos es un problema $\#P$ -completo, lo que implica que no se conoce (ni se cree que exista) ningún algoritmo polinomial para resolverlo en forma exacta.

Notar que el algoritmo de fuerza bruta más directo tendría una complejidad $O(N2^N L)$, siendo L la longitud de las expresiones. Esto es porque hay 2^N posibles asignaciones, y si las probamos todas para ver cuáles son posibles y cuáles no, para probar cada una de ellas tenemos que leer cada una de las N afirmaciones y evaluar la expresión lógica para verificar que sea igual a true (cuando la persona es caballero) o a false (cuando es escudero).

Una mejora posible a esta solución de fuerza bruta, es realizar la evaluación lógica pero utilizando una evaluación de "cortocircuito": para esto tenemos que leer las expresiones **de derecha a izquierda** . Si tenemos una expresión que termina en AND false, no hace falta mirar todo lo anterior: ya podemos dar directamente el resultado, que será false. Si en cambio termina AND true, este último paso no altera el resultado, y podemos seguir analizando la parte más interna de la expresión. Análogamente, OR true da inmediatamente true, pero OR false.

Una segunda mejora posible es observar la forma particular que tienen las expresiones lógicas de este problema, donde cada persona enuncia algo de la forma $(((l_1 \; 4 \; l_2) \; 4 \; l_3) \; 4 \cdots) \; 4 \; l_k$: cada 4 representa una operación de "AND" o de "OR" lógicos, y además cada l_i es una afirmación de la forma $x_i = 0$ o bien $x_i = 1$. En este problema, $L \leq 100$, pero 100 es mayor que 30 que es el mayor N posible en este caso.Por lo tanto si las expresiones son muy largas,tendrán necesariamente valores de x repetidos entre sus afirmaciones atómicas. Pero no tiene sentido en una expresión de esta forma, que el mismo x aparezca más de una vez: al ir evaluando

[3] http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/grafos-funcionales

en cortocircuito la expresión, al llegar por segunda vez a un mismo x_i, ya se sabe el valor que tendrá x_i gracias a la aparición previa. Con eso se puede determinar si la operación que se realiza con el segundo x_i no hará nada, o si terminará la evaluación (en cuyo caso, el resto de la expresión es redundante). Esto permite cambiar todas las expresiones por una completamente equivalente pero más corta,donde cada x_i aparece a lo sumo una vez.

Como ejemplo de esta simplificación, consideremos $(((x_6 \& x_5) | x_1) \& x_2) | no(x_1)$. Si evaluamos de derecha a izquierda en cortocircuito, sabemos que si $x_1 = 0$, automáticamente toda la expresión queda verdadera. Por lo tanto, la parte que sigue, $(((x_6 \& x_5) | x_1) \& x_2)$, solo es considerada cuando $x_1 = 1$. Podemos entonces ya calcular esta parte asumiendo que $x_1 = 1$. Esto garantiza que $((x_6 \& x_5) | x_1)$ será siempre verdadero (1) cuando el algoritmo lo considera, y como $1 \& x_2 = x_2$, podemos reemplazar todo el resto de la expresión por simplemente x_2, sin cambiar nada. La expresión simplificada queda entonces $x_2 | no(x_1)$, que es exactamente equivalente a la original.

Si en cambio hubiéramos tenido $(((x_6 | x_5) \& x_1) \& x_2) | no(x_1)$, con el mismo razonamiento al saber que $x_1 = 1$ siempre que el algoritmo continúa evaluando luego de la primera aparición (la de más a la derecha), sabemos que $((x_6 | x_5) \& x_1)$ será lo mismo que directamente $(x_6 | x_5)$, o sea que pudimos borrar completamente esta aparición de x_1 sin cambiar nada. La expresión nos queda entonces $((x_6 | x_5) \& x_2) | no(x_1)$, que es completamente equivalente a la original pero no tiene variables repetidas.

Siempre se puede simplificar de esta misma manera hasta que no queden variables repetidas, recorriendo la expresión de derecha a izquierda y recordando el valor que le queda fijado a cada variable luego de que aparece por primera vez, para así “resolver la situación directamente” cuando vuelva a aparecer: según el valor y la operación, puede haber que saltear directamente esa variable porque no aporta nada (como en el segundo ejemplo), o puede ser que directamente ya se termine la evaluación ahí, recortando así parte de la expresión (como en el primer ejemplo).

Entonces, simplemente simplificando las expresiones tendremos $L \leq N$, y obtenemos haciendo lo mismo de antes una complejidad $O(N^2 2^N)$. Y usando estas dos optimizaciones a la vez (simplificar las expresiones, y realizar una evaluación en cortocircuito), se puede demostrar que la complejidad de la fuerza bruta se reduce a $O(N 2^N)$

La solución completa esperada es utilizar un algoritmo de backtracking [4] con

[4] http://wiki.oia.unsam.edu.ar/algoritmos-oia/backtracking

podas.

Algunas podas e ideas (pero no las únicas posibles) que se pueden utilizar para obtener 100 puntos:

- Simplificar las expresiones, como se explicó antes.
- Buscar variables con un valor **forzado**, y ya reemplazarlas por su valor, reduciendo las expresiones. Por ejemplo, si la persona 1 dijera $x_2 \& no(x_1)$, ya se puede deducir que tiene que ser escudero, pues si dijera la verdad debería ser $no(x_1)$, pero entonces la primera persona sería escudero. Además, al ser escudero está mintiendo y $x_2 \& no(x_1)$ tiene que ser falso: esto solo es posible si x_2 es falso. En conclusión, ya podemos deducir que $x_1 \neq x_2 = 0$ y reemplazarlo en todas las demás expresiones para seguir analizando. Esta única poda es de hecho muy buena, y usando esta sola (además de simplificar) bastaba en este problema para obtener 100 puntos sin realizar backtracking, sino realizando fuerza bruta únicamente en las variables que quedan "sin decidir" luego de esta simplificación.
- Buscar si las variables se pueden separar en "dos conjuntos no relacionados", como si fueran dos problemas independientes entre sí. De ser así, puede resolverse cada uno por separado, y la respuesta será el producto de ambas cantidades.
- Utilizar máscaras de bits para implementar la solución más eficientemente. Se puede leer sobre máscaras de bits en http://wiki.oia.unsam.edu.ar/cpp-avanzado/operaciones-de-bits
- Al hacer backtracking, fijar primero el valor de la variable que más aparece, porque es lo que más "simplificará expresiones" (ya que esa variable tiene más efecto sobre el problema que una variable que aparezca en un único lugar, por ejemplo).

Capítulo 3

Certamen Jurisdiccional

3.1. Nivel 2

3.1.1. Problema 1: ¡Piedra, Papel, Tijera! [ppt]

http://juez.oia.unsam.edu.ar/#/task/ppt/statement

Este es un problema en el que se pueden pensar todas las posibilidades “a mano”. A veces vamos a tener problemas en los que haya muchos posibles inputs, pero este no es uno de ellos.Cada jugador tiene 3 posibilidades, lo que nos da un total de $3 \times 3 = 9$ posibles inputs distintos.

Podemos resolverlo de varias maneras, algo que podemos hacer es primero ver si ambos jugadores jugaron lo mismo. En ese caso,sabemos que es empate.Y ya analizamos 3 casos por el precio de 1.

Luego, para los otros casos,algo que podemos hacer es ver primero qué jugó el primero, y para cada posibilidad, ver si estamos en una de las posibilidades en las que, por ejemplo, gana B. Si no estamos en ninguna de esas, sabemos “por descarte” que va a ganar A. Entonces un pseudocódigo que resuelve elproblema podría ser algo como:

Algorithm 1 Solución al Problema 1 Nivel 2 Jurisdiccional 2017

```
1:  procedure Problema1Nivel2Jurisdiccional2017
2:      A ← leerT exto()
3:      B ← leerT exto()
4:      if A == B then
5:          imprimir(“Empate”)
6:      else
7:          ganaA ← True                                  . este es “el descarte”.
8:                                  . Si gana B habra que cambiar esta variable.
9:          if A == “Piedra” then
10:             if B == “Papel” then
11:                 ganaA ← False
12:         else if A == “Papel” then
13:             if B == “Tijera” then
14:                 ganaA ← False
15:         else                                          . Sabemos que es tijera
16:             if B == “Piedra” then
17:                 ganaA ← False
18:         if ganaA then
19:             imprimir(“Ana”)
20:         else
21:             imprimir(“Bartolo”)
```

```
#include<iostream>

using namespace std;

int main() {
	string A, B;
	cin>>A>>B;
	if(A==B) {
		cout<<"Empate";
	} else{
		bool ganaA=true; // este es "el descarte".
					// Si gana B habra que cambiar esta variable
		if(A=="Piedra") {
			if(B=="Papel"){
				ganaA=false;
			}
		} else if(A=="Papel") {
			if(B=="Tijera"){
				ganaA=false;
```

```
            }
        } else {// sabemos que A es tijera
            if(B=="Piedra") {
                ganaA=false;
            }
        }
        if(ganaA) {
            cout<<"Ana";
        } else {
            cout<<"Bartolo";
        }
    }
}
```

Otra manera un poco más compleja de resolverlo, podría ser usando un "map". Se puede leer sobre esa estructura acá:http://wiki.oia.unsam.edu.ar/cpp-avanzado/map. Básicamente, lo que hace es asociar, para un valor llamado "key" o clave, un valor. Entonces, la "key" en nuestro caso podría ser el par de strings (*A, B*), y que el valor asociado sea la respuesta esperada. Entonces por ejemplo tendríamos que el valor asociado a ("Piedra", "Papel") es "Bartolo", y el valor asociado a ("Papel", "Piedra") es "Ana".

Luego de setear los valores asociados para todos los posibles pares de palabras, simplemente hay que imprimir el valor asociado del mapa de (*A, B*).

3.1.2. Problema 2: Días Feriados [feriados]

http://juez.oia.unsam.edu.ar/#/task/feriados/statement

En el problema nos dan 3 números F ,D y N . Donde F es la cantidad de veces que podemos faltar sin que nos expulsen,D es la cantidad de días que tiene el calendario escolar (numerado de 1 a D inclusive), y N es la cantidad de días sin clase que hay en el calendario escolar de este año.A continuación se presentan N números, $f_1, \ldots, f_N$, que es la lista de los N días sin clase.

Una forma de visualizar y representar la entrada para pensar el problema, es hacer un arreglo de D posiciones (que vamos a numerar de 0 a $D - 1$ inclusive), y ubicaremos un 0 o un 1 en la i-ésima posición del arreglo, según si en el día $(i + 1)$ hay clase o no (un 0 si hay clase y un 1 si no hay clase).

Por ejemplo, si la entrada tiene $D = 10$, $N = 4$, $f_1 = 1$, $f_2 = 4$, $f_3 = 6$, $f_4 = 9$, queda presentado de la siguiente forma ($\hat{f}_i = f_i - 1$ por el cambio de indexación):

f_1			f_2		f_3			f_4	
↓			↓		↓			↓	
0	1	2	3	4	5	6	7	8	9
F	C	C	F	C	F	C	C	F	C

$\longrightarrow$

0	1	2	3	4	5	6	7	8	9
1	0	0	1	0	1	0	0	1	0

Una vez que tenemos esta forma de ver el problema, podemos entender qué nos pide el enunciado en términos de esta representación (o sea, el arreglo de la derecha, que a partir de ahora llamaremos A, y denotaremos con $A[i]$ a su i-ésima posición).

Como el enunciado dice "*Una vez que Javier comience a trabajar en su auto* ***no parará hasta terminarlo***", concluimos que Javier trabajará en **días consecutivos** (es decir, en un **subarreglo**). Ahora, en esos días consecutivos que Javier dedique a construir su auto, **no puede haber más de F días sin clase**.

Un subarreglo podemos definirlo utilizando solamente sus extremos. Utilizaremos la **convención cerrado-abierto**. Es decir, el subarreglo $[i, j)$ incluye todos los índices desde i hasta j incluyendo i, pero sin incluir j. Notemos que **el subarreglo $[i, j)$ está compuesto por $j - i$ números**. En el ejemplo anterior, subarreglos posibles son $[i_1, j_1) = [3, 8)$ o $[i_2, j_2) = [6, 10)$, como muestra la figura:

			i_1					j_1		
			↓					↓		
0	1	2	3	4	5	6	7	8	9	10
1	0	0	1	0	1	0	0	1	0	

						i_2				j_2
						↓				↓
0	1	2	3	4	5	6	7	8	9	10
1	0	0	1	0	1	0	0	1	0	

Entonces surge la pregunta, **¿cuántos subarreglos posibles hay?** Podemos identificar a cada subarreglo con un par (i, j) con $0 \le i < j \le D$. Por lo tanto, tenemos D posibilidades para j (desde 1 hasta D) y para cada elección de j, tenemos j posibilidades para i (desde 0 hasta $j - 1$). Entonces, tenemos

$$1 + 2 + 3 + 4 + \cdots + D = \frac{D \cdot (D + 1)}{2}$$

subarreglos posibles.

