DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Técnicas algorítmicas

5 de septiembre de 2014

Algoritmos y Estructuras de Datos III
Trabajo Práctico Nro. 1

Integrante	LU	Correo electrónico
Pablo Gomez	156/13	mago-1986@hotmail.com
Lucia Parral	162/13	luciaparral@gmail.com
Emanuel Lamela	21/13	emanuel93_13@hotmail.com
Petr Romachov	412/13	promachov@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Ejercicio 1

1.1. Introducción

Contexto

Estamos encarando una competencia, en la que dados distintos puentes colgantes y participantes, estos deben cruzar los puentes sin caerse, dado que bajo cada puente corre un río de lava hirviendo. Estos puentes tienen una característica importante, y es que no todos sus tablones están en óptimas condiciones, estando algunos rotos, y haciendo que si alguien los pisa, caiga sin remedio hasta el río de lava que fluye debajo. Esta es la única forma de caer del puente. Afortunadamente, estos tablones rotos se encuentran marcados, de forma que quien este cruzando el puente, sepa si puede pisar o no un tablón. Un requisito de la competencia es que los puentes solo pueden ser cruzados dando saltos, tanto para ingresar o salir del puente como para avanzar de tablón en tablón. En cuanto a los participantes, debido a su estado físico, cada uno tiene una capacidad máxima de tablones que pueden cruzar de un solo salto. Notar que si hay demasiados tablones rotos consecutivos, si un participante no tiene la capacidad suficiente de salto para cruzarlos a todos juntos, entonces ese participante no podrá cruzar el puente ya que indefectiblemente caerá si lo intenta. El ganador de la competencia será quien haya cruzado todos los puentes, dando la menor cantidad de saltos.

El problema a resolver

Debemos diseñar un algoritmo, que dado un participante, la capacidad máxima de tablones que puede cruzar de un solo salto (salto máximo), un puente, con un número n de tablones del mismo y un mapeo de los tablones rotos, indique si es posible cruzar el puente y de ser así, indique a qué tablones debería saltar para cruzarlo dando la menor cantidad de saltos posibles. El algoritmo deberá tener complejidad O(n), siendo n la cantidad total de tablones (rotos y no rotos) del puente colgante. En el caso en el que el participante no pueda cruzar el puente, debe devolverse la palabra "no". Así mismo, el programa debe soportar resolver muchas instancias de este tipo ingresadas.

Ejemplos

Para los ejemplos denotaremos:

C, a la cantidad de tablones que el participante en cuestion puede cruzar de un solo salto.
N, a la cantidad de tablones (rotos y no rotos) del puente en cuestion, donde a cada tablon los enumeraremos entre 1 y N.

1. C = 4, N = 8, donde los tablones rotos son: 1,3,6,8.

   El algoritmo debería devolver: 3 4 7 9

2. C = 4, N = 17, donde los tablones rotos son el 2,3,4,5.

   El algoritmo debería devolver: no, puesto que una vez en el tablón 1, el participante no puede pisar ningún tablón que su capacidad C le permite pisar.

3. C = 5, N = 41, donde no hay tablones rotos.

   El algoritmo debería devolver: 9 5 10 15 29 25 39 35 40 42

1.2. Desarrollo

Como idea principal para la resolución del problema, partiremos de la siguiente propiedad del problema: para cualquier tablón n_0 en que se esté situado en un momento dado, hacer un salto de distancia **X**, es siempre mejor que un salto de distancia **J**, para todo **J<X**. Este razonamiento 'goloso' es válido ya que vale que luego de saltar hasta x, se puede saltar a todos los que se podría haber saltado desde j, y posiblemente, más aún.

Es decir, si notamos CS[i], donde 0<i≤n, (con n = cantidad de tablones del puente), como la conveniencia de salto para un tablón i, vale que:

(∀ j,x:nat, j<x) CS[j] ≤ CS[x]

Por lo dicho anteriormente, concluimos (y en el siguiente inciso, demostraremos) que saltar a la posición más lejana posible dentro del rango de salto k (propio del jugador), nunca sera peor que hacerlo a una posición anterior a ésta y más aún, será la óptima decisión.

Con el siguiente pseudocódigo explicamos las bases principales de nuestro algoritmo:

Sea **S** el salto máximo posible.
Sea **C** la cantidad de tablones del puente pasado como parámetro (tablones válidos y no válidos).
Sea **saltos** el vector en el que se almacenarán las posiciones a las que se salte. Cuando se declara a este vector, se le reserva un tamaño **C**, ya que a lo sumo pueden realizarse **C** saltos (uno por tablón).
Sea **actual** última posición válida a la que se saltó.
Sea **tablon_en_revision** el tablón que estoy revisando en el ciclo.

```
1   Si (S > C) // Caso 1
2     agrego C+1 a Saltos
3     encontre solucion y salgo
4   Si no // Caso 2
5     si (S < 1)
6       no hay solucion y salgo
7   Si no // Caso 3
8     para (i entre 0 y C)
9       actualizo el tablon_en_revision
10      si (i es posicion valida)
11        actualizo salto_mas_largo_posible
12      si (tablon_en_revision - actual == S)
13        actualizo actual
14        agrego actual a saltos
15      si (actual + S > C)
16        agrego C+1 a saltos
17        encontre solucion y salgo
```

En el condicional de la línea 1, se chequea si el rango de salto S pasado como parámetro supera a la cantidad de tablones, es decir, si con un solo salto se puede recorrer todo el puente. Si ese es el caso, se sale del algoritmo.
En el condicional de la línea 2, se chequea si no hay rango de salto S, en ese caso, el problema no tiene solución.
En el último condicional (línea 7), se entra a un ciclo en el que se van a recorrer del 0 al C (sin incluir) todos los tablones, llevando un registro del tablón que se esta revisando en la i-ésima iteración (lo llamamos tablon_en_revision), así como también, cuál es la última posición válida hacia la que salté (en el pseudocódigo la llamaremos actual). Si tablon_en_revision es válido, se lo guarda como potencial candidato a salto, dado que, hasta el momento, es el salto más largo posible. Luego, se chequea si, dada la posición actual y el tablon en revisión, ya chequee todos los tablones a los que potencialmente podía acceder desde actual (línea 12), sí es así, debo guardar como salto realizado al potencial candidato a salto (en el pseudocódigo lo llamamos salto_mas_largo_posible) en el vector en el que almaceno los saltos y luego, actualizar como actual a esta posición, ya que se convertirá en la última posición válida hacia la que salté. Por último, verificamos si con el próximo salto desde la posición actual se llegará a cruzar el resto del puente, es decir, si al dar mi próximo salto encontré la salida (línea 15). Si ese es el caso, agrego la posición de salida (la C+1) y salgo del ciclo.
Finalmente, devolvemos el vector de saltos y su tamaño como solución del problema, en el formato pedido.
Vale la pena mencionar que repetiremos este algoritmo una vez por cada instancia pasada como entrada.

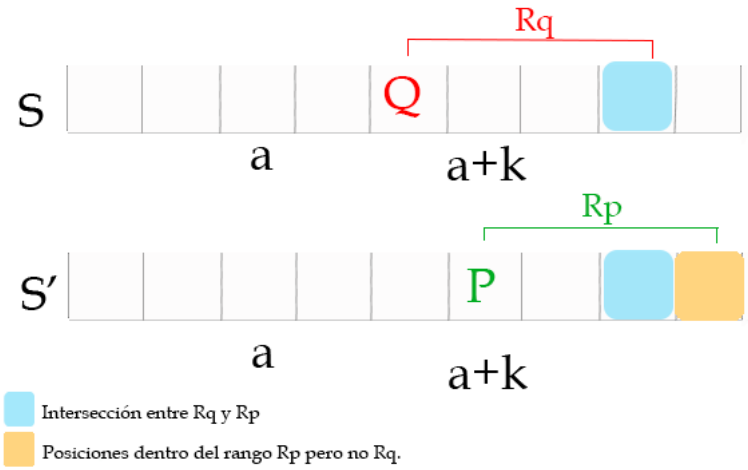
1.3. Correctitud

Sea **P** un puente pasado como parámetro y **C** la cantidad de tablones del mismo, describiremos **Z**, una solución del problema, como el arreglo que contiene un máximo C de elementos x donde $\forall$ x $\in$ Z, P[x] = 0, es decir, es una posición válida a la que el participante saltó (también nos referiremos a estas posiciones como a los "saltos"). Demostraremos la siguiente propiedad, llamémosla M:

M = para cualquier **a** tablón válido de un puente P, con a $\in \{1 \dots n\}$ y un salto máximo k que un jugador puede realizar, efectuar el salto desde a hacia p = max{a ... a+k} (siendo p una posición válida del puente) genera una menor o igual cantidad de saltos en Z que cualquier otra posición distinta de p elegida.

Esto es equivalente a afirmar que cumpliendo la propiedad **M**, una solución es óptima para el problema planteado, es decir, |M(Z)| $\leq$ |Z|. Esto es: la longitud de Z si para Z vale la propiedad M **es menor o igual** a la longitud de Z en otras condiciones. (Aclaración: por longitud nos referimos a la cantidad de saltos realizados)

Supongamos una solución **S** óptima, donde hay algún **a** para el cual no vale **M**, es decir, se escoge una opción de salto **Q** $\neq$ max{a ... a+k}, donde **Q** es una posición válida.
Consideremos entonces ahora a **S'**, una solución para la misma instancia que posee hasta la posición **a** los mismos saltos, pero que en **a**, en vez de realizar un salto a **Q** se realizó un salto a la posición válida **P** = max{a ... a+k}.
Llamaremos R_p al rango de posiciones válidas accesibles desde **P** $\in$ {P ... P+k} y R_q al **S** al rango de posiciones válidas accesibles desde **Q** $\in$ {Q ... Q+k}.
Sabemos que $R_p \cap R_q \neq \emptyset$ ya que por lo menos exite **P** que es posición válida que es accesible desde R_q y pertenece a R_p, ya que **Q** < **P** $\leq$ **Q+k** $\leq$ **P+k**.



Entonces pueden darse los siguientes casos:

(a) Existe al menos un **x** tal que **x** no pertenece a $R_p \cap R_q$ y x $\in R_p$, es decir, existe una posición que es accesible desde **P** pero no desde **Q**. Acceder a esta posición solo puede provocar que se generen en total menos o igual cantidad de saltos que en **S**, ya que, en caso de ser necesario acceder a estas posiciones, desde **S** se debería haber realizado un salto extra, en cambio en **S'**, estas posiciones son accesibles en el primer salto **P**. En el caso de que no sea necesario accederlas, se mantendría la misma cantidad de saltos que en **S**, por lo que **S'** sería igualmente óptima.

(b)En el caso de que no exista **x** tal que **x** no pertenece a $R_p \cap R_q$ y **x** $\in R_p$, R_p contiene las posiciones válidas entre {P ... Q+k}, por lo que como teníamos **Q** < **P** $\leq$ **Q+k**, se sigue que R_p esta incluído en R_q, entonces obtendremos la misma cantidad de saltos en **S'** que en **S**, ya que en **S**, desde **P** el próximo salto válido será **Q+k**, que era el próximo salto válido desde **Q** en **S**. De esta forma, mantengo igualada la cantidad de saltos: de **a** a **Q** y de **Q** a **Q+k** en **S** y de **a** a **P** y de **P** a **Q+k** en **S'**.

Quedan así verificados todos los casos, demostrándose que una solución que cumple con la propiedad **M** es óptima.

