Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Practico 1

Segundo Cuatrimestre 2014

Integrante	LU	Correo electrónico
Ricardo Colombo	156/08	ricardogcolombo@gmail.com.com
Federico Suarez	610/11	elgeniofederico@gmail.com
Juan Carlos Giudici	827/06	elchudi@gmail.com
Franco Negri	893/13	franconegri2004@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contenidos

Capítulo 1

Puentes sobre lava caliente

1.0.1. Introducción

Cada participante de una competencia debe cruzar un puente dando saltos de tablón en tablón, con la limitación de que pueden saltar como máximo una cantidad fija de tablones por vez. Sin embargo algunos de estos tablones están rotos. El objetivo del ejercicio es dar un algoritmo que nos devuelva un recorrido por los tablones del puente el cual sea mínimo en la cantidad de saltos requeridos para poder cruzarlo con una complejidad temporal de O(n) donde n es la cantidad de tablones del puente.

1.0.2. Ejemplos y Soluciones

Sea un puente de 14 tablones representado por el siguiente vector de 0 y 1, `[0,0,0,1,1,0,0,1,0,1,0,0,1,0]` llamado `vector_puente`, donde 1 representa que el tablón esta roto y 0 sano, y la primer posición del mismo el inicio del puente y la última el fin del mismo. Adicionalmente sabemos que la cantidad máxima de tablones que el participante puede saltar son 3 tablones.

Nuestro algoritmo va a encontrar la solución de la siguiente forma:

Primero creamos un vector `[0,0,0,0,0,0,0,0,0,0,0,0,0,0]` llamado `vector_de_distancias` que indica la cantidad mínima de saltos necesarios para llegar a cada tablón, manteniendo la correspondencia entre posiciones del arreglo y tablones del puente. La forma de ir actualizando el `vector_de_distancias` es la siguiente:

- Las posiciones que estén entre 0 y 2, la cantidad de saltos mínima es 1, con lo cual completamos el arreglo distancias con 1 en esas posiciones, quedándonos de la siguiente manera `[1,1,1,0,0,0,0,0,0,0,0,0,0,0]`
- Luego posición por posición vamos a buscar la cantidad mínima de saltos, para los tablones como el 3 y 4 como están rotos, dato conocido por el `vector_puente`, colocamos la distancia como infinito de la siguiente manera `[1,1,1,inf,inf,0,0,0,0,0,0,0,0,0]`.
- Siguiendo así con el quinto tablón ,agarramos los últimos 3 desde ahí, por ser 3 el salto máximo del participante, y buscamos el mínimo entre `[1[1,inf,inf]`, siendo este 1, le sumamos uno y lo colocamos en la 5 posición como la cantidad de saltos mñima hasta ese tablón, quedándonos `[1,1,1,inf,inf,2,0,0,0,0,0,0,0,0]`.
- Continuamos con las siguientes posiciones de la misma forma, al recorrer todo el vector, el Distancias será el siguiente `[1,1,1,inf,inf,2,3,inf,3,inf,4,4,inf,5]`.

Luego para armar la solución se recorre para atrás desde los últimos 3 elementos($[4,+\infty,5]$) elegimos el mínimo que en este caso es el 4 y agregamos a la solución la posición de dicho numero (en este caso el 12) , luego desde ahí repetimos la operación de seleccionar el mínimo entre los tres tablones anteriores a ese ($[3,+\infty,4]$) quedanonos el 3 y siendo su posición la novena, agregamos esta a la solución parcial, repetimos esta operación hasta recorrer todo el arreglo, nos queda el siguiente resultado [3,6,9,12] siendo estos los tablones a los cuales debe saltar para realizar la mínima cantidad de saltos y la cantidad mínima de saltos la longitud, en este caso 4.

1.0.3. Desarrollo

Para la solución de este problema recurrimos a la tecnica denominada programación dinamica. Por el enunciado sabemos que tenemos n tablones, el participante puede saltar C tablones de una sola vez y cuales son las posiciones de los tablones rotos.Nos armamos dos arreglos donde cada posición representa un tablón, en el primero guardamos 0 y 1 para indicar su estado (sano o roto respectivamente) llenando el mismo según la entrada, llamemoslo puente, en el otro iremos guardando la cantidad mínima de saltos para llegar a cada tablón,llamemoslo Distancias. Nuestro algoritmo ira recorriendo un tablón por vez, viendo si el mismo esta roto o no. En el caso de que el tablón está roto (según indica nuestro arreglo puente) continuo al siguiente tablón. Para los primeros C tablones que estan sanos pondremos 1 en el arreglo Distancias, cuando el algoritmo se encuentre en el tablón i , con C $\leq$ i $\leq$ n-1, calculamos su cantidad mínima de saltos de la siguiente manera:
- Buscamos el minimo entre i - C y i-1, en el arreglo de Distancias.
- Al mínimo encontrado le sumamos uno y lo colocamos en la posición i del arreglo Distancias.

Una vez completado el arreglo distancias debemos armar el recorrido, para el cual en primer paso buscamos dentro de las C ultimas posiciones del arreglo Distancias el mínimo, llamemos j a su posición. Luego agregamos este j a la solución como el ultimo tablón. A partir de ahí en cada paso buscamos el mínimo entre j-C y j-1, y lo agregamos adelante de nuestra solución y reemplazamos el j con la posición del nuevo mínimo. Una vez recorrido todo el arreglo tendremos nuestra solución optima.

1.0.4. Demostración

Demostraremos por inducción que en cada paso de este algoritmo obtenemos la mínima cantidad de saltos posible para llegar a ese tablón:

Sea P(i) = “ El valor guardado en la posición i-1 del arreglo Distancias es la cantidad mínima de saltos hasta el tablón i”

Caso base, P(i) con 0 $\leq$ i $<$ C : El caso base son los primeros C tablones, donde C es la cantidad máxima de tablones que un participante puede saltar de una sola vez. En este caso la cantidad mínima de saltos para cada tablón es trivialmente 1, ya que es el primer salto desde el punto de partida.

Paso inductivo, P(n) $\Rightarrow$ P(n+1) con n $\geq$ C : Por hipótesis inductiva sabemos que tenemos la cantidad de saltos mínima hasta el tablón n en el arreglo Distancias (entre las posiciones 0 y n-1). Si el tablón esta roto entonces la cantidad de saltos mínima será infinito. En el caso contrario, para calcular la mínima cantidad de saltos para llegar al tablón n+1 revisamos los últimos C tablones previos, y nos quedamos con el que requiera la menor cantidad de saltos para llegar hasta el, y llamaremos a esta cantidad Min. Entonces la cantidad mínima de saltos para llegar al tablón n+1 seria Min+1, y ahora veremos que efectivamente lo es. Si Min+1 no fuera la cantidad mínima de saltos para llegar al tablón n+1 entonces existe un tablón entre n-C y n-1 cuya cantidad mínima de saltos para llegar hasta el es menor a Min, lo cual es absurdo porque seleccione al mínimo. Puede pasar que exista un tablón entre n-C y n-1, distinto al seleccionado, cuya cantidad mínima de saltos para llegar hasta el coincida con Min. En ese caso esta solución es tan buena como la mía. Por lo tanto la cantidad de saltos mínima para llegar al tablón n+1 es efectivamente Min+1.

Para armar el recorrido, realizamos lo ya descripto en la sección de desarrollo y como siempre vamos tomando el mínimo en cada paso recorriendo el arreglo Distancias hacia atrás llegamos a una solución optima, que como vimos previamente, puede haber mas de una.

1.0.5. Complejidad

El siguiente es un pseudo-código de nuestro algoritmo.