Una primera idea que se nos puede ocurrir, es **probar todos los subarreglos posibles**. Dado un subarreglo fijo, tenemos que chequear que no haya más de F días sin clase en dicho subarreglo. Notemos que **la cantidad de días sin clase en un subarreglo de A es exactamente su suma** (o sea, $A[i] + A[i + 1] + \cdots + A[j - 1]$). Con esto en mente, ya podemos esbozar una primera solución.

Solución 1 :

- Comenzar con respuesta = $-\infty$
- Probar todos los subarreglos $[i, j)$ posibles.
- Calcular la *suma* de los números en el subarreglo.
- Si la *suma* es menor o igual a F y $j - i >$ respuesta, actualizar respuesta por $j - i$.
- Imprimir respuesta.

Dependiendo de cómo hagamos eltercer ítem, obtendremos un algoritmo de menor complejidad. Una primera opción para **calcular la *suma*** de un subarreglo $[i, j)$ es **recorrer lugar a lugar**. Esto es lineal en el subarreglo. En total, nos quedaría un algoritmo de complejidad $\boxed{O(D^3)}$ para resolver el problema (hay $O(D^2)$ subarreglos, y calcular su *suma* cuesta $O(D)$ para cada uno de ellos).

Otra opción es construir un arreglo auxiliar de *suma de prefijos* (comúnmente llamado ***tablita aditiva***). Este nuevo arreglo (que llamaremos S_A), consta de $D + 1$ posiciones, y guarda en el i-ésimo lugar la suma de todos los números en el intervalo $[0, i)$, es decir: $S_A[i] = A[0] + A[1] + \cdots + A[i-1] = \sum_{j=0}^{i-1} A[j]$. Notar que $S_A[0] = 0$, pues $[0, 0) = \emptyset$.

Ejemplo (cambiando momentáneamente el arreglo, para evidenciar que no tiene nada de especial en esta parte que nuestro arreglo tenga solo ceros y unos).

i:	0	1	2	3	4	5	6	7	8	9	10
A:	2	3	1	0	-1	4	-2	3	5	-2	*
SA:	0	2	5	6	6	5	9	7	10	15	13

Una forma de generar S_A es usar que $S_A[0] = 0$ y notar que para i desde 1 en adelante vale que $S_A[i] = S_A[i-1] + A[i-1]$.

Luego, si queremos saber la suma en el subarreglo $[i, j)$, simplemente debemos computar : $S_A[j] - S_A[i]$. Por ejemplo, en el arreglo anterior, para calcular la suma en el intervalo $[2, 7)$ podemos hacer $S_A[7] - S_A[2] = 2$.

Usando esta técnica en el tercer ítem de la solución vista, nos quedaría una complejidad de $\boxed{O(D^2)}$ para resolver el problema, pues la operación de la suma en un subarreglo demora un tiempo constante.

Veamos cómo obtener soluciones más eficientes. Todavía hay mucha estructura del problema que no estamos utilizando. Observemos lo siguiente:Si un intervalo $[i, j)$ tiene una suma menor o igual a F , entonces todos los subintervalos contenidos

en el subintervalo [*i, j*) también. Aquí estamos usando que **todas las posiciones del arreglo tienen números no negativos** (más aún, son 0 o 1), porque gracias a esto, al sacar elementos de algún extremo, la suma total siempre disminuye.

Entonces, usando la observación anterior, **si encontramos un intervalo de largo** L **con suma menor o iguala** F, sabemos que **existen intervalos con largo** $L-1, L-2, \ldots, 1, 0$ **con suma menor o igual a** F (por ejemplo, sacando elementos desde elextremo derecho al subintervalo original). De la misma forma, sabemos que si no hay un intervalo de largo L con suma menor o igual a F, tampoco habrá intervalos con largos mayores (pues de haberlos,podríamos obtener uno de largo L sacando elementos desde un extremo de este hipotético intervalo con largo mayor).

Esto nos da la pauta de que podemos hacer búsqueda binaria en el largo del subintervalo (que es exactamente la respuesta que buscamos). Queremos el mayor largo de un subintervalo con suma menor o igual a F. Inicialmente sabemos que hay un intervalo de largo 0 con suma menor o igual a F (por ejemplo el intervalo vacío) y sabemos también que no puede haber un intervalo de largo $D+1$ con suma menor o igual a F (pues no hay intervalo de largo $D+1$).

Solución 2 :

- Tomar a = 0 y b = D+1, como límites de la búsqueda binaria
- A cada paso de la *búsqueda binaria* (hasta que a y b sean consecutivos):
  - Considerar longitud = $b \quad \frac{(a+b)}{2} c$
  - Chequear todas las *sumas* de subintervalos de largo longitud (por ejemplo, usando S_A, aunque no es estrictamente necesario).
  - Si alguna de esas *sumas* es menor o igual a F, actualizar a = longitud. Sino, si todas son mayores a F, actualizar b = longitud.
- Imprimir a → respuesta (Notar que al finalizar la *búsqueda binaria*, tenemos guardado en a el **mayor largo para el que existe un subintervalo del arreglo original de ese largo con suma menor o igual a** F)

De esta manera, obtenemos un algoritmo de complejidad $\boxed{O(D \lg(D))}$ que resuelve el problema. La clave fue que al utilizar ***búsqueda binaria***, pasamos a tener que resolver un problema más sencillo donde tenemos fijo el tamaño del subintervalo (que sería longitud) y **solo tenemos que chequear si hay algún subintervalo que cumpla de ese largo específico** .

Sigamos analizando elproblema. Observemos ahora **qué ocurre alfijar el extremo izquierdo de un intervalo** Comenzando con el subintervalo que solo contiene al *i*-ésimo elemento, si vamos extendiendo a ese intervalo desde el extremo derecho, la suma de ese subintervalo, siempre se va agrandando (otra vez, estamos usando que los números son 1 o 0, que son no negativos).

Esto nos da otro algoritmo de complejidad $O(D \lg(D))$, que consiste en fijar el extremo izquierdo, y para cada i hacer búsquda binaria en el j (para obtener lo que llamaremos j_i, el mayor j tal que el subintervalo $[i, j)$ tiene suma menor o igual a F), y finalmente, tomar el más largo de estos intervalos.

Pensando en esta forma de resolver el problema, veamos **cómo encontrar para cada extremo izquierdo *i* a ese *j*** $_i$, pero de manera más eficiente.

Supongamos que ya tenemos un intervalo $[i, j)$ (notar que tiene suma menor o igual a F y la suma en $[i, j_i + 1)$ se pasa de F). Para este i, ya obtuvimos la respuesta. Ahora, ¿qué ocurre con j_{i+1} para el intervalo que empieza en $i + 1$?

El intervalo $[i + 1, j_i)$, tiene suma menor o igual a F (por estar contenido en $[i, j_i)$), por lo tanto, j_{i+1} estará más a la derecha que j, en otras palabras, $j_{i+1} \geq j_i$ para todo i. Otra forma de pensar esto es que j_i **es creciente en** i.

Entonces, podemos aplicar una ***ventana deslizante*** para no desperdiciar el valor de j_i a la hora de calcular el valor de j_{i+1}, manteniendo en todo momento la suma en la ventana $[i, j)$.

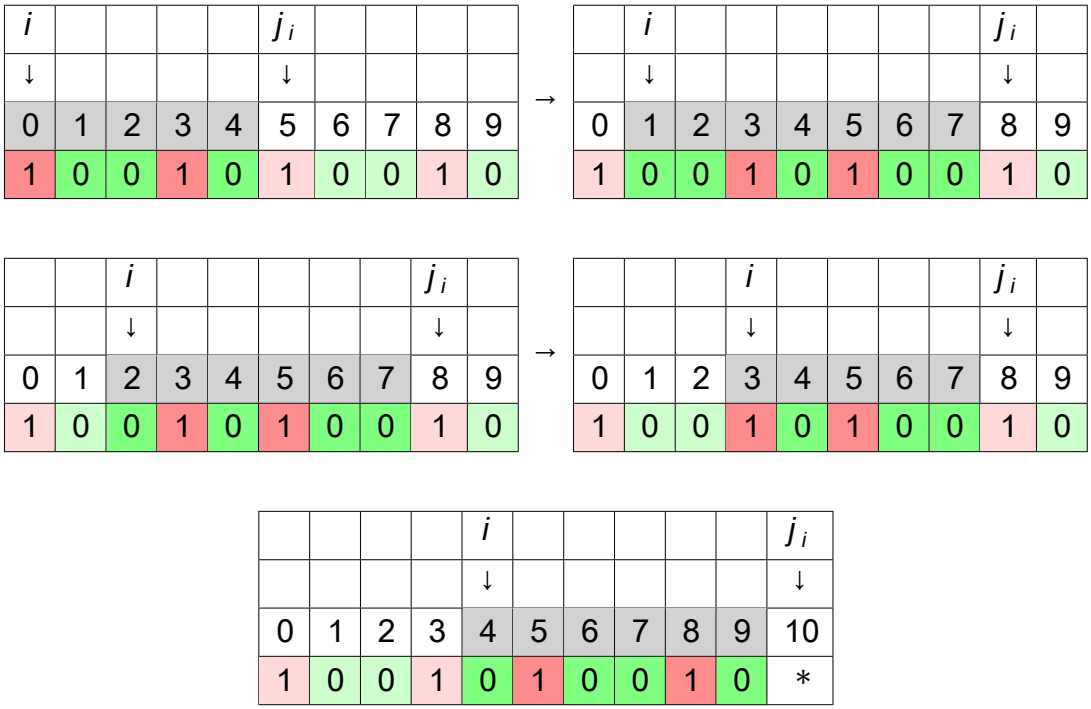
Solución 3 :

- Comenzar con j = 0, ventana = 0, respuesta = $-\infty$.
- Para cada extremo izquierdo i en orden creciente:
  - Mientras ventana ≤ F: Sumar $A[j]$ a ventana y aumentar j en uno.
  - Si j-i > respuesta, actualizar respuesta = j-i (Notar que al terminar el ítem anterior, j toma el valor deseado de j_i. Hay un detalle, no hay que aumentar j si ya llegamos al final)
  - Como vamos a aumentar i, restamos $A[i]$ a ventana
- Imprimir respuesta

Esto nos da un algoritmo de complejidad $\boxed{O(D)}$ que resuelve el problema. Notar que comenzamos con ventana = 0, que representa la suma en el intervalo vacío dado por [0, 0).

Alcanza dicha complejidad, porque **en cada paso o bien se aumenta i o se aumenta j, y cada uno de ellos no puede aumentarse más de D veces** (y las actualizaciones de i,j,ventana toman tiempo constante).

Veamos los pasos del algoritmo en el ejemplo del principio cuando $F = 2$.



El mayor valor de $j_i - i$ obtenido a lo largo del proceso es 7 en el segundo paso del algoritmo, que corresponde a la mayor cantidad de días seguidos sin ir a clase, faltando hasta 2 veces al colegio.

3.1.3. Problema 3: Recorriendo Venecia [venecia]

http://juez.oia.unsam.edu.ar/#/task/venecia/statement

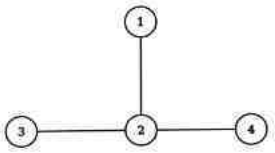
Para empezar, notemos que al problema lo podemos modelar como un grafo: Las esquinas son nodos, y las cuadras, que conectan siempre dos esquinas, son aristas. Entonces el problema lo vamos a pensar como, en un grafo, encontrar un camino que recorra todas las aristas, empezando y terminando en un mismo nodo dado.

Algo que podemos pensar primero es “cómo encontrar el camino más corto”. Ahora, veamos que en un grafo sencillo, 1-2-3, si empezáramos desde el nodo 1, no nos queda otra que recorrer cada cuadra dos veces. Por lo tanto, vemos que hay casos en los que no podemos mejorar el $2 \cdot L$ dado por el enunciado.

El enunciado dice que se da puntaje perfecto a $2 \cdot L$, por lo que veremos la solución que recorre todo el grafo en a lo sumo $2 \cdot L$. Lo interesante de esa cota, es que

lo que podemos hacer es pasar 2 veces por cada arista. Si por alguna pasáramos una vez, podríamos por otra pasar más de 2 veces, pero concentrémonos en encontrar un camino que pasa por todas las aristas a lo sumo 2 veces. Algo que podemos hacer es, desde elpunto inicial, ir recorriendo cuadras siempre que tengamos,desde nuestra posición actual, una cuadra que todavía no recorrimos. Entonces podemos seguir así hasta que llegamos a un lugar tal que todas sus cuadras ya fueron recorridas. Bueno, desde ahí podemos volver por este camino al punto de partida. Y de esta manera ya recorrimos todas las aristas de ese camino dos veces, y ya “nos sacamos de encima” a ese camino.

