Algoritmos y Estructuras de Datos III

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

RTP1

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, dada esta limitación de complejidad temporal ya podemos intuir que no debemos recorrer mas de una vez los mismo tablones y de esta manera poder cerciorarnos de que cumplimos esta complejidad.

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 está roto, 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: En cada paso el algoritmo goloso busca quedarse con la solución optima, esta es el salto máximo que el participante puede saltar, con lo cual en cada paso de nuestro algoritmo lo que realiza es intentar este salto si es factible, en caso que el tablón está sano saltamos a ese tablón y continuamos de manera iterativa, caso contrario comenzamos a revisar desde el salto máximo acercándonos tablón por tablón hasta la posición actual hasta encontrar un tablón sano. Esto podría no ocurrir y en ese caso quiere decir que hay una cantidad de tablones rotos mayor al salto máximo el participante y en esta situación no tenemos solución.

En el ejemplo el mejor recorrido es ir por los siguientes tablones: `3, 6, 9, 12, 15`

1.0.3. Desarrollo

Para la solución de este problema recurrimos a la técnica de algoritmos golosos, donde en cada paso en la construcción a la solución obtenemos el mejor salto. Por el enunciado sabemos que tenemos n tablones, el participante puede saltar C tablones de una sola vez y cuáles son las posiciones de los tablones rotos. Nos armamos un arreglo donde cada posición representa un tablón, en el cual guardamos 0 y 1 para indicar su estado (sano o roto respectivamente) llenando el mismo según la entrada, llamémoslo puente.
El caso trivial donde la cantidad maxima de tablones consecutivos que el participante puede saltar el algoritmo devuelve 1 salto y solo la posición final.

Nuestro algoritmo ira recorriendo el puente tratando de hacer el mayor salto cuando este es posible, dependiendo si el tablón al que iría a parar el participante está sano, en caso contrario comenzamos a iterar los tablones desde este salto máximo hacia donde se encuentra parado el participante uno por uno hasta encontrar un tablón sano.

Para cada tablón al cual se salta se guarda su posición en una lista siempre insertando atrás de la misma, una vez que cada participante cruzó todo el puente se devuelve esta lista como la sucesión de saltos realizada por el participante, en caso de no ser factible ya que en una sección del puente la cantidad de tablones rotos consecutivos es mayor al salto máximo se devuelve no, ya que no tiene solución.

1.0.4. Complejidad

El siguiente es un pseudo-código de nuestro algoritmo.

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

```
1: Si saltoMax > cantidadTablones    O(1)
2:    devolver cantidadTablones+1    O(1)
3: SI NO
4:    SI existe una cantidad de tablones rotos mayor o igual a salto_maximo en algún lugar del puente
   O(n)
5:       no existe solución
6:    SI NO nueva listaDeSaltos    O(1)
7:       MIENTRAS posActual < n and posActual >= 0    O(n)
8:          SI puente(posActual) == 0   O(1)
9:             Agrego posActual a Lista de Saltos   O(1)
10:            le sumo a posActual el largo del salto    O(1)
11:         SI NO
12:            Le resto uno a PosActual    O(1)
13: Devolver listaDeSaltos
```

Todas las asignaciones y comparaciones son en O(1) como está marcado en el pseudocodigo, ya que son números naturales y están acotados por la cantidad de tablones, las asignaciones a las lista son en O(1)

En la línea 4 para revisar si la cantidad de tablones rotos es mayor o igual al salto máximo en alguna parte del puente se realiza un ciclo que cuenta la cantidad de tablones rotos consecutivos, en caso de que se encuentre con uno sano resetea el contador y comienza de nuevo a contar, en el caso que el contador llegue a superar el salto máximo se devuelve que no tiene solución. Esto se realiza con un solo ciclo y la complejidad es O(n).

El ciclo de las líneas 7 - 13 se realiza n veces en el peor caso con lo cual la complejidad total del algoritmo es O(n), ya que los dos posibles peores casos son, cuando el puente está sano y salta de a un tablón por vez, o cuando los únicos tablones sanos son en las posiciones 1, Salto Máximo +1 , 2*(salto Máximo +1)..., de esta manera siempre salta el máximo y recorre los tablones hasta el primero para encontrar el sano o sucede que en un salto encuentra el tablón roto pero al próximo salto vuelva a repetir esta operación, dando como resultado que recorre todo el puente para encontrar los saltos.

1.0.5. Demostración

Para la prueba de correctitud, queremos demostrar que nuestro algoritmo siempre encuentra una sucesión de saltos que es la menor posible.

Luego para demostrar por absurdo, supongamos que no es así. Dicho más formalmente, existe una sucesión $V = \{v_1, v_2, ..., v_i\}$ de saltos tal que es menor a la sucesión $W = \{w_1, w_2, ..., w_i, w_{i+1}, ..., w_j\}$ que encuentra nuestro algoritmo.

Por condiciones del problema sabemos el tablon al que queremos saltar no puede estar a distancia mayor a k (el salto maximo posible), esto significa que:

$$v_1 \leq k$$

$$v_2 - v_1 \leq k$$

$$...$$

$$v_i - v_{i-1} \leq k$$

Sumando:

$$v_1 \leq k$$
$$v_2 \leq 2.k$$
$$...$$
$$v_i \leq i.k$$

Pero nuestro algoritmo, en cada paso, salta la mayor cantidad de tablones posibles y de allí comienza a retroceder hasta encontrar uno sano, dado que $v_1, v_2, ..., v_i$ deben ser tablones sanos, los tablones elegidos por nuestro algoritmo estarán acotados de esta manera:

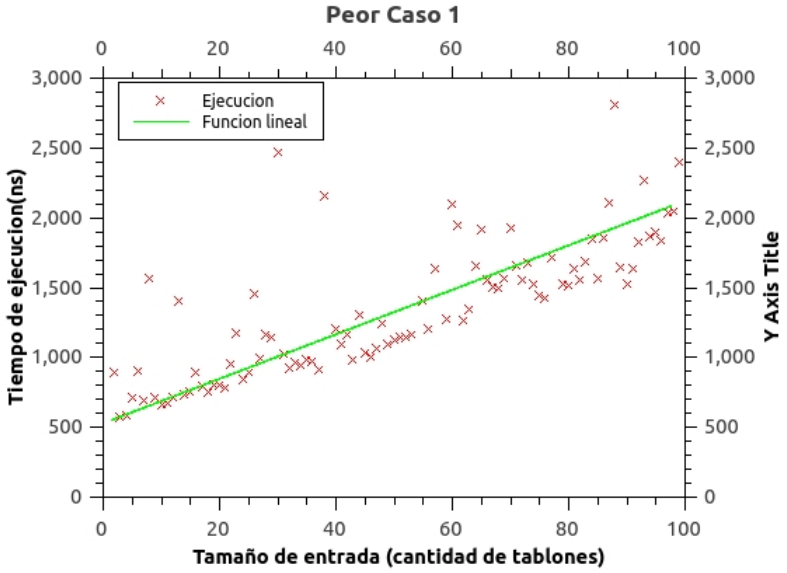
$$v_1 \leq w_1 \leq k$$
$$v_2 \leq w_2 \leq 2.k$$
$$...$$
$$v_i \leq w_i \leq i.k$$