Algorithm 1

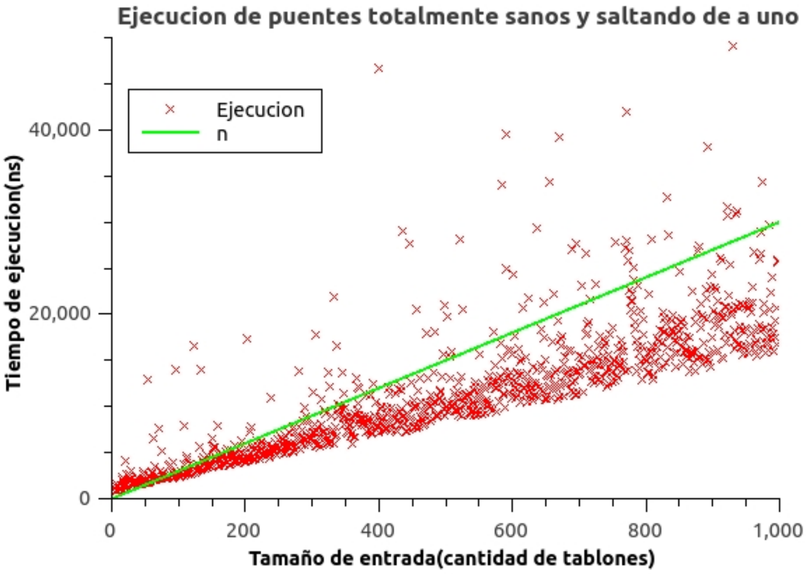
Saltos(saltoMax : natural, puente : arreglo(1's y 0's), distancias: arreglo(naturales) , cantidadTablones : natural)

1: Si saltoMax >cantidadTablones
2: devolver 1
3: SI NO
4: mientras posActual <cantidadTablones
5: SI puente.indice(posActual) es 0
6: posActual $\leftarrow posActual + 1$
7: SI NO
8: SI posActual - saltoMax <0
9: distancias.indice(posActual) $\leftarrow$ 1
10: posActual $\leftarrow posActual + 1$
11: SI NO
12: minimoSalto $\leftarrow BUSCOMINIMO(posActual - saltoMax, posActual, puente)\ \ O(saltoMax)$
13: distancias.indice(posActual) $\leftarrow minimoSalto + 1$
14: posActual $\leftarrow posActual + 1$

Todas las asignaciones y comparaciones son en O(1) como esta marcado en el pseudocodigo, ya que son números naturales y están acotados por la cantidad de tablones, en el caso del infinito este es representado por el máximo entero representable y este esta acotado.
Para el caso de la función BuscoMinimo, esta recorre en el arreglo puente buscando el mńimo entre las posiciones posActual-saltoMax y posActual-1, esto tiene una complejidad del orden O(saltoMax).
El ciclo de las lineas 5 - 15 se realiza n veces con lo cual la complejidad total del algoritmo es O(n*saltoMax), como saltoMax es una constante menor a n la complejidad total del algoritmo es O(n).

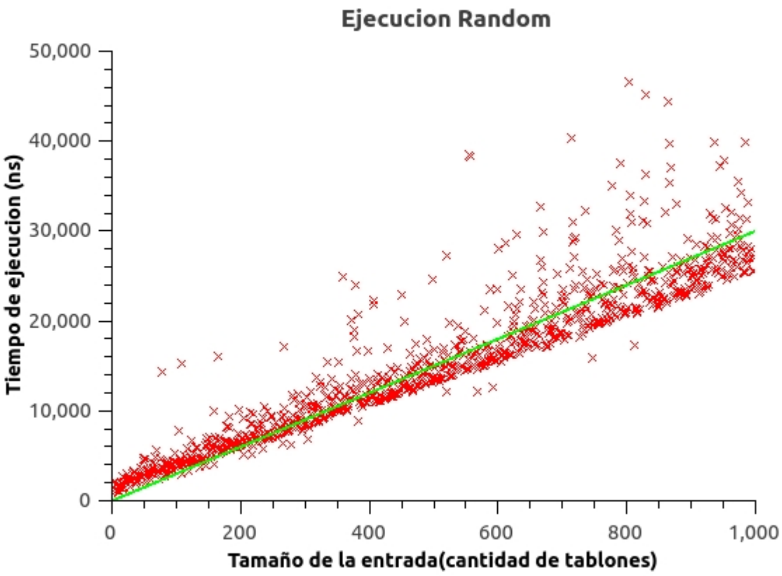
1.0.6. Experimentacion

Para la experimentación del problema en cuestión se realizaron dos test, el primero teniendo en cuenta el peor caso, siendo este que todos los tablones del puente estén sanos y el salto máximo del participante sea uno, para esto se fijo el salto máximo y se crearon 1000 instancias de puentes sanos en tamaños que varian de 1 a 1000 luego se midieron los tiempos de ejecución dando como resultado el siguiente gráfico



Como puede verse en el gráfico su complejidad esta en el orden lineal como puede compararse con la función, en el gráfico puede observar que algunos casos excede el orden, este ruido se debe a que puede haber ovaciones donde el sistema operativo este realizando otras tareas y afecte el orden de ejecución.

Para el segundo test se realizo un test aleatorio, para poder observar el comportamiento de todos los casos posibles, para esto se crearon 1000 instancias, en las cuales el tamaño del puente fue creciendo de 1 a 1000 y el salto máximo es un numero random entre 1 y la cantidad de tablones del puente para poder obtener también casos que no sean solución, dando comer resultado el siguiente gráfico.



Puede observarse que al igual en el test anterior el orden de complejidad esta en el orden lineal, de esta manera podemos concluir que nuestro algoritmo cumple con los ordenes de complejidad impuestos por el enunciado.

Capítulo 2

Horizontes lejanos

2.0.7. Introduccion

Se esta diseñando un software de arquitectura, para el cual es necesario que dado un conjunto de edificios representados como rectangulos apoyados sobre una base en comun, se devuelva el perfil definido en el horizonte.
Estos edificios vienen representados por tuplas de tres elementos que representan donde comienza el edificio, su altura y donde termina, de las cuales tenemos que ir tomando en cada momento donde comienza un edificio la altura maxima alcanzada en ese punto.

2.0.8. Ejemplos y Soluciones

Consideremos el siguiente ejemplo del problema:
$[< 3, 2, 5 >; < 1, 4, 2 >; < 4, 1, 6 >; < 6, 8, 10 >]$
Cada una de estas tuplas de tres elementos se indica donde comienza el edificio en la primera coordenada, su altura en la segunda y donde termina en la tercera coordenada.

Lo primero que hacemos es ordenar estas tuplas en orden creciente por lo que representa donde comienza el edificio (llamemosla pared izquierda), quedandonos de la siguiente manera:
$[< 1, 4, 2 >; < 3, 2, 5 >; < 4, 1, 6 >; < 6, 8, 10 >]$

Por otro lado ordenamos los edificios por la coordenada donde terminan (llamemosla pared derecha).
$[< 1, 4, 2 >; < 3, 2, 5 >; < 4, 1, 6 >; < 6, 8, 10 >]$

2.0.9. Desarrollo

Como mencionamos anteriormente, tenemos como datos de entrada la posición donde comienza y termina cada edificio y además su altura, con lo cual representaremos a los edificios con tuplas de 3 elementos (posición de inicio o pared izquierda, altura, y posición donde termina o pared derecha). Primero organizaremos a los edificios en dos arreglos, donde cada arreglo contendrá el total de edificios, uno con un orden ascendente según pared izquierda y el otro también con un orden ascendente pero según pared derecha. Además tendremos un conjunto en el que iremos agregando los edificios que “comience” y quitando los edificios que “terminen”, es decir, aquí estarán los edificios “activos” (que “empezaron” y no “terminaron”) en cada momento. La idea del algoritmo es ir recorriendo las posiciones (del eje x) en las que haya una o más paredes. En cada punto lo que haremos es agregar a mi conjunto de activos los edificios que en ese punto tengan su pared izquierda, o sea que están çomenzando", y quitar a los edificios que allí tengan su pared derecha, o sea que están “terminando”. Una vez completada esta labor, buscaremos el edificio activo que tenga la altura máxima y nos lo guardaremos, llamémoslo Max. Se puede ver claramente que el borde superior de la silueta en un punto dado va a estar determinado por el edificio más alto que haya en ese punto, es decir, el edificio activo más alto. Como en cada paso podemos conseguir el edificio activo más alto, proseguiremos así hasta el último punto y habremos armándonos la silueta.

2.0.10. Demostración

Demostraremos por inducción que en cada paso de este algoritmo obtenemos la altura del edificio más alto en ese punto.

P(i) = "Nuestro conjunto de edificios activos contiene todos los edificios que empezaron entre el punto de inicio y el punto i inclusive y que aún no han terminado, es decir, terminan en un punto estrictamente mayor que i"

Caso base, P(1): En este caso nos encontramos con nuestro primer conjunto de paredes, que puede tener una o más paredes. Este sólo puede tener paredes izquierda ya que es donde comienzan nuestros primeros edificios y ningún edificio ha empezado antes como para que aparezca una pared derecha indicando su finalización. Por lo tanto sólo tenemos un conjunto de edificios que comienzan en este punto, los cuales agregaremos a nuestro conjunto de edificios activos. Es claro ver que el conjunto de edificios activos ahora tiene todos los edificios que comenzaron y no han terminado hasta el primer punto inclusive ya que ningún edificio termina aquí como previamente hemos dicho y entonces todos estos empezaron y no terminaron.

