Solucionario de la Olimpíada Informática Argentina 2017

Autores

Brian Bokser

Sebastián Cherny

Agustín Santiago Gutiérrez

Facundo Martín Gutiérrez

Melanie Sclar

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: `fila`, `tiranic`, `ppt`, etc).

Capítulo 2

Selectivo para la IOI

2.1. Día 1

2.1.1. Problema 1: Armando la fila [fila]

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

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

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

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

4	5	8	2	7

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.1.2. Problema 2: Antártida [antartida]

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

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

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

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

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

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

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

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

estas dos alternativas. Concluímos que podemos obtener $maxhasta_i$ por medio de la siguiente fórmula:

$$maxhasta_i = \text{máx}(maxhasta_{i-1}, maxhasta_j + (b_i - a_i))$$

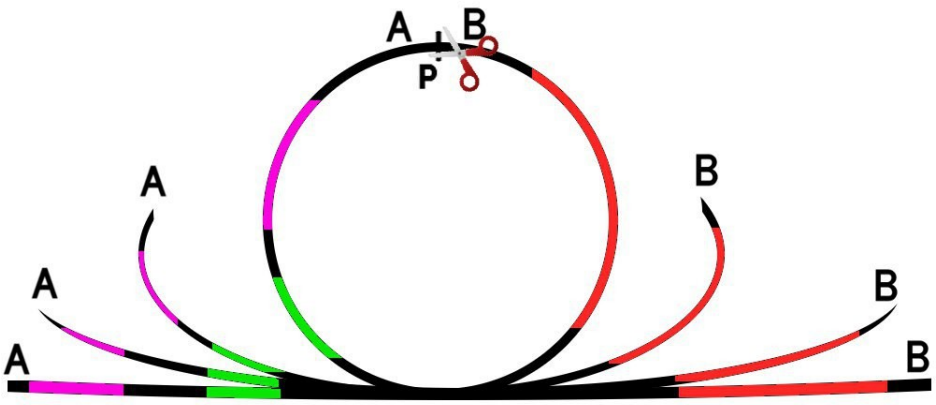
Donde j es el mayor índice tal que $b_j < a_i$. (Para solucionar el problema cuando no hay ninguno podemos definir $b_0 = -\text{ínf}$ y $maxhasta_0 = 0$)

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

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

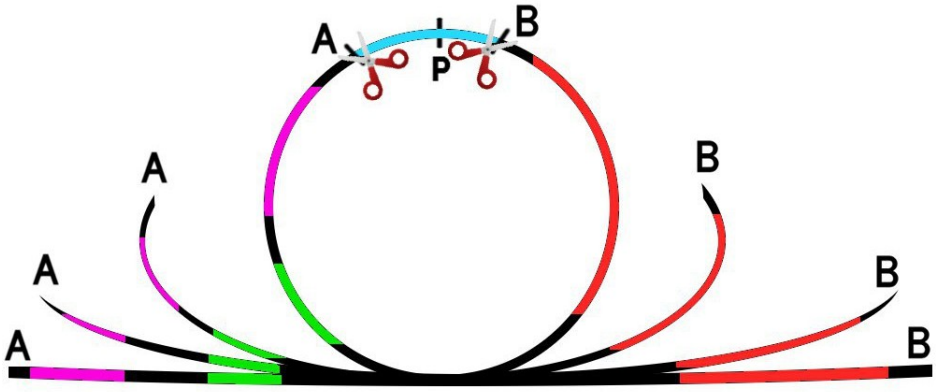
```
1: procedure MaximoCubrimientoIntervalos(intervalos)
2:     OrdenarPorFin(intervalos)
3:     maxhasta_0 ← 0
4:     N ← size(intervalos)
5:     for i ← 1...N do
6:         j ← calcularJ(intervalos, i)
7:         maxhasta_i = max(maxhasta_{i-1}, maxhasta_j + longitud(intervalos[i]))
8:     return maxhasta_N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

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



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

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



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

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

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

2.1.3. Problema 3: Encontrando el Tiranic [tiranic]

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

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

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

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

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

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

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

El problema incluye en su enunciado la siguiente garantía importantísima:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

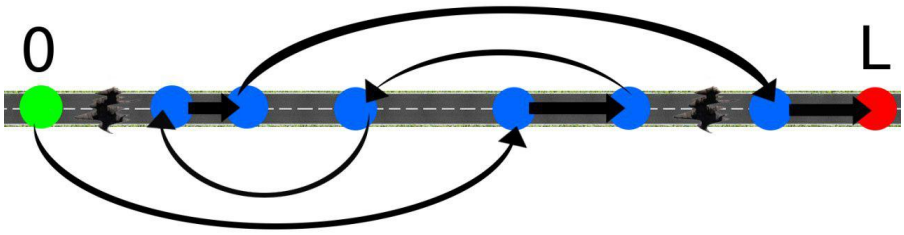
2.2. Día 2

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

2.2.2. Problema 2: Planificando la temporada [alquiler]

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

Este problema se puede resolver de dos maneras.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.3.1. Subtareas con afirmaciones atómicas

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

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

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

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

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

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

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

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

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

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

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

Los ejemplos más simples que hay son:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.3.2. Subtareas con afirmaciones generales

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

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

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

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

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

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

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

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

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

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

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

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

podas.

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

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

Capítulo 3

Certamen Jurisdiccional

3.1. Nivel 2

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

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

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

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

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

Algorithm 1 Solución al Problema 1 Nivel 2 Jurisdiccional 2017

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