Esta idea es interesante, pero qué pasa cuando tenemos por ejemplo 4 nodos, 3 aristas: El nodo 2 en el "medio", conectado al 1, 3 y 4:



Si tuviéramos que empezar desde el 1, según nuestro algoritmo que esbozamos recién, iríamos del 1 al 2, luego al 3 por ejemplo, y ahí volveríamos al 2 y luego al 1. Pero ahora lo que nos queda por hacer es sí o sí ir al 2, luego al 4, y ahí volver. Pero terminamos pasando por todas las cuadras dos veces excepto por la que conecta al 1 con el 2, que pasamos 4 veces.

Pensando en este ejemplo, algo que podemos es “no retroceder de un nodo hasta que no agotamos todos los caminos desde éste”. En nuestro ejemplo, sería “no volver del 2 al 1, si todavía tenemos la arista 4-2 no recorrida”. Entonces haríamos el camino $1 - 2 - 3 - 2 - 4 - 2 - 1$, cumpliendo nuestro objetivo de no pasar más de dos veces por ninguna arista.

Esta idea es muy cercana a la idea de DFS, que podemos traducir como *Búsqueda en Profundidad*, y hace eso: hasta que no agotamos un nodo no retrocedemos. Se puede leer más sobre esta idea acá: http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/dfs

Ahora, pensemos qué pasa si cada vez que llegamos a un nodo nuevo, que antes no había sido visitado, agotamos todas las aristas que llegan a él. Como en nuestro grafo podemos tener ciclos (una típica manzana podría ser un ciclo de 4 nodos), hay que tener cuidado ya que no siempre, agotando todas las aristas de un nodo, vamos a ir hacia nodos no recorridos. Pero esto en realidad no sería un problema,

ya que si al llegar a un nodo recorrido, tenemos aristas sin recorrer, entonces sí o sí tendríamos que volver a pasar por este nodo para utilizar esta arista.

Veamos que nuestra solución efectivamente recorre todas las cuadras a lo sumo dos veces. Sea c la cuadra que conecta a los nodos a y b. Supongamos que la primera vez que recorremos esta cuadra es de a a b. Luego, seguiremos recorriendo el grafo. Si en algún momento de nuestro recorrido estamos en el nodo a, porque llegamos a través de algún ciclo por ejemplo, entonces no vamos a usar la arista c ya que ya la usamos. Y lo mismo si llegamos a b a través de algún ciclo. La única vez que recorreríamos c de vuelta, sería de b a a, y esto sería porque ya agotamos todas las cuadras que usan al nodo b.

Entonces sabemos que nuestro algoritmo pasará a lo sumo dos veces por cada cuadra, obteniendo un recorrido total menor o igual a $2 * L$.

Ahora vamos al código. ¿Cómo guardar nuestro grafo? En generalse guarda para cada nodo, una lista de nodos adyacentes,es decir con los que comparte una arista. Pero en este problema, las aristas tienen longitudes, y además tienen índices que son muy importantes ya que nos lo piden para la salida. Algo que podemos hacer para ordenarnos mejor, es tener una objeto (en C++ struct, en Java Class), llamado por ejemplo Vecino, que tenga un *número de nodo*, el *índice de arista* y la *longitud de esta arista*, y entonces cada nodo tendrá una lista de elementos de tipo Vecino.

¿Cómo armar el algoritmo? Podemos tener una función recursiva DFS, que cada vez que entramos a ella con un nodo, va a cada vecino y llama a esta nueva función, que nunca usará aristas ya utilizadas. Y podemos tener dos variables globales, *camino*, en donde agreguemos las aristas cada vez que utilizamos una de ellas. Y *utilizada*, que en la posición i tenga un *bool* que nos diga si ya utilizamos la arista i o no.

De hecho, como vamos a recorrer todas las aristas dos veces, ni nos importa la longitud de la arista, ya que garantizamos una longitud total igual a $2 \cdot L$. Vamos a guardar el dato por comodidad.

Algorithm 3 Solución al Problema 3 Nivel 2 Jurisdiccional 2017

```
1: procedure crearVecino(nodo, arista, largo)
2:     vecino ← Vecino()
3:     vecino.nodo ← nodo
4:     vecino.arista ← arista
5:     vecino.largo ← largo
```

```
1: procedure DFS(nodo, camino, grafo, utilizado)
2:                          . Debemos pasar camino, grafo y utilizado por referencia
3:     for vecino in grafo[nodo] do
4:         if utilizada[vecino.arista] then
5:             continue
6:         camino.agregar(vecino.arista)
7:         utilizada[vecino.arista] ← True
8:         DFS(vecino.nodo, camino, grafo, utilizado)
9:         camino.agregar(vecino.arista)
10:                                  . el resultado esta almacenado en camino
```

```
1: procedure Problema3Nivel2Jurisdiccional2017
2:     N ← leerEntero()
3:     M ← leerEntero()
4:     inicio ← leerEntero()
5:     grafo ← Vector[N] {NuevoVector(), ..., NuevoVector() }
6:     for i = 1 to M do
7:         A ← leerEntero()
8:         B ← leerEntero()
9:         L ← leerEntero()
10:        grafo[A].agregar(crearVecino(B, i + 1, L))
11:        grafo[B].agregar(crearVecino(A, i + 1, L))
12:    camino ← lista()
13:    utilizado ← Vector[M+1]False, ..., False
14:    DFS(inicio, camino, grafo, utilizado)
15:    imprimir(camino.tamaño())
16:    for i = 1 to camino.tamaño() do
17:        imprimir(camino[i])
```

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

using namespace std;

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

struct vecino{
    int nodo, arista, L;
    vecino(int n, int a, int l) : nodo(n), arista(a), L(l) {}
};
```

```
vector<bool> utilizada;
vector<int> camino;
vector< vector<vecino> > grafo;

void DFS(int nodoActual){
	forn(i, grafo[nodoActual].size()){
		vecino v = grafo[nodoActual][i];
		if(utilizada[v.arista]){
			continue;
		}
		camino.push_back(v.arista);
		utilizada[v.arista]=true;
		DFS(v.nodo);
		camino.push_back(v.arista);
	}
}

int main(){
	int n, m, inicio;
	cin>>n>>m>>inicio;
	grafo.resize(n+1);
	utilizada.resize(m+1);
	forn(i, m){
		int a, b, l;
		cin>>a>>b>>l;
		grafo[a].push_back(vecino(b, i+1, l));
		grafo[b].push_back(vecino(a, i+1, l));
	}
	DFS(inicio);
	cout<<camino.size()<<endl;
	forn(i, camino.size()){
		cout<<camino[i]<<" ";
	}
}
```

3.1.4. Problema 4: Tanques de Agua [tanque2]

http://juez.oia.unsam.edu.ar/#/task/tanque2/statement

En este problema tenemos una estructura de tanques conectados por cañerías. Cada tanque, salvo el principal, tiene un caño de entrada que viene de otro tanque y puede tener uno o más caños de salida que lo conectan con otros tanques. A esta estructura la podemos modelar como un árbol donde cada nodo es un tanque y los hijos de un nodo son los tanques a los que está conectado por caños de salida.De ésta manera la raíz del árbol será el tanque principal.

Esta estructura de tanques se va llenando con agua. El problema nos pide que agreguemos un nuevo tanque de modo que sea el *K*-ésimo en llenarse completamente donde *K* es un número que viene en la entrada. Primero veamos cómo podemos descubrir, dada una estructura, en qué orden se llenan los distintos tanques.

El llenado sigue las siguientes reglas:

- Un tanque no se llena hasta que todos sus hijos se hayan llenado.
- El orden en que se llenan los hijos de un tanque es el orden en que están conectados los caños de salida de abajo hacia arriba. Es decir, de mayor a menor distancia a la boca del tanque.

Esto nos sugiere que para encontrar este orden podemos hacer un *dfs* (búsqueda en profundidad) sobre el árbol de tanques donde los hijos de un nodo los recorremos en orden de abajo hacia arriba. Un tanque se llena justo cuando terminamos de procesar a todos sus hijos. Lo que podemos hacer es agregar cada tanque al final de un vector de llenados cuando lo terminamos de procesar en el *dfs*. Al final nos queda en el vector el orden en que se llenan los tanques. A este orden se lo conoce como un *postorden* del árbol.

A continuación un posible algoritmo que calcula este orden. En este algoritmo *tanqueActual* representa el número de tanque que estamos llenando, *arbolT anques* guarda para cada tanque un vector con los tanques hijos, ordenados de menor a mayor altura y ordenLlenado es el vector que representa los tanques que ya se llenaron en el orden en que lo hicieron.

Algorithm 4 Pseudocódigo de la simulación de llenado

```
1: procedure simularLlenado(tanqueActual, arbolTanques, ordenLlenado)
2:     for hijo ← arbolT anques [tanqueActual] do
3:         simularLlenado(hijo, arbolT anques, ordenLlenado)
4:     ordenLlenado.agregarAlF inal(tanqueActual)
```

¿De qué nos sirve el orden de llenado de los tanques para resolver el problema?

Hay una estrategia sencilla que funciona que es la siguiente:

- Identifico cuál es el K-ésimo tanque en llenarse en la estructura original, llamémoslo T_k.
- Agrego el nuevo tanque como caño de salida de T_k a distancia 1 de la boca (lo más alto posible).

¿Por qué esto funciona?

La primera observación es que el orden en que se llenan los tanques no cambia salvo que se agrega el tanque nuevo entre el llenado de dos tanques viejos (o al principio). La segunda es que siseguimos esta estrategia lo único que cambia a la hora de llenarse los tanques es que justo antes de terminar de llenarse el tanque T_k se empieza a llenar hasta que lo hace completamente el tanque nuevo, y luego se llena T_k. Esto quiere decir que primero se llenan los $K - 1$ tanques de la estructura original, luego se llena el tanque nuevo y luego el tanque T_k, por lo que el nuevo es el k-ésimo en llenarse.
¿Qué pasa si ya hay otro tanque con un caño de salida a distancia 1 de la boca de T_k?

En este caso es imposible realizar lo pedido ya que el tanque nuevo se debería llenar entre el $K - 1$ y K en el orden original, pero el $K - 1$-ésimo en llenarse debe ser el tanque a distancia 1 de la boca de T_k, llamémoslo T_{k-1}. Esto nos dice que entre el llenado de T_{k-1} y el de T_k lo único que sucede es que se agrega el último litro de agua en T_k. Por lo tanto el nuevo tanque se debería agregar en T_k para evitar que se llene T_k pero tiene que agregarse arriba de la boca de salida que va a T_{k-1} para llenarse después que este. Como no hay espacio para realizar esto concluimos que este caso es imposible de resolver.
El enunciado nos dice que todos los casos son posibles por lo que podemos asumir que siempre es posible agregar el nuevo tanque a distancia 1 de la boca de T_k

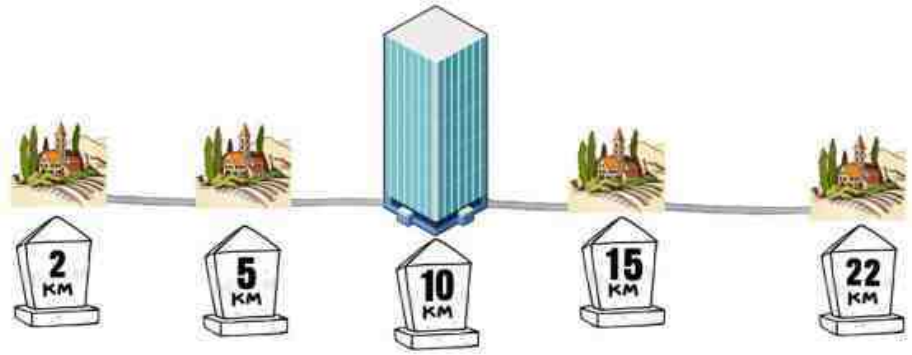
3.2. Nivel 3

3.2.1. Problema 1: Cableando por la ruta [cableando]

http://juez.oia.unsam.edu.ar/#/task/cableando/statement

Para analizar este problema, es muy bueno realizar dibujitos de análisis y ejemplos para tratar de encontrar alguna **relación** entre los números de entrada y

la respuesta. Es útil también aprovechar lo que tenemos,y empezar observando el dibujo de ejemplo que viene en el enunciado:



Vemos que en este ejemplo, lo mejor posible es realizar un cableado “lineal”: si bien está partido de a trozos, como todos los pueblos están en una misma ruta, el total de cable se extiende desde elpueblito más a la izquierda en el dibujo (el que está en el kilómetro 2) hasta el pueblito más a la derecha en el dibujo (el que está en el kilómetro 22). Justamente esa es la cantidad total de cable utilizada: $22 - 2 = 20$ es la distancia en kilómetros entre los dos pueblitos “extremos”.

Lo que ocurre en este ejemplo no es casualidad, sino que es un patrón que podemos observar en general: si ya tenemos armada una red de conexiones de alguna manera, como existe en la red alguna ruta por los cables que viaja desde uno de los pueblitos extremos hasta la megacentral, y también existe ruta desde esta megacentral hasta el otro pueblito extremo, al unir ambas tendremos una secuencia de conexiones con cables que viaja desde un pueblito extremo hasta el otro. Esta secuencia de cables eventualmente va pasando por pueblitos intermedios, y toca la megacentral, como ocurre en el ejemplo anterior. Pero en total, la cantidad de kilómetros viajada por la red para ir de un pueblo al otro siempre será, **como mínimo** , la distancia en kilómetros entre esos pueblos extremos.

Sabiendo que nunca es posible lograr una red que use menos cable que la distancia máxima entre pueblos, solo nos falta convencernos de que sí se puede armar una red con esa cantidad, y entonces sabremos que esa será la respuesta: será la mínima cantidad posible de cable a utilizar, para lograr el objetivo de conectar todos los pueblos a la megacentral.

La forma de lograr el objetivo es realizar todas las conexiones en forma “lineal”, o en “serie”, exactamente como en el ejemplo: Unimos el pueblito con mínimo número de kilómetro, con el que tenga el segundo menor número de kilómetro, luego con el tercero, y así siguiendo hasta llegar al otro pueblito extremo con máximo número de kilómetro. En el medio de este recorrido aparecerá la central, pero eso no es

problema, ya que como en el ejemplo, al pasar por ella también usamos un tramo de cable de exactamente la misma forma que si fuera otro pueblito más.

Por todo lo anterior entonces, la respuesta al problema es la distancia entre el pueblito más a la derecha (el de mayor número de kilómetro) y el de más a la izquierda (el de menor número de kilómetro).

Un caso especial ocurre cuando la megacentral tiene menor número de kilómetro que todos los pueblitos: en ese caso,la central es lo que está más a la izquierda, y la respuesta será la distancia máxima de la central al pueblito de más a la derecha. Ocurre algo similar si la central estuviera lo más a la derecha posible.

En otras palabras: **La respuesta finalsiempre es elmáximo de todos los números, menos el mínimo de todos los números** . Cuando decimos “todos los números”, nos referimos a los números de kilómetro de todos los *N* pueblos, y también al número de kilómetro de la megacentral.

Un breve pseudocódigo de solución para este problema sería por ejemplo algo como lo siguiente:

Algorithm 5 Solución al Problema 1 Nivel 3 Jurisdiccional 2017

