DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Informe y análisis de resultados.

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Benitti, Raul	592/08	raulbenitti@gmail.com
Mengarda,Lucas	787/10	l.j.mengarda@gmail.com
Scarpino, Gino	392/08	gino.scarpino@gmail.com
Vallejo, Nicolás	500/10	nico_pr08@hotmail.com

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

Índice

1. Problema 1

1.1. Introducción

El problema que se plantea es el siguiente: En una fábrica de quesos hay n máquinas que trabajan en forma independiente, cada una produciendo un determinado tipo de queso. Una vez encendida, la máquina i tarda p_i minutos en finalizar su producción. Por cuestiones de seguridad, las máquinas comienzan el día con sus tanques de combustible totalmente vacíos, y luego de que se llene su tanque, tarea en la que se demora c_i minutos, la máquina i comienda a producir sin ningun otro tipo de intervención. Como se cuenta con un único surtidor para todas las máquinas, la carga de los tanques debe realizarse de manera secuencial. Sin embargo, el tiempo que se tarda en mover el surtidor de una máquina a otra es despreciable.

Lo que se desea es encontrar un orden para la carga de combustible de manera tal que la producción completa de todas las máquinas se termine lo antes posible. El algoritmo que resuelva el problema debe tener una complejidad menor o igual a $O(n^2)$

P_i	10	4	6
c_i	1	2	4
i	1	2	3

Figura 1: Ejemplo de una instancia del problema. Se tiene tres máquinas, con sus respectivos tiempos de produccion y carga.

1.2. Desarrollo

Sea O un orden en que las máquinas son cargadas. Diremos que una máquina i demora en terminar lo que se tarda en esperar a que sea su turno de ser cargada, más su tiempo carga y de producción. El tiempo de espera equivale a la suma de los tiempos de cargas de las máquinas que son tratadas antes que ella. Entonces, definimos $T_O(i)$, el tiempo total que demora en terminar la i-esima máquina dentro del orden O, como

$$T_O(i) = \sum_{j=1}^{i} c_{O[j]} + P_{O[i]}$$

El tiempo que demora en terminar la producción completa es equivalente al tiempo que demora en finalizar su producción la ultima máquina. Definimos, entonces, T(O) como

$$T(O) = \max_{1 \leq i \leq n} T_O(i)$$

Buscamos encontrar un orden O tal que minimice T(O).

P_i	10	4	6
c_i	1	2	4
i	1	2	3
O[i]	1	2	3
$T_O(i)$	11	7	13

(a)

P_i	10	4	6
c_i	1	2	4
i	1	2	3
O[i]	1	3	2
$T_O(i)$	11	11	11

(b)

Figura 2: En estas figuras se pueden observar dos posibles órdenes de carga de máquinas. En **(a)** se sigue el orden O según la numeracion de las máquinas, con un resultado T(O) = 13. En cambio, en **(b)** se muestra un orden O óptimo, con resultado T(O) = 11.

La primera idea intuitiva que surge es tratar de tener la mayor cantidad de máquinas funcionando en simultáneo. Así, al momento de seleccionar cual es la siguiente máquina que debe ser cargada, tomamos aquella que tiene mayor tiempo de producción de manera que las máquinas restantes lleven a cabo sus trabajos mientras esta se encuentra produciendo.

Otra forma de verlo es la siguiente: como se quiere que todas las máquinas terminen lo antes posible (de manera que la ultima en hacerlo lo haga lo antes posible), y considerando que, para comenzar su producción, la i-esima máquina debe esperar su tiempo de carga c_i mas la suma de los tiempos de cargas de las máquinas anteriores, lo que buscamos es que la máquina con mayor tiempo de producción comience lo antes posible (que tenga el menor tiempo de espera posible); es decir, que sea la primera en cargarse dentro del conjunto de máquinas disponibles. Esta idea se ajusta claramente a la forma de un algoritmo goloso:

- decisión golosa: seleccionar y cargar la máquina con mayor P_i
- ordenar las máquinas todavia no cargadas de forma tal que la producción completa de estas termine lo antes posible

A partir de aquí se puede plantear la solución como una funcion recursiva

$$Orden(maquinas) = [seleccionGolosa(maquinas)] + Orden(maquinas - \{i\})$$

donde *Orden* devuelve un lista, *maquinas* es un conjunto de máquinas y *seleccionGolosa* devuelve la máquina cuyo tiempo de producción es el mayor que entre los las máquinas del conjunto. En la sección de correctitud veremos que no es necesario considerar los tiempos de carga al tomar la decisión golosa.

Ahora, el resultado de esta idea equivale simplemente a ordenar las máquinas de manera decreciente segun sus tiempos de producción. Por esto, en lugar de desarrollar un algoritmo en particular, utilizaremos alguno de los algoritmos de sorting ya conocidos (en este caso, dado la complejidad requerida, cualquiera de ellos es válido).

1.3. Algoritmo

El algoritmo *resolver* toma como argumentos de entrada los tiempos de carga (c) y procesamiento (P), y como argumentos de salida, el orden (O, un vector de n elementos, donde O[i] = i) y el tiempo total (T), y utiliza la implementación de sorting de la plataforma.

El algoritmo *calcTiempo*, que calcula el tiempo $T(O)$, solo recorre el orden O calculando los tiempos $T_O(i)$, y devuelve el máximo de todos ellos.

Algoritmo 1.1

RESOLVER(P, c, O, T)

1 Ordenar O segun el ordenamiento descendiente de los elementos en, de manera que P[O[i]] $\leq$ P[O[i+1]] para i = 1..n-1
2 $T =$ CALCTIEMPO(P, c, O)

Algoritmo 1.2

CALCTIEMPO(P, c, O)

```
1  resultado = 0
2  inicio = 0
3  for i in O
4        t_i = inicio + c[i] + p[i]
5        if (resultado < t_i)
6              resultado = t_i
7        inicio = inicio + c[i]
8  return resultado
```

1.3.1. Correctitud

Como mencionamos anteriormente, el algoritmo se basa en una idea golosa. Para demostrar la correctitud de ésta, observemos los siguientes puntos:

- EL problema presenta subestructura óptima:
  Como ya vimos, el problema de encontrar el orden óptimo puede plantearse como uno en el que se realiza una elección, y se resuelve un subproblema del

mismo tipo pero de tamano menor. Ahora, supongamos que el orden O es una solución óptima al problema original; es decir, el valor de $T(O)$ es el menor posible. Llamemos S = O[2..n] a la solución del subproblema que queda al elegir el primer elemento de O, de forma que $O = O[1] + S$. Si S no es un solución óptima para ese subproblema, podriamos encontrar algun otro orden S' que si lo fuera, tal que se cumpla $T(S') < T(S)$. Consiremeos entonces una nueva solución $O' = O[1] + S'$. Dado el caso en que $T_O(1) \leq T(S')$, al calcular $T(O')$ tendriamos

$$T(O) = max(O[1], T(S)) < max(O[1], T(S')) = T(O')$$

Sin embargo, esto no puede suceder pues supusimos O un orden óptimo. Por lo tanto, la subsolución S debe ser, tambien ella, óptima.

- Existe una solucion óptima que tiene, como primera elección, la elección golosa que propone el algoritmo:
  Supongamos que O es óptimo, y sea g la máquina con mayor tiempo de producción.
  Si $O[1] = g$, entonces se cumple lo que buscabamos.
  Si $O[1] \neq g$, entonces debe existir algun otro j, $1 < j \leq n$ tal que $O[j] = g$. Supongamos que modificamos O intercambiando su primer elemento con el elemento j-esimo, obteniendo asi un nuevo orden O' que cumple

  - $O'[1] = O[j] = g$,
  - $O'[j] = O[1]$ y
  - $O'[k] = O[k]$ para todo k distinto de 1 y j.

