Algoritmos y Estructuras de Datos III

Trabajo Práctico N°1

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

Integrante	LU	Correo electrónico
Izcovich, Sabrina	550/11	sizcovich@gmail.com
Garcia Marset, Matias	356/11	matiasgarciamarset@gmail.com
Orellana, Ignacio	229/11	nacho@foxdev.com.ar
Vita, Sebastián	149/11	sebastian_vita@yahoo.com.ar

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

Este trabajo práctico consiste en la resolución de ciertos problemas algorítmicos delimitados por algunas restricciones impuestas por la cátedra, como por ejemplo el orden de complejidad máximo de los mismos entre otras. Para justificar las implementaciones de los problemas en cuestión, fue necesaria la utilización de herramientas lógico-matemáticas que serán mencionadas a lo largo del desarrollo de cada ejercicio.
Para comprobar que nuestras soluciones resolvieran correctamente los problemas propuestos, debimos dividir el análisis de los mismos en secciones a fin de estudiar minuciosamente cada característica de éstos. Estas secciones se dividen de la siguiente forma:

- **Problema a resolver:** En esta sección, nos encargamos de describir detalladamente el problema a resolver dando ejemplos del mismo y sus soluciones.
- **Resolución coloquial:** En esta parte, nos dedicamos a explicar de forma clara, sencilla, estructurada y concisa las ideas desarrolladas para la resolución del problema en cuestión. Para ello, decidimos utilizar pseudocódigo y lenguaje coloquial combinando ambas herramientas de manera adecuada.
- **Demostración de correctitud:** Utilizamos este apartado para justificar que el punto anterior resuelve efectivamente el problema y demostramos formalmente la correctitud del mismo.
- **Complejidad del algoritmo:** En esta sección, nos ocupamos de deducir una cota de complejidad temporal del algoritmo propuesto en función de los parámetros considerados como correctos. Por otro lado, justificamos por qué el algoritmo desarrollado para la resolución del problema cumple con la cota dada.
- **Código fuente:** En esta parte, presentamos las funciones relevantes del código fuente que implementa la solución propuesta. Para ello, decidimos utilizar el lenguaje `C++` dado que éste cuenta con la librería *stl* que proporciona las estructuras necesarias para la realización de dicha tarea.
- **Instancias posibles:** Este sector presenta un conjunto de instancias que permiten verificar la correctitud del programa implementado cubriendo todos los casos posibles y justificando la elección de los mismos. Dichas instancias fueron evaluadas por el algoritmo realizado y los resultados obtenidos fueron comprobados.
- **Testing:** Por último, los tests consistieron en experimentaciones computacionales utilizadas para medir la performance del programa implementado. Para ello, debimos preparar un conjunto de casos de test que permitieran observar los tiempos de ejecución en función de los parámetros de entrada que fueran relevantes. De este modo, nos encargamos de generar instancias aleatorias como también con instancias particulares. Para que los resultados fueran visibles y claros, utilizamos una comparación gráfica entre los tiempos medidos y la complejidad teórica calculada.

2. Ejercicio 1

2.1. Problema a resolver

El siguiente ejercicio consiste en hallar una manera de implementar un sistema proporcionado respetando una cota determinada de orden de complejidad.
El problema se sitúa en un centro de distribución de correo que recibe paquetes todos los días cuyo destino final es la sede central de la empresa. Para el transporte de los mismos, éstos son cargados a camiones de igual capacidad. El encargado de logística, Pascual, tiene un sistema que utiliza desde hace años para agilizar la carga de los camiones asegurando el uso de una baja cantidad de los mismos para el envío de paquetes al final del día. Dicho sistema consiste en tomar los paquetes y ubicarlos en algún camión que ya tenga paquetes dentro, eligiendo, entre éstos, el que menos peso tenga cargado hasta ese momento. Si el peso del paquete permite que éste sea cargado en ese camión, se lo ubica allí, sino, se lo incluye en un nuevo camión.
El problema a resolver se basa en escribir un algoritmo que tome los pesos de los paquetes que hay que acomodar e indique cuántos camiones se van a utilizar y cuánto peso se cargará en cada uno de ellos al final del día considerando el sistema de Pascual. Para esto, se respeta el orden de llegada de los paquetes a medida que ingresan.
Las consideraciones a tener en cuenta son que los camiones tienen todos la misma capacidad de carga, que la cantidad disponible de los mismos alcanza para transportar todos los paquetes y que el peso de un paquete no supera la capacidad de carga de un camión. Del mismo modo, el tamaño de los paquetes no es tenido en cuenta, sino que se los identifica por su peso. Además, es importante aclarar que las cargas son valores enteros positivos.

Formatos de entrada y salida:

La entrada contiene varias instancias del problema. Cada instancia consta de una línea con el siguiente formato:

$$L\ n\ p_1\ p_2\ ...\ p_n$$

donde L es el límite de carga de los camiones, n es la cantidad de paquetes a acomodar y p_1**, ...,** p_n son los pesos de cada paquete en el orden en el que deben ser almacenados.