Paso inductivo, P(n) → P(n+1): Nuestra hipótesis inductiva nos dice que nuestro conjunto de edificios activos contiene todos los edificios que empezaron entre el punto de inicio y el punto n inclusive, y que aún no han terminado. Ahora veamos qué ocurre en n+1. En este punto podemos encontrar un conjunto que contiene tanto paredes izquierda como derecha, con lo cual separaremos a este conjunto de paredes en dos subconjuntos. Por un lado el conjunto de paredes izquierda, llamémosle I, y por otro el de paredes derecha, llamémosle D. Recorreremos primero el conjunto I agregando cada edificio de este conjunto al conjunto de edificios activos, es decir, agregaremos todos los edificios que comienzan en este punto. Acto seguido, recorreremos el conjunto D quitando cada edificio que aparezca en este conjunto de nuestro conjunto de edificios activos ya que todos estos edificios están terminando y por lo tanto ya no deben pertenecer a nuestro conjunto. Veamos ahora que efectivamente se está cumpliendo P(n+1). Los edificios agregados en este paso no pueden terminar aquí mismo, ya que eso implicaría que están empezando y terminando en la misma posición, lo cual no es válido. Por lo tanto estarían terminando en un punto estrictamente mayor a n+1. Los edificios que terminaban en el paso n+1 fueron ya removidos del conjunto. Ahora miremos qué pasa con los demás edificios que ya estaban en nuestro conjunto de edificios activos y no fueron eliminados. Por hipótesis inductiva, estos edificios no tienen su pared derecha entre el punto de inicio y el punto n, es decir, terminan en un punto estrictamente mayor que n. Pero como no fueron borrados de nuestro conjunto, quiere decir que no tienen su pared derecha en n+1, o sea que terminan en un punto estrictamente mayor a n+1. Entonces vale P(n+1).

Teniendo nuestro conjunto de edificios activos, en cada punto podemos buscar el edificio de mayor altura y registrar los cambios de altura cada vez que empieza y/o termina un edificio. Luego, es trivial armarnos la silueta ya que, con las fluctuaciones de las alturas máximas a lo largo de nuestra ciudad, ya tenemos su borde superior en cada tramo.

2.0.11. Complejidad

definimos edificio = <izq :natural x alto : natural x der : natural >definimos edificioenCero al que tiene todos sus elementos en cero.

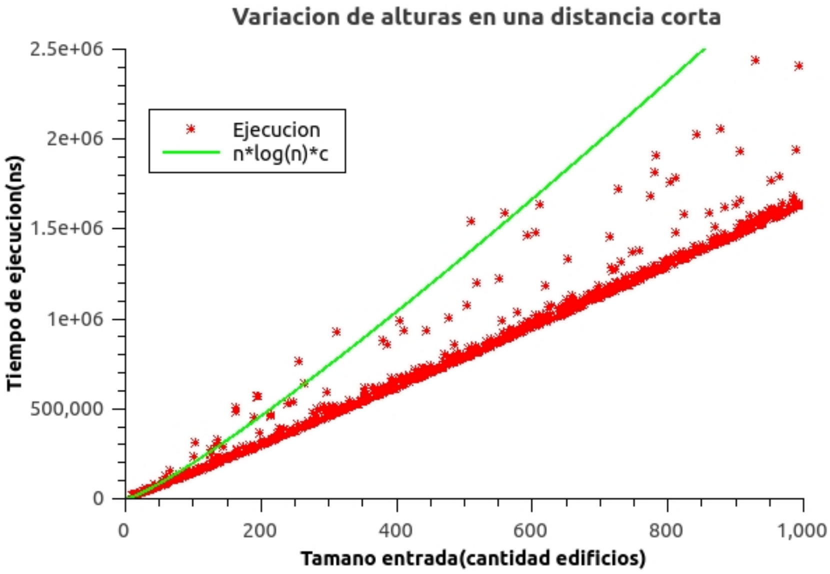
Algorithm 2 LaSilueta(ciudad: arreglo(edificios) , cantidadEdificios : natural)

1: arregloXIzq ← $ordenarXIzquierda(ciudad)$
2: arregloXDer ← $ordenarXDerecha(ciudad)$
3: EdificiosActivos ← $Multiconjunto(edificio)$
4: posIzquierdo , posDerecho ← 0
5: max ← $edificioenCero$
6: mientras posIzquierdo <cantidadEdificios && posDerecho <cantidadEdificios
7:     SI arregloXIzq.indice(posIquierdo).izq ≤ $arregloXDer.indice(posDerecho).der$
8:         auxiliar ← $arregloXIzq.indice(posIzquierdo)$
9:         mientras auxiliar.izq = arregloXIzq.indice(posIquierdo).izq && posIzquierdo != cantidadEdificios
10:             agregar(arregloXIzq.indice(posIquierdo), EdificiosActivos)
11:             SI arregloXIzq.indice(posIquierdo).alto >max.alto
12:                 max ← $arregloXIzq.indice(posIquierdo)$
13:     SI NO
14:         auxiliar ← $arregloXDer.indice(posDerecha)$
15:         mientras auxiliar.der = arregloXDer.indice(posDerecha).der posDerecha != cantidadEdificios
16:             SI arregloXDer.indice(posDerecha).id = max.id
17:                 dameMaximoSiguiente(arregloXDer.indice(posDerecha), EdificiosActivos)
18:                 sacar(arregloXDer.indice(posDerecha), EdificiosActivos)

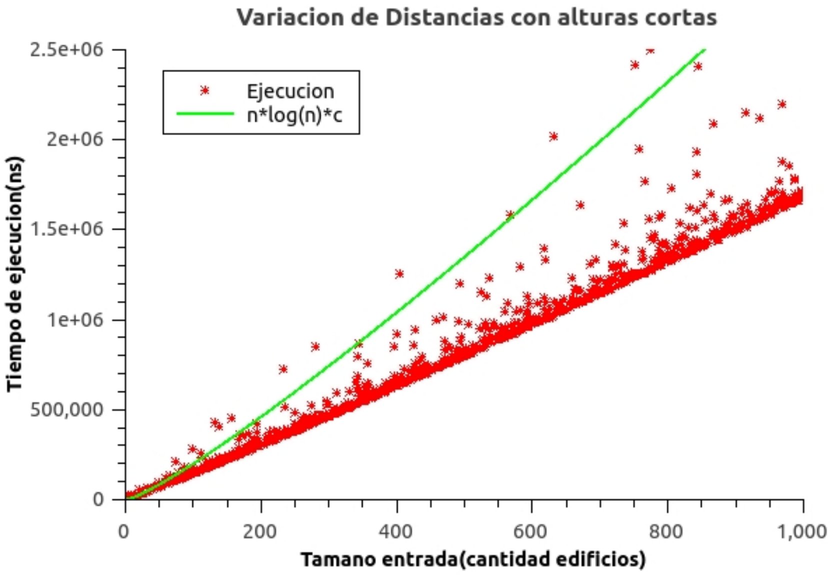
Los algoritmos ordenarXizquierda y ordenarXDerecha es el conocido algoritmo mergeSort sacado del libro de brassard, la complejidad del mismo es O(N*Log(N)) siendo N la cantidad de elementos en el arreglo, ósea todos los edificios, estos algoritmos ordenan los arreglos de tuplas, uno por la coordenada izq y el otro por la coordenada derecha respectivamente.
El multiconjunto EdificiosActivos esta representado con la estructura Multiset de la librería STL de c++, la relación de orden sobre los elementos que se definió es por la coordenada alt y en el caso que la altura sea igual por la componente der.
Para las operaciones para agregar, sacar elementos la complejidad es O(log(n)) siendo n la cantidad de elementos en el multiconjunto y para obtener el proximo maximo lo que se realiza es obtener la referencia al elemento que representa el máximo, esto lleva O(Log(n)) y luego se obtiene el posterior, en caso de no ser posible obtenemos el anterior, en el peor caso se tienen todos los edificios y la complejidad seria O(log(N)).
El ciclo de las lineas 9-12 se realiza para cada punto x donde haya paredes izquierdas, con lo cual la complejidad total sera la suma de todas las paredes izquierda, siendo esta N la cantidad de edificios. De la misma manera el ciclo de las lineas 15 a 18 se ejecuta para todas las paredes derechas de los edificios con lo cual el total de las iteraciones sera la suma de las paredes derechas ósea N. Como el ciclo principal se ejecuta mientras la cantidad de paredes recorridas izquierdas sea menor a la cantidad de edificios y la cantidad de paredes derechas sea menor a la cantidad de edificios y las iteraciones internas modifican las paredes izquierdas y derechas recorridas, el total de las iteraciones es 2*N, siendo N la cantidad total de edificios, con lo cual como la complejidad de agregar , sacar y obtener el máximo es O(log(N)) en el peor caso, el total es O(2*N*Log(N)) ⊂ O(N*Log(N)) siendo esta la complejidad total del algoritmo y esta estrictamente menor a O(N^2)$comosesolicitabaenelenunciado$.

