Trabajo Práctico 1

Viernes 3 de octubre de 2014 — Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Florencia Lagos	313/12	cazon_88@hotmail.com
Martín Caravario	470/12	martin.caravario@gmail.com
Federico Hosen	825/12	fhosen@hotmail.com
Bruno Winocur	906/11	brunowinocur@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellon I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autonoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Problema 1

1.1. Descripción del problema

Este problema consiste en realizar un algoritmo que se encargue de calcular la cantidad de saltos sobre los tablones de un puente, que debe realizar un participante de un concurso con el objetivo de cruzarlo en la menor cantidad de saltos posible.
Para esto contamos con la cantidad de tablones del puente a cruzar y el estado de estos,es decir si estan rotos o sanos. También disponemos de la capacidad de salto,es decir cual es el salto mas largo que puede realizar el participante. La solución del problema, si es que tiene, debera ser mostrada por pantalla, indicando la cantidad de saltos a realizar, seguido por los tablones a los cuales el participante debera saltar. Los tablones estan numerados de 1 a n, y asumimos que el participante comienza y termina fuera del puente. En caso de que el participante no pueda cruzar el puente, se retornará la palabra "no".
Una posible instancia de este problema podría ser una en la que el participante tenga una capacidad de salto 4, y el puente tenga solamente 3 tablones, sin importar el estado de estos. En ese caso la solucion seria de la forma: 1 4, ya que se necesita un solo salto para cruzar el puente, y saltaría al tablón número 4, lo cual indica que cae fuera del puente.
Otro ejemplo sería con una entrada de la forma 6 2 0 1 0 1 0 0, 6 es el tamaño del puente, 2 la capacidad de salto del participante y luego se ennumera el estado de los tablones del puente. Para esta instancia la salida seria de la forma 4 1 3 5 7, 4 saltos y los tablones por los que debería pasar el participante.
Un último ejemplo de instanciación posible podría ser uno con la siguiente entrada: 7 2 0 1 0 0 1 1 0, para el cual la solución deberia devolver el mensaje "no", pues el participante no puede cruzar donde hay 2 tablones rotos, siendo 2 su capacidad de salto.

1.2. Ideas desarrolladas

El algoritmo que desarrollamos va a ir leyendo la entrada un participante a la vez. Va a generar el puente correspondiente para ese participante, evaluará si con su capacidad salto y el estado de los tablones lo logra cruzar o no y va a actuar según lo indicado en la descripción de problema.
Aprovechamos varios datos que nos da la instancia,como el tamaño del puente, para no tener que calcularlos posteriormente.
Para la evaluación del puente ideamos una funcion llamada saltar_puente que tomando al parcipante, su puente y dos variables por referencia (saltos y tablones_recorridos) analiza si es factible el cruce del puente, devolviendo el resultado como un bool, y si es factible el cruce modifica las variables saltos y tablones_recorridos asociadas a ese participante para reflejar el camino que recorreria.

La funcion saltar_puente utiliza un comportamiento de algoritmo goloso, buscando siempre el tablon mas lejos al que puede saltar teniendo en cuenta la capacidad de salto del participante y el estado de los tablones (roto o sano).
Presentaremos en esta sección del informe un pseudocódigo de alto nivel y luego en el de análisis de complejidad lo veremos a mas bajo nivel.

```
SALTAR_PUENTE(in puente: vector(int), in c: int, in/out saltos: int, in/out tablones_recorridos: vector(int),
 1  int actual ← 0
 2  int tam ← c
 3   //el tamaño del puente se saca de la entrada para no tener que calcularlo.
 4  while el actual no sea mayor al tamaño del puente
 5    calculo el proximo salto con siguiente_salto en base a miactual
 6    if no hay siguiente salto viable
 7       then
 8            return false
 9       else
10            mi nuevoactuales lo que indico siguiente_salto
11            aumento la cantidad de saltos en 1
12            agrego el nuevo tablón recorrido a la solución parcial tablones_recorridos
13            endif
14    endwhile
15   //si se sale del ciclo es porque se completo el recorrido del puente por lo que hay solucion viable.
16  return true
```

Para encontrar los óptimos locales utilizamos la funcion siguiente_salto. El algoritmo busca factibilidad del resultado y a la vez busca maximizar la variable que retorna, es por eso que

comenzamos probando desde el tablon mas lejano al que el participante llega y se van evaluando hacia atras los otros tablones hasta llegar a uno sano o al mismo desde donde se parte.

```
SIGUIENTE_SALTO(in actual: int, in puente: vector(int), in c: int, in tam: int)→ int
1  int res ← c
2  if llego a saltar fuera del puente con mi capacidad de salto
3    then
4          return actual + c
5          endif
6  while el tablon que reviso esta dañado y no llegue al actual
7    res − −
8    endwhile
9  return res + actual
```

1.3. Definiciones

- **Puente** estara representado por un vector de ints, identificandose cada tablón por su posición en el vector y su estado por el valor del int guardado, siendo **0** un tablón sano y un **1** un tablón dañado
- **Actual** será el tablón donde se encuentra el participante y **C** su capacidad de salto
- **Tablones** representa al conjunto de tablones sanos que tiene el **Puente** P (representado cada uno por su posición en P). Notesé que no agregamos los tablones dañados ya que nunca pueden ser elegidos como parte de una solución (esta es una estructura auxiliar que utilizaremos para la demostración de correctitud del algoritmo)
- **Siguiente_salto** evualúa los tablones en el intervalo entre **actual** y **actual + c**, para el propósito de la demostración de correctitud le agregaremos un comportamiento extra a esta función que es eliminar de *Tablones* todos los elementos estictamente menores al elegido (esto se puede ver facilmente en la ejecución del algoritmo ya que en la siguiente iteración se llama nuevamente a **Siguiente_salto** con el parámetro **actual + c** [1])
- **Tablones_ recorridos** sera la secuencia que irá guardando la solución parcial y una vez completada la ejecución demostraremos que contendra la solución óptima
- **P(tablones_recorridos S, tablón t)** será una función auxiliar que utilizaremos en la demostración de correctitud, indica si el tablón t que se le pasa es viable agregarlo a S con la condición que P(s,t) $\Leftrightarrow (t > x \wedge t \leq x + c, x = \text{máx}(s))$

1.4. Análisis de Correctitud

Nosotros sabemos que al final de la ejecución "tablones_recorridos" va a estar constituido por los saltos realizados (en el caso de que la instancia evaluada tenga solución), siendo estos representado por los tablones recorridos para cruzar el puente en orden ascendiente, que como mínimo va a existir 1 elemento en la solución (siendo este elemento mayor a todos los elementos que había en Tablones al comienzo de la ejecución) y que Tablones se va a encontrar vacío.
En el caso de que la instancia no tenga solución se verá luego durante la demostración que el algoritmo lo denotará apropiadamente.

Una solución óptima al final de la ejecución es una que tenga la mínima cantidad de elementos, al ser el conjunto de todas las posibles soluciones un conjunto finito habrá al menos una que cumpla esto y como se explicó previamente esta solución no puede ser nunca vacía.
Entonces como existe al menos una solución óptima tablones_Recorridos, o como será llamada de ahora en adelante S^{opt} que empieza siendo vacía, es una sub-solución de alguna solución óptima. Definiendo sub-solución como "prefijo de".

Considerando ahora una k-esima iteración del ciclo de la función, sabemos que al inciarse esta iteración, S^{opt} es sub-solución de alguna solución óptima. Sea S^* una solución óptima, tal que S^{opt} sea prefijo de S^* y sea x = siguiente_salto(actual, puente, c). En esta iteración correspondería agregar x a la solución. Llegado a este punto es cuando nuestro algoritmo evalúa si la solución que se va construyendo es factible y se presentan dos casos:

[1] líneas 5 y 10 del pseudocódigo **saltar_puente**

- $x == actual$
  ó
- $x >$ máx$(tablones) \wedge x \leq actual + c$

Si ocurre el primer caso significa que siguiente_salto no encontro un tablón mayor al actual y que se encuentre sano por lo que se finalizará la ejecución y retornará el "*no*" solicitado por el enunciado.

El otro caso indica que x es compatible con S^{opt} por lo que se continúa la ejecución.
Se forma S'^{opt} que resultará de agregarle x a S^{opt}, y queremos ver que existe una solución óptima S'^{*} tal que S'^{opt} sea prefijo de S'^{*}.
Se presentan dos opciones frente a esto:

- $x \in S^{*}$
  ó
- $x \notin S^{*}$

Si $x \in S^{*}$ entonces como S^{opt} es prefijo de S^{*}, S'^{opt} será prefijo de S'^{*} y podemos tomar a $S'^{*} = S^{*}$.

Por otro lado si $x \notin S^{*}$, al agregarle x a S^{*}, S^{*} dejaría de ser una solución óptima ya que le agregamos un salto innecesario y estamos buscando la solución con la mínima cantidad de saltos. Pero también sabemos que S^{opt} es prefijo de S^{*}, que vale P(S^{opt}, x) y que x fue elegido por *siguiente_salto*.
Considerando esto, si tomamos el primer elemento y de S^{*} / S^{opt}, osea el tablón elegido en la construcción S^{*}. No puede pasar que y sea menor o igual al mayor elemento de S^{opt} ya que S^{opt} es prefijo de S^{*} y S^{*} es solución óptima, luego P(S^{opt}, y) vale.
Entonces:

- $y > x$
  ó
- $y \leq x$

Si pasa que $y > x$, entonces como y es compatible con S^{opt}, siguiente_salto debería haber elegido y y no x ya que y sería una opción mejor, pero como no ocurrió este caso queda descartado.
Ahora si que $y \leq x$, sabemos que $y \neq x$ ya que habíamos dicho que $x \notin S^{*}$ y como también sabemos que S^{*} es solución óptima, oséa tiene la misma cantidad de saltos que S'^{*} podemos en S^{*} sacar y y agregar x.
Esto sigue siendo solución ya que si tomamos al elemento y_2 como el siguiente elementos despues de y en S^{*} sabemos que $y_2 \leq y + C$ y como $y + C \leq x + C$ por transitividad queda que $y_2 \leq x + C$, entonces se podría saltar al siguiente tablón de S^{*} si se remplaza a y por x y todavía S^{*} se mantendría solución óptima.
Hecho este remplazo se puede tomar entonces a $S'^{*} = S^{*}$.

Finalmente al terminar el ciclo se termina de cruzar el puente [2], en el caso de que exista una solución, y tenemos una S^{opt} que es sub-solución de alguna solución óptima, sea S^{fin} esa solución, si $S^{opt} = S^{fin}$ entonces S^{opt} es solución óptima.
$Tablones$ se encontrará vacío ya que si $actual$ ¿máx($Tablones$) implica que cuando siguiente_salto eligio ese actual eliminó todos los elementos del conjunto de $Tablones$.
Sino, esto significa que $\exists f \in S^{fin}$ / S^{opt}, $f \in Tablones$ o f ¿máx($Tablones$), P(S^{finf}, f) vale por lo tanto P(S^{opt}, f) vale. Pero esto no puede pasar, ya que al finalizar el ciclo habíamos dicho que $Tablones$ se encuentra vacía y el último elemento de S^{opt} es mayor a todos los elementos de Tablones ya que el último salto salió del puente, como S^{opt} es prefijo de S^{fin} entonces S^{fin} tiene un último salto innecesario fuera del puente ya que S^{opt} ya se encuentra fuera del puente y esto haría que S^{fin} no fuera solución óptima y eso es absurdo, luego f no existe y $S^{fin} = S^{opt}$.

Notemos que de nuestra demostración de correctitud podemos abstraer otra idea, que es que no es necesario tomar siempre el tablón mas lejano. Si en todos o en algunos casos se toma algún tablón anterior al más lejano todavía se puede conseguir una solución óptima. Esto es posible ya que no es necesario llegar al último tablón del puente para realizar el último salto fuera del puente, con estar a una distancia estrictamente menor a la cantidad de tablones que le quedan al puente + c ya se puede completar el último salto y esto pasa porque se puede salir del puente con cualquier numeración de tablón mayor al último tablón del puente (puede ser +1 o +10 y ambas valdrían por igual).

Nuestro algoritmo al elegir siempre el tablón mas lejano diponible siempre llega a una solución óptima si es que existe solución alguna y eso es lo que queríamos demostrar.

[2] se ve en la negación de la guarda del **while** del algoritmo, línea 4 del pseudocódigo **saltar_puente**

1.5. Análisis de Complejidad

Dentro del main iremos procesando, a medida que vamos leyendo a cada participante, resolviendo y devolviendo los resultados.
El procesamiento de cada instancia va a estar dado por las funciones saltar_puente() y siguiente_salto().

```
SIGUIENTE_SALTO(in actual: int, in puente: vector(int), in c: int, in tam: int)→ int
1  int res = c //O(1)
2  if actual + c ≥ tam //O(1)
3     then return actual + c //O(1)
4          endif
5  while res! = 0&&puenteactual + res == 1 //O(c)
6    res − − //O(1)
7    endwhile
8  return res + actual //O(1)
```

Complejidad final : O(c)

Como se puede apreciar en el codigo de siguiente_salto() la complejidad va a estar determinada por el **while** que verifica cuál es el tablón mas lejano, siendo esta verificación acotada por c y teniendo en cuenta que si la función determina con el **if** previo que se puede salir del puente con un salto lo hace sin evaluar el caso innecesariamente.
El peor caso para este algoritmo va a presentarse cuando res sea igual a 1 al final, ya que eso indica que el algoritmo tuvo que evaluar c veces el **while**; en ningún caso el algoritmo evaluará mas iteraciones ya que la guarda del **while** se lo impide.

```
SALTAR_PUENTE(in puente: vector(int), in c: int, in/out saltos: int, in/out tablones_recorridos: vector(int),
1  int actual = 0 //O(1)
2  int tam = c //O(1)
3  while actual ≤ n − 1 //O(n)
4    int whereTo = siguiente_salto(actual, puente, c, n) //O(c)
5    if whereTo == actual //O(1)
6       then return false //O(1)
7       else
8            actual = whereTo //O(1)
9            saltos + + //O(1)
10           tablones_recorridos.push_back(actual) //O(1)
11           endif
12   endwhile
13 return true //O(1)
```

Complejidad final : O(n)

Saltar puente va a llamar constantemente a siguiente_salto para ver donde se puede saltar y va a ir avanzando.

El peor caso para este algoritmo estará dado cuando el puente este construido con un patron de dos tablones sanos consecutivos y luego c-1 tablones rotos. Esto forzará a que siguiente_salto evalue iteración de por medio su peor caso y al finalizar la ejecucion de saltar_puente todo el puente haya sido evaluado, pero nunca el mismo tabón dos veces, por lo que la complejidad de peor caso queda O(n). El participante en este caso irá saltando el puente en un patrón de: salto de 1 tablon, salto de c tablones.

Otro caso extremo de este tipo donde se ve mas facilmente la complejidad es una capacidad de salto igual al tamaño del puente -1 y solo el primer tablón sano, en la primera iteración del ciclo va a revisar todos los tablones y va a saltar al primero y en la segunda ya sale del puente llegando al tablon "tamaño del puente-1, quedando nuevamente la complejidad O(n).

Notese que la complejidad no queda $n * c$ porqué nunca se revisa el mismo tablón mas de una vez, se irá evaluando el puente en intervalos de tamaño c y en el peor caso la evaluación costara c, pero luego de una evaluación con ese costo temporal vendrá una evaluación de costo O(1) ya que de lo contrario el puente no tendría solución.

Esto se da ya que una evaluacion de costo c avanza al participante 1 solo tablón porque revisa todos los tablones en ese intervalo y se queda con el siguiente inmediato como resultado, por ende en el proximo intervalo si el primer tablón a evaluar no es elegible, como los otros $c - 1$ tablones ya fueron evaluados y descartados por la iteración, el puente no tiene solución.

Encontramos también que el mejor caso no trivial, tomando al mejor caso trivial como aquel donde C es mayor al tamaño del puente y de un salto se recorre todo, tiene una complejidad igual a $O(n/c)$, ya que si se puede aprovechar siempre la capacidad de salto al maximo sin encontrarse con tablones dañados no es necesario evaluar todos los tablones del puente.

1.6. Experimentación

Para la experimentación decidimos analizar 3 posibles casos sobre nuestros algoritmos, mejor caso, peor caso y caso promedio. Para esto decidimos crear 3 funciones, las cuales se encargan de generar estos posibles escenarios, pero siempre respetando que tengan una solución válida ya que una instancia sin solucion va a terminar antes y no necesariamente se evaluaría todo el puente y asi generando irregularidades en las estadisticas a evaluar.

Luego nos encargamos de crear 3 scripts(uno por cada escenario) a los cuales se les pasa tamaño del puente inicial, el tamaño final, de a cuanto se va a incrementar las distintas instancias, la capacidad de salto del participante y la cantidad de repeticiones que se va a evaluar la instancia para conseguir el menor tiempo de ejecucion y asi evitar posibles outliers.

Para medir los tiempos de ejecución de nuestros algoritmos utilizamos la libreria "chrono.h" de c++, la cual nos suministró la posibilidad de generar cloks que nos ayuden a medir los tiempos de ejecucion de nuestro algoritmo en microsegundos.

Para esto le agregamos unas lineas extra a nuestro codigo:

```
high_resolution_clock reloj;

size_t minimo = 99999999;

auto t1 = reloj.now;

bool res = saltar_puente(puente, c, copia_saltos,
copia_tablones, n);

auto t2 = reloj.now;
auto total = duration_cast<microseconds>(t2 - t1).count();
if (total < minimo) minimo = total;
```

Para generar instancias aleatorias lo que hicimos fue utilizar la función **random()** que se encargó de generar numeros aleatorios a los cuales les evalua modulo 2, para obtener los posibles valor de 0 y 1, obteniendo asi tablones sanos o rotos respectivamente. Esto lo realizamos teniendo en cuenta que la solución podria devolvernos un puente que no tenga solución, es por eso que chequeamos si había mas de c tablones rotos consecutivos y corregimos el valor del c-esimo para que el puente tenga solución.

1.6.1. Experimento 1

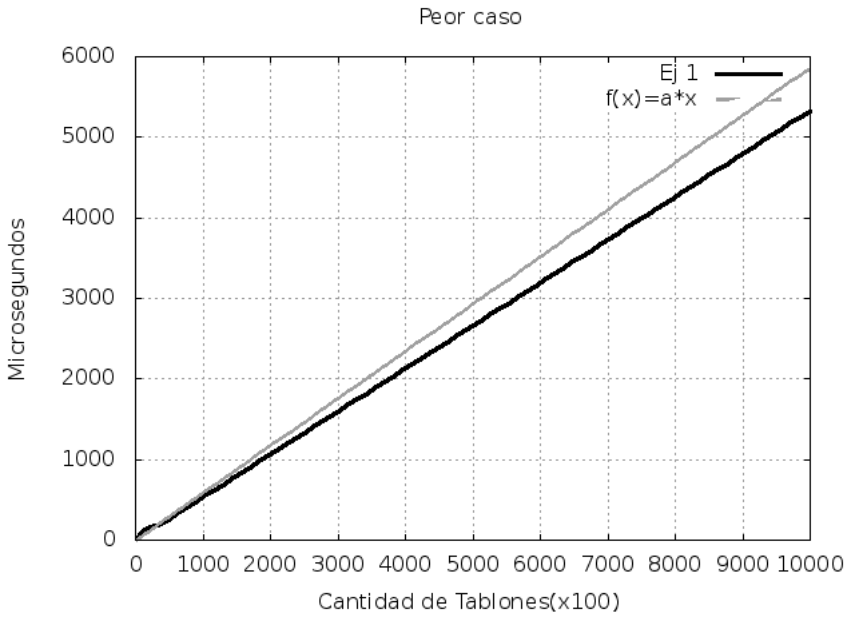
Nuestros primeros resultados de experimentacion distaban lejos de ser lineales como nuestro analisis de complejidad lo fue. Encontramos varios errores en nuestro codigo, algoritmicos y de uso de funciones del standar library de c++.

A nivel algoritmico nuestro problema surgia al pasar por copia los vectores que utiliza nuestra funcion, por otro lado, con guia de los ayudantes del laboratorio encontramos que nuestro uso de la clase vector estaba siendo mal aprovechada ya que al no tener en cuenta las reservas de memoria de la clase nos encontrabamos con complejidades mayores a las esperadas.
Como es de esperar con estos problemas, los graficos se acercaban mas a una funcion cuadratica que a una lineal.

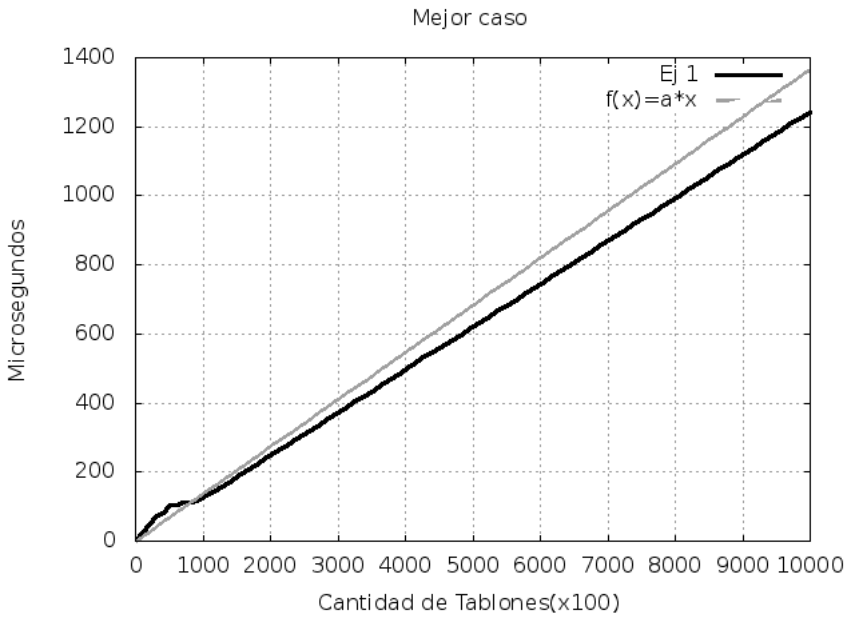
1.6.2. Experimento 2

Corregidos estos problemas se puede ver mas facilmente que la funcion es lineal.
Tomando nuestro generador de instancias separamos 3 casos para compararlos:

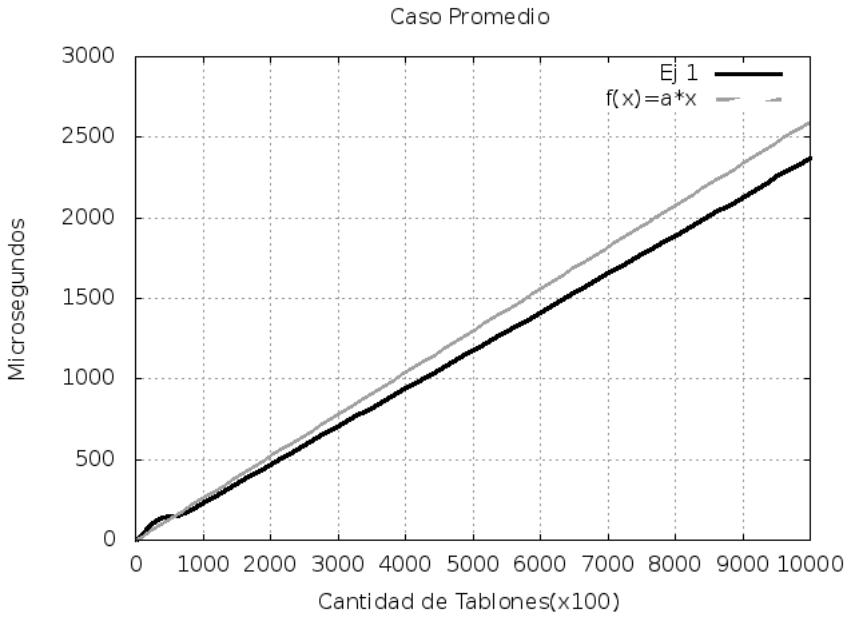
- Peor caso: el patron que van a seguir los tablones es: 2 tablones sanos, c-1 tablones rotos. Este patron fuerza a evaluar todo el puente y lo encontramos en nuestra evaluacion de complejidad cuando estabamos buscando el O() de la funcion.



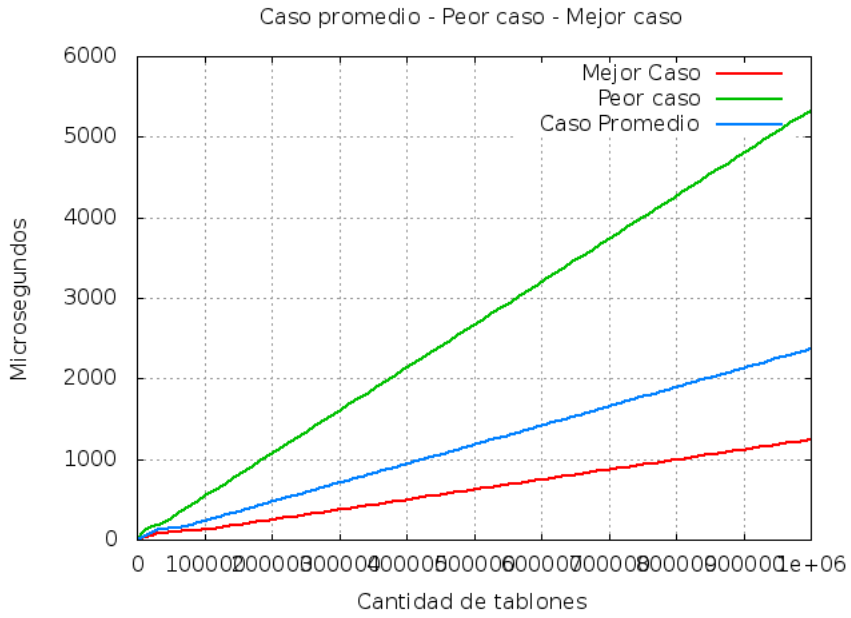
- Mejor caso: todos los tablones se encuentran sanos por lo que siempre se va a poder saltar la mayor cantidad posible y el algoritmo va a evaluar la cantidad minima de tablones.



- Caso promedio (o caso aleatorio): no intervenimos en la creacion de las instancias por lo que los casos son lo mas aleatorios posibles.



Finalmente para que la comparacion entre los 3 casos resulte mas facil de visualizar generamos el siguiente grafico con los resultados de los 3 casos:



1.7. Anexo reentrega

Entre la primera entrega y la segunda encontramos varios temas dentro de nuestro algoritmo que hacia que no cumpliera con nuestra complejidad teorica (ya previamente expuestos en la sección de experimentación de este problema). Resueltos estos temas si pudimos "linealizar" la funcion de forma esperada y asi demostrar que nuestra complejidad teorica era la esperada.
Veamos lo solicitado para la re-entrega:
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 verá incrementada en una unidad. Se pide desarrollar los siguientes puntos:

1. ¿Cómo afecta eso a su algoritmo? 2. ¿Qué característica del problema cambia en esta nueva versión?

Analizando los puntos nuevos sobre el ejercicio encontramos que:

1) Adaptar nuestro codigo al nuevo sistema de premios es bastante sencillo, como nosotros representamos el puente con un arreglo de 0 y 1, agregando el valor 2 cuando hay un tablón con premio y haciendo que la función "siguiente salto" modifique la capacidad de salto del participante cuando el tablón elegido es un tablon premiado es suficiente para tener en cuenta esta modificación. Sin embargo puede pasar que en con el algoritmo que definimos es posible que se retornen "falsos negativos", un "no" cuando con otra estrategia de elección podria haberse encontrado una solución ó, como analizaremos luego, soluciones sub-óptimas.

- Caso 1:
  Puente: 0 1 0 2 0 1 1 0
  Capacidad de salto: 2
- Caso 2:
  Puente: 2 0 2 0 2 0 2 0 2 0 2 0
  Capacidad de salto: 2

Con nuestra estrategia de elección de tablones el participante en el caso salta del tablón 1 al 3, luego al 5 (perdiendose el tablon con premio) y luego al no encontrar un salto viable se "rendiría" y la funcion devuelve un "no".
De haber escogido el tablón 4, que tenía premio, podría haber saltado los dos tablones rotos consecutivos y haber salido del puente.
Ahora si miramos al caso 2, se encuentra una solución factible pero no la óptima, ya que al priorizar siempre el tablón mayor se perderían todos los tablones premiados, mostramos a continuación una estrategia que si encuentra una solución óptima para este caso.

Esto nos lleva al siguiente punto.

2) Encontramos, buscando varias estrategias para optimizar la cantidad de saltos necesarios en tiempo lineal, que no solo se encuentran "falsos negativos" sino que aunque la estrategia de selección sea modificada, un algoritmo del tipo goloso en esta nueva situación dejaría de encontrar siempre una solución óptima.

Ya demostramos como nuestro algoritmo puede no encontrar una solución cuando si la hay, por eso queda descartada esa estrategia de elección de óptimos locales.

Ahora teniendo en cuenta los nuevos tablones podemos ver dos estrategías de selección de óptimos locales.

- Elegir el tablón premiado mas lejano y un candidato sano lo mas lejano posible (la llamaremos nueva_estrategia_1)
- Elegir el tablon premiado mas cercano y un candidato sano lo mas lejano posible (se cambiaría el código para que la búsqueda arranque en *actual* y vaya incrementando la variable *res* hasta llegar a $actual + c$ para que la complejidad siga siendo lineal) (la llamaremos nueva_estrategia_2)

Utilizando esta nueva_estrategia_1 podemos ver que por mas que se encuentre un buen candidato se sigue buscando entre los tablones hasta encontrar el tablon premiado mas lejano disponible. Esto resolvería el **caso 1** y si miramos el **caso 2** vemos que ese patrón (premiado - sano) hace que esta nueva estrategia encuentre mejores soluciones que nuestro algoritmo original (original encuentra solución de 6 saltos y nueva_estrategia_1 encuentra solución de 5 saltos) y que si se continúa el mismo patrón cuanto mas grande es el puente, mayor es la diferencia entre la cantidad de saltos requeridos por ambas estrategias para llegar al final.

- Ejecucion del Caso 2
  puente tamaño 10:
  capacidad de salto inicial: 2

  puente: 2 0 2 0 2 0 2 0 2 0
- Estrategia original:
  Tablones recorridos: 2 4 6 8 10 12
  Capacidad de salto: siempre 2
- Nueva estrategia 1:
  Tablones recorridos: 1 3 5 9 14
  Capacidad de salto: 2 3 4 5 5

Sin embargo si planteamos un nuevo caso podemos ver que esta estrategia tiene fallas al igual a la estrategia original.

- Caso 3:
  puente: 2 1 2 2 2 1 1 1 1 1
  capacidad de salto: 2

La nueva_estrategia_1 devuelve un "no" al llegar con capacidad de salto 5 a la secuencia final de tablones rotos y no poder saltarlo.

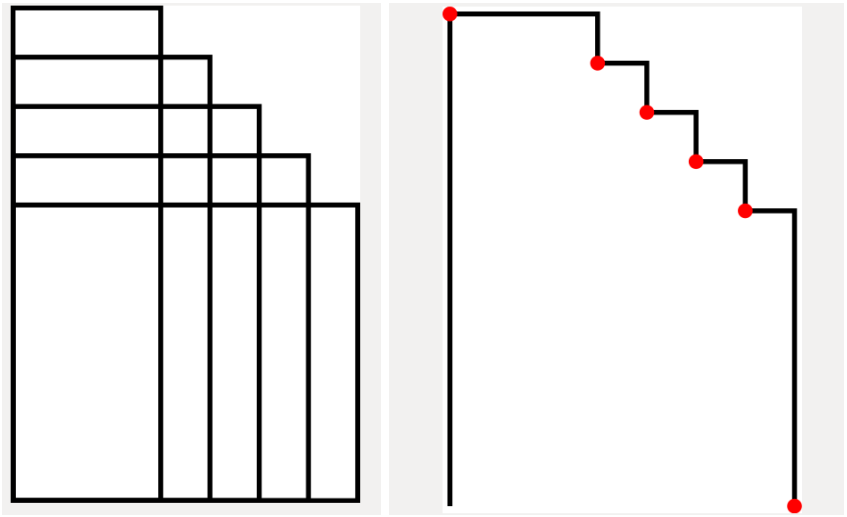
Pero volviendo a analizar el **caso 2** si vemos la nueva_estrategia_2, realiza la misma cantidad de saltos que la estrategia original ya que intenta pasar por todos los tablones premiados y existen la misma cantidad de tablones premiados que de tablones sanos.

Con lo presentado creemos que bajo las restricciones del problema no podemos presentar un algoritmo con complejidad lineal que siempre encuentre la solucion óptima al problema del "Puente sobre lava caliente", otras estrategias si podrían hacerlo (como un algoritmo de backtracking que pruebe todas las formas posibles de saltar el puente) pero no se cumpliría la complejidad solicitada.

2. Problema 2

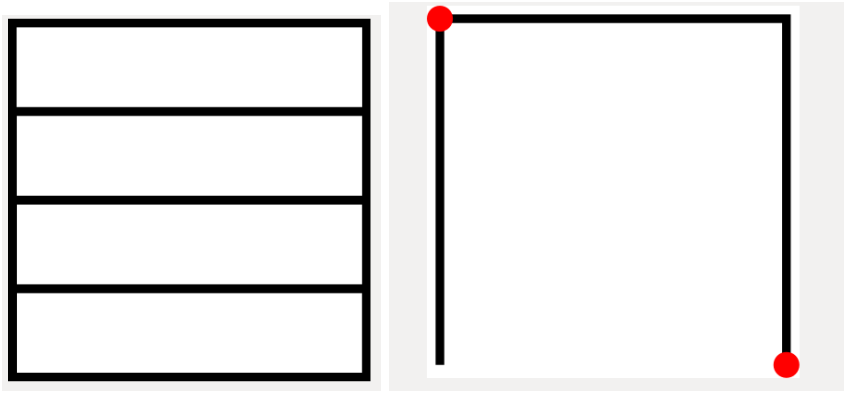
2.1. Descripción del problema

Este problema consiste en generar un contorno sobre un conjunto de edficios,eliminando las lineas que se superponen entre ellos.
Para esto contamos con la catidad de edificios que hay que procesar y con el conjunto de edificios a analizar. Estos edificios estan compuestos por tres componentes *izq alt der*, los cuales representan los márgenes y la altura del edificio.
Con respecto a la salida, esta debe mostrar los vértices, en los que se produzca un cambio de nivel entre edificios (siendo un vértice, un par de valores enteros x e y). Esta solución deberá estar ordenada con respecto a la coordenada x de cada vértice, es decir que se debe generar el contorno de izquierda a derecha.
Algunas posibles instancias son las siguientes:



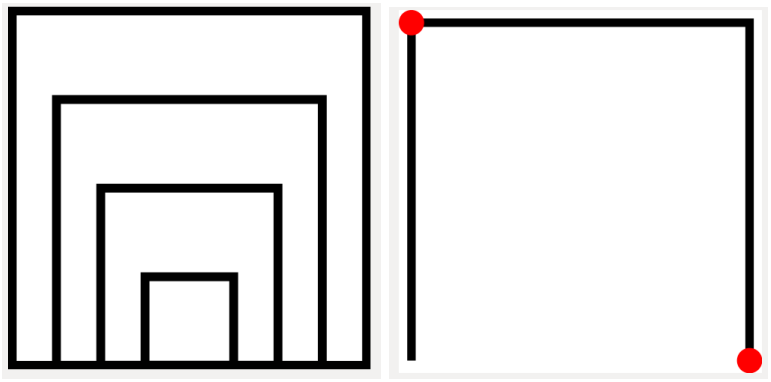
(a) Entrada (b) Resultado

Figura 1: Instancia : 5 20 11 5 18 13 5 16 15 5 14 17 5 12 19



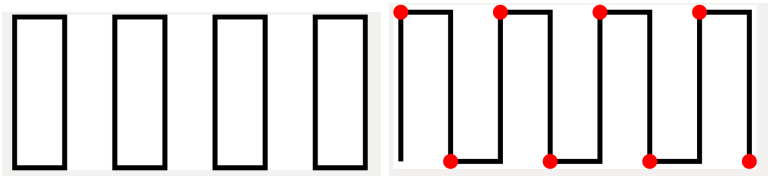
(a) Entrada (b) Resultado

Figura 2: Instancia : 4 2 8 10 2 6 10 2 4 10 2 2 10



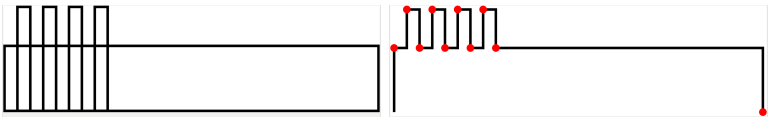
(a) Entrada (b) Resultado

Figura 3: Instancia : 4 1 8 9 2 6 8 3 4 7 4 2 6



(a) Entrada (b) Resultado

Figura 4: Instancia : 4 1 6 3 5 6 7 9 6 11 13 6 15



(a) Entrada (b) Resultado

Figura 5: Instancia : 5 1 5 30 2 8 3 4 8 5 6 8 7 8 8 9

Como ya dijimos, la solución del problema debe ser todos los puntos (desde ahora, vértices) donde se produce un cambio de nivel en el horizonte que forman los edificios. Entonces, ¿cuándo se producen dichos cambios de altura? Dichos cambios se dan cuando al seguir la línea generada por los edificios, o bien el edificio que tengo a mi derecha está pegado al que estoy viendo, o está separado por cierta distancia. Si está alejado, se dará un cambio de altura al bajar al piso cuando sigo por el margen derecho.

Si está pegado, entonces dependerá si este siguiente edificio es de la misma altura que el anterior o no. Si es de la misma altura, no se dará ningun cambio. Si, por el contrario su altura es mayor, el cambio de nivel será descripto por el vértice que forma la pared izquierda y la altura del edificio contiguo. En cambio, si la altura es menor, el vértice que deberá aparecer en el resultado será formado por el margen derecho del edificio anterior, y la altura del edificio siguiente.

Con un poco más de formalismo, si consideramos que un edificio consta de tres componentes x_1, y, x_2 (izq, alt y der), y un vértice consta de dos componentes **a, b** (coordenada eje x y coordenada eje y), diremos que la solución debe cumplir que:

$\forall\ r \in$ Resultado $\exists\ e\ edificio\ \in$ Entrada tal que:

- $(r_a = e_{x_1}) \wedge (r_b = e_y) \wedge (\nexists\ d$ edificio$\in$ Entrada tal que $d_y > e_y \wedge e_{x_1} \notin [d_{x_1}, d_{x_2}])$, o bien
- $(r_a = e_{x_2}) \wedge (\nexists\ q$ edificio $\in$ Entrada tal que $q_y > e_y \wedge e_{x_1} \notin [q_{x_1}, q_{x_2}]) \wedge (r_b = 0$ si $\nexists\ d$ edificio, $\vee\ r_b = d_y$, donde $d \in$ Entrada tal que $d_y < e_y \wedge e_{x_1} \in [d_{x_1}, d_{x_2})\ )$

2.2. Ideas desarrolladas

Para resolver este problema, luego de pensar otras soluciones como recorrer los edificios por eje x, llegamos a la conclusión de que la mejor idea es recorrer a los edificos por altura, pues es ésto en definitiva, la carácteristica principal del problema, dado que son los cambios de altura lo que nos marcará el cómo será el resultado.

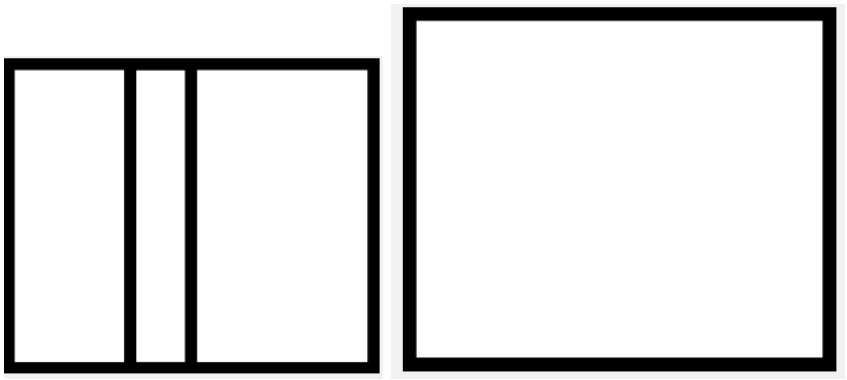
Dado un edificio del conjunto de entrada, nos interesa saber qué sucede con éste en relación a los demás. Es decir, tomando un edificio cualquiera, yo necesito saber si puedo agregar de manera segura a la solución su primer vértice (pues éste denota un potencial cambio de nivel) y ver qué hacer con el segundo vértice, cuyo x puede ser parte un futuro vértice del resultado al *chocar* con alguien de más abajo, o contra el piso. Para poder analizar ámbas cosas, necesito saber si existe o no un edifcio de mayor altura que me *haga sombra*, o sea, que el $\mathbf{x}_1$ de mi edificio no esté contenido en ningún intervalo formado por [x_1,x_2] de edificios más altos. Ésto es, a priori, muy costoso en términos computacionales, pues para verificar esta condición para un edificio, tendría que recorrer todo el conjunto de edificios, e ir compararando de a uno, teniendo *#edificios* comparaciones por cada edificio, obteniendo un total de *#edificios*^{2} comparaciones.

Teniendo ésto en cuenta, se debe establecer algún tipo de orden al conjunto de entrada para poder reducir la cantidad de comparaciones. Lo primero que se nos ocurrió fue la solución mas naive, ordenar a los edificios por eje x, y recorrerlos de forma creciente, pero nos topamos con un problema similar al de tener el conjunto desordenado. Potencialmente, un edificio se podía relacionar con todos los demás, y así sucesivamente. Resultaba difícil ver cómo dicho orden podía reducir significativamente la cantidad de comparaciones en el peor caso.

Al descartar el orden por eje x, quedaba entonces pensar cómo serviría usar un orden basado en el eje y.

Si el conjunto de edificios está ordenado por altura de forma decreciente, se puede asegurar que todos los edificios ya recorridos que pueden a llegar a relacionarse con el edificio que se está mirando son **estrictamente** mayores en altura, y los que me faltan recorrer que se relacionan son **estrictamente** menores. Y decimos **estrictamente** porque al momento de ordenar el conjunto de entrada, si dos edificios de la misma altura se relacionan (ésto es que están en contacto) los unimos, generando un edificio que empieze donde lo hacía el primero, y termine donde terminaba el segundo [3]. Es por ésto que usaremos un orden basado en las alturas para recorrer el conjunto de entrada. Un ejemplo mas ilustrativo seria el siguiente.

[3] Ésto lo podemos hacer porque el orden que establecemos si tienen igual altura es por x_1



(a) Edificios previo al merge (b) Edificios mergeados

Entonces, una vez que los edificios están ordenados, se recorren uno por uno. Al ir recorriéndolos, dado un edificio **e** tengo tres cosas para hacer:

1. Tengo que agregar el vértice **v** formado por e_{x_1} y e_y, si cumple que el **x** de **v** no esté contenido por ningún intervalo generado por un edificio de mayor altura. Para verficar ésto, tenemos una secuencia *techos* que tiene todos los intervalos que generaron los edificios en las iteraciones anteriores (dichos edificios son mayores, o iguales, pero si hay uno igual, el itervalo que generó no contiene a **x**). Si **v** no está contenido en ninguno de dichos intervalos, es parte de la solución, por lo consiguiente, lo agrego.
2. Debo ver si a e lo choca algún x_2 de un edificio más alto. Para ésto, necesito de una estructura auxiliar que tenga los distintos x_2 de los edificios que ya evalué. Si ésto sucede, debo agregar a la solución vértices (a, e_y), donde a son los x_2 de edificios mayores en altura que chocan con el que estoy mirando.
3. Por último, debo agregar a la estructura mencionada anteriormente el x_2 de e, si y sólo si, dicho valor no está contenido en la *sombra* de alguno de los edificios que ya evalué.

Una vez que todos los edificios hayan sido analizados, en la estructura auxiliar que guarda los x_2, me quedarán elementos (al menos uno, correspondiente al edificio con mayor x_2). Dichos elementos son los que *chocan* contra el suelo, pues no lo hicieron con ningún edificio. Entonces, debo agregarlos al resultado como vértices de aspecto $(x_2, 0)$.

Las estructuras que necesitaremos para implementar dicha idea son:

- **edificios**, una secuencia que tiene como elementos a los edificios ordenados de forma decreciente por altura, y en caso de igual altura, de forma creciente por x_1. Diremos que dicha estructura provee acceso en tiempo constante al próximo elemento a evaluar, y que tendrá los elementos que faltan analizar.
- **techos**, una secuencia ordenada crecientemente de intervalos, cuyos elementos representan los *espacios cubiertos* por los edificios ya analizados. Los elementos de esta estructura no se *solapan*, es decir, cualquier par de itervalos son disjuntos. Ésto nos facilitará la búsqueda de un elemento en la secuencia. La estructura es utilizada para saber si, dado un x, existe un itervalo que lo contenga, más especificamente, para ver si puedo o no agregar vértices al resultado, o los x_2 a *candidatos*. Usaremos una estructura que en tiempo logarítmico nos permita realizar una búsqueda.
- **candidatos**, una secuencia ordenada crecientemente de enteros, cuyos elmentos corresponden a los x_2 de los edificios ya evaluados tal que no existe intervalo en *techos* que lo contenga. Aquí también usaremos una estructura tal que la operación de busqueda tome tiempo logarítmico.
- **resultado**, una secuencia ordenada de vértices, donde cada elemento es un cambio de nivel. Necesitaremos una estructura que nos permita insertar elementos en tiempo logarítmico.

Como mencionamos anteriormente, la estructura **edificios** no contiene edificios de la misma altura que se solapen, ésto es: $\nexists\ e,\ d$ edificio $\in$ **edificios** con $e \neq d$ tal que:

$$e_y = d_y\ \wedge\ [e_{x_1}, e_{x_2}]\ \cup\ [d_{x_1}, d_{x_2}] \neq \emptyset$$

2.3. Definiciones

- **edificio**, tiene un margen izquierdo, una altura y un margen [4] derecho. Se representan con tres componentes x_1, y y x_2, que corresponden respectivamente a lo nombrado anteriormente.
- **vértice**, representa un punto en el plano, dado por el margen de un edificio y una altura. Tiene dos componentes **x** e **y**, formando (x, y).
- **primer vértice**, dado un edificio e, refiere al vértice generado por (e_{x_1}, e_y).
- **segundo vértice**, dado un edificio e, refiere al vértice generado por (e_{x_2}, e_y).
- **techo**, es análogo al concepto de intervalo, cuenta con dos componentes **a** y **b**, de forma que el techo es (a, b). Los techos son bien formados, es decir, $a < b$.
- **techo de un edificio**, dado un edificio e, su techo es (e_{x_1}, e_{x_2}).
- **solapar (techos)**, el techo **i** se solapa con el techo **j** si:
  - $i_a \leq j_a \leq i_b$, o bien
  - $j_a \leq i_a \leq j_b$
- **solapar (edificios)**, el edificio **e** se solapa con el edificio **d** si:
  - $e_{x_1} \leq d_{x_1} \leq e_{x_2}$, o bien
  - $d_{x_1} \leq e_{x_1} \leq d_{x_2}$
- **un techo i contiene a un techo j** cuando $i_a \leq j_a < j_b \leq i_b$
- **un elemento x pertenece a un techo i** si $i_a \leq x \leq i_b$
- **un elemento x pertenece *estrictamente a un techo i*** si $i_a \leq x < i_b$
- **fusionar (techos)**, fusionar dos techos **i,j** significa que, si éstos se solapan se crea un nuevo techo **k** tal que:
  - $k_a = min(i_a, j_a)$
  - $k_b = max(i_b, j_b)$
- **fusionar (edificios)** [5], fusionar dos edificios **e,d** significa que, si éstos se solapan se crea un nuevo edificio **f** tal que:
  - $f_{x_1} = min(e_{x_1}, d_{x_1})$
  - $f_y = e_y$
  - $f_{x_2} = max(e_{x_2}, d_{x_2})$

[4] Nos referiremos a margen y a pared como a la misma cosa.
[5] Notar que cuando se fusionan edificios, éstos son de la misma altura

2.4. PseudoCódigo

A continuación describiremos dos pseudocódigos, el primero corresponde al algoritmo que toma el conjunto de edificios de entrada y genera una secuencia de edificios ordenándolos por altura como primer criterio, y por x_1 como segundo. El segundo pseudocódigo corresponde al algoritmo que resuelve al ejercicio. Incluimos el primero porque no es trivial lo que hacemos al generar la estructura, pues afecta al análisis de la complejidad y a la demostración de correctitud del ejercicio.

```
generarEdificiosOrdenados(in edificios: Conjunto(edificios)) → Secuencia(edificios)
1  secuencia(edificios) resultado ← vacia
2  for e ∈ edificios
3    if ∃ d ∈ resultado, tal que e_y = d_y ∧ d se solapa con e
4      then fusionar todos los edificios que se solapan con e y agregarlafusion
5      else agragar e a resultado
6    endif
7  endfor
8  return resultado
```

```
generarHorizonte(in/out edificios_desordenados: Conjunto(edificios)) → Secuencia(vertices)
 1  Secuencia(techo) techos ← vacia
 2  Secuencia(pared_derecha) candidatos ← vacia
 3  Secuencia(vertices) resultado ← vacia
 4  Secuencia(edificio) edificios ← generarEdificiosOrdenados(edificios_desordenados)
 5  while edificios ≠ vacia
 6    edificio e ← primero de edificios
 7    eliminar primero de edificios
 8    if e_{x_1} no pertenece a algun techo
 9      then agregar(e_{x_1}, e_y) a resultado
10    endif
11    for i ∈ candidatos, tal que e_{x_1} ≤ i < e_{x_2}
12      agregar (i, e_y) a resultado
13      eliminar i de candidatos
14    endfor
15    if e_{x_2} no pertenece estrictamente a algun techo
16      then agregar e_{x_2} a candidatos
17    endif
18    techo nuevo ← (e_{x_1}, e_{x_2})
19    if existe algun techo tal que se solapa con nuevo
20      then fusionar todos los techos que se solapan con nuevo y agregarlafusiom
21      else agregar nuevo a techos
22    endif
23  endwhile
24  for c ∈ candidatos
25    agregar (c, 0) a resultado
26  endfor
27  return resultado
```

2.5. Análisis de Correctitud

Veamos por qué nuestro algoritmo resuelve el problema planteado.

A riesgo de ser repetitivos, como planteamos anteriormente, ordenamos los edificios por altura (sin solapamientos en igual altura) y los recorremos de mayor a menor, guardandonos los techos que generan los edificios, que representan los sectores del eje x que ya no están *disponibles*. Entonces, dado un edificio, sé que su primer vértice será resultado si no hay un edificio de mayor altura que le haga sombra, lo cual es verificar en **techos** si la pared izquierda pertenece a alguno de sus elementos; si no lo hace, lo agrego a la solución, de otra forma, lo descarto. Ésto funciona porque yo sé que en **techos** tengo *todos* los techos que fueron formando los edificios que ya revisé, y que por consiguiente, son de mayor altura [6].

Por otro lado, al revisar las posibles soluciones que están en **candidatos** me aseguro de generar todos los vértices que describen cambio de nivel que tienen la altura del edificio que estoy analizando y que están en el intervalo que generan mis dos márgenes (izq y der). Ésto es así porque dicha estructura tiene todos las paredes derechas de los edificios que ya revisé (o sea, que son mayores) tal que no se solapaban con ningún edificio más alto (pues antes de agregarlo a candidatos se verifica el solapamiento con **techos**), y sé que tienen que llevar esta altura y no una mayor, pues si fuera ese el caso, hubieran *chocado* con un edificio de una iteración previa.

Por último, veamos que los elementos que quedaron en **candidatos** al terminar todas las iteraciones deben llevar altura 0 y no una mayor. Si tuvieran que tener una altura distinta a 0, entonces en alguna iteración del ciclo principal, el intervalo formado por algún edificio los hubiera contenido, agregándolo como resultado. Al no ser así, no existe ningún edificio que *haga de piso* para dichos márgenes [7], entonces la altura que les corresponde es la del piso, es decir 0.

Es por ésto que el algoritmo funciona. Pero veámoslo formalmente.
En la descripción del problema ya definimos de manera formal cómo son los elementos que están en el resultado, y cómo están formados con respecto a la entrada. Pero antes de abordar el algoritmo *generarHorizonte*, veamos que efectivamente *generarEdificiosOrdenados* sea correcto.

2.5.1. Demostración de Correctitud de generarEdificiosOrdenados

Dijimos que queríamos ordenar al conjunto de entrada de tal forma que, no sólo estén de forma decreciente por altura, si no que los edificios de igual altura que se solapan queden *fusionados* en uno solo. Queremos ver que se cumple:

- $\nexists\ e,\ d$ edificio $\in$ **edificios_ordenados** con $e \neq d$ tal que:
  $e_y = d_y\ \wedge\ e$ y d se solapan.

Vemos que antes de comenzar el ciclo, **edificios_ordenados** es una secuencia vacía, con lo cual, se cumple que no existen dos edificios de igual altura solapados.

Digamos entonces que en la iteración k se cumple que:

- $\nexists\ e, d$ edificios $\in$ **edificios_ordenados** con $e \neq d\ \wedge e_y = d_y$ tal que e y d se solapan.

Ahora, queremos ver que al agregar un edficio nuevo e, si se solapa con algún edificio (puede ser más de uno), se fusionan creando un edificio que es la unión de ambos, quedando **edificios_ordenados** sin elementos que se solapen, siendo válido el predicado propuesto.

Al evaluar un edificios nuevo e, se pueden dar dos situaciones:

1. $\nexists\ d,\ edificio\ \in$ **edificios_ordenados** *tal que* $e_y = d_y \wedge e$ y d se solapan. O bien,
2. $\exists\ d\ edificio\ \in$ **edificios_ordenados** tal que $e_y = d_y \wedge e$ y d se solapan.

Si se da el caso (1), agrego el elemento y sé que no se solapa con nadie de su misma altura. Entonces vale el invariante.

En cambio, si se da el caso (2), existe al menos un edificio d tal que $e_y = d_y$ y se solapan. El algoritmo antes de agregar al elemento lo fusiona con d, de forma tal que ahora hay un edificio $f = (min(e_{x_1}, d_{x_1}), e_y, max(e_{x_2}, d_{x_2})$. Si existe más de un elemento, el algorimo los fusiona a todos. De esta forma, el nuevo elemento a agregar no se solapa con ningún edificio ya agregado que tenga su misma altura.

Entonces, al darse cualquiera de las dos situaciones descriptas, el algoritmo agrega el elemento de manera correcta, manteniendo lo pedido sobre la estructura, pues agrego un edificio que cumple con lo pedido, y **edificios_ordenados** cumplía el invariante antes de agregar al nuevo edificio.

[6] Aunque podría haber de igual altura, pero éstos no interesan, pues nunca pueden solaparse dos edificios de igual altura.
[7] Recordar que los valores de la secuencia representan los márgenes derechos de los edificios.

2.5.2. Demostración de Correctitud de generarHorizonte

Pasemos ahora a analizar la correctitud de **generarHorizonte**. Primero establezcamos invariantes sobre las estructuras usadas y veamos que dichos predicados se preservan al terminar el ciclo del algoritmo.

- **techos**: Sólo tiene techos que representan todos los techos generados por los edificios analizados en iteraciones previas, es decir, para cada techo t de un edificio ya analizado, existe un techo v en la estructura tal que v *contiene* a t. Todos los elementos en techos no se solapan entre sí.
- **candidatos**: Tiene todos los x_2 (paredes derechas) de los edificios analizados en las iteraciones previas, tal que no existe elemento en techos que contenga *estrictamente* a alguno.
- **edificios**: Tiene todos los edificios que me faltan analizar.
- **resultado**: Tiene todos los primeros vértices de los edificios ya analizados tal que no existía (al momento de agregar dicho vértice v) techo de forma tal que v_x perteneciese a dicho techo. También tiene todos los vértices formados a partir de las paredes derechas (que no se solapaban con un techo cuando se agregaron a candidatos), y con la altura del primer edificio (menor) cuyo techo contiene a dicha pared derecha.

Para establecer la correctitud del algoritmo, vamos a ver que primer ciclo respeta los invariantes declarados para cada estructura. Pero antes tenemos que ver que al entrar al ciclo, dichas estructuras respeten los invariantes.

Antes de comenzar el ciclo:

- **techos**: Como no analize ningún elemento, techos es vacío.
- **candidatos**: Al no analizar ningún edificio, no hay ningún candidato.
- **edificios**: Como todavía no evalué algún edificio, la estructura tiene todos los edificios.
- **resultado**: No tiene ningún cambio de altura, pues no se evaluó edificio alguno.

Ahora veamos que para cada estructura, en la iteración k, analizando el edificio e el ciclo preserva los invariantes :

- **resultado**

  Al querer agregar el primer vértice de e [8] se pueden dar dos casos:

  1. $\exists$ t *techo* $\in$ techos tal que e_{x_1} pertenece a t. O bien,
  2. $\nexists$ t *techo* $\in$ techos tal que e_{x_1} pertenece a t.

  Si se da el caso (1), el vértice no se agrega a **resultado**, con lo cual se mantiene su invariante, pues no se modificó.

  Si por el contrario, se da el caso (2), se agrega el vértice a **resultado** cumpliendo (2), que es exactamente lo que establece el invariante de la estructura, con lo cual, se mantiene el invariante.

  Cuando se revisan los elementos de **candidatos** [9], hay dos opciones, o no agrego ningún vértice a **resultado** porque ningún candidato pertenece al techo de e, o bien hay uno o más. Ésto es:

  1. $\nexists$ $c \in$ candidatos tal que $e_{x_1} \leq c < e_{x_2}$. O bien,
  2. $\exists$ $c \in$ candidatos tal que $e_{x_1} \leq c < e_{x_2}$.

  Si estamos frente al caso (1), **resultado** no se modifica, con lo cual su invariante se mantiene. Si no, entonces existen paredes derechas en **candidatos** tal que *pertenecen estrictamente* al techo de e. Éstas son agregadas al resultado como vértices formados por tales valores y por la altura de e. Llamemos S al conjunto formado por dichos candidatos, $S = \{c \in$ candidatos tal que $e_{x_1} \leq c < e_{x_2}\}$. Para cumplir el invariante de **resultado** debo ver que para cada $s \in S$:

[8] Líneas 7-9 del pseudocódigo.
[9] Líneas 10-13 del pseudocódigo.

- s es una pared derecha de un edificio analizado previamente.
- La altura con la que se agrega al resultado es la del primer edificio menor [10] cuyo techo contiene a s.

Ver que el primer ítem es verdadero es sencillo, pues si s es un elemento de S, lo es de **candidatos** (por definición de S), entonces son paredes derechas de edificios analizados en iteraciones previas (por el invariante de dicha estructura).

Veamos el segundo ítem. Si la altura de e no fuese la mayor de los edificios cuyos techos contienen a s, entonces habría un edificio d con $d_y > e_y$ tal que s *pertenecería estrictamente* al techo de d. Y d hubiese sido analizado en una iteración previa (pues **edificios** está ordenado por altura), y como el techo de d contendría a s, éste hubiese sido agregado al resultado y eliminado de **candidatos**, entonces no existiría en candidatos, lo cual es absurdo porque están en la iteración correspondiente a e, que es posterior a la de d. Por lo tanto, se cumple el segundo ítem.

Luego, se cumple el invariante de **resultado**.

- **techos**

Quiero ver que, al agregar el techo de e, éste no se solape con ningún otro techo. Y que lo que quede en **techos** represente todos los techos que ya estaban, junto con el nuevo techo.

Al momento de agregar el techo *nuevo* [11] que forma e, se pueden dar dos casos:

1. $\nexists\ t \in$ techos tal que t y *nuevo* se solapan.
2. $\exists\ t \in$ techos tal que t y *nuevo* se solapan.

Si se da (1) se agrega *nuevo* a **techos** cumpliendo 1 y *nuevo contiene* al techo de e (pues es el techo de e), y al no modificar los elementos que ya estaban, como vale el invariante estos elementos representaban a los techos de los edificios ya iterados, con lo cual se respeta el invariante.

Si se presenta el caso (2), veamos qué pasa. Llamemos T al conjunto de techos que se solapan con *nuevo*, es decir $T = \{t \in$ techos tal que t y *nuevo* se solapan $\}$. Quiero ver que al agregar *nuevo* a **techos**, T queda vacío y la fusión de todos *contenga* a dichos t y a *nuevo*.

Justamente, el algoritmo [12] fusiona a cada t con *nuevo*, eliminando dichos elementos que solapan, quedando entonces un nuevo techo que es la fusión de todos los t y *nuevo*. Entonces al momento de agregar *nuevo* a **techos** T es vacío, pues si no lo fuese, entonces habría algun t para fusionar con *nuevo*, pero el algoritmo los fusiona a todos. Y la fusión de todos los t con *nuevo* contiene a todos los t más *nuevo* (por definición de *fusión*). Entonces, tenemos la fusión que contiene a los t más el nuevo y que no se solapan con ningún elemento que quedó en **techos**. Luego, agregar la fusión es correcto, porque cumple con lo pedido en el invariante, y los elementos que quedaron en la estructura cumplen el invariante (pues éste valía al entrar al ciclo).

Entonces, en ambos casos, al agregar *nuevo* a **techos**, no existe ningún techo tal que se solapen y los elementos de la estructura representan a los techos de los edificios ya iterados, más el nuevo. Luego, se cumple el invariante de la estructura.

- **candidatos**

Para ver que el invariante de la estructura se preserva, hay que ver dos cosas:

1. Al eliminar elementos de la estructura, éstos pertenecen *estrictamente* al techo de e, pues a un elemento de techos no pueden pertenecer, porque si lo hubiese contradeciría al invariante. [13]
2. Al agregar e_{x_2}, éste no pertenece *estrictamente*a ningún techo. Si perteneciese *estrictamente* a algún techo y lo agregase, estaría contradiciendo el invariante. [14]

[10] O sea, el mayor de los menores.
[11] Líneas 17-21 del pseudocódigo.
[12] Línea 19 del pseudocódigo.
[13] Línea 10-13 del pseudocódigo.
[14] Línea 14-16 del pseudocódigo.

Ver (1) es sencillo, pues los elementos de **candidatos** que se eliminan son los que pertenecen *estrictamente* al techo de e. Si no los eliminase, al agregar el techo de e, estaría contradiciendo el invariante.

Ver (2) también es sencillo, ya que e_{x_2} sólo se agrega si no pertenece *estrictamente* a algún techo.

De ésta forma, en **candidatos** están todos las paredes derechas de los edificios tal que no existe elemento en techos que contenga *estrictamente* a alguno, porque antes de agregar el techo de e, valía el invariante, y al agregarlo también lo vale, pues los elementos que pertenecían *estrictamente* al techo de e fueron eliminados, y aquellos que no siguen estando. Y la pared derecha de e se agrega, si no pertenece *estrictamente* a algún techo.

Luego, se preserva el invariante de **candidatos** a lo largo de la iteración.

- **edificios**

  Ver que éste invariante se cumple es trivial, pues al hacer *eliminar primero de edificios*, estoy eliminando al elemento que evalúo, quedando sólo los edificios que me falta evaluar después de ésta iteración.

De ésta forma vemos como el ciclo preserva los invariantes de las estructuras. Es decir, que al terminar el ciclo, en las estructuras tenemos exactamente lo que dijimos tener. Ésto es, en **resultado** tenemos todos los elementos formados como (e_{x_1}, e_y) que deben ir, y otro elementos formados como (e_{x_2}, d_y), con $d_y < e_y$. Sin embargo, no hay ningún elemento construido como $(e_{x_2}, 0)$. Dichos elementos corresponden a todos los edificios cuyos márgenes derechos bajan hasta el suelo, y son los que están en **candidatos** al terminar el primer ciclo. Veamos por qué ésto es así.

Los elementos que quedan en **candidatos** al finalizar el ciclo principal corresponden a paredes derechas de edificios que no tenían un edificio mayor que les haga sombra, pues si hubiese habido uno, entonces habría habido un intervalo i en **techos** (al momento de ser agregado) tal que $x_2 \in (i_a, i_b)$, luego, dicho x_2 no se hubiese agregado a **candidatos**.

Tampoco hubo un edificio d de menor altura tal que dichos elementos pertenezcan al techo de d, ya que de haber existido hubiese sido agregado como solución y quitado de **candidatos** [15].

Entonces, al no chocar con ningún edificio, chocan contra el piso. Con lo cual les corresponde ser agregados como solución con altura 0. [16].

Veamos cómo se relaciona **resultado** con respecto a lo pedido en la descripción del problema.

Habíamos dicho que el resultado del algoritmo tenía que cumplir que:

$\forall\ r \in$ Resultado $\exists\ e\ edificio\ \in$ Entrada tal que:

- $(r_a = e_{x_1}) \wedge (r_b = e_y) \wedge (\nexists\ d$ edificio$\in$ Entrada tal que $d_y > e_y \wedge e_{x_1} \notin [d_{x_1}, d_{x_2}])$, o bien
- $(r_a = e_{x_2}) \wedge (\nexists\ q$ edificio $\in$ Entrada tal que $q_y > e_y \wedge e_{x_1} \notin [q_{x_1}, q_{x_2}]) \wedge (r_b = 0$ si $\nexists\ d$ edificio, $\vee\ r_b = d_y$, donde $d \in$ Entrada tal que $d_y < e_y \wedge e_{x_1} \in [d_{x_1}, d_{x_2})$)

Y sabemos que (por invariante): **resultado** tiene todos los primeros vértices de los edificios tal que no existía (al momento de agregar dicho vértice v) techo de forma tal que v_x perteneciese a dicho techo. También tiene todos los vértices formados a partir de las paredes derechas (que no se solapaban con un techo cuando se agregaron a candidatos), y con la altura del primer edificio (menor) cuyo techo contiene a dicha pared derecha.

Y además, resultado ahora tiene todos los vértices formados por paredes derechas, tal que no existe techo en **techos** de forma tal que dichas paredes pertenezcan *estrictamente* a ese techo, y por 0 como segunda componente.

Entonces, vemos que resultado tiene los primeros vértices de edificios tal que no existe edificio de mayor altura cuyo techo solape al otro, porque si hubiese existido tal edificio, su techo hubiera estado en la estructura **techos** (por invariante de techos) y no se hubiera agregado. Entonces todos los primeros vértices cumplen con el primer ítem de la caracterización del problema.

Por otro lado, vemos que resultado tiene los vértices formados por paredes derechas y (si existe) la altura del primer edificio menor a cuyo techo pertenece dicha pared. Y éste edificio es el de mayor altura de los de menor altura, pues **edificios** está ordenado. Y si no existe dicho edificio, los vértices se agregan con $y = 0$. En cualquier caso, se cumple el segundo ítem.

Damos así por concluida la demostración de correctitud de generarHorizonte.

[15] Este comportamiento corresponde a las líneas 9-12 del pseudocódigo de generarHorizonte.

[16] Ésto se realiza en las líneas 23-25 del pseudocódigo.

2.6. Detalles de implementación

A continuación describiremos algunos detalles sobre la implementación, para facilitar la comprensión del código con respecto al pseudocódigo. Y para esclarecer la complejidad de las operaciones realizadas.

- **techos** está implementado con un *set*, que nos provee tiempo logarítmico en insersión, búsqueda y borrado.
- **resultado** también está implementado con *set*, con el objetivo de no tener repetidos, mantener la estructura ordenada, e insertar en tiempo logarítmico.
- **candidatos** está implementado también como un *set*, pues el propósito es mantener la estructura ordenada para luego buscar en tiempo logarítmico.
- **edificios** está implementado con *set* también, por las razones ya mencionadas. Sin embargo, sólo se usa para recorrer los edificios de forma ordenada, sin importar nada más que eso. Ya que una vez inspeccionado un edificio no se accede a él nunca más, y nos interesa siempre evaluar el que está primero en la secuencia.

El set al que se hace referencia, es el contenedor provisto por la librería estandar de $C++$.

En el código se utiliza la herramienta *auto* para la deducción automática de tipos, sobre todo con los iteradores, para que el código tenga un aspecto más amigable.

2.7. Análisis de complejidad

El análisis de la complejidad lo vamos a hacer con respecto al pseudocódigo presentado anteriormente, pero considerando las estructuras sobre las cuales están implementadas.

Primero, veamos **generarEdificiosOrdenados**:

El algoritmo cuenta de un *for* que itera #Edificios veces. Y en cada iteración se procede a, o bien agregar el edificio sin mayor recaudo, o agregarlo fusionando todos con los que se solapa. A priori, uno podría pensar que en cada iteración se fusiona un edificio con todos los que ya están agregados, dando la percepción de ser un algoritmo cuadrático en la cantidad de edificios. Establezcamos entonces una fórmula que nos permita analizar su comportamiento:

$\sum_{i=1}^{n}(log(n_i) + m_i)$

Siendo n la cantidad de edificios, n_i la cantidad de elementos en el **resultado**, m_i la cantidad de edificios que se fusionan en la iteración i. El $log(n_i)$ es costo de buscar al primer edificio (de igual altura) que se solapa, dado que está ordenado por altura y luego por x_1, y no hay solapamientos, una vez encontrado el primero, se procede linealmente a buscar el resto. El costo de la fusión es constante.

Como m es la cantidad de elementos que se fusionan, dicho valor no puede ser mayor a la cantidad total de elementos existentes, es decir $m \leq n$. Y como al fusionarlos, se eliminan los edificios, $n_i = n_i - 1 + 1 - m_{i-1}$.

Entonces, en la iteración $i + 1$, valdrá que: $m_{i+1} \leq n_{i+1} = n_i + 1 - m_i$.

Es decir que la cantidad máxima de fusiones de una iteración, depende de las anteriores. No sólo eso, si no que en el peor caso (si todos los edificios tienen la misma altura y se solapan), en la totalidad de las iteraciones, voy a fusionar a todos los edificios.

Separemos la sumatoria:

$\sum_{i=1}^{n} log(n_i) + \sum_{i=1}^{n} m_i$

Por lo dicho anteriormente, podemos acotar al segundo término por n, pues a lo sumo la cantidad de fusiones es n en todas las iteraciones (pues hay n edificios).

$\sum_{i=1}^{n} log(n_i) + n$

Luego, como $n_i \leq n$, podemos acotar al primer término por $n * log(n)$, quedando como complejidad:

$O(n * log(n) + n)$, que es $O(n * log(n))$.

Veamos ahora la complejidad de **generarHorizonte**.

El algoritmo tiene 2 ciclos principales, analzaremos el primero.

Que la guarda del ciclo sea $edificios \neq vacia$ significa que habrá n iteraciones (siendo n la cantidad de edificios), pues en cada iteración se saca un elemento, y nunca se agregan edificios.[17]

Ver si e_{x_1} pertenece a algún techo cuesta $log(t_i)$, siendo t_i la cantidad de elementos en **techos**. Ésto es así porque la búsqueda toma tiempo logarítmico, y como la estructura no tiene solapamientos, una sola búsqueda basta para chequear la condición. Agregar el vértice al resultado también toma tiempo logarítimico, siendo entonces $log(t_i) + log(n_i)$ (siendo n_i la cantidad de edificios revisados en la i-ésima iteración).[18]

Nos entontramos ahora con un *for* [19] que busca todos los candidatos que *pertenecen estrictamente* al techo de e. Como los candidatos están ordenados, basta con buscar al primer elemento que cumpla la condición, y de ahí en más chequear linealmente hasta que no haya más paredes derechas que cumplan la condición. Luego, el agregado a resultado cuesta $log(n_i)$, entonces, para este for tenemos que su complejidad es:

$$log(c_i) + \sum_{j=1}^{d_i}(log(n_i))$$

Donde c_i es la cantidad de elementos en **candidatos**, d_i es la cantidad de candidatos que cumplen la condición dada, y n_i es la cantidad de edificios, en la iteración i-esima.

Lo que es lo mismo que:

$$log(c_i) + d_i * log(n_i)$$

El if que verifica si e_{x_2} *pertenece estrictamente* a algún techo tiene costo logarítimico, pues es buscar en **techos** el posible techo y nada más. Y luego agregar el margen derecho a **candidatos** tiene también costo logarítimico, siendo el costo total del if:

$$log(t_i) + log(c_i)$$

Siendo t_i la cantidad de techos y c_i los candidatos en la i-ésima iteración.

Para el último if tenemos una sentencia que dice *fusionar todos los techos que se solapan con nuevo y agregar la fusion*, que no es más que un ciclo implícito, pues cada elemento en techos que se solapa con nuevo se fusiona. El costo de fusión lo asumimos constante (pues es calcular el mínimo y máximo entre dos valores), el costo para encontar al primer elemento que cumple la condición vemos que es logarítmico (análogamente a lo explicado para **candidatos**); y como **techos** es una estructura ordenada y sus elementos no se solapan, basta con buscar el primero, y luego recorrer linealmente para encontrar todos los que se fusionan. Luego se eliminan los elementos fusionados, y se agrega la fusión de todos. Para eliminar todos los elementos fusionados se utiliza una función de borrado que es lineal en la cantidad de elementos a borrar más el logaritmo de la cantida de elementos. [20] Entonces tenemos que la complejidad para éste if es:

$$log(t_i) + u_i + log(t_i)$$

Siendo t_i el tamaño de **techos** en la iteración i y u_i los techos que se fusionan.

Entonces, la complejidad del ciclo principal es la suma de éstos ítems:

1. $\sum_{i=1}^{n}(log(t_i) + log(n_i))$
2. $\sum_{i=1}^{n}(log(c_i) + d_i * log(n_i))$
3. $\sum_{i=1}^{n}(log(t_i) + log(c_i))$
4. $\sum_{i=1}^{n}(2 * log(t_i) + u_i)$

Si acotamos a n_i, c_i, y t_i por n nos queda:

1. $n * log(n)$
2. $n * log(n) + \sum_{i=1}^{n} d_i * log(n_i)$
3. $n * log(n)$
4. $n * log(n) + \sum_{i=1}^{n} u_i$

[17] Línea 5 del pseudocódigo.
[18] Líneas 7-9 del pseudocódigo.
[19] Línea 10-13 del pseudocódigo.
[20] La utilización de *erase* de *set* con dos iteradores, uno al comienzo y otro al final del subconjunto que se desea eliminar está documenta en `http://es.cppreference.com/w/cpp/container/set/erase`.

Para las sumatorias restantes vamos a hacer un análisis más fino.[21]

La primer sumatoria restante corresponde al costo de agregar d_i elementos al resultado. Y c_i es la cantidad de elementos en **candidatos** en la i-ésima iteración, y una vez que se agregan al resultado los d_i elementos, se eliminan de la estructura donde estaban. Entonces en la siguiente iteración, **candidatos** tendrá d_i elementos menos, es decir, $c_{i+1} = c_i - d_i$. Con esto se quiere notar que a lo sumo, en las n iteraciones, se agregarán n candidatos al resultado (en realidad a lo sumo $n-1$ pues el último punto del resultado siempre está con altura 0). Entonces dicha sumatoria, es en realidad $O(n * log(n))$.

Para la segunda sumatoria restante, tenemos que calcular la cantidad de techos que fusionan en las n iteraciones. Pero como a lo sumo hay n techos a lo largo de todo el ciclo, y una vez que se fusionan techos se eliminan de la estructura, sabemos que $t_{i+1} = t_i - u_i$. Entonces (análogamente al razonamiento anterior) la sumatoria es en realidad $O(n)$.

Luego, nos queda:

1. $n * log(n)$
2. $n * log(n) + n * log(n)$
3. $n * log(n)$
4. $n * log(n) + n$

Siendo entonces la complejidad del ciclo principal $O(n * log(n))$.

Ahora veamos la complejidad del último ciclo. [22]

La cantidad de iteraciones del ciclo es a lo sumo n, pues a lo sumo todos las segundas paredes caen contra el piso. Agregar al resultado cuesta tiempo logarítimico en relación a la cantidad de elementos. Luego, es fácil notar que la complejidad de éste ciclo es $O(n * log(n))$.

Entonces, al ser la complejidad de **generarEdificiosORdenados**, la del ciclo principal, y la del último ciclo $O(n * log(n))$, la complejidad final del algoritmo que resuelve el problema es $O(n * log(n))$.

Es interesante notar que el algoritmo no tiene un *peor caso*, pues el *trabajo* que se ahorra en un lugar, lo termina haciendo en otro. Si, al armar el set de edificios, se fusionan muchos edificios, ésto aumentará la constante quizás, pero quedarán menos edificios para iterar en **generarHorizonte**. Por otro lado, al agregar techos al diccionario, si ninguno se fusiona entonces todos los edificios están separados entre sí, con lo cual todos los candidatos se agregarán con altura 0 al final el algoritmo. Entonces, de una forma u otra, el algoritmo termina haciendo el mismo trabajo.

De todas formas, el algoritmo sí tiene un mejor caso, el cual explicaremos en la experimentación.

2.8. Experimentación

Para la experimentación realizada en este ejercicio, se decidió tomar el tiempo de la misma forma que en el ejercicio 1. Para poder lograr esto, tuvimos que modificar nuestro código agregando lineas, y modificando la forma de generar estructuras guardando en una lista los valores que ibamos leyendo del stdin, para luego poder generar las estructuras correctamente recorriendo esta lista. Luego de guardar los datos de entrada de la instancia en una lista, procedimos a modificar el main para que reciba la cantidad de iteraciones a realizar, y asi poder cronometrar los tiempos de una manera mas precisa. Esto lo logramos de la siguiente forma:

```
while (iteraciones != 0){
auto t1 = reloj.now();
generar_estructuras(estructuras, por_altura);
generar_horizonte(por_altura, res);
auto t2 = reloj.now();
auto total = duration_cast<microseconds>(t2 - t1).count();
if (total < mi) mi = total;
iteraciones --;
}
```

[21] Los dos análisis que se presentan para dichas sumatorias, son análogos al de **generarEdificiosOrdenados**.
[22] Líneas 23-25 del pseudocódigo.

Con este código lo que logramos fue repetir la ejecución de la instancia obteniendo resultados mas precisos, y quedanonos con el minimo valor obtenido.

Para generar los edificios creamos un programa al que se le pasa la cantidad de edificios a crear, y este los genera de forma aleatoria utilizando la funcion **rand()** previamente explicada. Para lograr que estos edificios sean válidos, es decir $x_1 < x_2$, e $y > 0$, lo que hicimos fue chequear al generarlos si esta condición se cumplía. En caso de no cumplirse, lo que hicimos fue *swapear* los valores de x_1 y x_2.

Como mencionamos anteriormente, el algoritmo propuesto no tiene un peor caso, pero sí un mejor caso. Supongamos que todos los edificios del conjunto de entrada tienen la misma altura y se encuentran solapados. Ésto provocará que a medida que se va armando el set de edificios, se vayan mergeando unos con otros, pues cada edificio que agrego se solapa con el que ya está.

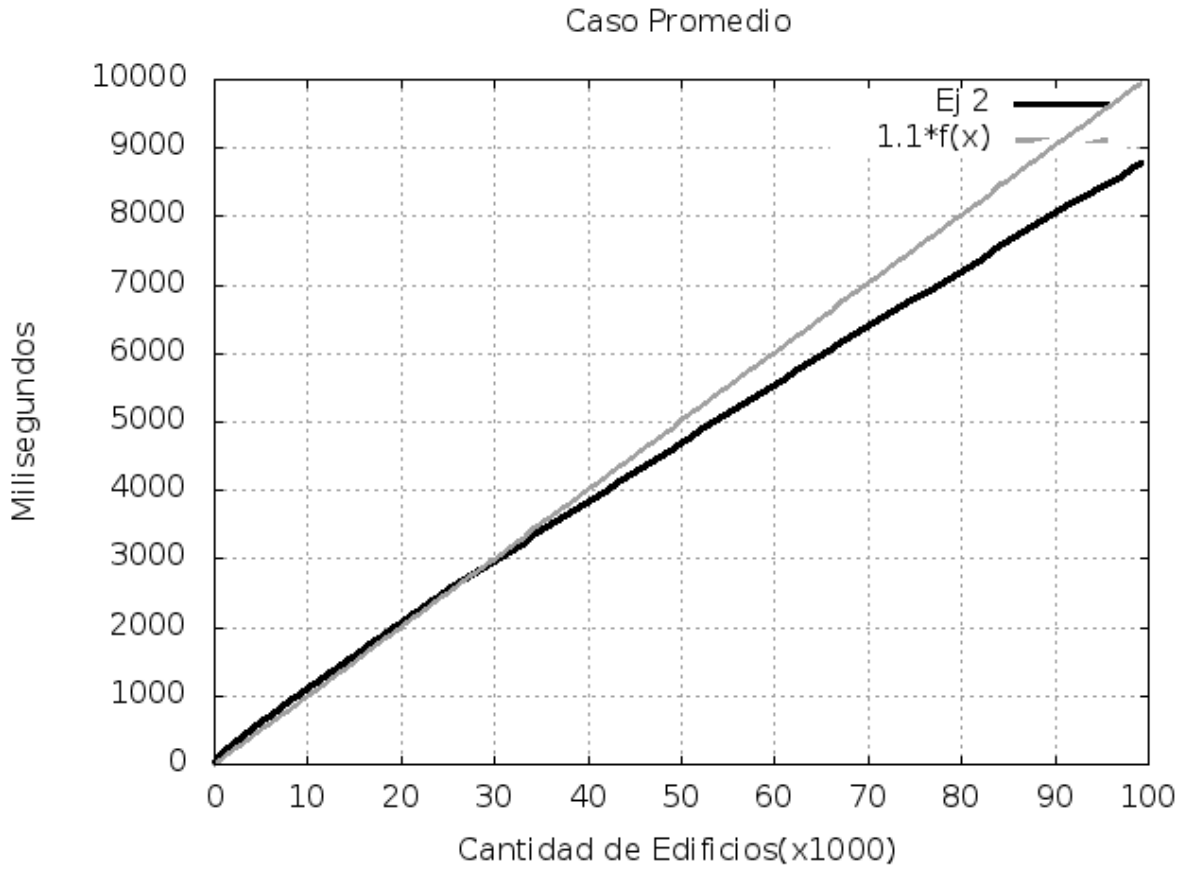
Entonces, al agregar el primer edificio, éste se insertará sin mayor problema, pues la estructura se encuentra vacía. Al agregar el segundo, como se solapa con el que ya está, se fusionarán ambos dejando un solo edificio. Lo mismo pasará con el próximo, y el próximo del próximo, y así sucesivamente hasta que el conjunto de entrada esté vacío. Entonces lo que estará sucediendo es que se harán n fusiones de costo constante.

Para recrear esta situación lo que hicimos fue crear un programa encargado de generar edificios con la misma altura y que cada uno comparta su segundo vertice con el primero del edificio siguiente.

Los gráficos de cada experimento fueron realizados dividiendo a nuestra función por el logaritmo de x. Esto nos permitió ver de forma lineal como se asemeja nuestra función con respecto a la complejidad desarrollada previamente.

2.8.1. Experimento 1

En este experimento analizaremos el caso promedio de nuestro algoritmo.Para el siguiente experimento se utilizo el número 15 como semilla para generar los edificios de forma aleatoria. El experimento consistió en correr nuestro algoritmo 1000 veces por cada instancia de edificios. Comenzamos con 100 edificios y fuimos incrementando la cantidad de a 1000 hasta llegar a los 100000 edificios.

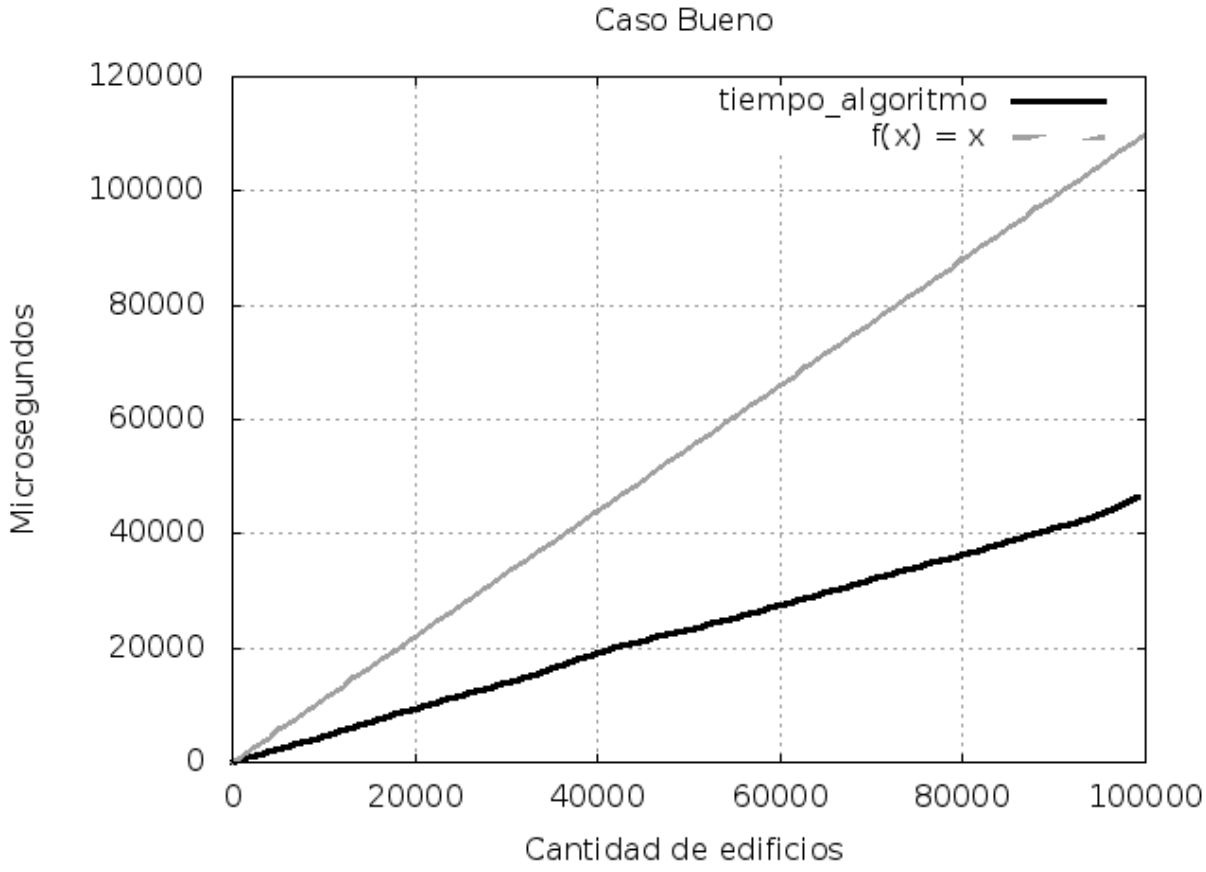


Para este experimento lo que hicimos fue dividir a nuestra función por log(x), logrando asi que nuestra función sea lineal. El objetivo de esto fue poder comparar a nuestra función contra una lineal, para poder analizar el comportamiento de la complejidad con mayor claridad Como se puede observar a partir de 35000 edificios aproximadamente, nuestra función queda acotada superiormente por la

función y = x. Esto nos permite afirmar que la complejidad en el caso promedio es de x*log(x), ya que a partir de un x, nuestra función queda acotada, luego de ser dividida por log(x).

2.8.2. Experimento 2

En este experimento analizaremos el mejor caso de nuestro algoritmo. El experimento consistió en generar 100000 edificios, todos con la misma altura y solapados, comenzando con 100 edificios y aumentandolos de a 1000. A cada instancia de los edificios se la corrió 1000 veces con el objetivo de eliminar posibles *outliers* y se utilizó como semilla para el generador de edificios el número 15.
El resultado del experimento fue el siguiente :



Lo que hicimos en este experimento fue comparar el tiempo de nuestro algoritmo con el de la función y = x. A diferencia del experimento anterior, no hizo falta dividir por la función log(x), ya que en este caso la complejidad es *lineal*. Esto se debe a que al ser todos los edificios de la misma altura y estar solapados, al momento de agregar un nuevo edificio, este deberá *mergearse* con el ya existente, generando asi un único edificio en cada iteración. El costo de *mergear* dos edificios es $O(1)$, pero como debo agregar n edificios, la complejidad termina siendo $O(n)$.

Una vez finalizadas las iteraciones, me quedará un único edificio producto de los n anteriores y procesarlo me costará $O(1)$ ya que al ser el único deberá estar en el resultado si o si.

De esto podemos concluir que el costo de complejidad en nuestro mejor caso se encuentra en la generación del set de edificios de nuestro algoritmo, mas precisamente en el mergeo de edificios, la cual termina siendo *lineal*.

2.9. Anexo reentrega

El Jefe de Desarrollo del módulo de Edificios 2D del sistema desarrollado, nos ha pedido una nueva funcionalidad para nuestro dibujador de contornos. Ahora se requiere que el algoritmo identifique aquellos edificios que no figuran en el contorno final de la ciudad y aquellos que figuran por completo en el mismo, es decir, aquellos cuya altura es la máaxima altura entre todos los edificios a lo largo de todo su ancho. Se pide desarrollar los siguientes puntos:

1. ¿Cómo afecta eso a su algoritmo?
2. ¿Qué impacto tiene en la complejidad?

Para poder considerar esta nueva funcionalidad, no habría que modificar significativamente el algoritmo o sus estructuras auxiliares. Dado que tenemos la estructura **techos**, para saber si un edificio está completamente en el horizonte, sólo hay que fijarse que su techo no se solape con ningún otro de los techos agregados hasta el momento. De igual manera, para saber si un edificio no figurará en el contorno final, basta con evaluar si *todo su techo* es contenido por algún techo de los edificios agregados hasta el momento.

En ambos casos, sólo se necesita hacer una búsqueda en **techos**, cuyo costo es logarítmico en la cantidad de elementos. Y al hacer éstas dos búsquedas para cada edificio, quedaría una complejidad de $O(n * log(n))$. Con lo cual, agregar esta funcionalidad no afectaría la complejidad del algoritmo, pues ya es $O(n * log(n))$.

3. Problema 3

3.1. Descripción del problema

En este ejercicio se nos presenta el problema de minimizar la cantidad de cambiones necesarios para transportar n productos químicos sabiendo que entre cada par de productos existe una cierta *peligrosidad* y que los camiones poseen un *umbral* de peligrosidad determinado. Es decir, es un problema de optimización, donde debemos encontrar la mínima cantidad de camiones de umbral m necesarios para transportar n productos que tienen una relación de peligrosidad entre ellos descripta por coeficientes.

Cada instancia del problema presenta la cantidad de productos a transportar, el umbral permitido para la flota de camiones y los coeficientes de peligrosidad para todas las combinaciones de pares de productos presentados.

A diferencia del problema 1, no tiene sentido aquí hablar de factibilidad, ya que al no existir un límite en la cantidad de camiones a utilizar se podría poner a cada producto en un camión distinto y sería un resultado posible. De esta forma, el desafío del problema consiste en encontrar *una* solución óptima. Una solución S sera óptima si cuando se la compara contra todas las soluciones posibles, S es una de las que menor cantidad de camiones utilizados tiene. Nótese que nos referimos a *una* de las soluciones posibles, y no a *la* solución óptima, ya que podrían existir varias.
Una posible instancia de este problema podría ser:

- 4 productos a transportar, con umbral de peligrosidad por camión de 7 y con los siguientes coeficientes de peligrosidad entre los productos:
  - *Producto 1*: 2 de peligrosidad con el 2, 3 de peligrosidad con el 3 y 1 de peligrosidad con el 4.
  - *Producto 2*: 2 de peligrosidad con el 1, 4 de peligrosidad con el 3 y 4 de peligrosidad con el 4.
  - *Producto 3*: 3 de peligrosidad con el 1, 4 de peligrosidad con el 2 y 5 de peligrosidad con el 4.
  - *Producto 4*: 1 de peligrosidad con el 1, 4 de peligrosidad con el 2 y 5 de peligrosidad con el 3.

Y una solución posible de dicha instancia:

Un entero indicando la cantidad de camiones utilizados y n enteros representando cada elemento en donde se indica el camion donde sera transportado. Siguiendo el ejemplo anterior una solucion optima seria:

Dos camiones, en el primero estám los productos 1 y 2, y en el segundo camión los productos 3 y 4.

3.2. Ideas desarrolladas

Para este problema tenemos que tener en claro que una solución no es cualquier distribución productos en los camiones, sino que es la solución que tiene la menor cantidad de camiones.

Entonces, tenemos que ir recorriendo todas las posibles soluciones para encontrar la correcta. Para ello el algoritmo recibe una lista de productos, un umbral y una matríz con los coeficientes de peligrosidad de los productos.

Para armar una solución posible, se va tomando de un producto a la vez y se evalua si se lo puede poner en el último camión de nuestra solución. De no ser posible, se incorpora un nuevo camión al resultado y el producto se agrega en dicho camión. Al realizar esta última acción se verifica si la cantidad de camiones es mayor o igual a la de la mejor solución que veníamos teniendo. Si es así, el proceso de armado de ese resultado no sigue y esa posible solución se descarta.

Una vez que se distribuyen todos los productos, la solucion que veníamos armando pasa a ser la mejor solución del problema.

Las estructuras que necesitamos para implementar esta idea son:

- **productos**, secuencia de producto. El producto es representado por un entero.
- **camion**, estructura que contiene un atributo *pelig_acum*, que indica la suma de las peligrosidades de los productos agregados a ese camion. También tiene el atributo *productos*, una secuencia que tiene como elementos a los productos que se transportan en ese camion.

- **resultado**, estructura que contiene al atributo *cant*, que es la cantidad de camiones de esa solucion, el atributo *camiones* que es una secuencia que tiene como elementos a los camiones que forman la solucion y por último el atributo *a_imprimir*, que es un arreglo que guarda en la posicion $i-1$ el número del camión donde esta guardado el elemento i, $\forall i \in productos$. Este último atributo es para facilitar la devolución del resulado.
- **matriz**, arreglo de $n-1$ arreglos de $n-i$ coeficientes de peligrosidad, $\forall i \in productos$. Dados dos productos i,j tal que $i \neq j \wedge i < j$ su coeficiente de peligrosidad es $matriz[i-1][j-i-1]$

La función principal de nuestro algoritmo es llenar camiones que recibe los siguiente parámetros:

- **res_par**: resultado donde se irá guardando la solución hasta el momento construida.
- **Q**: secuencia de los productos a colocar en los cammiones.
- **mz**: matriz de coeficientes de peligrosidad.
- **M**: umbral.
- **res_opt**: el mejor resultado obtenido.

Al ser una función recursiva, nos parece necesario para su comprensión aclarar cómo se llama por primera vez. El umbral y la matriz nunca se modifican, no así como el resto de los parámetros. La secuencia de productos se pasa *entera*, y en las llamadas recursivas se va modificando. En cambio, se crearán vacíos los resultados, res_par y res_opt; y res_opt se irá modificando a medida que se encuentren soluciones válidas. A res_opt, para que sí o sí sea reemplazado por una solución, se inicilizará con la cantidad de productos, describiendo de alguna forma que es la solución donde cada producto va en un camión distinto.

Observación: **LC** corresponde a **llenar_camiones**.

```
LC(in/out res_par: result, in/out Q: product, in mz: matriz, in M: Umbral, in/out res_opt: result)
 1  if ya coloque todos los productos en camiones
 2     then
 3          res_opt = res_par;
 4           //al ser recursivo el algoritmo, si entre en este caso significa que mi resultado parcial es mejor o igual al que estaba
 5           //guardado previamente.
 6     else
 7          se crea un iterador en la lista de productos;
 8          while no llegue al final de la lista
 9            if la cantidad de camiones en res_par ≤ cantidad de camiones en res_opt
10               then se copia res_par y se agregara el producto a la copia. Se evalua si se puede
11                    agregar el producto en el ultimo camion que tiene la copia res_par,si no
12                    es posible porque la sumatoria de coeficientes de peligrosidad supera el umbral,
13                    se crea un camión nuevo;
14                     //las recursiones del algoritmo facilitan que solo verificando el ultimo camion
15                     //sea suficiente para conseguir todas las soluciones posibles
16                    se copia la lista de productos y se borra el elemento que recien agregamos al
17                    camion;
18                    llamo recursivamente a llenar_camiones con la copia de res_par, la copia de
19                    la lista de productos, la matriz de coefcientes mz, el umbral M y la solucion optima
20                    res_opt;
21                    endif
22            avanzo el iterador con el que recorro la lista de productos;
23            endwhile
24          endif
```

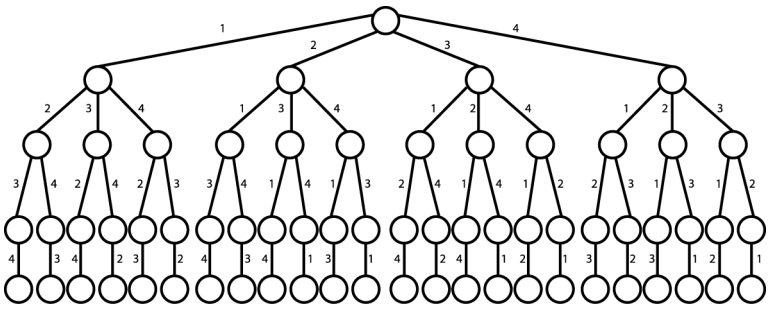
3.2.1. Recorriendo las soluciones

Para una mejor comprensión de cómo se generan las recursiones del algoritmo, a continuación se mostrará el recorrido que realiza el algoritmo en el árbol de soluciones. A modo ejemplo utilizaremos 4 productos con un umbral de valor 2.

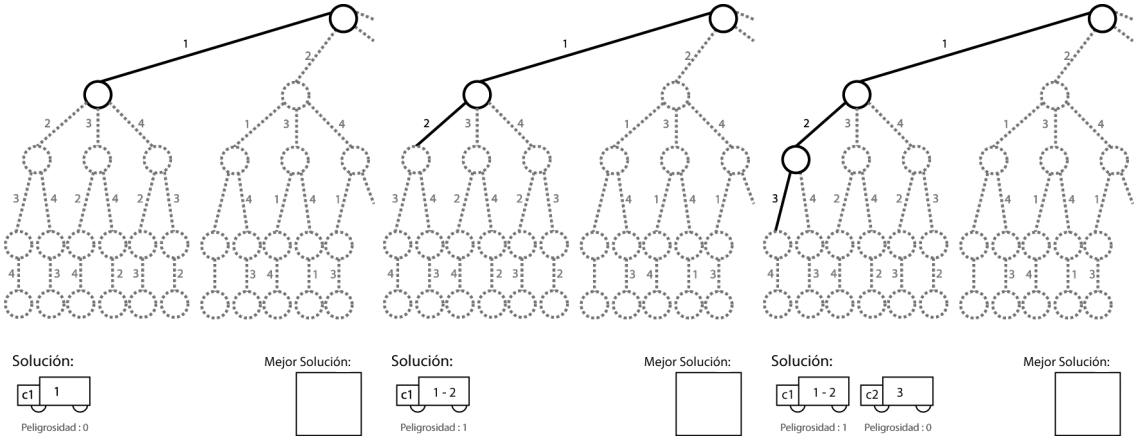
La matriz:

$$\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix}\begin{bmatrix}0\end{bmatrix}$$

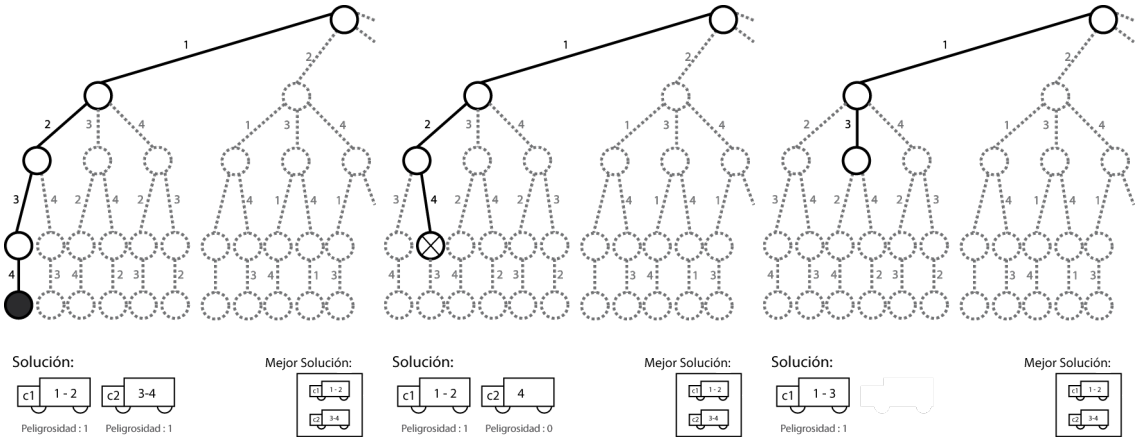
Figura 6: Árbol de soluciones de un input de 4 productos a rrecorrer por nuestro algoritmo



(a) Árbol de soluciones



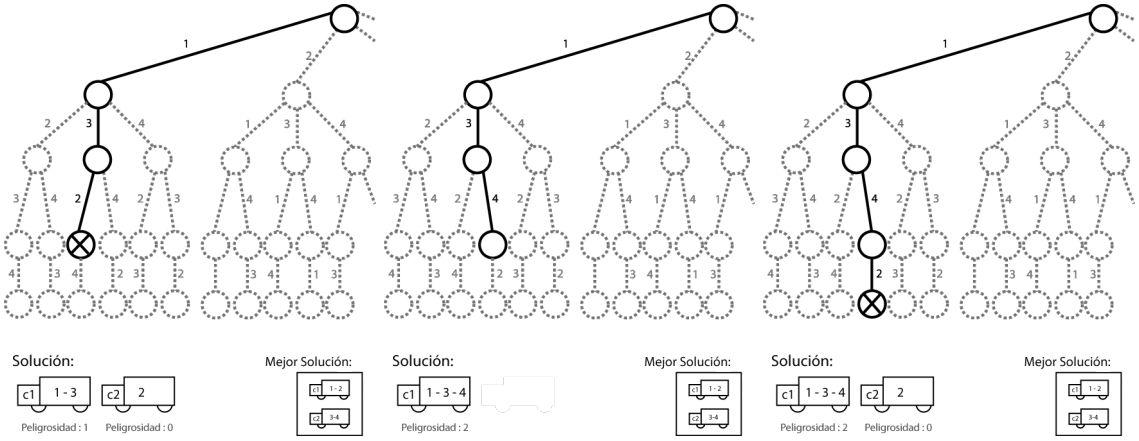
(b) Se acomoda el producto 1 (c) Se acomoda el producto 2 (d) Se acomoda el producto 3



(e) Se acomoda el producto 4. Como se llego a una hoja, entonces tenemos una posible solución. Se guarda como la una Mejor Solucion.

(f) Se vuelve hasta el nodo interno. Que es la solución parcial que tiene un camión con el producto 1 y 2. Acomoda el producto 4 agregándolo en otro camión. Esta solución parcial tiene la misma cantidad de camiones que la mejor solucion. Dada esta condición el algoritmo decide no sigue por esta rama.

(g) Regresa al nodo interno anteior, donde la solucion parcial es un camión con el producto 1. Acomoda el producto 3 en el camión.

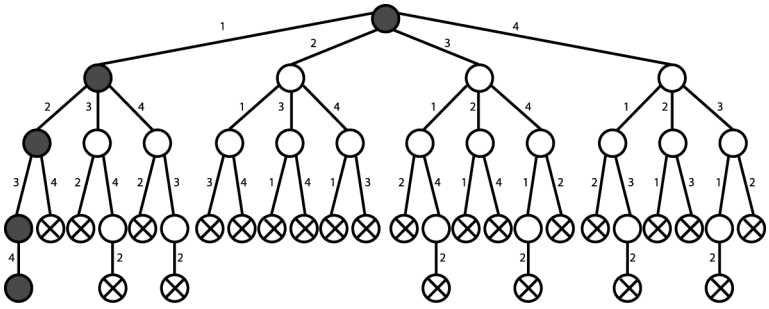


(h) Sigue avanzando por la misma rama. Cuando el algoritomo acomoda el producto 2, al necesitar otro camion, esta rama deja de ser una posible solución, ya que tiene igual cantidad de camiones que la Mejor solución.

(i) Vuelve al nodo interio anterior, donde la solución parcial es un camión con el producto 1 y 3. Acomoda el producto 4 en el mismo camión.

(j) Al acomodar el producto 2 en un nuevo camión, esta solucion es descartada ya que posee la misma cantidad de camiones que la Mejor Solución. EL algoritmo irá avanzando sobre el árbol de soluciones como venimos describiendo hasta llegar al final.

Figura 7: Aca podemos ver el recorrido completo que realizará el arlgoritmo. Los nodos coloreados son la Mejor solución. Los nodos en blanco, son aquellos por donde el algorimo pasó, evaluó y decidió continuar evaluando su siguiente. Y por último los nodos marcados con la cruz, son aquellos por donde el algoritmo paso, evaluó y decidió no continuar (descartó ese camino como solución).



(a) Recorrido completo del Algoritmo en un árbol cuyo input es de 4 productos, umbral 2 y la matriz dada anteriormente

3.3. Análisis de Complejidad

Para realizar en análisis teórico de Complejidad, al ser un algoritmo recursivo, utilizaremos su arbol de desición, para calcular la cantidad de nodos por los cual pasa el agloritmo y además, necesitaremos conocer los costos de las desiciones tomadas por cada nodo que evalúa. Una vez conocidos estos datos podremos conocer la complejiadad de nuestro algoritmo: $O(\#nodos \times costedelasdesiciones)$

3.3.1. Cantidad de Nodos

Tal como se mostró anteriormente, el árbol comienza desde el caso vacío y va avanzando en profundidad. Para comenzar tiene n ramas, siendo n la cantidad de productos, por las cuales avanzar. Cada una de estos caminos, posee n-1 ramas para recorrer, de las cuales se desprende de cada una n-2 ramas y así hasta llegar al nodo de grado 0. La altura del árbol es n. Entonces podríamos decir que la cantidad de nodos que tenemos es
$n+n\times(n-1)+n\times(n-1)\times(n-2)n\times(n-1)\times(n-2)\times(n-3)+...+n\times...\times(n-(n-2))+n! = \sum_{i=1}^{n} \frac{n!}{(n-i)!}$

Dado que $\frac{1}{(n-i)!} \leq 1, con 1 \leq i \leq n$, se puede acotar
$\sum_{i=1}^{n} \frac{n!}{(n-i)!} = n! \times \sum_{i=1}^{n} \frac{1}{(n-i)!} \leq n! \times \sum_{i=1}^{n} 1 = n! \times n$

Podemos acotar la cantidad de nodos por $n! \times n$.

3.3.2. Decisión

La desición que realiza nuestro algoritmo es chequear que la cantidad de camiones de la solución parcial sea menor a la cantidad de camiones de la solución óptima. De ser menor, el algoritmo continúa avanzando por dicha rama. Caso contrario, descarta ese camino.
Si llega a una hoja, entonces la solucion recorrida en ese momento, pasa a ser la solución óptima. Esta

desición lo hace la función Sigue_siendo_solucion. A continuación su Pseudo Código:

```
PELIGROSIDAD_ELEM(in ls: productos, p prodcto:, in mz: matriz)→ int
1  int r = 0;
2  while no llegue al final de la lista ls
3    res = res + el coeficiente de peligrosidad de p con el producto de la lista ls;
4    endwhile
5  return res;
```

Peligrosidad_elem recorre la lista de productos que tiene el último camión del resultado parcial. Esta lista siempre es menor a n, ya que como estoy evaluando el producto que quiero agregar a la solución, con el resto de los productos, estos a lo sumo son n-1, por lo que podemos poner la cota de n produtos. Entonces esta función es *O(n)*.

```
ES_POSIBLE_AGREGAR(in/out res_parcial: resultado, in p: productos, in mz: matriz, in M: umbral, in/out r
1  if la lista de camiones de res_parcial no es vacía
2    then int peligrosidad = paligrosidad acumulada en el ultimo camion de la solucion;
3         int peligrosidad_nueva = peligrosidad_elem(lista de productos del ultimo camion,p, mz);
4         return (peligrosidad + pelig_nueva)≤ umbral;
5         elseif return true;
```

Es_posible_agregar calcula en cuanto quedaría la peligrosidad del camión agregando el nuevo elemento. Si esta supera el umbral dado, entonces devuelve false. El algoritmo solo cuenta de asignaciones y operaciones matemáticas, por lo que le da la complejidad al algoritmo es la función peligrosidad_elem, que es *O(n)*.

```
SIGUE_SIENDO_SOL(in/out res: resultado, in p: productos, in mz: matriz, in M: umbral, in/out res: resulta
1  int cant_camiones = r.cantidad;
2  if no es_posible_agregar(r, p, mz, umbral, )
3    then cant_camiones + +;
4         endifreturn cant_camiones ¡cantidad de camiones de la solución óptima
```

Sigue_siendo_sol, es el algoritmo que toma la desición de contuinuar o no por la rama de esta posible solución. Dado a que si no es posible agregar el elemento al ultimo camión de la solución parcial, analiza si agregando un nuevo camión a la solución parcial continúo teniendo menor cantidad de camiones que la solución optima, de ser asi puedo seguir evaluando esta rama.
El algortimo va a tener la complejidad del algoritmo es_posible_agregar, *O(n)*. Por lo que el costo de la desición es *O(n)*.

3.3.3. Conclusión de análisis de complejidad

Como habíamos indicado anteriormente, nuestro algoritmo tiene una complejidad $O(\#nodos \times costedelasdesiciones)$. Calculando la cantidad de nodos de nuestro árbol de soluciones, y el costo de las deciones a realizar concluímos que su complejidad es $O(n! \times n^2)$

3.4. Experimentación

Nuevamnete como en el ejercicio 1, para la experimentación decidimos analizar 3 posibles casos sobre nuestros algoritmos, mejor caso, peor caso y caso promedio. Se crearon 3 funciones, las cuales se encargan de generar estos posibles escenarios. Luego creamos 3 scripts(uno por cada escenario) a los cuales se les pasa la cantidad de productos iniciales, el umbral, la cantidad de productos finales, de a cuanto se va a incrementar las distintas instancias, y la cantidad de repeticiones que se va a evaluar la instancia para conseguir el menor tiempo de ejecucion y asi evitar posibles outliers. Este ejercicio tiene la peculiaridad de no tener una buena complejidad, por lo que nos vimos forzados a no tener una gran cantidad de repeticiones.

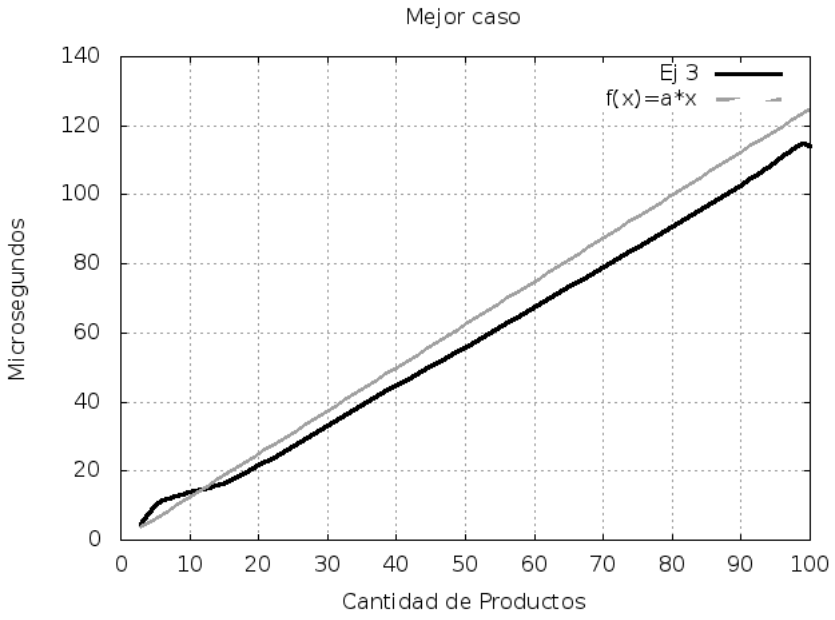
Para medir los tiempos de ejecución de nuestros algoritmos utilizamos la libreria çhrono.h "de c++, la cual nos suministró la posibilidad de generar relojes que nos ayuden a medir los tiempos de ejecucion de nuestro algoritmo en microsegundos.

Para esto le agregamos unas lineas extra a nuestro codigo:

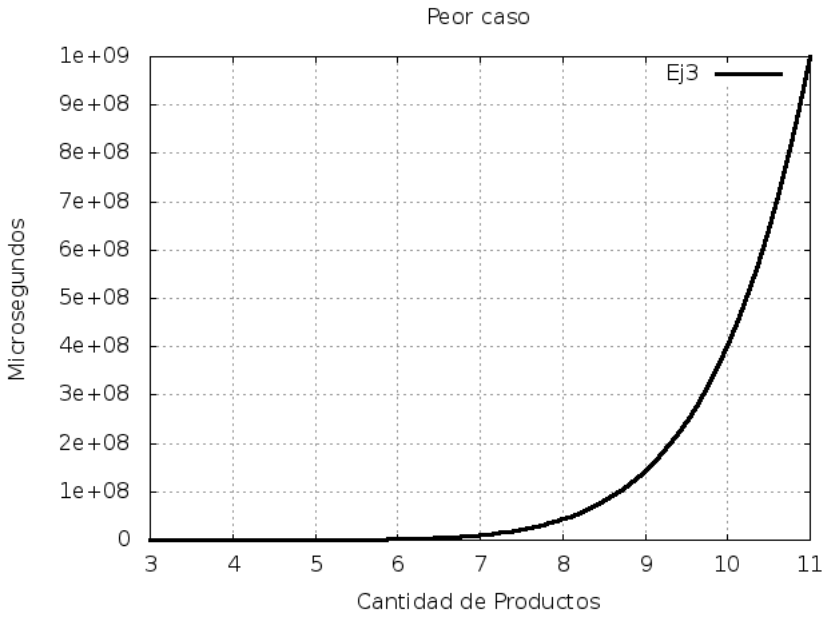
```
high_resolution_clock reloj;
size_t minimo = 99999999;
auto t1 = reloj.now;
llenar_camiones(vacio,lista_productos,matriz, umbral,res_g);
auto t2 = reloj.now;
auto total = duration_cast<microseconds>(t2 - t1).count();
if (total < minimo) minimo = total;
```

3.4.1. Experimento 1

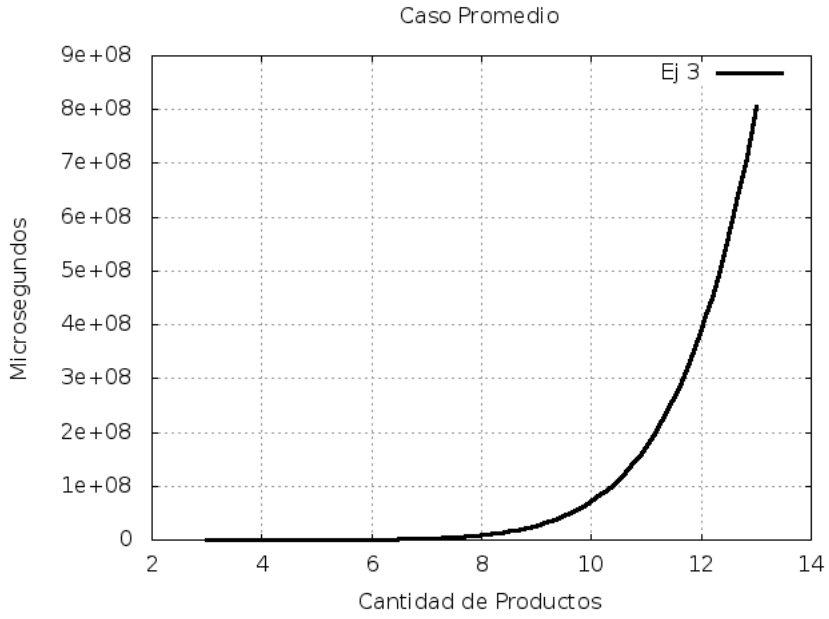
- **Mejor Caso:** El mejor caso para este algoritmos es cuando el primer resultado que obtiene son todos los productos en un solo camion. Ya que así solo se rrecorrerían los n productos que forman el resultado, más los n -1 productos del primer nivel. Esto nos daría un total de 2n-1 nodos, por lo que la complejida del mejor caso sería $O(2n^2 - n)$, que lo podemos acotar $O(n^2)$.
  Para generar este caso, lo que se hizo fue poner los coefiecientes de peligrosidad en 0.



- **Peor Caso:** El peor caso para este algoritmo se presenta cuando todos los coeficientes de peligrosidad son mayor al umbral, y por esto la solución es un producto por camión. Este caso está acotado por $O(n! \times n^2)$, ya que recorre todos los nodos del árbol de soluciones.



- **Caso Promedio**: no intervenimos en la creación de las instancias por lo que los casos son los más aleatorios posibles.



3.4.2. Poda: evaluando coeficiente mínimo

La idea de esta poda, es tomar el mínimo coeficiente de peligrosidad de los productos que faltan colocar en los camiones, y evaluar la cantidad de camiones que se formarían si todos esos productos restantes se relacionasen entre sí con ese coeficiente. Si dicha cantidad de camiones supera al de la mejor solución se descarta esa rama del arbol de soluciones. El razonamiento detrás de la poda es pensar en que si al darse el mejor caso posible, es decir, todos los elementos se relacionan con el mínimo coeficiente posible, la solución que estoy armando es peor que la óptima, entonces nunca será buena. Podría verse como pensar en el mejor escenario posible y ver que aún así no hay oportunidad de llegar a una solución optima.

Para esto, primero se busca el mínimo coeficiente de la matriz. Como ésta tiene $\frac{(n-1)\times n}{2}$ elementos, y buscar al coeficiente es lineal en el tamaño de la matriz, el costo de esta operación termina siendo $O(\frac{(n-1)\times n}{2})$.

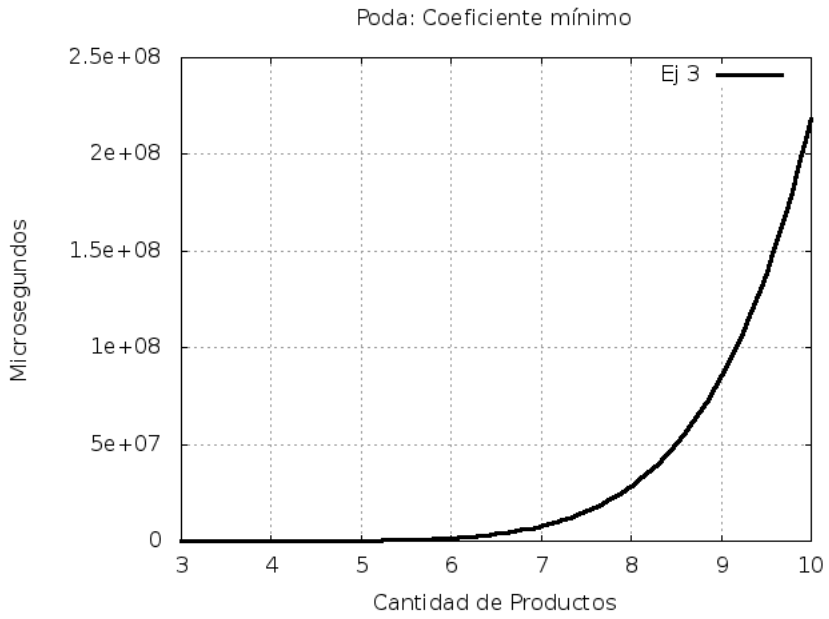
El siguiente paso es calcular la cantidad de productos que entran en el camión. [23].Para hacer ésto, se va calculando la peligrosidad del camión hasta que supera el umbral. Entonces a lo sumo se recorre la lista de n productos, por lo que tiene coste de $O(n)$.

[23] *Aclaración*, para conocer la cantidad de productos utilizamos la operación de list "size()", que tiene un costo lineal en algunas versiones de $C++$, pero en $C++0X$ es $O(1)$.

Por último, se calcula si la cantidad de camiones necesarios para colocar los productos restantes en base al menor coeficiente de peligrosidad supera a la cantidad de camiones de la mejor solución. Esta poda sólo se aplica cuando ya existe una solución y para cuando se le pasa una lista de productos con más de un producto.

La complejidad de esta última parte, es lineal en la cantidad de camiones que armo, que a lo sumo son n camiones (1 producto por camión), $O(n)$.

Entonces para la complejidad de esta poda tenemos: $O(\frac{(n-1)\times n}{2}) + O(n) + O(n) = O(\frac{n^2-n}{2} + 2n) = O(\frac{n^2-3n}{2}) = O(n^2 - n)$



3.4.3. Conclusión

Como se observa en la experimentación, la poda no funcionó. Y no sólo eso, si no que sumó un tiempo de ejecución considerable.

Ésto se debe a que es una poda muy liviana, que contempla pocos casos. Si en los productos que restan evaluar existe alguno que tenga un coeficiente bajo con algún otro producto, la poda se vuelve ineficiente, pues lo más probable es que casi ningún caso caiga aquí.

Entonces lo único que agrega la poda al algoritmo es una carga en el tiempo de ejecución, no cumpliendo su propósito. No es una poda que agregaríamos a nuestro algoritmo, pero nos pareció enriquecedor incluir el expirimento y los resultados en el informe.

3.4.4. Poda: evaluando productos peligrosos

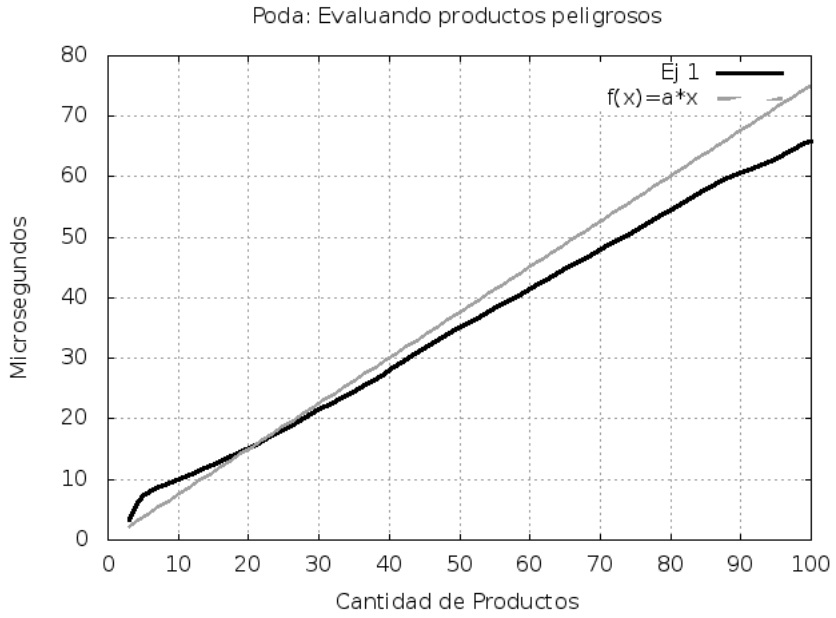
La idea de esta poda consiste en separar a los productos mas peligrosos en camiones separados, con el objetivo de aislarlos, dado que no pueden compartir un camión con ningún otro producto, pues su relación de peligrosidad con los demás productos es demasiado alta .

Para esto se deben recorrer los todos productos, analizando para cada uno de ellos si su coeficiente con los restantes supera el umbral de peligrosidad. En caso de que esto ocurra se deberá aislar a este producto en un camión y removerlo de la lista de productos restantes.

Con la implementación de esta poda, se logra mejorar el peor caso planteado en el primer experimento, ya que al aislar los productos, estamos restringiendo nuestro futuro árbol de decisiones y de esta forma, recorriendo menos nodos. Esto sucede ya que al precalcular que productos nunca podrían relacionarse con otros, los voy descartando antes de aplicar el algoritmo de backtracking. Por éste motivo consideramos que no es una poda *convencional*, en el sentido de que no se aplica al momento de tomar una desición o al estar en un punto de la ejución general del algoritmo, si no que sólo se realiza al comienzo.

La complejidad de esta poda es $O(n^2)$ debido a que por cada producto tengo que recorrer la matriz de coeficientes, chequeando si el coeficiente supera el umbral. Esta verificación tiene un costo constante, pues es una mera comparación de dos enteros, pero como se deben hacer n chequeos por cada producto, y tengo n productos, la complejidad de la poda termina siendo cuadrática.

Como esta poda la hacemos antes de llamar a la función de backtracking, la complejidad final del algoritmo con la poda termina siendo $O(n! * n) + O(n^2) = O(n! * n)$, es decir que aplicar la poda no altera la complejidad teórica original.



3.4.5. Conclusión

Si evaluamos el caso promedio, no vemos una disminución en el tiempo de ejecución, pues para que la poda tenga un efecto notable tiene que haber varios productos con altos niveles de peligrosidad.

Sin embargo, si vemos el peor caso del algoritmo (ver gráfico), es justamente el mejor caso de la poda, es decir, cuando todos los elementos van en camiones separados. Es aquí donde la poda se luce.

Es entonces una poda que se podría agregar al algoritmo sin perjudicar su performance, pues no altera la complejidad teórica y para casos puntuales es útil.

3.5. Anexo Experimentación

Debido a los altos tiempos de ejecución del algoritmo, no fue posible en la mayoría de los experimentos considerar casos con grandes tamaños en la entrada. Ya que a partir del elemento 13 los tiempos eran demasiado grandes.

3.6. Anexo reentrega

La empresa está a punto de contratar nuevos camiones y esta nueva flota no será homogénea, en el sentido del equipamiento de seguridad de los camiones. Con esto, cada camión tendrá definido un umbral de peligrosidad distinto. Se pide desarrollar los siguientes puntos:

1. ¿Cómo afecta eso a su algoritmo?
2. ¿Qué cosas se podían hacer con el problema anterior para acelerar los tiempos de ejecución que ahora ya no se pueden? (es posible que la respuesta del item anterior responda esta pregunta).

En principio tendríamos que identificar a los camiones por su umbral, ya que dejan de ser homogéneos, lo cual no es necesariamente malo, pero habría que mantener alguna estructura que indique cuál es el umbral del camión actual (el que está siendo llenado) y que también indique cuales son los camiones con sus respectivos umbrales que restan ser usados. Una estrategia sencilla y efectiva podría ser ordenar los camiones según su umbral y comenzar utilizando los que tengan el valor mayor.

Frente a las nuevas características del problema no podríamos ,tal como está implementada, nuestra poda "evaluando coeficiente mínimo" donde estimamos la cantidad de camiones que agregaríamos al resultado si desde ese punto en adelante todos los productos tienen peligrosidad igual a la mínima calculable entre los productos restantes.

Lo que nos impide usar esta poda tal como está implementada es que la predicción deja de tener sentido ya que, al ser los camiones de umbral variable, no se puede tomar un umbral de referencia para hacer los calculos (podría pasar que el próximo camión tenga umbral suficiente para acomodar todos los productos restantes, pero este caso ya es revisado por nuestra función sigue_siendo_solución que analiza si agregándole un camión más a la solución deja de ser óptima).

Para adaptarla, en vez de hacer el cálculo, se tendría que ir simulando qué sucedería al agregar los productos (asumiendo como antes que todos se relacionan con el coeficiente mínimo) a los distintos camiones, aumentando aún más la complejidad de la poda.

En cambio, para *evaluando_productos_peligrosos* la situación es más amigable. Se tendría que filtrar a los productos que superan el umbral más grande de todos y ponerlos (separados) en los camiones con el menor umbral, para maximizar el uso de los demás camiones. La complejidad de la poda seguiría siendo la misma si se puede encontrar en un tiempo razonable el umbral más grande de todos los camiones. El único recaudo extra que habría que tener en cuenta sería mantener actualizada la estructura de qué camiones (con qué umbrales) están ocupados y libres.