La salida debe contener una línea por cada instancia de entrada, con el siguiente formato:

$$k\ c_1\ c_2\ ...\ c_n$$

donde k es la cantidad de camiones utilizados y c_1**, ...,** c_k es el peso que se cargó en cada uno de los k camiones al final del día.

En lo que sigue, presentaremos dos ejemplos sobre el sistema impulsado por Pascual:

- **Ejemplo 1:**

  En este ejemplo, decidimos develar un caso en el que fuera agregado un paquete a un camión ya cargado. Por otro lado, quisimos contrarrestarlo insertando un paquete con una carga que superaba la capacidad del camión creado anteriormente. Por último, nos pareció importante mostrar un caso en el que al agregar un nuevo paquete, si bien un nuevo camión había sido creado, éste era colocado en el camión cuya carga fuera la menor.

  Formato de entrada:

  100 5 20 40 80 15 100

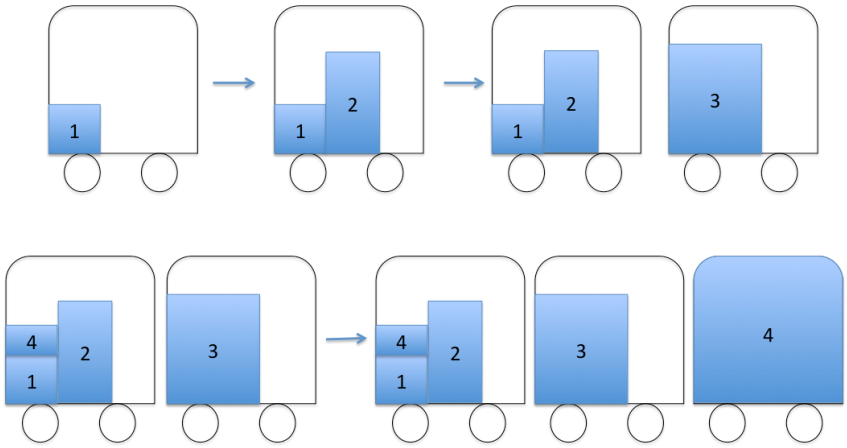

Figura 1: Ejemplo 1.

Formato de salida:

3 75 80 100

- ## Ejemplo 2:

En este ejemplo, quisimos mostrar lo que ocurría cuando cada paquete supera el 50 % de la carga disponible en un camión.

Formato de entrada:

150 3 80 75 82

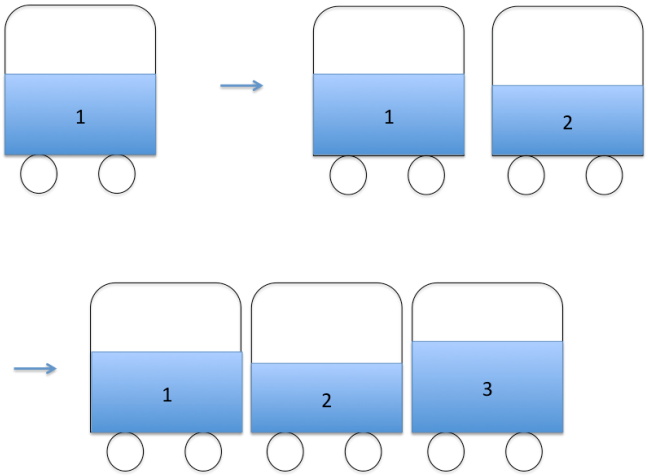

Figura 2: Ejemplo 2.

Formato de salida:

3 80 75 82

2.2. Resolución coloquial

Al analizar el problema a resolver, nos percatamos de que el algoritmo de Pascual corresponde a un *algoritmo goloso*. Dicho algoritmo consiste en la construcción de una solución seleccionando, en cada paso, la mejor alternativa sin considerar las implicancias de ésta.
Por otro lado, dicha técnica de diseño algorítmico resulta fácil de implementar, suele ser eficiente y permite construir soluciones razonables.

En el caso del algoritmo de Pascual, el algoritmo goloso consiste en tomar cada instancia de la entrada y ubicarla en el lugar más conveniente en ese momento determinado. Esto significa que, dada la i-ésima caja ingresada en un mismo día, ésta es ubicada en el camión se encuentra menos cargado en ese momento.

El pseudocódigo ideado para resolver el problema es el siguiente:

Algorithm 1: Algoritmo de Pascual

Input: Entero limite, Paquetes *ps*
Output: cantidadDeCamiones, listaDePesos

```
1  if ps = ∅ then devolver 0, ListaVacia
2  Camiones ca ← {Camion cNuevo(0)}
3  cantidadDeCamiones := 1
4  for Paquete p ∈ ps do
5  |  Camion c := min(ca)
6  |  if peso(p)+capacidad(c) ≤ limite then
7  |  |  capacidad(c) + = peso(p)
8  |  else
9  |  |  ca ← Camion cNuevo(0)
10 |  |  capacidad(cNuevo) = peso(p)
11 |  |  cantidadDeCamiones + 1;
12 |  end
13 end
14 devolver cantidadDeCamiones, ca
```

La función *cNuevo* crea un nuevo camión cuya carga inicial es 0.