Pero entonces, esto quiere decir que nuestro algoritmo encuentra una sucesión de saltos, para los cuales siempre se salta por lo menos, la misma cantidad de tablones que en la sucesión V. Luego nuestro algoritmo encuentra una solución W que tiene, como mucho, k saltos.

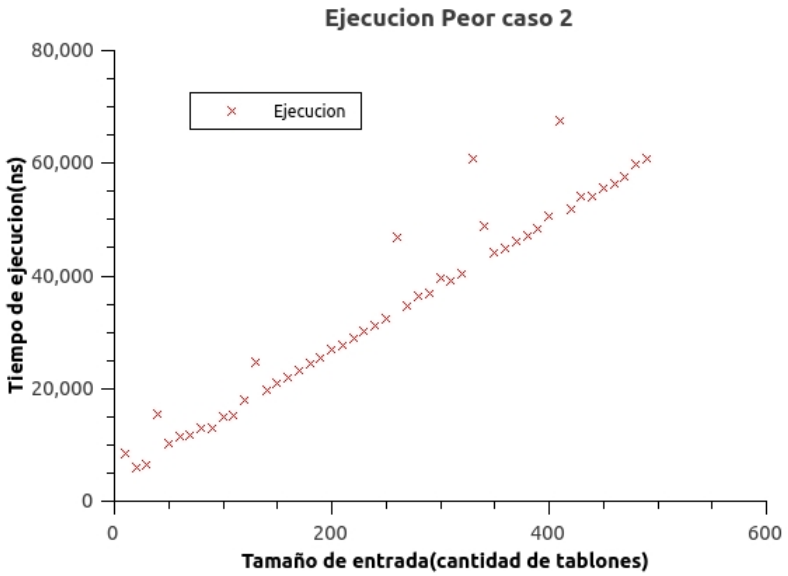
Dado que supusimos que nuestro algoritmo encontraba una sucesión que era peor a V, absurdo. Luego nuestro algoritmo encuentra siempre una sucesión de saltos que es mínima.

1.0.6. Experimentacion

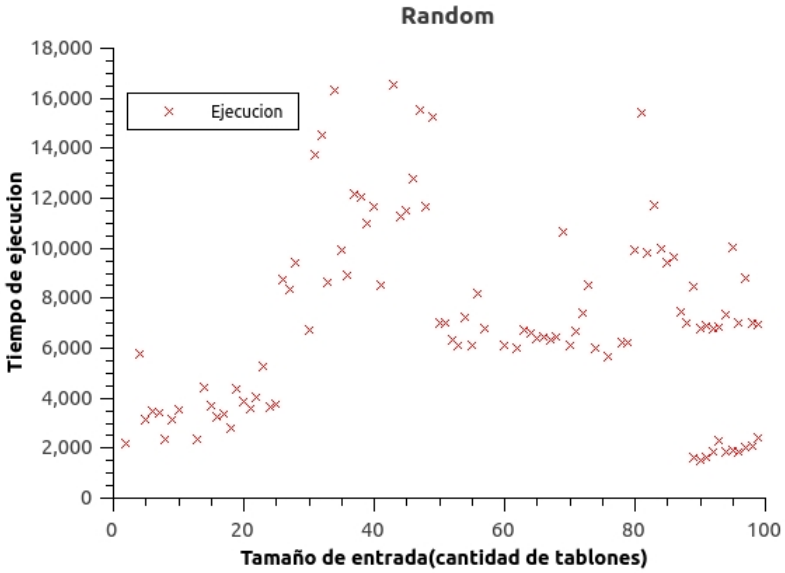
Para la experimentación del problema en cuestión se realizaron los dos test de peores casos como se mencionó en la sección de complejidad, el primero teniendo en cuenta el caso que todos los tablones del puente estén sanos y el salto máximo del participante sea uno, para esto se fijó el salto máximo y se crearon 100 instancias de puentes sanos en tamaños que van del 1 al 100 luego se midieron los tiempos de ejecución dando como resultado el siguiente gráfico.



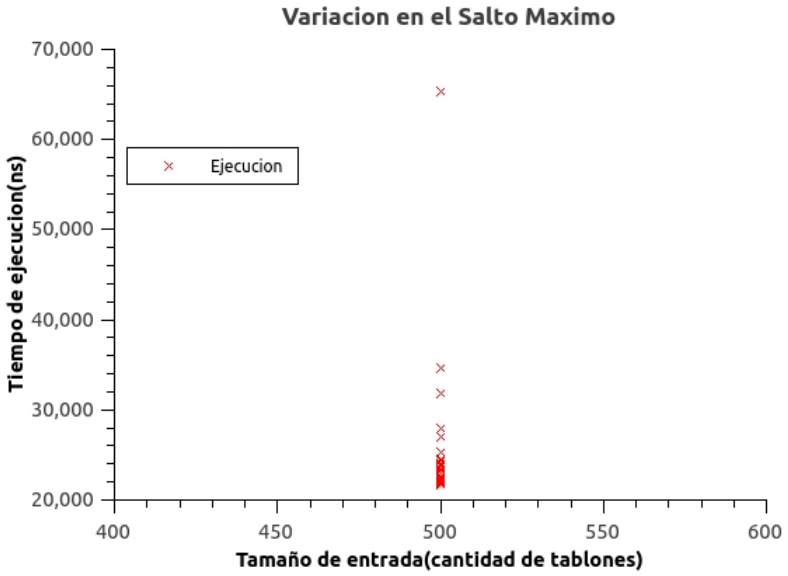
Como puede verse en el gráfico su complejidad está en el orden lineal como puede compararse con la función, en el gráfico puede observar que algunos casos excede el orden, este ruido se debe a que puede haber ovaciones donde el sistema operativo este realizando otras tareas y afecte el orden de ejecución. En el segundo test, se realizó el segundo caso mencionado en la sección complejidad y dando como resultado un gráfico con complejidad del orden lineal como se puede observar en el siguiente gráfico.



Para el tercer test se realizó 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 500 y el salto máximo es un número 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.



Como puede observarse en el grafico los tiempos están bastante esparcidos esto se debe al que para la misma cantidad de tablones y diferente salto máximo puede obtener distintos tiempos, con lo cual realizamos otro test donde se puede verificar que para una cantidad de 500 tablones y modificando el salto máximo obtenemos distintos tiempos en la misma línea de 500 tablones.



Nuestro algoritmo nos devuelve en O(1) si el k es mayor a la cantidad de tablones y caso contrario se ejecuta y esto podría darte los peores casos que observamos anteriormente, y es coherente que el tiempo de ejecución varia según este k como puede verse en el último gráfico con claridad.

Por lo tanto concluimos que nuestro algoritmo sigue un orden lineal con respecto a la cantidad de tablones.

1.0.7. Ejercicio Auxiliar

En la nueva versión del juego, algunos tablones sanos contienen premios. Si el participante cae en uno de estos tablones, a partir de ese momento su capacidad máxima de salto se vería incrementada en una Unidad. Se pide desarrollar los siguientes puntos:

¿Cómo afecta eso a su algoritmo?

Dado que cuando cae un tablón que tiene premio aumenta en uno la capacidad del máximo salto lo que debemos modificar de nuestro algoritmo es que cuando encuentra un tablón sano verificar si este contiene premio y sumarle una unidad antes de sumarlo a la posición actual para determinar el posible nuevo salto.

¿Qué característica del problema cambia en esta nueva versión?

Con esta nueva característica lo que podría suceder es que no encuentre una solución optima al problema, el caso trivial seria aquel el que todos los tablones están sanos y colocando premio en todas las posiciones excepto aquellas donde sea un múltiplo del salto máximo, de esta manera si nos movemos un tablón para atrás obtendríamos premio, teniendo más probabilidades de tener premio en el próximo salto y al aumentar el salto máximo podría obtener una solución difiera en la cantidad de saltos total. obteniendo una mejor solución. Si llevamos al ejemplo a una forma gráfica supongamos que este es nuestro puente: `[(1,p),(1,p),(0,p),(1,p),(1,p),(0,p),(1,p)(1,p)(0,p)(1,p)(1,p)(0,p)]`

Ahora supongamos que el salto máximo es 3. La solución obtenida serían los tablones (3,6,9,12,13), donde el 13 representa que salió del puente. Veamos que si el primero saltamos al 2 obtendríamos la siguiente solución (2,5,9,13) siendo esta una solución mejor a la obtenida utilizando el salto máximo.

Capítulo 2

Horizontes lejanos

2.0.8. 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.

2.0.9. 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).
$[< 1, 4, 2 >; < 3, 2, 5 >; < 4, 1, 6 >; < 6, 8, 10 >]$

2.0.10. 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.

2.0.11. 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) $\rightarrow$ 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.

2.0.12. Complejidad

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

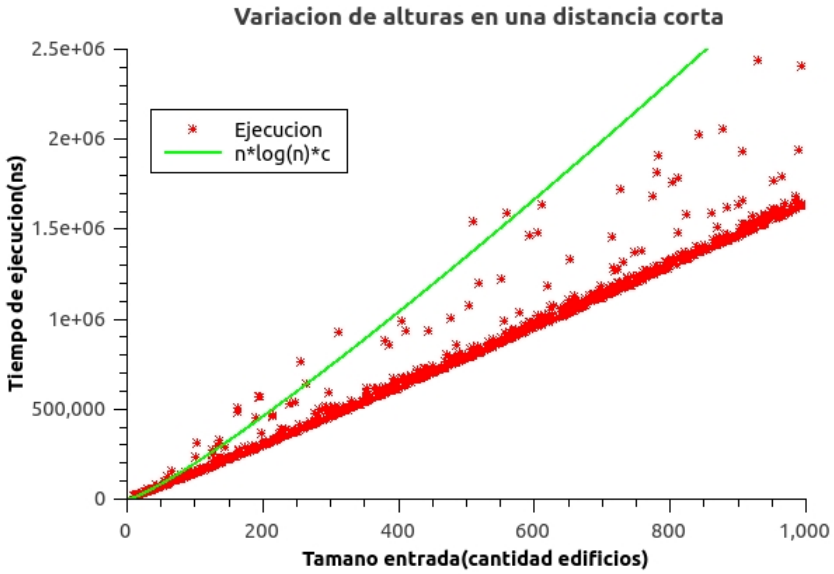
Algorithm 2 LaSilueta(ciudad: arreglo(edificios) , cantidadEdificios : natural)

```
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)
```

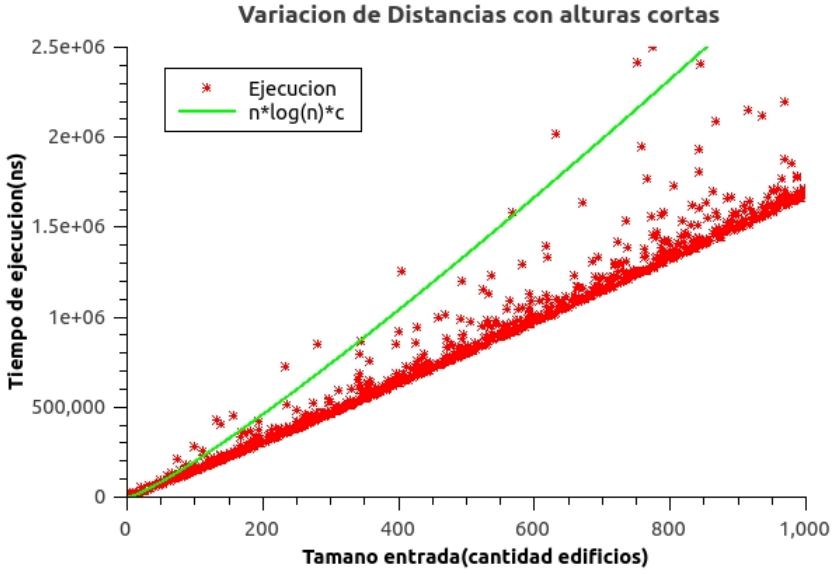
se ejecuta mientras la cantidad de paredes recorridas izquierdas sea menor a la cantidad de edificios y la cantidad de paredes derechas sea menor a la cantidad de edificios y las iteraciones internas modifican las paredes izquierdas y derechas recorridas, el total de las iteraciones es 2*N, siendo N 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 $O(2 * N * Log(N)) \subset O(N * Log(N))$ siendo esta la complejidad total del algoritmo y esta estrictamente menor a $O(N^2)$ como se solicitaba en el enunciado.

2.0.13. Experimentación

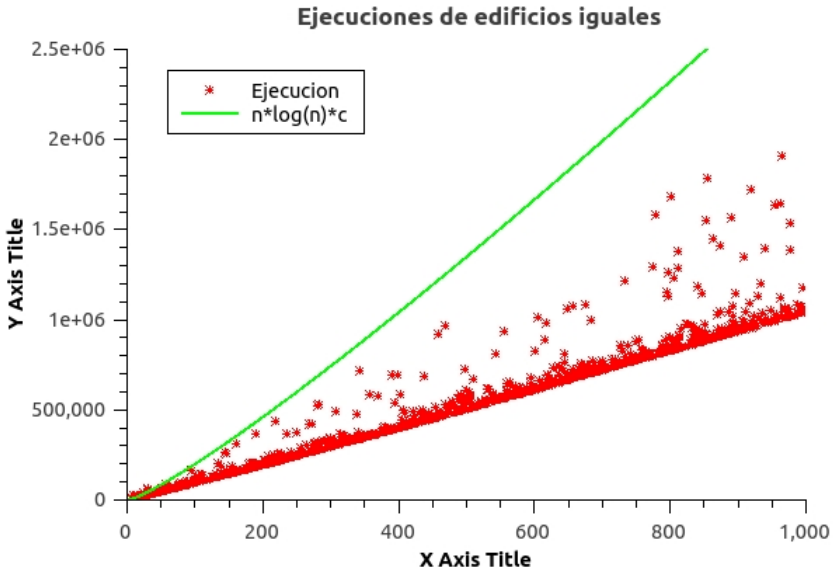
Se realizaron experimentaciones sobre varios tipos de caso para observar el comportamiento del algoritmo, en todos los casos se generaron 1000 entradas, a las gráficas resultantes se las comparo con la función n*log(n)*1000, siendo n la cantidad de edificios de entrada.
Para la primer ejecución se realizo 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.



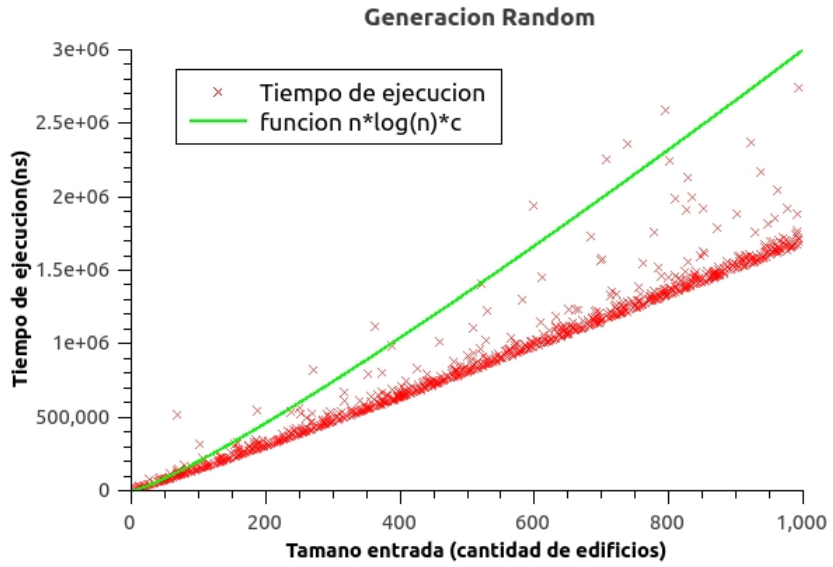
El segundo gráfico corresponde a instancias generadas de manera random donde la altura se encontraba entre 1 y 40, pero las distancias entre la primer y la segunda pared de cada edificio varia entre los 1 y 2000 , elegido de manera random la posición de ambos lados pero siempre que sea entrada valida, 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:
  - $h_i, i{+}1, h_i, i{+}2 \ldots h_i, n \rightarrow$ 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.

Dado que el problema no presenta ningun tipo de complejidad, pero debemos minimizar al cantidad de camiones cumpliendo cierta cota de peligrosidad, debemos siempre devolver una solucion por eso recurrimos a la tecnica de backtracking para generar dicha solución. Para lo cual lo que se va a realizar es probar todas las combinaciones entre quimicos, de igual manera se implementaran podas en el backtracking como se describen en la siguientes secciones para mejorar la complejidad obteniedo asi una solucion optima de una manera un poco mas eficiente que la solucion sin podas.

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 soluciónza 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, ya que supongamos que si tenemos una solucion que utiliza k cantidad de camiones, k natural , y en cierto punto de nuestro backtracking estamos obteniendo como solucion parcial llevamos m camiones y m>k ya estariamos obteniendo una solucion que no seria la optima ya que buscamos minimizar la cantidad de camiones. Es claro que de ser así, estamos la solucion parcial nunca podrá ser mejor, por lo tanto se podará toda esa familia de soluciones mejorando asi la complejidad temporal del algoritmo.

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()

1: Generar una matriz con las peligrosidades entre los distintos productos
2: Se inicializa la solucion final, como la peor de las soluciones
3: Backtrack(tablaDePeligrosidad, solucionParcial, solucionFinal)
4: 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:

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

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

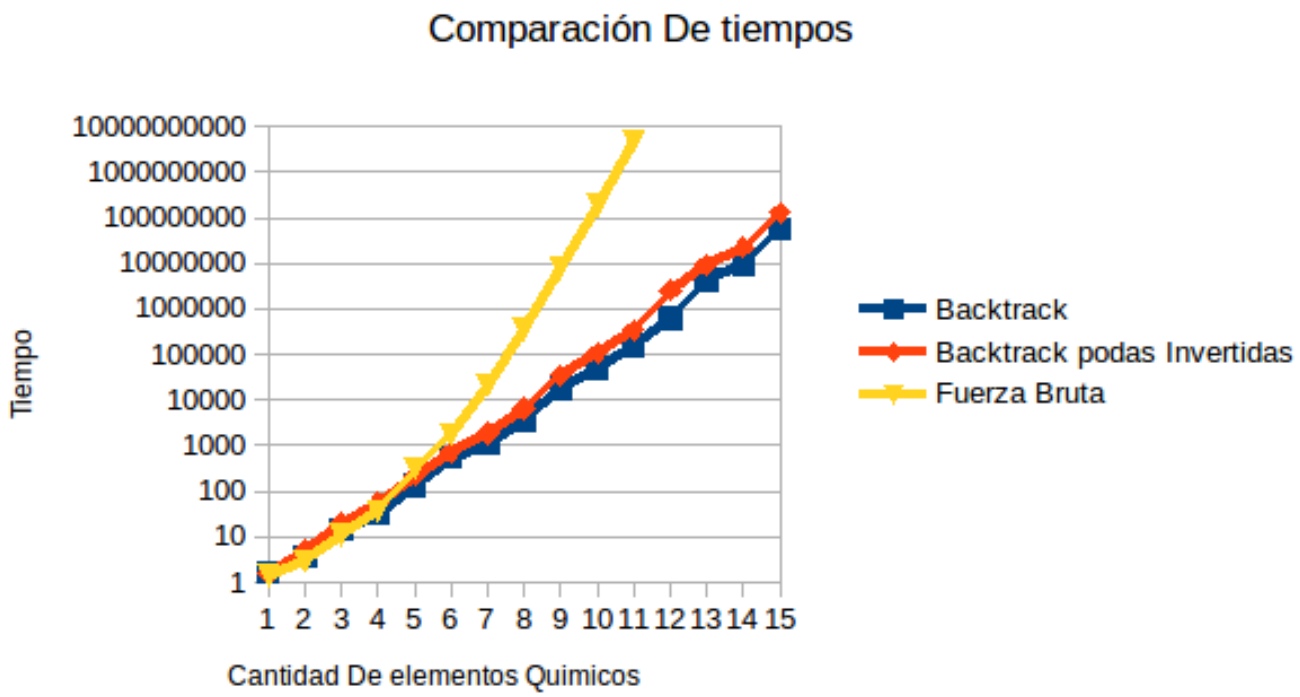
Luego para el peor de los casos el algoritmo tendrá una complejidad de $O(n^n)$.

3.5. Resultados

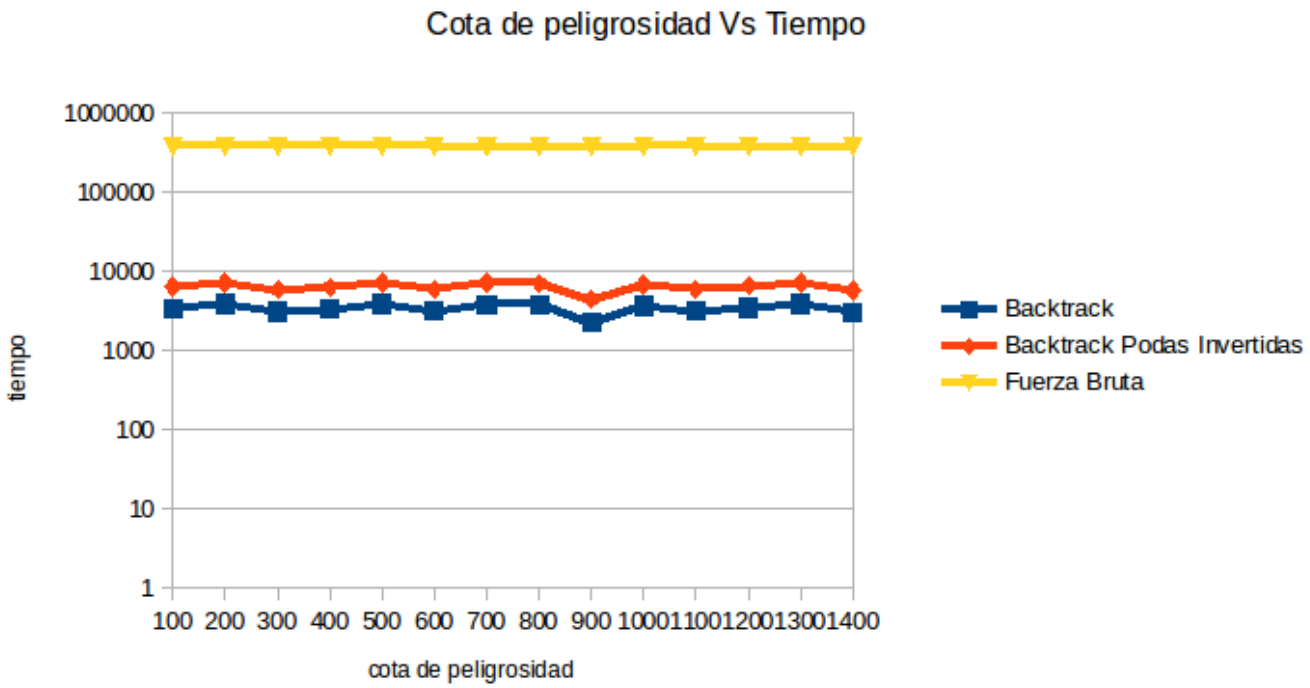
3.5.1. Testing

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:



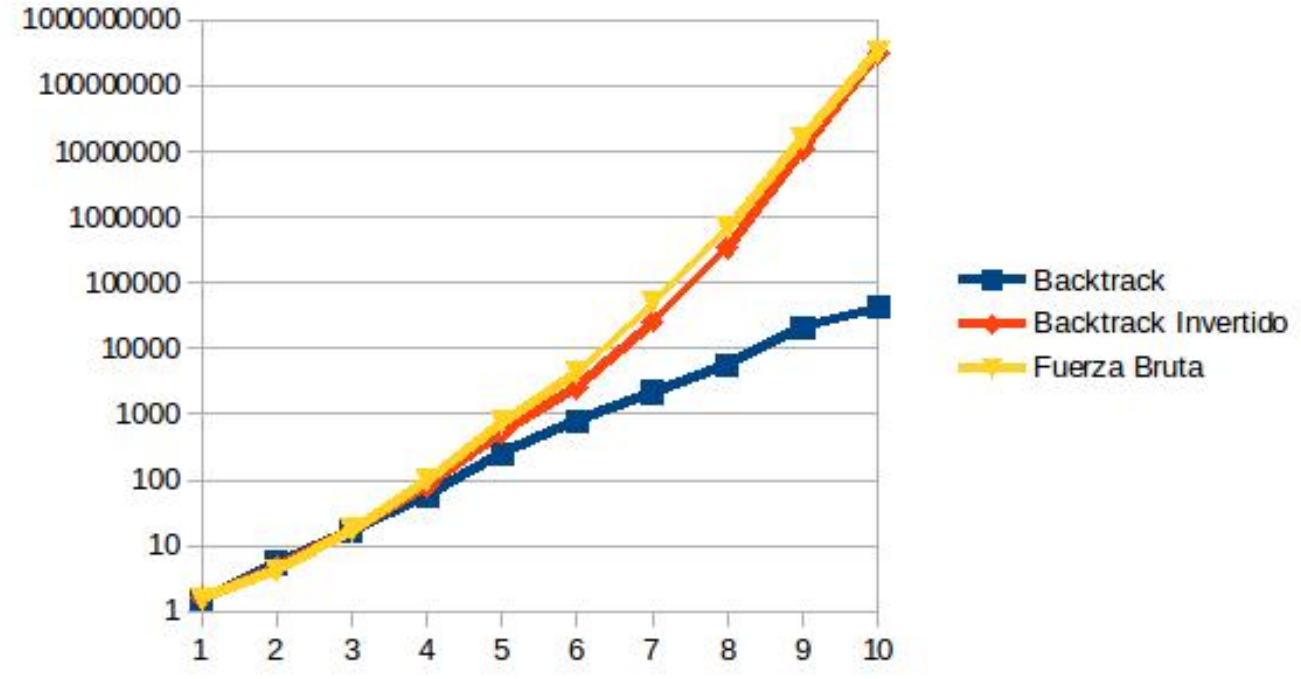
En el gráfico puede verse que nuestro algoritmo es notablemente superior a un algoritmo de fuerza bruta, ya que escala mucho mejor con respecto a n y levemente mejor al backtracking con las podas invertidas. Para los casos de 14 y 15 elementos el backtracking pudo arrojar una respuesta en un tiempo admisible, 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. Podemos Concluir entonces que nuestro algoritmo siempre que exista solucion la obtiene y ademas puede constatarse con los graficos que las complejidades temporales mejoran obteniendose el mismo resultado cuando el algoritmo utiliza las podas y cuando realiza sin podas el backtracking, el inconveniente de esta tecnica es que en casos muy grandes demora mucho por lo que no es la mejor opcion para casos de muchos productos ya que podria no terminar.

3.6. Adiconales

1) y 2)Ahora que los camiones no son homogeneos, se amplía el espacio de soluciónes a explorar, ya que, por ejemplo, antes con tres camiones homogeneos, bastaba chequear las soluciones:

- 1 2 3
- 1 1 2
- 1 1 1
- 1 2 1

Ahora que los camiones no son uniformes, esto ya no es así, ya que la cota de peligrosidad por camion varía. Por lo tanto ahora se deverán explorar las siguientes soluciónes:

- 1 2 3
- 1 1 1
- 2 2 2
- 3 3 3
- 1 1 2
- 1 1 3

- 2 2 1
- 2 2 2
- 3 3 1
- 3 3 2
- 1 2 3 ...

En resumen, pasamos de tener que chequear 4 posibles combinaciones a 27.

Dicho para cualquier cantidad n de productos quimicos, ahora tendrémos que chequear n^n combinaciones diferentes cuando antes solo teníamos que chequear $\frac{(2n-1)!}{n!(n-1)!}$ posibilidades (Esto se obtiene por combinatoria, es elegir para cada producto un camión n, sin que importara el orden y pudiendo repetir).

Luego se ve que el espacio de soluciones se amplía enormemente, por lo que el tiempo de ejecución del algoritmo será mucho peor.

En nuestro caso, ambas podas se podrán aplicar al algoritmo, oesa, tanto la poda que chequeaba si la mejor solucón que encontré hasta ahora sigue valiendo utiliza menos camiones que la que estoy construllendo en este momento, como la poda que chequeaba la cota la cota de peligrosidad en cada paso esta siendo sobrepasada, siguen siendo aplicables, con la pequeña modificación que al chequear esto ultimo se debe tener en cuenta en que camión se estan poniendo los productos quimicos.

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;
}

int main()
{
        while(true){
                int n, largo_del_salto;
                cin >> n;
                if(n == 0) {
                        return 0;
                }
                cin >> largo_del_salto;
                vector<int> puente;
                //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);
                }
                    //miro si tiene solucion
                auto begin = std::chrono::high_resolution_clock::now();
                if(!check_solution(puente,largo_del_salto)){
```

```
56                          //cout << "no" << endl;;
57                          continue;
58                  }
59                  // cout << n << " " << largo_del_salto << " ";
60                  // for(int i = 0 ; i< n ; i++){
61                  //      cout << puente[i] << " ";
62                  // }
63                  // cout << endl;
64                  if(largo_del_salto > n){
65                          //cout << "1 " << n+1 << endl;
66                          continue;
67                  }
68
69                  int posActual = largo_del_salto-1;
70                  list<int> recorrido;
71                  bool llegue = false;
72                  int iteraciones = 0;
73                  while(posActual<n&&posActual>=0){
74                          iteraciones++;
75                          if(puente[posActual]==0){
76                                  recorrido.push_back(posActual+1);
77                                  posActual = posActual + largo_del_salto;
78                          }else{
79                                  posActual--;
80                          }
81                  }
82                  recorrido.push_back(n+1);
83               auto end = std::chrono::high_resolution_clock::now();
84              std::cout << n << ' ' << std::chrono::duration_cast<std::chrono::
                    nanoseconds>(end-begin).count();
85              cout << std::endl;
86
87              //imprimir_vector(recorrido);
88    }
89    return 0;
90  }
```

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;
```

```
58      int i=low;
59      int j=pivot+1;
60      int elemento_h;
61      int elemento_j;
62      while((h<=pivot)&&(j<=high))
63      {
64        if(control == 0){
65          elemento_h = a[h].izq;
66          elemento_j = a[j].izq;
67        }else{
68          elemento_h = a[h].der;
69          elemento_j = a[j].der;
70        }
71
72          if( elemento_h < elemento_j  )
73          {
74              b[i]=a[h];
75              h++;
76          }else{
77              b[i]=a[j];
78              j++;
79          }
80          i++;
81      }
82      if(h>pivot)
83      {
84          for(k=j; k<=high; k++)
85          {
86              b[i]=a[k];
87              i++;
88          }
89      }else{
90          for(k=h; k<=pivot; k++)
91          {
92              b[i]=a[k];
93              i++;
94          }
95      }
96      for(k=low; k<=high; k++) {
97                  a[k]=b[k];
98      }
99  }
100
101
102 vector< sol > resolver(int cantEdificios, edificio* edificiosIzq, edificio*
       edificiosDer){
103   vector<sol> res;
104   //ordernar por izq
105   edificio* auxiliarParaOrdenar = new edificio[cantEdificios];
106
107   mergesort(edificiosIzq,auxiliarParaOrdenar,0, cantEdificios-1,0);
108
109   // //SOlo para test
110   //    cout << "orden por izq \n";
111   //    for(int i = 0; i<cantEdificios; i++){
112   //      cout <<"( " << edificiosIzq[i].id  << "," ;
113   //      cout << edificiosIzq[i].izq  << ",";
114   //      cout << edificiosIzq[i].alt  << ",";
115   //      cout << edificiosIzq[i].der  << ");";
116
117   //    }
```

```
118     //    cout << "\n";
119     //FIN TEST
120
121
122     //ordenar por derecha
123     mergesort(edificiosDer,auxiliarParaOrdenar,0, cantEdificios-1,1);
124
125     //Solo para test
126        // cout << "orden por der \n";
127        // for(int i = 0; i<cantEdificios; i++){
128        //   cout <<"( " << edificiosDer[i].id  << ",";
129        //   cout << edificiosDer[i].izq  << ",";
130        //   cout <<  edificiosDer[i].alt  << ",";
131        //   cout << edificiosDer[i].der  << ");";
132        // }
133        // cout << "\n";
134     //FIN TEST
135
136     multiset<edificio> magiheap;
137     int posDer = 0;
138     int posIzq =0;
139     edificio max = {0,0,0,0};
140     int posMax = 0;
141
142     sol nueva = {0,0};
143     bool finIzq = false;
144     while(posDer != cantEdificios){
145
146
147           if((finIzq) || (edificiosIzq[posIzq].izq > edificiosDer[posDer].der)){
148             //Saco edificio o edificios
149             int base = posDer;
150             int maxAnt = max.alt;
151             while ((edificiosDer[posDer].der == edificiosDer[base].der) && (posDer
                    != cantEdificios)){
152               multiset<edificio>::iterator it = magiheap.lower_bound(edificiosDer[
                      posDer]);
153
154               if (it->id == max.id){
155                 if(magiheap.size() > 1){
156                   multiset<edificio>::iterator itMax = magiheap.end();
157                   itMax--;
158                   if (it->id == itMax->id){
159                     itMax--;
160                   }
161                   max.alt = itMax->alt;
162                   max.izq = itMax->izq;
163                   max.der = itMax->der;
164                   max.id = itMax->id;
165                 }else{
166                     max.id = 0;
167                     max.izq = 0;
168                     max.alt = 0;
169                     max.der = 0;
170                 }
171               }
172               magiheap.erase(it);
173               posDer++;
174             }
175             if (maxAnt > max.alt){
176                 nueva.x = edificiosDer[base].der;
```

```
177                 nueva.alt = max.alt;
178                 res.push_back(nueva);
179             }
180           }else{
181             //Agrego edificio o edificios
182             int base = posIzq;
183             int maxAnt = max.alt;
184             while ((edificiosIzq[posIzq].izq == edificiosIzq[base].izq) && (posIzq
                    != cantEdificios)){
185               edificio nuevoEdi = *(magiheap.insert(edificiosIzq[posIzq]));
186               if (nuevoEdi.alt > max.alt){
187                 max = nuevoEdi;
188               }
189               posIzq++;
190             }
191             if (maxAnt < max.alt){
192                 nueva.x = max.izq;
193                 nueva.alt = max.alt;
194                 res.push_back(nueva);
195             }
196             if (posIzq==cantEdificios){
197               finIzq = true;
198             }
199           }
200     }
201 
202 
203     return res;
204 
205 }
206 
207 
208 int main(int argc, char *argv[]){
209 
210           string line;
211           bool primerLinea = true; //para saber cuando tomamos la primera linea,
                 el n.
212           int cantEdificios = 0;
213           edificio* edificiosIzq = NULL;
214           edificio* edificiosDer = NULL;
215           int leidas = 1;
216            timeval tm1, tm2;
217           //Si no llegue al final del archivo y sigo obteniendo lineas sigo
                 guardando
218           while ( getline (cin,line) )
219           {
220               if(line[0] == '0'){
221                 break;
222               }
223 
224               if(primerLinea){
225                 cantEdificios = atoi(line.c_str()); //El n de la entrada
226                 edificiosIzq = new edificio[cantEdificios];
227                 edificiosDer = new edificio[cantEdificios];
228                 primerLinea = false;
229                 leidas = 1;
230               }else{
231 
232 
233                 vector<string> entradaSplit;
234                 int fromIndex = 0;//inicio string
```

```
235                 int length = 0; //longitud string
236                 for(int i = 0; i<line.length();i++ ){
237                       if(i== line.length()-1){
238                         length++;
239                         entradaSplit.push_back(line.substr(fromIndex, length));
240                       }else if(line[i]== ' '){
241                         entradaSplit.push_back(line.substr(fromIndex, length));
242                         fromIndex = i+1;
243                         length = 0;
244                       }else{
245                         length++;
246                       }
247                   }//Fin for
248 
249                   int izq = atoi(entradaSplit[0].c_str());
250                   int alt = atoi(entradaSplit[1].c_str());
251                   int der = atoi(entradaSplit[2].c_str());
252                   edificio nueva = {leidas,izq,alt,der};
253                   //cout << nueva.id << " " << nueva.izq << " ";
254                   //cout << nueva.alt << " " << nueva.der << "\n";
255                   edificiosIzq[leidas-1]=nueva;
256                   edificiosDer[leidas-1]=nueva;
257 
258                   if(leidas == cantEdificios){
259                     primerLinea = true;
260                     //Aca llamamos a resolver
261 
262                     //para medir tiempos , mido antes de empezar
263                     //gettimeofday(&tm1, NULL);
264                   auto begin = std::chrono::high_resolution_clock::now();
265                   vector<sol> resultado = resolver(cantEdificios,edificiosIzq,
                        edificiosDer);
266                     // mido cuando termino
267                       auto end = std::chrono::high_resolution_clock::now();
268 
269                    cout << cantEdificios << " " << std::chrono::duration_cast<std
                        ::chrono::nanoseconds>(end-begin).count();
270                     //solo para imprimir tiempos descomentar la linea siguiente
271                       cout << endl;
272 
273 
274               //Una vez obtenida la solucion imprimimos el resultado
275                       // for(int i =0;i< resultado.size();i++){
276                       //      cout << resultado[i].x << " " << resultado[i].alt <<
                            " ";
277                       //  }
278                       //  cout << "\n";
279                   }
280 
281                   leidas++;
282 
283                 }
284               }
285 
286 
287 
288           return 0;
289 
290   }
```