1.4. Complejidad

Sea **S** el salto máximo posible.
Sea **C** la cantidad de tablones del puente pasado como parámetro (tablones válidos y no válidos).
Sea **saltos** el vector en el que se almacenarán las posiciones a las que se salte. Cuando se declara a este vector, se le reserva un tamaño **C**, ya que a lo sumo pueden realizarse **C** saltos (uno por tablón).
Sea **actual** última posición válida a la que se saltó.
Sea **tablon_en_revision** el tablón que estoy revisando en el ciclo.

```
1   Si (S > C) // Caso 1
2     agrego C+1 a Saltos
3     encontre solucion y salgo
4   Si no // Caso 2
5     si (S < 1)
6       no hay solucion y salgo
7   Si no // Caso 3
8     para (i entre 0 y C)
9       actualizo el tablon_en_revision
10      si (i es posicion valida)
11        actualizo salto_mas_largo_posible
12      si (tablon_en_revision - actual == S)
13        actualizo actual
14        agrego actual a saltos
15      si (actual + S > C)
16        agrego C+1 a saltos
17        encontre solucion y salgo
```

Dependiendo de los parámetros de entrada S y C se entrará por el Caso 1, Caso 2 o Caso 3 (notemos que entrar se puede entrar por un y solo un caso).
Tanto el Caso 1 como el 2 se basan en operaciones elementales que se ejecutan una cantidad constante de veces, por lo tanto, en ambos casos la complejidad es O(1).
En el Caso 3 tenemos como operación principal un ciclo for que se ejecuta a lo sumo C veces (puede llegar a ejecutarse menos de C veces si en alguna de las iteraciones, por el condicional de la línea 15, se sale del ciclo).
Luego, tenemos todas operaciones elementales, sea cual sea el caso de que valga cualquier combinación posible de los condicionales de las líneas 10, 12 o 15.
Finalmente, el Caso 3 tiene complejidad (en peor caso) de C operaciones elementales, es decir, C*O(1). Entonces, la complejidad temporal del Caso 3 es O(C).
Podemos ver entonces que la complejidad temporal del algoritmo termina siendo:

- O(1) si se entra por el Caso 1 o 2.
- O(C) si se entra por el Caso 3.

Por lo anterior, en peor caso, la complejidad temporal del algoritmo propuesto es O(C).

1.5. Experimentación

Para el proceso de experimentación del problema se plantearon distintos escenarios de test para corroborar que el algoritmo propuesto funcionara correctamente y que la cota de complejidad encontrada y justificada en la sección anterior, en la práctica, se cumpliera.

Llamamos escenario de test a un conjunto de pruebas que si bien son distintas, comparten alguna similitud.

Por ejemplo, un escenario es aquel en el cual un participante puede dar saltos de una distancia k constante, y las distintas pruebas del escenario variarían en el tamaño del puente y en el estado de sus tablones.

Dado que el CPU de la computadora utilizada para tomar los tiempos no está atendiendo únicamente a nuestro proceso, realizar una sola vez cada prueba podría darnos valores que no son cercanos a los reales. Para minimizar este margen de error, a cada prueba de cada escenario se la hizo ejecutar un total de 10.000 veces y se tomó el mejor valor. Notar que tomar el mejor valor no es una mala decisión, ya que mientras más chico sea el valor, más cerca estamos del valor real de tiempo que toma el algoritmo para una instancia dada.

En cada prueba se tomaron métricas para la posterior evaluación del algoritmo en la práctica. Notar que la medición no contempla tiempos de entrada/salida de datos, sino que contempla solamente el núcleo del algoritmo.

Para cada escenario testeado, se hicieron gráficos 2D que permitan ver de una manera más clara los resultados obtenidos en las pruebas del mismo. Estos fueron realizados con el software QitPlot que la cátedra proveyó.

En cuanto a qué casos testear, decidimos testear los casos “border” y casos aleatorios, mezclando en todos los casos distintas variaciones de saltos posibles, tamaño de los puentes y estado de los tablones de los mismos.

Los casos ”border”, son aquellos que están en los extremos de las capacidades del algoritmo, es decir, el mejor caso que el algoritmo puede resolver, y el peor. Para las instancias aleatorias, se diseñó un generador de estas, que dada una longitud, el estado de generaría los tablones se haría de forma aleatoria, es decir, dado un tablón t, en una prueba t puede ser válido y en otra no. Este generador es capaz de generar múltiples instancias aleatorias. Para todos los casos, se eligió una precisión de hasta 0,0001 ms (milisegundos). De ser menor, la notamos como 0.

En todos los casos se pudo comprobar que la práctica refleja lo expuesto en incisos anteriores.

1.5.1. Escenario de mejor caso del algoritmo

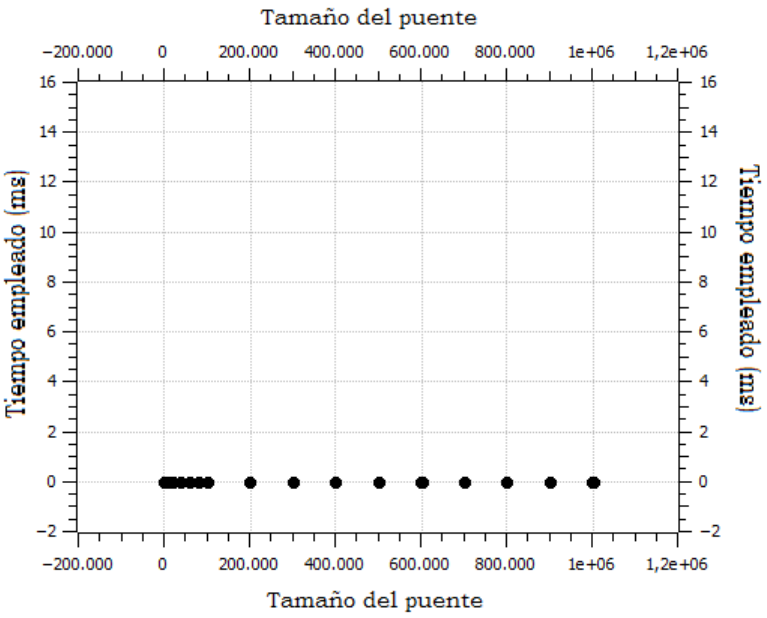
Para el algoritmo propuesto, el mejor caso de respuesta es el caso en el que la capacidad de salto del participante es superior a la longitud del puente, este caso tiene una complejidad de orden constante, es decir, el mejor caso es O(1).

Notar que para probar el mejor caso, la capacidad de salto no puede ser un parámetro variable, ya que es necesario que sea mayor que la cantidad de tablones.

Para evaluar el mejor caso, el generador de instancias, aplica como salto máximo, el tamaño del puente + 1. Por ende, para cada prueba realizada, si el puente contenía n tablones, el salto era de n+1. Dicho esto, se generaron distintas instancias del algoritmo, con distintos tamaños, y con puentes de estado aleatorios, para verificar que el mejor caso no del estado de sus tablones, sino de que el salto sea mayor estricto que la cantidad de tablones.

Luego de realizar las pruebas pertinentes de este escenario, (y de realizar 10.000 veces cada una) como declaramos al inicio del insiso, mostramos los resultados obtenidos en el siguiente gráfico:

Tablones	milisegundos
1.000.000	0,0003
900.000	0,0002
800.000	0,0003
700.000	0,0004
600.000	0,0002
500.000	0,0005
400.000	0,0002
300.000	0,0002
200.000	0,0003
100.000	0,0004
80.000	0,0003
60.000	0,0003
40.000	0,0004
20.000	0,0003
10.000	0,0003
1.000	0,0002
800	0,0003
400	0,0003
200	0,0003
100	0,0002
10	0,0002



En la figura superior se puede apreciar que el tiempo de resolución es constante a pesar de variar el tamaño de entrada y el estado de los tablones (generados aleatoriamente).

1.5.2. Escenario de peor caso del algoritmo

Para el algoritmo propuesto, el peor caso de respuesta, es decir, el caso que más tiempo demanda ejecutar, es aquel en el que el participante puede cruzar todo el puente (por lo que el algoritmo debe continuar hasta el final del mismo) y dando saltos mínimos de avance, es decir, saltos de longitud 1.

Este caso, le toma al algoritmo recorrer todo el puente, por lo que, si n es la cantidad de tablones, el algoritmo es O(n).

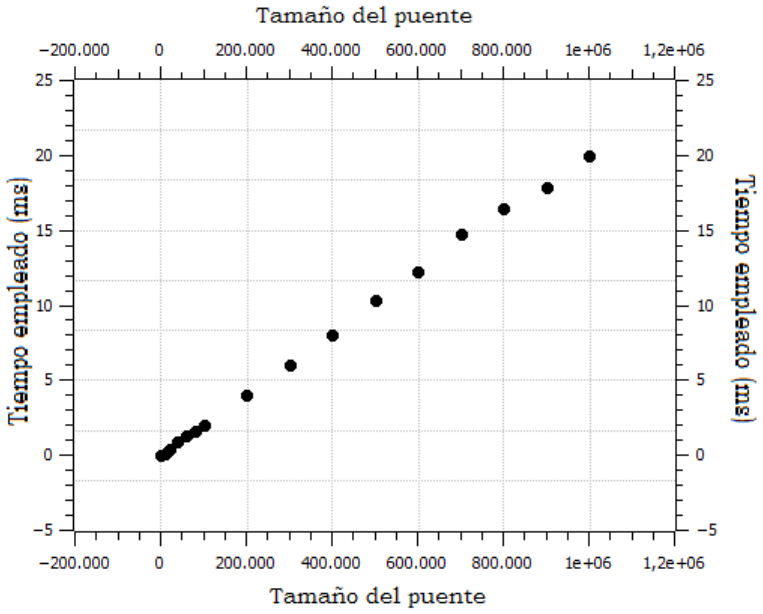
Notar que en este caso tampoco tiene sentido variar la longitud del salto ya que si no, no estaríamos evaluando el peor caso.

A diferencia del mejor caso, aquí tampoco podemos variar el estado de los tablones, ya que estos deben ser todos válidos, para que, dando saltos de 1 de longitud, el participante pueda cruzar todo el puente, forzando al algoritmo a su peor caso.

Por ende, el único parámetro a variar para realizar pruebas, es la longitud del puente.

Luego de realizar las pruebas pertinentes de este escenario, (y de realizar 10.000 veces cada una) como declaramos al inicio del insiso, mostramos los resultados obtenidos en el siguiente gráfico:

Tablones	milisegundos
1.000.000	20
900.000	17,9
800.000	16,5
700.000	14,8
600.000	12,3
500.000	10,4
400.000	8
300.000	6
200.000	4
100.000	2
80.000	1,6
60.000	1,3
40.000	0,9
20.000	0,4
10.000	0,1
1.000	0,009
800	0
600	0
400	0
200	0
100	0
10	0



En la figura superior se puede apreciar que el tiempo de resolución de peor caso del algoritmo es lineal al tamaño del puente de entrada.

1.5.3. Escenarios de casos promedio del algoritmo

En estos escenarios, decidimos evaluar y tomar métricas para casos no tan "border" como los de los escenarios anteriores. La idea es tomar muestras del algoritmo haciendo variar la longitud de salto del participante como el tamaño del puente, generando de manera aleatoria el estado de los tablones del mismo.

Para diferenciar bien los casos y poder analizar mejor, decidimos que cada escenario de las pruebas de caso promedio tenga una capacidad de salto constante. En cada uno de estos escenarios veremos cómo responde el algoritmo a medida que varía el tamaño de la entrada.

Notar que a diferencia del mejor y del peor caso, aquí no está garantizado que el participante vaya a cruzar todo el puente. En los casos anteriores, cruzar el puente era requerido para poder contemplar el caso de análisis, tanto para el mejor como para el peor caso. Para poder ver en los gráficos, no solo el tiempo de respuesta del algoritmo en función de la entrada, sino también si el participante pudo o no cruzar el puente, se marcó de color verde a las instancias en las que sí pudo, y de color rojo a las que no.

Como era de esperar, como el estado de cada tablón es aleatorio, mientras más pequeño es el salto del participante, mayor es la probabilidad de que este no pueda cruzar el puente, ya que la probabilidad de que haya tantos tablones inválidos consecutivos como la capacidad de salto, aumenta.

Dicha probabilidad también aumenta cuando el puente crece, ya que hay un mayor espacio para que dicha secuencia de tablones inválidos aparezca.

Como fue mencionado anteriormente, para realizar estas pruebas, se diseñó un generador de instancias de caso promedio, que dado un tamaño de entrada y una capacidad de salto, generaba tantas instancias distintas como se quisiera, respetando el tamaño asignado. Eso nos permitió evaluar la respuesta del algoritmo para distintas instancias del mismo tamaño, y poder tomar un promedio.