2.3. Demostración de correctitud

Para demostrar que nuestro algoritmo se corresponde con el algoritmo de Pascual, decidimos realizar una comparación, de forma paulatina, entre lo que define el algoritmo de Pascual y nuestro pseudocódigo.

En primer lugar, podemos ver que una vez que ingresa un paquete, se analiza si éste entra en el camión menos cargado ya conteniendo algún paquete. En nuestro pseudocódigo, ésto corresponde a la línea 6.

Luego, si el peso del paquete permite cargarlo en ese camión, se lo carga allí (línea 7). Caso contrario, se utiliza un nuevo camión para el paquete ingresado dándole como carga inicial la correspondiente a este último (líneas 9 y 10).

Por último, podemos observar que la cantidad de camiones que se utilizan al final del día se acumula en la variable *cantidadDeCamiones* y los pesos de camiones en orden se devuelven a través de *ca*.

2.4. Complejidad del algoritmo

Dado que el algoritmo presentado en la sección anterior puede ser fácilmente representado por una estructura de tipo *heap*, decidimos implementarlo sobre una cola de mínima prioridad para poder utilizar todas las funciones que nos provee *stl* sobre ésta.

Una cola de prioridad consiste en una estructura de datos en la que sus elementos son atendidos según la prioridad que éstos tengan asociada. Dicha estructura se caracteriza por admitir inserciones de nuevos elementos y la consulta y eliminación del elemento de mínima prioridad.
Las colas de prioridad suelen ser útiles para resolver algoritmos golosos ya que éstos suelen tener una iteración principal, y una de las tareas a realizar en cada una de dichas iteraciones es seleccionar un elemento de entre varios que minimiza un cierto criterio de optimalidad local.
Para adaptar el problema de Pascual a la cola de prioridad, decidimos que cada elemento debía representar la carga correspondiente a un camión determinado en la instancia dada junto con el orden de creación del mismo.

Para el análisis de complejidad de nuestro algoritmo, llamamos n a la cantidad de paquetes que ingresan como parámetro de entrada.

Debido a que el algoritmo realizado consiste en una serie de operaciones básicas, comparaciones y asignaciones (cuyas complejidades son constantes), la complejidad del mismo puede determinarse a partir de los ciclos que éste realiza.

Nuestro algoritmo fue implementado sobre una cola de mínima prioridad, luego, la extracción del camión menos cargado se realiza en $\mathcal{O}(log\ n)$[1] gracias a la función **pop()** provista por *stl*.

Por otro lado, la inserción de un nuevo camión (**push()**) se realiza en $\mathcal{O}(log\ n) + \mathcal{O}(1) = \mathcal{O}(log\ n)$[1]. Esto se debe a que la cola de prioridad está implementada sobre un vector cuyo tamaño es equivalente a la cantidad de paquetes (correspondiente al peor caso, e.g. *Ejemplo* 2). Para lograr esto, debimos agregar un camión por cada paquete a la cola de prioridad y luego retirarlos uno por uno para forzar la asignación de memoria al contenedor de la cola de prioridad y asi evitar que al agregar un nuevo elemento se tenga que pedir un nuevo espacio en memoria mas grande y no sea constante. ($\mathcal{O}(n)$).

El ciclo del código recorre los paquetes ingresados como parámetro. Debido a que los pasos que realiza el mismo constan de un condicional cuyos resultados consisten en asignaciones y operaciones simples, y la función *push*() y *pop*() ($\mathcal{O}(log\ n)$), la ejecución del mismo tiene un orden de $\mathcal{O}(n\ log\ n)$.

Además del ciclo, el algoritmo llama a *top*() ($\mathcal{O}(1)$[1]) y, luego, recorre la cola de prioridad en un ciclo para ordenar los camiones por orden de creación ($\mathcal{O}(n)$). El orden de estos pasos del algoritmo quedan envueltos por la complejidad del ciclo principal. Por lo tanto, pudimos comprobar que la complejidad temporal del algoritmo es $\mathcal{O}(n\ log\ n)$, siendo ésta estrictamente menor a $\mathcal{O}(n^2)$, que fue lo requerido por la cátedra.

[1] http://www.cplusplus.com/reference/queue/priority_queue/

2.5. Código fuente

```
Camion c = ca.top();
if ((ps[i]+c.second) <= limite){
    c.second += ps[i];
    ca.pop(); //elimina el camion menos cargado y en la siguiente linea lo vuelve a agregar
    ca.push(c);
}else{
    ca.push(make_pair(cantCamiones, ps[i])); //agrega un nuevo camion
    ++cantCamiones;
}
```

Figura 3: Pasos que se realizan al ingresar un nuevo paquete.

```
while(!ca.empty()){
Camion c = ca.top();
vectorCamionesOrdenados[c.first] = c.second;
ca.pop();
}
```

Figura 4: Ordenamiento de los camiones con respecto al momento en el que fueron creados.

2.6. Instancias posibles

Para verificar la correctitud de nuestro programa, dispusimos variar estratégicamente las instancias de entrada al ejecutarlo.