2.0.12. Experimentación

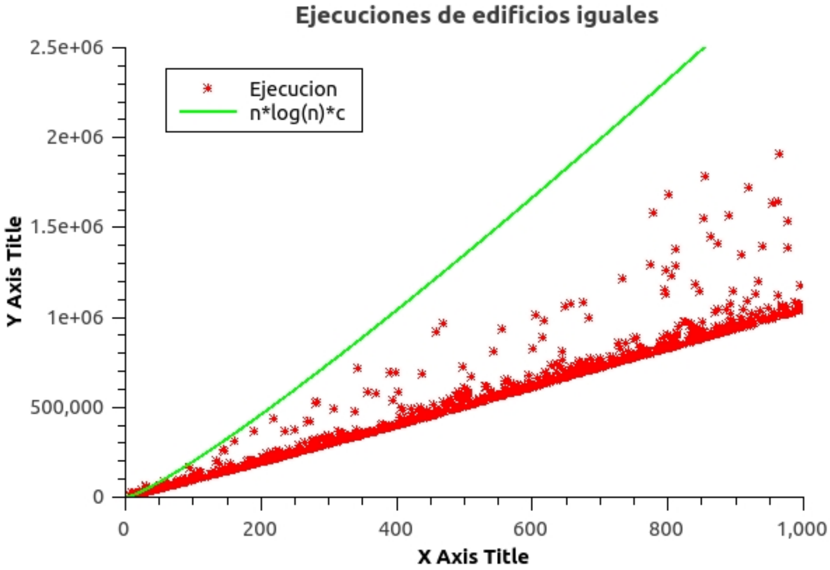
Se realizaron experimentaciones sobre varios tipos de caso para observar el comportamiento del algoritmo, en todos los casos se generaron 1000 entradas, a las gráficas resultantes se las comparo con la función n*log(n)*1000, siendo n la cantidad de edificios de entrada.
Para la primer ejecución se realizo un análisis sobre el algoritmo en edificios donde la distancia entre sus paredes izquierda y derecha era corta, estas se encontraban entre las posiciones 1 y 30, pero sus alturas variaban en el orden entre 1 y 10000, dando el siguiente gráfico como resultado.



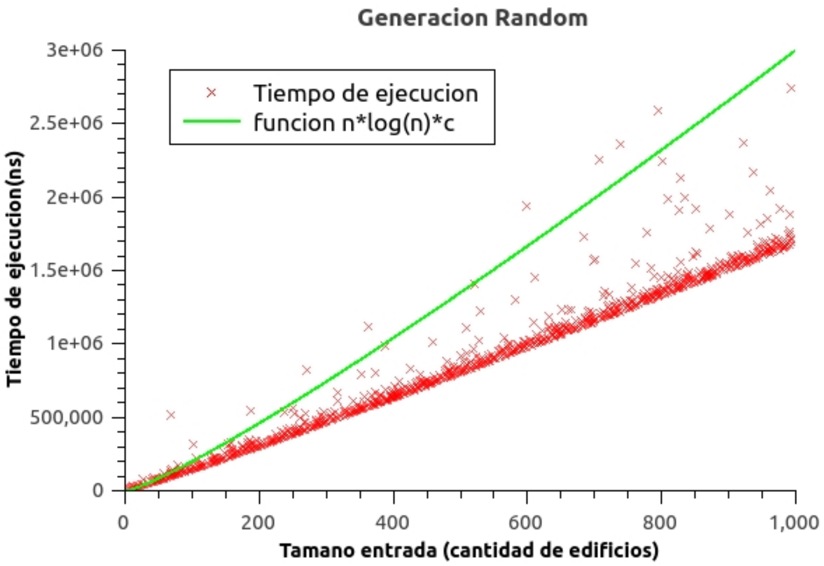
El segundo gráfico corresponde a instancias generadas de manera random donde la altura se encontraba entre 1 y 40, pero las distancias entre la primer y la segunda pared de cada edificio varia entre los 1 y 2000 , elegido de manera random la posición de ambos lados pero siempre que sea entrada valida, ósea el valor de la pared izquierda es mayor estricta a la de la pared derecha.



Luego para el tercer gráfico consideramos que era importante ver como se comportaba el multiconjunto cuando se agregaban todos los edificios de la entrada, de esta manera se generaron entradas donde los edificios eran todos iguales, así se ingresaban todos al multiconjunto y veíamos como se comportaba el algoritmo, obteniendo este gráfico como resultado donde se ve que su complejidad es del orden logarítmico en la cantidad de edificios de entrada por la cantidad edificios.



Por ultimo realizamos un gráfico con entradas random donde las mismas variaban combinando todas las anteriores, obteniendo de nuevo que el orden era logarítmico al compararlo con la función.



Concluimos por lo tanto que el algoritmo respeta los ordenes de complejidad requeridos.

Capítulo 3

Biohazard

3.1. Introducción

El problema para este ejercicio es el siguiente, se nos presentan n productos químicos, los cuales deben transportarse en camiones de un lugar a otro, el llevar al elemento i en el mismo camión que otro elemento j, conlleva una "peligrosidad" asociada h_{ij}. El objetivo del algoritmo será encontrar la solución que utilize la menor cantidad de camiónes posibles, pero que cada camión tenga una peligrosidad menor a una cota m. La entrada del problema consiste en:

- Un entero **n** $\rightarrow$ Representarán el número de productos químicos a transportar.
- Un entero **m** $\rightarrow$ Representará la cota de peligrosidad que ningun camión puede superar.
- **n-1** filas donde, para cada fila i consta de $n - i$ enteros:
  - $h_{i,i+1}, h_{i,i+2}$ ... $h_{i,n}$ $\rightarrow$ Representarán la peligrosidad asociada del elemento i con los elementos $i + 1$, $i + 2$ ... n.

La salida, por su parte, constará de una fila con:

- Un entero **C** $\rightarrow$ Representará la cantidad indispensable de camiónes que es necesaria para transportar los productos bajo las condiciones del problema.
- n enteros $\rightarrow$ Representarán en que camión viaja cada producto.

3.1.1. Ejemplo de entrada valida

Hagamos un pequeño ejemplo para que pueda ilustrarse bien el problema.
Supongamos que tenemos 3 productos químicos, el producto 1 es muy inestable, por lo que si es transportado con el producto 2 la peligrosidad asociada al camion en el que viajan estos dos productos asciende a 40, y si se transporta con el producto 3 la peligrosidad será de 35. El producto 2 en cambio es de naturaleza mas estable, por lo que si es transportado con el producto 3 solo produce una peligrosidad de 3.
Por otro lado queremos que la peligrosidad por camión no supere el valor de 39.
Entonces la entrada para este problema será:

3 39
40 35
3

Para una entrada de estas dimenciones es posible buscar la mejor solucioón a mano.
Las posibles combinaciones son que los tres productos viajen juntos, que los tres viajen separados en camiones distintos, que 1 y 2 viajen juntos en el mismo camion y el producto 3 viaje en otro camión diferente, que 1 y 3 viajen juntos y el 2 separado y que 2 y 3 viajen juntos y el producto sobrante viaje en

otro camión.
La primera dá una peligrosidad de $40+35+3$ por lo la cota de peligrosidad se ve superada, lo que lo vuelve una solución inviable, la segunda es valida, ya que la peligrosidad de cada camion es 0, pero se necesitan 3 camiones. Que 1 y 2 viajen juntos, tampoco es valida, la peligrosidad de ese camion es demaciado alta, y finalmente las ultimas dos son validas (peligrosidad 35 y 3, respectivamente) y solo son necesarios dos camiones.
Es claro, luego, que las dos ultimas soluciones son las que el algoritmo podría devolever.
Luego las dos salidas que podrá devolver el algoritmo son:

- 2 1 2 1

o

- 2 1 2 2

3.2. Idea General de Resolución

Luego la idea del algoritmo es simple, probar todas las combinaciones posibles de camiones y de entre todas determinar cual es la que cumple con la cota de peligrosidad pedida y usa la menor cantidad de camiones posible. Ademas, para aumentar la performance del algoritmo, se irán podando ramas de la familia de soluciones de manera tal de que no sea necesario chequear absolutamente todos los casos.
Antes de presentar el pseudocodigo vale aclarar un punto importante y es que el algoritmo debe encontar siempre una solución. Esto se debe a que siempre es posible poner todos los productos químicos en camiones separados, lo que nos dá una peligrosidad 0. Es posible usar esta como una cota contra la cual parar de chequear, si tenemos n productos químicos, es simple ver que a lo sumo usarémos n camiones. Denominaremos a esta como la "peor solución"a que es claro que es una solución valida, pero que usa la maxima cantidad de camiones.
En cuanto a las podas, utilizamos dos, una que en cada paso del backtrack chequea si la solución final que encontramos hasta el momento usa una cantidad menor de camiones que la solucion parcial que se esta construyendo. Es claro que de ser así, estamos la solucion parcial nunca podrá ser mejor, por lo tanto se podará toda esa familia de soluciones.
La segunda chequea que la solución parcial que estamos construyendo no exceda el limite de peligrosidad pedido por el ejercicio, en caso de ser así la solución no será valida. En caso de que esto suceda, tambien se poda.
Finalmente el pseudocodigo para resolver este problema queda así:

Algorithm 3 void FuncionPrincipal()

```
Generar una matriz con las peligrosidades entre los distintos productos
Se inicializa la solucion final, como la peor de las soluciones
Backtrack(tablaDePeligrosidad, solucionParcial, solucionFinal)
Mostar la solución final
```

Algorithm 4 Bool Backtrack(tablaDePeligrosidad, solucionParcial, solucioninal)

Llamo a la funcion check(tablaDePeligrosidad, solucionParcial, solucionFinal)
  Si check devuelve 2, la solucion parcial es mejor que la final
    Pongo la solucion parcial como final
    Corto la recursión y busco por otra rama
  Si check devuelve 0, la cota de peligrosidad fué sobrepasada
    esta rama no me sirve, podo
  Si check devuelve 3, la solucion anterior usa menos camiones
    esta rama no me sirve, podo
  Si check devuelve 1, la solucion es valida, pero no está completa
    continúo agregando camiones
Para cada valor i de 1 hasta n prueba meter el siguiente producto de la lista en el camion i y se llama a la funcion Backtrack()

Algorithm 5 int check(tablaDePeligrosidad, solucionParcial, solucionFinal)

Checkeo si la solucion final usa menos camiones $O(n)$
  Si es verdad
    Devuelvo 3
Checkeo si la cota de peligrosidad fue sobrepasada $O(n^2)$
  Si es verdad
    Devuelvo 0
Checkeo si la cada producto tiene un camion asignado $O(1)$
  Si es verdad
    Devuelvo 2
Devuelvo 1

3.3. Correctitud

Según lo desarrollado en la idea principal, sabemos que el algoritmo siempre tendá una solución y esta se encuentra acotada en un vector de n elementos cada uno entre 1 y n. Dado que el rango es acotado, un algoritmo que chequee todas las posibles soluciones y devuelva la solución valida que usa menos camiones, será un algoritmo correcto.
Ahora lo que resta demostrar es que las podas que realizamos, no quitan soluciones válidas.
La primera de las podas chequea que si la solución parcial excede la cota m. Es claro ver que cualquier solución que tenga una sub-solución inválida nunca podrá ser válida, por lo tanto se puede descartar sin necesidad de completarla.
La segunda de las podas chequea que la solución que estamos construyendo tenga menos camiones que la mejor solución que tenemos hasta el momento. Esto es, si sé que puedo llevar los n productos químicos en j camiones, no es necesario explorar las soluciones que contengan más de j camiones, estas soluciones jamás podrán ser mejores que la solución que ya tengo.
Luego quitando esa parte del espacio de soluciónes no estoy quitando en ningun momento posibles soluciones óptimas, ergo, el algoritmo es correcto.

3.4. Complejidad

Por cada producto químico, el algoritmo de backtrack intenta ponerlo en cualquiera de los n posibles camiones, y realiza $O(n^2)$ chequeos intentando podar.

Entonces la formula quedará:

$$T(i) = T(i-1)n + n^2$$

$$T(1) = n + n^2$$

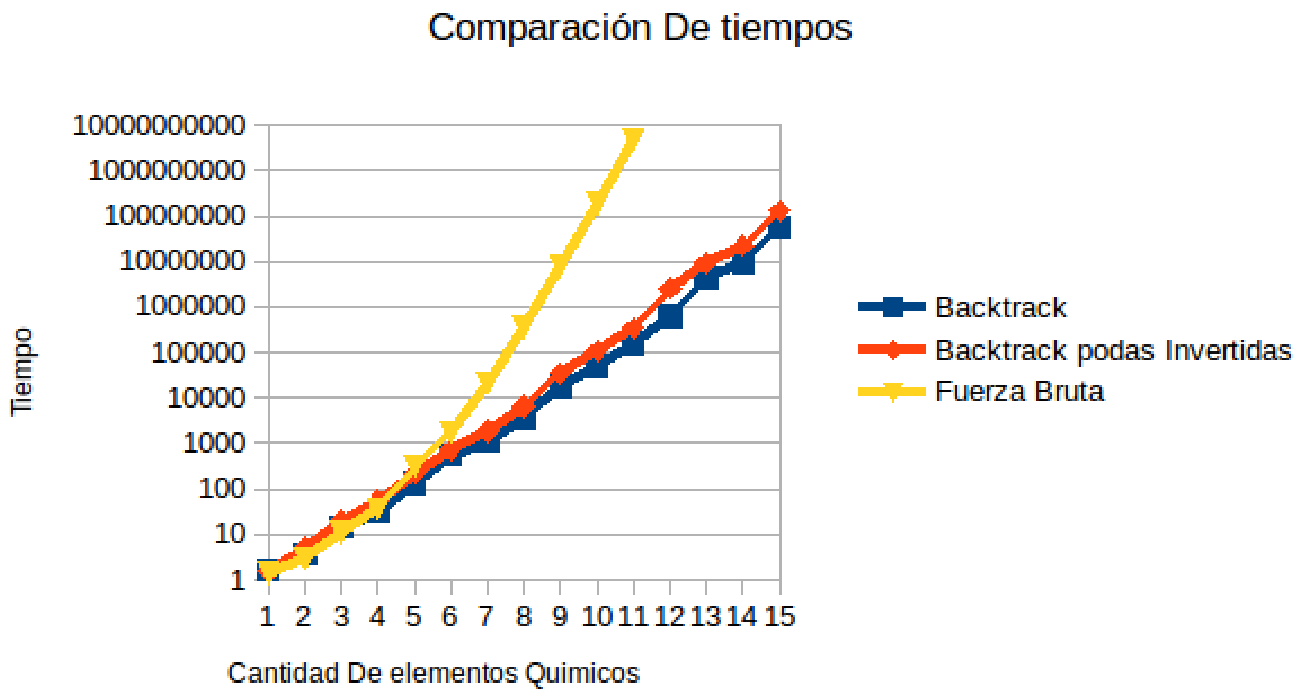
Luego para el peor de los casos el algoritmo tendrá una complejidad de $O(n^n)$.