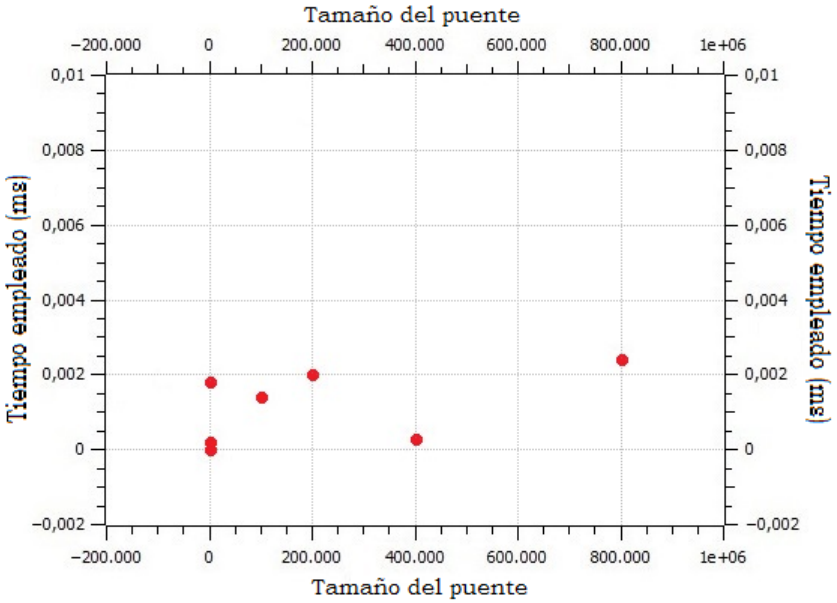
Para estos casos, también cada prueba de cada escenario fue repetida un total de 10.000 veces para aminorizar el margen de error producido debido a que el CPU no está atendiendo únicamente nuestro proceso.

Como última observación a hacer, es interesante notar que para los casos en los que el participante no pueda cruzar todo el puente, los tiempos no siguen ningún patrón ni concordancia, ya que, si llamamos C a la capacidad de salto de un participante, el hecho de que haya C tablones rotos consecutivos es totalmente aleatorio ya que el estado de los tablones fue generado de manera aleatoria.

Sin más, presentamos los distintos escenarios.

ESCENARIO DE SALTO = 2

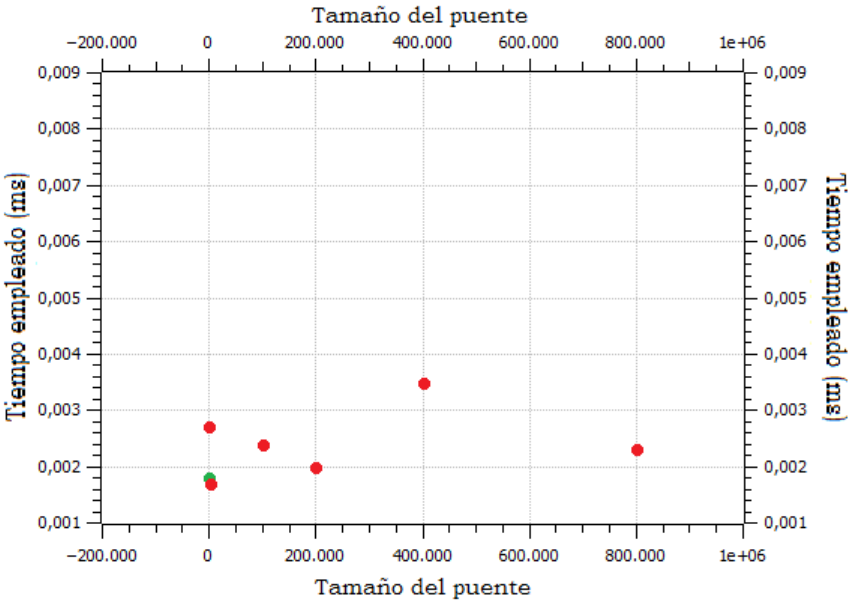
Tablones	milisegundos
100	0,0018
1.000	0,0002
100.000	0,0014
200.000	0,002
400.000	0,0003
800.000	0,0024



Tal como lo expresado anteriormente, para saltos de baja longitud, la probabilidad de no poder cruzar el puente aumenta, por lo que los tiempos del algoritmo pueden variar mucho de instancia a instancia.

ESCENARIO DE SALTO = 4

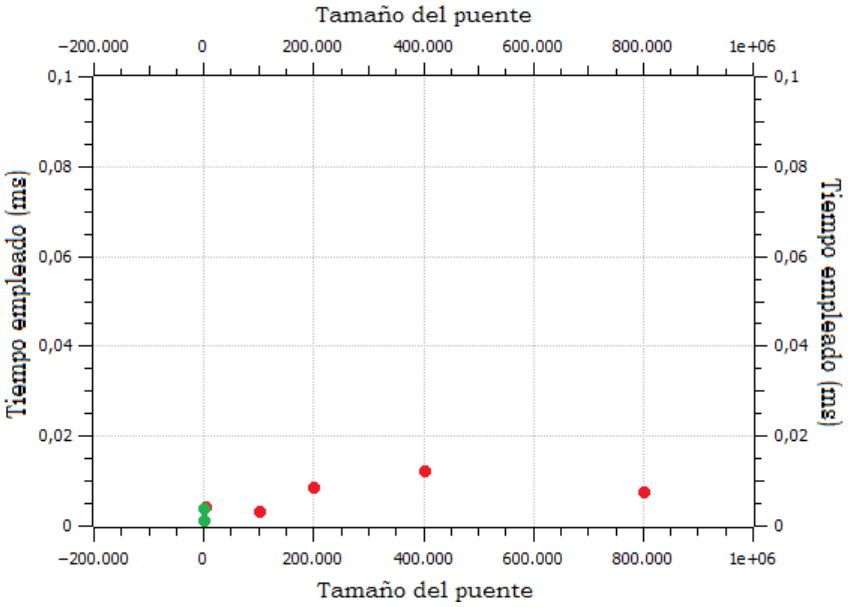
Tablones	milisegundos
10	0,0018
100	0,0027
1.000	0,0017
100.000	0,0024
200.000	0,002
400.000	0,0035
800.000	0,0023



Aquí se puede apreciar que una de las instancias generadas permitió al participante cruzar todo el puente, y que fue la instancia en la que más probabilidades tenía de hacerlo, es decir, la instancia con un puente de longitud = 10. Para el resto, continuaron apareciendo de manera aleatoria distintos baches en el puente que no permitieron al jugador terminar de cruzar el mismo. Notar que los baches se hicieron presentes al poco tiempo de recorrer el puente, tal como era esperado probabilísticamente.

ESCENARIO DE SALTO = 8

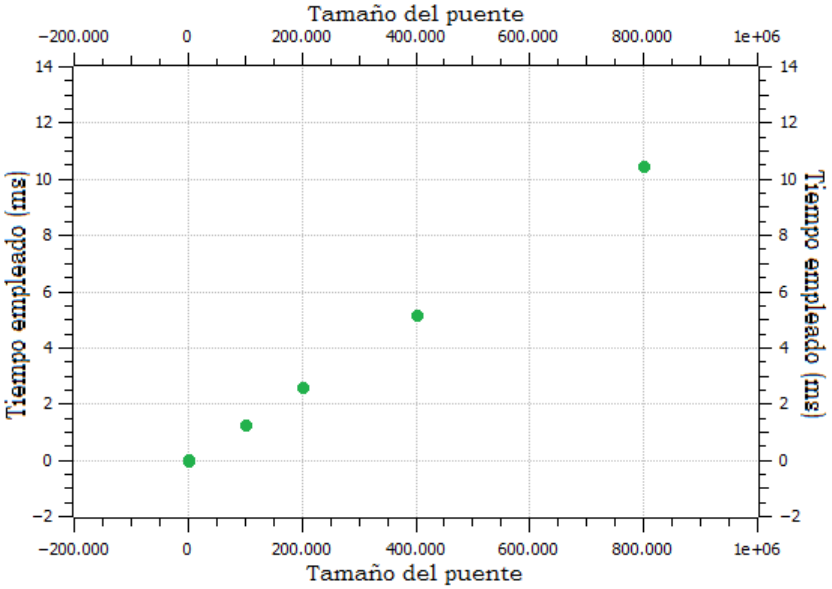
Tablones	milisegundos
10	0,0015
100	0,0041
1.000	0,0044
100.000	0,0034
200.000	0,0086
400.000	0,0125
800.000	0,0078



En saltos de tamaño 8, se continuó observando que la probabilidad de poder cruzar el puente en altas instancias continúa siendo muy baja y el algoritmo termina rápidamente debido a eso. Aunque como cada instancia no depende de las demás, no podemos acotar al tiempo del algoritmo, pero si podemos hacerlo, justificando que cierta cota es válida para el caso promedio, para saltos menores o iguales a 8.

ESCENARIO DE SALTO = 16

Tablones	milisegundos
10	0,0006
100	0,0022
1.000	0,0135
100.000	1,2844
200.000	2,5969
400.000	5,1812
800.000	10,4617

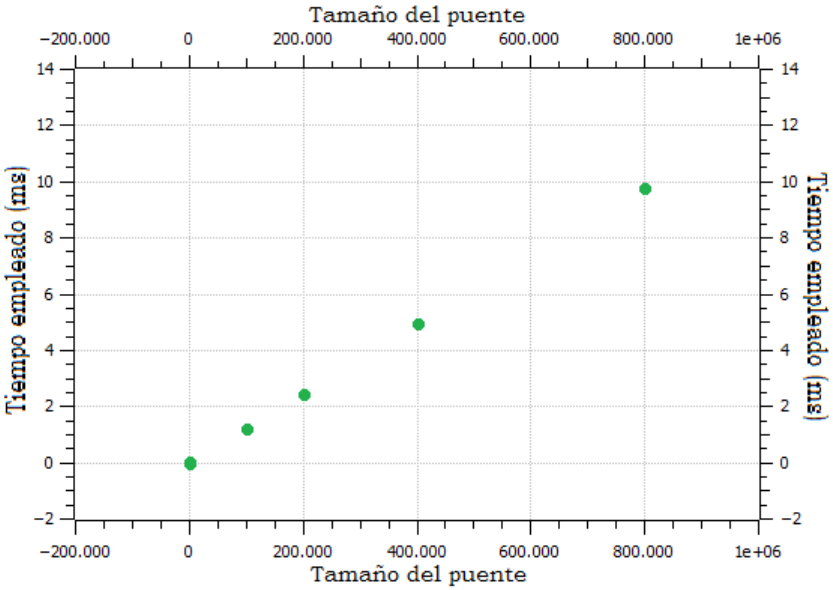


A partir de aquí, notamos que con las mismas longitudes de prueba que en los escenarios anteriores, no se produjo una caída del puente, en ninguna de las distintas instancias que se probaron para cada uno de los tamaños (10, 100,. . . ,800.000). Vale destacar, que con un salto de longitud = 16, se cae en el mejor caso, para la instancia de 10 tablones, ya que puede cruzarla toda de una, haciendo que la probabilidad de caer del puente pase a ser nula. Lo mismo ocurrirá con luego con saltos mayores, al superar a la instancia de 100 tablones.

Si bien en todas las pruebas de este escenario el participante pudo cruzar, eso no escapa de la probabilidad de que la generación aleatoria pueda dar 16 tablones rotos consecutivos. Por lo que no podemos dar una cota fija, pero podríamos estimar que en el caso promedio, el puente puede ser cruzado, lo que da una complejidad de O(n) para el caso promedio, con n igual a la cantidad de tablones. Es O(n) ya que el algoritmo debe recorrer todo el puente para terminar.

ESCENARIO DE SALTO = 32

Tablones	milisegundos
10	0,001
100	0,0025
1.000	0,0135
100.000	1,2237
200.000	2,4663
400.000	4,9551
800.000	9,7912

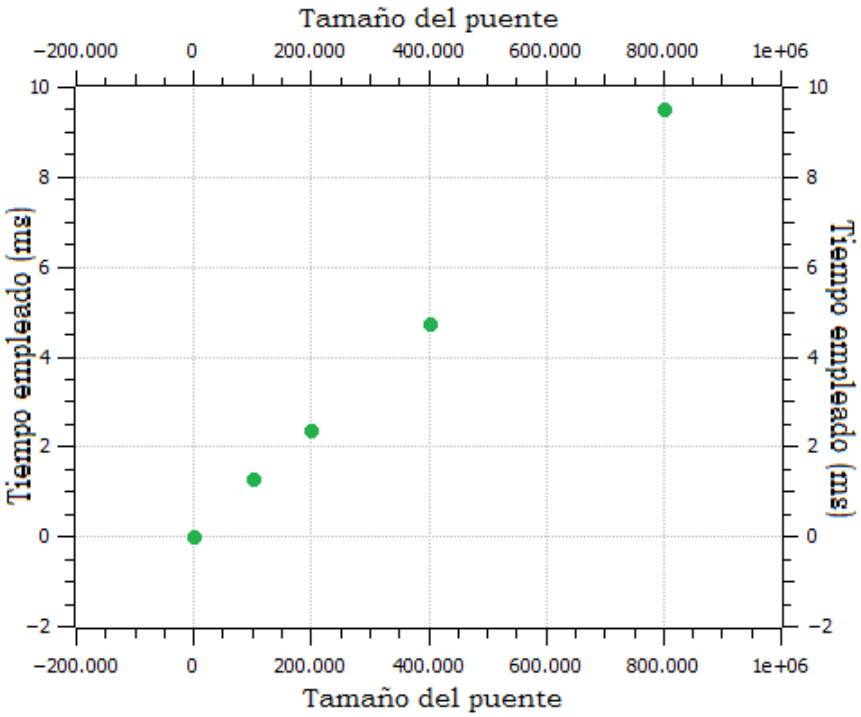


Aquí y en adelante, la probabilidad de que haya una cantidad de tablones rotos consecutivos igual a la cantidad del salto, es realmente baja, más allá del tamaño de la entrada; tendría que ser una entrada exageradamente grande para que la probabilidad aumente apenas un poco. Por ende de aquí en adelante, afirmamos que el caso promedio del algoritmo es O(n), (recordando que n es la cantidad de tablones del puente en cuestión).