- En primer lugar, ejecutamos el programa ingresando paquetes cuyos pesos fueran iguales al límite de los camiones. De este modo, nos aseguramos de no estar cometiendo errores de cotas. El resultado obtenido fue el esperado: una cantidad de camiones equivalente a la cantidad de paquetes ingresados cuyos pesos correspondían exactamente al de cada paquete.

  Parámetro de entrada:

  10 3 10 10 10

  Parámetro de salida:

  3 10 10 10

- Por otra parte, probamos no ingresar ningún paquete para corroborar que la salida estuviera comprendida por una cantidad de camiones nula y una lista vacía.

  Parámetro de entrada:

  0 0

  Parámetro de salida:

  0

- También, quisimos comprobar qué ocurría al tener dos o más camiones con la misma carga, con el fin de corroborar el orden de camiones de salida. Pudimos constatar que el orden de salida es el correcto, es decir que no importa la carga de los camiones sino que éstos son devueltos de acuerdo a su orden de creación.

Parámetro de entrada:

4 3 2 3 2

Parámetro de salida:

3 3 4 3

De este modo, logramos abarcar los casos límite en los que la implementación pudiera haber encontrado algún problema. Dado que los resultados obtenidos fueron los esperados, determinamos que para todas las instancias válidas posibles de entrada nuestra implementación resulta correcta.

2.7. Testing

Para realizar las pruebas de complejidad, generamos instancias aleatorias de pesos de cajas alterando la cantidad de las mismas pero manteniendo estático el peso máximo de carga de los camiones. Estas instancias fueron generadas en *Matlab* con la función randn de forma tal a poder acotarlas por la capacidad de carga de los camiones. De este modo, logramos medir las pruebas de nuestro algoritmo para comprobar que la complejidad correspondiera con la mencionada anteriormente.

3. Ejercicio 2

3.1. Problema a resolver

El siguiente ejercicio consiste en exponer un algoritmo capaz de devolver, dado un conjunto de intervalos, el subconjunto máximo de los que no se solapan entre sí. Luego, el algoritmo debe poder tomar las fechas ofrecidas por los profesores del programa de Profesores Visitantes de la FCEyN para sus cursos e indicar qué cursos deberían elegirse para maximizar la cantidad de cursos en el ciclo. Cada intervalo corresponde a la fecha en que un profesor dictará su curso. El primer elemento del intervalo corresponde a la fecha de inicio y el segundo a la fecha de fin, siendo siempre la de fin mayor o igual a la de inicio. Para la simplificación del manejo de los datos, éstos se representan con números enteros positivos.

Formatos de entrada y salida:

La entrada contiene varias instancias del problema. Cada instancia consta de una línea con el siguiente formato:

$$n\ i_1\ f_1\ i_2\ f_2\ ...\ i_n\ f_n$$

donde n es la cantidad de cursos ofrecidos por los profesores (numerados de 1 a n) y los valores $[i_1, f_1],\ ...,\ [i_n, f_n]$ representan los días de inicio y fin de cada uno de los n cursos. Todos los datos son enteros positivos. La entrada concluye con una línea comenzada por # que no debe ser procesada.

La salida debe contener una línea por cada instancia de entrada, donde se listan los números de los cursos elegidos para el ciclo de cursos.

En lo que sigue, presentaremos dos ejemplos sobre cómo debería comportarse nuestro algoritmo:

Ejemplos:

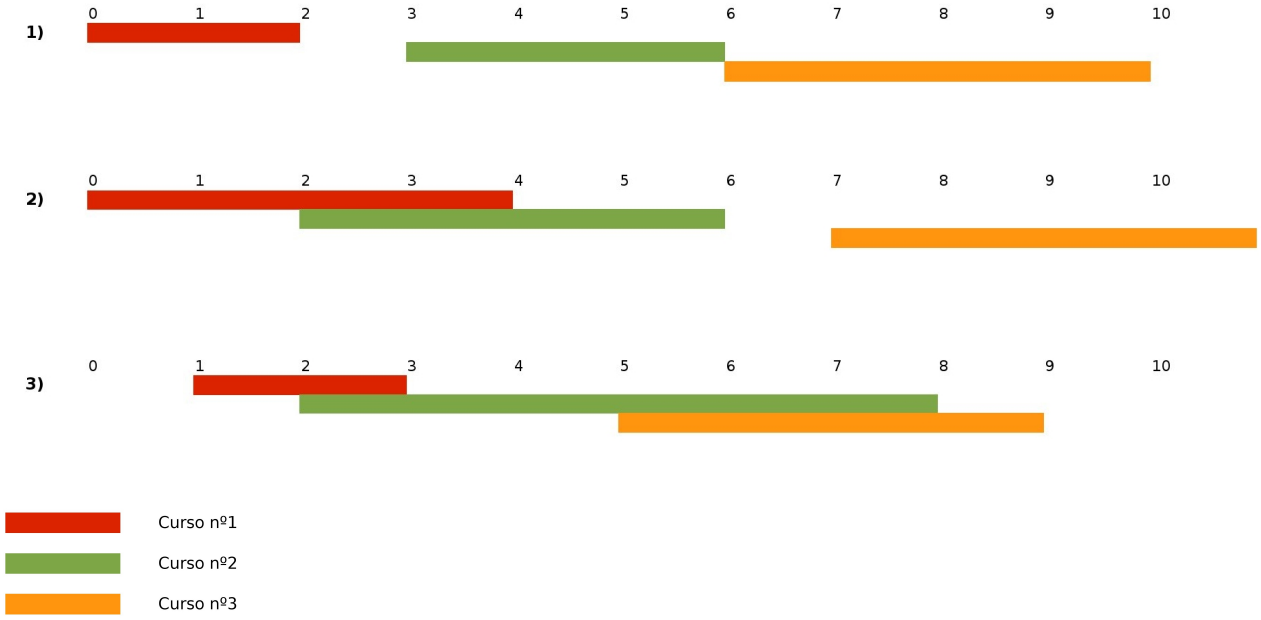

Figura 5: Ejemplos.

