Solucionario de la Olimpíada Informática Argentina 2018

Autores

Brian Bokser

Román Castellarin

Sebastián Cherny

Agustín Santiago Gutiérrez

Facundo Martín Gutiérrez

Carlos Miguel Soto

Ariel Zylber

Índice general

Capítulo 1

Introducción

Todos los enunciados de los problemas se encuentran disponibles en: http://www.oia.unsam.edu.ar/problemas-categoria-programacion

Además, se pueden realizar envíos de soluciones para los problemas en el juez online de la olimpíada http://juez.oia.unsam.edu.ar. Se puede entrar directamente la página de un problema particular accediendo a http://juez.oia.unsam.edu.ar/#/task/PROBLEMA/statement, reemplazando el texto `PROBLEMA` por el "código de problema" correspondiente (el nombre corto: `electromovil`, `solitario`, `gps`, etc).

Capítulo 2

Selectivo para la IOI

2.1. Día 1

2.1.1. Problema 1: Auto Eléctrico [electromovil]

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

Al comenzar nuestro viaje tenemos una cierta autonomía en nuestra batería y nuestro objetivo consiste en alcanzar la última estación de servicio (en la cual podemos asumir que cargamos nuestra batería por simplicidad).

Un primer enfoque posible para resolver el problema utiliza ***programación dinámica*** de la siguiente forma. Llamemos $f(j)$ a la menor cantidad de veces que debemos cambiar la batería para alcanzar la j-ésima estación (incluyendo el cambio de batería en la estación j). Sabemos que $f(0) = 0$ al comenzar nuestro recorrido.

Veamos qué ocurre con la j-ésima estación. Para esto puede sernos útil asumir que ya conocemos el resultado de $f(i)$ para todo $i < j$. A la estación j tuvimos que llegar habiendo cambiado la batería en alguna estación anterior. De todas estas posibiles estaciones anteriores nos interesan solo aquellas cuya *autonomía es suficiente para alcanzar a la estación* j (de no ser así necesariamente tendrá que pasar por una estación por la que sí alcance su autonomía, cuyo caso estamos contemplando).

Concluimos que $f(j) = \min_{i<j}\{f(i)+1\} = \min_{i<j} \overbrace{\{f(i)\}}^{\text{estación que vengo}} + \overbrace{1}^{\text{cambio la batería}}$
Teniendo esta recursión, aplicando programación dinámica (para más información en el tema se puede ver http://wiki.oia.unsam.edu.ar/algoritmos-oia/programacion-dinamica), obtenemos una solución de complejidad $\mathcal{O}(E^2)$, pues por cada estación debemos consultar todas sus anteriores.

Concretamente, si queremos calcular $f(i)$ debemos analizar la situación de todos los $0 \leq i < j$. En particular, solo nos interesan aquellas ubicaciones con `ubicacion[i] + autonomia[i]` $\geq$ `ubicacion[j]`, de todas ellas buscamos algún i_j que minimice $f(i)$. Finalmente $f(j) = f(i_j) + 1$. A continuación podemos ver un ejemplo en la iteración de $j = 4$.

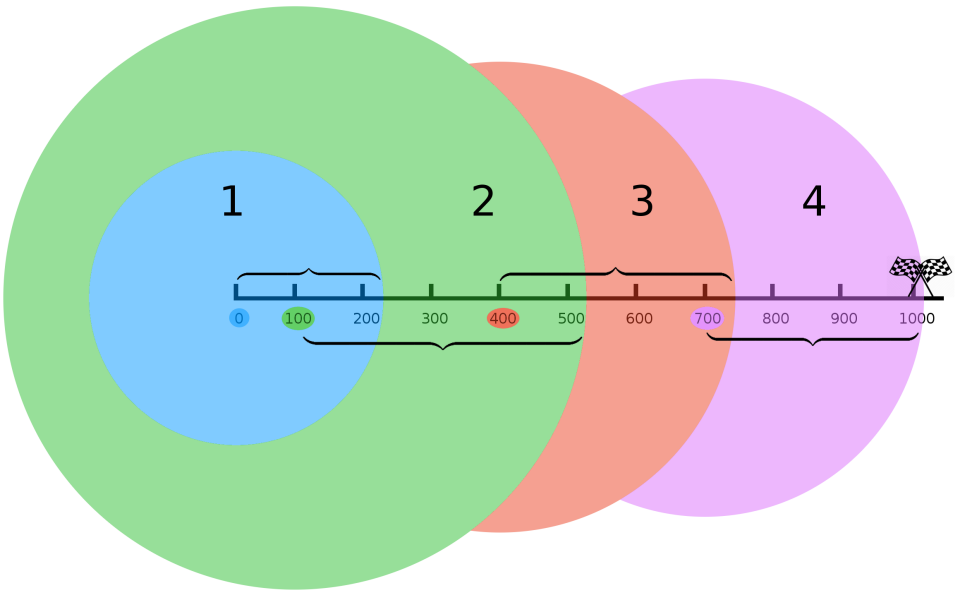
j	0	1	2	3	4	5	6	7	8	9	10
ubicacion[j]	0	100	200	300	400	500	600	700	800	900	1000
autonomia[j]	225	415	150	225	350	125	275	330	225	315	280
$f(j)$	0	1	1	2	2	2	3	3	4	4	4
padre[j]	-1	0	0	1	1	1	4	4	6	6	6

Para finalizar el problema con esta idea, resta ver cómo recalcular el camino. Esto puede hacerse con una idea relativamente estándar que consiste en *almacenar en un arreglo auxiliar dónde se realizó el mínimo* de $f(i)$ dentro de los $i < j$ al resolver el subproblema i. Corresponde a lo que anteriormente llamamos i_j, y en el ejemplo anterior está almacenado en `padre[j] = i`$_j$. A continuación se muestra un pseudocódigo que refleja este enfoque. Se asume que `ubicacion` y `autonomia` son dos arreglos de tamaño $E + 1$ incluyendo el punto de partida.

Algorithm 1 Solución 1 al Problema 1 de Selectivo 2018 Día 1 - electromóvil

```
1: function SORTUCHO(UBICACION: ARR. DE ENTEROS, AUTONOMIA: ARR. DE ENTEROS)(ARR. de ENTEROS)
2:     f ← [∞, ...,∞]                                  ▷ Arreglo de E+1 lugares
3:     padre ← [-1, ...,-1]                            ▷ Arreglo de E+1 lugares
4:     f[0] ← 0
5:     for j = 1 ... E do                              ▷ Recorremos en orden creciente
6:         for i = 0 ... j-1 do
7:             if ubicacion[i] + autonomia[i] ≥ ubicacion[j] then
8:                 if f[i] < f[j] then
9:                     f[j] ← f[i]+1
10:                    padre[j] ← i
11:    solucion ← []
12:    actual ← E
13:    while padre[actual] ≠ -1 do ▷ Usamos E ≥ 2 (y 0 no va en la solución)
14:        solucion.agregar_al_frente(ubicacion[actual])
15:        actual ← padre[actual]
16:    return solucion
```

Dado que en las cotas del enunciado tenemos que $E \leq 1{,}000{,}000$, no esperamos obtener el puntaje total por este problema con este enfoque. Antes de introducir el siguiente enfoque veamos qué ocurre con las estaciones vistas en orden. En un principio podremos llegar a cualquier estación con `ubicacion[j]` $\leq$ `autonomia[0]`. De todas ellas, ¿hay alguna que sea *mejor que otra* o que las *domine* en algún sentido? Sí. De todas las alcanzables, nos conviene tomar aquella que maximice $autonomia[j] + ubicacion[j]$, dado que es la que nos permite llegar más lejos con la misma cantidad de cambios. Hecha esta elección, la situación procede análogamente como se muestra en la figura, y se detalla en el siguiente pseudocódigo.



Algorithm 2 Solución 2 al Problema 1 de Selectivo 2018 Día 1 - electromóvil

```
1: function ELECTROMOVIL(UBICACION: ARR. DE ENTEROS, AUTONOMIA: ARR. DE ENTEROS)(ARR. de ENTEROS)
2:     desde ← ubicacion[0]
3:     hasta_actual ← ubicacion[0]
4:     hasta_proximo ← ubicacion[0]+autonomia[0]
5:     solucion ← []
6:     for j = 1 ... E do                          ▷ Recorremos en orden creciente
7:         if ubicacion[j] > hasta_proximo then     ▷ Imposible de alcanzar
8:             solucion ← []
9:             break
10:        else if ubicacion[j] > hasta_actual then     ▷ Cambio de batería
11:            solucion.agregar_al_final(desde)
12:            hasta_actual ← hasta_proximo
13:        if ubicacion[j]+autonomia[j] > hasta_proximo then     ▷ Siempre vemos si podemos mejorar la mayor distancia que podemos alcanzar
14:            desde ← ubicacion[j]
15:            hasta_proximo ← ubicacion[j]+autonomia[j]
16:    if solucion ≠ [] then
17:        solucion.sacar_del_frente()    ▷ Habíamos agregado ubicacion[0]
18:        solucion.agregar_al_final(ubicacion[E])
19:    return solucion
```

La complejidad de este algoritmo (que aprovecha que la entrada ya está ordenada) es lineal: $\mathcal{O}(E)$.

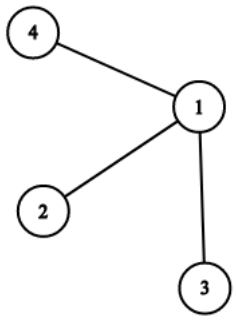
2.1.2. Problema 2: Contando subredes [subredes]

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

Este problema es interesante ya que fue el primer problema "Output-Only" en la OIA. En IOI este tipo de problemas son comunes: el participante tiene disponibles en su computadora para utilizar **todos los casos de prueba**, y solamente envía al juez online los archivos de salida con las soluciones, sin importar el código, lenguaje o herramientas que se hayan usado para generar esas salidas.

El problema en sí consiste en contar la cantidad de subgrafos (subredes) de un grafo (red) G, isomorfos a otro grafo (subred) H dado. En este caso, había 10 archivos de entrada, cada uno correspondía a un H distinto, y cada uno valía 10 puntos. La idea era ver los archivos de entrada para saber cuáles eran estos grafos, y escribir así soluciones específicas para ellos (aunque varias están relacionadas entre sí).

2.1.2.1. $K_{1,3}$



Cada aparición de esta subred determina en forma precisa cuál es el nodo de tipo "1" utilizado para esa aparición, pues es el único de grado 3 en la subred (esta idea de nodos "distinguibles" o "indistinguibles" es fundamental para evitar contar varias veces lo mismo). Por lo tanto, podemos contar todas las apariciones sumando por cada nodo v de la red completa, la cantidad de veces que aparece la subred teniendo a v como el nodo "1" de la figura (el de grado 3).

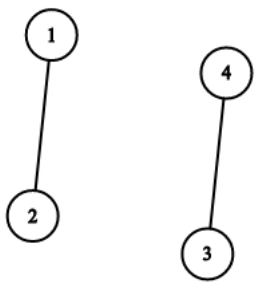
Una vez fijado que v representa al nodo de grado 3, se puede observar que para armar la subred simplemente hay que elegir 3 vecinos distintos de v. Hay exactamente una subred posible por cada elección de 3 vecinos. Como v tiene $d(v)$ vecinos, sumando todo la respuesta para este caso es directamente:

$$\sum_{v \in V} \binom{d(v)}{3} = \sum_{v \in V} \frac{d(v)(d(v)-1)(d(v)-2)}{6}$$

Donde es importante notar que si el grado es menor que 3, el combinatorio correspondiente debería dar 0, pues no hay subred posible para ese v.

Implementativamente, solo hay que calcular los grados en el grafo, lo que se puede hacer en tiempo lineal, y luego hacer la cuenta.

2.1.2.2. $K_2 \cup K_2$



Para fijar una subred en este caso, hay que elegir dos aristas que no se toquen.

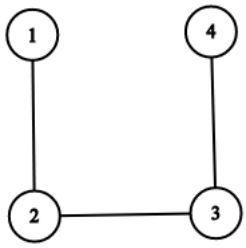
Para contar cuántas posibilidades hay, podemos iterar todas las aristas, y por cada una de ellas, sumar la cantidad de aristas que no la tocan.

Si una arista e une dos vértices de grados d_1 y d_2, entonces toca $(d_1-1)+(d_2-1)$ aristas diferentes (pues a los grados hay que restar uno, correspondiente a la propia arista e). Por lo tanto del total m de aristas, considerando que no están disponibles e ni sus "aristas vecinas", quedan $m-(d_1-1)-(d_2-1)-1 = m-d_1-d_2+1$ aristas disponibles para usar.

Sumando este número para todas las aristas habremos contado todas las subredes posibles, pero habremos contado a cada una dos veces: Una vez al pararnos en una de sus aristas, y otra vez al pararnos en la otra. Por esta razón, la respuesta final será la mitad de esa suma.

De forma similar al caso anterior, para resolver este caso basta con calcular los grados de los nodos en tiempo lineal, y luego recorrer las aristas sumando para hacer la cuenta (y dividir por dos al final), por lo que ambos códigos son casi iguales cambiando levemente la cuenta. Completando ambos se obtenían 20 puntos.

2.1.2.3. P_3

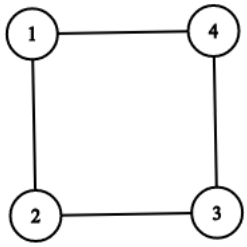


Para contar estas subredes, una posibilidad es notar que la subred determina en forma precisa la arista "central", entre los nodos 2 y 3. Podemos iterar las aristas sumando cuántas subredes de esta forma tienen a la arista e como central.

Fijada la arista e que une los nodos u y v, para completar la subred hay que elegir un nodo vecino de u y uno vecino de v **diferentes entre sí** (pues elegir el mismo formaría un triángulo, que no se corresponde con esta subred). La cantidad de formas de elegir los vecinos en total sería $d_1 \cdot d_2$, pero a eso debemos entonces restar la cantidad de vecinos en común entre u y v.

Si tenemos el grafo representado, por ejemplo, con matriz de adyacencia, es sencillo calcular la intersección entre los vecinos de u y v en tiempo $O(n)$: Basta probar todos los nodos x distintos de u y v, y ver si existen ambas aristas, la x–u y la x–v. Como esto se hace para cada arista, el tiempo total resulta $O(nm)$.

2.1.2.4. C_4



La idea que explicaremos para contar este subgrafo es bastante potente, y de hecho puede utilizarse para resolver también el caso anterior, aunque consideramos que el método ya explicado es más simple de entender y deducir.

La idea es precomputar una tabla de $n \times n$, que para cada par de nodos u, v, indique la cantidad de nodos x (que no sean ni u ni v) tales que existen las aristas x–u y x–v.

El algoritmo para precomputarla es muy simple: Se inicializa la tabla en cero, y luego se recorren todos los nodos. Al procesar x, se toman todos los pares de vecinos distintos u y v, y se suma 1 a la matriz en la posición (o posiciones) correspondiente al par u, v.

Es evidente que este algoritmo para computar la matriz tiene una complejidad de peor caso $O(n^3)$, pues tiene 3 for anidados que recorren hasta n como máximo. Pero de hecho si analizamos con cuidado, podemos demostrar que es $O(nm)$: Esto porque la cantidad total de veces que se iteran los fors interiores será proporcional a la suma total de todos los números en la matriz resultado (ya que en cada paso se incrementa alguno). Pero cada incremento "involucra" dos aristas: x–u y x–v. Y una arista en particular puede estar involucrada en un máximo de $2(n-2)$ incrementos en total, y como hay m aristas, esto hace que el total de incrementos máximo sea $O(nm)$.

Otra forma de demostrar lo mismo es contando la cantidad de pares:

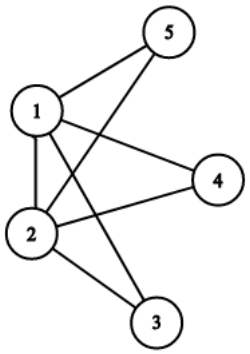
$$\sum_{v \in V} \binom{d(v)}{2} = \sum_{v \in V} \frac{d(v)(d(v)-1)}{2} \leq \sum_{v \in V} \frac{d(v)n}{2} = \frac{n}{2} \sum_{v \in V} d(v) = \frac{n}{2} \cdot 2m = nm$$

Finalmente, una vez que tenemos esa tabla precomputada, podemos observar que si fijamos un par de vértices (u, v) para ser una diagonal del cuadrado, lo que necesitamos para formar el cuadrado son dos vértices distintos que tengan aristas a

u y a v. Pero justamente, la matriz precomputada nos dice esto en la fila u columna v. Así que por cada par de vértices (u, v), si la matriz dice que existen t vértices con la propiedad, debemos sumar $\frac{t(t-1)}{2}$, pues se deben elegir dos de ellos.

Al número final se lo debe dividir por 2, pues como hay dos diagonales en cada cuadrado, los estamos contando doble a todos.

2.1.2.5. $(K_3 \cup K_1 \cup K_1)^c$

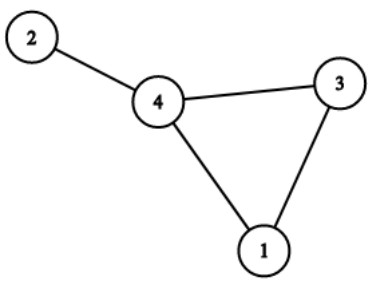


Este caso también puede resolverse con la misma matriz precomputada explicada para el subgrafo anterior: aquí podemos iterar cada para arista (u, v), y contar cuántos subgrafos hay en donde (u, v) corresponden a los nodos 1 y 2 del dibujo. Notar que estos nodos están destacados en la subred, pues son los únicos de grado 4, así que estaremos contando cada subred exactamente una vez.

De forma similar al caso anterior, podemos ver que lo que necesitamos ahora no son 2 sino 3 vértices distintos que tengan aristas tanto a u como a v. El código entonces es idéntico al del caso anterior, pero recorriendo únicamente aristas sumando $\frac{t(t-1)(t-2)}{6}$, y sin la división por 2 al final.

Otra forma posible de resolver este caso es notar que el nodo 1 se conecta a todos los demás, así que entre sus vecinos lo que buscamos es el grafo $K_{1,3}$ del caso 1. Esta misma idea se explica en más detalle para los dos grafos que siguen.

2.1.2.6. $(P_2 \cup K_1)^c$

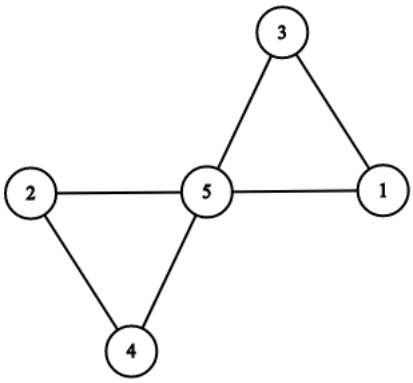


Veamos dos formas posibles de resolver este caso:

La primera se basa nuevamente en la matriz precomputada anterior. Iteramos con dos fors fijando un primero nodo u, que tomará el rol del nodo 4, y un segundo nodo v, que tomará el rol del nodo 1. Para determinar la red una vez fijados u y v, solamente falta elegir qué nodos cumplirán el rol de 3 y de 2. Como el nodo que cumpla el rol de 3 deberá ser vecino de u y v, tenemos en la matriz en la fila u columna v ya computado la cantidad de valores posibles para el nodo 3. Y finalmente, la cantidad de elecciones para el nodo 2 será simplemente $d(u) - 2$, pues en el conteo debemos excluir a los vecinos 1 y 3. Sumando todo esto y dividiendo por 2 (pues 1 y 3 son simétricos así que los contaríamos doble) obtenemos la respuesta.

La segunda se basa en notar que el nodo 4 es muy especial en la subred: se conecta a TODOS los demás. Si lo fijamos en un nodo v, podemos contar entre los vecinos de v en G en busca de una arista, y un nodo que no toque la arista. Si entre los vecinos de v existen t aristas, en total debemos sumar al procesar el nodo v un total de $t(d(v) - 2)$ subredes. Por cada nodo v, se puede calcular la cantidad t de aristas entre sus vecinos en tiempo $O(d(v)^2)$ probando todos los pares de vecinos. Como ya vimos antes, al sumar los cuadrados de los grados de un grafo el resultado es $O(nm)$, así que esta otra solución tiene la misma complejidad que la anterior.

2.1.2.7. $(C_4 \cup K_1)^c$



Podemos aplicar la misma idea de la segunda solución al caso anterior: el nodo 5 es muy particular pues se conecta con todos los demás de la subred. Iteramos entonces todos los posibles nodos v, contando en cada caso la cantidad de subredes que lo utilizan como nodo 5. Para esto, basta con elegir 2 aristas disjuntas entre sus vecinos: pero ya vimos en el caso 2 como puede contarse eso en tiempo lineal, así que lo aplicamos al grafo de vecinos de cada nodo, sumando. Como antes, la complejidad final resulta $O(nm)$

2.1.2.8. K_{10}

Los últimos archivos contienen subredes computacionalmente difíciles. Resolviendo todos los anteriores, ya tendríamos 70 puntos, y lo que sigue es únicamente para obtener los 30 restantes.

El problema general de contar cuántas veces aparece un subgrafo arbitrario dentro de otro es NP-hard, y por lo tanto nadie conoce un algoritmo general polinomial, y los expertos creen que no existe ninguno. Es por esto que estos últimos archivos contienen grafos más pequeños.

La solución para todos ellos es utilizar un buen algoritmo de backtracking, con las podas suficientes para que se pueda ejecutar en el tiempo de la prueba.

Para el caso de prueba en el cual la subred es un grafo completo de 10 nodos, hay que aprovechar la particularidad de esta red para obtener un backtracking eficiente (y probablemente, un poco más simple en su implementación). En este caso, el orden de los 10 nodos elegidos para la subred no es importante, solo importa contar cuántos subconjuntos de 10 nodos existen, tales que todos ellos estén conectados entre sí. Los backtracking generales que distinguen nodos, como los que describiremos en la sección siguiente, tienen un tiempo de ejecución proporcional a la cantidad de *automorfismos* de la subred: es decir, la cantidad de reordenamientos posibles de los vértices, de modo que se vuelve a obtener la misma subred. En el caso particular del grafo completo, todos los ordenamientos valen, así que esos algoritmos de backtracking serían 10! veces más lentos que el que proponemos aquí para este caso (más de tres millones de veces más lentos).

El algoritmo de backtracking propuesto para este caso es ir nodo por nodo, eligiendo si incluirlo o no en el conjunto de nodos de la subred. El estado de la recursión serían dos índices (i, k), que por ejemplo pueden iniciar en $(0, 10)$: i indica cuál es el siguiente nodo a considerar, y k indica cuántos nodos nos faltan elegir en la subred. Si elegimos el nodo i, se pasaría a $(i + 1, k - 1)$ y sino a $(i + 1, k)$.

Las podas fundamentales que proponemos aplicar para que esto funcione mucho más rápido son tres:

- Cuando el grafo de los $n-i$ nodos restantes es completo, se puede simplemente sumar $\binom{n-i}{k}$ a la respuesta, ya que cualquier conjunto posible de k nodos para elegir sirve. Esta situación puede detectarse eficientemente sin aumentar la complejidad asintótica del backtracking, manteniendo un contador de aristas durante el procesamiento del backtracking, pues si $2m' = n'(n' - 1)$ entonces el grafo es completo (donde $n' = n - i$ es la cantidad de nodos restantes, y

similarmente m' son las aristas restantes).

Para mantener eficientemente el conteo de aristas m', se puede mantener el grado de cada nodo en el grafo de los $n - i$ nodos restantes, y entonces $2m'$ será la suma de esos grados.

- Si entre los nodos restantes hay alguno cuyo grado restante es menor que $k-1$, estamos seguros de que no sirve para el subgrafo buscado, y podemos borrarlo. Esto altera el grado de sus vecinos, reduciéndolo en 1, y por lo tanto podríamos eliminar varios nodos en cascada en cada paso mediante este procedimiento.
- Intentar ordenar los nodos que se van examinando por grado. Esto es porque al elegir un nodo para la subred, se descartan a todos los que no son vecinos, así que un nodo con poco grado descarta a casi todos y por lo tanto reduce mucho el espacio de búsqueda.

Notar que en la transición del backtracking, es fundamental marcar como borrados (y en particular, actualizar los grados) aquellos nodos que no se conectan a alguno de los ya elegidos, de modo de no tener que verificarlo en cada paso, y reduciendo enormemente el espacio de búsqueda restante.

Esta forma de enumerar las subredes de 10 nodos es más que suficientemente eficiente para el problema en cuestión, terminando en uno o dos minutos. Otros backtrackings con menos podas podrían utilizarse para igualmente obtener los 10 puntos de este caso, pero tardan más en ejecutarse.