  Queremos ver que O' sigue siendo óptimo, o sea, que $T(O') \leq T(O)$. Notemos, en primer lugar, que para todo k tal que $j < k \leq n$ se cumple $T_{O'}(k) = T_O(k)$, pues, considerando que los primeros k elementos de O' son solo una permutacion de los primeros k elementos de O,

$$T_{O'}(k) = \sum_{i=1}^{k} c_{O'[i]} + P_{O'[k]} = \sum_{i=1}^{k} c_{O[i]} + P_{O[k]} = T_O(k)$$

  Por otro lado, observemos que en O[1..j] tenemos que $T_{O'}(j) = \text{máx}_{1 \leq i \leq j} T_O(i)$ ya que teniendo en cuenta que los c_i son naturales, que $k < j$ y que $P_{O[k]} \leq P_{O[j]} = P_g$,

$$T_O(k) = \sum_{i=1}^{k} c_{O[i]} + P_{O[k]} \leq \sum_{i=1}^{k} c_{O[i]} + \sum_{i=k}^{j} c_{O[i]} + P_{O[j]} = T_O(j)$$

  Entonces, lo que necesitamos ver es que tambien suceda $T_O(j) \geq \underset{1 \leq i \leq j}{\text{máx}} T_{O'}(i)$.
  Para las máquinas intercambiadas tenemos

$$T_{O'}(1) = \sum_{i=1}^{1} c_{O'[i]} + P_{O'[1]} \leq \sum_{i=1}^{j} c_{O'[i]} + P_{O'[1]} = \sum_{i=1}^{j} c_{O[i]} + P_{O[j]} = T_O(j)$$

$$T_{O'}(j) = \sum_{i=1}^{j} c_{O'[i]} + P_{O'[j]} \leq \sum_{i=1}^{j} c_{O'[i]} + P_{O'[1]} = \sum_{i=1}^{j} c_{O[i]} + P_{O[j]} = T_O(j)$$

Para las demas máquinas (k en el rango 2..j-1),

$$T_{O'}(k) = \sum_{i=1}^{k} c_{O'[i]} + P_{O'[k]} \leq \sum_{i=1}^{k} c_{O'[i]} + \sum_{i=k}^{j} c_{O'[i]} + P_{O'[k]} \leq \sum_{i=1}^{j} c_{O[i]} + P_{O[j]} = T_O(j)$$

Notemos que no hara falta considerar el orden segun los tiempo de carga: supongamos que tenemos una seguidilla $P_k...P_l$ donde $P_i = p$ para todo $k \leq i \leq l$. Sea $t = \sum_{i=1}^{k-1} c_i$

El tiempo que tarda en terminar la tarea P_l es $t + \sum_{i=k}^{l}(c_i) + p$.

El tiempo que tarda en terminar una cualquiera de las otras tareas en el grupo, sea P_j que cumple $i \leq j < l$, es $t + \sum_{i=k}^{j}(c_i) + p$.

De esto se tiene que

$$t + \sum_{i=k}^{j}(c_i) + p < t + \sum_{i=k}^{j}(c_i) + \sum_{i=j}^{l}(c_i) + p = t + \sum_{i=k}^{l}(c_i) + p$$

Es decir, no importa como se permuten los valores c_i, un grupo de máquinas con igual tiempo de producción siempre tardará lo mismo en terminar, y este valor será la suma de todos los tiempos de carga del grupo (y las máquinas anteriores, si es que hay) más el tiempo de producción p.

1.3.2. Análisis de complejidad

Sea n la cantidad de máquinas.

El algoritmo de sorting que se utiliza esta implementado en la STL de C++. Se asegura que el mismo tiene complejidad $O(nlog(n))$ [1]. El algoritmo *calcTiempo* solo tiene un loop que itera n veces, dentro del cual todas las operaciones son de orden O(1), lo que resultan en una complejidad final del orden de O(n).

Por lo tanto, el algoritmo *resolver* tiene complejidad $O(nlog(n)+n) = O(nlog(n))$, con lo que se cumple el requerimiento de encontrar una solucion en tiempo menor a $O(n^2)$

1.4. Pruebas

No distinguimos casos particulares que sean de interés específico en analizar, pues nuestra implementación depende de la implementación del algoritmo de sort de la STD, y se asegura que este no presenta casos patológicos. Por otra parte, a

[1] http://www.sgi.com/tech/stl/sort.html

priori podría parecer útil analizar casos como aquellos en los que se encuentran uno o más bloques de maquinas con los mismos tiempos de produccion, o probar distintas relaciones entre los tiempos de carga y producción. Sin embargo, estos detalles no son tenidos en cuenta en el algoritmo en sí, por lo que utilizarlos para separar familias de pruebas resulta superfluo. Dicho esto, presentamos un gráfico donde se muestra como en la práctica el algoritmo se ajusta a la la complejidad predicha.

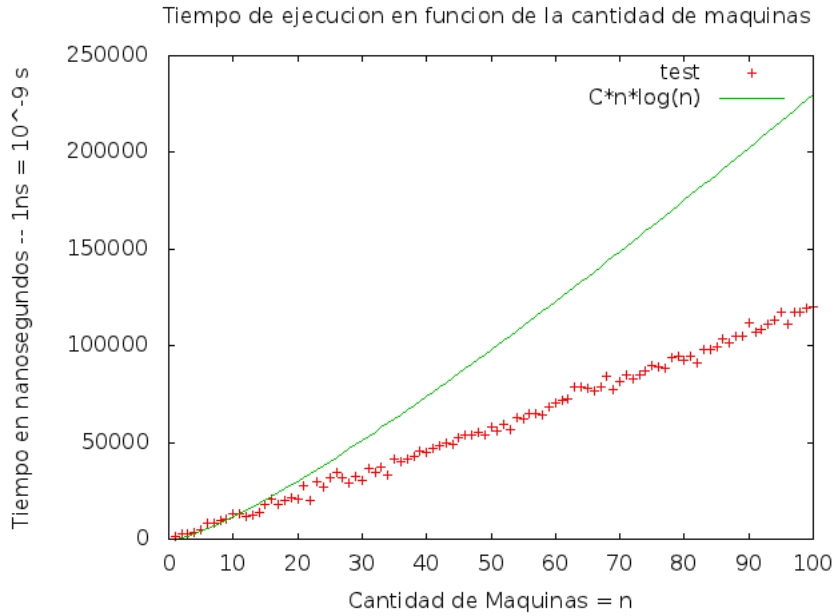

Figura 3

1.5. Conclusiones

En este ejercicio observamos como un problema, luego de ser analizado con un poco de detenimiento, puede convertirse en una instancia de otro problema para el cual pueden ya existir algoritmos satisfactorios. En ese caso, todo el proceso psoterior de implementación y análisis se simplifica.

2. Problema 2: Sensores defectuosos

2.1. Introducción

Se tiene un conjunto de sensores con sus respectivos intervalos de tiempo de medición y se conoce el número de la medición que falla. Se desea conocer el id del sensor que falló.

2.2. Desarrollo

Encaramos este problema rearmando la secuencia de mediciones de los sensores. Consiguiendo ésto, la solución al problema sería fijarse en esta secuencia, el id que figura en la posición de medición que falló.

Para armar la secuencia de mediciones, decidimos ir guardando para cada sensor en que instante de tiempo tendría que volver a realizar una medición. A medida que determinamos el sensor que va a medir, calculamos su siguiente tiempo de medición.

Teniendo un diccionario donde para cada sensor se puede obtener en qué momento le toca realizar una medición, se puede conocer al próximo que le toca medir, ya que es el mínimo de los tiempos.

En el momento inicial todos los sensores miden, por lo que la secuencia inicial no es vacía, sino que contiene todos los números de sensores ordenados por su id ya que desempatan por éste.

En cada paso de nuestro algoritmo, agregamos a la secuencia el siguiente sensor que midió y calculamos su próxima medición actualizando la tabla de próximas mediciones

Cuando la secuencia alcanza el tamaño igual al número de medición que falló, terminamos y el resultado es la última posición de esa secuencia.