En el ejemplo **1)**, decidimos mostrar un caso en el que no se solapara ningún curso. En esta ocasión, puede observarse que, dado que hay una solución óptima, ésta es única.

Formato de entrada:

3 0 1 3 5 6 9

Formato de salida:

1 2 3

En el ejemplo **2)**, resolvimos develar un caso en el que hubiera un único solapamiento con el fin de mostrar una situación cuyas soluciones óptimas fueran más de una. En esta oportunidad, hay dos salidas posibles que son solución del problema. Debido que nuestro algoritmo almacena, en primer lugar, los intervalos cuya fecha de fin es la más alta, dicho intervalo es el prioritario frente a cualquiera que se le solape.

Formato de entrada:

3 0 3 2 5 7 10

Formato de salida:

2 3

En el ejemplo **3)**, preferimos mostrar un caso en el que se solaparan todos los cursos. En esta situación, cualquier curso del ciclo es solución del problema pues es la máxima cantidad de intervalos que no se solapan.

Formato de entrada:

3 1 2 2 7 5 8

Formato de salida:

3

3.2. Resolución coloquial

Resolvimos el problema ordenando los cursos por fecha de fin y tomando siemrpe el menor curso (en términos de finalización) y agregandolo a la solucion siempre y cuando no se solape con los cursos que ya agregue como solucion.

Al analizar el problema a resolver, nos percatamos de que el algoritmo de maximización de intervalos no solapados corresponde a un *algoritmo goloso*. El pseudocódigo que describe el algoritmo es el siguiente:

Algorithm 2: Algoritmo de Maxima cantidad de intervalos no solapados

```
Input: Cursos CS
Output: listaDeCursos
1 if #CS == 0 then
2 |  devolver ListaVacia
3 else
4 |  CS = OrdenarPorFin(CS)
5 |  Lista res = filtrarSolapamientos(CS)
6 |  devolver #res ++ ordenar(res)
7 end
```

Algorithm 3: filtrarSolapamientos

```
Input: Cursos CS
Output: listaDeCursos
1 Curso anterior = Primer curso de CS
2 Lista res
3 for Curso c en CS do
4 |  if c no se solapa con anterior then
5 |  |  anterior := c
6 |  |  res ← c
7 |  end
8 end
9 devolver res
```

donde *Cursos* es una secuencia conteniendo la información (fecha de inicio y de fin) de los cursos del ciclo. *ordenarPorFin* es un algoritmo que toma los cursos y los ordena de acuerdo a su fecha de finalización. Por último, "no se solapa"verifica que la fecha de inicio del intervalo que se quiere agregar no se superponga con la fecha de fin del agregado precedentemente
Una observacion que hicimos:
O1: Los dias de inicio son menores a los dias de finalizacion de cada curso.

Modelo formal:
Sea v un intervalo, este intervalo está representado por dos valores: v_i es el inicio del intervalo y v_f es el fin del intervalo. Sea C un conjunto de intervalos. Nuestro modelo del problema mapea a los cursos como intervalos, donde el dia de incio es el valor v_i y el dia de finalizacion del curso es v_f, la solucion se obtiene encontrando un conjunto maximo de elementos cuyos intervalos no se solapen, esto significa:

$$\forall v^1, v^2 \in C, v^1 \neq v^2 \mid (v^1_f < v^1_i) \vee (v^2_f < v^1_i)$$

esta es condición suficiente ya que por O1 el valor de incio de un curso es menor al valor de fin del mismo curso para todos los cursos.

3.3. Demostración de correctitud

Veamos que, efectivamente, nuestro algoritmo encuentra una solución óptima S cuya cantidad de intervalos es k. Para ello, vamos a suponer que tenemos la salida ordenada por la fecha de fin de cada intervalo. Supongamos que existe una solución óptima S' cuya cantidad de intervalos es n, dicha solución se encuentra ordenada del mismo modo que S.
Sean S_1 y S'_1 los primeros intervalos de ambas soluciones. En el caso en el que $f_{S_1} \leq f_{S'_1}$, se puede reemplazar S'_1 por S_1 pues, si S'_2 no se solapa con S'_1, tampoco lo hará con S_1. El caso $f_{S'_1} < f_{S_1}$ no puede ocurrir pues nuestro algoritmo selecciona el intervalo cuya fecha de fin es la menor.
Por otra parte, si consideramos S_2 y S'_2 y por el mismo criterio utilizado anteriormente, si $f_{S_2} \leq f_{S'_2}$, se puede reemplazar S'_2 por S_2 pues, si S'_1 no se solapa con S'_2, tampoco se solapará con S_2. Este mismo razonamiento puede extenderse, con el mismo principio, hasta S_k.
Tal como mencionado, los intervalos de nuestra solución terminan igual o antes que los de la solución óptima, en particular, el último de ellos. Veamos ahora, qué relación existe entre k y n. Para ello, analicemos las diferentes posibilidades:

- $k = n$**:** En este caso, la solución proporcionada por nuestro algoritmo tiene la misma cantidad de intervalos que la óptima. Luego, es óptimo.
- $k > n$**:** Esta situación no es posible dado que n es la cantidad de intervalos de la solución óptima por lo que $k \leq n$.
- $k < n$**:** En este caso, existe un intervalo en S'_{k+1} cuya fecha de fin es mayor a la fecha de fin de S_k dado que éste reemplazó a S'_k. Pero esto resulta absurdo pues este intervalo debería estar también en S como lo estipula nuestro algoritmo.

Luego, podemos concluir que $k = n$, siendo, la solución de nuestro algoritmo, la óptima al problema.

3.4. Complejidad del algoritmo

Sea n la cantidad de cursos. La complejidad de nuestro algoritmo es $\mathcal{O}(n\ log\ n)$, antes de ver el porque de esto tengamos en cuenta algunos aspectos:

1. La complejidad del algoritmo **Sort** es $\mathcal{O}(n\ log\ n)$[2].
2. La complejidad del **constructor** que utilizamos sobre la estructura **vector** es $\mathcal{O}(1)$[3].
3. La complejidad del algoritmo **push_back** sobre la estructura **vector** es O(1) amortizado pero cuando se ultiliza previamente la funcion $reserve(n)$ agregar los primeros n elementos es O(1)[4].
4. La complejidad del algoritmo **size** sobre la estructura **vector** es O(1)[5].
5. La complejidad de la funcion **filtrarSolapamientos** recorre todos los cursos y en cada paso pregunta si se solapan, esto es constante ya que ver si se solapan se logra con comparaciones de enteros, y luego de ser asi, cambia una variable en un vector ya definido y se asigna una variable (todo esto en tiempo constante). Recorrer el for tiene un costo de complejidad lineal en el peor caso y no crece ya que lo que se hace adentro es constante.

[2] http://en.cppreference.com/w/cpp/algorithm/sort
[3] http://en.cppreference.com/w/cpp/container/vector/vector
[4] http://en.cppreference.com/w/cpp/container/vector/push_back
[5] http://en.cppreference.com/w/cpp/container/vector/size

3.5. Código fuente

3.6. Instancias posibles

Para verificar la correctitud de nuestro programa, dispusimos variar estratégicamente las instancias de entrada al ejecutarlo.

- En primer lugar, ejecutamos el programa ingresando cursos que no se solapan entre sí. De este modo, se logra verificar que los cursos de la entrada coincidan con los de la salida.
  Parámetro de entrada:

  3 0 1 3 5 6 9

  Parámetro de salida:

  3 1 2 3

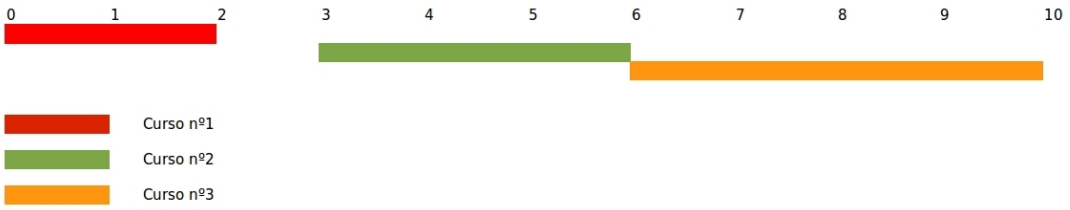

Figura 6: Instancia posible 1.

- Por otra parte, probamos el programa para la situación en la que no se ingresa ningún curso. Esto sería, por lo tanto, el caso vacío. Dado que este caso borde es muy significativo, nos pareció interesante mencionarlo.
  Parámetro de entrada:

  0

  Parámetro de salida:

  0

Figura 7: Instancia posible 2.

- Otro caso a tener en cuenta, es en el que todos los cursos finalizan el mismo día. Esto se debe a que nuestro algoritmo ordena por fecha de finalización, entonces cualquier intervalo podría llegar a ser solución y sirve para darse cuenta de cúal es el que selecciona nuestro algoritmo.
  Parámetro de entrada:

  3 1 5 3 5 2 5

  Parámetro de salida:

  1 1

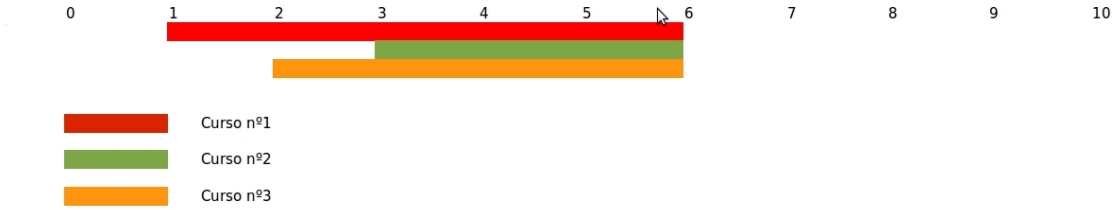

Figura 8: Instancia posible 3.