2.1.2.9. Grafos aleatorios

Los últimos dos casos (20 puntos) tienen grafos complicados aleatorios / arbitrarios. Para estos grafos no podemos aplicar el backtracking del paso anterior, pues los nodos no son todos equivalentes, y es importante tener en cuenta qué nodo del grafo representa qué nodo de la subred. Estos son los casos más difíciles de resolver de todo el problema.

El estado fundamental a mantener en este caso es más complejo, pues no bastan dos índices, sino que hay que saber el **conjunto** de nodos de la subred que falta asignar (no solamente su cantidad), y los nodos del grafo disponibles a los que pueden ser asignados.

Al igual que comentamos en el caso anterior, examinar los nodos en orden creciente de grado reduce notoriamente el espacio de búsqueda, pues los nodos con poco grado son heurísticamente los que más restringen los subsiguientes posibles nodos que elijamos (específicamente, generan mayor *cantidad* de restricciones).

Alternativamente, una idea adicional permite aumentar muchísimo la eficiencia: se pueden mantener listas o conjuntos de nodos, **uno por cada uno** de los 10 nodos de la subred. Cada uno de estos conjuntos indica **todos** los nodos del grafo que todavía es posible que se correspondan con el nodo de la subred asociado a ese conjunto.

Inicialmente, estos conjuntos contienen todos los nodos del grafo, pero cada elección que se realice filtra las listas. Si alguna se vacía, se puede podar, pues no ha quedado ningún nodo posible para "cumplir el rol" de ese nodo de la subred. En cada paso, se elige la lista más chica, y se prueban todas esas posibilidades para el nodo que tomará ese rol, filtrando las listas correspondientes a vecinos en la subred, eliminando aquellos nodos que no sean vecinos del nodo elegido en el grafo.

Como antes, un backtracking que use todas estas ideas de podas ejecuta sumamente rápido para los casos de prueba, pero no es estrictamente necesario realizar exactamente esta solución: un backtracking diferente, o uno similar con menos podas, puede obtener también la solución en el tiempo de prueba, aunque posiblemente demore más tiempo en ejecutarse.

Un detalle final es que si hay nodos indistinguibles, el algoritmo anterior (y casi cualquier backtracking razonable) cuenta cada subred múltiples veces (exactamente una vez por cada automorfismo de la subred), pues cuenta todas las formas diferentes de hacer corresponder nodos de la subred con nodos del grafo.

La solución es dividir por la cantidad de posibilidades existentes. En el caso particular de los casos de prueba, hay solo 2 automorfismos, así que basta con poner una división por dos en la respuesta final. Como la subred tiene solamente 10 nodos, la cantidad de automorfismos puede contarse fácilmente en $O(n!)$ utilizando por ejemplo el siguiente fragmento de código:

```
automorfismos = 0;
int permu[10];
for (int i=0;i<10;i++)
    permu[i] = i;
do
{
    bool ok = true;
    for(int i=0;i<10;i++)
    for(int j=0;j<10;j++)
        ok &= subred[i][j] == subred[permu[i]][permu[j]];
    automorfismos += ok;
} while (next_permutation(permu, permu+10));
```

2.1.3. Problema 3: Cartas [solitario]

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

Conviene separar este problemas en dos partes principales: la de descubrir la *posición* en el mazo inicial, de la carta que queda al final, y luego la de *identificar* qué carta hay en esa posición.

2.1.3.1. Posición

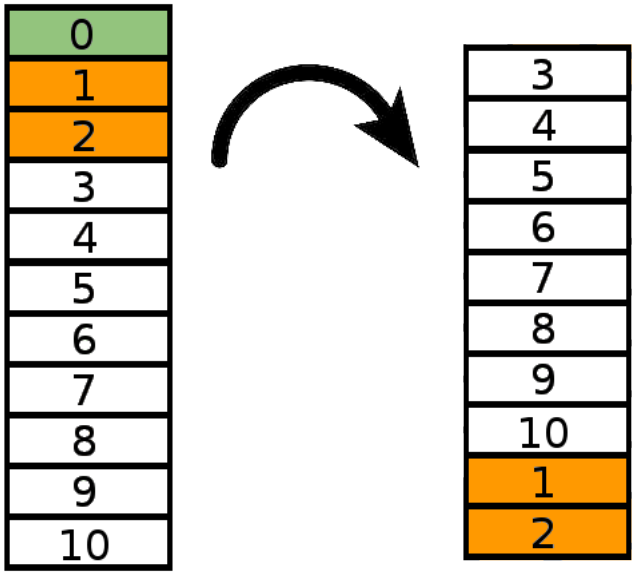
Supongamos que tenemos un mazo de N cartas numerado, de manera que la carta de arriba del mazo es la carta 0, la siguiente es la carta 1, y así hasta la carta de abajo de todo, que será la carta $N-1$.

Nuestro objetivo será computar $f(K, N)$, el número de carta que queda al final si realizamos el procedimiento indicado.

Esta función f puede calcularse por **simulación**: Podríamos por ejemplo tener un arrego que represente el mazo, con una carta por cada posición en el arreglo, y realizar todas las operaciones indicadas hasta que quede un arreglo de tamaño 1 con la carta final. Si se utiliza una cola de dos puntas eficiente (por ejemplo utilizando la estructura de datos `deque` en C++), esto puede realizarse con complejidad $O(NK)$ (ya que el procedimiento tiene K pasos por cada carta que se elimina, y hay N cartas).

Veamos a continuación como calcular f de un modo más "matemático", sin simular directamente el procedimiento.

Si pensamos en el primer paso, por ejemplo con $K = 2$ y $N = 11$ tenemos una situación como la que sigue:



La carta superior, que siempre es la 0, se indica en verde, y esta carta es **descartada** completamente del mazo.

Por otro lado, las siguientes $K = 2$ cartas, que estarán numeradas desde 1 hasta K, se indican en naranja: estas cartas pasan a la parte de abajo del mazo, como resultado de repetir K veces la acción de "pasar la carta de arriba a la parte de abajo".

El procedimiento anterior tiene una salvedad: Si fuera $K \geq N$, no sería cierto que simplemente se pasan hacia abajo las cartas de 1 a K, ya que al dar una vuelta entera al mazo se volverían a pasar por las mismas cartas. La observación aquí es que realizar la acción de pasar carta $N - 1$ veces seguidas no tienen ningún efecto (porque al haber pasado por todas las $N - 1$ cartas restantes, el mazo vuelve a quedar exactamente como estaba). Podemos entonces quedarnos con el resto $K' = \texttt{K\% (N-1)}$, que será un número entre 0 y $N - 2$ de cartas naranjas que sí darán una situación como la del dibujo de arriba.

La clave de esto es que luego del primer paso, **tenemos un mazo más chico**: Este mazo tiene únicamente $N - 1$ cartas. Por lo tanto, $f(K, N - 1)$ indicaría por definición cuál es la posición de la carta de este segundo mazo que sobrevive como resultado del procedimiento. Esto da lugar a una solución recursiva: interpretando esta posición $f(K, N - 1)$ dentro de este segundo mazo, según esa posición caiga en la parte blanca o en la parte naranja, podremos calcular finalmente cuál es el número de carta sobreviviente en todo el proceso de las N cartas, incluyendo el primer paso también.

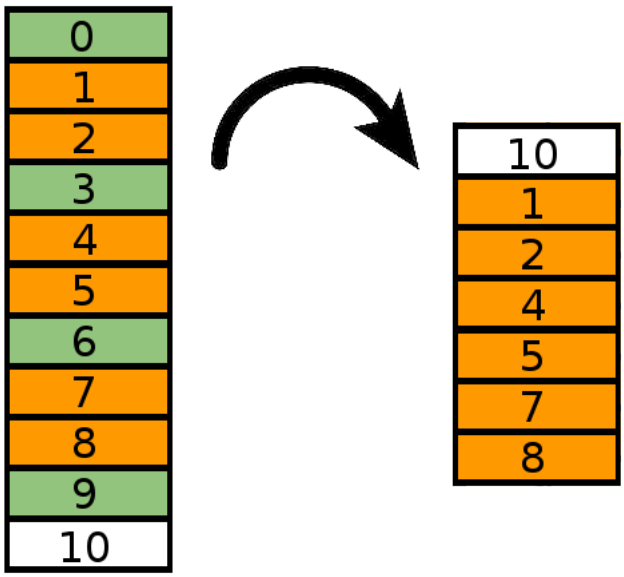
Esto da lugar a la siguiente función recursiva:

$$f(K,N) = \begin{cases} 0 & \text{si } N = 1 \\ \text{Ponemos: } K' = K \bmod (N-1) & \\ K' + 1 + f(K, N-1) & \text{si } f(K,N-1) < N - K' \\ K' + 1 + f(K, N-1) - N & \text{sino} \end{cases}$$

Notar la diferencia entre K y K'.

Como hay N números entre 1 y N, y para cada uno de esos valores la función llama al anterior, en total se calculan N valores de la función y la complejidad de este método es $O(N)$.

Podemos aún mejorarlo para obtener la solución esperada de 100 puntos: Observemos que si en lugar de imaginarnos únicamente el primer paso, nos imaginamos 2 pasos adicionales, tendríamos una situación como la que sigue:



Tenemos que todas las cartas cuya posición es múltiplo de $K+1$ son cartas verdes que son **descartadas** del mazo. De esta manera, con este método estaremos descartando $T = 1 + \left\lfloor \frac{N-1}{(K+1)} \right\rfloor$ cartas a la vez, todo en un único análisis de varios pasos seguidos. Notar que si $K = 0$ estaremos descartando todas las cartas y nos queda un mazo vacío, así que lo correcto será separar ese caso para los cálculos (aprovechando que $f(0,N) = N-1$).

La posición final de la carta en el mazo resultante de la derecha podemos

obtenerla, como antes, llamando a $f(K, N-T)$, y solo resta implementar la lógica para determinar cuál es el número de carta en el mazo original de la carta que está en esa posición del mazo de la derecha. Para esto, observemos que el mazo de la derecha se compone primero de una sección de cartas blancas, formadas por todas las que sobraron al final, que son exactamente $S = (N-1) \bmod (K+1)$. Y luego vienen las cartas naranja, que internamente se encuentran divididas en bloques de tamaño exactamente K: el primero tiene las cartas 1 a K, el segundo las cartas $1+K+1$ a $K+K+1$, y en general el i-ésimo (contando desde 0) tendrá las cartas $1+i(K+1)$ hasta $K+i(K+1)$.

Todo el análisis nuevo anterior será para el caso en que $K < N-1$, ya que sino se dará toda la vuelta al mazo, y entonces solo podremos sacar de a una carta exactamente como hacíamos antes.

$$g(K,N) = \begin{cases} 0 & \text{si } N = 1 \\ N-1 & \text{si } K = 0 \\ f(K,N) & \text{si } K \geq N-1 \\ \text{Ponemos: } S = (N-1) \bmod (K+1) & \\ \text{Ponemos: } T = 1 + \left\lfloor \frac{N-1}{(K+1)} \right\rfloor & \\ N - S + g(K, N-T) & \text{si } g(K,N-T) < S \\ \left\lfloor \frac{g(K,N-T)-S}{K} \right\rfloor + 1 + g(K,N-T) - S & \text{sino} \end{cases}$$

Notar que esta nueva función g más eficiente utiliza la anterior f cuando el valor de N se vuelve suficientemente pequeño, ya que si el K es grande en relación al N y se da una vuelta entera al mazo en cada paso, el cálculo como está en g no es correcto.

Se puede probar que la complejidad de este nuevo método es $O(K \ln N)$, muchísimo más eficiente para valores grandes de N gracias a que se descartan muchísimas cartas en cada paso.

2.1.3.2. Identificación de la carta

Esta es la parte más fácil de las dos en que se divide el problema. Como el mazo es copia de muchos mazos pequeños pero todos iguales, solo importa la posición de la carta final dentro de su mazo pequeño correspondiente.

Sabemos que la carta que sobrevive es $f(K,N) = g(K,N)$ que calculamos en la parte anterior. Pero ese número de carta es **desde la parte superior del mazo**. Los montones se arman de abajo hacia arriba, así que es más cómodo en realidad

saber la ubicación desde abajo (pero para todos los cálculos anteriores, sí era mucho más cómodo contar desde arriba). Entonces tenemos $p = N - 1 - f(K, N)$ que será la posición de la carta pero contada desde la parte de abajo del mazo.

Dado este p, como las cartas 0 a $M - 1$ (contando desde abajo) forman la primera copia del mazo, las M a $2M - 1$ forman la segunda, y así siguiendo, lo único que importa es p mod M, ya que eso indica la posición de la carta que sobrevive dentro de su correspondiente copia del mazo pequeño.

La respuesta final por lo tanto será `c[p%M]`, donde c es el arreglo donde se indican las M cartas que forman el mazo pequeño que es copiado muchas veces para obtener las N cartas totales.

2.2. Día 2

2.2.1. Problema 1: GPS anticongestión [gps]

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

El problema nos pide encontrar las longitudes de los k caminos más cortos entre dos nodos fijos en un grafo dirigido con pesos.

Un primer enfoque posible es comenzar con $k = 1$. ¿Cómo encontramos la distancia más corta entre dos nodos en un grafo con pesos? Para resolver este caso, empleamos el algoritmo de *Dijkstra*, que se explica en el siguiente link: http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/dijkstra.

En resumen, el algoritmo de Dijkstra se puede pensar como "explorar todos los caminos y ver cuál es más corto", pero con una optimizacion que evita mirar todos los (posiblemente infinitos) caminos. Esta optimizacion consiste en explorar, por cada nodo del grafo, sólo *uno* de los caminos que pasa por ese nodo. Para que el algoritmo siga siendo correcto, además, este camino que exploramos debe ser el *camino óptimo* hasta ese nodo, es decir, el que tiene menor longitud.

Implementativamente, esto se resuelve manteniendo un conjunto de "candidatos" a caminos óptimos. Si en cada paso exploramos el candidato más corto que termina en un nodo todavía no explorado, es fácil ver que ese candidato a camino óptimo era, en realidad, el camino óptimo. Una implementación del algoritmo sería así:

Algorithm 3 Implementación de Dijkstra

```
1: function DIJKSTRA(GRAFO ARR. DE ARISTAS, NODO_INICIAL ENTERO, NODO_DESTINO ENTERO)(ENTERO)
2:     visitado ← [No, ..., No]
3:     candidatos ← {}
4:     camino_inicial.longitud ← 0
5:     camino_inicial.nodo ← nodo_inicial
6:     candidatos.añadir_nuevo_camino(camino_inicial)
7:     distancia_a_nodo_final ← ∞
8:     while no candidatos.vacío() do
9:         longitud, nodo ← candidatos.camino_más_corto()
10:        candidatos.borrar_camino_más_corto()
11:        if no visitado[nodo] then
12:            visitado[nodo] ← Sí
13:            if nodo es nodo_destino then
14:                distancia_a_nodo_final ← longitud
15:            for peso, hijo en grafo[nodo] do
16:                camino.longitud ← peso+longitud
17:                camino.nodo ← hijo
18:                candidatos.añadir_nuevo_camino(camino)
19:    return distancia_a_nodo_final
```

Implementando este algoritmo sin más, podemos obtener `15` puntos de la primer subtarea ($k = 1$), pero para resolver el problema completo vamos a tener que generalizar el algoritmo de Dijkstra.

Como dijimos, la optimización en la que se basa Dijkstra es explorar solamente el camino óptimo que pasa por cada nodo. Pero al tratar de hacer esto en nuestro problema estamos descartando información importante acerca de las demás formas de llegar a ese nodo. Por esto, debemos explorar, por cada nodo, los k caminos más cortos que pasan por el mismo.

Implementativamente, esto es sólo una pequeña modificación sobre el dijstra común y corriente. En lugar tener un arreglo que nos dice si un nodo ya fue explorado por el algoritmo, debemos mantener un arreglo que nos diga cuántas veces fue explorado cada nodo, y seguimos explorando caminos por cada nodo hasta que este haya sido explorado k veces.

Además, para devolver la respuesta final, debemos anotar la longitud de cada camino que exploramos que termina en el nodo final. La implementación podría quedar así:

Algorithm 4 Dijkstra modificado, que resuelve el problema

```
1: function DIJKSTRA_MODIFICADO(GRAFO ARR. DE ARISTAS, NODO_INICIAL ENTERO, NODO_DESTINO ENTERO, K ENTERO)(ARR. de ENTEROS)
2:     veces_visitado ← [0, ..., 0]
3:     candidatos ← {}
4:     camino_inicial.longitud ← 0
5:     camino_inicial.nodo ← nodo_inicial
6:     candidatos.añadir_nuevo_camino(camino_inicial)
7:     distancias_a_nodo_final ← []
8:     while no candidatos.vacío() do
9:         longitud, nodo ← candidatos.camino_más_corto()
10:        candidatos.borrar_camino_más_corto()
11:        if visitado[nodo] <k then
12:            veces_visitado[nodo] ← veces_visitado[nodo]+1
13:            if nodo es nodo_destino then
14:                distancias_a_nodo_final.poner_al_final(longitud)
15:            for peso, hijo en grafo[nodo] do
16:                camino.longitud ← peso+longitud
17:                camino.nodo ← hijo
18:                candidatos.añadir_nuevo_camino(camino)
19:    return distancias_a_nodo_final
```

La complejidad de este algoritmo es $\mathcal{O}(Mk \cdot \log Mk)$, ya que cada nodo se revisa k veces, y por cada una de estas veces se revisan, entre todos los nodos, un total de M aristas, y cada una de estas conlleva a una operación de "añadir nuevo camino", que toma $\mathcal{O}(\log$ `candidatos.size()`$)$, y `candidatos.size()` está acotado a su vez por la cantidad de inserciones (que es, como ya dijimos, $\mathcal{O}(Mk)$)

Esta complejidad es suficiente para resolver el problema sin restricciones.

2.2.2. Problema 2: Secuenciando el ADN [secuenciando]

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

El problema nos pide descubrir una secuencia secreta de la que solo sabemos la longitud N y las posibles letras que la componen. Para ello podemos hacer preguntas sobre si una secuencia es subsecuencia de la secreta a lo que nos responderán sí o no. El objetivo es descubrir la secuencia secreta usando pocas preguntas.

Ls primera idea de la solución es darnos cuenta que podemos saber la cantidad de veces que aparece una letra con pocas preguntas. Por ejemplo, si preguntamos por la secuencia $aaaaaaaaaa$ de 10 letras a, la respuesta es positiva si la secuencia secreta tiene al menos 10 letras a y negativa si hay menos de 10. Esto nos dice que podemos hacer una busqueda binaria para saber cuantas a aparecen en la secuencia secreta.

Si queremos saber si hay al menos k letras a, preguntamos por la secuencia de k letras a consecutivas. Y esto funciona para cualquiera de las letras de la palabra. Por lo tanto en $log(N)$ preguntas podemos descubrir la cantidad de veces que aparece una letra.

Con esta idea podemos resolver la primera subtarea, ya que $log(N)$ es a lo sumo 10. Por lo que con 10 preguntas podemos averiguar la cantidad de a que hay y sabemos que el resto de las letras tienen que ser c y como todas las a vienen antes que las c, queda determinada la secuencia secreta. Por ejemplo, si sabemos que la longitud de la secuencia secreta es 10 y con las preguntas averiguamos que hay 4 letras a, las otras 6 tienen que ser c y la secuencia es *aaaacccccc*.

Para el resto de las subtareas se complica porque si bien podemos saber cuantas veces aparece cada letra, tenemos que descubrir como se intercalan las distintas letras de la secuencia. La segunda idea es que si sabemos cuantas letras a y cuantas b hay, por ejemplo, con algunas preguntas más podemos averiguar como se intercalan. La idea será ir averiguando cuantas a aparecen antes de todas las b y cuantas hay entre cada dos b que aparecen. Supongamos que hay 4 letras a y 6 letras b, y la palabra secreta es *abbaabbabb*, la estrategia para averiguarla será la siguiente:

- Preguntamos por la secuencia *abbbbbb*, una letra a y todas las b. Nos dicen que sí es porque hay una a que aparece antes que todas las b.
- Preguntamos por la secuencia *aabbbbbb*, agregando una a al principio. Nos dicen que no porque hay sólo una letra a antes que todas las b.
- Preguntamos por la secuencia *ababbbbb*, como nos dijeron que no, sacamos la a y la ponemos despues de la primera b. Nos dicen que no porque la próxima a aparece despues de la segunda b.
- Preguntamos por la secuencia *abbabbbb*, ahora nos responden que sí.
- Preguntamos por la secuencia *abbaabbbb*, agregando una a despues de la última que pusimos. Nos vuelven a decir que sí.
- Y así siguiendo, cada vez que nos responden que sí, agregamos una a más a la derecha de la última a que pusimos y cada vez que nos dicen que no sacamos la última a y la movemos después de la próxima b.

Notar que no hace falta preguntar cuantas a aparecen después de la última b porque sabemos cuantas a hay en total y por lo tanto al final tienen que aparecer todas las a que faltaron. Además, si ya ubicamos todas las a, no hace falta seguir preguntando,

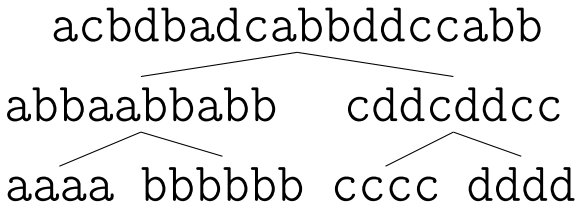
sabemos que todas las que siguen son b. Si la a aparece n_a veces y la b aparece n_b, el proceso termina:

- O bien cuando ubicamos todas las a. Notar que ubicamos una a cada vez que la respuesta es sí. Es decir cuando tenemos n_a respuestas afirmativas.
- O bien cuando ya averiguamos las ubicaciones de todas las a que aparecen antes de la última b. Como avanzamos una b cada vez que nos responden que no, esto quiere decir que nos dieron n_b respuestas negativas.

Esto quiere decir que como mucho hacemos $n_a + n_b - 1$ preguntas, ya que si hicimos esa cantidad de preguntas o bien tuvimos n_a respuestas afirmativas o n_b respuestas negativas.

Con esto podemos resolver las siguientes dos subtareas. Con $log(N)$ preguntas (a lo sumo 10) averiguamos cuantas g aparecen y con esto deducimos cuantas t aparecen. Luego, si la g aparece n_g veces y la t aparece n_t veces, con $n_g + n_t - 1$ preguntas más descubrimos la secuencia secreta. Notar que $n_g + n_t - 1 = N - 1$, por lo que hacemos a lo sumo 999 preguntas más.

Las siguientes subtareas pueden resolverse con esta misma técnica, pero utilizándola para "fusionar" distintas palabras parcialmente encontradas. En el caso anterior, usábamos la búsqueda binaria para tener como palabras iniciales *aaaa* y *bbbbbb*, y luego con la técnica de ir intercalando las a en diferentes posiciones sucesivas, podíamos fusionarlas con un costo total n_a+n_b-1, obteniendo la secuencia *abbaabbabb*.



Si la palabra original en cambio hubiera sido *acbbacabbccabb*, hubiéramos obtenido la misma palabra anterior al fusionar las a y las b, pero además sabríamos que la palabra tiene 4 letras c. Con lo cual, podemos hacer un paso más análogo al anterior de ir probando donde insertar la siguiente letra, para fusionar *abbaabbabb* con *cccc*. Como antes, una fusión entre dos palabras de longitudes l_1 y l_2 cuesta $l_1 + l_2 - 1$.

Si vamos fusionando todas las letras de a dos en forma de árbol binario balanceado, al estilo del algoritmo de Mergesort, se puede corroborar que el total

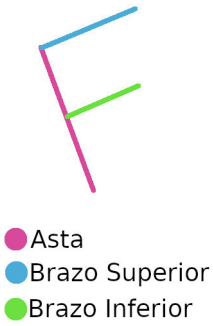
final de preguntas a realizar durante las fusiones no excede nunca $N \lg N$ preguntas (ya que al igual que en mergesort: hay $\lg N$ niveles de fusiones, y el costo total en cada nivel es $O(N)$). Esto es suficiente para resolver el problema.

Una estrategia aún mejor en casos con cantidades de letras desbalanceadas es fusionar siempre las dos palabras más cortas (lo cual tiene relación con los códigos de Huffman), pero no era necesaria esta optimización para resolver el problema (y en el peor caso, que ocurre cuando todas las letras aparecen la misma cantidad de veces, no se gana nada comparado a la solución balanceada anterior).

2.2.3. Problema 3: Buscando la F [buscandof]

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

El problema nos pide, dadas las coordenadas de ciertos puntos en el plano, hallar la máxima cantidad de estos puntos que pueden pertenecer a una efe.

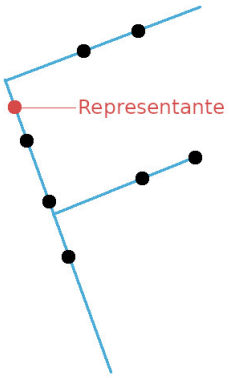


Para este problema, una posible solución consiste en iterar sobre todas las posibles efes, pero el problema es que hay infinitas de estas, por lo que tenemos que hallar una forma de quedarnos solo con "las que nos interesan".