Por lo dicho recién, en los siguientes escenarios era esperado que la respuesta del algoritmo sea líneal al tamaño de la entrada.

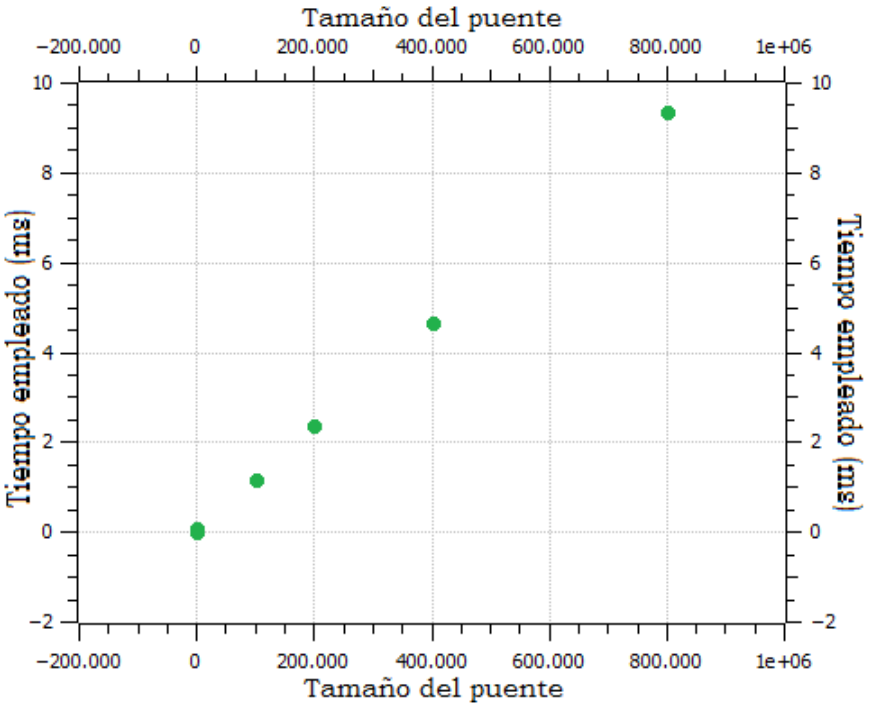
ESCENARIO DE SALTO = 64

Tablones	milisegundos
10	0,0007
100	0,0017
1.000	0,0122
100.000	1,1776
200.000	2,3938
400.000	4,7486
800.000	9,5021



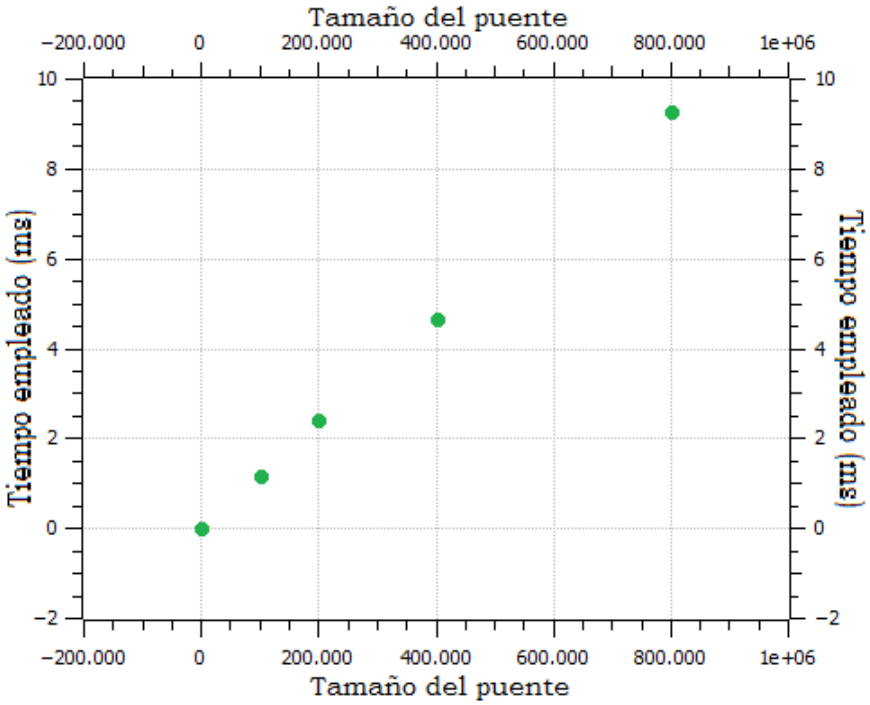
ESCENARIO DE SALTO = 128

Tablones	milisegundos
10	0,0006
100	0,0009
1.000	0,101
100.000	1,1593
200.000	2,3844
400.000	4,6654
800.000	9,3601



ESCENARIO DE SALTO = 256

Tablones	milisegundos
10	0,0008
100	0,0005
1.000	0,0083
100.000	1,1509
200.000	2,3955
400.000	4,6588
800.000	9,2855



1.5.4. Algunas conclusiones

Haciendo un análisis probabilístico de nuestro algoritmo, por lo antes explicado podemos afirmar que en saltos grandes, el tiempo de computo de una instancia es lineal al tamaño del puente de la misma. En cuanto a saltos pequeños no hay un patrón que se mantenga ya que el "hueco" sin tablones válidos por los cuales no se podría cruzar es totalmente al azar, pero probabilísticamente, si el salto es chico, dicho "hueco" se hará presente al poco tiempo de computo, por lo que el algoritmo responde eficientemente y podríamos acotar este tiempo para un caso promedio.

2. Ejercicio 2

2.1. Introducción

Contexto

En este apartado estamos encarando el diseño de un software de arquitectura, y como parte de este diseño estamos trabajando más específicamente sobre el módulo de Edificios 2D. Dicho módulo representa a todos los edificios de una ciudad en 2 dimensiones, es decir, en un plano bidimensional. En el plano los edificios son representados por rectángulos apoyados sobre una misma base común. El desafío del proyecto consiste en eliminar las líneas ocultas, dibujando únicamente el horizonte (contorno) que los edificios forman en dicho plano bidimensional.

El problema a resolver

Debemos diseñar un algoritmo que, dado n la cantidad de edificios de un plano, y dados para los n edificios su parante izquierdo, altura y parante derecho, calcule y devuelva el perfil de los n edificios en el horizonte, es decir, el contorno que estos forman en conjunto. Este contorno debera ser representado a partir de la coordenada en el eje horizontal X de cada punto en donde se produce un cambio de altura, junto con dicha altura. El problema deberá resolverse en una complejidad temporal estrictamente mejor que $O(n^2)$, y deberá poder resolver muchas instancias de planos ingresadas.

Ejemplos

Para los ejemplos denotaremos:

N a la cantidad de edificios del plano bidimensional.
IZQi a la coordenada x del parante izquierdo del iésimo edificio.
ALTi a la altura del iésimo edificio.
DERi a la coordenada x del parante derecho del iésimo edificio.

1. N = 4

   IZQ1 = 2 ALT1 = 5 DER1 = 6
   IZQ2 = 3 ALT2 = 2 DER2 = 4
   IZQ3 = 3 ALT3 = 6 DER3 = 4
   IZQ4 = 1 ALT4 = 4 DER4 = 5

   Contorno resultado: 1 4 2 5 3 6 4 5 6 0

2. N = 5

   IZQ1 = 8 ALT1 = 4 DER1 = 9
   IZQ2 = 6 ALT2 = 4 DER2 = 7
   IZQ3 = 1 ALT3 = 1 DER2 = 10
   IZQ3 = 4 ALT3 = 2 DER3 = 8
   IZQ4 = 2 ALT4 = 5 DER4 = 3

   Contorno resultado: 1 1 2 5 3 1 4 2 6 4 7 2 8 4 9 1 10 0

3. N = 5

   IZQ1 = 1 ALT1 = 2 DER1 = 3
   IZQ2 = 2 ALT2 = 2 DER2 = 4
   IZQ3 = 3 ALT3 = 2 DER2 = 5
   IZQ3 = 4 ALT3 = 2 DER3 = 6
   IZQ4 = 2 ALT4 = 2 DER4 = 5

   Contorno resultado: 1 2 6 0

2.2. Desarrollo

Para resolver este problema se analizo que los edificios se podrian llegar a tratar como horizontes individuales, ya que en el caso mas trivial, un edificio es un horizonte. Para mayor legibilidad, en el codigo y pseudocodigo desarrollados, se definieron los siguientes tipos, ademas del tipo ya conocido de edificios:

- Horizontes es un vector de Horizonte
- Horizonte es un vector de Coordenada
- Coordenada es una tupla(x, y) en un plano

Transformacion de Edificios a Horizontes

Entonces en primera instancia se transforman los edificios en horizontes con sus coordenadas respectivas: siendo (x1, y, x2) los valores que representan un edificio, realiza una transformacion a dos coordenadas (x1, y)(x2, 0), notar que estas coordenadas siempre estan ordenadas de menor a mayor respecto del eje horizontal X.

```
1 Horizontes EdificiosAHorizontes(edificios)
2   Para cada edificio i en edificios
3     horizontes[i][0].x <- edificio.x1
4     horizontes[i][0].y <- edificio.y
5     horizontes[i][1].x <- edificio.x2
6     horizontes[i][1].y <- 0
7   return horizontes
8 Fin
```

Una vez hecho esto, el problema radicaria en como unir los horizontes. Esto se desglosa en dos sub problemas a resolver:

1. Como unir dos horizontes.
2. Sabiendo el anterior, que tecnica usar para unir varios de ellos.

Tecnica para unir varios horizontes

Enfocar al problema desde este punto de vista es pensarlo desde una perspectiva **Divide & Conquer**, donde los casos base son: hay 1 horizonte, donde se devuelve el mismo, y hay 2 horizontes, en el cual se unen los dos. Para mas de dos horizontes se llama recursivamente con una mitad y la otra, y se los une:

```
1  Horizonte UnirHorizontes(horizontes, inicio, fin)
2    Si fin - inicio == 0
3      //Hay un horizonte en horizontes
4      return horizontes[inicio]
5    Si no
6    Si fin - inicio == 1
7      //Hay dos horizonte en horizontes
8      return Unir(horizontes[inicio], horizontes[fin])
9    Si no
10   Si fin - inicio > 1
11     //Hay mas de dos horizonte en horizontes
12     mitad <- [(fin-inicio)/2] + inicio
13     return Unir(UnirHorizontes(horizontes, inicio, mitad), UnirHorizontes(
           horizontes, mitad+1, fin))
14 Fin
```

Tecnica para unir dos horizontes

Ahora queda definir la estrategia para unir dos horizontes. La clave en este punto es tomar la aparicion de cada coordenada como una variacion en la altura, y al momento de analizar que hacer con ella, solamente tener que revisar la relacion con la ultima coordenada analizada del horizonte opuesto. Dicha relacion puede ser mayor, menor o igual en altura.

Se itera este este procedimiento con cada coordenada de ambos horizontes, siempre eligiendo la coordenada que aparece primero en el eje X para el analisis, y **manteniendo un invariante** en el que **las coordenadas en el resultado son el contorno parcial del horizonte final hasta ese momento**.