```
1:  procedure Problema1Nivel3Jurisdiccional2017
2:      N ← leerEntero() // Cantidad de pueblitos
3:      ubicacionM egacentral ← leerEntero()
4:      posicionM axima ← ubicacionM egacentral
5:      posicionM inima ← ubicacionM egacentral
6:      for i ← 1 . . . N do
7:          kilometroP ueblo ← leerEntero()
8:          if kilometroPueblo >posicionMaxima then
9:              posicionM axima ← kilometroP ueblo
10:         if kilometroPueblo <posicionMinima then
11:             posicionM inima ← kilometroP ueblo
12:     imprimir(posicionMaxima - posicionMinima)
```

Un ejemplo de programa completo en C++ para este problema, que corresponde textualmente al pseudocódigo anterior, sería el siguiente:

```
#include <iostream>

using namespace std;

int main() {
    int N; cin >> N;
```

```
    int ubicacionMegacentral; cin >> ubicacionMegacentral;
    int posicionMaxima = ubicacionMegacentral;
    int posicionMinima = ubicacionMegacentral;
    for (int i=1;i<=N;i++) {
        int kilometroPueblo; cin >> kilometroPueblo;
        if (kilometroPueblo > posicionMaxima)
            posicionMaxima = kilometroPueblo;
        if (kilometroPueblo < posicionMinima)
            posicionMinima = kilometroPueblo;
    }
    cout << posicionMaxima - posicionMinima << endl;
    return 0;
}
```

3.2.2. Problema 2: Elnúmero de Erdos-Darwin [erdosdarwin]

http://juez.oia.unsam.edu.ar/#/task/erdosdarwin/statement

En este problema, tenemos una red de colaboraciones y un cierto entero **d**, y queremos contar cuántos investigadores tienen número de Erdos-Darwin menor o igual que **d**.

La red de colaboraciones viene dada como un listado de relaciones de colaboración en papers científicos. Es decir, tenemos una lista de colaboraciones de tipo (a, b) que indican que a y b escribieron juntos algún paper. ¿A qué nos recuerda esto? ¡A la lista de adyacencia de un grafo! (Ver Representaciones de grafos en la wiki : http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos)

Podemos modelar este problema como un grafo, donde tenemos N nodos (uno para cada investigador), y M aristas (una para cada colaboración). Además sabemos que el investigador número 1 es Erdos, así que correspondientemente nuestro nodo número 1 representa a Erdos. Análogamente, el nodo numero N es Darwin.

¿Qué es el número de Erdos-Darwin?
Definimos el número de Erdos como la cantidad de colaboraciones que hay que pasar para llegar desde una persona dada hasta Erdos. Análogamente se define el número de Darwin. El número de Erdos-Darwin de una persona, es igual a la suma de su número de Erdos, y su número de Darwin.

¿Qué representan estos números en el grafo?
Utilizando nuestro modelado, el número de Erdos es igual a la mínima cantidad de

aristas que hay que pasar para llegar desde una persona hasta Erdos.Es decir, la **distancia** en el grafo desde la persona hacia Erdos. Notar además que la distancia desde una persona a Erdos es igual a la distancia desde Erdos a esa persona, ya que el **grafo es no dirigido** . Luego, el número de Erdos-Darwin es igual a la distancia desde Erdos a la persona, más la distancia desde Darwin.

Algorithm 6 BFS / Cálculo de distancias en grafo no dirigido

```
1:  function BFS(ListaDeAdj adj, Número origen)
2:      n ← adj.tamano()
3:      dist ← Numero[n] {INF .. INF }
4:      dist[origen] ← 0
5:      cola ← NuevaCola()
6:      cola.encolar(origen)
7:      while !cola.vacia() do
8:          nodo ← cola.tope()
9:          cola.desencolar()
10:         for vecino in adj[nodo] do
11:             if dist[vecino] >dist[nodo] + 1 then
12:                 dist[vecino] ← dist[nodo] + 1
13:                 cola.encolar(vecino)
14:     return dist
```

¿Cómo calculamos la distancia desde un nodo a otro? En un grafo no dirigido, la distancia desde un origen fijo a todos los nodos la podemos calcular utilizando busqueda a lo ancho (BFS en inglés).

Algorithm 7 Solución al Problema 2 Nivel 3 Jurisdiccional 2017 / Erdos-Darwin

```
1:  procedure ErdosDarwin
2:      n ← leerEntero()
3:      m ← leerEntero()
4:      d ← leerEntero()
5:      adj ← Vector[N] {NuevoVector().. NuevoVector() }
6:      for i = 1 .. M do
7:          a ← leerEntero()
8:          b ← leerEntero()
9:          adj[a].insertar(b)
10:         adj[b].insertar(a)
11:     distErdos ← BFS(adj, 1)
12:     distDarwin ← BFS(adj, n)
13:     res ← 0
14:     for i = 1 .. N do
15:         if distErdos[i] + distDarwin[i] ≤ d then
16:             res ← res + 1
17:     imprimir(res)
```

El algoritmo de BFS tiene complejidad $O(N + M)$. Como podemos ver en el pseudocódigo de más arriba,la clave para terminar el problema es que vamos a correr el BFS desde dos orígenes, Darwin y Erdos. Con las distancias que retornan, calculamos cuántos tienen número de Erdos-Darwin menor o igual que d.

Así, la complejidad nos queda $O(N + M)$, por haber realizado los dos BFS.

3.2.3. Problema 3: Al-Garín [algarin]

http://juez.oia.unsam.edu.ar/#/task/algarin/statement

En este problema tenemos que buscar la mayor cantidad de joyas que Al-Garin puede recolectar.
Recibimos como entrada una grilla de **m** × **n** que representaremos con una matriz del mismo tamaño.
Al-Garín puede moverse de a una casilla, de arriba hacia abajo, y de izquierda a derecha,incluyendo moverse en diagonal.Debe terminar en alguna posición de la última columna.
En la matriz tendremos cuatro tipos de casillas:

- 'M' con Malhechor
- 'A' la casilla inicial de Al-Garín.
- 'J' una casilla con una joya
- '.' una casilla libre

El ejercicio nos recuerda al problema de "camino en la matriz" y a otros similares que utilizan la técnica de **programación dinámica** .Veamos cómo aplicarla en el problema actual.

Vamos a calcular un valor **mejorJoyas** para cada posición de la matriz. Definamos mejorJoyas en palabras.
mejorJoyas (*i, j*) := La mayor cantidad de joyas que se pueden obtener usando un camino que termina la fila *i* y la columna *j*, es decir (*i, j*)

Notar que podemos calcular mejorJoyas, podemos resolver el problema inicial, ya que podemos tomar el máximo sobre todas las posiciones de la última columna.

Además, podemos obtener una definición recursiva de mejorJoyas.La mayor cantidad de joyas se pueden obtener terminando en una posición,está relacionada con la cantidad de joyas que se pueden obtener terminando en las posiciones

anteriores. Para calcular una posición (*i, j*), vamos a utilizar las posiciones de la grilla (*i* − 1, *j*), (*i, j* − 1) y (*i* − 1, *j* − 1).

$$mejorJoyas(i, j) = \begin{cases} -\infty, & i = 0 \\ 0, & M[i][j] = \text{'A'} \\ -\infty, & j = 1 \wedge M[i][j] \; 6 = \text{'A'} \\ -\infty, & M[i][j] = \text{'M'} \wedge joyasAnteriores = 0 \\ 0, & M[i][j] = \text{'M'} \wedge joyasAnteriores > 0 \\ joyasAnteriores(i, j) + joyasEn(i, j), & M[i][j] = \text{'J'} \vee M[i][j] = \text{'.'} \end{cases}$$

$$joyasAnteriores(i, j) = max \begin{cases} mejorJoyas(i - 1, j), \\ mejorJoyas(i, j - 1), \\ mejorJoyas(i - 1, j - 1) \end{cases}$$

$$joyasEn(i, j) = \begin{cases} 1, & M[i][j] = \text{'J}^{\,0}\text{'} \\ 0, & M[i][j] = \text{'.'} \end{cases}$$

En lo anterior, **joyasEn** (*i, j*) indica si una posición tiene joyas o no. Además, **joyasAnteriores** (*i, j*) será la mayor cantidad de joyas con las que se puede llegar a esa posición (notar que en los casos con *i* = 1 la recursión se va por fuera del mapa, pues se hace un llamado con *i* = 0).

Luego si una posición tiene una joya o un espacio vacío,la solución será igual a la suma entre **joyasEn** (*i, j*) y **joyasAnteriores** (*i, j*).

Si en la casilla hay una M, debemos perder todas nuestras joyas, lo cual significa que devolvemos 0. Más aún, si no teníamos joyas entonces Al-Garín no logra escapar, lo cual para nosotros quiere decir que la función retorna menos infinito.

Algorithm 8 MejorJoyas

```
1: function mejorJoyas(i, j, M)
2:     if dp[i][j] 6= -∞ then
3:         return dp[i][j]
4:     if i == 0 then
5:         return -∞
6:     if M[i][j] == 'A' then
7:         return 0
8:     if j == 1 ∧ M[i][j] 6= 'A' then
9:         return -∞
10:    joyasAnteriores ← -∞
11:    joyasAnteriores ← max(joyasAnteriores,
12: mejorJoyas(i - 1, j, M))
13:    joyasAnteriores ← max(joyasAnteriores,
14: mejorJoyas(i, j - 1, M))
15:    joyasAnteriores ← max(joyasAnteriores,
16: mejorJoyas(i - 1, j - 1, M))
17:    joyasEnPosicion ← 0
18:    if M[i][j] == '.' then
19:        joyasEnPosicion ← 1
20:    if M[i][j] == 'M' ∧ joyasAnteriores >0 then
21:        dp[i][j] ← 0
22:    if M[i][j] == 'M' ∧ joyasAnteriores == 0 then
23:        dp[i][j] ← -∞
24:    if M[i][j] == '.' ∨ M[i][j] == 'J' then
25:        dp[i][j] ← joyasAnteriores + joyasEnPosicion
26:    return dp[i][j]
```

Algorithm 9 Solución al Problema 3 Nivel 3 Jurisdiccional 2017 / Al-Garin