Para esto, hay que caracterizar de alguna forma todas las efes que vayan a ser candidatos a ser efes óptimas.

Podemos comenzar, por ejemplo, con las efes que tienen dos puntos en su asta. De estos puntos, podemos tomarnos el punto que está más cerca del brazo superior de la efe, y utilizarlo como "representante" de la efe, junto con la dirección del asta.

Para seguir con esta idea, supongamos que nos dicen este punto "representante" de la efe, y la dirección en la que va su asta. ¿Podemos encontrar cuál de todas las efes que tienen esas caracteríaticas es la óptima?

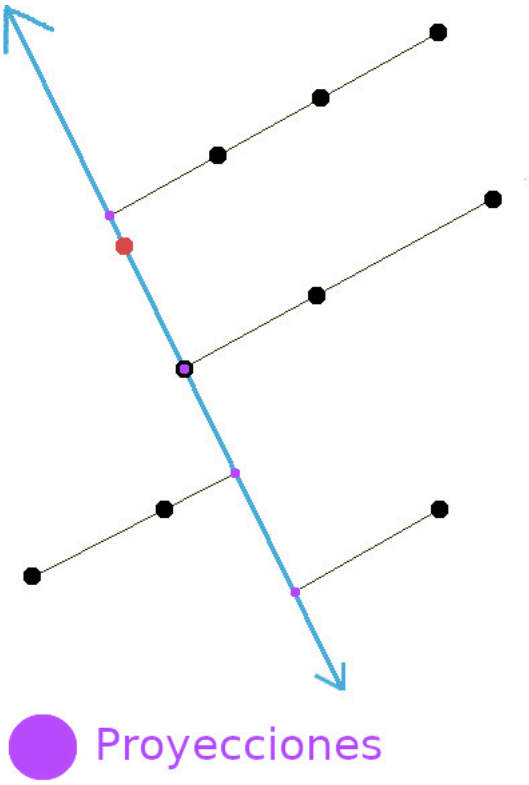


Inmediatamente podemos descartar todos los puntos del lado "incorrecto" (del lado para el que no va la efe), y también los puntos en la recta del asta que están más allá que nuestro representante, ya que, por como definimos representante, este es el que está más cerca del brazo superior.

Para optimizar qué puntos elegimos de los que nos

quedan, es necesario una observación clave: si sabemos
la dirección del asta de la efe, para que dos puntos estén en el mismo brazo de la misma, *deben tener la misma proyección hacia el asta.* (Ver nota al final).

Podemos entonces agrupar los puntos en base a su proyección, y de todos esos grupos, vamos a querer los que tengan más puntos. Pero no podemos simplemente elegir los dos que tengan más: para que nuestra efe sea válida, nuestro representante no puede "sobresalir" de la efe, es decir, tiene que estar antes de la proyección de uno de los brazos de la efe.



Teniendo en cuenta esto, si sabemos todas las proyecciones hacia el asta, y sabemos cuantos puntos comparten cada proyección, encontrar la cantidad óptima de puntos es fácil: para el brazo que tiene que estar por sobre el representante, nos tomamos la proyección que más puntos tenga que esté arriba del representante, y para el otro brazo nos tomamos la que tiene más puntos de las que sobraron.

Todo este proceso (encontrar la efe óptima dado el par (representante, dirección)), se puede hacer en tiempo $\mathcal{O}(n)$.

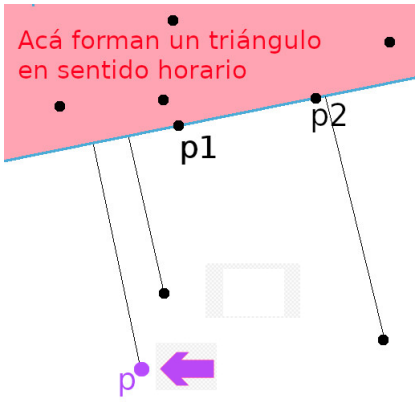
Ahora el problema reside en encontrar todos los posibles pares (representante, dirección) para probar. Una primera aproximación puede ser tratar de ver todas las efes que tengan al menos dos puntos en su asta: si sabemos su representante, sabemos que la dirección de su asta es en dirección de algún otro de los puntos del input. Podemos probar entonces hacer que cada uno de los n puntos sea un representante, y por cada uno de esos, nos tomamos las $n-1$ direcciones posibles (a cada uno de los otros puntos). Con esto iteraríamos un total de $\mathcal{O}(n^2)$ pares (representante, dirección), y cada uno cuesta $\mathcal{O}(n)$ procesarlo, por los que la complejidad queda $\mathcal{O}(n^3)$.

Ahora solo nos queda ver las efes con solo un (o cero) puntos en su asta. Notemos que podemos asumir, sin pérdida de generalidad, que toda efe tiene algún

punto en su asta, ya que si tuviese cero, podríamos "correr" el asta hasta que toque uno de los puntos de los brazos, y ahí tendría uno.

Vamos a hacer lo siguiente: por cada par de puntos p_1, p_2, vamos a construirnos un nuevo candidato (representante, dirección), y con estos vamos a recorrer todas las efes con un solo punto en su asta.

Dados los puntos p_1, p_2, vamos a tomarnos el punto p que tenga proyección sobre la recta $\overline{p_1, p_2}$ lo más hacia el lado de p_1 posible, pero tal que p_2, p_1 y p formen un triángulo en sentido antihorario. Luego vamos a recorrer las efes con el par (p, dirección perpendicular a $\overline{p_1 p_2}$).



Intuitivamente, estamos tratando de obtener la efe óptima que contenga a p_1 y a p_2 en uno de sus brazos, por lo tanto la dirección de su asta tiene que ser la perpendicular a $\overline{p_1 p_2}$.

Además, nos tomamos el punto "más hacia el lado de p_1", ya que esto nos da más espacio para poner más puntos en los brazos: Si teníamos una efe que era óptima, y podemos correr su asta hasta otro punto hacia el lado opuesto de los brazos, entonces seguirá siendo óptima.

La condición de que el triángulo sea antihorario no es más que decir "que p esté del lado correcto de la recta $\overline{p_1 p_2}$", como para que cuando construyamos la efe, el punto p no sobresalga por encima de la misma.

En total volvimos a tener $\mathcal{O}(n^2)$ pares (representante, dirección), y cada uno (igual que antes), nos cuesta $\mathcal{O}(n)$ tiempo procesarlo, por lo que esta segunda parte del algoritmo es también $\mathcal{O}(n^3)$.

Nota sobre proyecciones: aunque calcular proyecciones explícitamente es algo que se puede hacer, en este problema (como sólo queremos ver si dos puntos tienen la misma proyección), es más fácil utilizar el *producto interno*.

Dado un vector v, el producto interno de un punto con ese vector es muy muy fácil de calcular dadas las coordenadas, y nos dice básicamente "cuánto para allá" está el punto (donde "allá" es la dirección del vector v). Utilizando esto, que dos puntos tengan la misma proyección sobre la recta que estamos mirando es equivalente a que

tenga el mismo producto interno con el vector dirección del asta.

Si hacemos esto, una propiedad intereseante del producto interno es que se mantiene en enteros, lo que nos facilita el proceso de "agrupar de a proyecciones", ya que no tenemos que lidiar con las imprecisiones de los `double`s ni con fracciones.

Capítulo 3

Certamen Jurisdiccional

3.1. Nivel 1

3.1.1. Problema 1: Comprando rabanitos [rabanitos]

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

Este problema consiste en una correcta implementación de la descripción del enunciado. Vamos a recibir dos números, que llamaremos `precio_rabanin` y `precio_rabanon`. Con estos dos precios tenemos 3 casos posibles que debemos distinguir:

1. Si `precio_rabanin` $<$ `precio_rabanon`, entonces debemos devolver como respuesta `"RABANIN"`.

2. Si `precio_rabanin` $>$ `precio_rabanon`, entonces debemos devolver como respuesta `"RABANON"`.

3. Si `precio_rabanin` $=$ `precio_rabanon`, entonces debemos devolver como respuesta `"DA IGUAL"`.

En todos los casos *no debemos imprimir las comillas* (o sea, debemos imprimir una cadena de texto normalmente), y debemos tener el cuidado de *imprimir todas las letras en mayúscula*. Un pseudocódigo que resuelve el problema se ve de la siguiente forma.

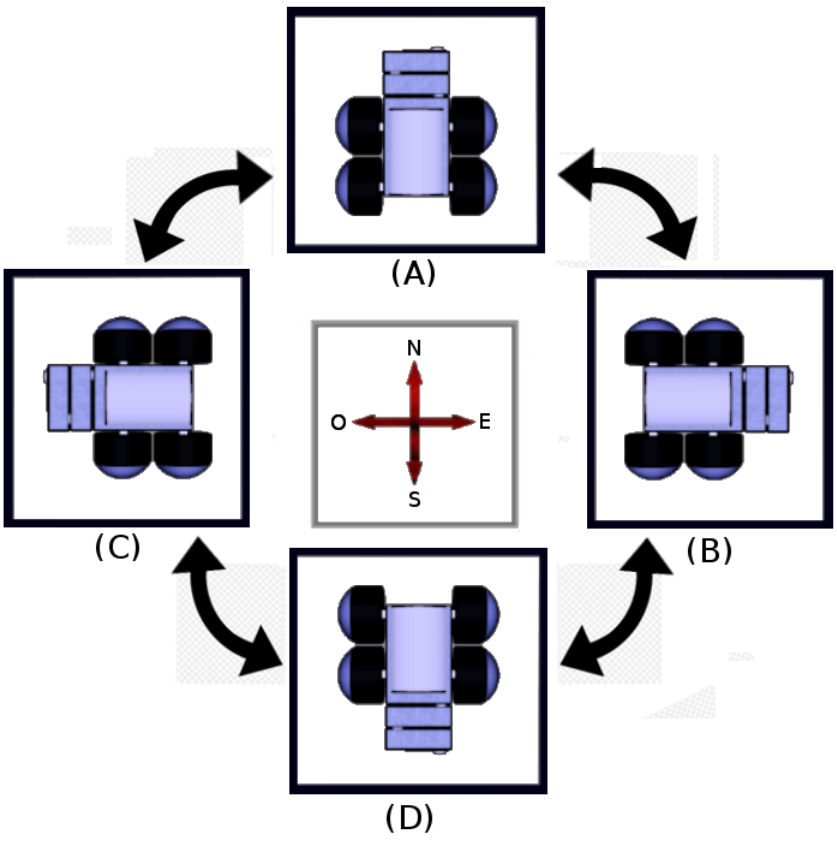
Algorithm 5 Solución al Problema 1 de Nivel 1 Jurisdiccional 2018 - rabanitos

```
1: function RABANITOS(PRECIO_RABANIN: ENTERO, PRECIO_RABANON: ENTERO)(PALABRA)
2:     if precio_rabanin < precio_rabanon then
3:         return RABANIN
4:     else if precio_rabanin > precio_rabanon then
5:         return RABANON
6:     else
7:         return DA IGUAL
```

3.1.2. Problema 2: Controlando al robot [robotito]

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

Originalmente tenemos un robot que mira hacia el norte. En la entrada nos dan una serie de comandos que indica si el robot gira en modo *horario* o *antihorario*. A continuación ilustramos la situación. El robot comienza mirando hacia el norte como en la figura **(A)**. En caso realizar un giro *horario* al comenzar (notado con la letra `'H'` en la entrada), alcanzaremos la situación de la figura **(B)**. En caso realizar un giro *antihorario* al comenzar (notado con la letra `'A'` en la entrada), alcanzaremos la situación de la figura **(C)**.



El robot aplicará la serie de comandos *secuencialmente en el orden que viene en la entrada*. Debemos responder *cuántos movimientos ocurren* la primera vez que el robot queda mirando hacia el sur (la situación **(D)** en la figura), en caso de que esto nunca ocurra debemos responder -1.

Hay *solo* 4 *situaciones o estados distintos*. Con este esquema en mente, surgen algunas observaciones que pueden hacerse y que facilitarán una correcta implementación del problema. Concretamente:

- En caso de realizar un giro *horario* y uno *antihorario* es lo mismo que no haber girado (volvemos a la situación original).
- Solo hay dos formas de llegar a la situación en la que miramos al sur (o bien viniendo del oeste o bien viniendo del este).
- La **diferencia entre giros horarios y antihorarios no puede ser mayor a 2**, pues de ser mayor a 2 necesariamente tuvimos que haber pasado por la dirección sur.

Veamos entonces cómo resolver el problema. Hay muchas formas de implementarlo. Siguiendo el hilo de esta explicación quizá la más natural sea tener dos contadores, `horario` y `antihorario` que cuenten la cantidad de giros que hicimos de cada tipo. En todo momento sabemos que $|\texttt{horario} - \textit{antihorario}| \leq 2$, y si es igual a 2 quiere decir que estamos mirando hacia el sur. Un pseudocódigo posible es el siguiente, se asume que `instrucciones` es una cadena de texto de largo `N`.

Algorithm 6 Solución al Problema 2 de Nivel 1 Jurisdiccional 2018 - robotito

```
1:  function ROBOTITO(INSTRUCCIONES: PALABRA) (ENTERO)
2:      horario ← 0
3:      antihorario ← 0
4:      total_instrucciones ← 0
5:      for i = 0 ... N-1 do
6:          total_instrucciones ← total_instrucciones + 1
7:          if instrucciones[i] == 'H' then
8:              horario ← horario + 1
9:          else          ▷ No hace falta ver si es 'A' porque solo hay dos opciones
10:             antihorario ← antihorario + 1
11:         if |horario - antihorario| == 2 then
12:             break
13:     if |horario - antihorario| == 2 then
14:         return total_instrucciones
15:     else
16:         return -1
```

3.1.3. Problema 3: Contando números escalonados [escalonados]

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

Lo que pide el problema es básicamente recorrer todos los números del 10 al N, y contar cuántos de ellos son escalonados. Algo que nos va a servir en este ejercicio, es *separarlo en dos partes*: Primero, pensar y programar la manera en que vamos a **chequear si un número dado es escalonado o no**. Luego, cuando ya tengamos esa parte resuelta, simplemente vamos a **recorrer los números**, y sumar uno a un contador que tendrá la respuesta al final si el número en cuestión es escalonado (para lo cual utilizamos el código ya escrito).

Ahora, ¿cómo podemos hacer un código que nos diga si un número es escalonado o no? Podemos pensar cómo haríamos nosotros "a mano": Vamos viendo dígito a dígito, empezando por el segundo, si es más grande que el anterior. Si alguno pasa que no, entonces no lo es. Si se cumple con todos los dígitos, entonces sí.

¿Cómo obtenemos el primer dígito? Bueno, no es tan fácil, hay que dividir al número por la potencia de 10 más grande que es menor al número (donde la división es la división entera, como cuando trabajamos con enteros en la compu). Por ejemplo, para el número 3780, si lo dividimos por 1000 (que es la potencia de 10 más grande que no se pasa), entonces obtenemos 3 como resultado, que es efectivamente el primer dígito. Pero para esto tenemos que calcular la potencia más grande de 10 que no se pasa.

No es tan difícil, y puede quedar como ejercicio pensarlo de esta manera, pero pensemos algo un poco distinto: ¿Qué pasa si en vez de ir de izquierda a derecha, vamos de *derecha a izquierda*? En vez de ver si el dígito que viene es mayor, ahora *debemos ver que sea menor* para que el número sea escalonado.

¿Y cómo obtenemos el último dígito de un número? Simplemente tomando el resto del número en la división por 10. Y cuando obtenemos el último dígito (y chequeamos lo que tenemos que chequear), para descartar este último dígito podemos tomar la división entera del número por 10, y listo, eliminamos el último dígito.

Entonces con esta idea tenemos lo necesario para hacer un código sencillo, que mira de derecha a izquierda los dígitos y compara el que está mirando con el anterior:

Algorithm 7 Cómo saber si un número es escalonado

```
1: function esEscalonado(x : ENTERO) (BOOLEANO)
2:     respuesta ← true
3:     ultimoDigitoVisto ← x % 10                    ▷ Tomamos módulo 10
4:     x ← x / 10                                    ▷ División entera, o sea ⌊x/10⌋
5:     while x > 0 do
6:         digitoActual ← x % 10
7:         if ultimoDigitoVisto ≤ digitoActual then
8:             respuesta ← false
9:         x ← x / 10
10:        ultimoDigitoVisto ← digitoActual
11:    return respuesta
```

Ahora que tenemos esa función, simplemente recorremos los números pedidos, y chequeamos si se cumple la condición en cada uno de ellos:

Algorithm 8 Solución Problema 3 Nivel 1 - Jurisdiccional - escalonados

```
1: function ESCALONADOS(N : ENTERO)(ENTERO)
2:     contador ← 0
3:     for i = 10 ... N do
4:         if esEscalonado(i) then
5:             contador ← contador + 1
6:     return contador
```

3.1.4. Problema 4: Lanzamiento de aceitunas [olivares]

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

Este problema es casi idéntico al problema `olivares2`, utilizado en nivel 3 (ver sección 3.3.2).

La única diferencia es que esta versión nivel 1 tiene cotas más bajas, y por lo tanto permite soluciones menos eficientes: por ejemplo, soluciones cuadráticas utilizando ordenamiento por burbujeo son suficientes para resolver esta versión.

Para un análisis detallado del problema, ver la versión en 3.3.2.

3.2. Nivel 2

3.2.1. Problema 1: Controlando al robot (compartido con nivel 1) [robotito]

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

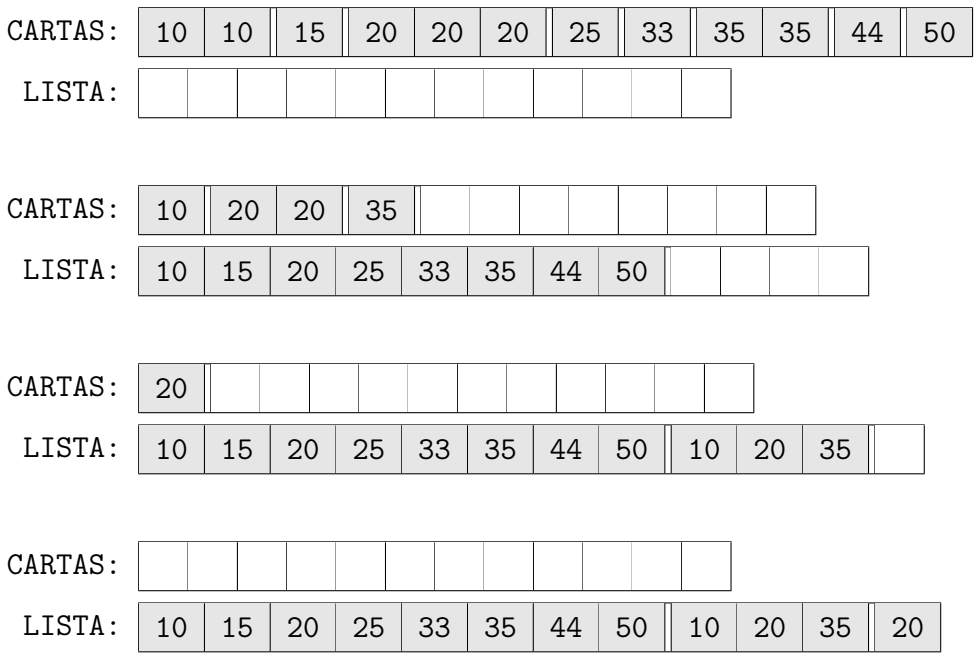
Este problema es exactamente igual al problema 2 del nivel 1 (ver 3.1.2).

3.2.2. Problema 2: Jugando al sortucho [sortucho]

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

En el problema hay tarjetas con un número en cada una. Primeramente se agrupan las cartas con un mismo número y se descartan la *mitad de las tarjetas redondeando hacia abajo*. Por lo tanto, si en un grupo de cartas con un mismo número había x cartas, para lo que resta del problema tendremos $\lceil \frac{x}{2} \rceil$ cartas con dicho número.

Una vez hecha esta simplificación, se toman los grupos de cartas y se ordenan por el número de tarjeta de *menor a mayor*. Finalmente, se genera una lista de números tomando las tarjetas restantes, sacando una carta de cada grupo (en orden de menor a mayor), y poniéndola al final de la lista. Este proceso se repite hasta que no queden cartas. En el ejemplo del enunciado, al comenzar este último paso se procede de la siguiente forma.



Un primer intento de solución podría consistir en simplemente simular el proceso que se describe en el enunciado. Si originalmente tenemos un arreglo `histograma` tal que `histograma[x]` nos dice cuántas tarjetas tienen el número x, lo primero que hacemos es descartar la mitad en cada grupo. Es decir, reemplazamos `histograma[x]` $\leftarrow \lceil \frac{\texttt{histograma[x]}}{2} \rceil$.

Veamos entonces cómo interpretar una entrada de N tarjetas para generar el arreglo `histograma`. Las líneas en azul marcan una variante que además devuelve el máximo valor almacenado en `histograma`. Aquí asumimos que `histograma` comienza inicializado con ceros y con un tamaño superior al máximo valor de una tarjeta (que llamamos `MAX_NUM`), se vería de la siguiente forma.

Algorithm 9 Análisis de Entrada - Prob. 2 de Nivel 2 Jurisdiccional 2018 - sortucho

```
1: function SORTUCHO_ENTRADA(TARJETAS: ARREGLO DE ENTEROS)(ENTERO)
2:     for i = 0 ... N-1 do
3:         histograma[tarjetas[i]] ← histograma[tarjetas[i]] + 1
4:     M ← −∞
5:     for x = 1 ... MAX_NUM do
6:         histograma[x] ← ⌈histograma[x]/2 ⌉
7:         M ← máx(M, histograma[x])
8:     return M
```

Recorriendo `histograma` en ***orden creciente***, debemos agregar un número a la lista por cada x con `histograma[x]` $\geq \mathbf{1}$. Finalmente deberíamos repetir este proceso con la salvedad de agregar un número a la lista por cada x con `histograma[x]` $\geq \mathbf{2}$, y así hasta el máximo valor que tenga almacenado `histograma`, que llamaremos $M = \max_{x \geq 0}$ `histograma[x]`, y que vimos cómo calcular más arriba (en azul).

Algorithm 10 Solución 1 al Problema 2 de Nivel 2 Jurisdiccional 2018 - sortucho

```
1: function SORTUCHO(TARJETAS: ARR. DE ENTEROS)( ARR. de ENTEROS)
2:     M ← sortucho_entrada(tarjetas)
3:     solucion ← []
4:     nivel ← 1
5:     for pasada = 1 ... M do
6:         for x = 1 ... MAX_NUM do
7:             if histograma[x] ≥ nivel then
8:                 solucion.agregar_al_final(x)
9:         nivel ← nivel + 1
10:    return solucion
```

Esta solución tal cual como está, si bien es correcta, es un tanto ineficiente. ¿En qué casos podríamos tener problemas?. Analicemos en detalle cuántas operaciones se realizan en cada caso. Al manipular la entrada en el primer algoritmo realizamos $\mathcal{O}(N + MAX\_NUM)$. En el segundo algoritmo por cada pasada de la primera iteración recorremos la totalidad de los MAX_NUM números, haciendo $\mathcal{O}(N \cdot MAX\_NUM)$ operaciones. Por lo tanto si en la entrada nos vienen 1,000,000 tarjetas todas iguales con el número 1,000,000 este algoritmo estaría haciendo alrededor de 10^{12} operaciones, lo cual es demasiado alto.

Como vimos, este algoritmo realiza tantas *pasadas* como el número de tarjeta que aparezca la mayor cantidad de veces. Ahora ¿Cuántos números pueden aparecer más de k veces a la vez? A lo sumo $\lfloor \frac{N}{k} \rfloor$ (de haber más nos pasamos de N números en total), por lo tanto recorrer todos los números en cada pasada es lo que lo hace ineficiente. Para ser concretos, en la primera pasada puede haber N números distintos, en la segunda $\lfloor \frac{N}{2} \rfloor$, en la tercera $\lfloor \frac{N}{3} \rfloor$.

Utilizando que $\sum_{k=1}^{N} \frac{N}{k} = N \cdot \sum_{k=1}^{N} \frac{1}{k} \sim \mathcal{O}(N \cdot \lg N)$, podríamos pensar que un algoritmo que en cada pasada solamente mira las tarjetas de las cuales todavía quedan números por pasar a la lista realiza alrededor de $N \cdot \lg N$ operaciones, pero implementado correctamente realiza solo $\mathcal{O}(N)$ operaciones, ¿por qué?. Si realizamos un *análisis amortizado* de la situación podemos ver que estamos realizando solo $\mathcal{O}(1)$ operaciones por cada tarjeta de la entrada, ¡pues estamos viendo cada tarjeta una sola vez!.

Ahora solo nos queda resolver el problema sabiendo que al mirar solamente las tarjetas que restan por poner en la lista no estaremos haciendo muchas operaciones. La idea clave consiste en **asignar a cada tarjeta la pasada en la cual será agregada a la lista**. Un posible algoritmo que realiza esta idea es el siguiente:

Algorithm 11 Solución 2 al Problema 2 de Nivel 2 Jurisdiccional 2018 - sortucho

```
1:  function SORTUCHO(TARJETAS: ARR. DE ENTEROS)(ARR. de ENTEROS)
2:      M ← sortucho_entrada(tarjetas)
3:      solucion_pares ← []                  ▷ Lista de pares {first,second}
4:      for x = 1 ... MAX_NUM do
5:          for nivel = 1 ... histograma[x] do
6:              solucion_pares.agregar_al_final({nivel,x})
7:      solucion_pares.ordenar()          ▷ Los pares se comparan lugar a lugar
8:      solucion ← []
9:      for {nivel,x} ∈ solucion_pares do
10:         solucion.agregar_al_final(x)
11:     return solucion
```

El problema de esta solución es que si bien solo recorremos $\mathcal{O}(N)$ tarjetas, al ordenar estamos realizando $\mathcal{O}(N \lg N)$ operaciones. Afortunadamente hay muchas formas de implementar la idea que vimos, veamos algunas más eficientes.

Una forma más eficiente que la anterior podría consistir en tener *dos listas*, la primera que comienza con todas las tarjetas distintas y la segunda vacía. Al realizar la primera pasada se agregan a la segunda lista de números solamente las cartas que tienen `histograma[x]` ≥ 2. Al finalizar la pasada se vacía la primera lista y se intercambian los roles entre ambas listas. Luego, basta repetir este procedimiento

(aumentando por `histograma[x]` $\geq 3, 4, \ldots$) hasta finalizar con ambas listas vacías. En cada paso estamos asegurándonos de recorrer en la próxima pasada solo las tarjetas que todavía faltan colocar.

Como dijimos, la idea es *asignar a cada tarjeta la pasada en la que será colocada* en la lista. Entonces, otra forma de implementarla consiste en tener una lista por cada pasada que contenga los números que serán colocados en esa pasada, para finalmente volcarlos en la lista `solucion`. Lo importante es que estos dos últimos enfoques realizan solo $\mathcal{O}(1)$ operaciones por cada número, obteniendo una complejidad de $\mathcal{O}(N)$ para resolver el problema. Un posible pseudocódigo de este último enfoque consiste en:

Algorithm 12 Solución 3 al Problema 2 de Nivel 2 Jurisdiccional 2018 - sortucho

```
1: function SORTUCHO(TARJETAS: ARR. DE ENTEROS)(ARR. de ENTEROS)
2:     M ← sortucho_entrada(tarjetas)
3:     solucion ← []
4:     pasada_num ← [[], ..., []]          ▷ Lista de listas con MAX_NUM lugares
5:     for x = 1 ... MAX_NUM do            ▷ Recorremos en orden creciente
6:         for nivel = 1 ... histograma[x] do
7:             pasada_num[nivel].agregar_al_final(x)
8:     for nivel = 1 ... M do
9:         for x ∈ pasada_num[nivel] do
10:            solucion.agregar_al_final(x)
11:    return solucion
```

3.2.3. Problema 3: Canción [cancion]

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

El problema nos pide encontar el "estribillo" de una string s dada en la input, que se define como *la mayor substring que se repite (al menos) dos veces, sin solaparse*

Para resolver este problema vamos a utilizar una estructura denominada *trie*, más precisamente, el *suffix trie* de nuestra string s.

Se puede leer más acerca de la estructura en http://wiki.oia.unsam.edu.ar/algoritmos-oia/trie, pero un trie es básicamente una estructura que guarda un conjunto de strings en forma de árbol: cada nodo del trie representa un *prefijo común* de algunas de las strings del conjunto (posiblemente una sola); es decir, el trie *fusiona los prefijos comunes* de las strings que están en el conjunto, haciéndolos un sólo nodo.

La estructura de trie es tal que si un nodo representa el prefijo p, luego su

padre es el representante del prefijo que se obtiene sacándole la última letra a p. Así, el único prefijo sin padre es el "prefijo vacío", y cumple la función de raíz del trie.

La estructura de suffix trie surge de poner en un trie todos los sufijos de una string dada (en nuestro caso, s). ¿Por qué nos sirve esto? Pues porque todas las substrings de s son el prefijo de alguno de los sufijos de s, por lo que los nodos de nuestro suffix trie se van a corresponder con los substrings de s.

Entonces, ahora logramos tener una estructura en forma de árbol que contiene todas las substrings de la string s, pero necesitamos más. Nos gustaría saber, para cada nodo del trie (que representa una substring):

1. cuál es su longitud (porque queremos la más larga), y
2. si se repite dos veces sin solaparse.

Para calcular si una cadena se repite sin solapar, es necesaria una observación clave. Dada una substring t, para ver si aparece dos veces sin solaparse, "conviene" tomar su primera y última aparición en s. Esto es así ya que estas son las dos apariciones de t que van a estar más lejos entre sí, por lo que si estas se solapan, todas las demás también se solaparán.

Además, si sabemos en qué índice de s comienzan la primera y la última aparición de una substring t, podemos calcular si estas se solapan o no: Como ya sabemos el tamaño de la substring t (por 1), basta con ver si la substring t cabe entre los dos índices, es decir, si la distancia entre la primera aparición y la última aparición es al menos la longitud de t.

Hay dos opciones para calcular estos tres valores (longitud, índice de primera aparición, índice de última aparición) en cada nodo del trie:

3.2.3.1. A) Guardar la información mientras se construye el trie

En este problema, como vamos a insertar todos los substring en el trie, utilizaremos dos fors para fijar primero la posición inicial del substring (i), y, por cada una de esas posibles posiciones iniciales, en un segundo for recorremos todas las posibles posiciones finales (j). La clave es que al recorrer así, cada substring que examinamos tiene exactamente un caracter más que la anterior, y entonces no vamos a insertarla en el trie desde cero, sino que simplemente avanzamos un nodo desde la posición donde quedamos tras procesar el substring anterior. Esto permite construir el suffix trie y a la vez recorrer explícitamente todos los substrings en forma sencilla, pero con complejidad cuadrática y no cúbica.

De esta manera, cuando un substring aparece muchas veces, al ir creando el trie recorremos su nodo varias veces, una por cada aparición. Y en el momento en que pasamos por ese nodo, conocemos (gracias a los índices i,j) el tamaño y la ubicación del substring en cuestión. Podemos entonces aprovechar para allí mismo anotar esta información en el correspondiente nodo. Más precisamente:

- Al pasar por un nodo, **siempre anotamos** su longitud, que será $j - i$, y **siempre pisamos** el valor almacenado de su última aparición con i (si barremos i crecientemente, el último valor con el que pisaremos será la última aparición).
- Al pasar por un nodo **por primera vez**, anotamos su primera aparición en i (si barremos i crecientemente, la primera vez que encontramos un nodo será en la primera aparición de ese substring).

3.2.3.2. B) Construir el trie, y luego recuperar la información del trie

La longitud de cada substring (nodo en el trie) es la más simple de las dos cosas a calcular. Dada una substring t de s, los antecesores del nodo de t en el árbol serán los prefijos sucesivos de t en orden decreciente de longitud, hasta llegar a la raíz que es el prefijo de longitud 0. Por esto, la longitud de t corresponderá con la prefundidad de t en el árbol, que se puede calcular haciando un `dfs` sobre el árbol.

Para calcular para cada substring su índice de primera y última aparición, hay que adentrarnos en la estructura del suffix trie. Cuando construimos el trie, algunos nodos se marcan como *nodos terminales*. Estos son los nodos que representan no cualquier substring de s, sino específicamente un sufijo (notar que las hojas siempre son nodos terminales, pero no necesariamente viceversa).

Todos los nodos terminales en el subárbol del nodo de una substring t representan todos los sufijos de s de los cuales t es prefijo, es decir, todas las apariciones de t en s.

Sabiendo esto, podemos guardar en cada nodo terminal el índice donde comienza su sufijo correspondiente (será simplemente n menos su longitud, pues todos los sufijos terminan donde termina todo el string). Luego, por cada substring t, si miramos el mínimo de esos índices de inicio para todos los nodos terminales en el subárbol del nodo correspondiente a t, obtenemos el índice de la primer aparición de t como substring de s.

Esta operación se puede realizar fácilmente con una dp sobre el árbol: en cada nodo queremos el mínimo sobre todos sus hijos. Además, análogamente se puede

obtener el *último* índice de aparición (tomando el máximo en vez del mínimo).

3.2.3.3. Solución final

Entonces, para cada substring t ya tenemos su índice de primer y última aparición, así como también su longitud. Ahora sólo basta con iterar sobre todas estas substrings, chequear que la primera y última aparición no se solapen, y tomar de todas estas la que tiene mayor longitud (Y en caso de no haber, imprimir `NO HAY`).

En términos de complejidad, esta solución es $\mathcal{O}(n^2)$ (n es la longitud de s), ya que insertar una string en un trie toma tiempo proporcional a su longitud, y estamos insertando todos los sufijos de la string s, que tienen suma de longitudes:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \mathcal{O}(n^2)$$

Por lo que, en particular, el trie tendrá también $\mathcal{O}(n^2)$ nodos.

Además, la complejidad de eventuales dfs para calcular la profundidad en el árbol, así como las dps para calcular apariciones, siempre es lineal en el tamaño del árbol, que es $\mathcal{O}(n^2)$.

Finalmente, iterar sobre todo el árbol para computar la substring de mayor longitud que se repite sin solaparse es de nuevo lineal en el tamaño del árbol, es decir $\mathcal{O}(n^2)$.

Luego la complejidad total del algoritmo es $\mathcal{O}(n^2)$, que con $n = 2000$ es suficiente.

3.2.4. Problema 4: Nuevas Autopistas [nautopistas]

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

Este problema, nos sugiere fuertemente que lo modelemos mediante grafos, al hablar de ciudades conectadas mediante autopistas que conectan ciudades. Recordemos que el objetivo en este problema es construir autopistas, de forma tal que el costo total sume un numero específico. Simplificamos la explicación al hablar de un solo número en lugar de una lista, pero al final veremos que dá lo mismo. Veamos como traducir cada parte del problema a una propiedad del grafo.

Tenemos N ciudades y N autopistas distintas que podemos construir. Cada una de las autopistas tiene un costo para construirse, y conecta en forma directa dos ciudades diferentes. Nuestro grafo tiene N nodos conectados mediante N aristas.

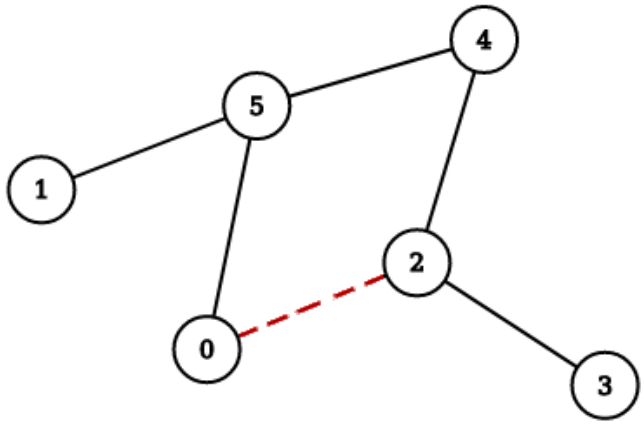
Cada arista conecta dos nodos distintos mediante un costo. Es un ¨grafo pesado¨

Es posible llegar desde cualquier ciudad a cualquier otra utilizando solo autopistas de la lista de posibles. Esto signficia que el grafo es conexo.

Queremos ver si podemos eliminar algunas autopistas de la lista, tal que las restantes sigan conectando todas las ciudades, y que además el costo total de construcción sea exactamente un cierto número F que le gusta al ministro. Entonces queremos tener un subgrafo del original que siga siendo conexo pero que además tenga costo total F. [1]

Ahora que tenemos este modelo de nuestro problema, podemos pensar propiedades que conocemos de la teoría de grafos. Sabemos que un Grafo de N nodos y N-1 aristas conexo es un árbol. ¿Qué ocurre con un grafo de N nodos y N aristas conexo? ¿Cómo es el grafo original? Podemos demostrar que tal grafo tiene **exactamente un ciclo**. Esto se debe a que podemos formarlo agregando una arista (u, v) a su árbol generador mínimo. Luego, como ya había un camino P entre u y v por ser conexo el árbol, P + (u, v) forma un ciclo. Y es el único pues el árbol no tenía ciclos.

Además tenemos que tener en cuenta que un grafo de N-2 aristas (donde N sigue siendo la cantidad de nodos) no es conexo. Demostración: Supongamos que tenemos un grafo de N-2 aristas conexo, al agregarle una arista cualquier, como sigue siendo conexo y tiene N-1 aristas, es un árbol. Pero como le agregamos una arista a un grafo conexo, tiene un ciclo. ¡Absurdo!



Concluimos entonces que a nuestro grafo original de N aristas le vamos a sacar

[1] Esto se relaciona con el problema de Árbol Generador Mínimo. Se puede leer más al respecto en http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/arbol-generador

0 ó 1 arista, de forma tal que se mantenga conexo.

Es fácil verificar si sacando 0 aristas logramos nuestro cometido. Simplemente chequeamos si la suma total de costos de autopistas (de acá en adelante T) es igual al número F.

Si el costo F es mayor a T, es imposible cumplirlo. Por último, si F es menor que T, el problema está en verificar si es posible lograrlo utilizando 1 arista.

Entonces buscamos una arista para sacar del grafo con las siguientes restricciones:

- Ser parte del ciclo.

- Tener costo T - F (que sabemos es positivo).

Tiene que ser parte del único ciclo, porque de lo contrario dejaríamos disconexo al grafo. Y tiene que tener costo $T - F$ para que al removerla nos quede un árbol de costo F. Así que vamos a tener que encontrar todas las aristas del ciclo, y verificar si alguna de ellas tiene costo exactamente $T - F$. Para eso vamos a usar el algoritmo de DFS: http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/dfs.

Durante la ejecución del DFS, recorremos el grafo de forma tal de apilar los nodos que vemos, hasta terminar de recorrer todas las ramas de sus hijos. Si en algún punto de ese proceso nos encontramos un nodo que ya está en la pila (nodo x), significa que recorriendo los hijos de x, ¡Volvimos a encontrar a x! Eso es exactamente un ciclo. Y el ciclo está formado por todas las aristas que están en la pila entre las dos apariciones del nodo x.

Algorithm 13 Solución al Problema 4 de Nivel 2 Jurisdiccional 2018 - autopistas

```
1: function AUTOPISTAS(G: ADYLISTA, NUMEROS: LISTA[ENTERO])(BOOLEANO)
2:     total ← sumarAristas(G)
3:     if total ∈ numeros then
4:         return true
5:     ciclo : Lista[Arista] ← encontrarCiclo(G)
6:     for arista : ciclo do
7:         if total − arista.costo ∈ numeros then
8:             return true
9: function SUMARARISTAS(G: ADYLISTA)(ENTERO)
10:    total ← 0
11:    for nodo : G do
12:        for arista : G[nodo] do
13:            total ← total + arista.costo
14:    return total
15: function ENCONTRARCICLO(G: ADYLISTA)(LISTA[ARISTA])
16:    ciclo ← []
17:    aristasEnStack ← []
18:    dfs(0, G, aristasEnStack, ciclo)
19:    return ciclo
20: function DFS(NODO: ENTERO, G: ADYLISTA, VISITADO: ARREGLO[BOOLEANO], ENSTACK: ARREGLO[BOOLEANO], ARISTASENSTACK: LISTA[ARISTA], CICLO: LISTA[ARISTA])(VOID)
21:    for arista: G[nodo] do
22:        visitado[nodo] ← true
23:        enStack[nodo] ← true
24:        if ¬visitado[arista.hasta] then
25:            aristasEnStack ← aristasEnStack + {arista.hasta}
26:            dfs(arista.hasta, G, aristasEnStack, ciclo)
27:            aristasEnStack ← aristasEnStack.pop()
28:        else if enStack[arista.hasta then
29:            ciclo ← aristasEnStack
30:        enStack[nodo] ← false
```

3.3. Nivel 3

3.3.1. Problema 1: Revisando el boletín [boletin]

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

Lo primero a resolver, es cómo hacer para ingresar varios números. Bueno, en realidad, no es "varios", porque para ingresar 10 números podríamos copiar y pegar 10 veces la línea para ingresar uno y listo (aunque *no es recomendable hacerlo*).

Cuando la cantidad total es variable, hay que hacer algo un poco más complejo.

Lo que hacemos es hacer un **for**, para que la computadora entre al "for" unas N veces, y adentro de cada ciclo, se ingrese un número. Hay que tener cuidado con esto porque, como el N es variable, no podemos declarar N variables y hacer "`ingresar var1`", "`ingresar var2`", ..., "`ingresar varN`", **¡porque no sabemos a priori cuánto vale** N**!**.

Entonces, vamos a guardar los datos ingresados en la misma variable. Para no perder la información de los números ingresados, vamos a necesitar usar el valor de alguna manera. Lo usual es almacenar el valor en un conjunto (`vector` en C++, `List` en Java), para al final de los N pasos tener todos los valores guardados. Pero en este caso no va a hacer falta, veamos por qué.

Para **calcular la suma** de un conjunto de números, podemos inicializar una variable en 0, e ir sumando uno por uno los números. En este ejericio, esto podemos hacerlo de dos maneras: O bien primero guardamos todos los números que nos dan, para luego recorrer esa lista e ir sumando de uno, o a medida que vamos leyendo los números, podemos ir sumando sus valores a la variable que inicializamos en 0.

Para **calcular el promedio**, dado que el promedio es la suma total dividido la cantidad de números sumados, una vez que tenemos la suma, y habiendo guardado el número incial N, simplemente debemos hacer la división $\lfloor \frac{SUMA}{N} \rfloor =$ `suma/N`.

Para **saber si aprueba o no**, además del promedio (que vimos antes cómo calcular), debemos guardar el último número ingresado. Si guardamos todos los números en una lista esto es fácil, accedemos a la última posición de la lista y nos fijamos si es más grande que 7 o no. Aquí tendremos que usar un **if** con dos condiciones, la del promedio y la de la última nota. Para eso usamos "`&&`" que es un "Y", es decir que queremos que el promedio sea mayor o igual a 7 *y además*, que la última nota sea mayor o igual a 7.

También podemos hacerlo con un **if** adentro del otro, aunque va a ser más molesto porque vamos a tener que escribir dos **else**, uno para el primer *if*, ya que si la primera condición no se cumple, queremos ir al *else* para imprimir que "NO" aprueba, pero además, si pasa la primera condición pero no la segunda, también queremos imprimir "NO", entonces debemos hacer otro comando *else*.

Si no tenemos la lista de los números, como al ingresar datos lo hacemos uno por uno, en la variable que sea (por ejemplo, si guardamos cada número en una misma variable "x"), nos quedó justo en esa variable el último valor ingresado al final.

Para **saber cuál es la mejor nota**, la manera de hacerlo es la siguiente: Inicializar una variable con el valor -1 (ya que todas las notas son entre 0 y 10), y a

medida que nos ingresan los números, nos fijamos, si el valor ingresado es mayor al que tenemos (de ser así hemos encontrado una nota mejor a la veníamos guardando), entonces actualizamos el valor con el ingresado recién. Al final, tendremos en esta variable la mejor nota de todas. Por eso empezamos con un -1, porque por más que sean todos ceros, ya la primera es mayor a lo que empezamos guardando, y entonces vamos a actualizar a nuestra variable con esta primera nota.

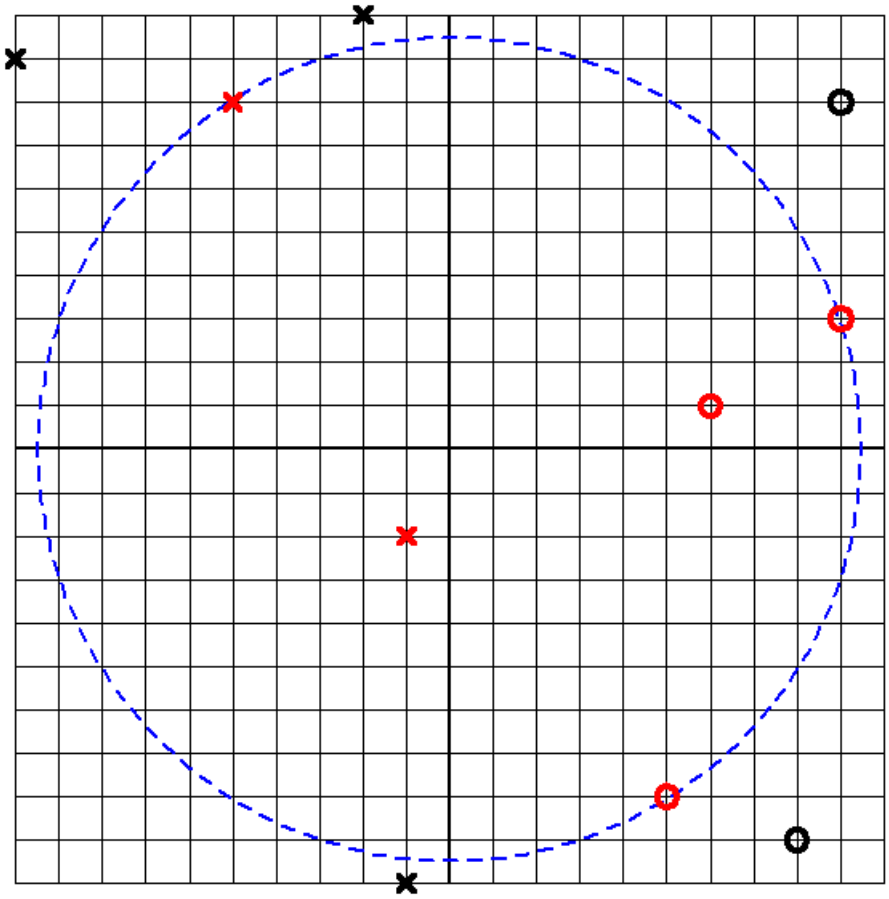
A continuación se muestra un código en C++ que además de resolver el problema sirve de ejemplo sobre cómo usar el **for** para ingresar N números.

```
#include<iostream>

using namespace std;

int main(){
    int N;
    int suma=0, mejorNota=-1;
    int dato;
    cin>>N;
    for(int i=0; i<N; i++){
        cin>>dato; // Guardamos el valor que se ingresa en la variable dato.
                   // Esta variable va a ir tomando todos los valores ingresados,
                  // y al final contendrá el valor del último número.
        suma = suma + dato;
        if(dato > mejorNota){
            mejorNota = dato;
        }
    }
    int promedio = suma/N;
    cout<<suma<<endl;
    cout<<promedio<<endl;
    if(promedio >= 7 && dato >= 7){
        cout<<"SI"<<endl;
    }else{
        cout<<"NO"<<endl;
    }
    cout<<mejorNota<<endl;
}
```

3.3.2. Problema 2: Lanzamiento de aceitunas [olivares2]

http://juez.oia.unsam.edu.ar/#/task/olivares2/statement

En el problema se nos pide encontrar el *valor del radio* (en realidad de su cuadrado) que maximiza la diferencia entre los tiros del jugador estrella del Oli y su archirival que caen adentro de la cancha. Para un cierto tiro realizado en (x, y) nos va a interesar saber para qué valores de R^2 el tiro estará adentro o afuera de la cancha, ¿Cómo podemos hacerlo?¿Cuál es el menor valor de R^2 para que el tiro esté adentro?.



El menor valor posible de R para que el tiro esté adentro está dado por la distancia del punto (x, y) al centro de la cancha (de esa forma la circunferencia que delimita la cancha pasa exactamente por el punto (x, y)). Esta distancia la podemos calcular por pitágoras, pues corresponde a la hipotenusa del triángulo recto con base x y altura y. Concluimos que R satisface la ecuación $R^2 = x^2 + y^2$. Para radios mayores, el tiro también caerá adentro de la cancha.

Por lo tanto fijado el valor de R^2, un lanzamiento en (x, y) se lo considera adentro si $x^2 + y^2 \leq R^2$ (notar que incluimos el igual, pues el enunciado dice que

línea es adentro). De cada punto solo nos interesa el valor de $x^2 + y^2$, primero que nada vamos a generar el arreglo `distancias` que almacenará el valor de x^2+y^2 para los lanzamientos del Oli y $-x^2 - y^2$ para los lanzamientos de su rival.

Algorithm 14 Entrada al Problema 3 de Nivel 2 Jurisdiccional 2018

```
1: function OLIVARES_ENTRADA(coordX: ARR. DE ENTEROS, coordY: ARR. DE ENTEROS)( ARR. de ENTEROS)
2:     distancias ← []
3:     for i = 0 ... N-1 do
4:         radio_cuadrado ← coordX[i]*coordX[i] + coordY[i]*coordY[i]
5:         if coordX[i] < 0 then
6:             radio_cuadrado ← -radio_cuadrado
7:         distancias.agregar_al_final(radio_cuadrado)
8:     return distancias
```

Una primera idea consiste en iterar todos los radios posibles, luego para cada radio fijo se pueden calcular cuántos tiros del lanzador del Oli y su archirival están adentro de la cancha. Finalmente bastaría con calcular la diferencia, y tomar aquel menor radio que maximice la diferencia buscada.

Un pseudocódigo que implemente esta idea es el siguiente. Aquí `MAX_RADIO` refiere al máximo valor posible de $x^2 + y^2$ dado por las cotas del enunciado (lo cual tendrá distinta relevancia dependiendo de la subtarea). La cantidad de operaciones que realiza este algoritmo será del orden de $\mathcal{O}(N \cdot MAX\_RADIO)$.

Una optimización que podemos hacer de manera relativamente sencilla radica en la observación de que *no hace falta probar todos los radios posibles*, probando solamente los distintos valores de $x^2 + y^2$ alcanza (pues en los valores intermedios no varía la cantidad de lanzamientos adentro o afuera).

En este último caso obtendríamos una complejidad temporal de $\mathcal{O}(N^2)$, y el único cambio corresponde a la línea 5 (debemos reemplazarla por el comentario en azul), y las líneas 9 y 11, donde debemos tomar módulo en `r_cuad` para tener en cuenta que podrían ser lanzamientos del rival.

Algorithm 15 Solución 1 al Problema 3 de Nivel 2 Jurisdiccional 2018 - olivares

```
1: function OLIVARES(COORDX: ARR. DE ENTEROS, COORDY: ARR. DE ENTEROS)(ENTERO)
2:     distancias ← olivares_entrada(coordX,coordY)
3:     max_dif ← 0  ▷ Siempre podemos obtener una diferencia de 0 con radio 0
4:     r_cuad_respuesta ← 0
5:     for r_cuad = 0 ... MAX_RADIO do          ▷ for r_cuad ∈ distancias do
6:         oli ← 0
7:         rival ← 0
8:         for i = 0 ...  N-1 do
9:             if distancia[i] > 0  and distancia[i] ≤ |r_cuad| then
10:                oli ← oli + 1
11:            else if distancia[i] < 0  and -distancia[i] ≤ |r_cuad| then
12:                rival ← rival + 1
13:        if  oli - rival > max_dif then
14:            max_dif ← oli - rival
15:            r_cuad_respuesta ← r_cuad
16:        else if oli-rival == max_dif and r_cuad < r_cuad_respuesta  then
17:            r_cuad_respuesta ← r_cuad          ▷ Nos piden el menor radio posible
18:    return r_cuad_respuesta
```

En el caso de nivel 1, esta última variante del algoritmo visto alcanza para obtener la totalidad de los puntos (teniendo cuidado con el *overflow* utilizando `long long` en C++ por ejemplo). Pero en el caso de nivel 3 tenemos que $N \leq 100{,}000$ y un algoritmo de complejidad $\mathcal{O}(N^2)$ no terminará de correr en el tiempo deseado.

Veamos entonces cómo resolver el problema en este último caso. La idea será aprovechar que si **vamos viendo los lanzamientos en orden** (por distancia al centro), entonces solamente se van agregando lanzamientos y en todo momento podemos ir manteniendo la diferencia entre ambos jugadores.

El mayor cuidado a la hora de implementar esta idea es cuando hay *varios lanzamientos con la misma distancia al centro, debemos incluir a todos a la vez.* Una correcta implementación de esta idea tiene una complejidad de $\mathcal{O}(N \lg N)$, dado que se requiere ordenar `distancias` por valor absoluto.

Algorithm 16 Solución 2 al Problema 3 de Nivel 2 Jurisdiccional 2018 - olivares

```
1: function OLIVARES(COORDX: ARR. DE ENTEROS, COORDY: ARR. DE ENTEROS)(ENTERO)
2:     distancias ← olivares_entrada(coordX,coordY)
3:     distancias.ordenar()        ▷ En orden creciente por valor absoluto
4:     max_dif ← 0  ▷ Siempre podemos obtener una diferencia de 0 con radio 0
5:     r_cuad_respuesta ← 0
6:     oli ← 0
7:     rival ← 0
8:     for i = 0 ... N-1 do                    ▷ Notar que las visitamos en orden
9:         if distancia[i] > 0  then
10:            oli ← oli + 1
11:        else if distancia[i] < 0  then
12:            rival ← rival + 1
13:        if i == N-1 or |distancia[i]| ≠ |distancia[i+1]| then ▷ Si falta agregar lanzamientos con la misma distancia o bien ya terminamos...
14:            if oli - rival > max_dif then        ▷ Si mejora la respuesta...
15:                max_dif ← oli - rival
16:                r_cuad_respuesta ← r_cuad
17:    return r_cuad_respuesta
```

Esta es solo una forma de implementar esta última idea. Otra opción posible (para agregar a todos los lanzamientos que están a la misma distancia del centro a la vez) es tener asignado para cada distancia cuántos lanzamientos hay del lanzador del oli y cuántos de su rival (con `map` en C++ por ejemplo).

3.3.3. Problema 3: Bolñitsy góroda [hospitales]

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

Breve abstracción: dado un grafo conexo, simple y ponderado; donde algunos vértices están marcados como especiales (hospitales), responder consultas de la forma x, z con tres valores y, d, c: el hospital y más cercano a x, la distancia óptima del camino $x \rightsquigarrow y \rightsquigarrow z$, y la cantidad de formas de hacer dicho camino.

Vista la gran cantidad de consultas que nos puedan hacer, es preferible tener precomputado para cada sitio su hospital más cercano, junto a la distancia y cantidad de maneras de realizar un camino óptimo desde un sitio a cualquier otro vértice.

Este problema es clásico en grafos: *hallar la distancia más corta para todo par de vértices*. En este enunciado, pedimos también contar la cantidad de caminos mínimos.

Existe una solución conocida que utiliza *programación dinámica*, llamada **Algoritmo de Floyd-Warshall**. Se puede leer más sobre este algoritmo en http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/floyd-warshall.

`dp(i, j, k)`: distancia más corta de i a j, utilizando como vértices intermedios $\{0..k\}$

Para $k = -1$, no podemos usar vértices intermedios, por lo que la expresión es simplemente:

$$dp(i, j, -1) = \begin{cases} d_{ij}, & \text{si hay una arista con peso } d_{ij} \text{ entre } i \text{ y } j \\ \infty, & \text{en caso contrario} \end{cases}$$

Para $k \geq 0$, o bien un camino óptimo utiliza el vértice k o bien no lo hace, de donde obtenemos:

$$dp(i, j, k) = \text{mín} \begin{cases} dp(i, k, k-1) + dp(k, j, k-1) & \text{(pasa por k)} \\ dp(i, j, k-1) & \text{(no pasa por k)} \end{cases}$$

Observación 1: Notemos que en el primer caso del mín, restamos uno al tercer argumento porque un camino óptimo no pasa dos veces por el mismo vértice (k).

Observación 2: Podemos calcular esta DP en orden creciente de k, para utilizar sólo $O(N^2)$ en memoria, en lugar de $O(N^3)$.

Una vez calculados y almacenados estos valores en un array `D[i][j]`, `D[i][j] = dp(i, j, N-1)`, podemos rápidamente iterar sobre los pares y hallar para cada sitio, el hospital más cercano.

Siguiendo una lógica similar, podemos contar la cantidad de caminos óptimos:

$$dp2(i, j, -1) = \begin{cases} 1, & \text{si hay una arista entre } i \text{ y } j \\ 0, & \text{en caso contrario} \end{cases}$$

$$dp2(i, j, k) = f(i, j, k) + g(i, j, k)$$

Donde f indica la cantidad de caminos óptimos que pasan por k, y g la cantidad de caminos óptimos que no lo hacen.

$$\text{f}(i, j, k) = \begin{cases} dp2(i, k, k-1) \times dp2(k, j-1, k-1) & \text{si un camino óptimo pasa por k} \\ 0 & \text{en caso contrario} \end{cases}$$

Se puede verificar si un camino óptimo pasa por k o no simplemente verificando la igualdad:

$$dp(i, j, k) == dp(i, k, k-1) + dp(k, j, k-1)$$

¡ya que un camino óptimo se compone de subcaminos óptimos!

Análogamente,

$$\mathrm{g}(i, j, k) = \begin{cases} dp2(i, j, k-1) & \text{si un camino óptimo no pasa por k} \\ 0 & \text{en caso contrario} \end{cases}$$

Condición que puede verificarse con:

$$dp(i, j, k) == dp(i, j, k-1)$$

Esto nos da una solución con complejidad:

$O(N^3 + M + Q)$ en tiempo,
$O(N^2)$ en memoria,

Que cumple ampliamente los límites del problema.

3.3.4. Problema 4: Tateti Zero [tatetizero]

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

Este es un problema bastante complejo. Primero veamos cómo resolver la subtarea. Decía que en un subconjunto de casos de prueba, habría *una sola casilla vacía*. Pensemos en resolver únicamente este problema: Hay un tablero de 3×3, con cuatro cruces y cuatro círculos ubicados de manera que no hay ningún tatetí. Como empieza el círculo, sabemos que *la novena jugada la hará quien juegue círculos*. Entonces, podemos simplemente **colocar un círculo en la casilla vacía, y ver si hay tatetí**.

¿Cómo hacemos para saber si en un tablero hay alguien que ganó? Simplemente chequeando en todas las posibles direcciones, si las 3 casillas consecutivas tienen la misma letra.

Para esto entonces, tenemos que decidir cómo guardar el tablero, para luego poder acceder/consultar el valor de una casilla. Una manera es una lista de lista de caracteres. Donde la primera lista de listas contiene los 3 caracteres de la primera

fila, en forma de lista. (Donde una lista es un `vector` o `array` en C++, una `List` en Java, etcétera.)

Otra un poco más sencilla es una lista de strings, donde el primer string es la primera fila, el segundo la segunda, y el último string contiene la última fila. Algo aún más sencillo es simplemente guardar 3 strings, siempre recordando cuál string es cada fila para no perdernos y modificar el tablero.

Vamos a trabajar con 3 strings, que llaramos $F1$, $F2$, $F3$ simplemente para hacer sencilla la notación. Si guardaran una lista de strings, $F1$ sería el primer elemento de esa lista.

Notemos con $F1_1$ a la primera celda de la primera fila (y respectivamente cambiando los números para las otras celdas de otras filas). Entonces para **chequear si hay tatetí**, simplemente tenemos que chequear si $F1_1, F1_2, F1_3$ son iguales, lo mismo para la columna $F1_1, F2_1, F3_1$, y las diagonales, como por ejemplo $F1_1, F2_2, F3_3$. Esto en todas filas/columnas/diagonales. Entonces, si hiciéramos una función llamada por ejemplo `hayTateti` que recibe los tres strings, ya sabemos cómo responder `True` o `False`. Si en alguna dirección de las mencionadas hay, devolvemos `True`, y si no `False`.

Es muy importante tomarse el debido tiempo para pensar esta función, no equivocarse en números de índices para las casillas, si es necesario poner un comentario para cada dirección (por ejemplo en el ´if´ de la primera fila, escribir como comentario ´chequeo tatetí fila 1´ o algo así).

Si esta función está mal, va a fallar todo y va a ser difícil descubrir por qué.

También es importante *poner el código en una función*, ya que, para el problema completo, lo vamos a usar muchas veces, y además aunque sea sólo para la primera parte, queda mucho más prolijo porque son un montón de "ifs".

Entonces la primera parte queda completa: Recibimos las tres filas, buscamos (con un for o simplemente viendo cada una de las 9 casillas) el punto (`'.'`), que sabemos que va a haber uno solo, reemplazamos ese caracter por una `'O'`, y llamamos a la función `hayTateti`. Si hay, le ponemos una `'G'` a esa celda. Si no, una `'E'` ya que sabemos que como no había tatetí al principio, no podrá haber tatetí de las `'X'` si sólo agregamos una `'O'`. Imprimimos lo que haga falta y listo.

Algo importante a tener en cuenta, es que si el tablero fuera por ejemplo de 10×10, entonces buscar una casilla con un puntito se hace prácticamente imposible si queremos mirarlas de a una. Para eso vamos a necesitar un **for**. Y si guardamos las filas en 10 variables de tipo `string` distintas, vamos a necesitar un **for** por cada

fila y quedaría todo súper engorroso. Por eso es que convendría muchísimo guardar una lista de strings: simplemente hacemos un **for** para pasar por cada elemento de la lista (es decir, cada fila), y adentro de ese otro **for** para ver cada celda, y con un **if** vemos si el valor es un punto o no.

Ahora que vimos la subtarea, y que sabemos cómo chequear si en un tablero hay tatetí, veamos cómo resolver el problema en su totalidad.

Lo que queremos hacer es simular un juego **óptimo**, es decir, donde cada jugador hace lo mejor para ganar, asumiendo que la otra persona hará lo mismo. Entonces, lo que podríamos programar es una función que nos diga la jugada óptima dado el tablero actual, y a quién le toca. ¿Cómo podemos pensar a la función?

Supongamos que le toca a A, y queremos responder "si A juega en la celda c, ¿cómo sería el resultado cuando le toque a mi oponente?". Para responder a esa pregunta, podemos pensar en modificar la celda c como si A jugara ahí, y luego pensar "qué es lo mejor que le puede pasar a B con este nuevo tablero". Y para responder a esta pregunta, podemos **llamar a esta misma función, pero ahora pensando que es el turno de** B. Si esta función nos dice "con este tablero, B ganaría", entonces A seguro que no va a querer jugar ahí. Pero si la función nos dice "con este tablero, B perdería", entonces A sí va a querer jugar ahí y ganar (ya que la función nos dice lo mejor que le puede pasar a B, en este caso no podría evitar perder).

Esta es una función **recursiva**, que se llama a sí misma. ¿Cómo funciona esta función? Lo que hará será recorrer todas las celdas vacías, ubicar la letra que corresponda en esta celda, y llamar a la función con el nuevo tablero, indicando que le toca al otro jugador. Si alguna celda nos da que nuestro contrincante perdería, entonces jugando ahí ganamos. Ahora si la función nos da que la otra ganaría, entonces ahí perdemos. Y si nos da que hay empate, entonces jugando ahí hay empate.

Se puede leer de recursión acá: http://wiki.oia.unsam.edu.ar/algoritmos-oia/recursion , y un poco más de utilidad para este problema acá: http://wiki.oia.unsam.edu.ar/algoritmos-oia/backtracking

Ahora, como toda función recursiva, necesita una **situación en la que se deje de llamar a sí misma**, ya que de no ser así se quedaría colgada haciendo infinitas llamadas a sí misma. ¿En qué situación podemos dejar de llamarla? Pensemos un segundo.

¡Cuando el tablero está lleno! Si se llama a la función con el tablero lleno, como no hay casilla vacía para llenar, ya sabremos seguro el resultado de la partida. Pero

hay una situación más donde sabemos el resultado antes de llenar el tablero ...

¡Cuando el partido terminó antes, porque hay tatetí!. Y acá es cuando vamos a usar la función `hayTateti` que vimos antes. Si a la función le pasamos un tablero indicando que le toca a X, y hay tatetí de O, entonces la función devolverá ´pierde´. Y si no hay tateti, buscará las casillas vacías y hará de vuelta la recursión. Si el tablero está lleno, y no hay tateti, la respuesta será ´empata´.

Entonces veamos un pseudocódigo de esta función, que quizás ilustre un poco mejor la situación. En el pseudocódigo, `tablero` será una lista de 3 strings, y `turno` será una letra, `X` u `O` según a quién le toque.

Algorithm 17 Mejor resultado dado un tateti y a quién le toca

```
1: function mejorResultado(tablero : ARREGLO DE PALABRAS, turno : LETRA ) (PALABRA)
2:     if ultimoDigitoVisto ≤ digitoActual then
3:         return "pierde"
4:     puedoGanar ← False
5:     puedoEmpatar ← False
6:     hayAlgunaVacia ← False
7:     oponente ← 'X'
8:     if turno == 'X' then
9:         oponente ← 'O'
10:    for i = 0, 1, 2 do
11:        for j = 0, 1, 2 do
12:            if tablero[i][j] == '.' then
13:                hayAlgunaVacia ← True
14:                tablero[i][j] ← turno
15:                resultado ← mejorResultado(tablero, oponente)
16:                if resultado == "pierde" then ▷ Jugando en (i,j) el rival pierde
17:                    puedoGanar ← True
18:                else if resultado == "empata" then
19:                    puedoEmpatar ← True
20:                tablero[i][j] ← '.'     ▷ Como modificamos el tablero, para seguir probando otras casillas, volvemos a su estado inicial
21:    if hayAlgunaVacia == False then
22:        return "empata"
23:    if puedoGanar == True then
24:        return "gana"
25:    if puedoEmpatar == True then
26:        return "empata"
27:    return "pierde"
```

Entonces, ahora que tenemos programada esa función, una vez que recibimos `tablero`, lo que tendríamos que hacer es:

1. Ver a quién le toca. Para esto, con un par de **for** contamos cuántas `'X'` y `'O'` hay. Si hay igual cantidad le toca a `'O'` y si hay una `'O'` más, le toca a `'X'`.

2. Construir un tablero aparte, donde iremos poniendo `'G'`, `'E'` o `'P'` en las vacías, pero *no modificar el tablero real del juego*.

3. Recorrer todas las celdas. Si encontramos un punto (`'.'`), lo reemplazamos por la letra que corresponda según lo que hicimos en el ítem 1, y averiguamos el resultado llamando a la misma función, pero con el turno de la otra persona. Si esa función nos devuelve `"pierde"`, entonces sabemos que jugando acá vamos a ganar. Si nos devuelve `"empata"`, sabemos que jugando acá empatamos, y si devuelve `"gana"`, sabemos que jugando acá perdemos.

4. Mientras recorremos las celdas y vamos averiguando si corresponde poner una `'G'`, `'E'` o `'P'`, colocamos estas letras en nuestra copia del tablero.

5. Si en algún lado pusimos una `'G'`, la situación es ganadora. Si en algún lado pusimos una `'E'`, la situación es empatadora. Y si fueron todas `'P'`, es perdedora. Para saber esto podemos poner un contador que cuenta la cantidad de `'G'` y `'E'` que ponemos, o simplemente una variable booleana que nos diga si colocamos una `'G'` o no (y lo mismo para `'E'`).

6. Imprimimos la situación seguida de nuestro tablero paralelo con las letras correspondientes.

Capítulo 4

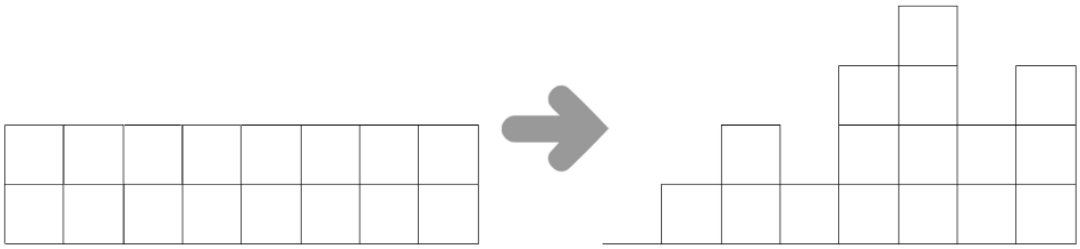
Certamen Nacional

4.1. Nivel 1

4.1.1. Problema 1: Ordenando la habitación [zapatos]

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

El enunciado nos dice que originalmente había P pilas de cajas de zapatos, y que todas las pilas tenían la misma cantidad de cajas. La idea clave es que al mover cajas de zapatos de una pila a la otra, **la cantidad total de cajas no varía** (la caja que sale de una pila, se agrega en otra).

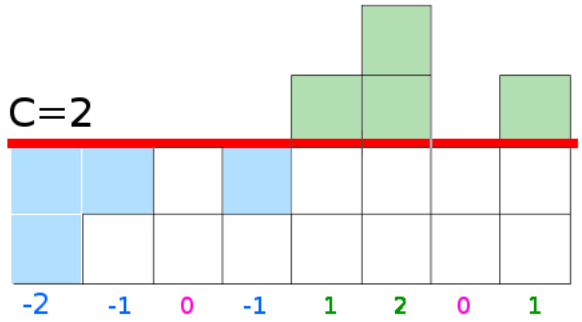


Como originalmente todas las pilas tenían la misma cantidad de cajas de zapatos, llamando C a esa cantidad, sabemos que en total habrá $C \cdot P$ cajas de zapatos (C cajas en cada una de las P pilas). Por otro lado, esta cantidad total la podemos obtener sumando la cantidad que hay actualmente en cada pila. Si llamamos $SUMA$ a la cantidad total de cajas de zapatos que vienen en la entrada, tenemos la siguiente relación:

$$C \cdot P = SUMA \Rightarrow C = \frac{SUMA}{P}$$

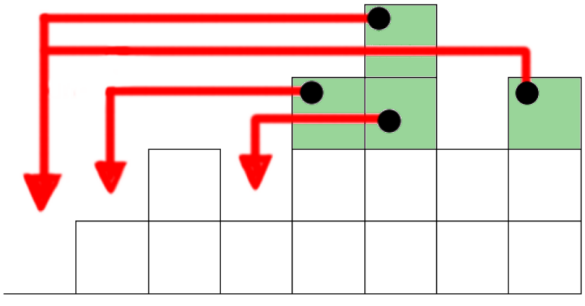
Donde tanto $SUMA$ como P son conocidos o podemos calcularlos a partir de los datos que vienen en el enunciado.

Finalmente, para cada pila podemos calcular cuántas cajas de zapatos le faltan o le sobran respecto de la cantidad que tenían originalmente (que es C). Llamando $cajas_i$ a la cantidad de cajas que tiene actualmente la pila i, podemos concluir que $cajas_i - C$ será un número positivo si le sobran cajas a la i-ésima pila, y será un número negativo si le faltan cajas a dicha pila. Llamemos S a la *cantidad total de cajas de zapatos que sobran.*



Para volver a cada pila a la cantidad original de cajas de zapatos, al menos debemos mover todas las que sobran, es decir S. Si cada una de estas cajas la ponemos en algún lugar donde falten, habremos hecho exactamente esa mínima cantidad de movimientos necesaria (y por lo tanto lo habremos realizado de manera óptima).

Para calcular S se puede sumar $cajas_i - C$ cuando este número sea positivo. En el ejemplo la respuesta es 4 y corresponde a sumar las cajas de zapatos que sobran en cada pila donde sobran cajas (la cantidad de cajas verdes en la imagen). Notar que la cantidad de cajas que sobran es igual a la cantidad de cajas que faltan en todo momento (si habría más de una que de la otra, la cantidad total de cajas habría variado).



Algorithm 18 Solución al Problema 1 de Nivel 1 Nacional 2018 - zapatos

```
1: function ZAPATOS(P : ENTERO, CAJAS : ARREGLO DE ENTEROS)(ENTERO)
2:     SUMA ← 0
3:     for i = 0 ... P-1 do
4:         SUMA ← SUMA + cajas[i]
5:     C ← SUMA/P
6:     cajas_a_mover ← 0
7:     for i = 0 ... P-1 do
8:         if cajas[i]-C > 0 then
9:             cajas_a_mover ← cajas_a_mover + (cajas[i]-C)
10:    return cajas_a_mover
```

4.1.2. Problema 2: Procesador de textos [pluralizador]

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

En este problema se nos da un conjunto de sustantivos en singular y nuestra tarea es pasarlos al plural *siguiendo las reglas del enunciado para calcular el plural de una palabra*. Estas reglas dicen:

1. Si el sustantivo **termina en vocal** se agrega **"s"**.
2. Si el sustantivo **termina en "s" o "x"** entonces **queda igual**.
3. Si el sustantivo **termina en z** entonces **se cambia la "z" por "ces"**.
4. Si el sustantivo **termina en otra consonante** entonces **se agrega "es"**

Además, nos piden que almacenemos en un arreglo cuántas veces fue utilizada cada regla. Por lo tanto, el problema en cuestión pide solamente implementar correctamente lo que está descrito en el enunciado. Una forma posible de hacerlo se muestra en el siguiente pseudocódigo.

Se asume que el arreglo `cantidadesPorRegla` comienza inicializado con 0, y que podemos preguntar directamente si un cierto elemento pertenece a un contenedor (si tuviéramos que hacerlo nosotros tendríamos que iterar el contenedor, no debería ser una dificultad mayor). Además utilizaremos el símbolo `+`, para *concatenar* dos palabras.

Una observación del enunciado es que en este problema no hace falta devolver nada, sino que *al terminar de ejectuarse, las variables* (en este caso arreglos) *que vienen en la entrada deben modificarse para almacenar la respuesta*.

Algorithm 19 Solución al Problema 2 de Nivel 1 Nacional 2018 - pluralizador

```
1: function PLURALIZADOR(N : ENTERO, PALABRAS : ARREGLO DE PALABRAS,
   CANTIDADESPORREGLA : ARREGLO DE ENTEROS)
2:     vocales ← ['a','e','i','o','u'] ▷ Solo vienen palabras en minúscula
3:     for i = 0 ... N-1 do                                  ▷ Para cada palabra:
4:         largo ← palabras[i].largo()
5:         if palabra[i][largo-1] ∈ vocales then                    ▷ Regla 1
6:             palabra[i] ← palabra[i] + 's'
7:             cantidadPorRegla[0] ← cantidadPorRegla[0]+1
8:         else if palabra[i][largo-1] ∈ ['s','x'] then             ▷ Regla 2
9:             cantidadPorRegla[1] ← cantidadPorRegla[1]+1
10:        else if palabra[i][largo-1] == 'z' then                  ▷ Regla 3
11:            palabra[i][largo-1] ← 'c'
12:            palabra[i] ← palabra[i] + 'es'
13:            cantidadPorRegla[2] ← cantidadPorRegla[2]+1
14:        else                                                     ▷ Regla 4
15:            palabra[i] ← palabra[i] + 'es'
16:            cantidadPorRegla[3] ← cantidadPorRegla[3]+1
```

4.1.3. Problema 3: Armando cartas numerológicas [numerologo]

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

El enunciado describe un proceso que se le debe realizar a un número para obtener el siguiente en una *secuencia de buena suerte*. Concretamente, para generar el siguiente número se realiza lo siguiente:

1. Se factoriza el número en factores primos.
2. Se ordenan los factores (con repeticiones) de menor a mayor.
3. Se concatenan todos los números para formar uno más grande.

Si sabemos realizar correctamente cada paso, entonces el problema resulta relativamente sencillo, dado un número en la entrada debemos generar el siguiente hasta alcanzar un número primo u obtener un número mayor que 10,000.

¿Cómo sabemos que el proceso termina?¿Por qué no puede ciclar entre dos (o más números) que no son primos? Una forma sencilla es implementar el proceso y probar todos los casos posibles en la computadora. Otra forma es ver que si el número no es primo, entonces el siguiente número obtenido por el proceso es estrictamente mayor que el original.

La idea detrás de que la secuencia es **estrictamente creciente** cuando el número N dado no es primo, podemos escribirlo como $N = A \cdot B$, ambos A y B mayores o iguales que 2.

Llamemos c a la cantidad de cifras de B, entonces la concatenación de A y B está dada por el número $A \cdot 10^c + B$. Notemos que $10^c > B$ (pues c es la cantidad de cifras). Finalmente queremos probar que $A \cdot 10^c + B > A \cdot B \iff 10^c \cdot A > A(B-1) \overset{\text{porque } A>0}{\iff} 10^c > B - 1$, lo cual ya probamos que ocurre.

Finalmente, si tenemos la factorización de un número $N = p_1 \cdot p_2 \cdot \cdots \cdot p_k$, con $p_i \leq p_{i+1}$ y k la cantidad de factores primos que tiene el número (contando repetido). Aplicando el argumento secuencialmente con los primeros dos números, conluimos que $N = p_1 \cdot p_2 \ldots p_k < (p_1 p_2) \cdot p_3 \ldots p_k < (p_1 p_2 p_3) \ldots p_k < \ldots < \underbrace{(p_1 p_2 \ldots p_k)}_{\text{siguiente en la secuencia}}$, que es lo que queríamos probar.

Veamos entonces cómo implementar cada una de las operaciones. Para **factorizar** un número, podemos simplemente iterar por todos los números menores y chequear si lo dividen o no. En caso de que lo divida, debemos *agregar todas las apariciones de ese divisor primo*. Otra opción es usar la *criba de eratóstenes* pero dadas las cotas del enunciado realmente no hace falta. En caso de estar interesados en esto último, recomendamos leer el post en la wiki http://wiki.oia.unsam.edu.ar/algoritmos-oia/enteros/criba-de-eratostenes.

Para **ordenar** podemos utilizar cualquier algoritmo de ordenamiento que conozcamos (la cantidad de números a ordenar es la cantidad de factores primos de un número, que es $\mathcal{O}(\lg N)$). En muchos lenguajes de programación ya hay implementado un algoritmo de ordenamiento eficiente (`sort` en C++ por ejemplo). Veamos cómo utilizar todo esto para calcular la lista ordenada de factores de un número como así también la cantidad de factores primos distintos que tiene.

Algorithm 20 Factores ordenados de N y su cantidad de factores primos distintos

```
1:  function FACTORIZAR(N: ENTERO, LISTAFACTORES: ARR. DE ENTEROS)(ENTERO)
2:      factores_distintos ← 0
3:      divisor_primo ← 2
4:      while divisor_primo ≤ N do  ▷ Iterando en orden nos ahorramos ordenar
5:          if (N% divisor_primo) == 0 then
6:              factores_distintos ← factores_distintos + 1
7:              while (N% divisor_primo) == 0 do ▷ Contar todas las apariciones
8:                  lista_factores.agregar_al_final(divisor_primo)
9:                  N ← N/divisor_primo
10:         divisor_primo ← divisor_primo + 1
11:     return factores_distintos
```

Restaría ver cómo **concatenar** dos números, pero en la demostración de que la secuencia es creciente vimos una opción para hacer eso. Si tenemos dos números A y B, si calculamos c la cantidad de cifras de B, entonces la concatenación $(AB) = 10^c \cdot A + B$. Auxiliarmente, veamos cómo calcular 10^c para un número B cualquiera.

Algorithm 21 Dado un número B calcula 10^c, con c la cantidad de cifras de B

```
1: function POT10CIFRAS(B : ENTERO)(ENTERO)
2:     respuesta ← 1
3:     while B > 0 do
4:         respuesta ← respuesta * 10
5:         B ← ⌊B/10⌋
6:     return respuesta
```

Luego basta con repetir la operación a todos los números de la lista. En el siguiente algoritmo `lista_factores` es un arreglo de largo `K`.

Algorithm 22 Concatenar los números de una lista

```
1: function CONCATENAR(LISTAFACTORES: ARR. DE ENTEROS)(ENTERO)
2:     respuesta ← 0
3:     for i = 0 ... K-1 do
4:         respuesta ← respuesta * pot_10_cifras(lista_factores[i])
5:         respuesta ← respuesta + lista_factores[i]
6:     return respuesta
```

Juntando todo esto, un pseudocódigo del algoritmo que resuelve el problema se encuentra a continuación. En el mismo asumimos que `secuencia` es un contenedor que comienza vacío.

Algorithm 23 Solución al Problema 3 de Nivel 1 Nacional 2018 - numerologo

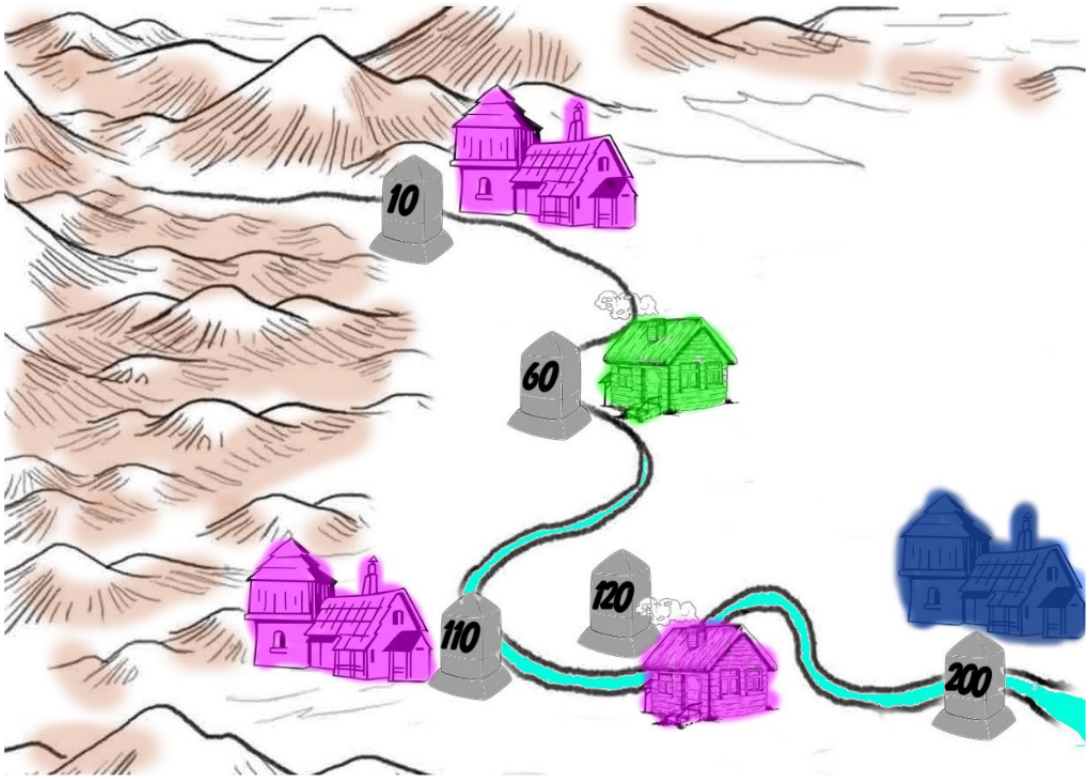
```
1: function NUMEROLOGO(N: ENTERO, SECUENCIA: ARR. DE
   ENTEROS)(ENTERO)
2:     lista_factores ← []
3:     factores_distintos ← factorizar(N,lista_factores)
4:     actual ← N
5:     while actual ≤ 10000 do
6:         secuencia.agregar_al_final(actual)
7:         actual ← concatenar(lista_factores)
8:         lista_factores ← []
9:         auxiliar ← factorizar(actual,lista_factores)
10:    return factores_distintos
```

4.2. Nivel 2

4.2.1. Problema 1: Dividiendo pueblos [pueblitos]

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

De todas las asignaciones válidas de los pueblos a los hijos, se busca una que **maximice la mínima distancia entre dos pueblos vecinos asignados a hijos distintos**. Notemos que la longitud de estos intervalos corresponde a la diferencia entre la ubicación de dos pueblos consecutivos. La idea clave para resolver este problema radica en que si se tiene una configuración donde todos los pueblos de un mismo hijo no son consecutivos, entonces **se puede obtener otra configuración donde todos los pueblos de todos los hijos sean consecutivos sin que la respuesta empeore**. Quizá la imagen del enunciado no nos ayuda a ver esto, pero puede verse que existe una configuración que no empeora la situación intercambiando la asignación de los primeros dos pueblos.



Veamos cómo probar esto. Si tenemos un hijo del rey, que llamaremos A, con pueblos no consecutivos, entonces hay dos rangos *no vacíos* $\mathcal{A}_1 = [l_1, r_1]$, y $\mathcal{A}_2 = [l_2, r_2]$ donde todos los pueblos en ambos rangos corresponden al hijo A. Sin pérdida de generalidad podemos asumir que $r_1 < l_2$ (de no ser así intercambiamos

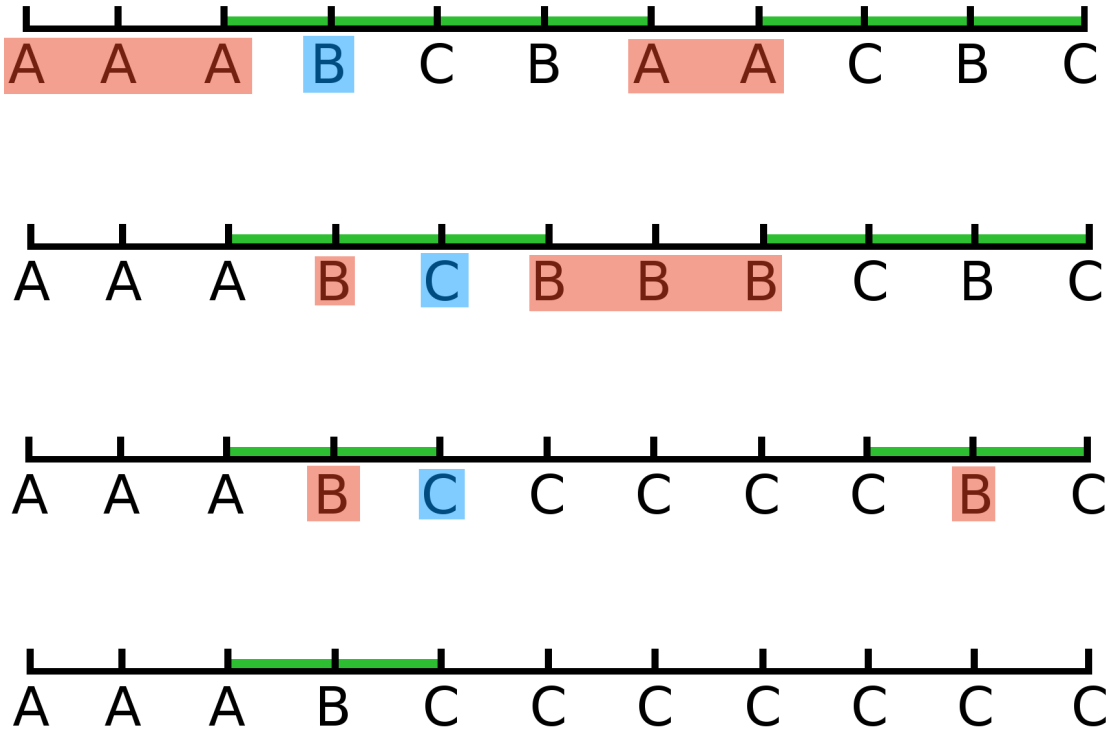
los roles de $\mathcal{A}_1$ con $\mathcal{A}_2$) y que los rangos son *maximales* (fueron tomados lo más grande posibles, no se pueden extender ni a izquierda ni a derecha con pueblos del hijo A).

Además de estos dos rangos, necesariamente debe haber otro hijo del rey, que llamaremos B, entre r_1 y l_2, pues de no ser así $\mathcal{A}_1$ y $\mathcal{A}_2$ estarían pegados y los hipotéticos pueblos de A serían consecutivos. Consideremos entonces la configuración que toma todos los pueblos igual que antes, salvo a los que están comprendidos en $\mathcal{A}_2$ y ahora se los asigna al hijo B. Veamos que la situación es factible y que no empeoró.

Para ver que es factible solo hay que ver dos cosas. Primeramente que *cada pueblo está siendo entregado a un hijo*, lo cual ocurre trivialmente (todos los pueblos tienen la misma asignación que antes salvo los de $\mathcal{A}_2$ a los cuales les asignamos otro hijo, de todas formas, todo pueblo tiene a un hijo asignado). Y por último tenemos que ver que *cada hijo recibe al menos un pueblo*. Esto ocurre gracias a que la configuración anterior era válida, como al único hijo al que le sacamos peublos es a A, el único que podría no cumplir esto es A. Sin embargo, todavía tiene al menos todos los pueblos de $\mathcal{A}_1$, por lo que la asignación sigue siendo válida.

Nos resta ver que la respuesta no empeoró. Solo nos interesan las vecindades entre pueblos asignados a hijos distintos. Observemos que los únicos pueblos que cambiaron son los de $\mathcal{A}_2$. Con esto en mente, notemos que entre los pueblos consecutivos interiores a $\mathcal{A}_2$ la situación no cambió, pues siguen siendo pueblos asignados a un mismo hijo (solo que ahora es B en lugar de A). Por lo tanto, las únicas vecindades que cambiaron ocurren inmediatamente a la izquierda de l_2 e inmediatamente a la derecha de r_2. Como tomamos $\mathcal{A}_2$ de forma maximal, sabíamos que estos pueblos inmediatamente aledaños son distintos de A, por lo tanto antes estaban siendo considerados en la respuesta, y por ende no hemos empeorado la situación (en todo caso, si algún vecino aledaño era B podríamos haber mejorado la respuesta).

Veamos esto con un esquema. Marcamos con verde los intervalos que son considerados en la respuesta (formalmente, la respuesta será la máxima longitud de estos intervalos), notemos que los intervalos verdes que quedan al final son siempre un subconjunto de los originales por lo que explicamos. Los pueblos marcados en naranja representan a $\mathcal{A}_1$ y $\mathcal{A}_2$ en cada paso, y el pueblo marcado en azul es uno cualquiera intermedio que es el que le será asignado a $\mathcal{A}_2$ en el paso siguiente.



Algo importante que hay que notar es que **en ningún momento hace falta implementar en la computadora** todo esto. Todo lo que hicimos fue un razonamiento de "lápiz y papel" que nos ayudará en la resolución del problema. Más aún en una instancia de competencia, con suficiente confianza en la intuición propia (lo cual es un arma de doble filo) este razonamiento podría haber sido utilizado sin una demostración formal.

Utilizando que ahora podemos asumir que los pueblos asignados a un mismo hijo son consecutivos es fácil ver que *la cantidad de intervalos distintos hijos en los extremos serán exactamente* $K - 1$ (pues esa es la cantidad de veces que pasamos de un hijo a otro, y todos deben ser utilizados). Nuestro objetivo es *maximizar la mínima distancia de uno de estos intervalos* que tienen distintos hijos en sus extremos, ¿cómo deberíamos tomarlos entonces? ¡Lo más grandes posibles!

Por lo tanto, la solución al problema consiste en tomar todas las diferencias entre pueblos consecutivos, y encontrar una asignación factible que tenga distintos hijos solamente en los $K - 1$ intervalos más grandes. Más aún, la máxima mínima distancia buscada corresponde a la $(K - 1)$-ésima distancia entre consecutivos más grande. Veamos un pseudocódigo de cómo calcular este valor y generar una asignación factible. Se asume que `ubicacion` es un arreglo de tamaño `N` que viene ordenado (como indica el enunciado). La respuesta estará en el arreglo `asignacion` que tiene tamaño N.

Algorithm 24 Solución al Problema 1 de Nivel 2 Nacional 2018 - pueblitos

```
1: function PUEBLITOS(K : ENTERO, UBICACION : ARREGLO DE ENTEROS)(ENTERO)
2:     diferencias ← []                          ▷ Arreglo de tamaño N-1
3:     for i = 0 ... N-2 do
4:         diferencias[i] ← ubicacion[i+1] - ubicacion[i]
5:     diferencias.ordenar()      ▷ En orden decreciente (mayor a menor)
6:     hijo ← 1
7:     dist_respuesta ← diferencias[K-2]     ▷ Notar que indexamos desde 0
8:     for i = 0 ... N-1 do
9:         asignacion[i] ← 1          ▷ Asignamos todo al hijo 1 para empezar
10:    for i = 1 ... N-1 do
11:        if hijo < K and ubicacion[i]-ubicacion[i-1] ≥ dist_respuesta then
12:            hijo ← hijo + 1
13:        asignacion[i] ← hijo
14:    return dist_respuesta
```

La complejidad temporal de este algoritmo es $\mathcal{O}(N \lg N)$, pues debemos ordenar un arreglo de tamaño N. El resto del algoritmo es $\mathcal{O}(N)$, ya que solo hacemos iteraciones lineales sobre arreglos de tamaño $\mathcal{O}(N)$.

4.2.2. Problema 2: Vigilando la ciudad [vigilantes]

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

4.2.2.1. Subtarea $x_i \in \{1, 2\}$

En este caso muy especial, hay solamente dos calles norte-sur que contienen a todos los vigilantes. Podemos observar que en tal situación, como máximo son necesarios 3 envíos de señales, así que podemos analizar la situación de cada vigilante directamente:

- Aquellos que están en la misma x o la misma y que el jefe, los marcamos a distancia 1.
- Si en la misma y que el jefe existió otro vigilante (que por el ítem anterior, estaba a distancia 1), entonces podemos cubrir ambas calles sin problema, y ya podemos marcar todos los vigilantes que quedan con distancia 2.
- Si no se da la situación del paso anterior, pero hay algún vigilante de los que están en la misma x que el jefe, que comparte la y con alguno de los que falta, podemos marcar con $d = 2$ a todos aquellos vigilantes que están en la otra x

pero tienen un y compartido con otro vigilante, y con $d = 3$ a todos los demás restantes.

- Si tampoco se da la situación anterior, entonces todos los vigilantes que quedaron sin marcar quedan con el valor -1, pues no se puede llegar a ellos.

Lo anterior puede implementarse eficientemente con algunos fors + ifs, y chequeos de pertenencia a conjuntos (para verificar si hay alguien con la misma y).

4.2.2.2. Subtareas de tamaño pequeño

Este es un problema clásico de camino mínimo, y puede modelarse naturalmente con grafos.

El modelo más natural es poner un nodo por cada vigilante, y una arista entre dos de ellos exactamente cuando los correspondientes vigilantes están en la misma fila o en la misma columna. Luego, el dato que pide calcular el enunciado no es más que la distancia desde un nodo origen (el que corresponde al jefe de todos los vigilantes) y todos los demás nodos.

Esto puede calcularse utilizando el algoritmo de BFS. Se puede leer sobre este algoritmo en la wiki: http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/bfs.

Por cada nodo, para poder ejecutar el algoritmo de BFS, lo importante es poder iterar sus vecinos. Es decir, por cada vigilante debemos poder computar aquellos que ve. Hay dos mecanismos básicos, y cuál conviene dependerá de los tamaños de la subtarea:

- Iterar cada uno de los N vigilantes, y verificar si tiene la misma coordenada x o la misma coordenada y que el vigilante actual. Tiempo $O(N)$

- Iterar todas las ubicaciones del mapa en la misma x que el vigilante actual, para ver en cada una si hay vigilante o no. Luego lo mismo con las y. Esto toma tiempo $O(x_{max} + y_{max})$, y además es necesario llenar inicialmente una matriz de tamaño $x_{max} \times y_{max}$ que indica el número de vigilante que hay en cada esquina, si lo hay.

Utilizando estas ideas se consiguen 71 puntos en este problema.

4.2.2.3. Solución completa

Para obtener una solución completa de 100 puntos, es necesario realizar el BFS más eficientemente, sin que sea necesario volver a explorar todos los vecinos de cada nodo. Esto es así porque, por ejemplo, si todos los vigilantes están en una misma calle, se ven todos entre sí, y por lo tanto es inevitable un tiempo de orden N^2 si se iteran todas las vecindades explícitamente.

Hay varias maneras de implementar un BFS más eficiente para este caso especial. La que proponemos consiste en dar vuelta los roles: en el grafo anterior, los vigilantes son nodos, y las calles indican las aristas. Lo que haremos ahora será que **las calles sean los nodos**, y que **los vigilantes sean las aristas**. La arista de un vigilante une siempre exactamente dos calles: la vertical y la horizontal que corresponden a la esquina del vigilante.

La pregunta a responder ahora es la distancia a la cual **cada arista** del grafo está de una cierta arista inicial. Como los caminos de nodos y de aristas son básicamente equivalentes, lo que podemos hacer es lanzar un BFS sobre este grafo, con dos nodos iniciales a distancia 0: Los extremos de la arista que corresponde al jefe de vigilantes.

Al terminar, la distancia a cada arista (u, v) sera $1 + min(d_u, d_v)$, excepto la arista del jefe de vigilantes que tendrá distancia 0 por ser la inicial (y no 1 como indicaría la cuenta).

Esta solución da un algoritmo de tiempo lineal, pues el tamaño del grafo (es decir, la cantidad de nodos y aristas) es proporcional a la cantidad de vigilantes.

4.2.3. Problema 3: Viaje de egresados [egresados]

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

Este era indudablemente el problema más difícil del certamen nivel 2. No se esperaba que ningún participante apuntara a 100 puntos en este problema durante la prueba, sino solamente a puntajes parciales.

La clave para resolver este problema es utilizar búsqueda binaria en la respuesta: Es decir, haremos búsqueda binaria en el tiempo T permitido entre una habitación de estudiante y su coordinador más cercano. Para cada valor de T, nuestro algoritmo deberá poder indicar si es posible ubicar los C coordinadores de forma tal que todos los estudiantes puedan ser alcanzados por alguno de los coordinadores en un tiempo máximo T.

Se pueden leer ejemplos y explicaciones sobre esta técnica de búsqueda binaria

sobre la respuesta en la web de OIA: http://www.oia.unsam.edu.ar/charlas/

4.2.3.1. Subtarea $P = 1$

El caso particular en el que tenemos un único piso daba 16 puntos y es muchísimo más simple. En estos casos el descanso puede reemplazarse por una habitación no reservada, pues lo único que lo diferencia de un espacio vacío normal es la capacidad de cambiar de piso allí, por lo cual si no hay otros pisos podemos pensar que tenemos nada más letras N y S.

Para este caso podemos dar un **algoritmo goloso** que es completamente válido: Si consideramos la primera habitación reservada (es decir la primera S), deberá ser cubierta por alguno de los coordinadores que utilicemos, es decir, o bien debe tener un coordinador en esa misma habitación, o bien el coordinador más cercano debe estar a una distancia máxima T. Pero como todas las demás habitaciones están más a la derecha, lo que conviene es colocar un coordinador en la habitación más a la derecha que todavía pueda cubrir esta habitación, ya que cualquier otra elección cubre un subconjunto de las habitaciones cubiertas por esa opción de más a la derecha posible.

El algoritmo goloso consiste en repetir este paso hasta cubrir todas las habitaciones: En cada paso, se toma la primera habitación sin cubrir (la primera S de la cadena que aún no fue cubierta), y de todas las habitaciones (incluyendo esa misma), se elige la de más a la derecha entre todas las que cubren esa habitación (que podría ser esa misma habitación, ya sea porque es la única que queda por cubrir, o porque todas las demás que quedan están demasiado lejos, a más de T pasos).

Como este algoritmo goloso cubrirá el piso utilizando la mínima cantidad posibles de coordinadores, basta verificar si la cantidad usada fue a lo sumo C: si se requirieron más de C coordinadores, T no era un valor posible así que la respuesta tendrá que ser un T mayor, mientras que si utilizaron C o menos coordinadores, es posible obtener una separación T y la respuesta final será T o menos. Continuando con la búsqueda binaria que utiliza el algoritmo goloso en cada paso, obtenemos la respuesta.

4.2.3.2. Subtarea $P = 2$

Para esta subtarea, tenemos la complicación de que al haber dos pisos, es posible que un coordinador de un piso cubra habitaciones que están en el otro.

Podemos primero calcular el T óptimo cuando asumimos que esto no ocurre: Es decir, haciendo búsqueda binaria en T como antes, podemos computar para cada

piso en forma independiente la mínima cantidad de coordinadores que necesita, digamos que son C_1 y C_2: y entonces, si resulta $C_1 + C_2 \leq C$, se puede con ese T, y sino, no se puede. Este método nos dará el T óptimo en el caso de que los coordinadores no crucen de piso.

Falta considerar el caso en que los coordinadores cruzan de piso. Una observación clave para simplificar el estudio es que no tiene sentido que un coordinador pase del piso 1 al 2, y que también un coordinador pase del 2 al 1. Si así fuera, podemos observar que para alguno de los dos pisos, una cierta habitación está siendo cubierta por un coordinador del otro piso, cuando también podría ser cubierta en igual o menor tiempo por un coordinador del mismo piso. De manera similar, podemos asumir que existe una única habitación de la cual sale un coordinador que puede cruzar al otro piso: ya que si hubiera más de una, solamente la más cercana al descanso es relevante, pues la otra tarda más en llegar al otro piso.

Teniendo esto en cuenta, como hay un máximo de 2000 habitaciones, podemos realizar fuerza bruta probando las 2000 posibilidades de habitación que contiene al coordinador que puede ir al otro piso. Lo que hacemos entonces es probar para cada habitación, cuál es la mejor solución posible que usa esa habitación para poner allí un coordinador, y tal que ese coordinador es el único que vamos a considerar que puede cruzar al otro piso. Por las observaciones anteriores, considerando todas estas posibilidades y tomando la mejor, obtenemos la respuesta óptima.

Para esto marcamos esa habitación y todas las que alcanza como ya cubiertas, **incluso aquellas que alcanza en el otro piso**. Luego, en cada piso, utilizamos exactamente el mismo goloso del caso $P = 1$, con la salvedad de que como vimos, ya iniciamos con un coordinador ubicado y ciertas habitaciones cubiertas, pero el método sigue siendo el mismo: Tomamos siempre la primera habitación (la más a la izquierda) del piso que aún no fue cubierta, y la cubrimos poniendo el coordinador lo más a la derecha posible.

Notar que en cuanto a código fuente, la implementación de esta idea es muy poco código adicional además del que ya se tenía del caso $P = 1$, pues basta con un for que itere las 2000 habitaciones, y el código para marcar y procesar las que esta habitación especial preelegida ya cubre.

4.2.3.3. Solución completa

Utilizando el algoritmo explicado anteriormente, se obtienen 40 puntos. Se explica a continuación un algoritmo de programación dinámica que puede resolver eficientemente todos los casos. Como siempre, utilizaremos búsqueda binaria en la respuesta, de modo que el problema que nos enfocamos en resolver es calcular

la mínima cantidad de coordinadores tal que todos quedan vigilados a distancia máxima T.

La idea principal es utilizar en cada piso el método goloso, pero ahora solamente desde las puntas hacia el descanso. Es decir, como antes, tomamos el primer S no cubierto, y lo cubrimos con el coordinador más a la derecha posible, y repetimos, pero solo mientras que esta habitación S **no sea alcanzable desde el descanso**[1]. En cuanto esta habitación se encuentra a menos de T del descanso, dejamos pendiente su cubrimiento con el goloso. Lo mismo realizamos tomando el último S no cubierto, y lo cubrimos con el coordinador más a la izquierda posible, siempre y cuando la habitación se encuentre a T o más unidades del descanso. Es decir, realizamos el goloso desde las dos puntas, aproximándose al descanso.

Este mismo proceso podemos completarlo en cada piso nuevo que procesemos. De esta forma, en cada piso, las habitaciones que posiblemente quedaron sin cubrir están a una distancia máxima $T - 1$ del descanso. Esto significa que, si para cubrir una de estas habitaciones se utiliza un coordinador de ese mismo lado del descanso en ese mismo piso, basta con un coordinador para cubrir todas las que haya, y además ese coordinador conviene ubicarlo en la habitación más cercana al descanso que haya de ese lado, pues eso maximiza la cobertura a otros pisos y al otro lado del descanso en el mismo piso.

En otras palabras, luego de los coordinadores que puso el goloso en cada piso, a lo sumo habrá que agregar 1 o 2 por piso, que deberán estar sí o sí en las habitaciones más cercanas al descanso de cada lado. Luego en cada piso nos quedan solo 4 posibilidades para elegir: Poner 0 coordinadores adicionales, poner uno a izquierda, uno a derecha, o poner los dos. Con nuestro algoritmo de programación dinámica consideramos las 4 opciones y tomamos la mejor.

Esto se puede ver como un problema de camino mínimo: para construir nuestra solución iremos eligiendo en cada piso una de 4 opciones, cada una de las cuales nos llevará a un estado diferente, y el costo de cada movimiento viene dado por la cantidad de coordinadores que estamos utilizando según la opción que elegimos (es decir, las "aristas" de los movimientos tienen pesos 0, 1 o 2). Queremos llegar al final a un estado donde hayamos cubierto todas las habitaciones S del hotel, con costo mínimo. Se podría utilizar el algoritmo de Dijkstra, pero como el proceso no tiene ciclos (pues siempre pasamos al piso siguiente), se puede utilizar directamente el método de programación dinámica.

[1] En realidad, que su distancia al descanso sea T o más, pues si es exactamente T, como no hay coordinador posible justo en el descanso, necesariamente la cubre un coordinador que no cruza el descanso.

El estado que se debe guardar es un par (i, k) donde i indica el piso actual que vamos a procesar, y k es un entero que puede ser positivo o negativo: Un valor positivo de k significa que de los pisos anteriores, gracias a un coordinador que ya pusimos hay un tiempo remanente k desde el descanso, que puede servirnos para cubrir habitaciones en este piso y posteriores. Un valor negativo de k indica que en los pisos anteriores quedaron cosas sin cubrir, y por lo tanto para llegar a alcanzarlas estamos obligados a poner un coordinador a un tiempo máximo $-k$ del descanso, ya sea en este piso o posteriores. Un valor $k = 0$ equivale a no tener restricciones, y es la situación inicial: no podemos usar coordinadores anteriores, pero tampoco quedó nada pendiente de cubrir en pisos anteriores.

Una vez que sabemos todos los coordinadores que se ubican en este piso, podemos calcular su aporte para pisos futuros y ver si cumplen la deuda pendiente de pisos anteriores (Recordar que con el método de programación dinámica vamos a probar las 4 opciones posibles en cada piso). Si el más cercano de los coordinadores está a una cierta distancia d del descanso, como los coordinadores pueden viajar un tiempo T, este coordinador llega al descanso con un sobrante de tiempo $s = T - d$ (si $d \geq T$ podemos ignorarlo y tomar $s = 0$, pues este coordinador no puede aportar nada yendo hasta el descanso).

Si $k < 0$ y $-k \leq s$, se cubre lo que se necesitaba de pisos anteriores y ahora el sobrante es simplemente $k_{nuevo} = s$, mientras que sino, sigue siendo necesario cubrir lo anterior y el coordinador de este piso no sirve, por lo que seguiremos teniendo el mismo $k_{nuevo} = k$. Si en cambio $k \geq 0$, nos quedamos como sobrante para futuros pisos con $k_{nuevo} = \text{máx}(k, s)$.

Finalmente, al pasar al piso siguiente hay que actualizar este sobrante k_{nuevo} restando el tiempo t_e que insume un piso de escalera. Si k_{nuevo} era negativo, al restar sigue siendo negativo (la deuda de cubrir una habitación anterior no desaparece), pero si era no negativo, no puede volverse negativo sino que lo dejamos en 0 (pues si el sobrante ya no nos sirve porque estamos lejos, igualmente no se convierte en deuda).

La complejidad de este método, adicional al costo lineal del algoritmo goloso, es $O(PT_{max})$, proporcional a los estados recorridos con programación dinámica. Como el tiempo máximo posible T_{max} está acotado por $1000 + P \cdot t_e + 1000 \leq 3000$, esta complejidad es suficiente para resolver el problema eficientemente en el tiempo provisto.

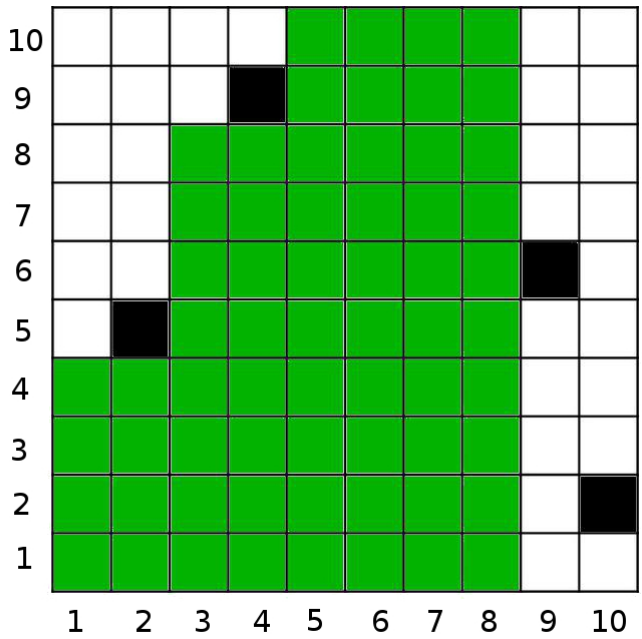
4.3. Nivel 3

4.3.1. Problema 1: El Genio de la Lámpara [genio]

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

El problema puede resumirse de la siguiente forma. Se tiene una grilla de $N \times M$, con A casilleros donde hay artefactos malditos. Se debe elegir *un número* $K \leq M$, y *un arreglo creciente de* K *posiciones* llamado `H` que corresponde al hechizo del genio. Este hechizo tiene en el $j - simo$ lugar a la altura que se elige para cada columna con índices desde 1 hasta K.

El hechizo abarca a todas las posiciones (i, j), con $1 \leq j \leq K$ y $1 \leq i \leq$ `H[j]`. No cualquier hechizo que cumpla estas condiciones será válido. Además, para que un hechizo sea válida se pide que entre todas las posiciones que abarca, no exista un artefacto maldito. En el siguiente ejemplo, se muestra en verde a las casillas que abarca el hechizo y en negro a los artefactos malditos, con valores de $K = 8$ y un hechizo `H` $= [4, 4, 8, 8, 10, 10, 10, 10]$.



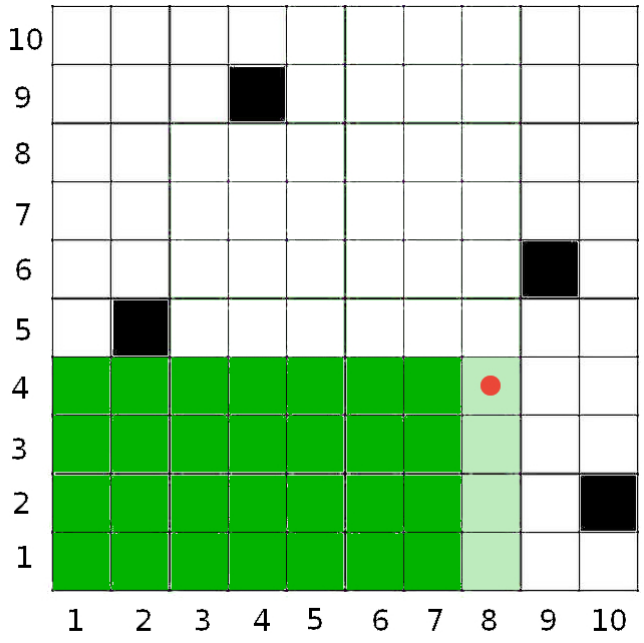
Naturalmente, el problema nos pide encontrar la *mayor cantidad de casillas que puede abarcar un hechizo válido*. Veamos una primera idea para calcular esto:

Consideremos $f(i, j)$ a la mayor cantidad de casillas que abarca un hechizo que tiene como $K = j$ y `H[j] = i` (es decir que la casilla superior derecha del hechizo se alcanza en (i, j)). Claramente $f(i, j) = -\infty$ (inválido) si en alguna casilla

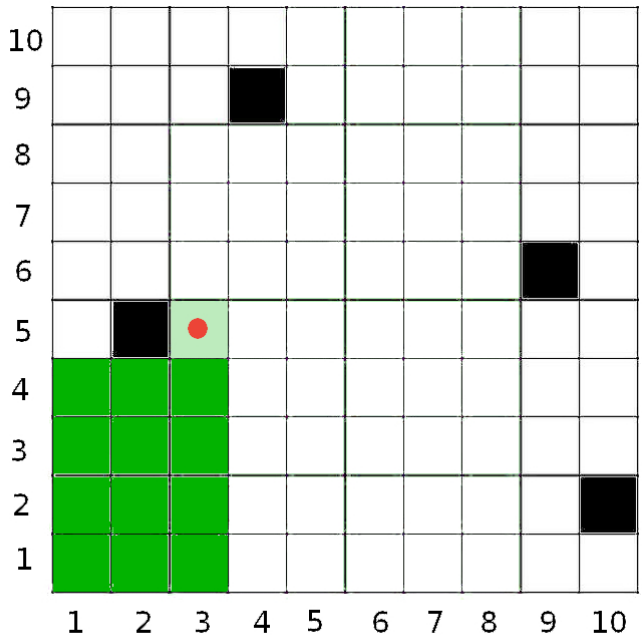
de esa columna j existe un artefacto maldito en alguna fila r con $1 \leq r \leq i$, pues cualquier hechizo con `H[j] = i` **sería inválido**. Si tenemos que pensar gráficamente a las casillas donde $f(i,j) = -\infty$, podemos partir desde cada artefacto maldito, todas las casillas que estén por encima (pero en la misma columna), tendrán como $f(i,j) = -\infty$.

Ahora debemos calcular $f(i,j)$ en las casillas restantes. Naturalmente, la solución al problema será el mayor valor de $f(i,j)$ para algún (i,j) en el tablero. Podemos calcular $f(i,j)$ a partir de los valores de $f(i,j)$ en casillas vecinas. Concretamente, tenemos dos opciones para extender a una solución existente.

Si utilizamos una solución con casilla superior derecha en la casilla inmediatamente a la izquierda de (i,j), es decir $(i,j-1)$, podemos obtener una solución agregando toda la columna j hasta la fila i, obteniendo una solución que abarca $f(i,j-1)+i$ casillas. En la siguiente figura se ejemplifica este caso con $i = 4$ y $j = 8$. En verde oscuro marcamos a las casillas abarcadas por la solución brindada por $f(i,j-1)$, y con verde claro las i que se agregan a la nueva solución.



Otra opción es extender una solución utilizando que conocemos la solución de la casilla que está inmediatamente por debajo. En este caso, simplemente se agrega una casilla a la solución, como se muestra en la siguiente figura con $i = 5$ y $j = 3$.



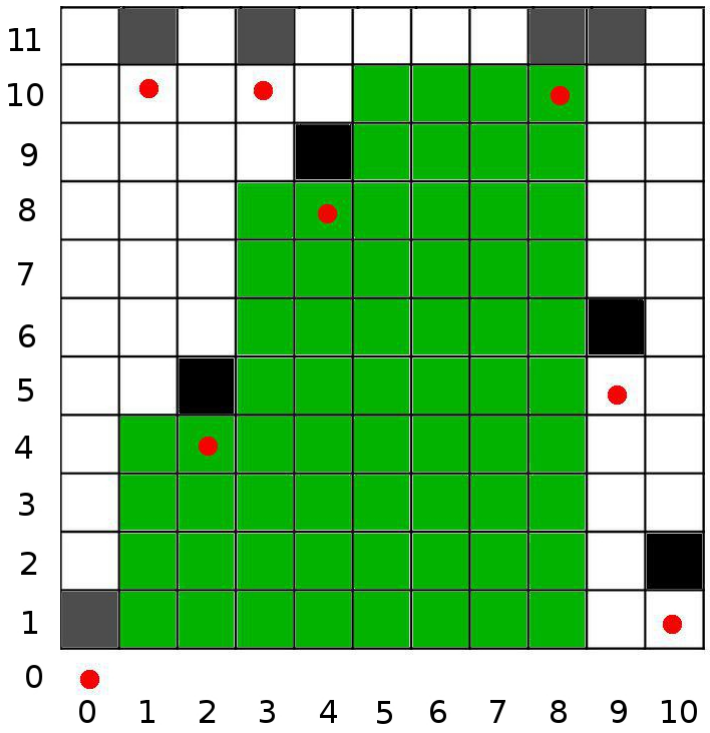
Esta recursión, que si se quiere escribir explícitamente sería de la forma $f(i,j) = \text{máx}\{f(i-1,j)+1, f(i,j-1)+i\}$, nos da una solución posible al problema con complejidad $\mathcal{O}(N \cdot M)$. Una forma de implementarla sería recorrer la grilla de forma *bottom-up* (de forma que al querer calcular $f(i,j)$ ya estén calculados los valores de $f(i,j-1)$ y $f(i-1,j)$). Veamos si podemos mejorar esto.

La siguiente idea será analizar si toda elección de (i,j) tiene sentido. Si logramos descartar algunas soluciones (porque encontramos otras que *dominan* a esta solución), no hace falta que las consideremos. En este espíritu analicemos los siguientes casos motivados por lo visto en las transiciones anteriores:

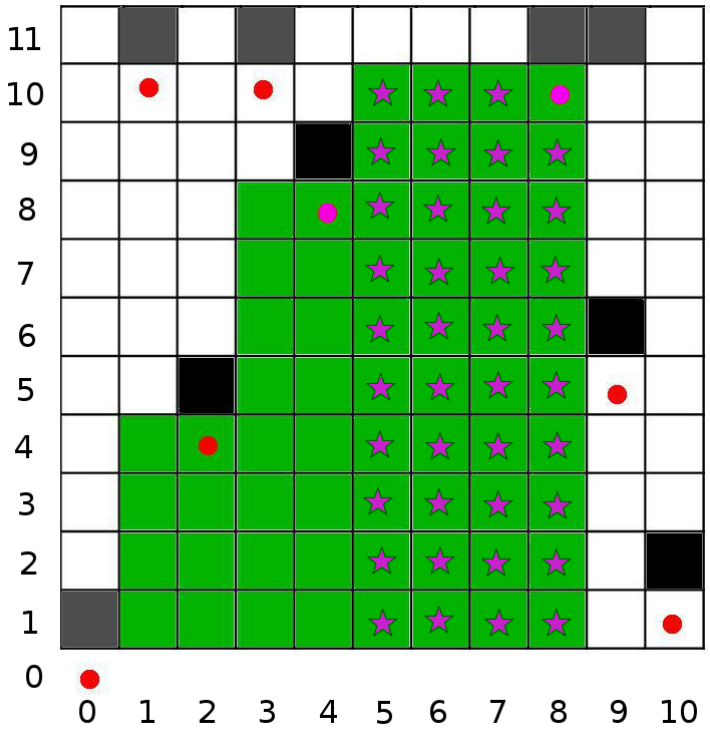
- ¿Podrá ser solución al problema un hechizo con casilla superior derecha en (i,j) si sabemos que la casilla $(i+1,j)$ está libre? No, pues podemos tomar el mismo hechizo que tiene la solución en (i,j), y aumentar la última coordenada del hechizo en una unidad, obteniendo una solución que sigue siendo válida (pues H sigue siendo creciente, y solo cambiamos una casilla), y alcanza un mayor valor.
- ¿Podrá ser solución al problema un hechizo con casilla superior derecha en (i,j) si sabemos que todas las casillas de la forma $(r,j+1)$ con $1 \leq r \leq i$ están libres? No, pues podemos extender el hechizo de manera similar, obteniendo una mejor solución.

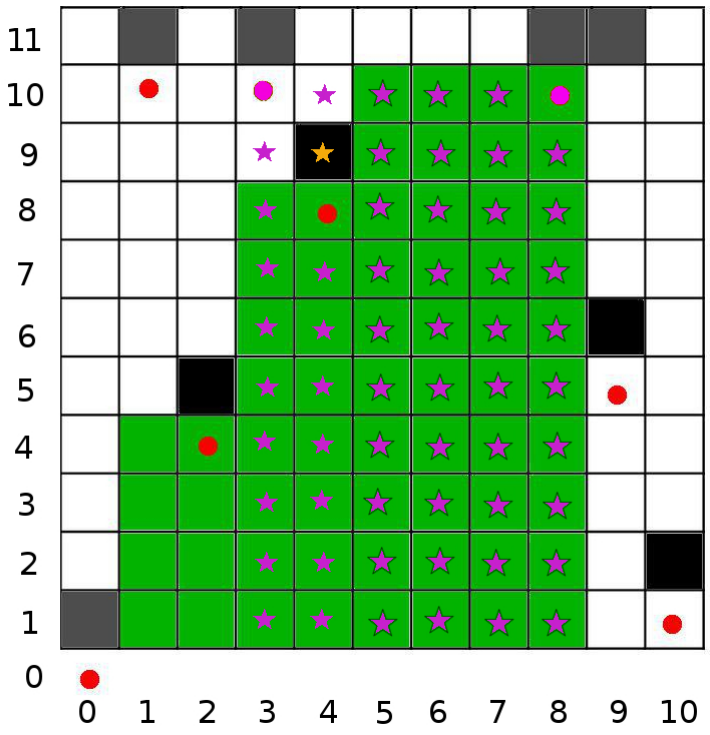
Por lo tanto, por cada artefacto maldito, tendremos a lo sumo dos soluciones

candidatas a analizar. Una, la que tiene su casilla superior derecha en la casilla que está inmediatamente por debajo (no permitiendo que se extienda en vertical su última columna), y otra en la casilla de la columna anterior a la del artefacto y en la fila más alta que tenga todas sus casillas con fila menor libres. En la figura se muestran los candidatos marcados con un círculo rojo. Además de los candidatos se agregó en la figura a una fila y una columna extra en los bordes, junto con un *candidato centinela* que puede ser de ayuda en la implementación (para evitar casos bordes).



