DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Algoritmos y Estructuras de Datos III

Primer Cuatrimestre de 2014

Alumno	LU	E-mail
Aboy Solanes, Santiago	175/12	santiaboy2@hotmail.com
Almansi, Emilio Guido	674/12	ealmansi@gmail.com
Canay, Federico José	250/12	fcanay@hotmail.com
Decroix, Facundo Nicolás	842/11	fndecroix92@hotmail.com

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

Índice

1. Introducción

El objetivo de este informe es describir, desarrollar y presentar una solución algorítmica a tres problemas de maximización/minimización u optimización. Por otro lado, demostraremos la correctitud de las soluciones propuestas, y que su complejidad temporal cumple los requerimientos pedidos. Realizamos diversos experimentos que permiten verificar la correctitud, así como también realizamos experimentaciones computacionales para medir la performance de la implementación de nuestra solución. Los resultados obtenidos y la discusión de los mismos se encuentran en sus secciones correspondientes.

El código fuente de las soluciones se encuentran en su totalidad en la carpeta *src*, mientras que sus secciones más relevantes se pueden leer en los apéndices de este informe.

2. Consideraciones generales

2.1. Lenguaje de implementación

Para implementar las soluciones algorítmicas desarrolladas en cada problema se utilizó el lenguaje C++, el cual presenta una serie de características muy convenientes. Este lenguaje es imperativo, al igual que el lenguaje de pseudocódigo utilizado para describir las soluciones y probar su correctitud. Adicionalmente, el mismo posee librerías estándar muy completas, versátiles y bien documentadas, lo cual permite abstraer el manejo de memoria, la implementación de estructuras de datos y algoritmos de uso frecuente, y provee mecanismos para realizar mediciones de tiempo de manera fidedigna.

2.2. Algoritmo de ordenamiento

En los algoritmos 1 y 2 usamos el algoritmo sort de la stl. Para lograr calcular las complejidades de ambos, necesitamos saber la complejidad de dicho algoritmo.

Buscando en la pagina de http://www.cplusplus.com/reference/algorithm/sort/, encontramos que su complejidad es O(n.log n) comparaciones. Como con sólo esta informacion no podíamos asegurar que tenga complejidad O(n.log n) operaciones, buscamos que hace el algoritmo sort. Encontramos que para casos chicos hace InsertionSort (ERA ESTE?), y en casos mas grandes IntroSort. IntroSort intenta ordenar usando QuickSort, si no lo resuelve en n.log n pasos, usa HeapSort para garantizar O(n.log n) comparaciones. Viendo el código del algortimo llegamos a que ademas de hacer O(n.log n) comparaciones tambien hace a lo sumo O(n.log n) swaps.

Como en ambos casos donde usamos el algoritmo sort de la stl, nuestros parametros son vector¡int¿ sabemos que la comparaciones y swaps son O(1). Por lo cual podemos garantizar que sort tiene una complejidad O(n log n) operaciones.

Dicho esto pasamos a demostrar que la complejidad del algoritmo utilizado para resolver el problema 1 es O(n log n).

2.3. Mediciones de performance

Para llevar a cabo mediciones de performance sobre las implementaciones desarrolladas, se midió el tiempo consumido para resolver instancias de sucesivos tamaños en función de un parámetro a definir según el caso. Se procuró medir exclusivamente el tiempo consumido por la etapa de resolución, ignorando tareas adicionales propias al proceso como, por ejemplo, la generación de la instancia a ser resuelta.