```
1: procedure Al-Garin
2:     m ← leerEntero()
3:     n ← leerEntero()
4:     grilla ← Char[m][n]
5:     for i = 1 .. m do
6:         for j = 1 .. n do
7:             grilla[i][j] ← leerCaracter()
8:     for i = 1 .. m do
9:         for j = 1 .. n do
10:            dp[i][j] ← -∞
11:    res ← -∞
12:    for i = 1 .. m do
13:        res ← max(res, mejorJoyas(i, n, casillas))
14:    imprimir(res)
```

¿Cuál es la complejidad de esta solución?
Utilizando la fórmula **# subproblemas** × **costoSubproblema** obtenemos una complejidad de $O(mn) \cdot O(1) = O(mn)$. Tenemos $O(mn)$ subproblemas, uno por posición en la grilla (así como casillas de la matriz dp). Cada subproblema se resuelve en $O(1)$ ya que no contamos la recursión. Cada llamada **mejorJoyas** solo hace recursión y operaciones de tiempo constante (comparaciones y asignaciones).

3.2.4. Problema 4: Tanques de Agua [tanque3]

http://juez.oia.unsam.edu.ar/#/task/tanque3/statement

En este problema tenemos una estructura de tanques conectados por cañerías. Cada tanque, salvo el principal, tiene un caño de entrada que viene de otro tanque y puede tener uno o más caños de salida que lo conectan con otros tanques. A esta estructura la podemos modelar como un árbol donde cada nodo es un tanque y los hijos de un nodo son los tanques a los que está conectado por caños de salida.De esta manera la raíz del árbol será el tanque principal.

Esta estructura de tanques se va llenando con agua. El problema nos pide que dadas distintas cantidades de agua que se quieren almacenar en la estructura, avergiuar para cada una cuántos tanques quedan con agua.Primero veamos cómo podemos descubrir, dada una estructura, cómo se llenan los distintos tanques.

El llenado sigue las siguientes reglas:

- Primero se empieza a llenar el tanque principal.
- Cuando se está llenando un tanque, este se llena hasta que el nivel de agua llegue a una boca de salida o al tope del tanque.
- En el primer caso, se deja de llenar el tanque actual y se empieza a llenar el tanque hijo que sale de esa boca.
- En el segundo caso, se sigue llenando el tanque padre del actual (del que llega la boca de entrada).

Notemos que se puede simular este proceso por medio de un *df s* (búsqueda en profundidad) sobre el árbol de tanques, siempre que recorramos los hijos de un tanque en orden de las bocas de salida de abajo hacia arriba. Hay que tener cuidado de no llenar los tanques de a una unidad porque esto es muy lento.

Un posible algoritmo para simular el proceso es el que presentamos a continuación. En este algoritmo *tanqueActual* representa el número de tanque que

estamos llenando, *nivelT anques* es un vector que guarda cuanta agua tiene cada tanque en cada momento y *arbolT anques* guarda para cada tanque un vector con pares (*indice, altura*) que representan el índice de cada tanque hijo y la altura de la boca de salida que lleva a él, ordenados de menor a mayor altura.

Algorithm 10 Pseudocódigo de la simulación de llenado

```
1: procedure simularLlenado(tanqueActual, nivelTanques, arbolTanques)
2:     for hijo ← arbolT anques [tanqueActual] do
3:         nivelT anques[tanqueActual] ← hijo.altura . Lleno hasta la siguiente
   boca
4:         simularLlenado(hijo.indice, nivelT anques, arbolT anques)
5:     nivelT anques[tanqueActual] ← 10000 . Lleno hasta el tope
```

Notar que si tenemos una cantidad de agua limitada, en cada línea que agregamos un tanque,deberíamos chequear que queda suficiente agua y actualizar cuánta agua queda después de llenar. Como la solución final no usará esto, omitimos los detalles de cómo se implementaría.
Una primera idea sería simular el llenado de tanques para cada consulta guardando en un vector cuánta agua tiene cada tanque en cada momento. Esta idea resuelve el problema pero no es lo suficientemente eficiente para lograr el puntaje completo, ya que el costo temporal de la simulación es la de el *df s* que es $O(T)$ donde T es la cantidad de tanques y hay Q consultas por lo que el tiempo total sería $O(T\,Q)$.

Para lograr mejorar la solución, podemos tratar de obtener toda la información necesaria para responder las consultas con una sola simulación de llenado. Por ejemplo, basta recordar para cada tanque cuántas unidades de agua se tuvieron que usar hasta que dejó de estar vacío. Este vector lo obtenemos en $O(T)$ que es lo que cuesta la simulación. Ahora debemos agregar a la simulación, una variable que vaya guardando cuánta agua usamos para poder calcular la primera vez que agregamos agua a un tanque cuánta agua en total necesitamos.

Una vez calculado esto, para cada consulta basta calcular cuantos números del vector son menores o iguales que la cantidad de agua de la consulta. Para hacer esto eficientemente podemos primero ordenar el vector en $O(T \lg T)$ y luego para cada consulta hacer una búsqueda binaria sobre el vector para averiguar cuántas posiciones son menores o iguales que la cantidad de agua de la consulta.Con esto respondemos cada consulta en $O(\lg T)$ y por lo tanto el tiempo de ejecución del algoritmo termina siendo $O((T + V) \lg T)$.

Capítulo 4

Certamen Nacional

4.1. Nivel 1

4.1.1. Problema 1: Seleccionando al mejor proveedor [fabricante]

http://juez.oia.unsam.edu.ar/#/task/fabricante/statement

Vamos a revisar uno por uno todos los fabricantes y analizar cuál sería la ganancia que Camila obtendría si elige como proveedor a cada uno de ellos.Como son a lo sumo 1000 fabricantes distintos, lo podemos hacer en un ciclo rápidamente.

Por simplicidad, enumeremos los fabricantes de 1 a F y supongamos que estamos analizando el fabricante número i. Este fabricante requiere que le compremos al menos $cantidadCompra_i$ unidades y nos las vende a precio $precioCompra_i$. Lo primero importante a notar es que debemos venderle al comprador todas las unidades que pide, sí o sí (es decir $cantidadVenta$ unidades). Por lo tanto, la cantidad de unidades del producto que debemos comprar es $max(cantidadVenta, cantidadCompra_i)$. Como venderemos cada una a $precioVenta$, la ganancia **por unidad** es de $precioVenta - precioCompra_i$, lo que nos da una ganancia total de:

$$max(cantidadVenta, cantidadCompra_i) \cdot (precioVenta - precioCompra_i)$$

¡Pero cuidado! Esta ganancia es factible solamente si tenemos el dinero para comprarle al fabricante al principio del día. Es decir, si $max(cantidadVenta, cantidadCompra_i) \cdot precioCompra_i \leq P$.

Además, notemos que si el precio de venta es menor al precio de compra, nuestra ganancia será negativa y por ende Camila resignará el negocio. En este caso, debemos devolver -1.

Veamos cómo quedaría un pseudocódigo para esta solución:

Algorithm 11 Solución al Problema 1 Nivel 1 Nacional 2017

```
1:  procedure Problema1Nivel1Nacional2017(P, precioVenta, cantVenta,
    precioCompra, cantCompra, fabricante)
2:      maximaGanancia ← −1
3:      mejorF abricante ← −1
4:      for i ← 1 . . . F do
5:          if m´ax(cantV enta, cantCompra[i]) · precioCompra[i] ≤ P then
6:              gananciaActual ← m´ax(cantV enta, cantCompra[i]) · (precioV enta −
    precioCompra[i])
7:              if maximaGanancia < gananciaActual then
8:                  maximaGanancia ← gananciaActual
9:                  mejorF abricante ← i
10:     return maximaGanancia, mejorFabricante
```

4.1.2. Problema 2: Reconstruyendo el caminito [caminito]

http://juez.oia.unsam.edu.ar/#/task/caminito/statement

En el problema se nos presenta un camino formado por baldosas. Originalmente en el camino solo había baldosas de tres colores posibles: blanco (‘B’), gris (‘G’) y negro (‘N’). Además, el camino cumple con la propiedad de que *baldosas contiguas tienen distinto color*.

Con el paso del tiempo, algunas baldosas del camino se perdieron o fueron removidas (‘R’), y no sabemos qué color tenían originalmente. Nuestra tarea es ubicar baldosas de alguno de los 3 colores en cada lugar donde actualmente falta una baldosa, teniendo el cuidado de que se siga cumpliendo que las baldosas que comparten un lado tienen distinto color.

Por ejemplo, originalmente el camino podría haber tenido las siguiente disposición de baldosas (notar que las baldosas vecinas tienen distinto color):

Si fueron removidas las baldosas en las posiciones 1,2,4 y 7,el caminito que nosotros vemos (y que viene en la entrada es).

Si bien el enunciado deja explícitamente claro que existe una forma de ubicar las baldosas cumpliendo la regla mencionada (pues el camino originalmente tenía todas las baldosas puestas y cumplía con la regla, así que una forma válida de ubicar las baldosas restantes es volver a la original), uno puede ver que efectivamente siempre hay solución.

Para ver esto, vamos a tener que notar una observación clave, que nos permitirá resolver el problema. **¿Cuántos vecinos tiene una baldosa?**

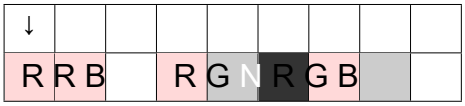
Una baldosa tiene a lo sumo 2 vecinos (notar que las baldosas en los extremos solo tienen 1 vecino). Por lo tanto, **cada baldosa puede tener a lo sumo 2 colores distintos entre sus baldosas vecinas** (pues cada vecino aporta a lo sumo un color distinto).

Tenemos a disposición 3 letras distintas ('B','G', 'N') para ubicar en cada lugar donde falta una baldosa. Esto quiere decir que para toda baldosa ***siempre tenemos a disposición algún color que no está entre sus vecinos***, sin importar qué colores tengan esos vecinos.

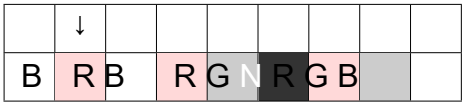
De aquí surge una forma clara de hallar la solución al problema.

- Recorrer todas las baldosas
- Si una *baldosa* tiene una 'R':
  - Para cada *color* en *{*B*,* G*,* N*}*:
    - Si ninguna baldosa vecina es del mismo color que el *color* elegido, entonces pintamos a la *baldosa* de este *color* (notar que por lo que vimos, siempre hay al menos un color del que vamos a poder pintar).

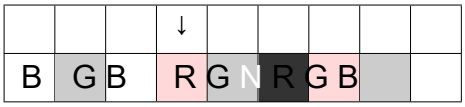
En el ejemplo anterior, se vería de la siguiente forma:

1. 

Utilizamos cualquiera de los 3 colores.

2. 

No podemos utilizar blanco

3. 

Solo podemos utilizar negro

4. 

Solo podemos utilizar blanco

5.  Solución final

Si bien el caminito reconstruido resultó igual al original, esto no necesariamente ocurre (pensar por ejemplo cómo hubiera seguido la secuencia si en el paso 1 pintábamos de negro o gris en vez de blanco).

4.1.3. Problema 3: Cambiando las reglas del dictado [dictado]

http://juez.oia.unsam.edu.ar/#/task/dictado/statement

El problema nos pide que dada una palabra *P* que nos dictó la maestra agreguemos la menor cantidad de letras a la derecha para formar un palíndromo.

Para hacer esto iremos probando incrementalmente la cantidad de caracteres que debemos agregar a *P* para formar un palíndromo. Primero probaremos sin agregar nada, luego agregando un caracter, luego dos, y así siguiendo. Sabemos que como mucho necesitaremos agregar $longitud(P) - 1$ elementos a nuestra palabra. Veámoslo con un ejemplo:

Si *P* = *a b c d*, el menor palíndromo a formar es

a b c d **c b a**

Como iremos probando incrementalmente,ni bien encontremos una cantidad de caracteres tal que agregándolos a *P* formamos un palíndromo detendremos nuestra búsqueda.

Ahora bien, ¿cómo sabemos sipodemos podemos agregar *i* caracteres a la derecha de nuestra palabra *P* y formar un palíndromo? La forma más directa de hacer esto es intentando construir un palíndromo con las restricciones pedidas. Como sabemos que elprimer y último caracter de un palíndromo son iguales, podemos deducir qué caracter ocupará el lugar de más a la derecha. De la misma forma, el segundo y el anteúltimo lugar deberán ser ocupados por el mismo caracter, y así sucesivamente.Siguiendo con el ejemplo anterior, nuestro algoritmo debería ir completando de la siguiente forma (suponiendo que ya deduje que necesito al menos 3 caracteres más para agregar):

a b c d ? ? ?

a b c d ? ? **a**

a b c d ? **b a**

a b c d **c b a**

Si en algún momento notamos que no podemos hacer que la palabra sea palindrómica con las restricciones impuestas, sabremos que es porque necesitamos al menos un caracter más. Continuemos con el mismo ejemplo pero esta vez pidiendo que se agregue solo 2 caracteres a la palabra:

a b c d ? ?

a b c d ? **a**

a b c d **b a**

a b *c d* **b a**

Al llegar a analizar *c* con *d* llegamos a un absurdo, porque no podemos cambiar ninguna de las letras y las mismas son diferentes entre sí.

Veamos cómo sería un pseudocódigo para el algoritmo recién descripto.

Algorithm 12 Solución al Problema 3 Nivel 1 Nacional 2017