Aclaración: en la implementación del diccionario se usa directamente un minHeap. Más adelante se explica el motivo de tal decisión.

Pseudocódigo:

Algoritmo 2.1 Entero sensorDefectuoso(Conjunto sensores, Entero medDefectuosa)

```
1: Diccionario diccTiempos = sensores
2: Entero medicion = | sensores |
3: for all Sensor s in sensores do
4:    agregar(mediciones,id(s))
5: end for
6: while medicion ¡medDefectuosa do
7:    proximo = proxSensor(diccTiempos,sensores)
8:    agregar(mediciones,proximo)
9:    medicion = medicion + 1
10: end while
```

Donde

medDefectuosa es el número de medición que falla.

diccTiempos es el diccionario que para cada sensor guarda el tiempo de su próxima medición.

sensores es el conjunto de sensores.

medicion es el número de medición parcial. Como al inicio hubo la cantidad de sensores, se inicializa con esa cantidad.

mediciones es la secuencia de mediciones con los ids de los sensores.

proxSensor es la funcion que devuelve el id del mínimo de los tiempos que están presentes en el diccionario, y además actualiza el tiempo para la próxima medición.

2.2.1. Correctitud

En alguna instancia del problema, si tuviéramos la siguiente tabla con los momentos t_i para la siguiente medición de cada sensor s_i con $(1 \leq i \leq n)$ donde n es la cantidad total de sensores:

s_1	s_2	...	s_{n-1}	s_n
t_1	t_2	...	t_{n-1}	t_n

determinar el siguiente sensor que le toca realizar su medición, sería elegir el de menor t. Caso contrario, se estaría eligiendo un sensor con tiempo mayor, por lo que haría una medición anterior cuando en realidad le correspondería a un sensor con menos tiempo, generando una incongruencia en el orden de mediciones.

Por lo que alcanzaría con ver que nuestro algoritmo genera esa tabla/diccionario de forma correcta.

En el instante inicial, el diccionario contendrá los tiempos de intervalos de medición ya que todos midieron de entrada. Para el primero, elegimos el mínimo de esos tiempos. Una vez que sabemos el id, lo agregamos a la secuencia de mediciones, y redefinimos en el diccionario su tiempo de la siguiente forma:

$$t'_i = t_i + intervalo(s_1)$$

es decir, que al tiempo que ya figuraba en el diccionario, se le agrega el tiempo de intervalo para determinar el tiempo de su próxima medición. Por lo que, en cada

paso donde se elige el sensor que mide, se actualiza el diccionario. De esta forma, se mantiene la correctitud del significado del diccionario una vez elegido el próximo sensor a medir.

Se repite este proceso hasta llegar a la medición que falla inclusive, con lo que el id del sensor quedaría guardado en la secuencia de mediciones.

2.2.2. Análisis de complejidad

Se cargan los datos de los sensores en vectores. En uno, el tiempo de intervalo entre mediciones. En otro, una tupla que representaría la primer componente el id del sensor y la segunda componente el tiempo de próxima medición. A la vez que se cargan los datos, ya se colocan los ids por orden en el vector de mediciones ya que todos miden en el instante inicial.

La complejidad de lo anterior es O(n), donde n es la cantidad de sensores.

En el caso que el número de medición sea menor a la cantidad de sensores, el algoritmo termina devolviendo el valor del vector de mediciones según la posición correspondiente a la medición que falla.

Caso contrario, calculamos la secuencia de mediciones. Para ello necesitamos el ya mencionado diccionario. Decidimos utilizar un min-Heap como diccionario. Esto nos permitiría obtener el mínimo valor de tiempo de próxima medición en O(log n) siendo n la cantidad de sensores. Crear el minHeap tiene un costo de $O(3 * n)$.

Una vez creado el minHeap, comenzamos un ciclo donde se busca el siguiente sensor que mide. Como ya dijimos, al usar un minHeap, ésto tiene un costo de O(log n). Luego, se agrega el id del sensor a la secuencia de mediciones con un costo constante.

La cantidad de iteraciones del ciclo es el número de medición que falla menos la cantidad de sensores existentes. Ésto se debe a que, en el momento inicial, todos los sensores realizan una medición. Por lo que de entrada se tienen n mediciones.

Por lo tanto, la complejidad total del algoritmo termina siendo:

$$\sum_{i=k}^{n} logn = (k - n) * logn \leq k * n$$

Termina siendo estrictamente menor a $O(k * n)$ como se pedía en la consigna.

2.3. Resultados

Para analizar el comportamiento del nuestro algoritmo, decidimos generar un test aleatorio donde se va variando la cantidad de sensores. En el mismo nos queda una grfica:

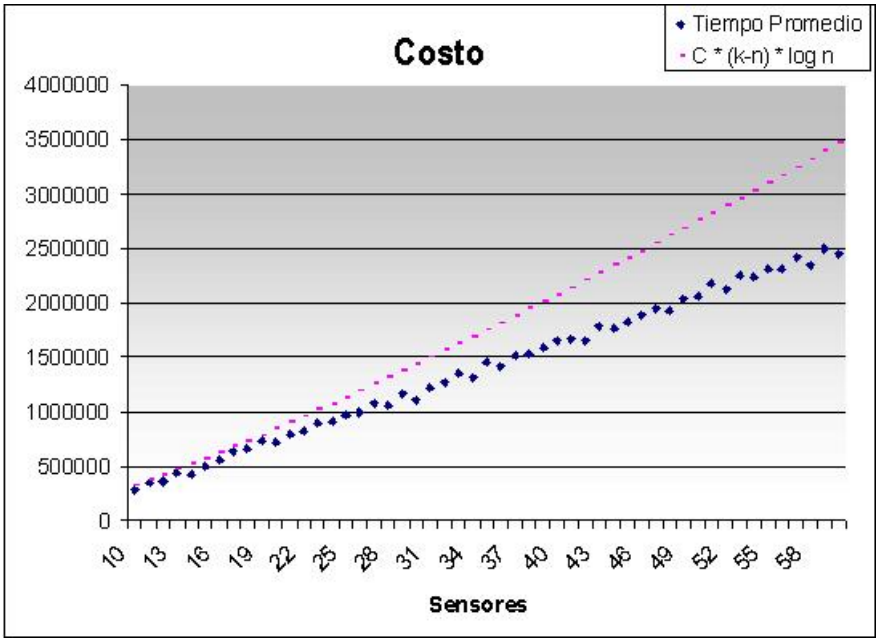

Figura 4: Se compara el tiempo que tardó en promedio en obtener un resultado contra el tiempo teórico del peor caso

Armamos el test de la siguiente manera. Vamos a hacer instancias que vayan de 10 sensores a 100 y por cada cantidad de sensores realizamos 1000 repeticiones. Decidimos que la falla se produzca en la medición: $20 * n + r$. Donde n es la cantidad de sensores y r un valor aleatorio entre 1 y n.

Para cada ejecución guardamos el tiempo que tardó usando la función *clock-gettime*

2.4. Conclusiones

Como era de esperar, los resultados mostraron que el tiempo de ejecución es asintóticamente menor al tiempo teórico calculado para el peor caso. Siendo ese tiempo, menor al pedido por la consigna del trabajo práctico.

Analizar bien las posibles estructuras de datos a utilizar fue fundamental para volver más eficiente el algoritmo. El problema de obtener en cada instante el mínimo de un conjunto de valores utilizando un minHeap logró bajar la complejidad lineal a logarítmica.

Luego, también era necesario considerar la parte del problema en que todos los sensores realizan una medición en el momento inicial. Lo que permite realizar n iteraciones menos de nuestro algoritmo, donde n es la cantidad de sensores.

3. Problema 3: La caja en el plano

3.1. Introducción

En este caso, el problema que se nos presentó para resolver fue el siguiente:

Se tienen n puntos en el plano con coordenadas enteras y se tiene además una caja, representada por un rectángulo de dimensiones dada. La caja puede ubicarse en cualquier lugar del plano, pero no puede rotarse, es decir, la base de la caja debe quedar paralela al eje x y la altura de la caja debe quedar paralela al eje y. Un punto sobre un borde de la caja se considera dentro de la misma. Se desea ubicar la caja de manera tal que la cantidad de puntos que queden dentro de la caja sea máxima. El algoritmo que resuelva este problema debe hacerlo con una complejidad de a lo sumo $\mathbf{O}(n^3)$.

Es oportuno realizar alguna aclaraciones. En primer lugar, que el problema siempre tiene solución óptima, es decir que siempre habrá una ubicación de la caja que cubra mayor o igual cantidad de puntos que todas las demás. En segundo lugar, que la solución óptima no siempre es única.

A continuación se provee un ejemplo de una posible instancia del problema con una posible solución y que además sirve para mostrar las aclaraciones hechas anteriormente:

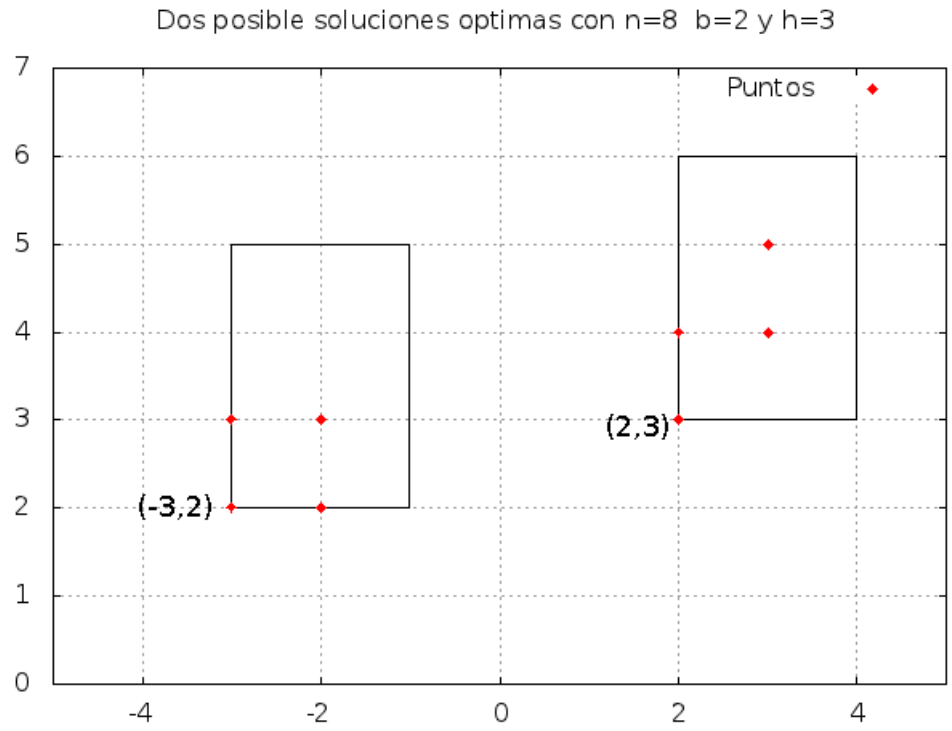

Figura 5: Aquí se observa como existe más de una solución óptima para el problema, puesto que tanto $(-3, 2)$ como $(2, 3)$ cubren la máxima cantidad de puntos posibles, en este caso cuatro.

Para resolver este problema hemos asumido que por lo menos se tiene algún punto en el plano (es decir que $n > 0$) y que además los puntos que se proveen por entrada no están repetidos.

3.2. Desarrollo

Dado el problema a resolver, la primera noción que se extrajo al momento de analizarlo fue el hecho de que siempre existe una solución óptima, aunque no siempre es única.

Una primera aproximación intuitiva para la resolución consistió en ir posicionando la esquina inferior izquierda de la caja en todos los lugares posibles de un área acotada por cuatro puntos, cada uno el más extremo hacia la izquierda, la derecha, arriba o abajo. Si bien parece lógico encontrar la solución deseada de esta manera, el algoritmo sería muy complejo (y en particular la complejidad estaría condicionada por la lejanía entre los puntos extremos), por lo que resultaba necesario acotar de alguna manera las posibles posiciones donde se evaluaría la esquina inferior izquierda de la caja.

Una segunda aproximación consistió en ir probando cada punto en los cuatro vértices de la caja y evaluar cuántos de los demás puntos entrarían en ella. Sin embargo, esta idea no resultó ser fructífera: solamente estábamos evaluando potenciales soluciones que cumplían con tener por lo menos un punto en el vértice y no es cierto que siempre existe una solución óptima que tenga por lo menos un punto en el vértice.

Luego de analizar más detenidamente el contexto del problema, se comprendió que a partir de un posicionamiento de la esquina inferior izquierda de la caja con la condición de que por lo menos un punto quedara cubierto por ella, uno podría encontrar otro posicionamiento que cumpliera que por lo menos un punto cubierto por ella se encuentre en el borde inferior de la caja y otro en el borde izquierdo de la caja (puede darse el caso que sean el mismo punto, si es así el punto estaría ubicado en el vértice inferior izquierdo de la caja) y que la cantidad de puntos dentro de la caja de este posicionamiento es mayor o igual que la cantidad del primer posicionamiento.

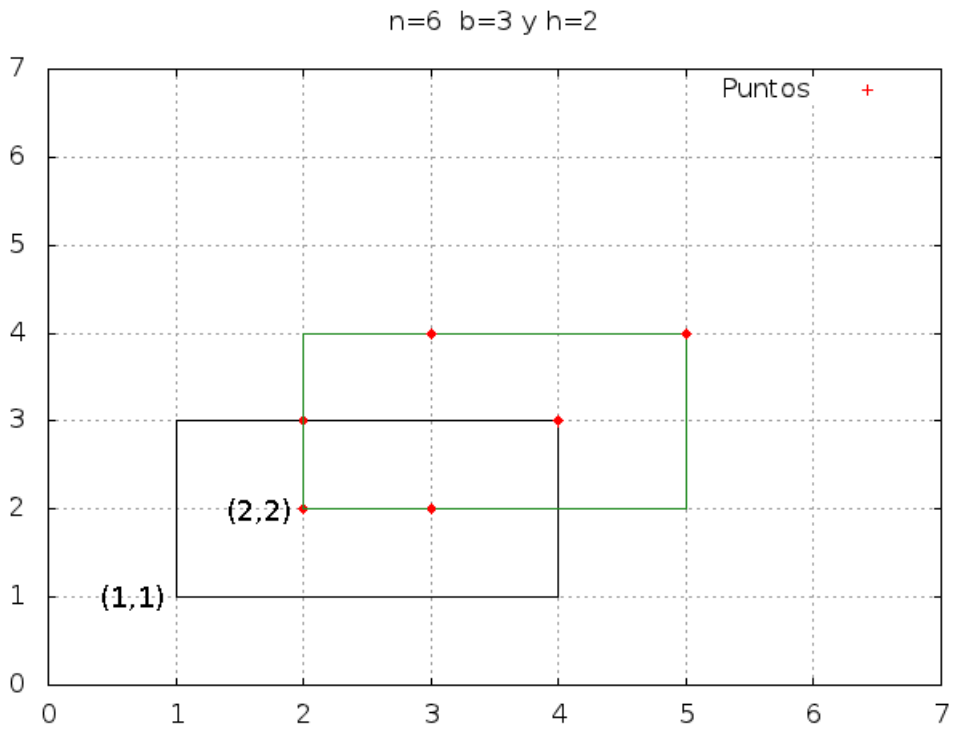

Figura 6: Aquí se observa como a partir de la posición $(1, 1)$ de una caja se puede hallar otra (la $(2, 2)$) que tenga por lo menos un punto en el borde,los mismos puntos que la $(1, 1)$ y que puede llegar a cubrir más puntos.

De aquí se extrae que siempre hay por lo menos una solución óptima que cumpla con estos criterios, puesto que la solución óptima debe cubrir por lo menos un punto. En particular, esa solución óptima cumple con que el componente x de su esquina inferior izquierda se corresponde con la coordenada x de alguno de los puntos que se tienen en el plano y que el componente y de su esquina inferior izquierda se corresponde con la coordenada y de alguno de los puntos que se tienen en el plano (y que no necesariamente son el mismo punto).Una demostración de todo esto se puede encontrar en la sección de Correctitud de este problema.

De esta manera hemos reducido el conjunto de posibles soluciones a uno más pequeño y que sabemos con seguridad que contiene una solución óptima.

Luego, las potenciales soluciones son las esquinas inferiores izquierdas que cumplen que su coordenada x se corresponde con la componente x de alguno de los puntos en el plano y que su coordenada y se corresponde con la componente y de alguno de los puntos en el plano.

Así, hay a lo sumo n^2 candidatos a solución a analizar que se obtienen de combinar la coordenada x de cada punto con las coordenadas y de todos los puntos. En particular, nuestro algoritmo recorre estos candidatos a solución y devuelve el candidato que más puntos cubra, siendo así una solución óptima.

3.3. Algoritmo

Queremos hacer un par de aclaraciones: n, b y h son de tipo *int* y representan la cantidad de puntos, la base de la caja y la altura de la caja respectivamente. A su vez, puntosX contiene en la posición i la coordenada x del i-ésimo punto de la entrada. Análogamente, puntosY contiene en la posición i la coordenada y del i-ésimo punto de la entrada. Por lo tanto, la longitud de ambos vectores es n, la cantidad de puntos en el plano.

Además queremos mencionar que, en caso de existir más de una solución óptima, la solución que devuelve el algoritmo está fuertemente influenciada por el orden en el que se pasaron los puntos por la entrada. De esta manera se pueden pasar por entrada dos instancias iguales, con la única diferencia del orden en el que se pasan los puntos, y el algoritmo devovlver dos soluciones distintas en cada instancia, siendo sin embargo las dos óptimas.

El algoritmo recorre los candidatos a solución que cumplen que su coordenada x coincide con la de algún punto del sistema y que su coordenada y coincide con la de algún punto del sistema (que puede no ser el mismo que el de la coordenada x).

Esto se basa en el hecho de que en dicho conjunto de puntos seguro existe por lo menos una solución óptima. Se provee la demostración de esta afirmación en la sección Correctitud de este problema.

Algoritmo 3.1

RESOLVER($n, b, h, vector < int > puntosX, vector < int > puntosY$)

```
 1  int cant = 0
 2  int m = 0
 3  int resx = puntosX[0]
 4  int resy = puntosY[0]
 5  for int i desde 0 hasta n
 6        for int j desde 0 hasta n
 7              m = cuantosEntran(n, b, h, puntosX[i], puntosY[j], puntosX, puntosY)
 8              if (cant < m)
 9                    cant = m
10                    resx = puntosX[i]
11                    resy = puntosY[j]
12  return cant , resx, resy
```

Algoritmo 3.2

```
CUANTOSENTRAN(n, b, h, x, y, vector < int > puntosX, vector < int > puntosY)
1  int cant = 0
2  for int i desde 0 hasta n
3        if ((x ≤ puntosX[i] ≤ x + b) AND (y ≤ puntosY[i] ≤ x + h))
4              cant ++
5  return cant
```

3.3.1. Correctitud

Sean s= (s_x,s_y) una coordenada entera que representa la esquina inferior izquierda de la caja, P el conjunto de los puntos en el plano, n la cantidad de puntos en el plano, b la base de la caja y h la altura de la caja. Y sea P_s=$\{p_1...p_k\}$ el conjunto de los puntos que quedan cubiertos por la caja cuya esquina inferior izquierda está en s.

Definimos una solución óptima para este problema como aquel s que cumple que $\#(P_s) \geq \#(P_r)$ para todo r coordenada entera.

En primer lugar, veamos que $\exists$ un t solución óptima tal que $t_x = e_x \wedge t_y = f_y$ para algunos $e, f \in P$ (no es necesario que $e = f$).

Para ello, veamos que siempre existe una solución óptima con por lo menos un punto en el borde de la caja.

Sea s una coordenada. Supongamos $P_s = \{p_1...p_k\} \neq \emptyset$, es decir que la caja representada por s cubre por lo menos un punto.
Para todo $p \in P_s$ vale que

$$s_x \leq p_x \leq s_x + b \wedge$$
$$s_y \leq p_y \leq s_y + h$$

puesto que están cubiertos por la caja.

Ahora bien, sean $a = \text{mín}_{i:1...k}((p_i)_x)$ y $c = \text{mín}_{i:1...k}((p_i)_y)$.
Definamos ahora s' tal que $s'_x = a$ $s'_y = c$.
Entonces, para todo i desde 1 hasta k:

$$s'_x = a = \text{mín}_{i:1...k}((p_i)_x) \leq (p_i)_x \wedge$$
$$s'_y = c = \text{mín}_{i:1...k}((p_i)_y) \leq (p_i)_y$$

Es decir,

$$s'_x \leq p_x \wedge$$
$$s'_y \leq p_y$$

para todo $p \in P_s$.

Además, por como definimos s':

$$s_x \leq s'_x \wedge$$
$$s_y \leq s'_y$$

Por lo tanto, para todo $p \in P_s$:

$$s_x \leq s'_x \leq p_x \leq s_x + b \leq s'_x + b \wedge$$
$$s_y \leq s'_y \leq p_y \leq s_y + h \leq s'_y + h$$

Es decir que:

$$s'_x \leq p_x \leq s'_x + b \wedge$$
$$s'_y \leq p_y \leq s'_y + h$$

y eso quiero decir que para todo punto $p \in P_s$ vale que $p \in P_{s'}$.
Luego, $P_s \subseteq P_{s'}$.
Es decir que $\#P_s \leq \#P_{s'}$.
Notar que no vale la vuelta, puesto que al reubicar la caja se pueden estar cubriendo puntos que antes no estaban cubiertos.

En particular, si s es una solución óptima, vale que $\#(P_s) \geq \#(P_r)$ para todo r coordenada entera (recordamos que en la plano hay por lo menos un punto y que necesariamente la solución óptima tiene que cubrir por lo menos un punto).
Luego, $\#P_s \geq \#P_{s'}$. Pero como $\#P_s \leq \#P_{s'}$, entonces:

$$\#P_s = \#P_{s'}$$

y entonces s' también es solución óptima. Como siempre se puede encontrar una solución óptima s, entonces también siempre se puede encontrar una solución óptima s'. Es decir que hemos encontrado una solución óptima que cumple que tiene por lo menos un punto en el borde.

Definamos ahora el conjunto de coordenadas

$$Z = \{z | z_x = e_x \text{ para algún } e \in P \wedge z_y = f_y \text{ para algún } f \in P\}$$

Es decir que Z es el conjunto de coordenadas que cumplen que su componente x se corresponde con el de algún punto en el plano y que su componente y se corresponde con la algún punto en el plano (no necesariamente el mismo punto).

Es fácil observar que de la manera en que lo definimos, $s' \in Z$.
Esto es porque $s'_x = \text{mín}_{p \in P_{s'}}(p_x) \wedge s'_y = \text{mín}_{p \in P_{s'}}(p_y)$.
Además como siempre puedo encontrar un s' con estas condiciones que sea solución óptima, eso quiere decir que seguro existe una solución óptima en Z.
Por lo tanto, $\exists$ una coordenada t solución óptima tal que $t_x = e_x \wedge t_y = f_y$ para algunos $e, f \in P$ (no es necesario que $e = f$), que es lo que queríamos demostrar.

Ahora veamos como nuestro algoritmo encuentra una solución correcta.
Es fácil ver que en efecto el conjunto Z se puede formar combinando las componentes x de cada punto en el plano con las componentes y de los todos puntos en el plano. Como ya hemos demostrado, en ese conjunto seguro hay alguna solución óptima s que en particular cumple que para todo $z \in Z$ $\#P_s \geq \#P_z$.
El algoritmo recorre entonces todos los elementos z del conjunto Z. Mediante la función *cuantosEntran* se cuenta la cantidad de puntos que entran en la caja que tiene como esquina inferior izquierda a z, es decir que devuelve $\#P_z$ y lo conserva como solución parcial si y sólo si es la que mayor puntos cubre hasta ese momento.Es decir que en cada iteración la solución parcial (llamémosla v) cumple que $\#P_v$ es mayor que la todos los puntos z que se evaluaron hasta ese momento.
Luego, cuando se iteraron todos los elementos de Z la solución s que tenemos cumple que $\#P_s \geq \#P_z$ para todo $z \in Z$.
Pero como sabemos que en ese conjunto existe una solución óptima, entonces s también lo es. En particular, lo que puede ocurrir es que devuelva esa solución óptima que demostramos que existe en el conjunto Z (es decir la que tiene puntos en los bordes de la caja) o que devuelva otra que no cumpla con esa condición, pero que al cumplir que $\#P_s \geq \#P_z$ también es solución óptima.

□

3.3.2. Análisis de Complejidad

Analicemos en primer lugar la complejidad de *cuantosEntran*. Asignarle un valor a la variable *int cant* cuesta tiempo constante, es decir $\mathbf{O}(1)$. A su vez, evaluar la condición del If también cuesta tiempo constante.Esto es porque realizar la comparación de *int*'s es constante y, de acuerdo a la documentación de C++, obtener un elemento de un vector mediante el operador [] también lo es. Por lo tanto, evaluar la condición del If es $\mathbf{O}(1)$. Luego, se entre o no al cuerpo del If, en cada iteración del ciclo se ejecuta algo de costo constante. Como el ciclo se itera n veces (siendo n la cantidad de puntos que hay en el plano), el algoritmo *cuantosEntran* tiene un costo de

$$\mathbf{O}(1 + n*1)$$

con lo cual tiene una complejidad temporal **O**(n), es decir lineal en la cantidad de puntos.

Analicemos ahora *resolver*. Las primeras cuatro asignaciones se realizan en tiempo constante (incluso las que obtienen un valor del vector con el operador [ ]), es decir en **O**(1).

Veamos que ocurre en el cuerpo del segundo For. Se ejecuta *cuantosEntran*, que ya sabemos que lo hace en tiempo lineal, y lo que devuelve se lo asigna a una variable (en tiempo constante). Luego se hace una comparación en **O**(1).

Ahora bien, si no entra al cuerpo del If no se hace nada más y se pasa a la próxima iteración. En este caso el costo de la iteración fue de **O**(n+2), con lo cual tiene una complejidad temporal de **O**(n).

En cambio, si entra al cuerpo del If, se realizan tres asignaciones, todas en tiempo constante. Luego lo que está adentro del cuerpo del If tiene un costo de **O**(1). Es decir que en este caso también una iteración cuesta **O**(n).

De aquí se extrae que lo que ocurre en cada iteración de este For tiene un costo lineal. Como este For se ejecuta n veces, su complejidad temporal será **O**(n^2).

Entonces, el primer For ejecuta en cada iteración algo con un costo cuadrático. Como se itera n veces se obtiene que el costo de este For es **O**(n^3). Es decir que el algoritmo cuesta

O(1+1+1+ n^3)

con lo cual tiene un complejidad temporal de **O**(n^3), es decir cúbica en la cantidad de puntos, cumpliendo entonces con la cota que se nos exigía en el enunciado del problema.

3.4. Pruebas y Resultados

Primero queremos realizar una aclaración. Por la manera en que implementamos el algoritmo, no es esperable un caso patológico. Esto es porque el algoritmo siempre itera en las n^2 combinaciones posibles que se obtienen de las componentes x e y de todos los puntos. Quizá es oportuno mencionar que esto se podría optimizar generando un vector de combinaciones y quitando los elementos repetidos. En este caso, si para todo par de puntos de la entrada vale que tienen coordenadas x e y distintas, ese vector de combinaciones sería de tamaño n.

Se realizaron casos de prueba de manera aleatoria. El programa permite la creación se estos casos de prueba si así lo quisiese el usuario. En particular, se crearon tres archivos de entrada generados de manera aleatoria que se pueden encontrar en la misma carpeta que el código fuente. Cada uno de estos archivos contempla casos que contienen desde un punto hasta doscientos. A partir de estos tres archivos, y utilizando la función $get_clocktime()$, se lograron obtener los tiempos de ejecución para cada eso. Con estos datos se confeccionaron los siguientes gráficos que relacionan el tiempo de ejecución del algoritmo con la cantidad de puntos en el plano:

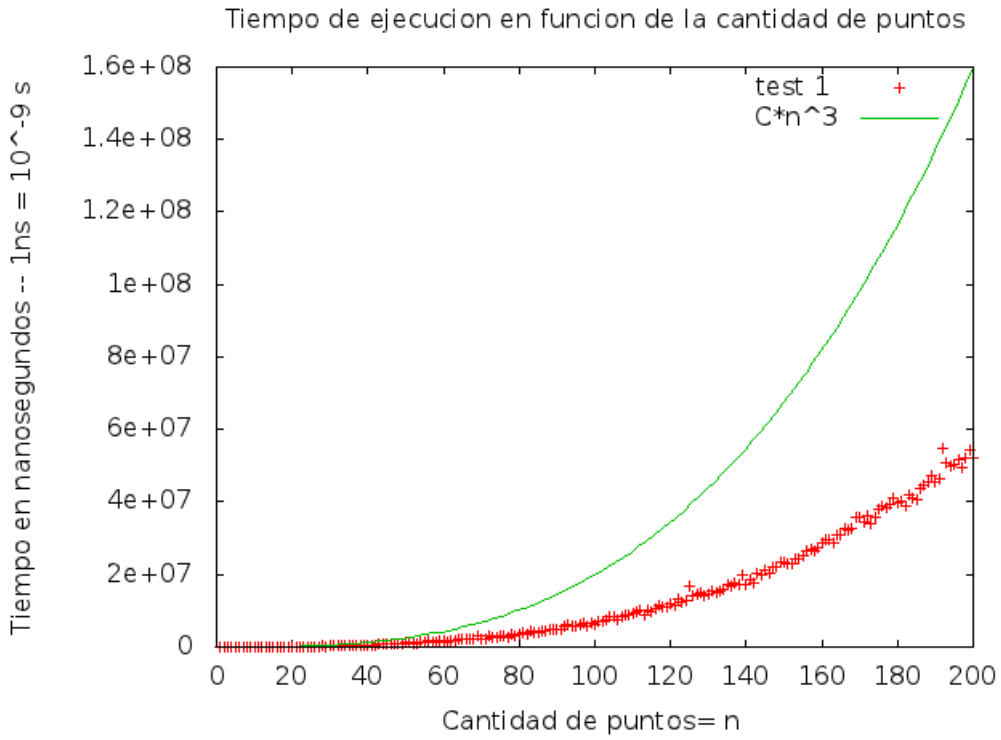

Figura 7: Gráfico de tiempo de ejecución en función de la cantidad de puntos para el archivo de entrada *testingAzar*1.*txt* comparada con la cota teórica O(n^3)

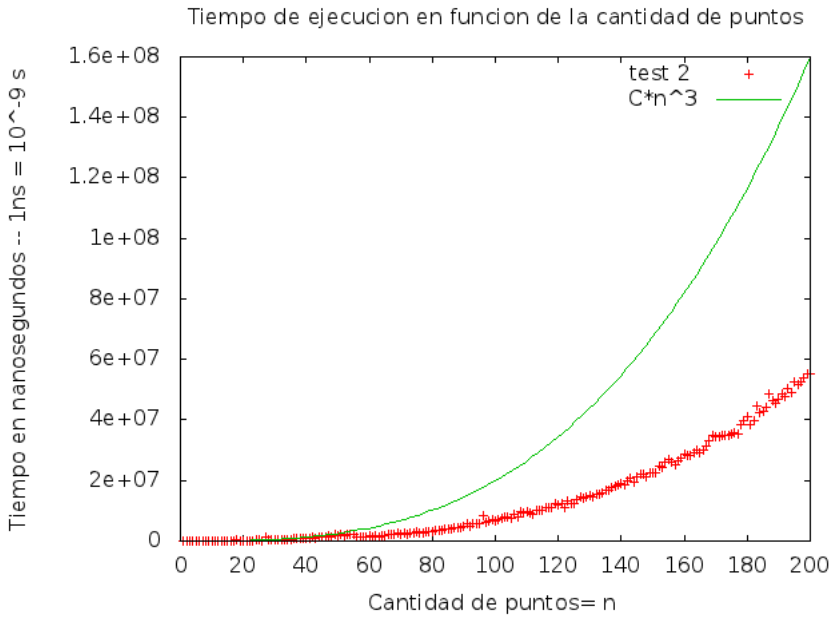

Figura 8: Gráfico de tiempo de ejecución en función de la cantidad de puntos para el archivo de entrada *testingAzar*2.*txt* comparada con la cota teórica O(n^3)

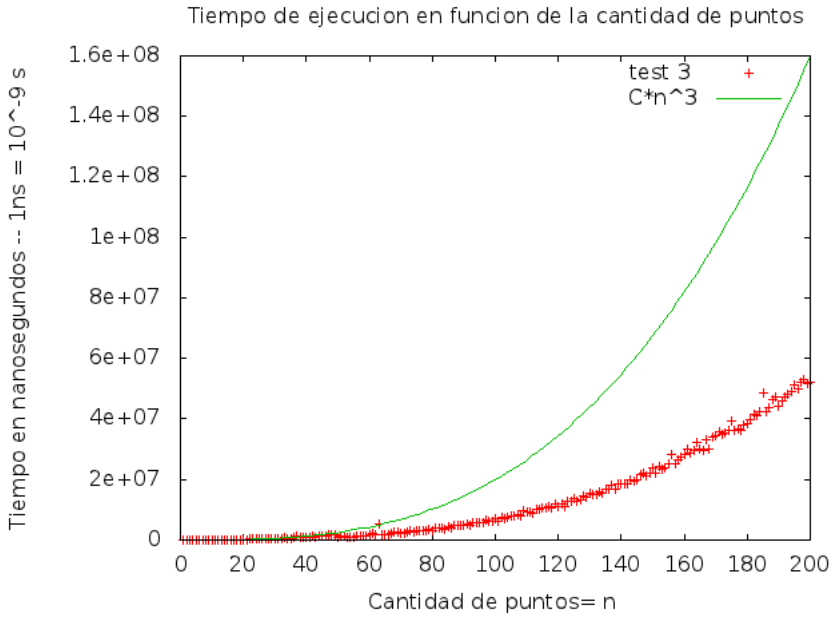

Figura 9: Gráfico de tiempo de ejecución en función de la cantidad de puntos para el archivo de entrada *testingAzar*3.*txt* comparada con la cota teórica O(n^3)

De estos gráficos se puede extraer que los tres casos aleatorios parecen cumplir con la cota teórica cúbica que habíamos conjeturado. Es decir que a partir de algún valor de n todos las mediciones de tiempo se ajustan de la curva $C * n^3$ que representa a la cota teórica.

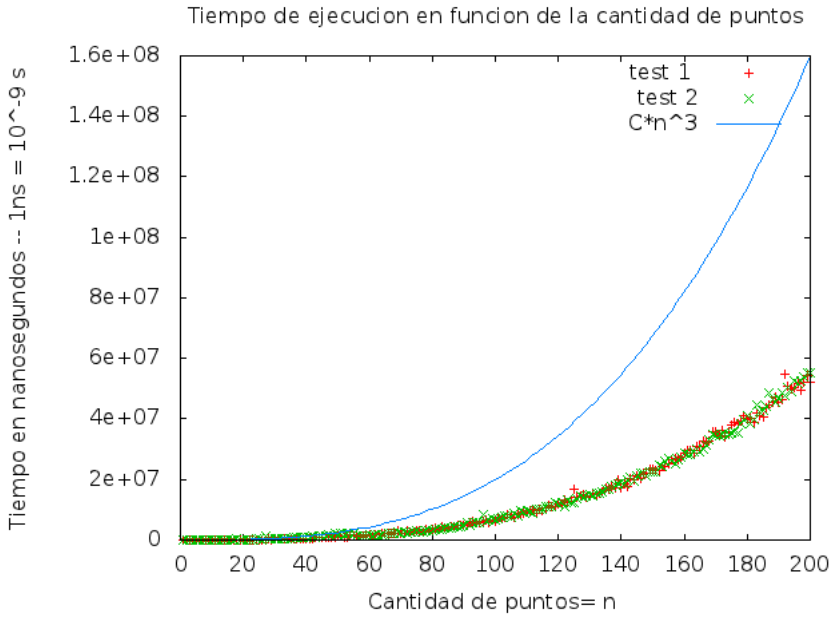

Figura 10: Gráfico de comparativo del tiempo de ejecución en función de la cantidad de puntos para los archivos de entrada *testingAzar*1.*txt* y *testingAzar*2.*txt* comparada con la cota teórica O(n^3)

Este gráfico expone que fijado un n los tiempos de ejecución para dos instancias del problema parecen comportarse en cuanto a tiempo de ejecución de maneras similares. No se proveyó un gráfico que comparase los tres archivos de entradas aleatorias en favor de la visualización del gr'afico.

3.5. Conclusiones

Se concluye que el algoritmo parece responder de la forma esperada en cuanto a performance temporal se refiere, cumpliendo con la cota teórica $\mathbf{O}(n^3)$. Además, a través de los casos de testing estudiados, se deduce que fijada una cantidad de puntos n dos intancias con dicho valor de n se ejecutan en tiempos similares.

4. Problema 4: Puzzle de casilleros

4.1. Introducción

Para este tipo de problemas, se suele usar la técnica de *backtracking*. No se conocen algoritmos polinomiales para resolver el problema. También se podría resolver con fuerza bruta pero la ventaja de esta técnica es que se busca armar e ir probando posibles soluciones. A medida de que se van construyendo, se puede determinar ciertos casos que no son posible solución, por lo que no se intentaría resolver el problema en esos casos evitando ejecuciones innecesarias. En el peor caso, *backtracking* es igual a intentar resolverlo por fuerza bruta. Se nos pide encontrar, si existe, alguno de los menores subconjuntos de piezas de tal forma que cubran todo el tablero. Donde cada casillero del tablero es de color negro o blanco, y cada pieza de forma rectangular, posee casilleros de los mismos colores. La pieza para poder ubicarse en cierta posición del tablero tiene que coincidir exactamente en el tablero y se la puede rotar.

4.2. Desarrollo

Este ejercicio contiene varios problemas. Para empezar, nos encontramos con el requerimiento de dado un conjunto armar todos los subconjuntos posibles. Se conoce como *power set*. La complejidad de este problema es exponencial, no se conocen algoritmos polinomiales hasta el momento.

Reducimos el problema a generar todos los subconjuntos posibles de un tamaño determinado. Para eso, vamos tomando todos los números enteros (uno por vez) del 1 hasta 2 elevado a la cantidad de piezas. Definimos este límite superior debido a que vamos a usar la representación binaria de los números de ese rango. Cada dígito binario, representa si está o no presente en el conjunto, el índice del número que se está mirando en el momento. Nos quedamos con aquellos conjuntos donde la suma de sus dígitos binarios es igual al tamanõ deseado.