La función del sistema que se escogió para medir intervalos de tiempo es la siguiente:

```
int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

de la librería **time.h**. La misma permite realizar mediciones de alta resolución, específicas al tiempo de ejecución del proceso que la invoca (y no al sistema en su totalidad), configurando el parámetro clk_id con el valor CLOCK_PROCESS_CPUTIME_ID[1].

[1] Referencia http://linux.die.net/man/3/clock_gettime.

Por otro lado, dado que la medición de tiempos en un sistema operativo activo introduce inherentemente un cierto nivel de ruido, cada medición se realizó múltiples veces. Una vez obtenidos los distintos valores para una misma medición (es decir, para instancias de mismo tamaño), se registra como valor definitivo la mediana de la serie de valores. Se escogió este criterio en vez de, por ejemplo, tomar el promedio, ya que es menos suceptible a la presencia de valores atípicos o outliers.

3. Problema 1: Camiones sospechosos

3.1. Descripción del problema

Bajo la sospecha de que una determinada empresa de transporte trafica sustancias ilegales en sus camiones, se ha contratado a un experto para inspeccionar los vehículos durante su paso por un puesto de control. El período de contratación del experto tiene una duración de D días consecutivos, con fecha de inicio a determinar. Adicionalmente, se cuenta con un listado de n entradas, indicando los días $d_1, d_2, ..., d_n$ en que los camiones de la empresa pasarán por el puesto de control.

Se desea conocer la máxima cantidad c de camiones que el experto puede llegar a inspeccionar durante su período de contratación, y un día d para tomar como fecha de inicio de forma tal que efectivamente logre inspeccionar dicha cantidad de camiones.

Formalmente, dado D un numero natural y una lista no vacía de n números naturales $d_1, d_2, ..., d_n$ (no necesariamente distintos), se desea hallar un valor d natural de forma tal que el intervalo $[d, d + D)$ contenga a la máxima cantidad posible de elementos de la lista. Además, se desea conocer la cantidad total c de elementos contenidos en el intervalo.

Cada instancia del problema, así como su solución, se codifica como una lista de naturales separados por espacios, representando los siguientes valores:

Entrada: $D\ n\ d_1\ d_2\ ...\ d_n$
Salida: $d\ c$

3.1.1. Ejemplos y observaciones

La lista de días de llegada de los camiones no necesariamente se encuentra ordenada, y además puede contener días repetidos, que deben ser contados con su debida multiplicidad al computar la solución. Esto es razonable ya que muchos camiones pueden llegar al puesto de control en un mismo día. Por lo tanto, el siguiente es un ejemplo de instancia válida y su respectiva solución:

Entrada: 1 5 23 23 23 23 2
Salida: 23 4

Por otro lado, es importante notar que si bien la cantidad máxima c de camiones inspeccionados es única, la fecha de inicio del período de contratación no lo es. Distintos valores para d pueden generar intervalos $[d, d + D)$ incluyendo igual cantidad de elementos de la lista.

Entrada: 3 2 6 7
Salida: 5 2 ó
Salida: 6 2

3.2. Desarrollo de la solución

El algoritmo desarrollado para resolver el problema considera como posibles candidatos para fecha de inicio únicamente a los valores del conjunto $\{d_1\ d_2\ ...\ d_n\}$, haciendo uso de la siguiente propiedad (demostración en la sección 3.4):

Propiedad 1. *Siempre existe un intervalo óptimo $[d, d + D)$ (en el sentido de que comprende la mayor cantidad posible de elementos de la lista) cuyo límite inferior sea $d \in \{d_1, d_2, ..., d_n\}$.*

Para cada candidato d_i, se computa la cantidad de elementos contenidos en el intervalo $[d_i, d_i + D)$, guardando un registro del día d para el cual dicha cantidad es mayor. Finalmente, se emite como solución el valor de d y la cantidad c de números contenidos en su respectivo intervalo.

Dado el requerimiento de que la solución tenga complejidad temporal subcuadrática, no es viable el procedimiento ingenuo de tomar cada día de la lista d_i y, para cada valor $1 \leq j \leq n$, evaluar la condición de pertenencia $d_i \leq d_j < d_i + D$.

En cambio, la resolución propuesta realiza primeramente un ordenamiento sobre la lista, y luego recorre la misma linealmente mediante dos índices i, j, manteniendo en cada iteración el siguiente invariante[2]:

[2]Los subíndices refieren a los elementos de la lista ya ordenada