```
1: procedure Problema3Nivel1Nacional2017(P)
2:     largo ← longitud(P )
3:     while ¬palindromoP osible (P, largo) do
4:         largo ← largo + 1
5:     return largo
```

Nota: "¬"significa negación. Si *palindromoP osible*(*P, largo*) era verdadero, ¬*palindromoP osible*(*P, largo*) será falso y viceversa.

Algorithm 13 Verificar si es posible construir un palíndromo de cierta longitud agregando letras a derecha

```
1: procedure palindromoPosible(P, largo)
2:     posible ← V erdadero
3:     for i ← 1 . . . blargo/2c do
4:         if longitud(P ) − i ≤ longitud(P ) then
5:             charDerecha ← P [longitud(P ) − i]
6:         else
7:             charDerecha ← P [i]
8:         if P [i] 6= charDerecha then
9:             posible ← F also
10:            break
11:    return posible
```

4.2. Nivel 2

4.2.1. Problema 1: Buscando la mayor ganancia [ganancia]

http://juez.oia.unsam.edu.ar/#/task/ganancia/statement

El enunciado nos habla de dos tipos de entidades:

- Compradores, con un precio a pagar por unidad; y una cantidad de unidades que compra
- Fabricantes, con un precio al que vende por unidad, y una cantidad de unidades mínimas de fabricación

Queremos encontrar el par fabricante y comprador que maximicen la ganancia, siempre y cuando la compra al fabricante no supere nuestro presupuesto P.

Enfoque cuadrático:

Lo primero que notamos es que buscamos un par (c, f) de un comprador y un fabricante respectivamente. Entonces una opción fácil de programar es iterar sobre todos los posibles compradores, y dentro de esa iteración, por todos los posibles fabircantes. De esta manera probamos todos los posibles pares (c, f),chequeamos que se cumplan las condiciones y elegimos aquel que maximice la ganancia.

Para calcular la ganancia de un par, utilizamos una función como esta:

Algorithm 14 Ganancia de un par

```
1: procedure gananciaPar(f, c)
2:     if f.cantidad >= c.cantidad then
3:                          . La diferencia entre lo que nos pagaron y lo que sale producir
4:         return c.precio * c.cantidad - f.precio * f.cantidad;
5:     else
6:                                        . Aca solo nos importa la diferencia de precio
7:                                                  . Vendemos todo lo que producimos
8:         return c.cantidad * (c.precio - f.precio);
```

Enfoque eficiente:

Como vimos en el enfoque anterior, para realizar el cómputo de gananciaPar, tenemos que tener en cuenta los dos casos:

- El comprador quiere una cantidad menor o igual a la fabricada
- El comprador quiere una cantidad mayor a la mínima producida por el fabricante.

Algorithm 15 Ganancia cuadratica

```
1:  procedure ganancia(P, fabricantes, compradores, Fab, Comp )
2:      res ← -1
3:      Fab ← 0
4:      Comp ← 0
5:      for i = 0 .. |fabricantes| do
6:          for j = 0 .. |compradores| do
7:              f ← fabricantes[i]
8:              c ← compradores[j]
9:              produccion ← max(c.cantidad, f.cantidad)
10:             if f.precio * produccion ≤ P then          . Si el precio de producir es
    menor que P
11:                 ganancia ← gananciaPar(f, c)
12:                 if ganancia >res then
13:                     res ← ganancia
14:                     Fab ← i
15:                     Comp ← j
16:     return res
```

Queremos bajar la complejidad temporal de nuestra solución. Queremos buscar algo mejor que cuadrático (mejor que $F \cdot C$, probar todos los posibles pares). Para esto, vamos a buscar a través de todos los compradores. Una vez fijo un comprador, queremos el fabricante que mayor ganancia nos da. Para eso podemos pensar en dos posibles fabricantes:

- Aquel que maximiza el caso 1
- Aquel que maximiza el caso 2

Supongamos que fijamos un comprador *c*. Analizamos la cuenta en ganaciaPar y vemos que el fabricante que maximiza el caso 1 es el que tiene menor valor de precio * cantidad entre aquellos fabricantes que tienen una producción mayor o igual a la requerida (o sea que efectivamente pertenecen al caso 1).

Análogamente, el fabricante que maximiza el caso 2 es el que tiene menor precio entre aquellos que tienen una producción menor a la requerida.

Si pensamos a los fabricantes y compradores como una recta ordenada por cantidad (comprada o mínimo de fabricación), vemos que cada comprador busca a su derecha (caso 1) y a su izquierda (caso 2). En cada consulta, a izquierda o derecha, solo buscamos el mínimo de un cierto valor, lo cual se puede computar eficientemente. Algo que podemos aprovechar para el cómputo es que solo buscamos mínimos en prefijos y sufijos.

Algorithm 16 Ganancia eficiente

```
1: procedure ganancia(P, fabricantes, compradores, Fab, Comp )
2:     res ← -1
3:     Fab ← 0
4:     Comp ← 0
5:                          . Llamo participantes a todos los fabricantes y compradores
6:     ps ← ListaVacia()                                   . Lista de participantes
7:     for i = 0 .. |fabricantes| do
8:         f ← fabricantes[i]
9:                 . Como el problema requiere devolver el indice de los fabricantes y
   compradores elegidos, nos guardamos esto
10:        ps.agregar({f.cantidad, true, f.precio, i + 1});
11:    for i = 0 .. |compradores| do
12:        c ← compradores[i]
13:        ps.agregar({c.cantidad, true, c.precio, i + 1});
14:    Ordenar(ps)                    . Es importante ordenar primero por cantidad
15:                                                    . minimo precio e indice
16:                                  . Se va a ir actualizando de izquierda a derecha
17:    minPrecio ← INF, -1
18:    for p en ps do
19:        if p.esFabricante then
20:            f ← p
21:            if minPrecio.first >f.precio then
22:                minPrecio ← {f.precio, f.indice}
23:        else
24:            c ← p
25:                                                  . Si no me quedo sin presupuesto
26:            if c.cantidad * minPrecio.first ≤ P then
27:                                                  . es como desglosar gananciaPar
28:                ganancia ← c.cantidad * (c.precio - minPrecio.first);
29:                if ganancia >res then res ← ganancia Comp ← c.indice Fab ←
   minPrecio.second
30:    minCosto ← INF, -1
31:      . Ahora invertimos la lista, para buscar “desde la derecha”, o mejor dicho,
   desde mayor cantidad
32:    InvertirLista(ps)
33:    for p en ps do
34:        if p.esFabricante then
35:            f ← p
36:            costo ← f.precio * f.cantidad
37:            if costo ≤ P then
38:                if minCosto.first >costo then
39:                    minCosto ← {costo, f.indice}
40:        else
41:            c ← p
42:            ganancia ← c.cantidad * c.precio - minCosto.first
43:            if ganancia >res then res ← ganancia Comp ← c.indice Fab ←
   minCosto.second
44:    return res
```

4.2.2. Problema 2: Reconstruyendo el sendero [sendero]

http://juez.oia.unsam.edu.ar/#/task/sendero/statement

Vamos a presentar dos maneras de resolver este problema.

Un primer pensamiento intuitivo es ubicar siempre la baldosa más barata que se pueda. Veamos primero un ejemplo en el cual esa idea así como se lee, falla. Supongamos que los precios de *Blanco*, *Gris* y *Negro* son 1, 2 y 100 respectivamente.

1	2	3	4	5
N	R	R	R	B

Si seguimos con esa línea de pensamiento, querríamos poner primero una 'B' en la posición 2 y luego una 'G' en la posición 3. De hacer esto, nos vemos forzados a poner una 'N' en la posición 4, obteniendo un costo total de 103. Sin embargo, existen mejores formas de llenarlo. Por ejemplo, podemos obtener un costo de 5 de la siguiente forma.

1	2	3	4	5
N	G	B	G	B

Si bien esta primera idea falló, veamos cómo podemos ajustarla para obtener una solución óptima. Una primera observación es que cada bloque de baldosas removidas ('R') lo podemos tratar por separado, ya que si dos bloques no son consecutivos, las baldosas de uno no impondrán ninguna restricción para completar las del otro bloque.

Analicemos entonces el problema de resolver un solo bloque. Completar todo el bloque con la baldosa más barata es obviamente óptimo en cuanto a costos, pero no podremos hacerlo a menos que el bloque sea de una baldosa (pues si hay más de una baldosa en el bloque tendremos baldosas vecinas del mismo color). Por lo tanto, se resume a utilizar la baldosa más barata que sea factible (como hay 3 colores y cada baldosa tiene 2 vecinas, sabremos que existirá al menos un color disponible).

Supongamos entonces que tenemos bloques con más de una baldosa. Llamemos X, Y, Z a los colores ordenados por precio (siendo X el más barato, Y el siguiente y Z el más caro). Nuevamente, de ser posible queremos completar el bloque o bien

X	Y	X	Y	. . .

o bien

Y	X	Y	X	. . .

, será óptimo en cuanto a costos, pero no necesariamente cumplirá que no haya vecindades del mismo color. Si la longitud del bloque es par nos da lo mismo en cuanto a costos, y si la longitud es

impar, resultará mejor empezar con la más más barata ya que deberemos comprar una más de la primera que coloquemos.

Ahora, en los casos que no sea factible llenarlo de alguna de estas formas, nos implica que al menos vamos a tener que utilizar la baldosa más cara una vez. ¿Cuántas veces será necesario utilizar una de las baldosas más caras?

Como el bloque tiene largo mayor o igual a 2, para completar la primera baldosa siempre tendremos disponible alguna de X o Y.Luego podemos ir alternando entre las dos más baratas a excepción de la última baldosa. Al llegar al final tendremos dos vecinos que ya están pintados (mientras llenábamos el resto solo teníamos el vecino izquierdo pintado). Por lo tanto podríamos vernos forzados a colocar la más cara. ¿Cuántas veces utilizamos la baldosa más cara? ¡Una sola!.

Basándonos en esta idea podemos llegar a una solución, que consiste en mirar cada bloque por separado,ver si podemos completarlo con las dos más baratas,y si no, ir completando de izquierda a derecha o de derecha a izquierda con la más barata posible.

La otra solución que proponemos no se basa en analizar casos particulares del problema, ni en un algoritmo goloso que busca hacer lo mejor en cada momento, sino que consiste en analizar todas las soluciones posibles (sin explícitamente crearlas todas).

Analicemos lo siguiente, ¿de cuántas formas distintas se pueden llenar las baldosas removidas? Olvidándonos de las vecindades en los extremos del bloque, tendríamos 3 posibilidades para la primera baldosa, 2 posiblidades para la segunda (la primera está fijada y no puede compartir su color). Para la tercera también tendremos 2 posibilidades. Análogamente hasta llegar al final, nos da aproximadamente 2^n posibilidades, donde n es la cantidad de baldosas del bloque.

Claramente no podemos probar todas esas posibilidades. Veamos qué podemos hacer para resolver el problema.

Pensemos en buscar la mejor solución *asumiendo que ya tenemos todas las primeras i baldosas colocadas*,y estas fueron colocadas para minimizar el costo de la colocación de esas i baldosas.Entonces asumiendo esto,vamos a querer colocar la baldosa $i + 1$. Al colocar esta nueva baldosa, nos importa qué baldosa colocamos como $i - esima$, ya que eso nos limitará a la hora de colocar la baldosa $i + 1$. Pero no importan las baldosas anteriores a la $i - esima$, ya que no nos restringen en nada.

Concretamente vamos a guardar **elmenor costo necesario para colocar**

las primeras *i* baldosas, terminando con una baldosa de color *x* para *x* en *{*B, G, N*}*.

Supongamos que ya conocemos estos valores hasta la baldosa *i* ¿cómo podemos ubicar una nueva baldosa en el lugar *i*+1 de manera óptima? Sabemos que al colocar la baldosa *i* + 1 solo nos importan las baldosas consecutivas, que son la *i* y la *i* + 2. Como estamos yendo de izquierda a derecha, a menos que la *i* + 2 no haya sido removida, por ahora seguirá removida y no nos impondrá restricciones. Entonces sólo nos importa la baldosa *i* a la hora de poner la *i* + 1. Y esto es lo que reduce tanto el tiempo de nuestro algoritmo y las posibilidades.

Entonces, si los colores son *x, y, z* pensemos cómo responder ¿cuál es el mejor costo de que puedo obtener en las primeras *i* + 1 posiciones, si en la posición *i* + 1 coloco una baldosa de color *x*?. Para este caso sabemos que en la baldosa *i* no podía haber una baldosa de color *x*, pero podrá ser de color *y* o *z*. De estas dos posibilidades buscamos aquella que nos de el menor costo (que ya tenemos porque tenemos el menor costo de colocar las primeras *i* baldosas terminando en *x*, *y* o *z*), y utilizamos esa opción.

Utilizando *programación dinámica* para no volver a resolver dos veces un mismo subproblema de la forma (*i, x*), podemos dar un algoritmo eficiente que resuelve el problema. El único cuidado que hay que tener en esta solución es qué pasa cuando llegamos a una baldosa que no fue removida. Supongamos en el ejemplo anterior que la baldosa *i* era de color *x* y no fue removida. En ese caso, lo que queremos es no mirar ninguna posibilidad que coloque en *i* una baldosa *y* o *z*. Para eso, una idea común es poner como menor costo de colocar las primeras *i* baldosas terminando en una *y* o una *z* como infinito (o algún valor muy grande que sea cota en nuestro problema). La idea es que como siempre hay una solución factible, obtendremos soluciones con valor menor a infinito al colocar las primeras *i* baldosas, y entonces en nuestro algoritmo no la tendremos en cuenta.

Algorithm 17 Solución al Problema 2 Nivel 2 Nacional 2017