3.5. Resultados

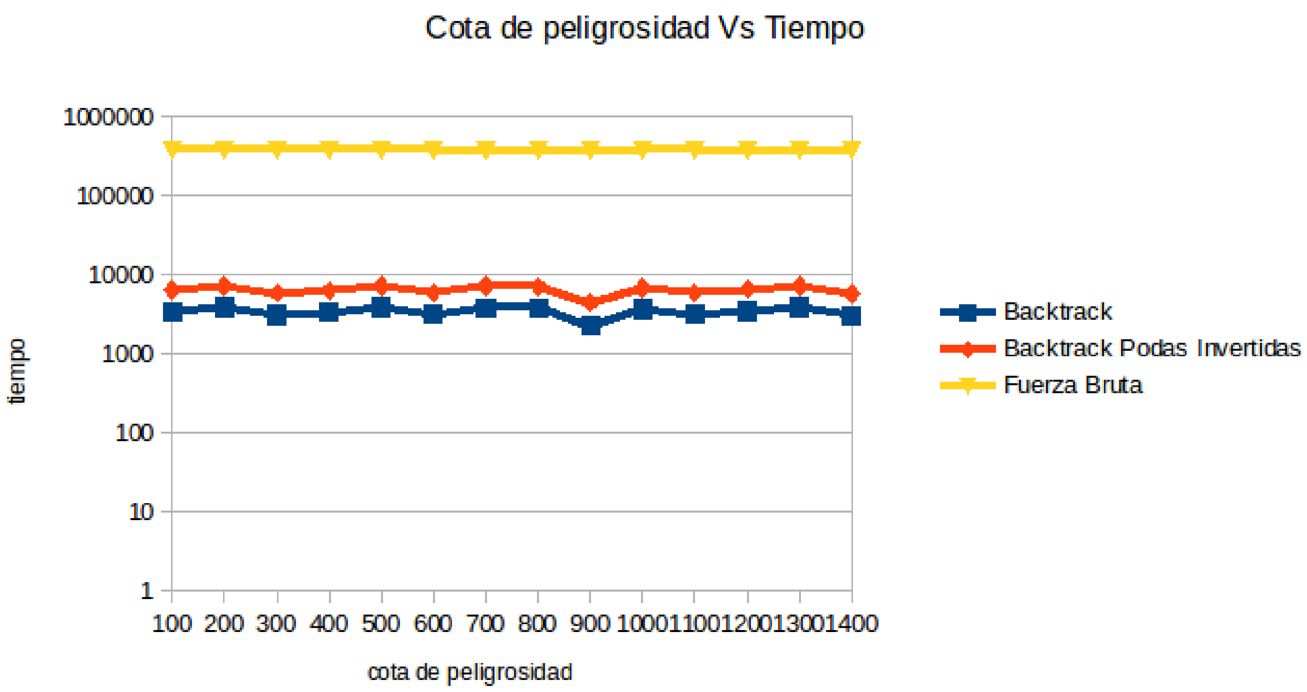
3.5.1. Testing

3.5.2. Caso Random

Para testear la performance de nuestro algoritmo se creó un generador de entradas que fabricará instancias random del problema. Luego se comparará el tiempo que tarda nuestro algoritmo contra uno que encuentre la solución utilizando fuerza bruta y también contra otro backtracking, pero con las podas invertidas, o sea primero chequea si la cantidad de camiones de la solución parcial es menor a la cantidad de camiones de la mejor solución hasta el momento y luego chequea que la cota de peligrosidad sea la correcta, para ver si esto varía de alguna manera la complejidad.
El testeo consistió en generar 40 instancias del problema para cada n diferente, con un m fijo y valores de peligrosidad entre productos químicos que varían entre 1 y m. Luego se tomó la media y compararon los resultados:



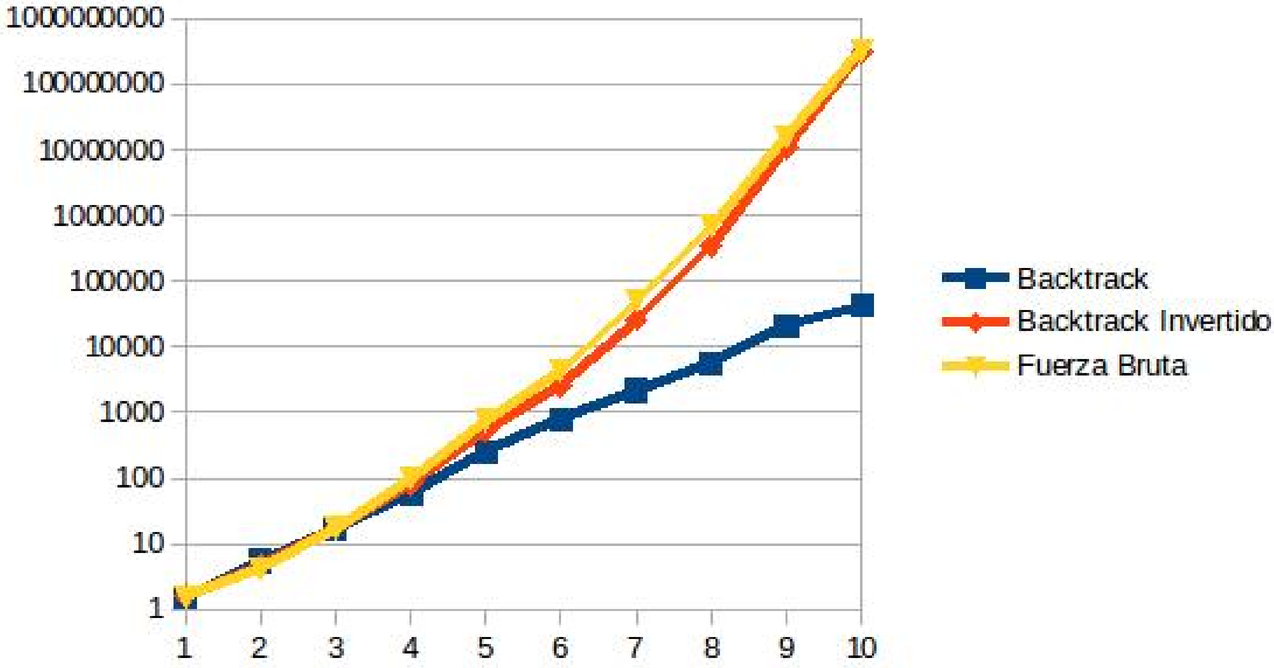
En el gráfico puede verse que nuestro algoritmo es notablemente superior a un algoritmo de fuerza bruta, ya que escala mucho mejor con respecto a n y levemente mejor al backtracking con las podas invertidas. Para los casos de 14 y 15 elementos el backtracking pudo arrojar una respuesta en un tiempo admisible, mientras que el de fuerza bruta ya tardaba tiempos completamente fuera de escala.
Para comprobar de manera experimental que la complejidad del algoritmo solo depende de n también se realizó lo mismo variando la cota de peligrosidad m, para un n fijo igual a 9.
Los resultados arrojados pueden verse en el siguiente gráfico:



Luego es claro que ninguno de los algoritmos depende de m. Además en este gráfico puede volverse a apreciar de manera visible la mejora de nuestro algoritmo con respecto a uno de fuerza bruta.

3.5.3. Peor caso

Otro test que podemos intentar realizar es ver si nuestro algoritmo presenta alguna mejora al de fuerza bruta en el peor de los casos, esto es, en el caso de que cada producto tenga que viajar forzosamente en un camión diferente, obligando de cierta manera a nuestro algoritmo a chequear todos los reslutados posibles, sin poder realizar podas significativas.
Para testear esto tomamos nuevamente 40 muestras aleatorias, con un m fijo en 10, valores de peligrosidad los productos entre m y $m+2$ las corremos en los tres algoritmos.
Los resultados pueden verse en el siguiente cuadro:



Puede verse aquí que nuestra solución continúa siendo mejor que uno de fuerza bruta. En este caso, el otro backtracking con las podas invertidas muestra una notable perdida de performance, siendo casi tan malo como el algoritmo de fuerza bruta.

Capítulo 4

Apéndice

4.1. Medicion de los tiempos

Para este tp como trabajamos bajo el lenguaje de programacion C++, decidimos calcular los tiempos utilizando 'chrono' de la libreria standard de c++ (chrono.h) que nos permite calcular el tiempo al principio del algoritmo y al final, y devolver la resta en la unidad de tiempo que deseamos.

4.2. Código Fuente

4.2.1. Ej1.cpp

```
#include "Ej1.h"

using namespace std;

void imprimir_vector(list<int> vec)
{
  for ( std::list<int>::iterator it=vec.begin(); it != vec.end(); ++it)
  {
    if((*it) == INT_MAX) {
      cout << "inf ";
    } else if((*it) < 0) {
      continue;
    } else {
      cout << (*it) << " ";
    }
  }
  cout << "\n";
}

bool check_solution(vector<int> puente, int largo_del_salto){
  int contador_de_escalones_rotos_seguidos = 0;
  for (int i = 0; i < puente.size(); ++i)
  {
    if(puente[i] == 1) {
      contador_de_escalones_rotos_seguidos++;
    } else{
      contador_de_escalones_rotos_seguidos = 0;
    }
    if(contador_de_escalones_rotos_seguidos == largo_del_salto) {
      return false;
    }
  }
  return true;
}

list<int> obtener_recorrido(vector<int> distancias, int largo_del_salto,int n)
{
  list<int> recorrido;
  recorrido.push_back(n + 1);
  for (int i = (n); i > 0;)
  {


    int inicio_del_recorrido = 0;
    int pos_salto = 0;
    if ( i - largo_del_salto < 0) {
      pos_salto = min_element( distancias.begin(), distancias.begin() + i) -
         distancias.begin();

    } else {

      pos_salto = min_element( distancias.begin() + i - largo_del_salto,
         distancias.begin() + i) - distancias.begin();

    }

```

```
      //cout << "valor: " << distancias[i] << " i: " << i <<  " minimo en: " <<
         pos_salto << "\n";
      i = pos_salto;

      if( i > 0 ) {
        recorrido.push_front(pos_salto + 1);
      }
    }
    return recorrido;
}

int main()
{
    while(true){
      int n, largo_del_salto;
      cin >> n;
      if(n == 0) {
        return 0;
      }

      cin >> largo_del_salto;

      vector<int> puente;
      vector<int> distancias;

      //Creacion del primer elemento artificial con cantidad de saltos 0
      for (int i = 0; i < n; ++i)
      {
        int tablon;
        cin >> tablon;
        puente.push_back(tablon);
        distancias.push_back(INT_MAX) ; //inicializo todo en inf, que para mi es
           INT_MAX
      }

      //miro si tiene solucion
      if(!check_solution(puente,largo_del_salto)){
        cout << "no\n";
        continue;
      }


      if(largo_del_salto > n){
        cout << "1 " << n+1 << endl;
        continue;
      }
        auto begin = std::chrono::high_resolution_clock::now();

      //El algoritmo!
      for (int pos_puente = 0; pos_puente < n; ++pos_puente)
      {
        //Si el escalon esta roto, lo salteo
        if(puente[pos_puente] != 0) {
          continue;
        }

        //Busco el rango de tablones que puedo pisar
        int pos_puente_inf;
        int dist_min;
        if( pos_puente - largo_del_salto <= 0){
          pos_puente_inf = 0;
```

```
        dist_min = 1;
      } else {
        /* INICIO MODIFICACION RICARDO*/
        /******* Codigo CHUDI
        // dist_min = *(min_element(distancias.begin() + pos_puente_inf , distancias.begin() + pos_puente));
        //dist_min = dist_min + 1;
          ***/

        bool encontre = false;
        pos_puente_inf = pos_puente - largo_del_salto;
        while(pos_puente_inf<=pos_puente && !encontre){
          if(!puente[pos_puente_inf]){
            encontre = true;
          }
          pos_puente_inf++;
        }
        if (pos_puente_inf == pos_puente)// si recorri todo el rango y estan todos rotos no hay solucion
        {
          break;
        }
        dist_min = distancias[pos_puente_inf];
        dist_min = dist_min + 1;
        /* FIN MODIFICACION RICARDO*/


      }
      distancias[pos_puente] = dist_min;
    }
    list<int> recorrido = obtener_recorrido(distancias, largo_del_salto,n);
    recorrido.push_back(recorrido.size());
    auto end = std::chrono::high_resolution_clock::now();
   // std::cout << n << ' ' << std::chrono::duration_cast<std::chrono::nanoseconds>(end-begin).count();
   // cout << std::endl;

    //std::reverse(recorrido.begin(),recorrido.end());

    imprimir_vector(recorrido);
  }
  return 0;
}
```

4.2.2. Ej2.cpp

```
#include <iostream>
#include <sstream>
#include <stdlib.h>
#include <stdio.h>
#include <vector>
#include <sys/time.h>
#include <set>

#include    <cstdlib>
#include    <ctime>
#include    <sys/timeb.h>

using namespace std;
unsigned long long operaciones;

struct edificio{
        int id;
        int izq;
        int alt;
        int der;
};



struct sol{
    int x;
    int alt;
};

bool operator <(const edificio& a, const edificio& b){

  if(a.alt == b.alt) {
    return a.der < b.der;
  } else {
    return a.alt < b.alt;
  }
}
vector<sol> resolver(int cantEdificios, edificio* edificiosIzq, edificio*
   edificiosDer);


void merge(edificio*,edificio*,int,int,int,int);
void mergesort(edificio *a,edificio *b, int low, int high,int control)
{

        int pivot;
    if(low<high)
    {
                pivot=(low+high)/2;
        mergesort(a,b,low,pivot,control);
        mergesort(a,b,pivot+1,high,control);
        merge(a,b,low,pivot,high,control);
    }
}
void merge(edificio *a,edificio *b, int low, int pivot, int high,int control)
{
    int k;
    int h=low;
```

```
    int i=low;
    int j=pivot+1;
    int elemento_h;
    int elemento_j;
    while((h<=pivot)&&(j<=high))
    {
      if(control == 0){
        elemento_h = a[h].izq;
        elemento_j = a[j].izq;
      }else{
        elemento_h = a[h].der;
        elemento_j = a[j].der;
      }

        if( elemento_h < elemento_j  )
        {
            b[i]=a[h];
            h++;
        }else{
            b[i]=a[j];
            j++;
        }
        i++;
    }
    if(h>pivot)
    {
        for(k=j; k<=high; k++)
        {
            b[i]=a[k];
            i++;
        }
    }else{
        for(k=h; k<=pivot; k++)
        {
            b[i]=a[k];
            i++;
        }
    }
    for(k=low; k<=high; k++) {
                a[k]=b[k];
    }
}


vector< sol > resolver(int cantEdificios, edificio* edificiosIzq, edificio*
    edificiosDer){
  vector<sol> res;
  //ordernar por izq
  edificio* auxiliarParaOrdenar = new edificio[cantEdificios];

  mergesort(edificiosIzq,auxiliarParaOrdenar,0, cantEdificios-1,0);

  // //SOlo para test
  //    cout << "orden por izq \n";
  //    for(int i = 0; i<cantEdificios; i++){
  //      cout <<"( " << edificiosIzq[i].id  << "," ;
  //      cout << edificiosIzq[i].izq  << ",";
  //      cout << edificiosIzq[i].alt  << ",";
  //      cout << edificiosIzq[i].der  << ");";

  //    }
```

```
    //    cout << "\n";
    //FIN TEST


    //ordenar por derecha
    mergesort(edificiosDer,auxiliarParaOrdenar,0, cantEdificios-1,1);

    //Solo para test
       // cout << "orden por der \n";
       // for(int i = 0; i<cantEdificios; i++){
       //   cout <<"( " << edificiosDer[i].id  << ",";
       //   cout << edificiosDer[i].izq  << ",";
       //   cout <<  edificiosDer[i].alt  << ",";
       //   cout << edificiosDer[i].der  << ");";
       // }
       // cout << "\n";
    //FIN TEST

    multiset<edificio> magiheap;
    int posDer = 0;
    int posIzq =0;
    edificio max = {0,0,0,0};
    int posMax = 0;

    sol nueva = {0,0};
    bool finIzq = false;
    while(posDer != cantEdificios){


          if((finIzq) || (edificiosIzq[posIzq].izq > edificiosDer[posDer].der)){
            //Saco edificio o edificios
            int base = posDer;
            int maxAnt = max.alt;
            while ((edificiosDer[posDer].der == edificiosDer[base].der) && (posDer
                != cantEdificios)){
              multiset<edificio>::iterator it = magiheap.lower_bound(edificiosDer[
                  posDer]);

              if (it->id == max.id){
                if(magiheap.size() > 1){
                  multiset<edificio>::iterator itMax = magiheap.end();
                  itMax--;
                  if (it->id == itMax->id){
                    itMax--;
                  }
                  max.alt = itMax->alt;
                  max.izq = itMax->izq;
                  max.der = itMax->der;
                  max.id = itMax->id;
                }else{
                    max.id = 0;
                    max.izq = 0;
                    max.alt = 0;
                    max.der = 0;
                }
              }
              magiheap.erase(it);
              posDer++;
            }
            if (maxAnt > max.alt){
                nueva.x = edificiosDer[base].der;
```

```
              nueva.alt = max.alt;
              res.push_back(nueva);
          }
        }else{
          //Agrego edificio o edificios
          int base = posIzq;
          int maxAnt = max.alt;
          while ((edificiosIzq[posIzq].izq == edificiosIzq[base].izq) && (posIzq != cantEdificios)){
            edificio nuevoEdi = *(magiheap.insert(edificiosIzq[posIzq]));
            if (nuevoEdi.alt > max.alt){
              max = nuevoEdi;
            }
            posIzq++;
          }
          if (maxAnt < max.alt){
              nueva.x = max.izq;
              nueva.alt = max.alt;
              res.push_back(nueva);
          }
          if (posIzq==cantEdificios){
            finIzq = true;
          }
        }
  }


  return res;

}


int main(int argc, char *argv[]){

        string line;
        bool primerLinea = true; //para saber cuando tomamos la primera linea, el n.
        int cantEdificios = 0;
        edificio* edificiosIzq = NULL;
        edificio* edificiosDer = NULL;
        int leidas = 1;
         timeval tm1, tm2;
        //Si no llegue al final del archivo y sigo obteniendo lineas sigo guardando
        while ( getline (cin,line) )
        {
            if(line[0] == '0'){
              break;
            }

            if(primerLinea){
              cantEdificios = atoi(line.c_str()); //El n de la entrada
              edificiosIzq = new edificio[cantEdificios];
              edificiosDer = new edificio[cantEdificios];
              primerLinea = false;
              leidas = 1;
            }else{


              vector<string> entradaSplit;
              int fromIndex = 0;//inicio string
```

```
                int length = 0; //longitud string
                for(int i = 0; i<line.length();i++ ){
                      if(i== line.length()-1){
                        length++;
                        entradaSplit.push_back(line.substr(fromIndex, length));
                      }else if(line[i]== ' '){
                        entradaSplit.push_back(line.substr(fromIndex, length));
                        fromIndex = i+1;
                        length = 0;
                      }else{
                        length++;
                      }
                  }//Fin for

                  int izq = atoi(entradaSplit[0].c_str());
                  int alt = atoi(entradaSplit[1].c_str());
                  int der = atoi(entradaSplit[2].c_str());
                  edificio nueva = {leidas,izq,alt,der};
                  //cout << nueva.id << " " << nueva.izq << " ";
                  //cout << nueva.alt << " " << nueva.der << "\n";
                  edificiosIzq[leidas-1]=nueva;
                  edificiosDer[leidas-1]=nueva;

                  if(leidas == cantEdificios){
                    primerLinea = true;
                    //Aca llamamos a resolver

                    //para medir tiempos , mido antes de empezar
                    //gettimeofday(&tm1, NULL);
                  auto begin = std::chrono::high_resolution_clock::now();
                  vector<sol> resultado = resolver(cantEdificios,edificiosIzq,
                     edificiosDer);
                    // mido cuando termino
                      auto end = std::chrono::high_resolution_clock::now();

                   cout << cantEdificios << " " << std::chrono::duration_cast<std
                      ::chrono::nanoseconds>(end-begin).count();
                    //solo para imprimir tiempos descomentar la linea siguiente
                      cout << endl;


              //Una vez obtenida la solucion imprimimos el resultado
                     // for(int i =0;i< resultado.size();i++){
                     //      cout << resultado[i].x << " " << resultado[i].alt <<
                        " ";
                     //  }
                     //  cout << "\n";
                  }

                  leidas++;

                }
              }



        return 0;

}
```

4.2.3. Ej3.cpp

```
#include "Ej3.h"
#include <sys/time.h>
#include <sys/timeb.h>

// de la manera en que esta creado el algoritmo,
// busca la solucion de abajo para arriba
// esto signigica, al principio intenta meter todos los productos en el camion 1
// si no, mete uno en el camion 2, y asi...
// todavÃa tengo que justificar, que en el momento de parar, entrega la mejor
   solo
// pero a simple vista me parece que falta algo.

using namespace std;

// n-> cantidad de productos a transportar
// m-> nivel de peligrosidad
// n-1 lineas:
//      h1,2 h1,3 h1,4 ... h1, n
//      h2,3 h2,4 h1,5 ... h2, n
//      ...
//      h(n-1),n


void imprimir_tab(tablaDePeligrosidad tab)
{
 for (int i = 0; i < tab.peligrosidad.size(); i++)
 {
   vector<int> v = tab.peligrosidad[i];
   for (int j = 0; j < v.size(); j++)
   {
     cout << tab.peligrosidad[i][j] << " " ;
   }
   cout << endl;
 }
 cout << endl;
}



tablaDePeligrosidad InicializarTablaDePeligrosidad() {
  tablaDePeligrosidad tab;
  cin >> tab.n;
  if(tab.n == 0)
    exit(0);
  cin >> tab.m;
  //construyo una matriz de n x n inicializada en 0
  vector< vector<int> > vec(tab.n, vector<int>(tab.n));
  tab.peligrosidad = vec;

  //le meto en la matriz los valores de peligrosidad de a pares
  for (int i = 0; i < tab.n; i++) {
    for(int j = i; j < tab.n; j++) {
      //la diagonal no tiene sentido
      if(j == i) {
        tab.peligrosidad[i][j] = 0;
      } else {
        int valor;
        cin >> valor;
```

```

        tab.peligrosidad[i][j] = valor;
        tab.peligrosidad[j][i] = valor;
      }
    }
  }
  return tab;
}

void borrarTablaDePeligrosidad(tablaDePeligrosidad *tab)
{
  for(int i = 0; i < tab->peligrosidad.size(); i++)
      tab->peligrosidad[i].clear();
  tab->peligrosidad.clear();
}


bool backtracking(tablaDePeligrosidad *tab, vector <int> &solParcialCamiones,
    vector <int> &solFinalCamiones)
{
  //imprimirResultado(solParcialCamiones);
  int resultadoCheck;
  if(solParcialCamiones.size() > tab->n) {
    return false;
  }

  resultadoCheck = check(tab,solParcialCamiones,solFinalCamiones);

  //es una olucion
  if(resultadoCheck == 2)
  {
      solFinalCamiones = solParcialCamiones;
      return true;
  }
  // solParcial usa mas camiones que solFinal
  if(resultadoCheck == 3) {
    return false;
  }

  if(resultadoCheck == 0) {
    return false;
  }

  //Me faltan asignar un producto a un camion
  for(int i = 0; i < tab->n; i++)
  {
    solParcialCamiones.push_back(i);
    backtracking(tab,solParcialCamiones,solFinalCamiones);
    solParcialCamiones.pop_back();
  }
  return false; // solo para completar casos, es imposible que se llegue a este
      punto.
}


//Si la solucion parcial usa mas camiones que la mejor solucion, podo.

bool solucionFinalUsaMenosCamiones(vector<int> &solParcialCamiones ,vector <int>
      &solFinalCamiones)
{
  int cantidad_de_camiones_en_la_solucion_parcial = solParcialCamiones[0];
```

```
  int cantidad_de_camiones_en_la_solucion_final = solFinalCamiones[0];

  for(int i = 0; i < solParcialCamiones.size(); i++) {
    if(cantidad_de_camiones_en_la_solucion_parcial < solParcialCamiones[i]) {
      cantidad_de_camiones_en_la_solucion_parcial = solParcialCamiones[i];
    }
  }

  for(int i = 0; i < solFinalCamiones.size(); i++) {
    if(cantidad_de_camiones_en_la_solucion_final < solFinalCamiones[i]) {
      cantidad_de_camiones_en_la_solucion_final = solFinalCamiones[i];
    }
  }

  return cantidad_de_camiones_en_la_solucion_final <
      cantidad_de_camiones_en_la_solucion_parcial;
}

bool cotaDePeligrosidadSobrepasada(tablaDePeligrosidad *tab, vector<int> &
    solParcialCamiones)
{
  //por cada camion, pongo su peligrosidad en 0
  vector<int> peligrosidad;
  for(int k = 0; k < solParcialCamiones.size() +3; k++) {//seteo los n valores
      del vector en 0.
    peligrosidad.push_back(0);
  }


  for(int i = 0; i < solParcialCamiones.size(); i++) //determino la peligrosidad
      por camion.
  {
    // miramos a partir del elemento i en el camion, cuanto le cuesta combinarse
        con el elemento j, que j es estrictamente mayor a i
    for(int j = i+1; j< solParcialCamiones.size(); j++) {
      int camion_del_producto_i = solParcialCamiones[i];
      int camion_del_producto_j = solParcialCamiones[j];
      //estan en el mismo camion???
      if(camion_del_producto_j == camion_del_producto_i) {
        peligrosidad[camion_del_producto_i] += tab->peligrosidad[i][j];

      }
    }
  }

  //reviso si alguna peligrosidad se paso de la cota
  for(int h = 0; h < solParcialCamiones.size(); h++)
  {
    if(tab->m < peligrosidad[h])
      return true;
  }
  return false;
}

// check:
// si retorna 0 no es sol valida
// si retorna 1, es valida pero falta agregar camiones
// si retorna 3 la solucion anterior usa menos camiones
// si retorna 2 es solucion valida
int check(tablaDePeligrosidad *tab, vector<int> &solParcialCamiones ,vector <int
    > &solFinalCamiones)
```

```
{

  //mi resultado final sigue siendo mejor
  if(solucionFinalUsaMenosCamiones(solParcialCamiones,solFinalCamiones))
    return 3;

  //me pase, abort!
  if(cotaDePeligrosidadSobrepasada(tab,solParcialCamiones))
    return 0;

  //
  if(tab->n == solParcialCamiones.size())
    return 2;


  return 1;
}

void imprimirResultado(vector<int> solParcialCamiones)
{
 int max = solParcialCamiones[0];
 for(int i = 1; i < solParcialCamiones.size(); i++)
   if(solParcialCamiones[i] > max)
    max = solParcialCamiones[i];
 cout << max + 1<< " ";
 for(int i = 0; i < solParcialCamiones.size(); i++)
   cout << solParcialCamiones[i] + 1 << " ";
 cout << endl;
}



void inicializarPeorSol(vector<int> &sol,int n)
{
  sol.clear();
  for(int i = 0; i < n; i++)
      sol.push_back(i);
}

int main()
{
  timeval tm1, tm2;
  gettimeofday(&tm1, NULL);
  while(true){
    tablaDePeligrosidad tab = InicializarTablaDePeligrosidad();
    vector<int> solParcialCamiones;
    vector<int> solFinalCamiones;

    inicializarPeorSol(solFinalCamiones,tab.n);

    //Agrego el primer producto al camion y lanzo la recursion
    solParcialCamiones.push_back(0);
    bool sol = backtracking(&tab, solParcialCamiones,solFinalCamiones);
    imprimirResultado(solFinalCamiones);
    solParcialCamiones.clear();
    solFinalCamiones.clear();
    borrarTablaDePeligrosidad(&tab);
  }

  return 0;
}
```