```
#include<iostream>

using namespace std;

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

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

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

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

3.1.2. Problema 2: Días Feriados [feriados]

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

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

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

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

$\hat{f}_1$			$\hat{f}_2$		$\hat{f}_3$			$\hat{f}_4$	
↓			↓		↓			↓	
0	1	2	3	4	5	6	7	8	9
F	C	C	F	C	F	C	C	F	C

⟶

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

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

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

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

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

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

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

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

subarreglos posibles.

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

Solución 1:

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

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

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

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

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

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

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

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

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

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

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

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

Solución 2:

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

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

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

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

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

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

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

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

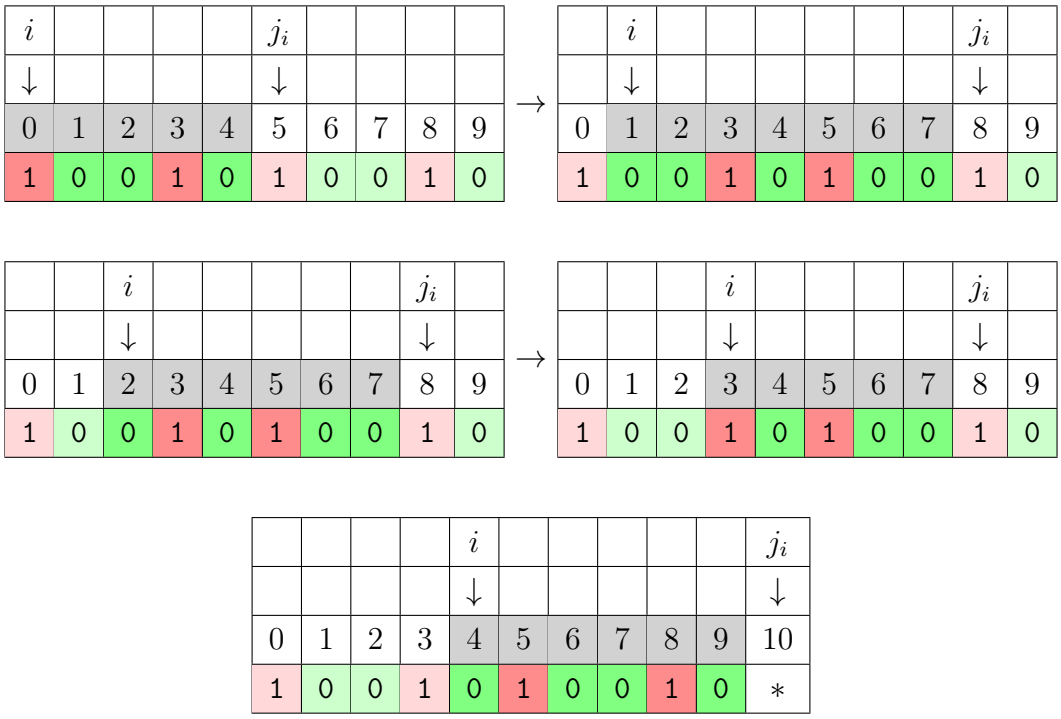
Solución 3:

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

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

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

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



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

3.1.3. Problema 3: Recorriendo Venecia [venecia]

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

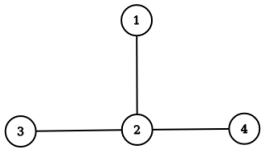
Para empezar, notemos que al problema lo podemos modelar como un grafo: Las esquinas son nodos, y las cuadras, que conectan siempre dos esquinas, son aristas. Entonces el problema lo vamos a pensar como, en un grafo, encontrar un camino que recorra todas las aristas, empezando y terminando en un mismo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