```
1: procedure completarBaldosas(S, precioB, precioG, precioN)
2:     for i ← 1 . . . longitud(S) do
3:         for j ← 1 . . . 3 do
4:             menorCosto[i][j] ← 0
5:     for i ← 1 . . . longitud(S) do
6:         if S[i] 6= R then
7:             minHastaAntes ← INF
8:             for anterior ← 1 . . . 3 do
9:                 if indice(S[i]) 6= anterior then
10:                    minHastaAntes ← min(minHastaAntes, menorCosto[i −
    1][anterior])
11:                    menorCosto[i][anterior] ← IN F
12:            menorCosto[i][indice(S[i])] = minHastaAntes
13:            continue
14:        for j ← 1 . . . 3 do
15:            minAnterior ← INF
16:            for anterior ← 1 . . . 3 do
17:                if anterior 6= j then
18:                    minAnterior ← min(minAnterior, menorCosto[i −
    1][anterior])
19:            menorCosto[i][j] ← costo[j] + minAnterior
20:    return min(menorCosto[longitud(S)][0], min(menorCosto[longitud(S)][1],
    menorCosto[longitud(S)][2]))
```

Nota: Inicializar INF en algo muy grande en este caso, y podriamos hacer que en la posicion 0, todos los valores sean 0 como para a partir de ahi ir sumando al colocar una baldosa. La funcion indice es tal que devuelve 1 para un color, 2 para otro, y 3 para el ultimo

4.2.3. Problema 3: Dictado de nivelación [prueba]

http://juez.oia.unsam.edu.ar/#/task/prueba/statement

El problema nos pide que dada una palabra P que nos dictó la maestra agreguemos la menor cantidad de letras a izquierda y/o derecha para formar un palíndromo.

Lo primero que debemos notar es que nunca ocurrirá que debamos agregar letras a izquierda y a derecha al mismo tiempo, ya que si lo hiciéramos ese palíndromo no sería el más corto posible. Veamos por qué.

Supongamos que para cierta palabra P la solución óptima requiere agregar al menos un caracter a la izquierda y otro a la derecha. Sea $P = p_1 p_2 \ldots p_n$ la palabra original, sean $I = i_1 i_2 \ldots i_j$ los caracteres que agregamos a la izquierda en ese orden y sean $D = d_1 d_2 \ldots d_k$ los caracteres que agregamos a la derecha en ese

orden.

La palabra que es solución a nuestro problema es la siguiente, y por definición será un palíndromo ($I\ P\ D$):

$$i_1\ i_2\ \ldots i_j\ p_1\ p_2 \ldots p_n\ d_1\ d_2 \ldots d_k$$

Ahora bien, como es un palíndromo sabemos que $i = d_k$. Si a un palíndromo le quitamos su primer y último caracter el resultado seguirá siendo palíndromo, y en este caso quedaría la siguiente cadena de caracteres:

$$i_2\ \ldots i_j\ p_1\ p_2 \ldots p_n\ d_1\ d_2 \ldots d_{k-1}$$

Este último palíndromo es más corto que el anterior pues tiene dos caracteres menos. Es decir que lo que supusimos inicialmente es imposible, y podemos concluir que la palabra solución se genera agregando caracteres o bien a la izquierda o bien a la derecha (o en ninguno, si la palabra P ya es palíndromo).

¿Pero cómo esto nos ayuda a resolver el problema? ¡Nos facilita mucho, porque ahora tenemos que probar muchas menos posibilidades! Si no nos diéramos cuenta de lo anterior, deberíamos haber probado todas las cadenas que agregan elementos a la izquierda y a la derecha.

Solo nos resta encontrar lo siguiente:

- La cadena más corta D que se puede agregar a la derecha de forma tal que $P\ D$ sea un palíndromo. Este problema es exactamente el explicado en 4.1.3.
- La cadena más corta I que se puede agregar a la izquierda de forma tal que $I\ P$ sea un palíndromo. O equivalentemente, la cadena más corta I tal que *reversoCadena*(P) *reversoCadena*(I) sea palíndromo, que es el problema explicado en 4.1.3.

Como queremos la solución que agregue la menor cantidad de caracteres tomaremos el valor mínimo entre los dos. A continuación veamos el pseudocódigo.

Algorithm 18 Solución al Problema 3 Nivel 2 Nacional 2017

```
1: procedure Problema3Nivel2Nacional2017(P)
2:     agregarADerecha ← P rob3N ivel1(P )
3:     agregarAIzquierda ← P rob3N ivel1(reversoCadena(P ))
4:     return min (agregarADerecha, agregarAIzquierda)
```

Algorithm 19 Revertir una cadena de caracteres

```
1: procedure reversoCadena(P)
2:     for i ← 1 . . . blongitud(P )/2c do
3:         aux ← P [i]
4:         P [i] ← P [longitud(P ) − i]
5:         P [longitud(P ) − i] ← aux  (en P [i] ya no está el valor viejo, ¡lo modificamos!)
6:         return false
7:     return P
```

Nota: un error muy común en la implementación de reversoCadena es realizar el *f or* sobre toda la cadena y no hasta la mitad. Esto es un error porque como invertimos los valores de $P[i]$ y $P[longitud(P)-i]$, si lo hacemos dos veces volveremos a tener la misma cadena que al comienzo.

4.3. Nivel 3

4.3.1. Problema 1: Armando el negocio [compra]

http://juez.oia.unsam.edu.ar/#/task/compra/statement

Veamos que nunca necesitaremos comprarle a más de 1 fabricante.

Si consideramos un comprador cualquiera – que nos va a comprar X productos – entonces los fabricantes que piden un mínimo menor a X nos podrán vender exactamente X, haciéndonos gastar X multiplicado por el precio al que vendan el producto. Si consideramos los fabricantes que piden un mínimo de compra mayor a X, sí o sí les compraremos el mínimo que pidan (ya que si les compramos más nos sobrarían), por lo que gastaremos el precio unitario que cobra el fabricante multiplicado por su cantidad mínima.

Es claro que de todos esos valores (o bien X por el precio de esos fabricantes, o el mínimo por el precio), hay un valor que es el menor. Y ese es el fabricante que querremos seleccionar para el comprador que nos comprará X productos. Podría haber más de un fabricante con ese menor valor pero no es de importancia, ya que si hay excedentes deberemos descartarlos y solo podremos vender las X piezas. Por lo tanto, si cumplimos el requisito de comprar al menos X productos no importa cuántos compramos ni a qué precio: solo importa el precio total.

Una vez hecha esta observación clave,es el mismo problema exacto que en nivel 2 (ver 4.2.1).

4.3.2. Problema 2: Dictado de nivelación [nivelacion]

http://juez.oia.unsam.edu.ar/#/task/nivelacion/statement

Daremos una solución recursiva al problema planteado. Esto quiere decir

que pensaremos a la solución del problema general como la unión de algunos subproblemas equivalentes más pequeñosEscribamos conceptualmente cuál es la función que queremos calcular, que llamaremos f. De aquí en más, llamamos $p_1\ p_2\ \cdots\ p_n$ a los n caracteres que forman la palabra original P.

$$f[i][j] = \text{mínima cantidad de caracteres a agregar para que la cadena formada por los caracteres } p_i\ p_{i+1}\ \cdots\ p_j \text{ se convierta en un palíndromo}.$$

La solución al problema completo será $f[1][n]$, es decir, la mínima cantidad de caracteres a agregar para que la cadena original$P$ se convierta en un palíndromo. Ahora bien, ¿cómo hacemos para calcular $f[i][j]$ para todo $i \leq j$? Una vez hecho esto, lo que restará será simplemente imprimir $f[1][n]$.

Analicemos la subcadena $p_i\ p_{i+1}\ \cdots\ p_{j-1}\ p_j$, viendo cómo según el valor de los caracteres de los bordes exteriores podemos reducir el problema de hallar $f[i][j]$ a uno más pequeño.

Si $p_i = p_j$ entonces los bordes exteriores ya cumplen lo necesario para formar un palíndromo y por ende no necesitamos agregar ningún caracter. Nos resta agregar caracteres para asegurarnos de que $p_{i+1}\ p_{i+2}\ \cdots\ p_{j-2}\ p_{j-1}$ se convierta en un palíndromo. Queremos hacer esto utilizando la menor cantidad posible de caracteres agregados, o sea que deberemos agregar $f[i+1][j-1]$ caracteres.

Si $p_i \neq p_j$ debemos agregar un caracter para que los dos caracteres exteriores sean iguales entre sí.Podemos hacerlo agregando un caracter a la izquierda o a la derecha.

- Si agregamos un caracter a la izquierda.

$\mathbf{p_j}$	p_i	p_{i+1}	...	p_{j-1}	p_j	. . .

Gráficamente, podemos ver que si agregamos un caracter p_j a la izquierda solo resta por hacer palíndroma la subcadena $p_i\ p_{i+1}\ \cdots\ p_{j-2}\ p_{j-1}$. Esto se puede hacer con $f[i][j-1]$ caracteres como mínimo.

- Si agregamos un caracter a la derecha.

. . .	p_i	p_{i+1}	...	p_{j-1}	p_j	$\mathbf{p_i}$

Gráficamente, podemos ver que si agregamos un caracter p_i a la derecha solo resta por hacer palíndroma la subcadena $p_{i+1}\ p_{i+2}\ \cdots\ p_{j-1}\ p_j$. Esto se puede hacer con $f[i+1][j]$ caracteres como mínimo.

En conclusión, si $p_i \neq p_j$ entonces necesitamos agregar como mínimo $min(f[i][j-1], f[i+1][j])$ caracteres para hacer palíndroma la subcadena $p_i\ p_{i+1}\ \ldots\ p_{j-1}\ p_j$.

En resumen, podemos expresar $f[i][j]$ de la siguiente manera:

$$f[i][j] = \begin{cases} f[i+1][j-1] & \text{si } p_i = p_j \\ min(f[i][j-1], f[i+1][j]) & \text{si } p_i \neq p_j \end{cases} \quad (4.1)$$

Yendo a la cuestión implementativa, podemos calcular los $f[i][j]$ guardándolos en una matriz de $n \times n$ e ir completando todos los valores válidos de la matriz (los $f[i][j]$ con $i > j$ no importan porque son inválidos). Para calcular cada $f[i][j]$ necesitamos saber los resultados de f's de intervalos más pequeños, pero podemos observar observar que no es posible continuar esto infinitamente. ¿Qué sucede para $f[i][i]$ y $f[i][i+1]$, en donde la fórmula de 4.1 llevaría a evaluar en casos inválidos?

Como no podemos aplicar la fórmula de 4.1, tenemos que calcular explícitamente f para estos casos. Afortunadamente, estos casos son muy simples:

- $f[i][i]$ representa saber cuántos caracteres hay que agregar para que una cadena de un solo caracter sea palíndroma. Toda cadena de un solo caracter es palíndroma, y por ende $f[i][i] = 0$.

- $f[i][i+1]$ representa saber cuántos caracteres hay que agregar para que la cadena $p_i\ p_{i+1}$ sea palíndroma.

  - Si $p_i = p_{i+1}$, no hace falta agregar nada, y por lo tanto $f[i][i+1] = 0$
  - Si $p_i \neq p_{i+1}$, bastará con agregar un solo caracter. $f[i][i+1] = 1$.

Todo lo discutido anteriormente puede resumirse en el pseudocódigo que sigue a continuación. Para ir rellenando adecuadamente la matriz comenzaremos desde las subcadenas más pequeñas hasta las mayores (k será la longitud de la cadena a analizar).

Algorithm 20 Solución al Problema 2 Nivel 3 Nacional 2017