Ejemplo: Sea un conjunto de 3 nodos: $\{0, 1, 2\}$

se usa una variable, en este caso *valor*, que va a iría desde 1 a $2^3 - 1$

- valor = 1:

  $(valor)_2 = 001$

  conjunto =$\{0\}$

- valor = 2:

  $(valor)_2 = 010$

  conjunto =$\{1\}$

- valor = 3:

  $(valor)_2 = 011$

  conjunto = $\{0, 1\}$

- valor = 4:

  $(valor)_2 = 100$

  conjunto = $\{2\}$

- valor = 5:

  $(valor)_2 = 101$

  conjunto = $\{0, 2\}$

- valor = 6:

  $(valor)_2 = 110$

  conjunto =$\{1, 2\}$

- valor = 7:

  $(valor)_2 = 111$

  conjunto =$\{0, 1, 2\}$

Para k = 2, los conjuntos parciales serían:

$\{0, 1\}$ con valor = $3 = (011)_2$

$\{1, 2\}$ con valor = $6 = (110)_2$

$\{0, 2\}$ con valor = $5 = (101)_2$

Es muy importante tener en cuenta, que para poder utilizar este método es necesario tener identificado los elementos del conjunto desde 0 hasta $(n - 1)$, donde n es la cantidad del conjunto.

Para resolver el problema principal, lo que hicimos fue ir probando cada uno de los subconjuntos con tamaño desde 1 hasta la cantidad de piezas. Como punto de corte razonable, es haber encontrado algún subconjunto como solución.

Para determinar si cierto subconjunto de piezas es solución, primero determinamos una poda. Esta consiste en que, al buscar una solución óptima, el subconjunto de piezas debía de cubrir exactamente el tablero. Si la superficie de las fichas era menor al del tablero, entonces no era solución. En cambio, si era mayor, podría ser solución, pero si lo era, no era óptima porque sobraba alguna pieza.

Luego implementamos una función recursiva en la cual para cada ficha se buscaba sus posibles posiciones en el tablero. Para cada posición, se creaba una instancia del tablero en la cual se la ubicaba ahí. Luego se llamaba devuelta a la función con ese tablero y sin esa pieza.

El caso base de la función es cuando no quedan más piezas para ubicar. Entonces se fija en el tablero si está completamente cubierto, en caso afirmativo, se encontró una solución al problema.

La función tiene en cuenta las rotaciones de las piezas. Por lo que si no se encontrón solución, se rota la pieza. Si es idéntica a la original, se corta esa rama. Caso contrario, se busca todas las posibles posiciones para esa pieza rotada, y se repite lo anterior.

Una vez que se encuentra solución, si existe, se devuelve el vector con la información de las piezas colocadas en el tablero. Éste vector, lo maneja la clase *Tablero* que hicimos, al cual agrega una pieza cada vez que es posible ubicarla en el tablero.

En todo momento del desarrollo tuvimos que tener en cuenta generar todas las podas posibles para evitar ejecuciones innecesarias debido al tipo de problema.

Creamos las clases *Tablero* y *Pieza* para que cada se encargue de las funciones que les competen propiamente a dicha abstracción.

En la clase *Tablero*, podemos crear un tablero con el tamaño deseado, obtener las posibles posiciones de una pieza, si encaja la pieza en cierta posición, si está completo o no. Si cierto conjutno de piezas puede llegar a cubrir todo el tablero, obtener las piezas ubicadas, etc.

En la clase *Pieza*, podemos crear una pieza de cierto tamaño, rotarla, etc.

Utilizamos top-down y clases porque nos pareció la mejor forma de organizar el programa para resolver el ejercicio. Haciendo más prolijo el código y *tirar* los problemas del tablero se encarga su clase y el de la pieza el de ella misma.

4.2.1. Correctitud

En cuanto la problema de la generación de todos los subconjuntos posibles de un conjunto, utilizamos un método que dada la representación binaria de un número los elementos presentes en un conjunto son aquellos donde la posición es igual al id del elemento y el bit está en 1, es decir, es un elemento presente. Como se recorre desde el valor 1 hasta 2^n (n es cantidad de elementos), se recorre todas las posibles combinaciones de elementos presentes y no presentes. Si bien esta forma sigue siendo un algoritmo exponencial, creemos que es más eficiente que haber hecho una función recursiva que genere todos estos conjuntos.

4.2.2. Análisis de complejidad

Las funciones que utlizamos del tablero son:

- **getColor:** dada una posición del tablero, devuelve el color, negro o blanco. Tiene un costo O(1) debido a que solo tiene que devolver un valor accediendo a una posición de una matriz.
- **posiblesPosiciones:** recorre todo el tablero, fijándose en cada posición, si es posible ubicar la pieza ahí. Para esto llama a la función encaja. Guarda en un vector resultado las posiciones posibles de ubicar la pieza. La complejidad termina siendo la cantidad de posiciones del tablero (filas por columnas) iteraciones de la función encaja: O(filas * columnas * filasPieza * columnasPieza)
- **encaja:** dada una pieza y una posición, devuelve verdadero si es posible ubicar esa pieza en la posición pasada por parámetro. Para esto, recorre toda la superficie de la pieza, por lo que cuesta O(filasPieza * columnasPieza)
- **ocupado:** dada una posición, devuelve si ya está ocupada por una pieza o no. Se tiene que acceder a una matriz, por lo que cuesta O(1).

- **completo:** determina si el tablero está totalmente cubierto por piezas. Para ello, tiene que recorrer todas las posiciones y llamar a la función ocupado. Tiene coste O(filas * columnas)
- **ubicarFicha:** dada una pieza y una posición modifica la instancia de tablero, seteando las posiciones correspondientes para considerarlas ocupadas. Ademaá, guarda en un vector el id de la pieza, el grado de rotación de la misma, y la posición del extremo superior izquierdo. Tiene un costo O(filasPieza * columnasPieza)
- **obtenerPiezas:** la clase tablero guarda un vector con las piezas ubicadas en el mismo. Esta función devuelve una copia de ese vector. Tiene costo O(n) donde n es la cantidad de piezas.
- **cubreTodo:** es una función para podar casos no posibles de ser solución como se explicó anteriormente. Calcula la superficie del tablero. Recorre todas las piezas, restando a la superficie del tablero la superficie de la pieza. Para cubrirlo exactamente tiene que quedar en 0. Caso contrario, se poda. Tiene un costo de O(n) donde n es la cantidad de piezas.

4.3. Conclusiones

Considerando posibles mejoras sería que en vez de buscar *linealmente* cual es la menor cantidad de piezas necesarias, se podría utilizar una busqueda binaria. Con linealmente nos referimos en el sentido de que probamos con una pieza, luego dos, y así sucesivamente hasta probar con todas las piezas. En cambio, con búsqueda lineal, se prueba con la mitad de piezas, si es solución, entonces existe una solución menor o igual a esa. Por lo que se busca una nueva solución entre la 1era mitad en cantidad de piezas. Sino, entonces era necesario más piezas y se busca en la 2da mitad en cantidad de piezas.

Hay que tener en cuenta que al ser un problema que no se resuelve polinomialmente, intentar resolver con un valor grande en cantidad de piezas puede demorar demasiado en comparación si se hubiese hecho búsqueda lineal. Si se pudiera hacer un estudio sobre las piezas y el tablero y se conociera una estadística sobre la entrada del problema, se podría determinar casos donde conviene utlizar uno u otro.

Con este problema, pudimos notar la importancia de determinar buenas podas o puntos de corte para tratar de realizar un backtracking eficiente. De esta forma, se puede llegar a resolver problemas difíciles muchísimo más eficiente que si se hubiese realizado fuerza bruta.