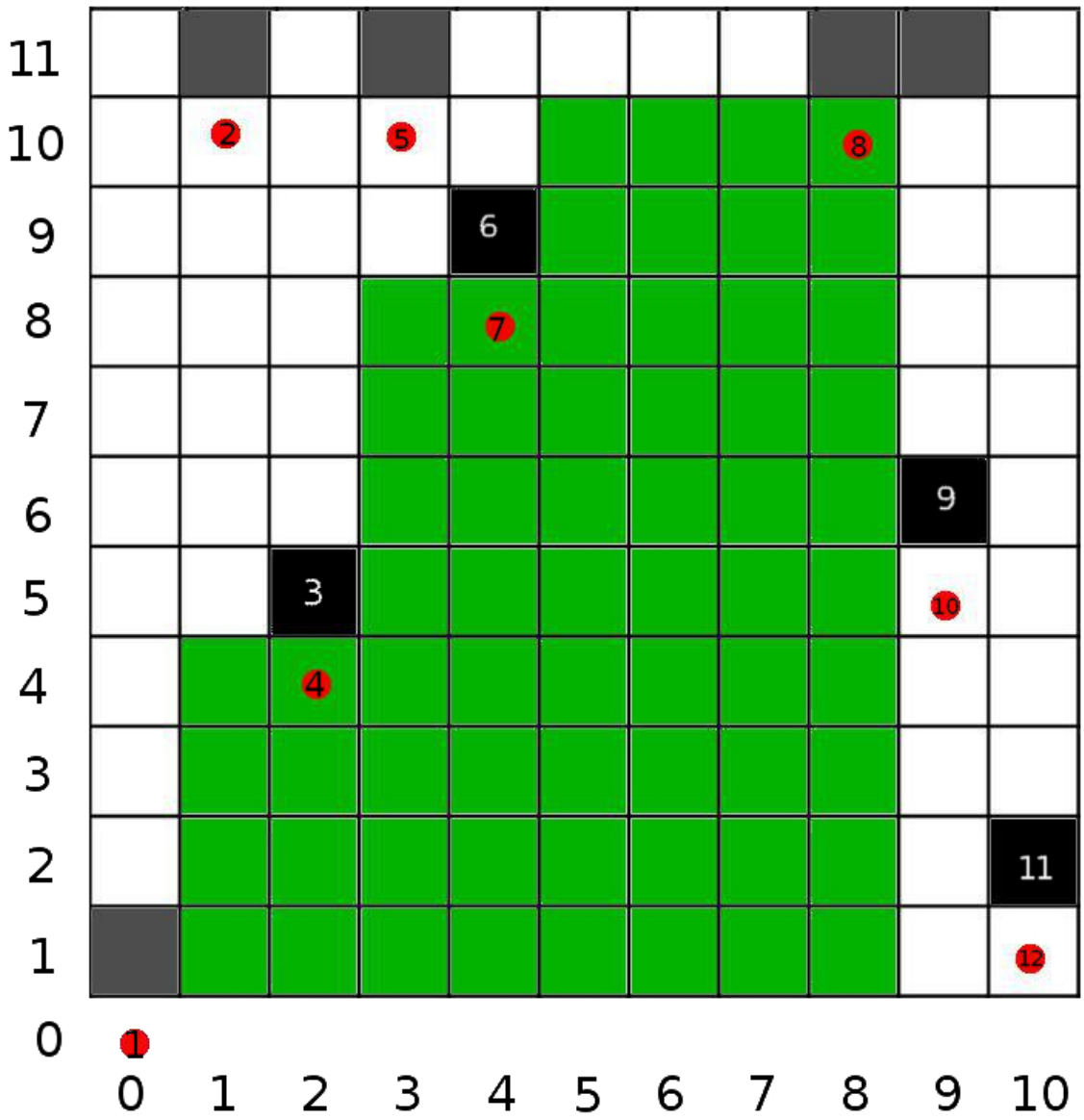
Supongamos entonces que tenemos ordenados a los candidatos (puntos rojos del ejemplo) crecientes en las columnas, y en caso de empate en las columnas, decrecientes en las filas (es decir, de izquierda a derecha y de arriba a abajo). Si estamos en el k-ésimo candidato (notar que es minúscula, no confundir con K el largo del hechizo), situado en la casilla (i_k, j_k), entonces una opción es iterar entre todos los candidatos anteriores y verificar si es factible unir ambas soluciones. Para ello, al analizar si podemos utilizar la solución de un candidato q con $q < k$, necesariamente deberá ocurrir que $i_q < i_k$ (el hechizo debe ser creciente) y $j_q < j_k$ (pensar por qué no puede haber dos candidatos en la misma columna). Además deberá estar libre de artefactos malditos el rectángulo con esquinas opuestas (i_k, j_k) y $(1, j_q + 1)$, como se muestra en las siguientes figuras (en la primera se puede, mientras que en la segunda no es válido).





Según cómo hagamos esta verificación, podremos resolver distintas subtareas. Una forma relativamente sencilla y eficiente resulta de utilizar una técnica que suele denominarse *sweep line* (barrido de línea). Cabe destacar, que este nombre es el de la técnica más general y muchas veces se dirá que un problema utiliza esta técnica ignorando las partes propiamente del problema.

Lo que vamos a hacer es aprovechar el orden que le dimos a los candidatos (cabe aclarar, el orden propuesto anteriormente no fue arbitrario) y vamos a agregar en el orden de los candidatos a los artefactos malditos. De esta forma, vamos a recorrer tanto a candidatos como a artefactos malditos en dicho orden, tal y como se muestra en la figura. Además, vamos a mantener una pila auxiliar mientras recorremos cuyo tope guarde *el candidato más a la derecha que es factible ubicar antes del candidato actual*.



Al encontrarnos con un candidato en el recorrido vamos a asignarle como candidato anterior al tope de la pila, y luego lo vamos a agregar como nuevo tope a la pila. Al encontrarnos con un artefacto maldito vamos a retirar candidatos de la pila, hasta que el tope de la pila esté en una fila menor a la del artefacto maldito.

De esta forma nos aseguramos que la región buscada entre el candidato en cuestión y el tope de la fila está libre, pues, de haber un artefacto en ese rectángulo hay dos opciones, según si está en una fila mayor a la del tope de la pila o si está en una fila menor o igual.

En el primer caso, tendríamos que deberíamos haber apilado al candidato generado por el artefacto maldito en el rectángulo, contradiciendo quién es nuestro tope de la pila actual. En el segundo caso, la presencia del artefacto hace que deberíamos haber desapilado al tope de la pila en cuestión. Esto garantiza que el rectángulo está libre hasta la columna donde está el candidato en cuestión, pero por cómo generamos los candidatos, sabemos que debajo del candidato está toda la columna libre.

Más aún, puede verse que por cómo se generan los candidatos a partir de los artefactos malditos, no puede haber dos candidatos válidos anteriores para un candidato, por lo tanto con este método en realidad estamos obteniendo al único candidato que puede venir anteriormente.

De esta forma, calculando el área de los rectángulos en cuestión entre candidatos, podemos calcular $f(i, j)$ para los candidatos (cuya cantidad es menor o igual a $2A$), lo cual concluye una solución eficiente al problema.

4.3.2. Problema 2: Creando un emporio [emporio]

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

Para pensar este problema, conviene plantearlo en términos de grafos: pondremos un nodo por cada uno de los P puntos de interés, y una arista por cada una de las R rutas que los interconectan.

Lo primero que debemos entender para poder resolver este problema es el **costo**[2] que produce cada arista. La arista corresponde a una ruta, y cada ruta tiene tres atributos:

- T, la cantidad de personas que la transitan diariamente.
- D, la cantidad de dinero que cada una de esas personas gastaría.
- M, el costo de manteniemiento fijo, que no depende de la cantidad de personas.

Si en esta ruta se instala el paseo, entonces, en total el empresario tendrá un costo M por su instalación, pero a este costo hay que restarle lo que recupera como

[2]Podríamos trabajar con el beneficio y tener todos los signos al revés, es igual.

ganancia de lo que compran las personas: gastarán en total $T \cdot D$, por lo cual el costo final es $c = M - T \cdot D$. Si este costo es negativo, la ruta es un buen lugar para instalar un paseo comercial y de hecho genera ganancia neta para el empresario.

Por esta razón, **siempre conviene crear paseos comerciales en todas las rutas con** $c < 0$. Para ver que esto es cierto, podemos suponer que tuviéramos una solución en la cual en una de estas rutas no se creó paseo comercial. Al incorporar un paseo comercial en esa ruta, como $c < 0$ el costo total baja (equivale a decir que la ganancia neta es mayor), y si los P puntos de interés estaban conectados por paseos en la solución anterior, al agregar una ruta con paseo siguen conectados. Con lo cual, si en una solución no incorporamos el paseo, podemos obtener otra solución válida y mejor incorporándolo. Es por esto que todas las rutas con $c < 0$ van a tener paseo, y debemos incorporarlas.

De las que quedan, la intuición es que como tienen $c \geq 0$ generan un costo y no querríamos crear paseo allí, pero deberemos construir algunos para tener un emporio válido, es decir, para que todos los puntos de interés estén interconectados mediante paseos comerciales.

Este problema es muy similar al de árbol generador mínimo: http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/arbol-generador

En efecto, si imaginamos que cada componente conexa que se forma al mirar las aristas con $c < 0$ fuera un único nodo, y las aristas restantes con $c \geq 0$ unen estos nodos, como debemos interconectarlos con costo mínimo y para eso no vamos a querer usar ciclos (pues al no quedar aristas negativas, los ciclos nunca benefician), lo que necesitamos es un AGM de estos nodos correspondientes a las componentes.

Alternativamente, una implementación más sencilla es aprovechar que el algoritmo de Kruskal para AGM ya va formando "componentes" en su ejecución (a saber, tiene las componentes representadas por la estructura de union-find: http://wiki.oia.unsam.edu.ar/algoritmos-oia/estructuras/union-find). Lo que podemos hacer es correr el algoritmo de Kruskal, aprovechando que como ordena las aristas por c, primero recorrerá todas las aristas con $c < 0$, que son las que queremos agregar sí o sí. La única salvedad que hay que hacer entonces es que, a diferencia del algoritmo de Kruskal común, para las aristas con $c < 0$ no vamos a verificar si forman ciclo antes de agregarlas, sino que siempre las agregaremos al conjunto solución directamente.

4.3.3. Problema 3: Respondiendo pedidos de Radiotaxi [radiotaxi]

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

En este problema se presentan T taxis en ubicaciones (x_i, y_i) de la grilla, y lo mismo ocurre con C clientes que se encuentran ubicados también sobre la grilla.

El tiempo de viaje en este problema entre dos puntos de la grilla (calles de la ciudad) viene dado por la llamada **distancia manhattan**. Esta es la distancia que es necesario recorrer en una grilla cuadriculada para viajar entre dos intersecciones: Así, la distancia entre los puntos $(0, 1)$ y $(2, 2)$ es de 3 cuadras, a pesar de que físicamente su distancia en línea recta sería de $\sqrt{5} = 2{,}236$.

Más precisamente, si tenemos un punto (x_1, y_1) y un punto (x_2, y_2), su distancia manhattan es $|x_1 - x_2| + |y_1 - y_2|$, es decir, "la diferencia de las x más la diferencia de las y" (o dicho de otra manera, la cantidad de cuadras norte-sur que hay que recorrer, más la cantidad de cuadras este-oeste). En este problema, esta distancia es importante pues nos indica el tiempo de viaje desde la ubicación de un taxista, hasta la ubicación de un cliente que desee levantar.

Como hay suficientes taxis para levantar a todos los clientes (pues se garantiza que $C \leq T$), debemos encontrar para cada cliente qué taxi lo levantará. Si los tiempos correspondientes (calculables con la distancia manhattan, una vez elegida la asignación que nos dice qué taxi va con cada cliente), son $t_1, t_2, \cdots, t_C$, **el más grande de ellos** corresponderá al último cliente en subir a un taxi, y por lo tanto es lo que nos interesa para este problema. Es decir, queremos encontrar la asignación que **minimice** $\max(t_1, t_2, \cdots, t_C)$.

4.3.3.1. Subtarea $T \leq 10$

Para esta subtarea, es posible iterar exhaustivamente todas las asignaciones posibles, hacer la cuenta con cada una, y tomar la mejor (ya que hay a lo sumo $10! = 10 \cdot 9 \cdot 8 \cdots 2 \cdot 1 = 3628800$ asignaciones como máximo.

4.3.3.2. Subtarea $C = 1$

En este caso, simplemente podemos iterar todos los T taxis para calcular el tiempo que cada uno toma (calculando la distancia manhattan hasta el único cliente), y tomar el mínimo de todos como la respuesta. Tiempo: $O(T)$

4.3.3.3. Subtarea $C = 2$

Podemos hacer una pequeña variación sobre la idea anterior: como solamente hay dos clientes, para cada uno de ellos hacemos lo mismo que en la subtarea anterior, revisando para todos los taxis el tiempo al cliente y seleccionando al mejor.

El único caso para considerar es aquel en que el mejor taxi sea el mismo para ambos clientes, ya que en ese caso no se puede asignar a ambos. En ese caso, conviene asignarlo a alguno de los 2, y como son solo 2 clientes, probamos ambas posibilidades:

- Si lo asignamos al primer cliente, queda un problema con un solo cliente y un taxi menos, y lo resolvemos como la subtarea anterior.
- Lo mismo ocurrirá si decidíamos usar el mejor taxi para el segundo cliente.
- Probamos ambas posibilidades, y nos quedamos con la que haya dado un tiempo menor.

La complejidad de vuelta es $O(T)$

4.3.3.4. Subtarea de calle única y $C = T$

En este caso particular, todos los taxis y clientes están sobre una misma calle, por lo cual tenemos el problema en una sola dimensión. En efecto, la parte $|y_2 - y_1|$ de la distancia Manhattan será siempre 0, y la distancia será simplemente la diferencia de x.

Para este caso particular en que hay la misma cantidad de taxis que de clientes, se puede utilizar un algoritmo goloso especial que resuelve el problema: **ordenar** todos los clientes y taxis por coordenada x, y luego emparejar el primer cliente con el primer taxi, el segundo cliente con el segundo taxi, y así siguiendo.

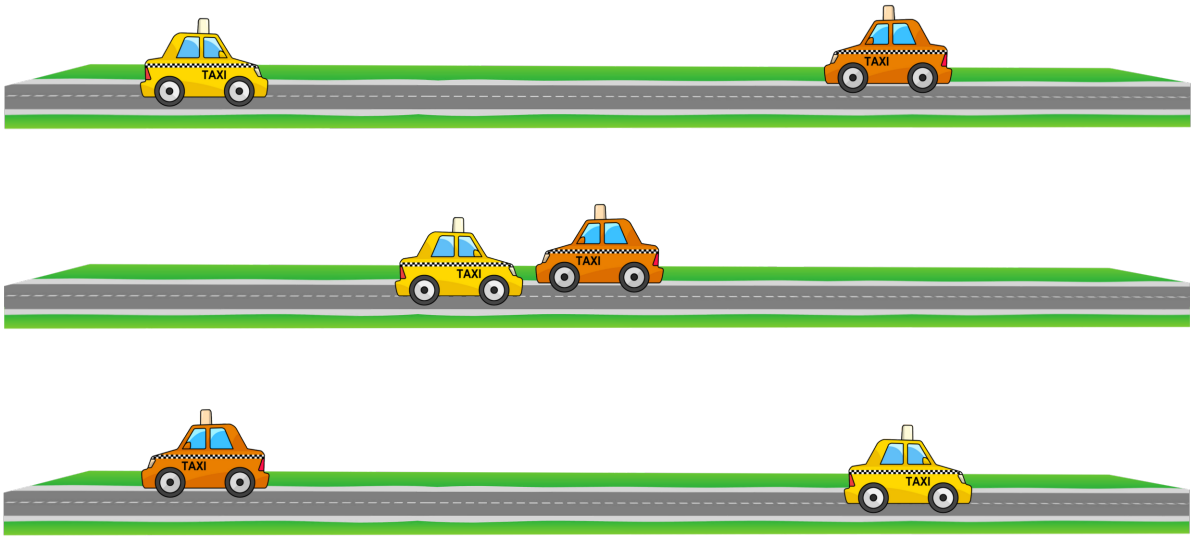
La razón por la que esto es así es que, si no los emparejáramos de esta forma, tendríamos una solución en la que hay *cruces*: dos clientes en posiciones $c_1 < c_2$ y dos taxis en posiciones $t_1 < t_2$, pero de tal forma que c_1 es levantado por t_2 y c_2 es levantado por t_1.

En este caso en que tenemos el cruce, los tiempos correspondientes a estos dos clientes son $|c_1 - t_2|$ y $|c_2 - t_1|$. Si los emparejásemos de la otra forma (es decir, en orden), de modo que c_1 fuera levantado por t_1 y c_2 por t_2, los tiempos serían $|c_1 - t_1|$ y $|c_2 - t_2|$.

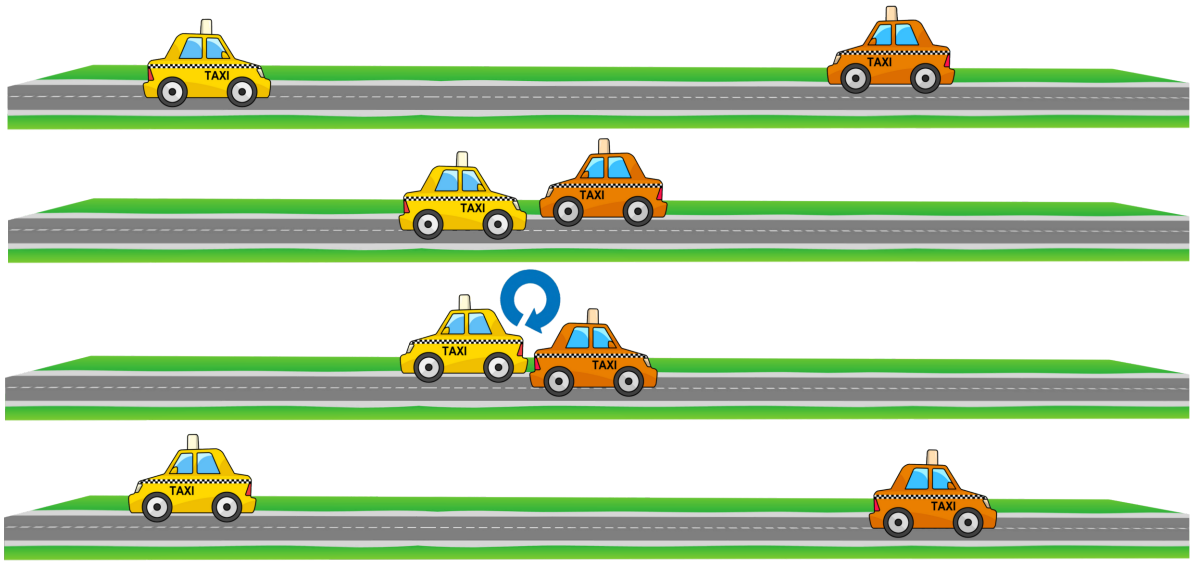
Se puede comprobar en este caso, gracias al orden, que siempre resulta $\max(|c_1 - t_1|, |c_2 - t_2|) \leq \max(|c_1 - t_2|, |c_2 - t_1|)$. Por esta razón, **dar vuelta un**

cruce no empeora la solución. Con lo cual, si empezando en un emparejamiento cualquiera, damos vuelta cruces hasta que ya no haya más, no tendremos un emparejamiento peor: es decir, cualquier emparejamiento es peor o igual al ordenado. Por esta razón, el ordenado (que es el único sin cruces) es el mejor.

Para razonar por qué dar vuelta un cruce no empeora, observemos que si tenemos una situación de cruce, e imaginamos el viaje de los taxis hasta el cliente: inicialmente, $t_1 < t_2$, pero como al final t_1 termina donde lo espera c_2, y t_2 termina donde lo espera c_1, los taxis se cruzan en algún momento del camino:



y por lo tanto en ese instante de tiempo, se encuentran en el mismo lugar. Por lo tanto, en ese momento ambos taxis son indistinguibles, y para lo que queda de trayecto, da igual a fines de tiempos que cambien de plan y t_1 prosiga hasta c_1 y t_2 hasta c_2:



Por lo tanto, es posible que t_1 busque a c_1 y t_2 busque a c_2 sin tener un peor tiempo total, como afirmamos antes.

4.3.3.5. Subtarea de calle única

Solución con programación dinámica:

De la observación que hicimos en la explicación de la subtarea anterior, podemos ver que no es conveniente que dos taxis se crucen en su camino. Es decir, una vez que decidimos **cuáles** son los C taxis que van a levantar a los clientes, la asignación de qué taxi usar para qué cliente queda ya determinada como en la sección anterior: el primer taxi irá con el primer cliente, el segundo con el segundo, y así siguiendo (cuando los ordenamos por aparición en la calle).

Esto da lugar a una solución de programación dinámica con complejidad $C(T-C)$. Planteamos $dp(t, c)$, con $t \geq c$, como el mínimo tiempo para que los primeros c clientes estén subidos a un taxi, si se los asigna de forma óptima a algunos de los primeros t taxis. La solución al problema será entonces $dp(T, C)$.

La clave el que como el taxi de más a la derecha va con el cliente de más a la derecha, hay solo dos opciones en cada paso: o bien no utilizar el taxi de más a la derecha, o bien utilizarlo, y entonces sabemos que va a buscar al cliente de más a la derecha.

Si no utilizamos el taxi de más a la derecha, la mejor solución posible es $dp(t-1, c)$. Y si lo usamos, para asignar a los $c-1$ clientes restantes lo mejor posible será $dp(t-1, c-1)$, pero hay que tener en cuenta que como el taxi t busca al cliente c, el tiempo final será $max(dp(t-1, c-1), dist(t, c))$

Entonces, la recursión queda:

$$dp(t,c) = \begin{cases} \text{mín}(dp(t-1,c), \text{máx}(dp(t-1,c-1), dist(t,c))) & \text{si } t \geq c > 0 \\ +\infty & \text{si } t < c \\ 0 & \text{si } c = 0 \end{cases}$$

El segundo caso ya que cuando $t < c$, no alcanza la cantidad de taxis para los clientes, y entonces es imposible cumplir lo pedido, así que indicamos el peor valor posible de tiempo, para que la fórmula recursiva prefiera siempre la otra opción posible cuando existe.

Solución con búsqueda binaria y algoritmo goloso:

La solución eficiente esperada para este problema se basa en utilizar búsqueda binaria en la respuesta. Es decir, iremos probando con búsqueda binaria distintos valores posibles X del tiempo máximo permitido, y en cada paso tendremos que ver si es posible asignar de modo que ningún cliente tarde más de X en subir a su taxi. Si esto es posible, en la búsqueda binaria pasamos a probar valores más chicos, y si no es posible, valores más grandes, hasta encontrar el valor extremo que da la respuesta.

Para saber si con un cierto valor X es posible, lo que nos estamos preguntando es si es posible asignar a cada cliente un taxi que no esté a más de X unidades de distancia. Es decir, si el taxi está en la posición t y el cliente en la posición c de la calle, tiene que ser $|t - c| \leq X$. O sea que por cada cliente, tenemos un **intervalo** de taxis posibles que pueden levantarlo. Se puede demostrar que en este caso, conviene usar el primer taxi posible (el que tenga menor coordenada x) para levantar al primer cliente, y así siguiendo, el menor taxi posible (entre los restantes) para levantar al segundo cliente, hasta que se hayan cubierto todos los clientes o quede un cliente sin taxis disponibles a distancia aceptable. Este método goloso determina correctamente en este caso si es posible levantar a todos los clientes sin pasarse del tiempo permitido X.

4.3.3.6. Caso general

En el caso general, podemos hacer búsqueda binaria en la respuesta, de manera exactamente idéntica a lo que planteamos en la subtarea anterior, pero ahora habrá que cambiar la verificación de factibilidad de un cierto valor solución al tener la grilla 2D en lugar de 1D.

Lo que podemos hacer es, una vez fijada la distancia manhattan máxima

permitida, calcular las distancias entre los distintos pares de clientes y taxis, y solo quedarnos con las parejas cuya distancia no supere la máxima.

Con todas estas parejas podemos formar un grafo bipartito[3], que tenga clientes por un lado y taxis por otro. Cada pareja válida corresponde a una arista entre el cliente y el taxi asociados. Como un matching asigna clientes a taxis, lo que queremos saber es si existe un matching que asigne todos los clientes, o sea un matching de tamaño C. Para esto podemos utilizar un algoritmo de matching máximo bipartito, y verificar si es C: Si el tamaño de un matching máximo bipartito es C, es posible asignar a todos los clientes. Si el tamaño es menor, no es posible.

Finalmente, para que esto tenga una complejidad final razonable y alcance el tiempo de ejecución permitido, hay que aplicar una optimización que nos permite reducir el número de taxis en todos los casos:

Como el máximo de taxis es 100.000 pero el de clientes es 100, en los casos grandes hay muchísimos más taxis que clientes. Entonces quedarán muchísimos taxis sin usar.

En efecto, cada cliente querría usar su mejor taxi (el que tenga más cerca). La única razón para que no lo use, es que ya esté asignado. En ese caso, querría usar su segundo mejor taxi. La única razón para no hacerlo sería que también esté ya asignado. En ese caso, querría usar su tercer mejor taxi. En general, cada cliente quiere utilizar el taxi más cercano a él, que no sea uno de los que utilizan los demás clientes, ya que si esto no ocurriera la solución sería mejorable (o al menos, el valor de ese cliente, sin afectar a ningún otro).

Pero como solo hay $C - 1$ otros clientes además de él, en el análisis anterior podemos ver que un cliente como mucho utilizará el taxi número C, es decir que por cada cliente basta quedarnos con sus mejores C taxis, y esos son los únicos candidatos posibles. De esta manera, con un cómputo inicial de costo $O(CT \lg(CT))$ podemos reducir la cantidad de taxis a C^2 como máximo. Más aún, como por cada cliente tenemos identificados C candidatos, estos son los únicos que debemos considerar a la hora de colocar aristas en el grafo para analizar la factibilidad de una solución. Por lo tanto, la cantidad de aristas del grafo bipartito será $O(C^2)$, y entonces el costo de encontrar un matching máximo será $O(C^3)$, que es suficientemente bueno como para poder resolver este problema.

[3]Se puede leer en la wiki sobre grafos bipartitos y matching máximo: http://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/grafos-bipartitos/maximo-matching-bipartito