```
procedure CAMIONES-SOSPECHOSOS(D, n, ⟨d_1, ..., d_n⟩)
    ordenar(⟨d_1, ..., d_n⟩)                                   O(n log(n))
    d ← 0, c ← 0                                               O(1)
    i ← 1, j ← 1                                               O(1)
    while i ≤ n do                                             O(1)
        if 0 < i ∧ d_{i−1} = d_i then continue                 O(1)
        while (j ≤ n) ∧ (d_i ≤ d_j < d_i + D) do               O(1)
            j ← j + 1                                          O(1)
        if c < j − i then                                      O(1)
            c ← j − i                                          O(1)
            d ← i                                              O(1)
        i ← i + 1                                              O(1)
    return d, c
```

Figura 1: Camiones Sospechosos. Pseudocódigo.

$$(\forall k : i \leq k < j)\ d_k \in [d_i, d_i + D)$$

Si sucede que $d_{i-1} = d_i$, se está en presencia de un candidato que ya ha sido contemplado, y por lo tanto se saltea. De lo contrario, vale $d_{i-1} < d_i$ y por lo tanto d_i es el primer elemento de la lista contenido en $[d_i, d_i + D)$. Esto permite computar la cantidad total de elementos contenidos en el intervalo mediante la expresión $j - i$, cuyo valor se utiliza para actualizar la solución parcial.

Adicionalmente, la búsqueda del máximo j que cumple el invariante puede realizarse a partir del valor de j de la iteración anterior[3], fundamentado en la siguiente propiedad (demostración en la sección 3.4):

Propiedad 2.

$$((\forall k : i \leq k < j)\ d_k \in [d_i, d_i + D)) \Rightarrow ((\forall k : i + 1 \leq k < j)\ d_k \in [d_{i+1}, d_{i+1} + D))$$

Es decir, si vale el invariante para (i, j), entonces vale para $(i + 1, j)$. Esto permite obtener el valor máximo de j correspondiente a cada i mediante un único recorrido lineal, ya que j nunca decrece. En la figura 1 se incluye pseudocódigo describiendo el algoritmo desarrollado en detalle, y en el apéndice A se incluye el código completo.

3.3. Complejidad temporal

El pseudocódigo de la solución (figura 1) incluye al final de cada línea la complejidad temporal de las instrucciones contenidas en la misma.

En primer lugar, la etapa de ordenamiento es realizable en tiempo $O(n\ log\ n)$ mediante conocidos algoritmos como Mergesort, Heapsort o Introsort.

Luego, dado que j se inicializa en 1 y su valor es incrementado en el cuerpo del ciclo interno, la guarda del mismo no puede ser verdadera más de n veces. Como además siempre vale que $d_i \leq d_i < d_i + D$, este se ejecuta al menos una vez por cada valor de i. Por lo tanto, la operación $j \leftarrow j + 1$ se ejecuta n veces, incurriendo en un costo lineal en n en la totalidad del algoritmo (no así por cada valor de i).

Por otro lado, el ciclo externo se ejecuta exactamente n veces, con $i = 1, ..., n$. En peor caso, los elementos son todos distintos y jamás se saltea el cuerpo del ciclo. Las operaciones de actualización del máximo tienen un costo constante (incluso si hay que realizar la actualización), por lo cual en la totalidad del algoritmo comprenden una cantidad de operaciones proporcional a n.

En conjunto, el procedimiento de búsqueda del máximo tiene una complejidad $O(n)$; por esta razón, la etapa de ordenamiento domina la complejidad del algoritmo, permitiendo dar la cota $O(n\ log(n))$ para la solución.

[3]En el caso base, se toma $j = 1$ para comenzar a partir del primer elemento.

3.4. Demostración de correctitud

Para demostrar la correctitud del algoritmo, vamos a demostrar 2 propiedades:

Primero queremos ver que siempre existe una solucion optima que empieza en algunos de los dias dados.

$(\forall\ n,\ D\ , S = \{d_1, .., d_n\})\ \exists\ 0 < i \leq n\ /\ [d_i, d_i + D)$ es optimo

Tomo cualquier solucion optima T que empieza en el dia d'. Si $(\exists i : 0 < i \leq n)\ d' = d_i$ para algun 0 < i ≤ n, encontre lo que buscaba.

Si $(\forall i : 0 < i \leq n)\ d' \neq d_i$, divido en 2 subcasos

Si $(\exists i : 0 < i \leq n)\ d' < d_i$, tomo el minimo i / d' $< d_i$ Ahora construyo una nueva solucion desde d_i, veamos que todo elemento de S perteneciente a [d',d'+D) tambien pertenece a $[d_i, d_i + D)$

sea $d_j \in [d', d' + D) \Rightarrow d' \leq d_j < d' + D$

q.v.q.$d_j \in [d', d' + D) \Rightarrow d_i \leq d_j < d_i + D$

$d' \leq d_j < d' + D \Rightarrow d_i \leq d_j < d_i + D \Leftrightarrow d' \leq d_j \Rightarrow d_i \leq d_j$ como sabemos que $d' < d_i$ este es valido ya que sabemos que d_i es el minimo elemento de S mayor que d' $\Rightarrow (!\exists k : 0 < k \leq n) d' < d_j < d_i$

Ahora queremos demostrar la propiedad 2 : $((\forall k : i \leq k < j)\ d_k \in [d_i, d_i + D)) \Rightarrow ((\forall k : i + 1 \leq k < j)\ d_k \in [d_{i+1}, d_{i+1} + D))$

Vamos a demostrar esto por el absurdo. Tomando como verdadero el antecedente y suponiendo falso el consecuente.

Para eso suponemos que $(\exists k' : i + 1 \leq k' < n)\ d_{k'} \notin [d_{i+1}, d_{i+1} + D)$

Primero vamos a enumerar varias propiedades que sabemos por el antecedente y porque los elementos d_i estan ordenados

A : $d_i < d_{i+1}$

B : $d_{i+1} \leq d_{k'}$ ya que $i + 1 \leq k'$

C : $d_{k'} < d_i + D$ ya que $d_{k'} \in [d_i, d_i + D)$, por el antecedente

q.v.q $d_{k'} \notin [d_{i+1}, d_{i+1} + D)$ es absurdo.

Equivalentemente: $d_{k'} < d_{i+1} \vee d_{k'} \geq d_{i+1} + D$

$d_{k'} < d_{i+1}$ es falso por B.

$d_{k'} \geq d_{i+1} + D > d_i + D$ por A.

$d_{k'} > d_i + D$ lo cual es falso por C.

Al ser falso ambas partes del ∨, la afirmacion es falsa, lo cual nos lleva al absurdo que queriamos encontrar.

3.5. Experimentación

La primer etapa de experimentación consistió en verificar empíricamente la cota de complejidad temporal obtenida teóricamente para el algoritmo completo. Estas primeras cuatro figuras nos permiten ver que efectivamente nuestro algoritmo es $O(n * log(n))$

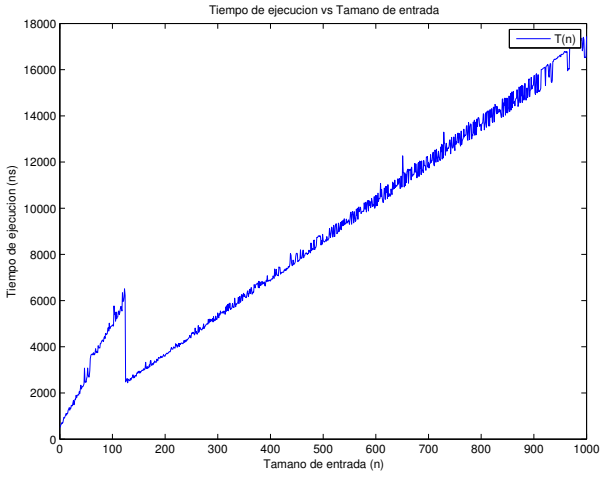

Figura 1: Caso $n = 1000$

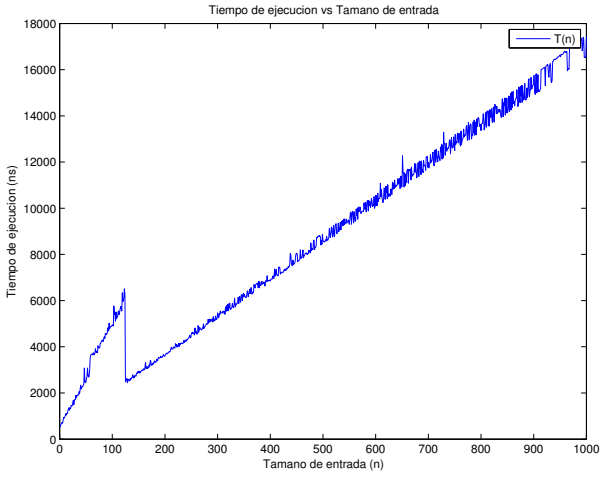

Figura 2: Idem, pero dividido por $log(n)$

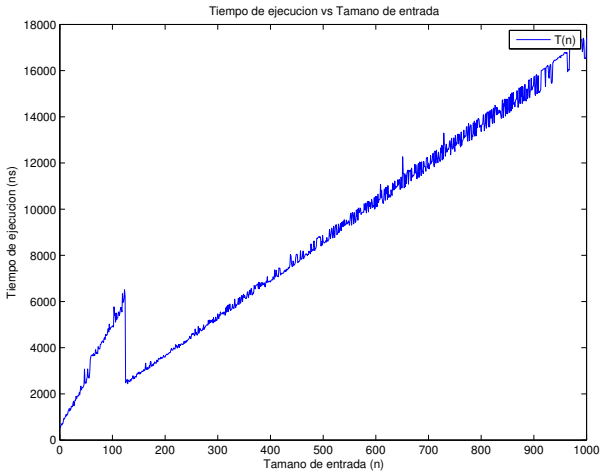

Figura 3: Caso $n = 10000$

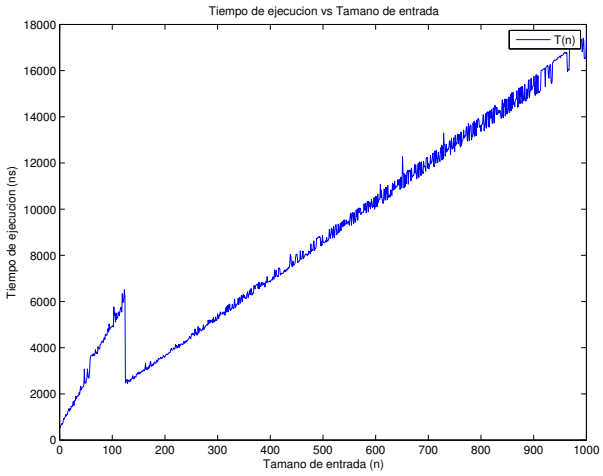

Figura 4: Idem, pero dividido por $log(n)$

Como se puede observar en las figuras 2 y 4, una vez que dividimos por *logn* nuestro algoritmo es una recta. Esto significa que la complejidad del algoritmo en su totalidad es de $O(n * log(n))$.

En los gráficos presentados arriba, se puede observar un pico alrededor de $n = 64$. Como hablamos anteriormente, esto se debe al algoritmo de sorting que realiza un sorting especial con n relativamente chico.

Según nuestro análisis, la complejidad temporal de la solución es dominada por la etapa de ordenamiento. Dado que el algoritmo utilizado para este fin pertenece a librerías estándares, la experimentación subsiguiente se realizó sobre instancias donde la lista de entrada se encuentra ordenada, eliminando la instrucción de ordenamiento. Esto permitió constatar si el ciclo final incurre efectivamente en un costo a lo sumo lineal, y verificar la preponderancia del ordenamiento como parte de la solución.

En las figuras siguientes podemos observar que, sin contar la etapa de ordenamiento, efectivamente nuestro algoritmo es $O(n)$. Para lograr esto, simplemente utilizamos entradas en las cuales los camiones estén ordenados, y por esto no haga falta realizar el sorting.

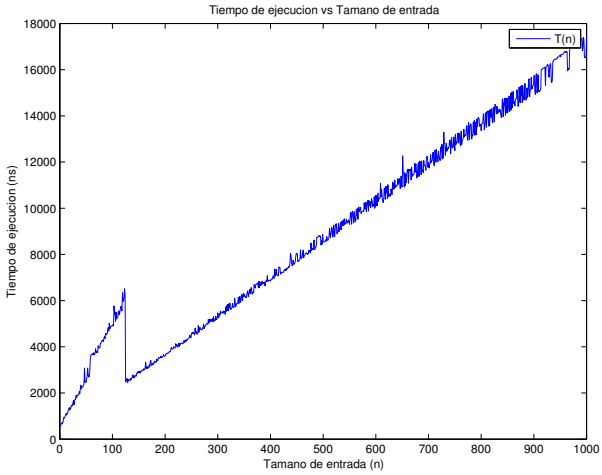

Figura 5: Caso n=1000, sin sorting

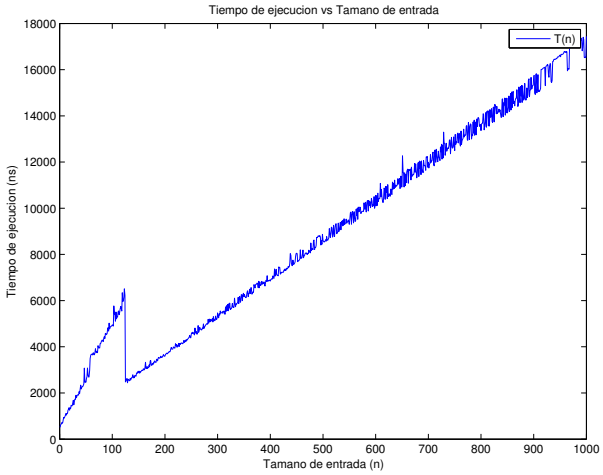

Figura 6: Idem, pero n=10000

3.6. Conclusión

Para concluir, nos parece importante destacar que la complejidad temporal de la solución es dominada por el algoritmo de ordenamiento. Esto implica que no va a ser posible mejorar la cota de complejidad teórica de nuestra solución. Sin embargo, cabe destacar que en el caso en que los datos están ordenados al momento de leer, podemos mejorar la complejidad temporal de $O(n * logn)$ a $O(n)$.

4. Problema 2: La joya del Río de la Plata

4.1. Descripción del problema

Este problema se trata de un joyero que debe fabricar un conjunto de piezas para luego venderlas.

La problemática radica en que cada pieza i de este conjunto tiene una cantidad de días que requiere para su fabricación (t_i), y además cada una pierde una fracción de su valor (p_i)por cada día que pasa.

Lo que este problema nos pide hacer un algoritmo que determine un orden para la fabricación de estas piezas que minimice las perdidas, y además mostrar cuál es dicha pérdida. La complejidad del algoritmo utilizado debe ser $O(n^2)$.

A continuación vamos a dar un ejemplo del problema planteado junto con su solución. Supongamos que tenemos las siguientes piezas:

Pieza	Pérdida	Tiempo
1	3	1
2	2	1
3	1	1

En este ejemplo la solución es la siguiente secuencia:

Solución = [Pieza 1, Pieza 2, Pieza 3]

La pérdida total para esta solución es: $3 * 1 + 2 * 2 + 1 * 3 = 10$

En cambio si eligiéramos como solución otra secuencia, como por ejemplo [Pieza 2, Pieza 1, Pieza 3], la pérdida total sería de: $2 * 1 + 3 * 2 + 1 * 3 = 11$. Si eligieramos [Pieza 3, Pieza 2, Pieza 1], la pérdida total sería de: $1 * 1 + 2 * 2 + 3 * 3 = 14$.

4.2. Desarrollo de la solución

Hallamos que el orden óptimo es de menor a mayor según el coeficiente $\frac{Tiempo_i}{Perdida_i}$, lo cual está demostrado en su sección correspondiente. Siguiendo este orden, podemos asegurar que una solución óptima cumple con el siguiente invariante:

$$(\forall i \in [1..\#piezas)) \frac{Tiempo_i}{Perdida_i} < \frac{Tiempo_{i+1}}{Perdida_{i+1}}$$

Lo cual es equivalente a:

$$(\forall i \in [1..\#piezas)) Tiempo_i * Perdida_{i+1} < Tiempo_{i+1} * Perdida_i$$

Este último es el que usamos debido a que al usar enteros, preferimos usar multiplicación antes que división.

Nuestro algoritmo crea un vector, y coloca una a una las piezas una atrás de otra. Luego, ordena según el orden previamente enunciado. Finalmente, calcula la pérdida total. El pseudocódigo de nuestro algoritmo es el siguiente:

```
Tipo de dato Pieza es Tupla ⟨ id : entero, pérdida : entero, tiempo : entero ⟩
procedure La joya del Río de la Plata(⟨p_1, ..., p_n⟩, L)
    V ← nuevo vector de Pieza
    Cargo_Piezas(V)
    Sort(V)        // Este sort se hace de menor a mayor según el coeficiente anteriormente dicho
    return V, Calcular_perdida(V)
```

4.3. Complejidad temporal

Nuestro algoritmo tiene una complejidad de O(n.log n) operaciones. Para eso vamos a analizar el pseudocodigo de a partes.

Tipo de dato Pieza es Tupla ⟨ id : entero, pérdida : entero, tiempo : entero ⟩
procedure La joya del Río de la Plata($\langle p_1, \ldots, p_n \rangle, L$)
$V \leftarrow$ nuevo vector de Pieza $O(1)$
Cargo_Piezas(V) $O(n)$
Sort(V) $O(n * log(n))$
// Este sort se hace de menor a mayor según el coeficiente anteriormente dicho
return V, Calcular_perdida(V) $O(n)$

$V \leftarrow$ **nuevo vector de Pieza**

Tiene complejidad O(1), ya que es simplemente crear un vector.

Cargo_Piezas(V)

Lo que hace la función Cargo_Piezas, es leer la entrada que nos pasan y agregar las piezas al vector creado anteriormente. Esto tiene complejidad O(n), ya que agregar n elementos a un vector tiene costo O(n).

Sort(V)

Como vimos en la sección del ejercicio 1, la función Sort tiene complejidad O(n.log n).

return V, Calcular_perdida(V)

Devolver el vector es O(1), y calcular la pérdida es O(n), ya que hay que recorrer todas las piezas.

Por algebra de ordenes O(1) + O(n) + O(n.log n) + O(1) + O(n) = O(n.log n), como queriamos demostrar.

4.4. Demostración de correctitud

Para resolver este problema tenemos que encontrar un orden óptimo para armar las piezas.

La propiedad que queremos demostrar es la siguiente:

Sea S un conjunto de elementos $s_1, ..., s_n$ y R un permutación de los elementos de S / $(\forall\ 1 \leq i < n)$ $\frac{t(R[i])}{p(R[i])} \leq \frac{t(R[i+1])}{p(R[i+1])}$, minimiza la función C(R)

Siendo C(R) = $\sum_{i=1}^{n} t(R[i]) \sum_{j=i}^{n} p(R[j])$

Para demostrar esto vamos a hacer inducción en el tamaño de R.

El caso base es ‖ R‖ = 1 :

Este caso es trivial porque sólo existe una permutación de R, por lo cual claramente es la mínima.

Para continuar con la demostración debemos realizar el paso inductivo, que es el siguiente:

$(\forall n > 1)$ P(n-1) ⇒ P(n)

Para realizar el paso inductivo vamos a usar como Hipótesis inductiva que vale P(n-1) y a partir de eso vamos a demostrar que vale P(n).

Tomamos un permutación óptima R = $(r_1, ..., r_n)$ y construyo R' = $(r'_1, ..., r'_n)$/ $r'_1 = r_1$ *y* $(r'_2, ..., r'_n)$ es una permutacion de $(r_2, ..., r_n)/(\forall 2 \leq i < n) \frac{t(r'_i)}{p(r'_i)} \leq \frac{t(r'_{i+1})}{p(r'_{i+1})}$

Osea, R' tiene el primer elemento igual al primer elemento de R, y los otros $n - 1$ elementos están ordenados según nuestro orden propuesto.