- Por último, consideramos interesante considerar el caso en el que existe al menos un curso que se solapa. Dicho caso sería el más común en el problema a resolver. En este mismo caso, decidimos que podía ser interesante considerar el caso de cursos cuya fecha de inicio fuera igual a la fecha de fin para comprobar que las cotas del programa estuvieran definidas correctamente.
  Parámetro de entrada:

  4 0 3 3 5 7 10 8 9

  Parámetro de salida:

  2 1 4

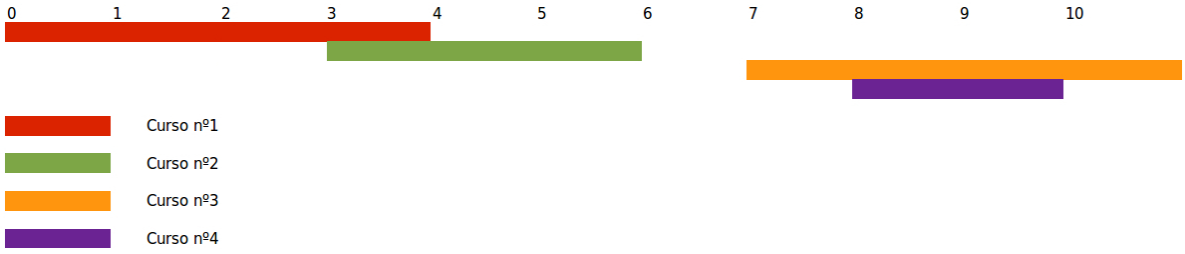

Figura 9: Instancia posible 4.

3.7. Testing

4. Ejercicio 3

4.1. Problema a resolver

El siguiente ejercicio se da en el contexto de un museo donde se requiere, por cuestiones de seguridad, colocar sensores de forma tal que todo el piso del museo esté cubierto por los láser que emiten, existen dos tipos de sensores, lo direccionales (que emiten señales horizontales o verticales) y los bidireccionales (que emiten señales hacia ambos lados) y su valor es $4000 y $6000 respectivamente. Se pide también que un sensor no este apuntando hacia otro por que esto podría provocar que algún sensor deje de funcionar y no es lo deseado, además se pide que en ciertos lugares, definidos como ïmportantes", haya dos láser pasando simultáneamente. El objetivo entonces es encontrar la forma de colocar estos sensores de forma tal que el suelo quede completamente cubierto y que los lugares importantes queden con ambos láser pasando por el y además que el costo total de los sensores sea mínimo, también hay que tener en cuenta que en el museo puede haber paredes que interfieran en los láser de los sensores.

Un ejemplo de este problema es el que está provisto por la cátedra

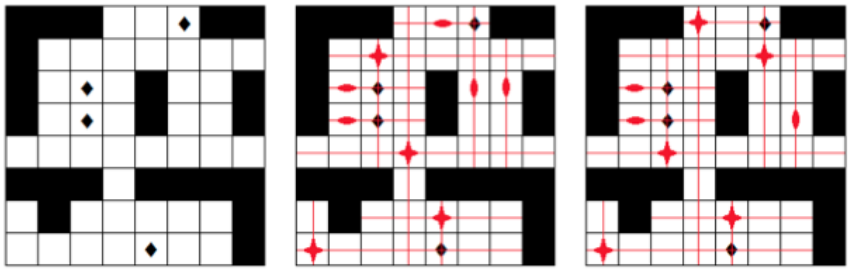

Figura 10: Un ejemplo y dos soluciones distintas.

En el ejemplo de la Figura 1 el museo se ve representado por una cuadricula donde donde los cuadrados blancos representan el suelo, los cuadrados negros representan las paredes y los cuadrados que tienen un rombo dentro representan los lugares importantes, a continuación de la imagen se muestran dos posibles soluciones al problema, en las cuales las cruces rojas representan los sensores bidireccionales, y las lineas rojas gruesas representan los sensores horizontales y verticales, desde estos sensores se extienden lineas rojas que representan el área que cubren estos sensores.

En el primer caso el costo total es de $44000 y la segunda solución tiene un costo de $42000

4.2. Resolución coloquial

Para resolver este ejercicio usamos, como recomendación de la cátedra en el enunciado, la técnica de backtracking que consiste en probar todas las posibles combinaciones de soluciones dentro del problema hasta llegar a las validas y guardar así la mejor de ellas usando en casa paso, criterios de parada que nos permitan deducir si las soluciones parcialmente obtenidas son candidatas a mejores soluciones o si son soluciones validas.

El modelo de nuestro problema mapea las el piso, las paredes y los lugares importantes del museo en casillas dentro de una matriz.