Una vez que termino de recorrer todo un horizonte, se agregan a continuacion todas las del otro. Esto es debido a que si termino de recorrer uno, significa que la ultima de sus coordenadas aparecia antes que las del contrario, por lo tanto estas ultimas van a formar parte del horizonte resultado.

```
1  Horizonte Unir(h1, h2)
2    Horizonte resultado
3    Coordenada ulth1 <- ultima coordenada analizada de h1
4    Coordenada ulth2 <- ultima coordenada analizada de h2
5    i = 0, j = 0
6
7    Mientras i < h1.size() o j < h2.size() //falta recorrer h1 o h2
8      Coordenada actualh1 <- h1[i]
9      Coordenada actualh2 <- h2[j]
10
11     Si i >= h1.size() //ya se recorrio todo h1
12       resultado.agregar(actualh2)
13       j++
14     Si no
15     Si j >= h2.size() //ya se recorrio todo h2
16       resultado.agregar(actualh1)
17       i++
18     Si no
19
20     Si actualh1.x < actualh2.x
21       Si actualh1.y > ulth2.y
22         resultado.agregar(actualh1)
23       Si actualh1.y <= ulth2.y
24         Si ulth1.y > ulth2.y
25           Coordenada auxiliar(actualh1.x, ulth2.y)
26           resultado.agregar(auxiliar)
27       Si actualh1.y == 0 y ulth2.y == 0
28         resultado.agregar(actualh1)
29       ulth1 <- actualh1
30       i++
31     Si no
32     Si actualh1.x > actualh2.x
33       Caso analogo al anterior
34     Si no
35     Si actualh1.x == actualh2.x
36       resultado.agregar(mayorAltura(actualh1, actualh2))
37       ulth1 <- actualh1
38       i++
39       ulth2 <- actualh2
40       j++
41
42   Fin Mientras
43
44   return resultado
45 Fin
```

2.3. Correctitud

EdificiosAHorizontes

Esta es la primer funcion que se llama al principio de todo el algoritmo, sirve para transformar los edificios que se reciben, a un tipo de dato mas facil de manipular para las funciones que le siguen y ademas asegurar ciertas precondiciones: el contenido de todo horizonte tiene estrictamente dos coordenadas, y estas a su vez estan ordenadas de menor a mayor. Se puede ver facilmente esto en el desarrollo, ya que x1 y x2, representando el parante izquierdo y derecho respectivos de un edificio, siempre se cumple que x1 $<$ x2.

UnirHorizontes

Esta funcion usa la tecnica algoritmica Divide & Conquer para distribuir la union de los horizontes. Se llama recursivamente: Utiliza un calculo de indices para poder usar una referencia al contenedor de horizontes, evitando asi la copia en cada llamado recursivo. Dichos indices sirven para marcar que horizontes analizar, y siempre se le pasa la mitad del tamaño original, por lo tanto siempre disminuye y va a llegar a un caso base obligatoriamente, que es cuando hay un solo horizonte, o hay dos.

Como nucleo el algoritmo se basa en la funcion Unir, que es la que se encarga de efectivamente unir a a dos horizontes en uno solo.

Unir

Aca queremos probar, que efectivamente la funcion Unir une dos horizontes en uno solo, formando el contorno resultado entre ambos. Para esto se va a utilizar un **invariante** de ciclo:

Llamaremos a partir de ahora:

- H1 y H2 a ambos horizontes respectivos que recibe por parametro,
- i y j a los indices de hasta que posicion se recorrio de H1 y H2 respectivamente.
- Hres al horizonte resultado, en el que se van a ir acumulando el resultado de los analisis entre las coordenadas de H1 y H2
- actualh1 y actualh2 a las coordenadas actuales respectivas que se analizan de H1 y H2
- ulth1 y ulth2 a las ultimas coordenadas analizadas de H1 y H2 respectivamente
- intervalo a la altura que hay entre una posicion x de una coordenada y la siguiente, en la cual se produce un cambio de altura
- I al invariante
- B a la guarda del ciclo: i $<$ h1.size() || j $<$ h2.size()

Invariante del ciclo:

I1. Hres esta ordenado desde la posicion 0 hasta i+j de menor a mayor segun el eje x de cada coordenada

I2. $\forall$intervalo $\in$ Hres, intervalo == mayor intervalo en esa posicion entre H1 y H2.

I3. $\forall$c1: coordenada $\in$ Hres marcan un cambio de altura, es decir, $\nexists$ c2:coordenada $\in$ Hres tal que c1 esta al lado de c2 y c1.y == c2.y

Entonces queremos demostrar:

I. Al inicio de la funcion vale I

II. I sigue valiendo al finalizar cada iteracion

III. Con cada iteracion nos acercamos mas a $\neg$B

IV. I $\wedge$ $\neg$B $\Rightarrow$ Qc: Se formo el contorno resultado de unir ambos horizontes totalmente

[1] Se sabe por la funcion **EdificiosAHorizontes**, que se utiliza al comienzo del algoritmo, que cada uno de los horizontes transformados tienen sus coordenadas ordenadas de menor a mayor por orden de aparicion en el eje horizontal X, y que ademas la ultima coordenada de estos, la altura vale 0. Entonces podemos suponer que las coordenadas que recibe la funcion Unir siempre estan ordenadas, gracias al invariante sabemos que el resultado va a seguir siempre este orden, concluyendo en que todo horizonte esta ordenado.

I. Al inicio de la funcion vale el invariante I

Como al inicio i = 0 y j = 0, ⇒

I1. Hres esta ordenado desde la posicion 0 hasta 0 de menor a mayor segun el eje x de cada coordenada

I2. No hay ningun intervalo, entonces el para todo se hace verdadero

I3. No hay ninguna coordenada, entonces el para todo se hace verdadero

Entonces vale el invariante porque se cumplieron todas las condiciones de el.

II. El invariante I sigue valiendo al finalizar cada iteracion

Existen cinco casos a analizar

1. **Si i >= h1.size:**

   Esto implica que no hay mas elementos de h1 que comparar ya que se los termino de recorrer, por lo tanto agrega al actualh2 al resultado y avanza el j.

   I1. Como actualh2.x va a tener siempre un valor mayor que ulth1.x, porque sino no hubiese terminado de recorrer h1 y por lo tanto no estaria en este caso, entonces se cumple que sigue estando ordenado.

   I2. Como el intervalo de h1 a partir de que termino quedo en 0, el valor de la coordenada en Hres para el intervalo correspondiente de actualh2.x, va a ser actualh2.y. Entonces se sigue cumpliendo que Hres tiene la mayor altura de intervalos de H1 y H2 en todas sus posiciones.

   I3. Como el H1 no tiene mas coordenadas, se van a seguir agregando las coordenadas de H2 y se sabe por la entrada, que esta no tiene ninguna coordenada que tenga otras coordenadas al lado que repitan su altura.

   El unico caso que se puede llegar a dar, es que justo termino de recorrerse H1 y en la siguiente iteracion a Hres se le empiecen a agregar coordenadas de H2, pero como H1 termino, significa que la altura de su ultima coordenada quedo en 0, y se vera en casos mas adelante, que se lo ignora o transforma por la altura que tiene el intervalo H2 en ese punto. Entonces queda imposible que tenga dos coordenadas consecutivas con la misma altura Hres.

2. **Si j >= h2.size**

   Analogo al anterior, con H1 cambiado por H2

3. **Si actualh1.x < actualh2.x**

   Aun no se termino de recorrer ningun horizonte, por lo tanto elejimos al que este primero segun el eje X (por **[1]** sabemos que estan ordenados, y vamos a estar eligiendo siempre al menor).

   Como la coordenada actual depende unicamente del ultimo cambio en altura que se hizo, es decir, la altura que venia teniendo el intervalo, y debido a que las coordenadas de un mismo horizonte cumplen con el invariante (si llegase a agregar dos coordenadas seguidas del mismo horizonte no tendria problema ya que cumplen que tienen diferentes alturas) entonces solo se debe analizar respecto de la ultima coordenada analizada del horizonte contrario: necesitamos ver si es mayor, menor o igual respecto a su altura.

   a) Si actualh1.y > ulth2.y

   Si la coordenada que estoy analizando actualmente, supera en altura al ultimo intervalo que existe, entonces estoy marcando un nuevo intervalo, por lo tanto se agrega esta nueva coordenada.

   b) Si actualh1.y <= ulth2.y

   Dado que la coordenada que estoy analizando actualmente, es menor igual en altura al intervalo de H2 en ese punto, esto puede significar dos cosas:

   1) ulth1.y > ulth2.y

   El valor del intervalo del horizonte cuya coordenada que estoy analizando, estaba por arriba del intervalo del horizonte contrario y disminuyo hasta quedar por debajo o igual. Entonces la maxima altura a partir de este punto actualh1.x, deberia ser ulth2.y. Se agrega al resultado entonces la coordenada (actualh1.x, ulth2.y).

2) El valor del intervalo del horizonte cuya coordenada que estoy analizando, estaba por debajo o era igual al del intervalo del horizonte contrario y su nueva altura sigue sin superarlo. Entonces lo ignoro ya que no presenta ningun cambio en la altura maxima del intervalo en este punto.

c) Si actualh1.y == 0 && ulth2.y == 0

Si la coordenada que estoy analizando actualmente tiene altura 0, significa que el contorno que se estaba dibujando se acaba de terminar y ademas la altura del intervalo del horizonte opuesto en ese punto es 0, o sea que ya no habia ningun contorno desde su ultima coordenada, entonces lo unico que importa es que el intervalo del horizonte actual, bajo en altura hasta 0, por lo tanto se agrega la coordenada actualh1 al resultado.

Una vez terminado todo este analisis, como en todos los casos se agrego o se ignoro la coordenada actual, se guarda como la ultima que se proceso y se avanza a la siguiente (i++).

Dado que:

I1. Se demostro que Hres sigue estando ordenado desde la posicion 0 hasta i+j de menor a mayor segun el eje x de cada coordenada, porque elegimos al menor de ambos siempre.

I2. Por todo el analisis anterior, en el que se revisa caso por caso cual es el intervalo con mayor altura que deberia ir, quedo demostrado que la eleccion de como elegir la coordenada para agregarla al resultado, va a seguir manteniendo la propiedad en la que el intervalo que marca la nueva coordenada es la mayor altura para ese intervalo en esa posicion entre H1 y H2

I3. Es imposible que se repita la altura que venia teniendo el intervalo, porque de suceder esto, habria estado repetida tambien en H1, y esto no sucede nunca. Ademas, en el caso que la altura que queriamos agregar ya era ¡= al intervalo de la ultima coordenada de H2, la ignoramos. Por lo tanto, todas las coordenadas que se encuentran una al lado de otra, marcan diferentes alturas.

Entonces quedaron demostradas todas las propiedades, probando que **se sigue cumpliendo el invariante**.

4. **Si actualh1.x > actualh2.x**

Segundo caso a darse, que la primer coordenada en aparecer no sea la de H1, sino la de H2, es analogo al caso anterior

5. **Si actualh1.x == actualh2.x**

Es el ultimo caso a analizar, en el que ambas aparecen en el mismo momento, entonces la desicion a tomar es elegir la que tiene mayor altura, que va a ser la que marca el intervalo mas alto desde ese punto, por lo tanto se guarda en el resultado la que mayor altura tiene, se guardan ambas coordenadas como las ultimas procesadas, y se avanzan a las siguientes.

Dado que:

I1. Se demostro que Hres sigue estando ordenado desde la posicion 0 hasta i+j de menor a mayor segun el eje x de cada coordenada, por **[1]** siempre se elige a la primer coordenada de cada horizonte, y en este caso ambas aparecen en el mismo momento sobre el eje x, por lo tanto coincide en ambos horizontes, y es la siguiente en orden de aparicion para estar en Hres.

I2. Quedo comprobado que el intervalo que marca la nueva coordenada es la mayor altura para ese intervalo en esa posicion entre H1 y H2

I3. Es imposible que se repita la altura que venia teniendo el intervalo, porque de suceder esto, habria estado repetida tambien en H1 o H2, y esto no sucede nunca. Por lo tanto no existe una coordenada en Hres que tenga al lado otra con la misma altura

Entonces quedaron demostradas todas las propiedades, probando que **se sigue cumpliendo el invariante**.

III. Con cada iteracion nos acercamos mas a ¬B

En el pseudocodigo se puede observar que hay 3 casos que abarcan toda la iteracion:

1. Si actualh1.x $<$ actualh2.x $\Rightarrow$ Se incrementa i
2. Si actualh1.x $>$ actualh2.x $\Rightarrow$ Se incrementa j
3. Si actualh1.x == actualh2.x $\Rightarrow$ Se incrementa i y j

B == i $<$ h1.size || j $<$ h2.size.

Se sabe que h1.size y h2.size son fijos y siempre son mayores o iguales a 0, y ademas **siempre se va a estar incrementando alguna de las variables** i o j, que arrancan desde 0. Entonces por cada iteracion siempre se va a estar acercando cada vez mas al valor de h1.size o h2.size.

Una vez que i alcanza el tamaño de h1.size, esto significa que finalizo de recorrer h1, entonces pasaria a agregar solamente elementos de h2 y a avanzar j. Lo mismo sucede si finaliza de recorrer h2 antes, agrega solamente elementos de h1 y a avanzar i. En el caso de que ambos valores llegan simultaneamente a h1.size y h2.size, entonces ya cumplimos ¬B.

IV. I $\wedge$ ¬B $\Rightarrow$ Qc

¬B $\Rightarrow$ i == h1.size && j == h2.size

Haciendo reemplazo en de i y j, obtenemos:

I1. Hres esta ordenado desde la posicion 0 hasta h1.size + h2.size de menor a mayor segun el eje x de cada coordenada

I2. $\forall$intervalo $\in$ Hres, intervalo == mayor intervalo en esa posicion entre H1 y H2.

I3. $\forall$c1: coordenada $\in$ Hres marcan un cambio de altura, es decir, $\nexists$ c2:coordenada $\in$ Hres tal que c1 esta al lado de c2 y c1.y == c2.y

Lo cual significa que tenemos el resultado del contorno completo de unir h1 y h2.

Dado que tenemos **demostrada la correctitud del ciclo**, sabemos que vale Qc: Se formo el contorno resultado de unir ambos horizontes totalmente.

Sabemos la funcion Unir cumple la precondicion de i = 0 y j = 0, y realiza el ciclo previamente demostrado, **Entonces sabemos que la funcion Unir resuelve correctamente la union de dos horizontes.**

La funcion UnirHorizontes para resolver el problema correctamente depende de Unir para unir sus casos recursivos, como demostramos que Unir es correcto, y ademas que UnirHorizontes siempre llega a un caso base, **queda demostrado que UnirHorizontes es correcto**.

Correctitud de todo el algoritmo:

El algoritmo principal se encarga de ejecutar dos cosas:

1. La transformacion de EdificiosAHorizontes: se encarga de cumplir las precondiciones de entrada para UnirHorizontes, que los necesita para pasarselos a Unir
2. UnirHorizontes con todos los horizontes que se transformaron anteriormente

Dado que se cumplen los prerrequisitos antes de llamar a UnirHorizontes, y que UnirHorizontes es correcto debido a que Unir es correcto, **entonces sabemos que el algoritmo en su totalidad es correcto.**

2.4. Complejidad

Para demostrar la complejidad de este algoritmo, vamos a proceder a analizar por separado las 3 funciones que se utilizan:

1. **EdificiosAHorizontes**, se usa al principio del algoritmo, dado que la transformacion de un edificio a horizonte, tiene complejidad constante $\Theta(1)$, entonces hacerlo para n edificios tiene costo de $\Theta(n)$

2. **Unir** recibe dos Horizonte, que son contenedores de Coordenada, y va iterando sobre cada una hasta haber analizado que hacer con todas las coordenadas, dicho analisis tiene costo $\Theta(1)$. Llamemos *c1* y *c2* a la cantidad de coordenadas de cada horizonte respectivamente, realizar entonces el analisis sobre todas las coordenadas tiene una complejidad final de $O(c1 + c2)$, pero como al comienzo del algoritmo siempre se le pide reservar memoria al contenedor para poder almacenar todas las coordenadas, esto pasa a ser $\Theta(c1 + c2)$.

   Si bien este es un analisis con la cantidad de coordenadas y no de edificios, por la transformacion de edificio a horizonte que se hace al comienzo de todo el algoritmo, se puede ver que en realidad un edificio es lo mismo que dos coordenadas. Pero esto solo vale al principio, ya que una vez realizada la union, la cantidad de coordenadas disminuye o se mantiene igual. Por lo tanto la cantidad de coordenadas siempre se mantiene acotada por el doble de la cantidad de edificios que hubo en esa union.

   Llamemos *n1* y *n2* a la cantidad de edificios de cada horizonte respectivamente, y n a cantidad total de edificios de entrada, por la justificacion anterior entonces se puede ver que $\Theta(c1 + c2) \in \Theta(2n1 + 2n2)$, que termina siendo $\Theta(2n)$ y finalmente, $\Theta(n)$

3. **UnirHorizontes** realiza dos llamados recursivos con la mitad de horizontes en cada lado, el costo de las operaciones que no son recursivas van a ser *f(n)*, que queda dependiendo de la funcion Unir, ya que todas las demas operaciones son $\Theta(1)$. El caso con un solo horizonte, que simplemente lo devuelve, tiene complejidad $\Theta(1)$.

   Esto nos deja en condiciones necesarias y suficientes para poder usar el **Teorema Maestro**, que sirve para resolver recurrencias de la forma:

$$T(n) = \begin{cases} \alpha T(n/c) + f(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

   Donde: $\alpha = 2$, c = 2, y $f(n) = \Theta(Unir) = \Theta(n) = \Theta(n^{log_2 2})$.

   Entonces segun el **Teorema Maestro**, si $f(n) \in \Theta(n^{log_2 2}) \Rightarrow \boldsymbol{T(n) = \Theta(n \log n)}$

Sumando entonces las complejidades de **EdificiosAHorizontes** y **UnirHorizontes**, la complejidad final del algoritmo es $\Theta(n) + \Theta(n \log n) = \Theta(n \log n)$.

2.5. Experimentación

Para el proceso de experimentación del problema se plantearon distintas pruebas para corroborar que el algoritmo propuesto funcionara correctamente y que la cota de complejidad encontrada y justificada en la sección anterior, en la práctica, se cumpliera.

Al igual que en el ejercicio 1, dado que el CPU de la computadora utilizada para tomar los tiempos no está atendiendo únicamente a nuestro proceso, realizar una sola vez cada prueba podría darnos valores que no son cercanos a los reales. Por lo que para minimizar este margen de error, a cada prueba se la hizo ejecutar un total de 10.000 veces y se tomó el mejor valor. Notar que tomar el mejor valor no es una mala decisión, ya que mientras más chico sea el valor, más cerca estamos del valor real de tiempo que toma el algoritmo para una instancia dada.

En cada prueba se tomaron métricas para la posterior evaluación del algoritmo en la práctica. Notar que la medición no contempla tiempos de entrada/salida de datos, sino que contempla solamente el núcleo del algoritmo.

Se representó la información tomada mediante gráficos 2D que permitan ver de una manera más clara los resultados obtenidos en las pruebas. Estos fueron realizados con el software QitPlot que la cátedra proveyó.

En cuanto a qué casos testear, nuestro algoritmo no presenta casos "border". Es decir, no tiene un peor/mejor caso, sino que para cualquier instancia de edificios cargada, realizará el mismo procedimiento, tomando $\Theta(n \log n)$. Ni el tamaño de los edificios, ni su posición en el suelo de la ciudad, ni la posición relativa entre ellos afecta el tiempo de computo del algoritmo, por lo que el único parámetro variable a la hora de realizar pruebas es la **cantidad** de edificios.

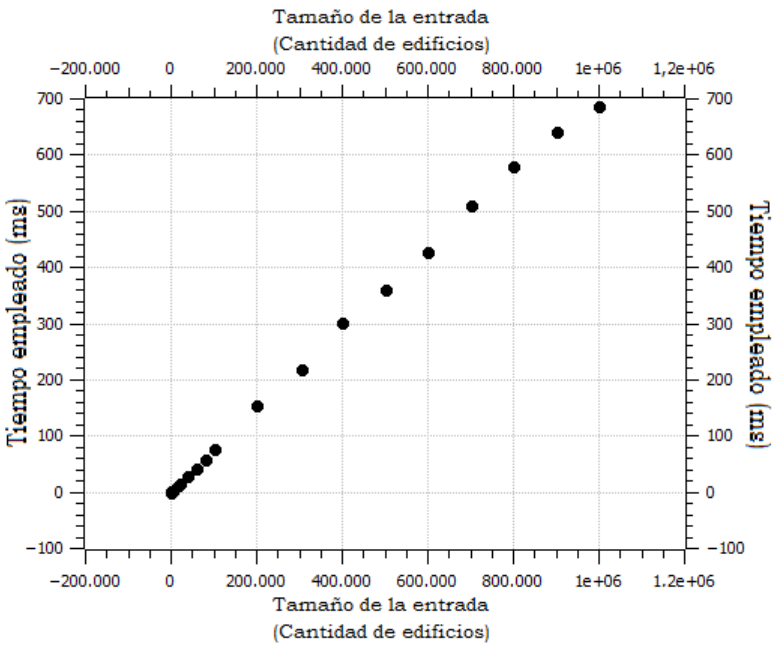
Esto es así ya que el algoritmo que diseñamos, utiliza la estrategia de Divide & Conquer para su resolución, como fue mencionado en insisos anteriores. Esta técnica va dividiendo el problema en varios sub-problemas, sin importar la relación entre los edificios de la instancia. Por lo que, si suponemos que existiera un mejor/peor caso de entrada, de todas formas este sería fraccionado en sub-problemas más pequeños hasta llegar al caso base, y el formato de entrada original se habría perdido. Por este motivo es que el algoritmo no presenta un peor ni un mejor caso de resolución.

Dicho, eso, se diseñó un programa que dado un numero n de edificios, genera n edificios aleatorios para probar el algoritmo. Para facilitar la tarea de experimentación, dicho programa era capaz de generar más de una instancia aleatoria, con distintas cantidades de edificios (cada una elegida en la interfaz de dicho programa).

Para todos los casos, se eligió una precisión de hasta 0,0001 ms (milisegundos). De ser menor, la notamos como 0. En todos los casos se pudo comprobar que la práctica refleja lo expuesto en incisos anteriores.

A continuación presentamos el gráfico 2D que refleja las pruebas realizadas. Para cada tamaño se realizaron pruebas con instancias distintas y aleatorias y las diferencias muy sutiles (del orden de los microsegundos). A sí mismo, a cada una de esas distintas instancias experimentadas de cada tamaño, se la hizo ejecutar un total de 10.000 veces por los motivos explicados anteriormente.

edificios	milisegundos
10	0
100	0
200	0
400	0,1
800	0,4
1.000	0,6
10.000	7
20.000	14
40.000	29
60.000	43
80.000	58,7
100.000	75,5
200.000	153,9
300.000	215,7
400.000	305,4
500.000	360
600.000	427
700.000	510
800.000	580
900.000	640
1.000.000	686,7



3. Ejercicio 3

3.1. Introducción

Contexto

Una importante empresa de logística de sustancias debe llevar a cabo la tarea de transportar una cantidad determinada de químicos desde una fábrica hasta un depósito. Las sustancias a transportar tienen entre cada par de ellas, una propiedad llamada "peligrosidad". Para realizar esta tarea, la empresa cuenta con una cantidad ilimitada de camiones con umbral determinado (e igual para todos los camiones) de peligrosidad, es decir, puede soportar hasta un cierta cantidad de sustancias en base a la peligrosidad que estas tienen entre sí. La empresa quiere determinar cual es el mínimo número de camiones necesarios para transportar todos los productos sin que en ningún camión se supere el umbral de peligrosidad y además saber, en qué camion fue colocado cada producto.

El problema a resolver

Dado n el número de productos a transportar, los coeficientes de peligrosidad entre cada par de productos i, j (con i $\neq$ j), h_{ij} y M, la capacidad máxima de "peligrosidad"que un camión puede transportar, devolver una configuración que utilice la mínima de camiones necesarios para transportar todos los productos sin que en ningún camión la peligrosidad de los mismos exceda el umbral M y también devolver la cantidad de camiones que se utilizaron.

Un aspecto a tener en cuenta es que hay varias posibles configuraciones válidas posibles, que incluso requieran la misma cantidad mínima de camiones, siendo cualquiera de éstas una respuesta posible y correcta. Dado esto, se nos pide que utilicemos la técnica de *Backtracking* inteligentemente para que sea lo más veloz posible.

Ejemplos

Para los ejemplos denotaremos:

M, al umbral de peligrosidad máxima que pueden soportar los camiones.
h(x,y) = h(y,x), a la función que define la peligrosidad entre 2 productos x e y.
$h(C) = \sum_{\substack{p_i,p_j \in C \\ i<j}} h_{i,j}$, a la función que define la peigrosidad de un camión.

1. Supongamos un M = 90 y que se nos da el siguiente conjunto de productos:

   $\{p1, p2, p3, p4\}$

   Y sus peligrosidades:

   h(p2,p1) = 10 h(p3,p2) = 10 h(p3,p1) = 10
   h(p4,p3) = 30 h(p4,p2) = 20 h(p4,p1) = 10

   La siguiente configuración es la que debería dar como salida el algoritmo:

   Camión 1 = $\{p1, p2, p3, p4\} \implies$
   h(Camión 1) = h(p2,p1) + h(p3,p2) + h(p3,p1) + h(p4,p3) + h(p4,p2) + h(p4,p1) = 90

   Notar que, al ser conjuntos y la peligrosidad no estar afectada por el orden, la siguiente también es correcta:

   Camión 1' = $\{p4, p3, p2, p1\} \implies$ h(Camión 1) = h(Camión 1')

2. Supongamos un M = 50 y los mismo productos con sus respectivas peligrosidades del Ejemplo1. La configuración de 1 solo vehículo rompe las reglas de seguridad. En este caso, el algoritmo podría optar por organizar las sustancias de esta manera:

   Camión 1 = $\{p1, p2, p3\} \implies$ h(Camión 1) = h(p3,p2) + h(p3,p1) + h(p2,p1) = 30
   Camión 2 = $\{p4\} \implies$ h(Camión 2) = 0

   O bien así:

   Camíon 1' = $\{p1, p2\} \implies$ h(Camión 1') = h(p2,p1) = 10
   Camión 2' = $\{p3, p4\} \implies$ h(Camión 2') = h(p4,p3) = 30

   Cualquiera de estas 2 opciones que decida dar como resultado el algoritmo es considerada correcta, tanto como cualquier otra que utilice nada más 2 vehículos.

3.2. Desarrollo

Para resolver el problema planteado en el inciso 1, una posible opción era generar todas las posibles configuraciones de productos en camiones y luego ver a) si era una configuración válida, es decir, ningún camión superaba el umbral M y b) cuáles de las configuraciones válidas requerían usar la minima cantidad de productos y luego, elegir una de éstas.

Esta solución de fuerza bruta, donde se probaban todas las opciones posibles y luego se elegía la mejor fue descartada rapidamente, ya que la cantidad de configuraciones(válidas o no) equivale a la cantidad de particiones posibles de un conjunto de n productos, y ésta se representa con la formula recursiva de Bell donde:

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

Este número de Bell crece exponencialmente, por lo que salta a la luz la necesidad de utilizar la técnica de backtracking para frenar la generación de soluciones inválidas y de soluciónes que, de antemano, sepamos que no van a ser óptimas.

Backtracking es una técnica algorítmica que, recursivamente va construyendo candidatos a soluciones y abandona cada candidato parcial cuando determina que no va a poder completarse en una solución válida. En nuestro problema, al armar las distinas configuraciones, el algoritmo que presentamos irá generando recursivamente las particiones, pero siguiendo los lineamientos de backtracking, una vez que se detecta que la configuración ya no va a poder ser válida, esta configuración parcial es descartada, así como también, todas las configuraciones que iban a generarse a partir de ésta.

En este problema, lo que determina que una configuración que aún está incompleta sea considerada inválida es:

I. Algún camión superó por los productos contenidos en él el umbral de peligrosidad

II. Se ha superado la mínima cantidad de camiones necesarios para ser solución, es decir, previamente se había detectado que había otra configuración que requería una menor cantidad de camiones.

De esta forma, vemos que usar la técnica de Backtracking genera la solución en un tiempo mucho mejor que al usar fuerza bruta, dado que no llega a generar una gran cantidad de candidatos a solución inválidos a priori, según los lineamientos I. y II., surgidos de la problemática presentada en el inciso 3.1). Vale recalcar que esa cualidad de tener 'candidatos a solución', en nuestro caso, subconjuntos de productos en camiones o 'particiones incompletas' junto con la posibilidad de testear si ésas pueden llegar a producir soluciones válidas, es lo que habilita el uso de esta técnica.

De todas formas, el algoritmo todavía dista de proveer una solución en tiempo polinomial, por lo que incluímos algunos intentos previos de acotar las posibles soluciones, de forma tal que el backtracking sea más efectivo. A estos fines, intentamos:

- Calcular previamente (y de una manera naive) cual es el número mínimo de camiones a partir del cuál es relevante buscar la solución. Caso contrario, el algoritmo de backtracking, más allá de las podas que realizara y que anteriormente describimos, debería considerar en un primer momento, que una solución con n camiones (con n cantidad de productos) es válida. Ejecutando este primer cálculo, acotamos la búsqueda antes de entrar a la función de backtracking, proporcionándole información extra, de forma tal que, en mejor caso (o caso promedio), descartará varias configuraciones parciales por ya poseer la información de que no son solución óptima.

```
1  int dar_Cota_Inicial(Productos, Peligrosidades, Umbral M)
2    res = 0, primeroACargar = 0, peligrosidadActual = 0
3
4    Para todo i desde 0 hasta Productos.size
5      Para todo j desde primeroACargar hasta i
6        peligrosidadActual += Peligrosidad(Producto[i], Producto[j],
             Peligrosidades)
7        Si peligrosidadActual >= M ∨ noHayMasProductos
8          res++
9          peligrosidadActual = 0
10         primeroACargar = i
11       Fin Si
12     Fin Para
13   Fin Para
14   return res
15 Fin
```

Dado el pseudo código, uno puede observar lo 'naive' que se mencionaba anteriormente. Si este algoritmo preliminar al backtracking, que lejos está de chequear todas las posibilidades, devuelve una cantidad de camiones x, con $1 \leq x < n$, entonces el algoritmo de backtracking que hace un análisis exhaustivo de las posibilidades debe dar una solución s tal que $1 \leq s < x$.

Seguido de esto viene el paso de backtracking que efectivamente va a resolver el problema, pero con el agregado de la cota que nos provée el algoritmo recién mencionado. Este pseudo código ilustra la idea:

```
1  void backtracking(Productos,Peligrosidades,M,CamionesActuales,OptimaCant,
       CamionesRes)
2    //Nos fijamos si esta solucion parcial es valida y optima, luego si es una
         solucion final
3    Si CamionesActuales.size ≤ OptimaCant ∧ CheckUmbral(CamionesActuales,
         Peligrosidades,M)
4      Si YaVimosTodosLosProductos
5        OptimaCant = CamionesActuales.size
6        CamionesRes = CamionesActuales
7        return
8      Fin Si
9
10     //Faltan agregar Productos, pruebo el proximo producto en cada camion y en
           un camion nuevo.
11     Producto x = Productos.ultimo
12     Productos.SacarUltimo
13
14
15     Para todo i desde 0 hasta CamionesActuales.size
16       CamionesActuales[i].PonerAlFinal(x)
17       backtracking(Productos,Peligrosidades,M,CamionesActuales,OptimaCant,
             CamionesRes)
18       //Saco el producto para poner en el proximo camion
19       CamionesActuales[i].SacarUltimo
20     Fin Para
21
22     //Lo pongo en un camion aparte
23     Camion NuevoCamion
24     Camion.PonerAlFinal(x)
25     CamionesActuales.PonerAlFinal(NuevoCamion)
26     backtracking(Productos,Peligrosidades,M,CamionesActuales,OptimaCant,
           CamionesRes)
27   Si no //Estoy en un caso donde se supera el umbral o la cantidad de camiones
         supera la optima
28     return
29   Fin si
30 Fin
```

3.3. Complejidad

Algunas aclaraciones previas al análisis de complejidad del ejercicio:

1. Producto es int, Productos es Vector(int), Camion es Vector(int), Camiones es Vector(Camion), Peligrosidades es Vector(Vector(int))
2. Asumo que las operaciones de vector PonerAlFinal(push back), sacarUltimo (pop back), Constructor por defecto (sin indicador de tamaño), el operator[], size y la asignación (por referencia) tienen complejidad $O(1)$. Cabe aclarar que esto no es siempre cierto para push back, pero para simplificar el argumento voy a tomarlo como tal.
3. La subrutina CheckUmbral(Camiones,Peligrosidades,M) comprueba que los camiones sean válidos, o sea que no excedan el umbral M, Camiones y Peligrosidades se pasan por referencia, al igual que el M. La función que caracteriza su complejidad pertenece a O(n^2), donde n son los productos dispersos por los camiones. La función responde a este pseudo codigo.

```
1  bool CheckUmbral(Camiones,Peligrosidades,M)
2    Para todo i desde 0 hasta Camiones.size
3      int acum = 0
4      Si Camiones[i].size > 1
5        Para todo x desde 0 hasta Camiones[i].size - 1
6          Para todo y desde x+1 hasta Camiones[i].size
7            Producto px = Camiones[i][x]
8            Producto py = Camiones[i][y]
9            acum += Peligrosidad(px,py,Peligrosidades)
10         Fin Para
11         Si acum >= m
12           return falso
13         Fin Si
14       Fin Para
15     Fin Si
16   return verdadero
17 Fin
```

Dicho esto, el análisis de la complejidad del algoritmo de backtracking esta sujeta a esta función recursiva:

$$T(i) = \begin{cases} (n-(i+1)+1)T(i-1) + O((n-i)^2) & \text{, si } 0 < i < n \\ O(n^2) & \text{, si } i = 0 \end{cases}$$

Cancelando los índices:

$$T(i) = \begin{cases} (n-i)T(i-1) + O((n-i)^2) & \text{, si } 0 < i < n \\ O(n^2) & \text{, si } i = 0 \end{cases}$$

Donde i es el número del próximo producto a ser puesto en los distintos camiones. Si el for principal del algoritmo esta sujeto a la cantidad de camiones, la pregunta es, ¿Por qué la función de recursión no lo está también? La respuesta yace en el análisis del peor caso. La máxima cantidad de llamados recursivos ocurre cuando cada camión tiene 1 solo producto, dado que se genera la mayor cantidad de camiones para dicho llamado recursivo. Si el próximo producto a ser analizado es el i, entonces los elementos i+1...n ya fueron colocados en distintos camiones, por ende la cantidad en peor situación es (n-(i+1)). Sumado a esto, ocurre un último llamado recursivo en el que se coloca el producto i en un nuevo camión, por ello el +1 multiplicando a T(i-1). El segundo término de T(i) describe la complejidad de hacer CheckUmbral en dicho paso. El caso base es hacer CheckUmbral con todos los químicos colocados.

Supongamos una cantidad n (con n $\geq$ 5 para el caso particular de esta demostración) de productos, esto implica que el último producto es $p_n - 1$, o el producto (n-1), desarrollando la función desde (n-1) se obtiene lo siguiente:

$T(n-1) = (n-(n-1))T(n-2) + O((n-(n-1))^2) = 1T(n-1) + O(1) = 2T(n-3) + O(2^2) + O(1) =$

$2(3T(n-4) + O(3^2)) + O(2^2) + O(1) = 3!T(n-4) + O(2*3^2) + O(1) =$

$3!(4T(n-5) + O(4^2)) + O(2*3^2) + O(1) = 4!T(n-5) + O(3!*4^2) + O(2!*3^2) + O(0!*1^2) =$

$...(i = 0)... = (n-1)!T(0) + \sum_{k=1}^{n-1} O((k-1)! * k^2) = (n-1)!T(0) + O(\sum_{k=1}^{n-1}(k! * k))$

Por lo demostrado en el ejercicio 1.f de la Práctica 1, por propiedades de la función O y de la función factorial, nos queda que:

$(n-1)!T(0) + O(((n-1)+1)!) - 1 = (n-1)!O(n^2) + O(n!) - 1 = O((n-1) * n^2)) + O(n! - 1) =$

$O(n * n!) + O(n!)$

Además, como n $\in \aleph \implies (n * n! \geq n! \iff n \geq 1)$ lo cual es cierto ya que n es un natural. Por ende, $O(n * n!) + O(n!) = O(max\{n * n!, n!\}) = O(n * n!) = T(n-1)$ que es de donde arrancamos. Y $T(n-1)$ define la complejidad del algoritmo. Finalmente, el algoritmo tiene complejidad O$(n * n!)$.

Esto no es todo ya que a esta complejidad hay que sumarle el costo temporal del algoritmo Dar_Cota_Inicial. El análisis de él es más simple. La peor circumstancia que puede ocurrir para Dar_Cota_Inicial es que todos los productos puedan concatenarse en un único camión, ya que para meter un producto i, tengo que obtener la peligrosidad (en $O(1)$) de i con respecto a todos los elementos desde 0...i-1 anteriores. Se harían $1 + 2 + 3 + ... + (n-1) + n = \frac{n*(n+1)}{2}$ (por Ejercicio 1.a, Práctica 1) pedidos de coeficientes de peligrosidad. El algoritmo es de orden temporal cuadrático.

En conclusión, si decimos que $f(n)$ es la función que caracteriza la complejidad del algoritmo que resuelve el problema, podemos decir que $f(n) = O(n * n!) + O(n^2)$. Sabemos, por el ejercicio 3.b de la Práctica 2, que dado r $\in \aleph$, $n^r \in O(n!) \implies n^r \in O(n * n!)$. Entonces, $f(n) = O(n * n!)$.

Si bien la cota de lo que podría tomar resolver el problema es holgada, recordemos que realmente se llega a ella en el caso en que ningún producto pueda viajar a la par de otro, ya que se deben descartar todas las particiones posibles antes de poder decir con seguridad que es la manera más eficiente de transportarlos.

3.4. Experimentación

Para el proceso de experimentación del problema se plantearon distintas pruebas para corroborar que el algoritmo propuesto funcionara correctamente y que la cota de complejidad encontrada y justificada en la sección anterior, en la práctica, se cumpliera.

Al igual que en el ejercicio 1 Y 2, dado que el CPU de la computadora utilizada para tomar los tiempos no está atendiendo únicamente a nuestro proceso, realizar una sola vez cada prueba podría darnos valores que no son cercanos a los reales. Por lo que para minimizar este margen de error, a cada prueba se la hizo ejecutar un total de 750 veces (menos que las de los ejercicios 1 y 2 ya que la complejidad de este algoritmo no es polinomial) y se tomó el mejor valor. Notar que tomar el mejor valor no es una mala decisión, ya que mientras más chico sea el valor, más cerca estamos del valor real de tiempo que toma el algoritmo para una instancia dada.

En cada prueba se tomaron métricas para la posterior evaluación del algoritmo en la práctica. Notar que la medición no contempla tiempos de entrada/salida de datos, sino que contempla solamente el núcleo del algoritmo.

Se representó la información tomada mediante gráficos 2D que permitan ver de una manera más clara los resultados obtenidos en las pruebas. Estos fueron realizados con el software QitPlot que la cátedra proveyó.

Para el testeo, se diseñó un generador de instancias aleatorias, que dado un umbral y una cantidad de productos, genera aleatoriamente la peligrosidad entre los mismos. Dicho software es capaz de generar múltiples instancias que el algoritmo del ejercicio 3 resolvería todas juntas.

Con este software pudimos evaluar cuanto toma nuestro algoritmo para distintas instancias aleatorias del problema.

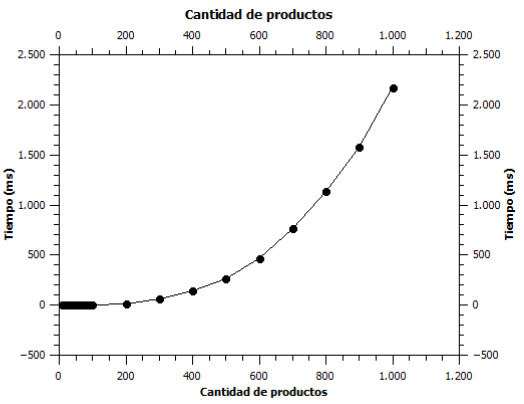
Para todos los casos, se eligió una precisión de hasta 0,0001 ms (milisegundos). De ser menor, la notamos como 0.

En todos los casos se mantuvo constante a M, y se probaron distintas cantidades de edificios, dejando de manera aleatoria la peligrosidad entre ellos, tal como fue explicado antes cuando se describió al generador de instancias del problema. También aprovechamos los gráficos para realizar una comparación, entre lo que tarda el algoritmo normalmente, y lo que tarda cuando retiramos la çota inicial"que pusimos en el desarrollo del programa para mejoras de performance.

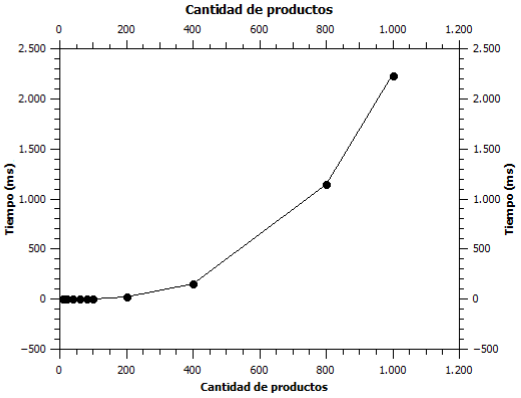
A continuación presentamos los distintos gráficos en 2D que reflejan las pruebas realizadas. Para cada tamaño se realizaron pruebas con instancias distintas y las diferencias fueron muy sutiles (del orden de los microsegundos).

ESCENARIO CON M = 2

Productos	milisegundos
10	0
20	0
30	0
40	0
50	0
60	0
70	1
80	1
90	2
100	2
200	15
300	62
400	141
500	265
600	468
700	765
800	1.140
900	1.578
1.000	2.172

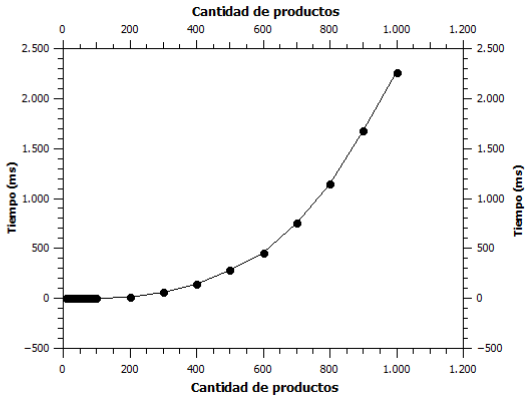


productos	milisegundos
10	0
20	0
40	0
60	0
80	1
100	2
200	20
400	150
800	1143
1000	2234

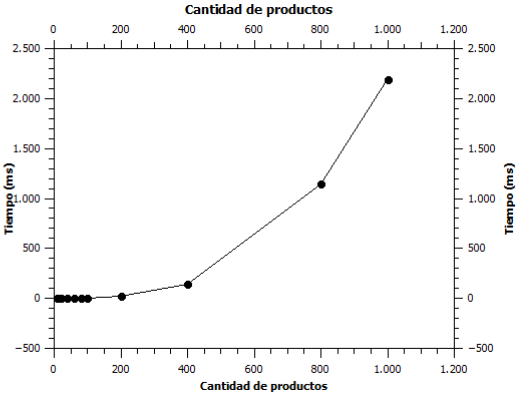


ESCENARIO CON M = 16

Productos	milisegundos
10	0
20	0
30	0
40	0
50	0
60	0
70	0
80	0
90	0
100	0
200	15
300	62
400	140
500	281
600	453
700	750
800	1.149
900	1.682
1.000	2.258

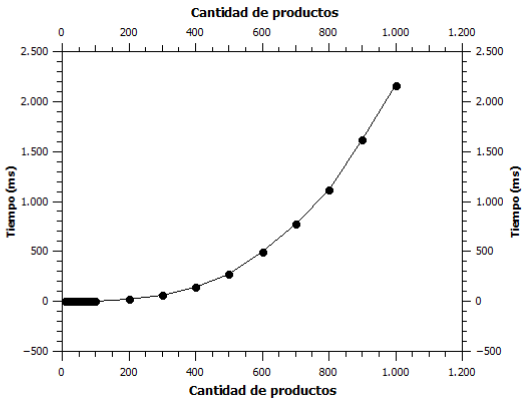


productos	milisegundos
10	0
20	0
40	0
60	0
80	1
100	2
200	20
400	142
800	1150
1000	2187

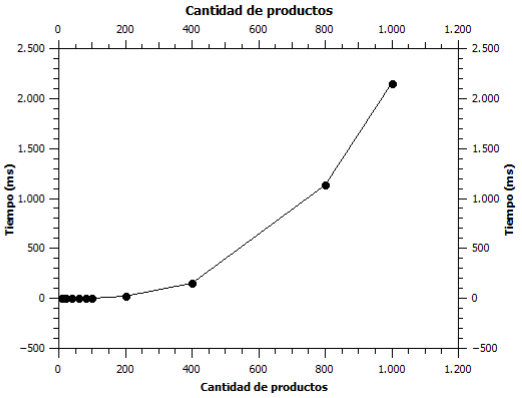


ESCENARIO CON M = 64

Productos	milisegundos
10	0
20	0
30	0
40	0
50	0
60	0
70	1
80	1
90	2
100	2
200	21
300	65
400	144
500	277
600	490
700	774
800	1.119
900	1.615
1.000	2.156

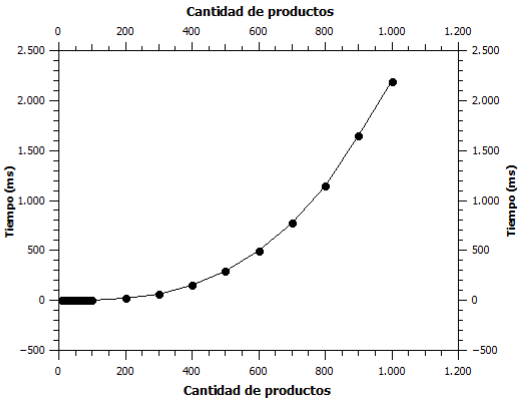


productos	milisegundos
10	0
20	0
40	0
60	0
80	1
100	3
200	19
400	150
800	1132
1000	2144

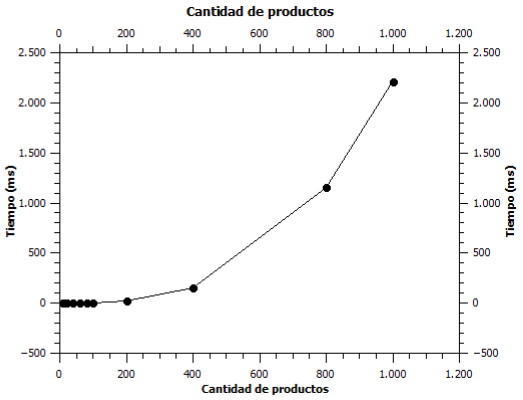


ESCENARIO CON M = 256

Productos	milisegundos
10	0
20	0
30	0
40	0
50	0
60	0
70	1
80	1
90	2
100	3
200	21
300	65
400	148
500	290
600	492
700	778
800	1.148
900	1.644
1.000	2.187



productos	milisegundos
10	0
20	0
40	0
60	0
80	1
100	3
200	20
400	149
800	1158
1000	2212



3.4.1. Algunas conclusiones

Para este ejercicio, nos sucedió que su complejidad dificultaba la experimentación, por lo que cada experimento demoró mucho tiempo, ya que el algoritmo es exponencial.
Hay dos cuestiones que, con el tiempo con el que dispusimos, intentamos analizar en las pruebas:

1. Diferencias entre el tamaño de la cantidad de umbrales: A pesar de que nuestra intuición nos llevó a pensar que las instancias de umbral muy pequeño iban a arrojar un peor resultado que las de umbral mayor, no pudimos comprobar ésto en las pruebas realizadas, ya que no llegamos a observar mayores diferencias en los resultados de las distintas instancias.

2. Diferencias de la performance del algoritmo al utilizar la función de dar cota Inicial: En este caso, nuevamente nuestra intuición nos llevó a pensar que dar la cota inicial podía llegar a reducir considerablemente el tiempo de performance de nuestro algoritmo, no pudimos comprobarlo empíricamente, ya que no pudimos comprobar que se mejoraran los tiempos en las instancias que testeamos.

Sin embargo, una vez concluídos los experimentos que llegamos a realizar, y sin más tiempo de realizar otros posteriores, creemos que una buena idea hubiera sido realizar los gráficos utilizando una escala logarítmica, para poder apreciar mejor los cambios en la curva, ya que al ser exponencial y crecer tan rápidamente, estas variaciones resultaron impercibibles en los gráficos.