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

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

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

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



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

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

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

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

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

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

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

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

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

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

Algorithm 3 Solución al Problema 3 Nivel 2 Jurisdiccional 2017

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

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

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

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

using namespace std;

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

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

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

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

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

3.1.4. Problema 4: Tanques de Agua [tanque2]

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

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

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

El llenado sigue las siguientes reglas:

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

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

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

Algorithm 4 Pseudocódigo de la simulación de llenado

```
1: procedure SIMULARLLENADO(tanqueActual, arbolTanques, ordenLlenado)
2:     for hijo ← arbolTanques[tanqueActual] do
3:         simularLlenado(hijo, arbolTanques, ordenLlenado)
4:     ordenLlenado.agregarAlFinal(tanqueActual)
```

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

Hay una estrategia sencilla que funciona que es la siguiente:

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

¿Por qué esto funciona?

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

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

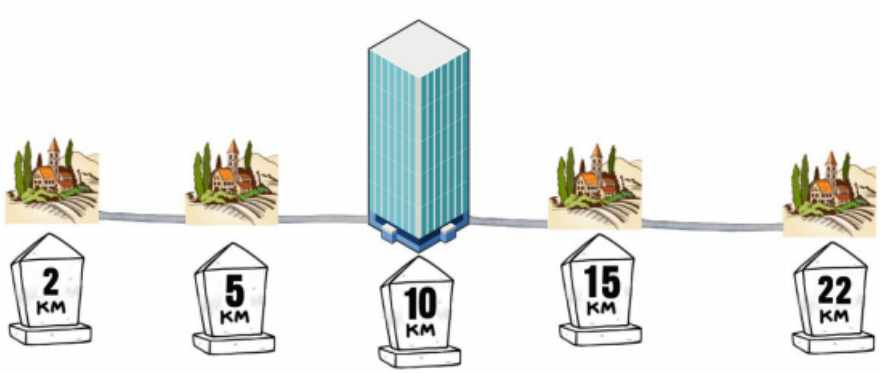
3.2. Nivel 3

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

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

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

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



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

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

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

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

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

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

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

En otras palabras: **La respuesta final siempre es el máximo de todos los números, menos el mínimo de todos los números**. Cuando decimos "todos los números", nos referimos a los números de kilómetro de todos los N pueblos, y también al número de kilómetro de la megacentral.

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

Algorithm 5 Solución al Problema 1 Nivel 3 Jurisdiccional 2017

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

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

```
#include <iostream>

using namespace std;

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

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

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

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

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

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

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

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

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

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

Algorithm 6 BFS / Cálculo de distancias en grafo no dirigido

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

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

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

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

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

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

3.2.3. Problema 3: Al-Garín [algarin]

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

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

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

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

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

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

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

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

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

$$joyasAnteriores(i,j) = max \left\{ \begin{array}{l} mejorJoyas(i-1,j), \\ mejorJoyas(i,j-1), \\ mejorJoyas(i-1,j-1) \end{array} \right\}$$

$$joyasEn(i,j) = \left\{ \begin{array}{ll} 1, & M[i][j] = \text{'J'} \\ 0, & M[i][j] = \text{'.'} \end{array} \right\}$$

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

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

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

Algorithm 8 MejorJoyas

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

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

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

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

3.2.4. Problema 4: Tanques de Agua [tanque3]

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

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

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

El llenado sigue las siguientes reglas:

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

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

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

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

Algorithm 10 Pseudocódigo de la simulación de llenado

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

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

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

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

Capítulo 4

Certamen Nacional

4.1. Nivel 1

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

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

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

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

$$\text{máx}(cantidadVenta, cantidadCompra_i) \cdot (precioVenta - precioCompra_i)$$

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

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

Veamos cómo quedaría un pseudocódigo para esta solución:

Algorithm 11 Solución al Problema 1 Nivel 1 Nacional 2017

```
1: procedure ProblemaNivel1Nacional2017(P, precioVenta, cantVenta, precioCompra, cantCompra, fabricante)
2:     maximaGanancia ← −1
3:     mejorFabricante ← −1
4:     for i ← 1 . . . F do
5:         if máx(cantVenta, cantCompra[i]) · precioCompra[i] ≤ P then
6:             gananciaActual ← máx(cantVenta, cantCompra[i]) · (precioVenta − precioCompra[i])
7:             if maximaGanancia < gananciaActual then
8:                 maximaGanancia ← gananciaActual
9:                 mejorFabricante ← i
10:    return maximaGanancia, mejorFabricante
```

4.1.2. Problema 2: Reconstruyendo el caminito [caminito]

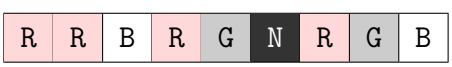
http://juez.oia.unsam.edu.ar/#/task/caminito/statement

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

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

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

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



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

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

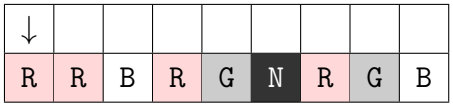
Una baldosa tiene a lo sumo 2 vecinos (notar que las baldosas en los extremos solo tienen 1 vecino). Por lo tanto, **cada baldosa puede tener a lo sumo 2 colores distintos entre sus baldosas vecinas** (pues cada vecino aporta a lo sumo un color distinto).

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

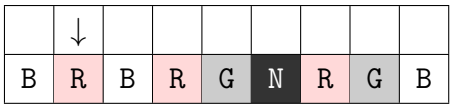
De aquí surge una forma clara de hallar la solución al problema.

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

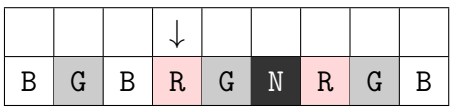
En el ejemplo anterior, se vería de la siguiente forma:

1. 

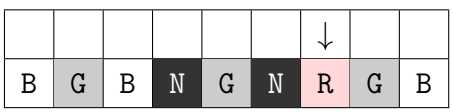
Utilizamos cualquiera de los 3 colores.

2. 

No podemos utilizar blanco

3. 

Solo podemos utilizar negro

4. 

Solo podemos utilizar blanco

5. 

Solución final

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

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

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

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

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

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

$$a\ b\ c\ d\ \mathbf{c\ b\ a}$$

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

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

$$a\ b\ c\ d\ ?\ ?\ ?$$

$$a\ b\ c\ d\ ?\ ?\ \mathbf{a}$$

$$a\ b\ c\ d\ ?\ \mathbf{b\ a}$$

$$a\ b\ c\ d\ \mathbf{c\ b\ a}$$

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

$$a\ b\ c\ d\ ?\ ?$$

$$a\ b\ c\ d\ ?\ \mathbf{a}$$

$$a\ b\ c\ d\ \mathbf{b\ a}$$

$$a\ b\ \boxed{c\ d}\ \mathbf{b\ a}$$

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

Veamos cómo sería un pseudocódigo para el algoritmo recién descripto.

Algorithm 12 Solución al Problema 3 Nivel 1 Nacional 2017

```
1: procedure PROBLEMA3NIVEL1NACIONAL2017(P)
2:     largo ← longitud(P)
3:     while ¬palindromoPosible(P, largo) do
4:         largo ← largo + 1
5:     return largo
```

Nota: "¬"significa negación. Si $palindromoPosible(P, largo)$ era verdadero, $\neg palindromoPosible(P, largo)$ será falso y viceversa.

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

```
1: procedure PALINDROMOPOSIBLE(P, largo)
2:     posible ← Verdadero
3:     for i ← 1 ... ⌊largo/2⌋ do
4:         if longitud(P) − i ≤ longitud(P) then
5:             charDerecha ← P[longitud(P) − i]
6:         else
7:             charDerecha ← P[i]
8:         if P[i] ≠ charDerecha then
9:             posible ← Falso
10:            break
11:    return posible
```

4.2. Nivel 2

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

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

El enunciado nos habla de dos tipos de entidades:

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

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

Enfoque cuadrático:

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

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

Algorithm 14 Ganancia de un par

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

Enfoque eficiente:

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

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

Algorithm 15 Ganancia cuadratica

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

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

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

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

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

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

Algorithm 16 Ganancia eficiente

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

4.2.2. Problema 2: Reconstruyendo el sendero [sendero]

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

Vamos a presentar dos maneras de resolver este problema.

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

1	2	3	4	5
N	R	R	R	B

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

1	2	3	4	5
N	G	B	G	B

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

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

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

X	Y	X	Y	...

o bien

Y	X	Y	X	...

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

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

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

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

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

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

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

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

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

Concretamente vamos a guardar **el menor costo necesario para colocar**

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

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

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

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

Algorithm 17 Solución al Problema 2 Nivel 2 Nacional 2017

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

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

4.2.3. Problema 3: Dictado de nivelación [prueba]

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

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

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

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

orden.

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

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

Ahora bien, como es un palíndromo sabemos que $i_1 = d_k$. Si a un palíndromo le quitamos su primer y último caracter el resultado seguirá siendo palíndromo, y en este caso quedaría la siguiente cadena de caracteres:

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

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

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

Solo nos resta encontrar lo siguiente:

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

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

Algorithm 18 Solución al Problema 3 Nivel 2 Nacional 2017