```
1:  procedure Problema2Nivel3Nacional2017(P)
2:      f ← matriz de longitud(P ) × longitud(P )
3:      for i ← 1 . . . longitud(P ) do
4:          f [i][i] ← 0
5:
6:      for i ← 1 . . . longitud(P ) − 1 do
7:          if p[i] = p[i+1] then
8:              f [i][i + 1] ← 0
9:          else
10:             f [i][i + 1] ← 1
11:
12:     for k ← 2 . . . longitud(P ) − 1 do
13:         for i ← 1 . . . longitud(P ) − k do
14:             j ← i + k
15:             if P [i] = P [j] then
16:                 f [i][j] ← f [i + 1][j − 1]
17:             else
18:                 f [i][j] ← min(f [i][j − 1], f [i + 1][j])
19:
20:     return f [1][longitud(P )]
```

4.3.3. Problema 3: Reconstruyendo la vereda [vereda]

http://juez.oia.unsam.edu.ar/#/task/vereda/statement

Este era probablemente el problema más difícil del Certamen Nacional, aunque por tener un enunciado relativamente fácil de entender, hubo muchos envíos con intentos de soluciones mediante algoritmos golosos y heurísticas incorrectas.

Contamos a continuación las soluciones principales:

4.3.3.1. Solución exponencial mediante backtracking

La solución más directa posible sería probar todas las posibilidades para la asignación, cortando la búsqueda tan pronto como se encuentre solución válida, o se determine que la solución parcial armada hasta el momento ya no puede extenderse hasta una solución válida de todo el problema (podas). A este método general se le denomina backtracking.

Una forma clásica de implementar un algoritmo de backtracking para este problema sería mediante una función recursiva, que vaya asignando los colores de cada baldosa en orden de aparición:

Por ejemplo, notar que gracias al if de la línea 4 de la función pintar, se

Algorithm 21 Solución al Problema 3 Nivel 3 Nacional 2017

```
cad es un String global
cant es un arreglo global de 3 enteros
col es un string global de longitud 3
 1: procedure vereda(B,G,N, baldosas)
 2:     cad ← baldosas
 3:     cant[0] ← B
 4:     cant[1] ← G
 5:     cant[2] ← N
 6:     col[0] ← 'B'
 7:     col[1] ← 'G'
 8:     col[2] ← 'N'
 9:     pintar(0, 'X')
10:     baldosas ← cad
 1: function pintar(i, colorProhibido) : boolean
 2:     if i ≥ longitud(cad) then
 3:         return true
 4:     if cad[i] 6= colorP rohibido then
 5:         if cad[i] 6= 'R' then
 6:             return pintar (i + 1, cad[i])
 7:         else
 8:             for c ← 0 . . . 2 do
 9:                 if col[c] 6= colorP rohibido AND cant[c] > 0 then
10:                     cant[c] ← cant[c] − 1
11:                     cad[i] ← col[c]
12:                     if pintar(i + 1, col[c]) then
13:                         return true
14:                     cad[i] ← 'R'
15:                     cant[c] ← cant[c] + 1
16:     return false
```

retorna inmediatamente, sin hacer una nueva llamada recursiva, cuando acaba de realizarse una pintada inválida. Similarmente, en el if de la línea 9 solamente se consideran los colores viables de utilizar hasta el momento (deben quedar baldosas disponibles de ese color, y no puede coincidir con el color de la baldosa anterior, que siempre se pasa en elparámetro *colorP rohibido*). De este modo, solamente se exploran las ramas parciales "viables" en el proceso de ir armando una solución.

Esta solución de muy corta implementación produce respuestas correctas, pero no obtiene tanto puntaje como las siguientes ya que solo termina en tiempo razonable para casos muy pequeños.

4.3.3.2. Solución $O(n^3)$ utilizando programación dinámica

Este problema puede resolverse en tiempo polinomial utilizando un algoritmo programación dinánica. Podemos basarnos en la solución anterior, en la que $f(i, c)$ devuelve un booleano que indica si es posible pintar las baldosas restantes, a partir de la i, de manera válida, sabiendo que el color c está prohibido porque fue el de la última baldosa pintada. Ahora bien, en esa solución con backtracking, se llega al mismo valor de (i, c) **muchas veces** , dependiendo de la forma exacta como se hayan pintado las baldosas anteriores.¿Pero qué es lo importante al llegar a la baldosa i en el proceso de pintar? ¿Importa la distribución **exacta** de colores en baldosas anteriores? ¿Es necesaria toda esa **información** (disponible en el arreglo global cad de la solución de backtracking) para seguir pintando a partir de la baldosa i?

La realidad es que no importa la distribución exacta de colores: únicamente importan las **cantidades** de baldosas de cada color que ya hemos utilizado (o equivalentemente, las cantidades de baldosas de cada color que aún quedan disponibles para utilizar), y el color **de la última** baldosa utilizada, ya que esa es la única que nos restringe a la hora de seguir pintando (las anteriores de más a la izquierda ya quedaron atrás).

Podemos entonces definir $f(c, i, B, G, N)$ como una función de 4 parámetros enteros i, B, G, N , y un **color prohibido** c que no se permite utilizar para la primera baldosa a pintar (porque repetiría el color de la baldosa previa), que devuelva un booleano que indique si es posible o no pintar a partir de la iésima R del archivo de entrada, dado que nos quedan para usar B baldosas blancas, G grises y N negras, y sin usar el color c en la primera baldosa.

En efecto, podemos utilizar el código de la función pintar anterior, pero aplicando memoization con los nuevos parámetros: si se llama a pintar con los mismos valores de *i, colorP rohibido, cant*[0]*, cant*[1]*, cant*[2], su valor ya estará guardado, y podemos retornarlo inmediatamente. También se puede implementar el algoritmo bottom-up, lo que sería más eficiente.

El caso base será $f(c, T, 0, 0, 0) = \text{true}$,ya que si llegamos hasta el final, no quedan baldosas, y sin importar el color prohibido, ya está todo hecho y ganamos.

La solución anterior es $O(n^4)$ ya que hay esa cantidad de posibles estados para calcular en la tabla de programación dinámica, y cada uno se calcula en $O(1)$.

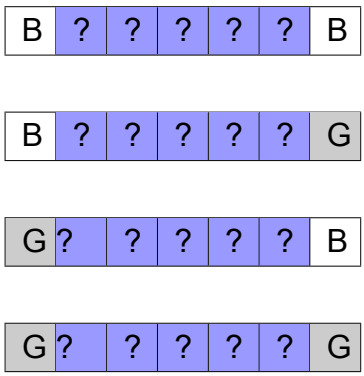
No obstante, es posible reducir la complejidad temporal a $O(n^3)$ observando que no todos los estados son posibles: Si llamamos T a la cantidad total de baldosas disponibles original (es decir, a la cantidad de letras R en la entrada), siempre tenemos $i + B + G + N = T$, ya que por cada baldosa que avanzamos

al pasar de i a $i + 1$, alguno de B, G o N baja en uno, así que la suma se conserva. Por lo tanto, podemos no guardar i y redefinir una función $g(B, G, N) = f(T - B - G - N, B, G, N)$. Como tiene un parámetro menos, la solución calculando g será de complejidad $O(n^3)$. El cálculo es exactamente igual pero reemplazando cada aparición de $f(i, B, G, N)$ por $g(B, G, N)$, y usando $T - B - G - N$ en lugar de i.

4.3.3.3. Solución $O(n)$ con un algoritmo goloso

Existe un algoritmo goloso basado en varias observaciones sobre los grupos de R consecutivas existentes en la cadena,que puede utilizarse para resolver este problema en tiempo lineal.

En primer lugar, es cómodo implementativamente no tener ninguna letra R en los extremos de la cadena, ya que así podemos suponer siempre que cada R tiene exactamente dos baldosas vecinas. Para esto, notemos que en la solución final, siempre sería posible extender los extremos de forma válida agregando alguna de B / G de un lado, y lo mismo del otro. Son 4 intentos, y si hay solucion al problema original (la parte en azul), esa misma solución funciona en alguno de los 4 casos extendidos.



Gracias a esto, todos los grupos de R consecutivas quedan delimitados en sus dos extremos con baldosas de color,que es una situación más simétrica y fácil de pensar.

Luego para cada uno de esos 4 intentos independientes: mientras quede **al menos una baldosa disponible de cada color** , el algoritmo realiza los siguientes pasos:

1. **Regla par** : Si hay un grupo de R de longitud par, bordeado por dos baldosas **del mismo color** ,jugar cualquiera de las dos posibles en un extremo del bache es una jugada segura, así que la hacemos.

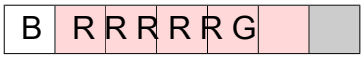
Por ejemplo, en una situación así:

Es siempre seguro pintar una baldosa así:

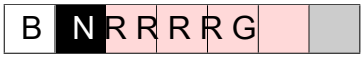
o así:

2. **Regla impar** : Si hay un grupo de R de longitud impar, bordeado por dos baldosas **de distinto color** (A y B), jugar el tercer tipo de baldosa (la C) en un borde es jugada segura.

   Por ejemplo, en una situación así:



   Es siempre seguro pintar una baldosa así:



3. **Regla delsalto** : Si **no es posible aplicar nila regla 1 ni la regla 2** , entonces en cualquier grupo de 2 o más R, la operación *ZRR → ZRZ* es jugada segura, donde *Z* es cualquier color. Notar que esta operación es posible ya que estamos suponiendo siempre que nos queda almenos una baldosa de cada color.

   Por ejemplo, en una situación así:

   Será seguro pintar una baldosa así:

   Notar que aplicar esta regla nunca genera una situación en la que puedan volver a aplicarse las reglas 1 y 2, por lo cual en el algoritmo una vez que se agotan las reglas 1 y 2, se aplica únicamente la regla 3 mientras sea posible y queden baldosas de todos los colores.

4. Si aplicando todas las reglas anteriores **todo lo posible** no se acabó ningún color, entonces se llega a una situación en la que todos los grupos de R son de tamaño 1, es decir, no quedan nunca dos R consecutivas (o se podría aplicar todavía la regla 3). En este caso, cada una de las R que quedan tiene vecinos de igual color (sino, se podría aplicar la regla impar), por lo que tiene dos colores permitidos. Hay entonces 3 tipos de R en juego: Las R con “blanco prohibido”, las R con “gris prohibido”, y las R con “negro prohibido”. Si llamamos b_g a la cantidad de R con blanco prohibido que vamos a pintar de gris, b_n a la cantidad de R con blanco prohibido que vamos a pintar de negro; g_b a la cantidad de R con gris prohibido que vamos a pintar de blanco, y así siguiendo, tenemos que la situación resultante queda modelada por las siguientes ecuaciones:

$$\begin{aligned} b_g + n_g &= G & b_g + b_n &= R_b \\ b_n + g_n &= N & g_b + g_n &= R_g \\ g_b + n_b &= B & n_b + n_g &= R_n \end{aligned}$$

Donde G, N y B son las cantidades aún disponibles de cada color, y R_b, R_g y R_n son las cantidades de R con blanco prohibido, con gris prohibido y con negro prohibido. Las 6 variables buscadas deberán tomar valores no negativos. Si fijamos por ejemplo b_g, se pueden despejar todos los demás valores:

$$\begin{aligned} n_g &= G - b_g \\ n_b &= R_n - n_g \\ g_b &= B - n_b \\ g_n &= R_g - g_b \\ b_n &= N - g_n \end{aligned}$$

Con lo que simplemente podemos probar exhaustivamente todos los valores posibles de b_g, y para cada uno de ellos realizar los despejes para ver si todas las variables quedan no negativas.Podemos tomar el primer valor de b_g que funcione para generar la solución.

Si durante la aplicación de las reglas anteriores, en algún momento se agotan las baldosas disponibles de algún color, tenemos un caso donde solamente hay dos colores disponibles para usar. En este caso, cada grupo de R debe pintarse **alternadamente** (o bien GNGNGN o bien NGNGNG) y ahora lo resolvemos aplicando el siguiente método goloso sencillo (supondremos que B es el color agotado, y solamente quedan para usar G y N):

1. Los grupos de R que tengan una G o una N en uno de sus extremos están determinados, ya que una sola de las dos opciones posibles es válida. Llenamos todos ellos de la única manera posible.
2. Quedan entonces únicamente los grupos que tienen en sus extremos dos baldosas B, que pueden llenarse alternadamente de cualquiera de las dos formas posibles.
3. En los grupos de longitud par, no importa cuál de las dos formas usamos, porque ambas usan exactamente la misma cantidad de baldosas de cada color.
4. En los impares, una de las opciones aumenta en uno la diferencia de “negros menos grises”, y la otra la disminuye en 1. Basta entonces con elegir aquella que modifique la diferencia actual en la dirección correcta. Cuando la diferencia actual es correcta, se puede elegir cualquiera de las dos opciones libremente (quedando ya el próximo grupo condicionado).

Para demostrar la correctitud de este algoritmo, conviene encarar una demostración de la propiedad siguiente:

Para cada paso del algoritmo, vale que si suponemos que existe alguna solución al problema que queda por resolver desde ese paso, entonces sigue existiendo alguna solución luego de que el algoritmo realice ese paso.

Probar esto basta para probar que el algoritmo sea correcto, pues como sabemos que inicialmente hay solución, y el algoritmo va pintando cada vez más baldosas, en algún momento llega a una configuración con todo pintado donde aún existe solución, es decir, que ha llegado finalmente a una solución válida.

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: