DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Informe y análisis de resultados.

Algoritmos y Estructuras de Datos III

Grupo Theta

Integrante	LU	Correo electrónico
Rey, Martin	483/12	marto.rey2006@gmail.com
Marta, Cristian Gabriel	079/12	cristiangmarta@gmail.com
Lebedinsky, Alan	802/11	alanlebe@gmail.com
Coronel, Ezequiel	352/08	ecoronel@dc.uba.ar

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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

Contents

Introducción **3**

1 Puentes sobre lava caliente **4**
1.1 Introducción . . . 4
1.2 Desarrollo . . . 4
1.3 Demostración de correctitud . . . 4
1.4 Complejidad . . . 5
1.5 Testing . . . 7
1.6 Experimentación . . . 7

2 Horizontes Lejanos **9**
2.1 Introducción . . . 9
2.2 Desarrollo . . . 9
2.3 Demostración de correctitud . . . 11
2.4 Complejidad . . . 12
2.5 Casos de test . . . 13
2.5.1 Todos los edificios iguales . . . 13
2.5.2 A termina en la posición x y B comienza en la posición x . . . 13
2.5.3 No hay superposición entre edificios . . . 13
2.5.4 Escalera . . . 13
2.5.5 Uno adentro del otro . . . 14
2.6 Experimentación . . . 14

3 Biohazard **17**
3.1 Introducción . . . 17
3.2 Desarrollo . . . 17
3.3 Complejidad . . . 18
3.4 Testing . . . 18
3.5 Experimentación . . . 18

A Ejercicio 1 **19**

B Ejercicio 2 **20**

C Ejercicio 3 **22**

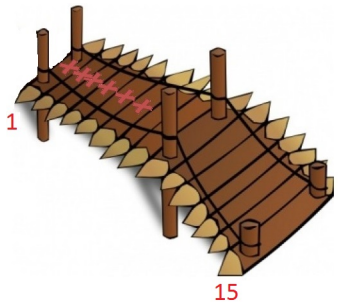
Introducción

En el presente trabajo práctico se intenta resolver mediante algoritmos ciertos problemas brindados por la cátedra. Se decidió implementar dichas soluciones usando el lenguaje *C++*[**?**]. Junto con las implementaciones, se adjunta el presente informe que especifica los detalles sobre el desarrollo de las soluciones, así como pruebas, cálculos de complejidad temporal, mediciones, etc.
Para calcular la complejidad de los algoritmos involucrados, utilizamos el **modelo de costos uniforme**, que es el solicitado por el enunciado.

1 Puentes sobre lava caliente

1.1 Introducción

Este problema trata sobre una competencia en la cual los participantes tienen que atravesar una serie de puentes colgantes, donde cada puente tiene una cantidad de tablones y se sabe cuáles son los tablones que están rotos en cada puente.
Para cruzar cada puente los participantes solo pueden saltar de tablón a tablón, y no se puede saltar a tablones que están rotos pues se caería debajo donde hay un rio de lava. También cada participante cuenta con un límite en la cantidad de tablones que puede saltar de una sola vez.
Los ganadores de la competencia son aquellos que crucen los puentes con la menor cantidad de saltos posibles.
A continuación se muestra un ejemplo con su correspondiente solución:



(a)

Solución			
Saltos	7	10	16

(b)

Figure 1: Instancia de un puente con 15 tablones donde los primeros 6 estan rotos con un limite en la cantidad de tablones que se puede saltar de una vez de 7 con una posible solución

1.2 Desarrollo

Este problema lo resolvimos usando una técnica golosa, la cual consiste en "saltar lo más lejos posible", es decir que siempre que se esté en un tablón i del puente con i entre 0 y n se mira si puede saltar c tablones, donde c es el salto más largo que puede hacer un participante. Si el tablon número $i+c$ está sano, se procede a saltar a dicho tablón. En caso contrario se mira el tablon $i+c-1$, es decir uno anterior y se efectúa la misma pregunta. Así sucesivamente hasta:

- Saltar a un tablón sano.
- No poder saltar a ningún tablón, lo que implica que hay un tramo del puente donde hay k tablones rotos seguidos, con $k \epsilon N$ y $k > c$. Entonces no se puede cruzar el puente

Pseudocódigo:

```
1 Mientras no termine de cruzar el puente
2     Si puedo saltar a algun tablon
3         Salto lo mas lejos posible
4     Sino
5         Retorno que no se puede cruzar el puente
6 Fin mientras
```

1.3 Demostración de correctitud

Notemos que para este problema siempre hay una solución posible, ya sea cruzar el puente o no, para toda solución donde se puede cruzar el puente requiere una cantidad de saltos determinada entonces existe al menos una solución óptima, para la solución en la que no se puede cruzar el puente esta es la solución óptima.
Definimos una solución como el vector de los saltos que se realizaron para cruzar el puente ordenados por cómo se cruzaría el puente, entonces una sub solución es ser prefijo de un vector solución. Además una solución donde el vector==vacío implica que no se puede cruzar el puente.

Nuestro algoritmo comienza con S=Vacío que es una sub solución de alguna solución óptima.
También la cantidad de tablones del puente es finita por lo que eventualmente nuestro algoritmo terminara y nos dará una solución. Esto se debe a que el algoritmo ira saltando tablones y en algún momento:

- (a) Terminara de cruzar el puente
- (b) No podrá cruzar el puente

Veamos que la solución dada por el algoritmo es óptima:
(a) Dado S={ s_0,s_1,...,s_n} alguna solución optima y S'={ s'$_0$,s'$_1$,...,s'$_n$} la solución de nuestro algoritmo,

- Si S=S' entonces nuestra solución es optima
- Si S != S' entonces tomamos el primer salto donde son distintos, es decir, el primer s_i != s'$_i$ con 0≤ i ≤ n y podemos saber a qué tablón estamos saltando haciendo la sumatoria $\sum_{k=1}^{i} s_k$, llamemos t_s y $t_{s'}$ a los respectivos tablones, luego como la estrategia de nuestro algoritmo es saltar lo más lejos posible → $t_s < t_{s'}$ pues si hasta el salto s_{i-1}, S-s_i era igual a S'-s'$_i$ no puede ser que el salto s_i llegue a un tablón más lejano porque quiere decir que s'$_i$ no es el salto más lejano y hubiera saltado a ese tablon. Por lo tanto en el primer salto donde difieren las soluciones, nuestro algoritmo llega a un tablón más lejano.

  Sabiendo esto si seguimos mirando el resto de los saltos de S y S' uno a uno en ningún momento puede pasar que algún s_j llegue a un tablón t_p posterior que $s_{j'}$ con i< j ≤ n por la misma razón que mencionamos (como está definido nuestro algoritmo hubiera saltado a t_p), esto quiere decir que en cada salto nuestro algoritmo cae en el mismo tablón que el de la solucion optima o en alguno más adelante.
  Finalmente llegaríamos a que con s_n y $s_{n'}$ terminamos de cruzar el puente y como comparamos salto a salto todos los saltos de ambas soluciones, S y S' tienen la misma cantidad de saltos (si S' tuviera menos saltos entonces S no sería optima, que es absurdo porque S es óptima) entonces S' es solución óptima.

(b) Se debe a que hay una cantidad de tablones rotos consecutivos > c. Llamemos p_r al primero de los tablones rotos consecutivos, el algoritmo ira avanzando hasta llegar al tablón anterior a p_r e intentara saltar pero debido a que los tablones rotos consecutivos son más que lo máximo que se puede saltar no tendrá forma de hacerlo. Luego devolverá como solución que no se puede cruzar el puente, la cual es solución óptima.

1.4 Complejidad

Vamos a analizar la complejidad en peor caso de nuestro algoritmo, para esto vamos a definir los casos donde ocurre esto, que son los siguientes:
Sean
c = máxima cantidad de tablones q se puede saltar de una sola vez
n = cantidad de tablones del puente

- **Caso 1**: Que todos los tablones del puente estén sanos y $c = 1$
- **Caso 2**: Que haya una cantidad de tablones rotos consecutivos mayor a c con $c < n$.
- **Caso 3**: Si numeramos los tablones desde 1 a n, que solo los tablones (1, 1+c, 1+c+1, 1+c+1+c, ...) estén sanos y el resto rotos.

A continuación el extracto más importante del código para analizar la complejidad:

```
1  uint16_t j = 0, k = 0, n = tablones.size();
2
3  if (c < n) {
4    while (j < n) {
5      for(k = c; k > 0; k–) {
6        if ((j+k-1 < n) ∧ (tablones[j+k-1] != false)) {
7          saltos.push_back(j+k);
8          break;
9        }
10     }
11     if (k == 0) {
12       saltos.clear();
13       break;
14     }
15     j += k;
16   }
17 } else {
18   saltos.push_back(n);
19 }
```

Línea 1: Son asignaciones que cuestan O(1) (la función *size* sobre vectores de c++ tiene costo constante[1])
Línea 3: La guarda del if son comparaciones que cuestan O(1) y el cuerpo si se cumple la guarda del if cuesta O(n) (línea 4 en cualquier caso) y sino se cumple cuesta O(1) (linea 18) ⇒ O(1*n)=O(n)
Línea 6: La guarda del if son comparaciones que cuestan O(1) y el cuerpo cuesta O(1)+O(1)=O(1) (líneas 7 y 8) ⇒ O(1*1)=O(1)
Línea 7: La función *push_back* sobre vectores de c++ tiene costo constante[2] ⇒ O(1)
Línea 12: La función *clear* sobre vectores de c++ tiene costo lineal sobre el tamaño del vector[3] ⇒ O(n)
Línea 15: Es una asignación que cuesta O(1)
Línea 18: La función *push_back* sobre vectores de c++ tiene costo constante[4] ⇒ O(1)

Las siguientes líneas las analizamos dependiendo el escenario de peor caso en que estamos:

- **Caso 1**:
  - Línea 11: La guarda es una comparación que cuesta O(1) y como k=1 (por linea 5) no se cumple la guarda ⇒ O(1)
  - Línea 5: Como c=1 ⇒ k=1 y el for solo itera 1 vez que cuesta O(1) y el cuerpo cuesta O(1) (línea 6) ⇒ O(1). Ademas como se cumple la guarda del if de la linea 6 pues todos los tablones estan sanos, el break de la linea 8 va a cortar el for dejando k=1.
  - Línea 4: k=1 (por línea 5) y como j incrementa dependiendo de k (línea 15) entonces el ciclo itera n veces O(n), y el cuerpo cuesta O(1)+O(1)+O(1)=O(1) (líneas 5, 11 y 15) ⇒ O(n*1)=O(n)
- **Caso 2**:
  - Línea 11: La guarda es una comparación que cuesta O(1) y el cuerpo (por línea 5) sabemos que itera n veces ⇒ k=0, se cumple la guarda y el cuerpo cuesta O(n)+O(1)=O(n) (líneas 12 y 13) ⇒ O(1*n)=O(n)
  - Línea 5: Es un for que sabemos que nunca se va a cumplir la guarda de la línea 6 y entonces va a iterar c veces, como el cuerpo cuesta O(1) (línea 6) ⇒ O(c*1)=O(c) ademas sabemos que $c < n$ luego podemos acotar por n ⇒ O(n)
  - Línea 4: Por línea 11 sabemos que va a cortar el ciclo con el *break* entonces solo va a iterar 1 vez O(1), y el cuerpo cuesta O(n)+O(n)+O(1)=O(2n)=O(n) (líneas 5, 11 y 15) ⇒ O(1*n)=O(n)
- **Caso 3**:

[1] http://es.cppreference.com/w/cpp/container/vector/size
[2] http://es.cppreference.com/w/cpp/container/vector/push_back
[3] http://es.cppreference.com/w/cpp/container/vector/clear
[4] http://es.cppreference.com/w/cpp/container/vector/push_back

- Línea 11: La guarda es una comparación que cuesta O(1) y como k >0 (por linea 5) no se cumple la guarda ⇒ O(1)
- Línea 5: El for va a iterar c veces porque hay tramos del puente con c-1 tablones rotos, el cuerpo cuesta O(1) (linea 6) ⇒ O(c*1)=O(c)
  Tambien como siempre hay un tablon sano al que va a saltar, eventualmente en alguna iteracion se va a cumplir la guarda del if de la linea 6 terminando el for, mirando con mas detalles se ve que siempre termina con k=1 o k=c.
- Línea 4: El ciclo va a iterar dependiendo de k (por linea 15) que sabemos que k=1 o k=c (por linea 5) entonces va a iterar $\frac{n}{c}$ veces, y el cuerpo cuesta O(1)+O(c)+O(1)=O(c) (lineas 5, 11, 15) ⇒ O($\frac{n}{c}$*c)=O(n).

Por lo tanto como la complejidad del algoritmo es el de la línea 3 que es **O(n)**.

1.5 Testing

Testeamos los siguientes casos de interés:

- Todos los tablones sanos y $c = 1$
- $c > n$
- Instancia generada aleatoriamente
- Instancia donde haya una parte del puente que tenga j tablones continuos rotos y $c < j$ con $j \epsilon N$

Los test se pueden ver en el archivo: "TP1/ej1/src/tester.cpp"

1.6 Experimentación

Una vez hecho el análisis de la complejidad teórica, realizamos experimentos con el fin de contrastar los resultados empíricos y comprobar que el algoritmo implementado efectivamente tenía una complejidad temporal de $O(n)$. Para esto tomamos instancias aleatorias del problema, es decir se tomó una distribución discreta uniforme entre 1 y 100 y se utilizaba este valor para la cantidad de saltos del jugador. Para reducir los posibles errores de medición y evitar que los resultados se vean alterados por entradas de peor o mejor caso sesgando los resultados, se decidió tomar una cantidad fija de 5000 instancias generadas de la forma antes descrita para cada n y luego tomar como estimador el promedio. En el siguiente gráfico se pueden observar los resultados del experimento:

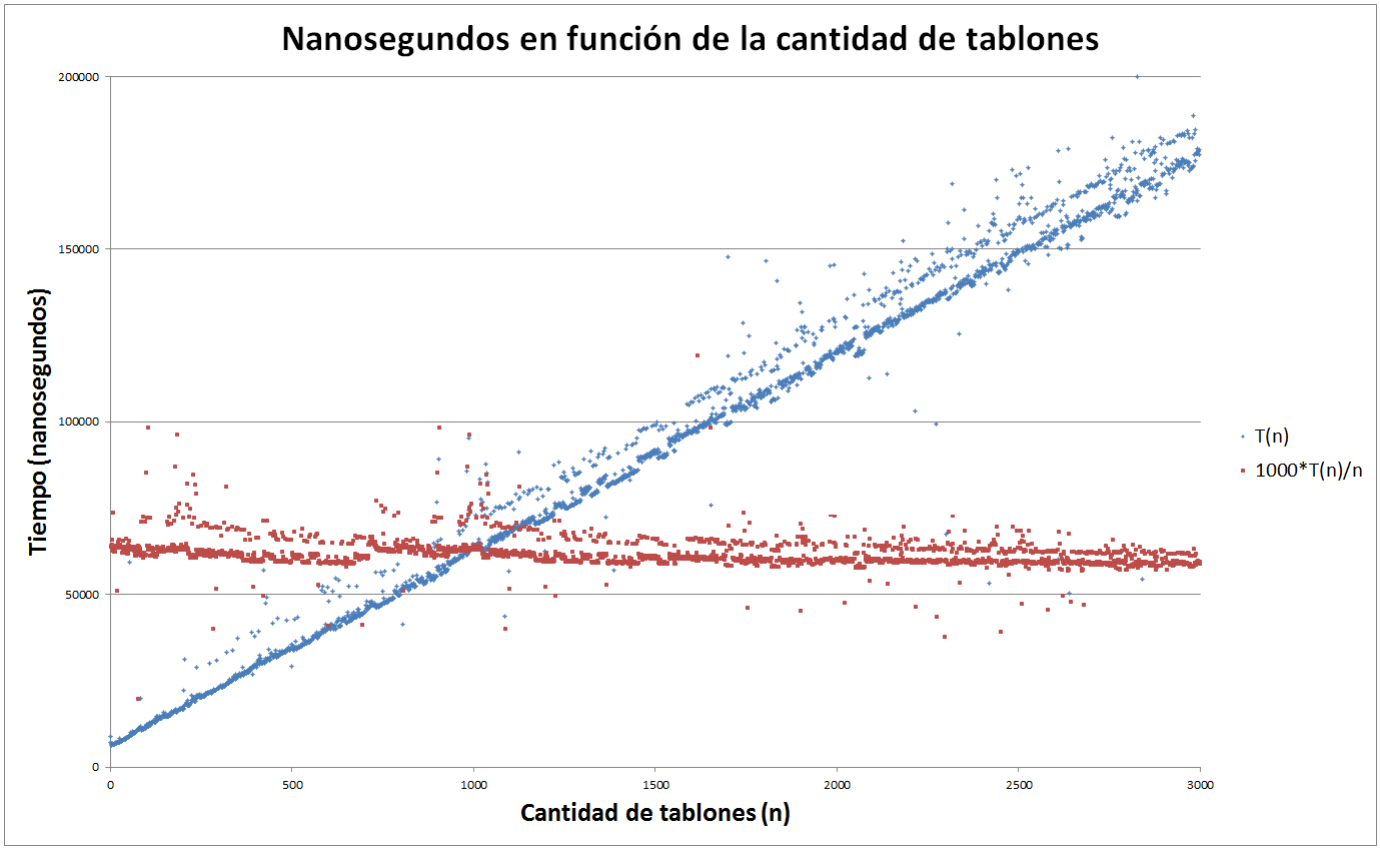


Gráfico de tiempos para entradas aleatorias. Notar que al graficar T(n)/n se la multiplica por 1000 para que el gráfico pueda ser apreciado con fácilidad

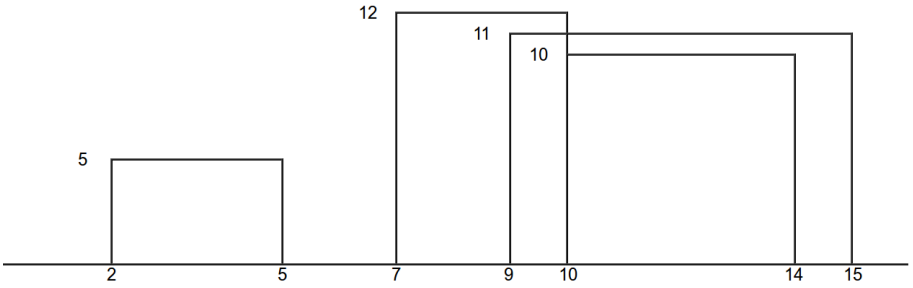
2 Horizontes Lejanos

2.1 Introducción

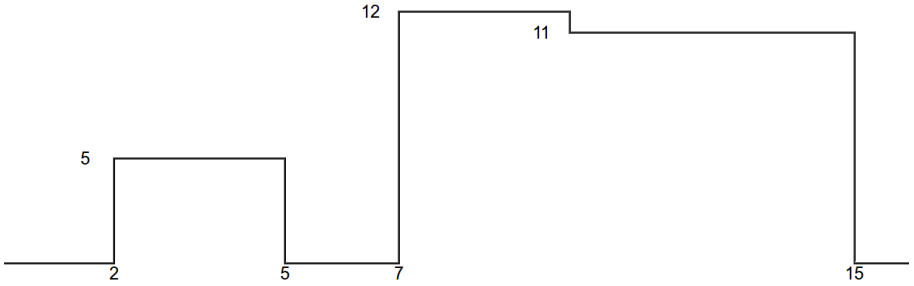
En este problema en particular hay que diseñar un software de arquitectura sobre el módulo de Edificios 2D. En este, se representan a los edificios como un conjunto de rectángulos apoyados sobre un mismo eje.
Ahora bien, el problema consiste en dibujar el perfil del conjunto de edificios (es decir el contorno que se vería desde el horizonte) eliminando las líneas ocultas.

El siguiente es un ejemplo del problema con su respectiva solución:

Entada
4
9 11 15
7 12 10
2 5 5
10 10 14



Al eliminar las lineas internas, la vista desde el "horizonte" será la siguiente:



La salida del algoritmo solo deberá contener una línea con una sucesión de pares de valores (x, y) representando cada cambio de altura en el horizonte, donde **x** representa la coordenada horizontal en donde se produce el cambio de altura e **y** representa la nueva altura a partir de **x**. Adicionalmente, nos piden que dichos pares estén ordenados de menor a mayor por su coordenada **x**.

Luego, la salida para el ejemplo anterior sera la siguiente: (2 5), (5 0), (7 12), (10 11), (15 0)

2.2 Desarrollo

Decidimos dividir el problema en dos partes:

- Identificar los "flancos ascendentes" → los cambios de altura hacia una altura mayor.
- Identificar los "flancos descendentes" → los cambios de altura hacia una altura menor.

En el ejemplo anterior:

Ascendentes:	(2 5), (7 12)
Descendentes:	(5 0), (10 11), (15, 0)

Luego combinaremos los dos para devolver la salida final.

Como en realidad, ambas partes son similares, ya que un cambio de altura en la posición **x** hacia una altura menor mirando el gráfico de izquierda a derecha, será un cambio de altura hacia una altura mayor mirando el gráfico de derecha a izquierda (en la misma posición x).
Luego las unicas diferencias entre ambas será el criterio de orden para el vector y la manera de chequear si dos efificios se superponen o no.

Pseudocodigo:

1 Ordenamos el vector de edificios.
2 Creamos un heap ordenado por mayor altura (vacio).
3 Agregamos al heap el **piso** (edificio de altura 0 que abarca desde la posicion 0 hasta el final)
4 Para cada edificio **a** del vector ya ordenado
5 (*) Tomamos el mayor elemento del heap (llamese **b**).
6 Si **a** y **b** no se superponen, sacamos a **b** del heap y volvemos a (*)
7 Si **a** es más alto que **b**, anotaremos un flanco ascendente donde comienza **a**
8 Agregamos **a** al heap.

Orden en FlancosAscendentes:

1 El que tenga menor izquierda primero
2 El que tenga mayor altura primero
3 El que tenga menor derecha primero

En el ejemplo anterior, los edificios quedarán en el siguiente orden: (2 5 5), (7 12 10), (9 11 15), (10 10 14)

Orden en FlancosDescendentes:

1 El que tenga mayor derecha primero
2 El que tenga mayor altura primero
3 El que tenga mayor izquierda primero

En el ejemplo anterior, los edificios quedarán en el siguiente orden: (9 11 15), (10 10 14), (7 12 10), (2 5 5)

Superposicion

A se superpone con **B** si la parte izquierda de **A** es menor que la parte derecha de **B**

Figura 1

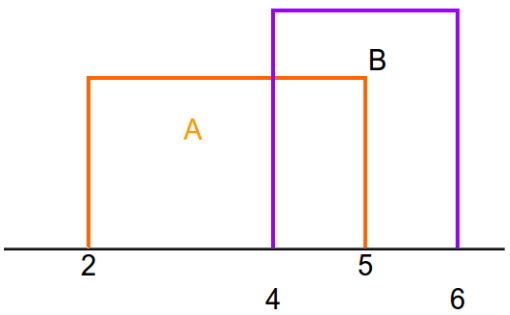
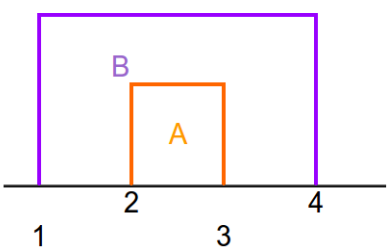


Figura 2



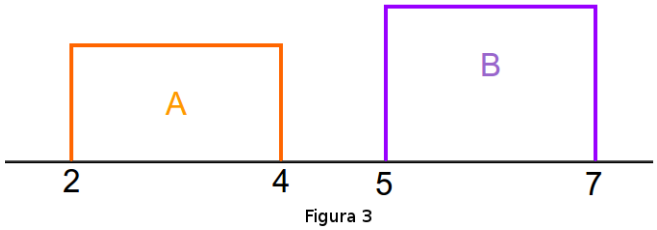


Figura 3

2.3 Demostración de correctitud

Antes que nada veamos que no puede existir un cambio de altura en el horizonte en una posición que no sea el inicio o el final de un edificio. Debido a que los edificios son rectangulares y con alturas > 0, las únicas posiciones donde pueden cambiar de altura son en su lado izquierdo o derecho.

Analizemos entonces cuando es que hay un cambio de altura hacia una altura mayor:
En primera instancia podríamos decir que cada vez que empieza un edificio **A** (el valor izq del mismo) hay un flanco ascendente, pero esta afirmación no es verdadera cuando A se "superpone" con otro edificio **B** más alto que A, ya que en el horizonte, B estaría tapando la primera parte de A.

Luego, analizemos nuestro algoritmo para ver que se cumple la siguiente condición:
Cambio de altura en A.izq $\Leftrightarrow \nexists B \in$edificios tal que $B.alt > A.alt$

Como podemos ver en el pseudocódigo, la idea general de resolución es ir tomando los edificios de a uno (de manera ordenada) y comparar el mismo con los que ya hallamos iterado.

Lo primero que hacemos es agregar "el piso" al heap, con esto nos aseguramos de tener siempre un elemento en el heap, ya que al abarcar desde el inicio hasta el final se estaría "superponiendo" con todos los edificios y por lo tanto nunca entrará al **if** que chequea si no se superponen (y por consiguiente nunca será sacado del heap). Con esto nos estamos asegurando la terminación del **while**.

Lo que hacemos entonces es ordenar los edificios de menor a mayor segun su valor de inicio (izq). De esta manera podremos iterarlos de manera ordenada.

Sean a_1 a_2 a_3 .. a_n todos los edificios ya ordenados.

Al iterar a_1, estamos comparando si se superpone con el piso (ya que es el único elemento del heap), esto siempre es cierto , y como la altura del piso es 0, siempre habrá un flanco ascendente al comienzo del primer edificio (**).

Al iterar a_i, lo que hacemos es chequear si es mas alto que todos los edificios que estan dentro del heap con los que se superpone. En particular en la primera posición del heap siempre estará el edificio más alto, llamese **B**, y tendremos 2 casos

- Si a_i se superpone con B, pasamos a chequear la altura
  - Si no se superponen, debemos seguir buscando en el heap un edificio con uno con el que si lo haga. Siempre existirá uno, el piso, que será el último por tener altura 0.
    Adicionalmente como B no se superpone con a_i (B.der $<$ a_i.izq), ningún a_k con k $>$ i lo hará tampoco, porque a_i.izq $\leq$ a_k.izq con k $>$ i (por el orden que le dimos). Seremos libres entonces de quitar a B del heap para evitar futuras comparaciones en vano.
- Si a_i es más alto que B, habrá un flanco ascendente, ya que dentro del heap todos los edificios serán mas bajos que B.
- Si B es más alto que a_i, no tendremos cambio de altura al comienzo del edificio a_i ya que el mismo estará "tapado" por B.

Como nuestro algoritmo recorre el inicio de todos los edificios, chequea que no exista un edificio más alto con el que se superponga y solo entonces agenda un cambio de altura. Y no puede existir un cambio de altura hacia una altura más alta en otra posicion que no sea al comienzo de un edificio, podemos decir que nuestro algoritmo registra todos los flancos ascendentes que existen en el horizonte.

El procedimiento para flancosDescendentes es analogo.

(**) Siempre y cuando $a_1.alt \geq b.alt \forall b \in edificios$ talque $b.izq == a.izq$.
Para evitar el caso anterior, lo que hacemos es poner como segunda razon de orden los edificios que tengan mayor altura, de esta manera si dos edificios comienzan en el mismo punto, tomaremos siempre el más alto primero y solo se registrará un solo cambio de altura.

2.4 Complejidad

Analizemos la complejidad temporal de flancosAscendentes sobre el extracto del código

```
1
2  void flancosAscendentes(const edificio_t& piso) {
3   vector<edificio_t> heap;
4   make_heap(heap.begin(), heap.end(), Comp());
5   heap.push_back(piso);
6   push_heap(heap.begin(), heap.end(), Comp());
7
8   sort(edificios.begin(), edificios.end(), orden);
9
10  for (size_t i = 0; i < n; i++) {
11   edificio_t edi = heap.front();
12   while (edificios[i].izq >= edi.der) {
13    pop_heap(heap.begin(), heap.end(), Comp());
14    heap.pop_back();
15    edi = heap.front();
16   }
17
18   if (edi.alt < edificios[i].alt)
19    solucion.push_back(make_pair(edificios[i].izq, edificios[i].alt));
20
21   heap.push_back(edificios[i]);
22   push_heap(heap.begin(), heap.end(), Comp());
23  }
24 }
```

- La creación del heap y el agregado del piso (lineas 3-6) tienen todas complejidad constante.
- La función sort (linea 8) tiene una complejidad de O(n*log(n))
- Las funciones push_heap (linea 22) y pop_heap (linea 13) tienen ambas complejidad O(log(k)) con k igual al tamaño del heap. En particular el heap nunca tendrá más de n elementos. Por lo tanto podemos acotarlo por O(log(n))
- Las funciones push_back (linea 21) y pop_back (linea 14) tienen complejidad constante
- Tanto la condicion del if (linea 18) como la del while (linea 12) son solo comparación de enteros, luego tienen complejidad O(1)
- El for (linea 10) itera exactamente n veces, una por cada edificio.

Si bien a simple vista deberiamos multiplicar la complejidad del while con la del for (porque que está anidado), si nos ponemos a analizar más en detalle podemos ver que por cada iteración del while estamos sacando un edificio del heap. Como a lo sumo podemos sacar n edificios del heap, solo puede haber como máximo n iteraciónes del while.

Es decir, en las n iteraciones que hace el for, solo estaremos entrando al while a lo sumo n veces.
Luego, la complejidad total de flancos ascendentes nos quedaría:

$$O(sort) + O(for) + O(while)$$

$$O(n * \log(n)) + O(n * O(push\_heap)) + (n * O(pop\_heap))$$
$$O(n * \log(n)) + O(n * \log(n)) + O(n * \log(n))$$
$$O(n * \log(n))$$

2.5 Casos de test

2.5.1 Todos los edificios iguales

Entrada
3
3 2 8
3 2 8
3 2 8

Salida
3 2 8 0

Testeamos el caso borde en que todos los edificios tienen igual izquierda, altura y derecha. Desde el horizonte solo se vería la figura de uno solo.

2.5.2 A termina en la posición x y B comienza en la posición x

Entrada
2
3 6 6
6 10 9

Salida
1 5 4 0 5 6 7 0

En este caso podriamos habernos confundido al estar registrando 2 cambios de altura en la posición x, uno hacia una altura 0 (cuando termina A) y otro hacia una altura b.alt (cuando empieza B), y esto no sería correcto ya que solo puede haber un cambio de altura en x.

También chequeamos la misma situación pero con a.alt y b.alt sean iguales. Desde el horizonte solo se ve un edificio más ancho.

Entrada
2
1 5 4
5 6 7

Salida
1 5 4 0 5 6 7 0

2.5.3 No hay superposición entre edificios

Entrada
4
1 2 3
5 6 7
10 1 15
20 5 21

Salida
1 2 3 0 5 6 7 0 10 1 15 0 20 5 21 0

Ningun edificio se superpone con otro. Desde el horizonte se los vería a todos perfectamente (no hay lineas ocultas)

2.5.4 Escalera

Entrada
3
1 7 3
1 5 5
1 3 7

Salida
1 7 3 5 5 3 7 0

Analizamos el caso en que dos o más edificios empiezan en la misma posición y tinen diferente altura. Para chequear el (**) en la sección de correctitud.

2.5.5 Uno adentro del otro

Entrada
2
1 8 5
3 1 4

Salida
1 8 5 0

Vimos que el algoritmo funcione cuando un edificio esta totalmente oculto por otro.

2.6 Experimentación

Al igual que en el ejercicio 1, les presentemos un análisis de complejidad en el ámbito experimental. Todos los puntos del gráfico son promedios de los tiempos por tamaño de 5000 instancias. Decidimos utilizar los siguientes conjuntos de datos para analizar:

1. Instancias **aleatorias**

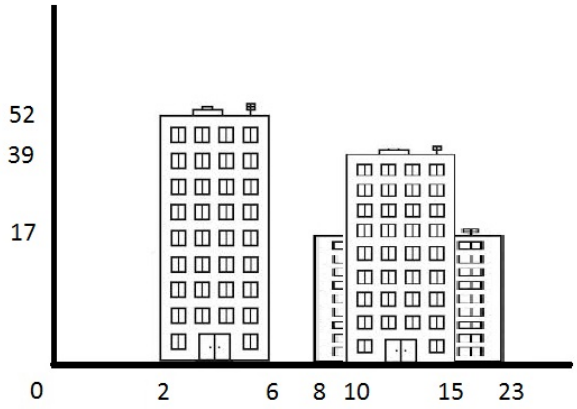


Figure 2

2. **Edificios** espaciados de a pares (es decir $\forall e1, e2 \in Edificios, e1 \neq e2, der(e1) < izq(e2) \bigwedge izq(e1) < izq(e2) \bigwedge der(e1) < der(e2)$)

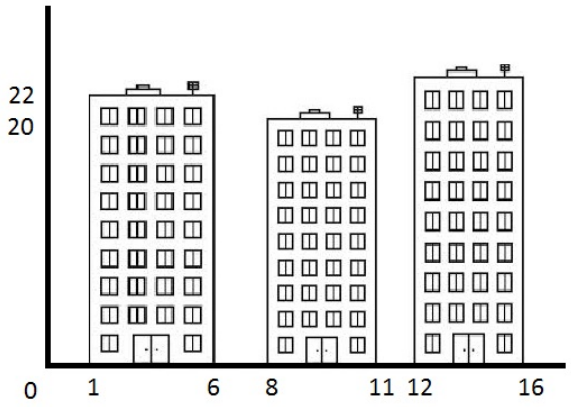


Figure 3

3. **Edificios** formando una escalera descendente, (es decir $\forall e1, e2 \in Edificios, e1 \neq e2, izq(e1) < izq(e2) \bigwedge der(e1) < der(e2) \bigwedge alt(e1) < alt(e2)$)

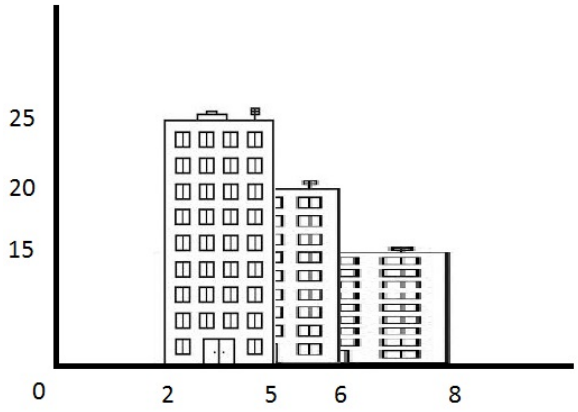


Figure 4

Para verificar que nuestra complejidad teórica se verifica en la práctica, lo que tratamos fue de "'linealizar"' las muestras. Es decir, si la muestra m_i es $O(nlogn)$ entonces graficando $\frac{m_i}{(nlogn)}$ deberíamos obtener una recta (esto se puede visualizar en la figura GG).

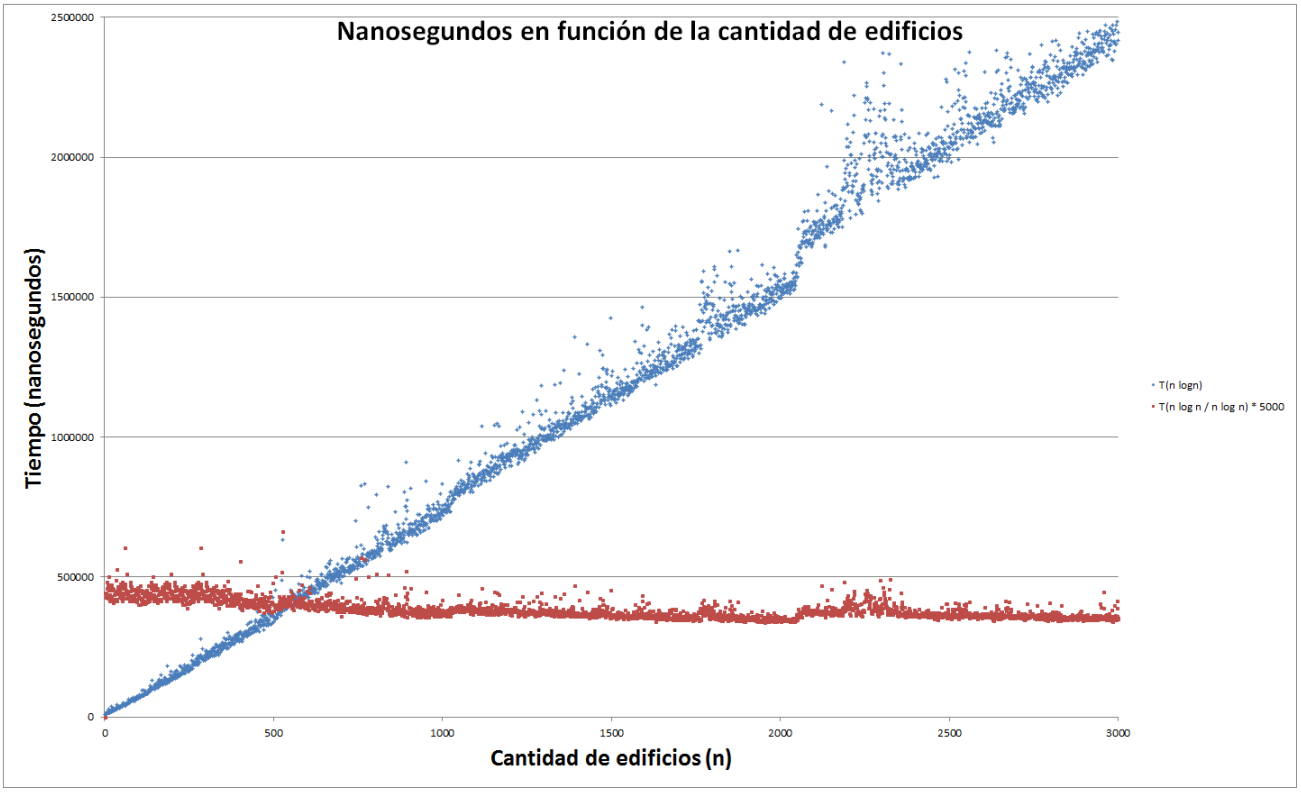


Figure 5: Comparación de tiempos

Notar que al graficar $T(nlogn)/nlogn$ se la multiplica por 5000 para que el gráfico pueda ser apreciado con fácilidad. Ahora bien, nos resulto interesante realizar experimentos para distintos tipos de entrada. Dados los conjuntos de entradas antes descriptos, mostramos los resultados que obtuvimos:

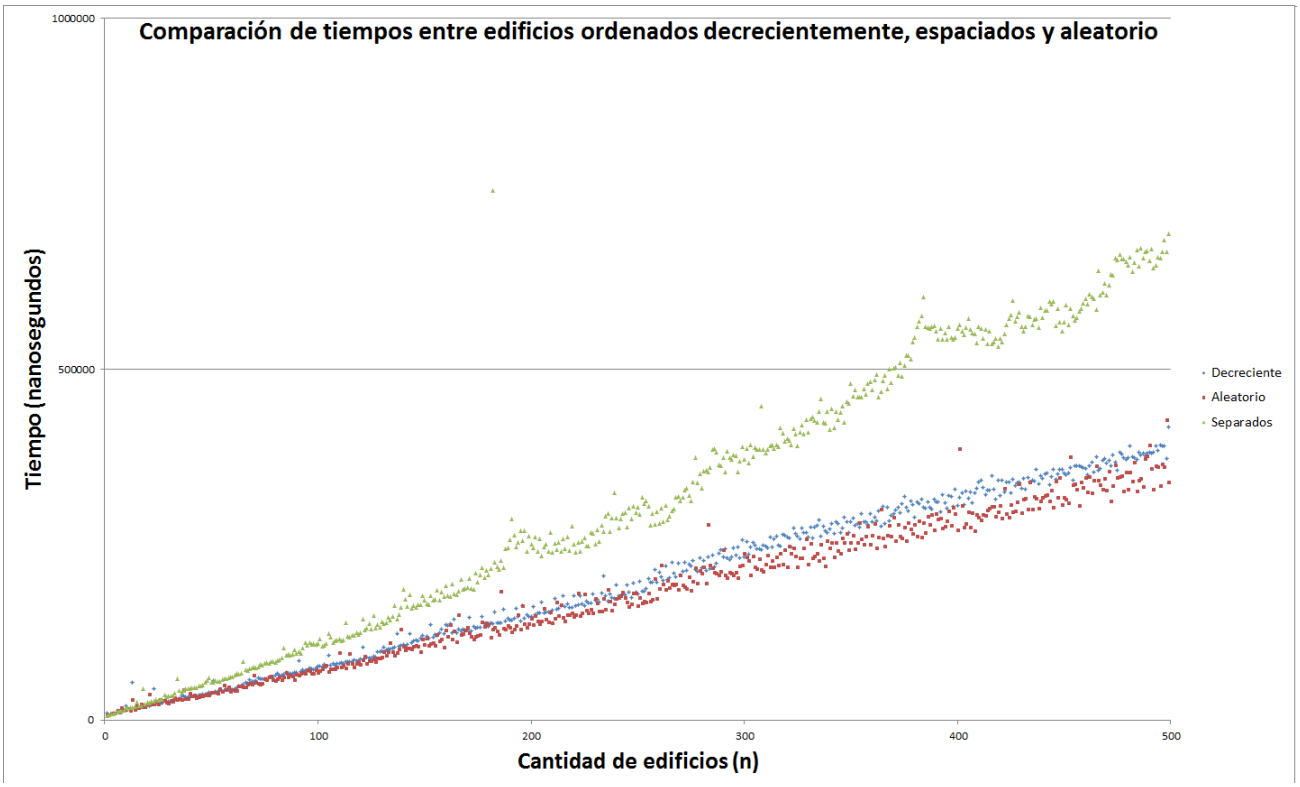


Figure 6: Comparación de tiempos

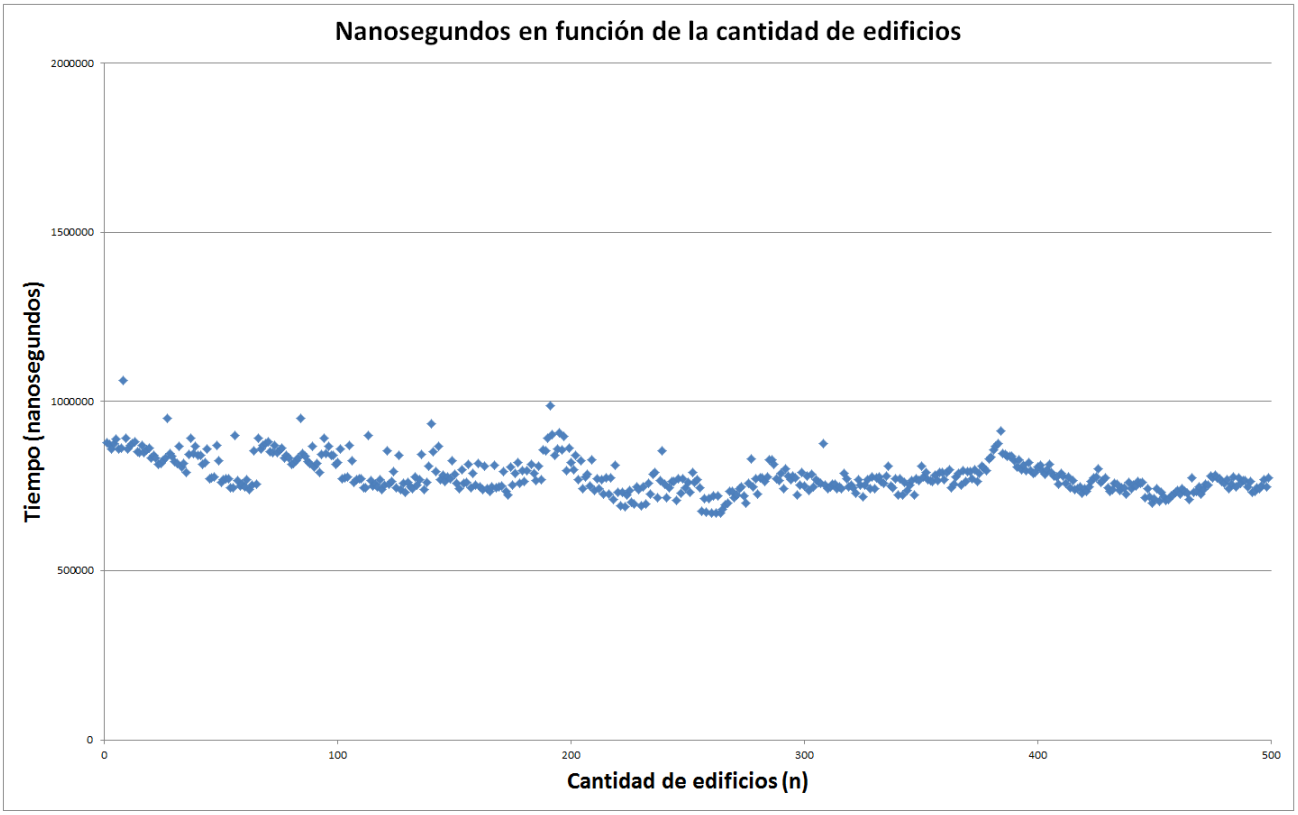


Figure 7: Comparación de tiempos

3 Biohazard

3.1 Introducción

Este problema trata sobre transportar una cantidad n de productos químicos en camiones especiales para su traslado, sabemos que cada par de productos p_i y p_j reaccionan de distinta forma y por eso tiene un coeficiente de peligrosidad denominado h_{ij}.
Por normas de seguridad el nivel de peligrosidad de los productos transportados en un camión no puede superar un valor M, dicho nivel se mide de la siguiente manera:

$$h(P) = \sum_{\substack{p_i, p_j \epsilon P \\ i<j}} h_{ij}$$

Nuestro objetivo es indicar como distribuir los productos químicos en los distintos camiones de forma de utilizar la menor cantidad de camiones posibles.

A continuación un ejemplo del problema con su solución:

Cantidad de químicos:	2	
M:	20	
Peligrosidad Quimicos	1	2
1	0	7
2	7	0

Figure 8: Para este ejemplo los dos quimicos entran en 1 camion

3.2 Desarrollo

Para la resolución de este problema se no pidió que usáramos la técnica de Backtracking, por lo que nuestra idea del algoritmo comenzó por ir probando para cada producto químico todas las posibilidades de guardarlo en n camiones (ya que como mucho metemos cada producto en un camión distinto), es decir la posibilidad de meter el químico i al camión 1 y el resto de los químicos en distintos camiones, la posibilidad de meter los químicos i y j con i != j en el camión 1 y el resto en otros camiones, etc. Eventualmente nos quedamos con la posibilidad generada en la cual se usan menos camiones y no se supere el umbral M.
Luego pensamos podas y estrategias para mejorar al algoritmo, se nos ocurrieron las siguientes:

- Si probamos todas las posibilidades de distribuir los químicos en los distintos camiones con los químicos i y j con i != j en el camión 1, no vamos a probar lo mismo para el camión 2. Es decir, elimina todas las permutaciones en otros camiones de la misma solucion.
  Esta poda es buena en los casos donde hay muchos químicos que transportar pues entre más químicos hay, mas ramas de soluciones aparecen.

- Si en una rama del algoritmo tenemos un camión donde ya metimos x productos químicos y al agregar un nuevo producto supera el umbral, entonces descartamos esa posibilidad.
  Esta estrategia es algo básica pero permite descartar ramas innecesarias las cuales no conducirán a una solución para el problema

Pseudocódigo:

1

3.3 Complejidad

Analicemos primero la naturaleza del problema. Tenemos n productos químicos los cuales tenemos que enviarlos a una fábrica mediante camiones. Nuestro objetivo es minimizar la cantidad de camiones contratados. Entonces podemos pensar un primer acercamiento en el cual comenzamos con n camiones. Esta va a ser nuestra solucion trivial, un camion por producto. A continuación probamos todas las maneras de distribuir los productos en estos camiones de manera tal que la cantidad de camiones contratados sea minima. Algo a resaltar es que un producto x pertenece a un camión nada mas (los productos son distinguibles). Entonces, cuantas maneras de distribuirlos tenemos?

Tenemos los productos p_1 p_2 y p_3. Pensemos esto como anagramas donde cada producto y camion es una letra. Sea p_i la letra asociada al producto i y — la letra asociada al camion. Entonces lo que queremos buscar son la cantidad de anagramas que podemos formar con las letras p_1—p_2—p_3 en donde el orden de los camiones no nos importa. Si n = # productos y m= # camiones la formula sería:

$\frac{(n+m-1)!}{(m-1)!}$

En nuestro caso, m = n quedando

$\frac{(2n-1)!}{(n-1)!}$

Claramente la complejidad es exponencial.

Esta versión es nuestro algoritmo de backtracking sin ninguna poda. Ahora bien, algo importante a notar es lo siguiente: supongamos que tenemos 5 productos, y la distribucion optima es poner el producto p_1 p_2 y p_3 en un camion y el producto p_4 y p_5 en otro. Como nosotros partimos de tener n posibles camiones (en este caso serían 5), se solaparian las distribuciones donde p_1 p_2 y p_3 estan en el camion 1 mientras que los productos p_4 y p_5 en el camion 2 con las que esten p_1 p_2 y p_3 en el camion 2 con p_4 y p_5 en el camion 3. Entonces un segundo acercamiento sería sacar "'todas las permutaciones"' posibles de una distribucion dada (poda ya descripta). Esto sería lo mismo que contar los posibles subconjuntos de un conjunto cuyos elementos son los productos. Es decir, a nuestro caso $\{p_1, p_2\} = \{p_2, p_1\}$ quedando una complejidad de $O(2^n)$.

3.4 Testing

Decidimos tomar las siguientes instancias del problema para corroborar los resultados:

Caso 1: 2 productos químicos y el límite de peligrosidad 2. $h_{1,2}$ es 1.
En este caso es obvio que ambos productos pueden viajar en 1 solo camión.

Caso 2: 3 productos químicos y el límite de peligrosidad 6. $h_{1,2} = 6, h_{1,3} = 7, h_{2,3} = 8$.
En un sólo camión no entran los 3 productos (tendría peligrosidad 21). Puedo poner el producto 1 y 2 en un camión, mientras que el 3 en otro camión. El mínimo en este caso es 2.

Caso 3: 3 productos químicos y el límite de peligrosidad 6. $h_{1,2} = 7, h_{1,3} = 7, h_{2,3} = 8$.
En este caso se puede ver que cada producto debe ir en camiones separados ya que la peligrosidad de a pares siempre supera el límite. En este caso el mínimo es 3.

Caso 4: 3 productos químicos y el límite de peligrosidad 8. $h_{1,2} = 2, h_{1,3} = 3, h_{2,3} = 3$.
En este caso se puede ver que todos los productos pueden viajar en un solo camión ya que la suma de sus peligrosidades es 8 y no supera el límite, que también es 8.

Estos casos de pruebas pueden encontrarse en **ej3/tester.cpp**.

3.5 Experimentación

A Ejercicio 1

.

```
1  void puentes(const std::vector<bool>& tablones, uint16_t c, std::vector<uint16_t>& saltos) {
2   uint16_t j = 0, k = 0, n = tablones.size();
3
4   // caso que puedo saltar todo de una
5   if (c < n){
6    // no puedo saltar todo de una
7    while (j < n){
8     for(k = c; k > 0; k--){
9      if ((j+k-1 < n) && (tablones[j+k-1] != false)){
10      saltos.push_back(j+k);
11      break;
12     }
13    }
14    if (k == 0){ // "No hay solucion";
15     saltos.clear();
16     break;
17    }
18
19    j += k;
20   }
21  }
22  else {
23   saltos.push_back(n);
24  }
25 }
```