```
1: procedure Problema3Nivel2Nacional2017(P)
2:     agregarADerecha ← Prob3Nivel1(P)
3:     agregarAIzquierda ← Prob3Nivel1(reversoCadena(P))
4:     return min(agregarADerecha, agregarAIzquierda)
```

Algorithm 19 Revertir una cadena de caracteres

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

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

4.3. Nivel 3

4.3.1. Problema 1: Armando el negocio [compra]

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

Veamos que nunca necesitaremos comprarle a más de 1 fabricante.

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

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

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

4.3.2. Problema 2: Dictado de nivelación [nivelacion]

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

Daremos una solución recursiva al problema planteado. Esto quiere decir

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

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

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

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

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

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

- Si agregamos un caracter a la izquierda.

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

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

- Si agregamos un caracter a la derecha.

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

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

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

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

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

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

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

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

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

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

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

Algorithm 20 Solución al Problema 2 Nivel 3 Nacional 2017

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

4.3.3. Problema 3: Reconstruyendo la vereda [vereda]

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

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

Contamos a continuación las soluciones principales:

4.3.3.1. Solución exponencial mediante backtracking

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

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

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

Algorithm 21 Solución al Problema 3 Nivel 3 Nacional 2017

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

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

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

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

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

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

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

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

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

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

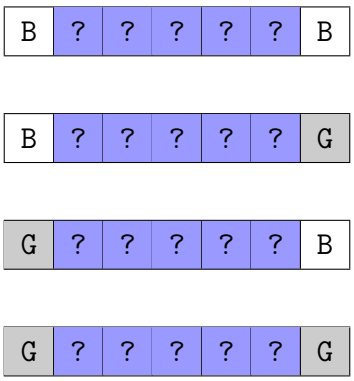
No obstante, es posible reducir la complejidad temporal a $O(n^3)$ observando que no todos los estados son posibles: Si llamamos T a la cantidad total de baldosas disponibles original (es decir, a la cantidad de letras `R` en la entrada), siempre tenemos $i + B + G + N = T$, ya que por cada baldosa que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

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

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

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

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

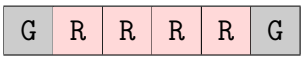


Gracias a esto, todos los grupos de R consecutivas quedan delimitados en sus dos extremos con baldosas de color, que es una situación más simétrica y fácil de pensar.

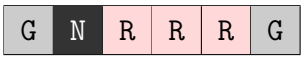
Luego para cada uno de esos 4 intentos independientes: mientras quede **al menos una baldosa disponible de cada color**, el algoritmo realiza los siguientes pasos:

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

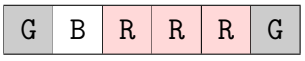
Por ejemplo, en una situación así:



Es siempre seguro pintar una baldosa así:

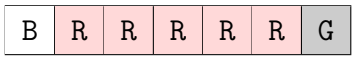


o así:

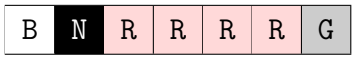


2. **Regla impar**: Si hay un grupo de R de longitud impar, bordeado por dos baldosas **de distinto color** (A y B), jugar el tercer tipo de baldosa (la C) en un borde es jugada segura.

   Por ejemplo, en una situación así:

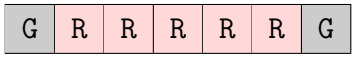


   Es siempre seguro pintar una baldosa así:



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

   Por ejemplo, en una situación así:



   Será seguro pintar una baldosa así:

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

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

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

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

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

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

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

1. Los grupos de R que tengan una G o una N en uno de sus extremos están determinados, ya que una sola de las dos opciones posibles es válida. Llenamos todos ellos de la única manera posible.

2. Quedan entonces únicamente los grupos que tienen en sus extremos dos baldosas B, que pueden llenarse alternadamente de cualquiera de las dos formas posibles.

3. En los grupos de longitud par, no importa cuál de las dos formas usamos, porque ambas usan exactamente la misma cantidad de baldosas de cada color.

4. En los impares, una de las opciones aumenta en uno la diferencia de "negros menos grises", y la otra la disminuye en 1. Basta entonces con elegir aquella que modifique la diferencia actual en la dirección correcta. Cuando la diferencia actual es correcta, se puede elegir cualquiera de las dos opciones libremente (quedando ya el próximo grupo condicionado).

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

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

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