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;
```

```
58
59          tab.peligrosidad[i][j] = valor;
60          tab.peligrosidad[j][i] = valor;
61        }
62      }
63    }
64    return tab;
65  }
66
67  void borrarTablaDePeligrosidad(tablaDePeligrosidad *tab)
68  {
69    for(int i = 0; i < tab->peligrosidad.size(); i++)
70        tab->peligrosidad[i].clear();
71    tab->peligrosidad.clear();
72  }
73
74
75  bool backtracking(tablaDePeligrosidad *tab, vector <int> &solParcialCamiones,
        vector <int> &solFinalCamiones)
76  {
77    //imprimirResultado(solParcialCamiones);
78    int resultadoCheck;
79    if(solParcialCamiones.size() > tab->n) {
80      return false;
81    }
82
83    resultadoCheck = check(tab,solParcialCamiones,solFinalCamiones);
84
85    //es una olucion
86    if(resultadoCheck == 2)
87    {
88        solFinalCamiones = solParcialCamiones;
89        return true;
90    }
91    // solParcial usa mas camiones que solFinal
92    if(resultadoCheck == 3) {
93      return false;
94    }
95
96    if(resultadoCheck == 0) {
97      return false;
98    }
99
100   //Me faltan asignar un producto a un camion
101   for(int i = 0; i < tab->n; i++)
102   {
103     solParcialCamiones.push_back(i);
104     backtracking(tab,solParcialCamiones,solFinalCamiones);
105     solParcialCamiones.pop_back();
106   }
107   return false; // solo para completar casos, es imposible que se llegue a este
        punto.
108 }
109
110
111 //Si la solucion parcial usa mas camiones que la mejor solucion, podo.
112
113 bool solucionFinalUsaMenosCamiones(vector<int> &solParcialCamiones ,vector <int>
        &solFinalCamiones)
114 {
115   int cantidad_de_camiones_en_la_solucion_parcial = solParcialCamiones[0];
```

```
116   int cantidad_de_camiones_en_la_solucion_final = solFinalCamiones[0];
117
118   for(int i = 0; i < solParcialCamiones.size(); i++) {
119     if(cantidad_de_camiones_en_la_solucion_parcial < solParcialCamiones[i]) {
120       cantidad_de_camiones_en_la_solucion_parcial = solParcialCamiones[i];
121     }
122   }
123
124   for(int i = 0; i < solFinalCamiones.size(); i++) {
125     if(cantidad_de_camiones_en_la_solucion_final < solFinalCamiones[i]) {
126       cantidad_de_camiones_en_la_solucion_final = solFinalCamiones[i];
127     }
128   }
129
130   return cantidad_de_camiones_en_la_solucion_final <
        cantidad_de_camiones_en_la_solucion_parcial;
131 }
132
133 bool cotaDePeligrosidadSobrepasada(tablaDePeligrosidad *tab, vector<int> &
      solParcialCamiones)
134 {
135   //por cada camion, pongo su peligrosidad en 0
136   vector<int> peligrosidad;
137   for(int k = 0; k < solParcialCamiones.size() +3; k++) {//seteo los n valores
        del vector en 0.
138     peligrosidad.push_back(0);
139   }
140
141
142   for(int i = 0; i < solParcialCamiones.size(); i++) //determino la peligrosidad
        por camion.
143   {
144     // miramos a partir del elemento i en el camion, cuanto le cuesta combinarse
          con el elemento j, que j es estrictamente mayor a i
145     for(int j = i+1; j< solParcialCamiones.size(); j++) {
146       int camion_del_producto_i = solParcialCamiones[i];
147       int camion_del_producto_j = solParcialCamiones[j];
148       //estan en el mismo camion???
149       if(camion_del_producto_j == camion_del_producto_i) {
150         peligrosidad[camion_del_producto_i] += tab->peligrosidad[i][j];
151
152       }
153     }
154   }
155
156   //reviso si alguna peligrosidad se paso de la cota
157   for(int h = 0; h < solParcialCamiones.size(); h++)
158   {
159     if(tab->m < peligrosidad[h])
160       return true;
161   }
162   return false;
163 }
164
165 // check:
166 // si retorna 0 no es sol valida
167 // si retorna 1, es valida pero falta agregar camiones
168 // si retorna 3 la solucion anterior usa menos camiones
169 // si retorna 2 es solucion valida
170 int check(tablaDePeligrosidad *tab, vector<int> &solParcialCamiones ,vector <int
      > &solFinalCamiones)
```

```
171 {
172
173   //mi resultado final sigue siendo mejor
174   if(solucionFinalUsaMenosCamiones(solParcialCamiones,solFinalCamiones))
175     return 3;
176
177   //me pase, abort!
178   if(cotaDePeligrosidadSobrepasada(tab,solParcialCamiones))
179     return 0;
180
181   //
182   if(tab->n == solParcialCamiones.size())
183     return 2;
184
185
186   return 1;
187 }
188
189 void imprimirResultado(vector<int> solParcialCamiones)
190 {
191  int max = solParcialCamiones[0];
192  for(int i = 1; i < solParcialCamiones.size(); i++)
193    if(solParcialCamiones[i] > max)
194      max = solParcialCamiones[i];
195  cout << max + 1<< " ";
196  for(int i = 0; i < solParcialCamiones.size(); i++)
197    cout << solParcialCamiones[i] + 1 << " ";
198  cout << endl;
199 }
200
201
202
203 void inicializarPeorSol(vector<int> &sol,int n)
204 {
205   sol.clear();
206   for(int i = 0; i < n; i++)
207       sol.push_back(i);
208 }
209
210 int main()
211 {
212   timeval tm1, tm2;
213   gettimeofday(&tm1, NULL);
214   while(true){
215     tablaDePeligrosidad tab = InicializarTablaDePeligrosidad();
216     vector<int> solParcialCamiones;
217     vector<int> solFinalCamiones;
218
219     inicializarPeorSol(solFinalCamiones,tab.n);
220
221     //Agrego el primer producto al camion y lanzo la recursion
222     solParcialCamiones.push_back(0);
223     bool sol = backtracking(&tab, solParcialCamiones,solFinalCamiones);
224     imprimirResultado(solFinalCamiones);
225     solParcialCamiones.clear();
226     solFinalCamiones.clear();
227     borrarTablaDePeligrosidad(&tab);
228   }
229
230   return 0;
231 }
```