B Ejercicio 2

.

```
1  typedef struct edificio_st {
2   size_t izq;
3   size_t alt;
4   size_t der;
5   edificio_st() : izq(0), alt(0), der(0) {}
6   edificio_st(size_t i, size_t a, size_t d) : izq(i), alt(a), der(d) {}
7  } edificio_t;
8
9  // Comparacion para el heap.
10 struct Comp {
11  bool operator()(const edificio_t& s1, const edificio_t& s2) {
12   return s1.alt <= s2.alt;
13  }
14 };
15
16 // de menor a mayor con los valores de la izq
17 bool orden(const edificio_t& a, const edificio_t& b) {
18  return ((a.izq < b.izq) ||
19    (a.izq == b.izq && a.alt > b.alt) ||
20    (a.izq == b.izq && a.alt == b.alt && a.der < b.der));
21 }
22
23 // de mayor a menor con los valores de la der
24 bool orden2(const edificio_t& a, const edificio_t& b) {
25  return ((a.der > b.der) ||
26    (a.der == b.der && a.alt > b.alt) ||
27    (a.der == b.der && a.alt == b.alt && a.izq > b.izq));
28 }
29
30 class Horizonte {
31 public:
32  Horizonte() : n(0) {
33  }
34
35  void resolver() {
36   flancosAscendentes(piso);
37   flancosDescendentes(piso);
38   sort(solucion.begin(), solucion.end());
39   // Para borrar los repetidos
40   solucion.erase(unique(solucion.begin(), solucion.end()), solucion.end());
41  }
42
43  void clear() {
44   solucion.clear();
45  }
46
47  size_t size() const {
48   return n;
49  }
50
51  friend ostream & operator<<(ostream &os, Horizonte& h);
52  friend istream & operator>>(istream &is, Horizonte& h);
53
54 protected:
55  size_t n;
56  vector<edificio_t> edificios;
57  vector<pair<size_t, size_t> > solucion;
58  edificio_t piso;
59
60  void flancosAscendentes(const edificio_t& piso) {
61   vector<edificio_t> heap;
62
63   make_heap(heap.begin(), heap.end(), Comp()); // Notar que al comienzo heap esta vacio.
64   heap.push_back(piso);
65   push_heap(heap.begin(), heap.end(), Comp());
66
67   sort(edificios.begin(), edificios.end(), orden);
68
69   for (size_t i = 0; i < n; i++) {
70    edificio_t edi = heap.front();
```

```
71      while (edificios[i].izq >= edi.der) {
72       pop_heap(heap.begin(), heap.end(), Comp());
73       heap.pop_back();
74       edi = heap.front();
75      }
76
77      if (edi.alt < edificios[i].alt)
78       solucion.push_back(make_pair(edificios[i].izq, edificios[i].alt));
79
80      heap.push_back(edificios[i]);
81      push_heap(heap.begin(), heap.end(), Comp());
82     }
83    }
84
85    void flancosDescendentes(const edificio_t& piso) {
86     vector<edificio_t> heap;
87
88     make_heap(heap.begin(), heap.end(), Comp());
89     heap.push_back(piso);
90     push_heap(heap.begin(), heap.end(), Comp());
91
92     sort(edificios.begin(), edificios.end(), orden2);
93
94     for (size_t i = 0; i < n; i++) {
95      edificio_t edi = heap.front();
96      while (edificios[i].der < edi.izq) {
97       pop_heap(heap.begin(), heap.end(), Comp());
98       heap.pop_back();
99       edi = heap.front();
100     }
101
102     if (edi.alt < edificios[i].alt)
103      solucion.push_back(make_pair(edificios[i].der, edi.alt));
104
105     heap.push_back(edificios[i]);
106     push_heap(heap.begin(), heap.end(), Comp());
107    }
108   }
109  };
```

C Ejercicio 3

```
typedef struct sustancia_st {
 uint16_t numero;
 vector<uint16_t> coeficiente;
 sustancia_st(uint16_t i, uint16_t n) : numero(i)  {
  coeficiente.resize(n);
 }
} sustancia_t;

class Camion {
public:
 vector<sustancia_t*> sustanciasTransportadas; // Sustancias transportadas por el camion
 uint16_t peligrosidad; // Nivel de peligrosidad de las sustancias transportadas

 /// Constructor
 Camion() : peligrosidad(0) {
 }

 /// Agrega sust al camion y le sube la peligrosidad
 void agregarSustancia(sustancia_t * sust) {
  peligrosidad += calcularPeligrosidad(sust);
  sustanciasTransportadas.push_back(sust);
 }

 /// Saca "sust" del camion y baja la peligrosidad. Siempre es la ultima sustancia agregada
     por el contexto en que la uso.
 void sacarSustancia(sustancia_t * sust)  {
  sustanciasTransportadas.pop_back();
  peligrosidad -= calcularPeligrosidad(sust);
 }

 /// Chequea si al agregar "sust" al camion no se pasa el umbral de peligrosidad
 bool noEsPeligroso(const uint16_t &umbral, const sustancia_t * sust) {
  uint16_t x = calcularPeligrosidad(sust) + peligrosidad;
  return (x <= umbral);
 }

private:
 /// Calcula la peligrosidad de agregar "sust" al camion (no la agrega, solo la calcula)
 uint16_t calcularPeligrosidad(const sustancia_t * sust) {
  uint16_t suma = 0;
  for (auto& st: sustanciasTransportadas)
   suma += sust->coeficiente[st->numero];
  return suma;
 }
};

class Biohazard {
public:
 Biohazard() :
  n(0), m(0),
  minimosCamionesUsados(numeric_limits<uint16_t>::max()),
  usadas(0) {
 }

 void backtracking() {
  camiones.resize(n);
  recursion(0);
  return;
 }

 void clear() {
  camiones.clear();
  solucion.clear();
  transportados.clear();
  usadas = 0;
 }

 uint16_t size() const {
  return n;
 }
```

```
70
71   friend ostream & operator<<(ostream &os, Biohazard& b);
72   friend istream & operator>>(istream &is, Biohazard& b);
73
74  protected:
75   uint16_t n;
76   uint16_t m;
77   uint16_t minimosCamionesUsados;
78
79   vector<sustancia_t> productos;
80   vector<Camion> camiones;
81   vector<uint16_t> transportados;
82   uint16_t usadas; /// Cantidad de sustancias utilizadas
83   vector<uint16_t> solucion;
84
85   void recursion(const uint16_t &x) {
86    uint16_t contador = contar();
87
88    if (contador < minimosCamionesUsados) {
89
90     if (todosUsados()) {
91      minimosCamionesUsados = contador;
92      solucion = transportados;
93     }
94
95     for (uint16_t i = x; i < n; i++) { // i itera las sustancias
96      if (transportados[i] == 0)   { // Si no esta en ningun camion...
97       sustancia_t * sust = &productos[i];
98       for (uint16_t c = 0; c < n; c++) { // c itera los camiones
99        Camion * truck = &camiones[c];
100        // Si no supera el umbral de peligrosidad al agregar la sustancia sust al truck...
101       if (truck->noEsPeligroso(m, sust))
102        // Agrego sust al camion{
103        truck->agregarSustancia(sust);
104        usar(sust->numero, c + 1); // Agendo sust como ya transportada
105        recursion(i);
106        // Vuelvo a dejar como si no lo lo hubiese agregado, para probar con la siguiente
               sustancia
107        desUsar(sust->numero);
108        truck->sacarSustancia(sust);
109       }
110      }
111     }
112    }
113   }
114
115  }
116
117  /// Agenda "sust" como ya transportada por el camion "c"
118  void usar(uint16_t sust, uint16_t c) {
119   transportados[sust] = c;
120   ++usadas;
121  }
122
123  /// Agenda "sust" como no transportada por ningun camion
124  void desUsar(uint16_t sust) {
125   transportados[sust] = 0;
126   --usadas;
127  }
128
129  /// Devuelve true <==> estan todas las sustancias en algun camion
130  inline bool todosUsados() {
131   return (usadas == n);
132  }
133
134  /// Para saber cuantos camiones se usaron y tambien sirve para un par de "podas"
135  uint16_t contar() {
136   uint16_t res = 0;
137   for (uint16_t i = 0; i < n; i++)
138    res = max(res, transportados[i]);
139   return res;
140  }
141 };
```