Para resolver el problema recorremos la matriz secuencialmente y cada paso tomamos una decisión sobre el casillero actual, siento estas colocar un sensor o dejar el casillero sin ningún sensor, esta genera un árbol de decisiones en el cual cada una de sus ramas representan un escenario diferente en la distribución de sensores en la matriz que representa nuestra modelo, de esta manera nos garantizamos obtener todas las posibles combinaciones de configuraciones de la matriz.

4.3. Demostración de correctitud

Sea C el conjunto de configuraciones posibles de la matriz, si en cada paso puedo probar 4 configuraciones diferentes para cada casillero disponible, y si mi matriz mide $n * m$, entonces por combinatoria tengo 4^{n*m} configuraciones finales posibles de la matriz, como nuestro algoritmo hace esencialmente eso (excepto por las podas y criterios de terminación que no representan soluciones validas) podemos decir que este encuentra todas las posibles configuraciones de la matriz.

Sea s una solución óptima al problema, ya que s representa una configuración particular de la matriz entonces como C representa el conjunto de todas las posibles soluciones del problema y la solución óptima, si es que existe alguna solución, está dentro de ese conjunto, entonces nuestro algoritmo debería encontrarla y devolverla.

Es correcto por que usa backtracking.-

4.4. Complejidad del algoritmo

Pseudocódigo:

Algorithm 4: Algoritmo de Backtracking

```
Input: Matriz grilla
Output: lista sensores
Lista casillasLibres ← ObtenerPosicionesLibres(grilla)
for Posicion p ∈ grilla do
    if p es importante then
        RestringirPorImportantes(p)
    end
end
sensores := backtrack(grilla, casillasLibres)
devolver sensores
```

Algorithm 5: backtrack

```
Input: Matriz grilla, Lista casillasLibres
Output:
Lista casillaActual := Proxima casilla libre en casillasLibres
if Puedo poner un sensor bidireccional en casillaActual then
    Restringir por láser bidireccional
    Saco casillaActual de casillasLibres
    backtrack(grilla, casillasLibres)
end
if Puedo poner un sensor vertical en casillaActual then
    Restringir por láser vertical
    Restringir casilleros horizontales
    Saco casillaActual de casillasLibres
    backtrack(grilla, casillasLibres)
end
if Puedo poner un sensor horizontal en casillaActual then
    Restringir por láser horizontal
    Restringir casilleros verticales
    Saco casillaActual de casillasLibres
    backtrack(grilla, casillasLibres)
end
Dejo el casillero en blanco
backtrack(grilla, casillasLibres)
devolver ContarSensores(grilla)
```

4.5. Código fuente

4.6. Instancias posibles

Para verificar la correctitud de nuestro programa, dispusimos variar estratégicamente las instancias de entrada al ejecutarlo.

- En primer lugar, ejecutamos el programa ingresando un único espacio libre el cual contenía un lugar importante. La idea de esto es comprobar que nuestro algoritmo devuelve -1 en el caso de que no haya solución ya que el espacio especial no podia ser cubierto por 2 láser's.

  Parámetro de entrada:

  3 3
  0 0 0
  0 2 0
  0 0 0

  Parámetro de salida:

  -1

- Por otra parte, probamos el programa cuando todo el museo es una pared y no contiene ni espacios libres ni espacios importantes, en otras palabras, el caso vacío. Lo interesante de esto es ver que nuestro algoritmo funciona en dicho caso borde y que devuelve que el problema tiene solución y que la solución es no colocar ningún láser.

  Parámetro de entrada:

  3 3
  0 0 0
  0 0 0
  0 0 0

  Parámetro de salida:

  0 0

- Un caso interesante, es ver cuando todos los espacios están libres. La idea de esto es comprobar que el algoritmo nos da la solución mas barata al problema

  Parámetro de entrada:

  3 3
  1 1 1
  1 1 1
  1 1 1

  Parámetro de salida:

  3 12000
  2 0 0
  2 1 2
  2 2 0

- Otro caso interesante, es ver cuando todos los espacios son importantes. Este es un caso particular en el cual no existe solución ya que no podrían pasar dos láser's por el mismo punto especial sin que uno de estos toque a otro láser

  Parámetro de entrada:

  3 3
  2 2 2
  2 2 2
  2 2 2

  Parámetro de salida:

  -1

- Otro caso interesante, es ir alternando entre piso y pared. Lo interesante de este caso es que no se puede aplicar ninguna poda.
  Parámetro de entrada:

  3 3
  1 0 1
  0 1 0
  1 0 1

  Parámetro de salida:

  5 20000
  2 0 0
  2 0 2
  2 1 1
  2 2 0
  2 2 2

- Por ultimo, esta el caso en el que tenemos los 3 tipos de casilleros, este seria el caso mas común del problema.

  Parámetro de entrada:

  3 3
  1 1 1
  1 2 0
  0 0 1

  Parámetro de salida:

  3 14000
  1 0 1
  2 1 0
  2 2 2

4.7. Testing

5. Referencias

- http://dacap.com.ar/blog/cpp/cola-de-prioridades-stl-priority-queue/
- https://www.lsi.upc.edu/ ps/downloads/matdoc/transparencias/ps/ps$_V I_h eaps.pdf$