Primero vamos demostrar que R' es óptima, para esto calculo C(R) y C(R') :

C(R) = $t(r_1) \sum_{j=1}^{n} p(r_j) + C(R[2..n])$

C(R') = $t(r_1) \sum_{j=1}^{n} p(r'_j) + C(R'[2..n])$

Como R es optimo, se que C(R) ≤ C(R')

Tambien se por H.I. que R'[2..n] es óptima, por lo cual:

C(R'[2,n]) $\leq$ C(R[2..n]) $\iff$

$t(r_1)\sum_{j=1}^{n} p(r_j)$ + C(R'[2..n]) $\leq t(r_1)\sum_{j=1}^{n} p(r_j)$ + C(R[2..n])

Sabemos que $\sum_{j=1}^{n} p(r_j) = \sum_{j=1}^{n} p(r'_j)$ ya que R' es una permutacion de R, Entonces:

$t(r_1)\sum_{j=1}^{n} p(r'_j)$ + C(R'[2..n]) $\leq t(r_1)\sum_{j=1}^{n} p(r_j)$ + C(R'[2..n]) $\iff$

C(R') $\leq$ C(R)

Pero habíamos dicho que R es óptimo, por lo tanto C(R) $\leq$ C(R').

Entonces, como C(R') $\leq$ C(R) $\wedge$ C(R) $\leq$ C(R'), entonces C(R) = C(R'). Por lo tanto R' es optimo.

Por ultimo queremos ver que R' cumple con la condición de P(n), sabemos que R'[2..n] tiene a sus elementos ordenados según $\frac{t(r'_i)}{p(r'_i)}$. Nos falta ver que R' completa esta ordenada, para esto sólo hace falta ver r'_1 esta ordenado, que es lo mismo que decir que $\frac{t(r'_1)}{p(r'_1)} \leq \frac{t(r'_2)}{p(r'_2)}$

Para esto tomamos R" = $(r'_2, r'_1, r'_3, ..., r'_n)$

C(R') = $t(r'_1)\sum_{j=1}^{n} p(r'_j) + t(r'_2)\sum_{j=2}^{n} p(r'_j) + C(R'[3,n])$

C(R") = $t(r'_2)\sum_{j=1}^{n} p(r'_j) + t(r'_1)\sum_{j=1,j\neq 2}^{n} p(r'_j) + C(R''[3,n])$

C(R') $\leq$ C(R") por ser R' optimo

$t(r'_1)\sum_{j=1}^{n} p(r'_j) + t(r'_2)\sum_{j=2}^{n} p(r'_j) + C(R'[3,n]) \leq t(r'_2)\sum_{j=1}^{n} p(r'_j) + t(r'_1)\sum_{j=1,j\neq 2}^{n} p(r'_j) + C(R''[3,n]) \iff$

Como R'(3..n) = R"(3..n) entonces C(R'(3..n)) = C(R"(3..n)) y los puedo cancelar.

$t(r'_1)\sum_{j=1}^{n} p(r'_j) + t(r'_2)\sum_{j=2}^{n} p(r'_j) \leq t(r'_2)\sum_{j=1}^{n} p(r'_j) + t(r'_1)\sum_{j=1,j\neq 2}^{n} p(r'_j) \iff$

$t(r'_1)\sum_{j=1,j\neq 2}^{n} p(r'_j) + t(r'_1) * p(r'_2) + t(r'_2)\sum_{j=2}^{n} p(r'_j) \leq t(r'_1)\sum_{j=1,j\neq 2}^{n} p(r'_j) + t(r'_2)\sum_{j=2}^{n} p(r'_j) + t(r'_2) * p(r'_1) \iff$

Cancelo un término:

$t(r'_1) * p(r'_2) + t(r'_2)\sum_{j=2}^{n} p(r'_j) \leq +t(r'_2)\sum_{j=2}^{n} p(r'_j) + t(r'_2) * p(r'_1) \iff$

Cancelo el otro:

$t(r'_1) * p(r'_2) \leq t(r'_2) * p(r'_1)$

$\frac{t(r'_1)}{p(r'_1)} \leq \frac{t(r'_2)}{p(r'_2)}$

□

4.5. Experimentación

La primer etapa de experimentación consistió en verificar empíricamente la cota de complejidad temporal obtenida teóricamente para el algoritmo completo. Estas primeras cuatro figuras nos permiten ver que efectivamente nuestro algoritmo es $O(n * log(n))$

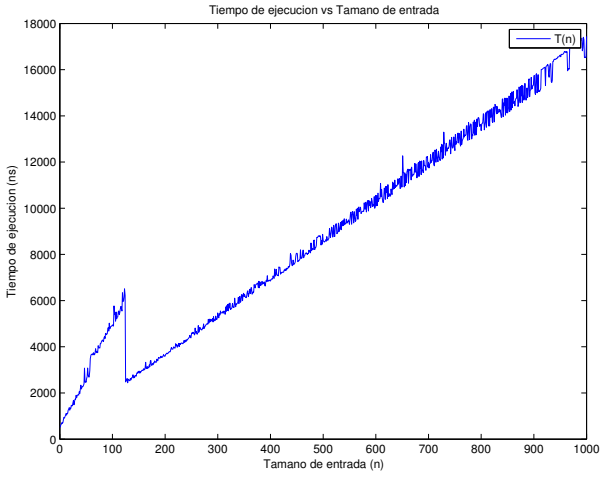

Figura 1: Caso $n = 1000$

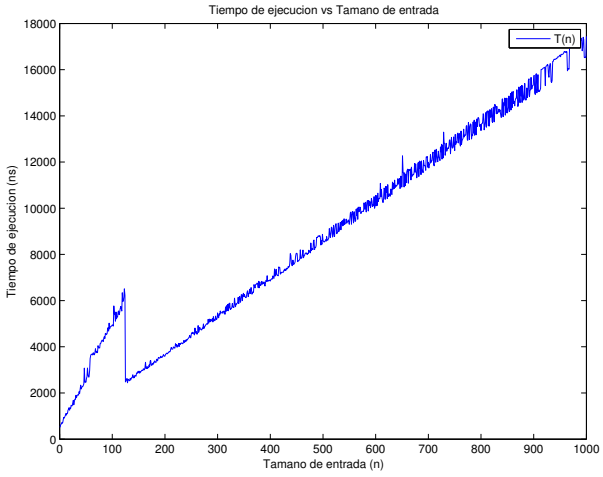

Figura 2: Idem, pero dividido por $log(n)$

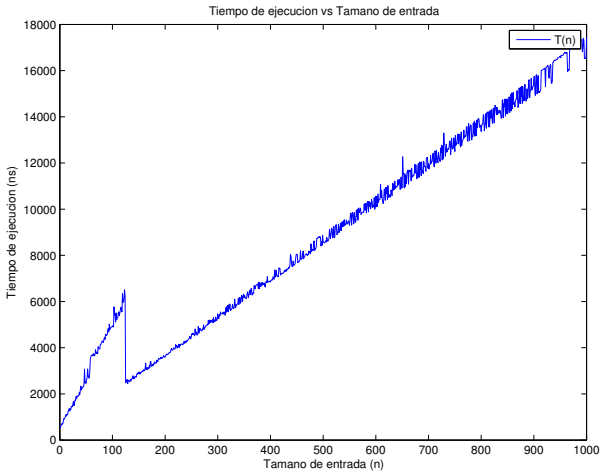

Figura 3: Caso $n = 10000$

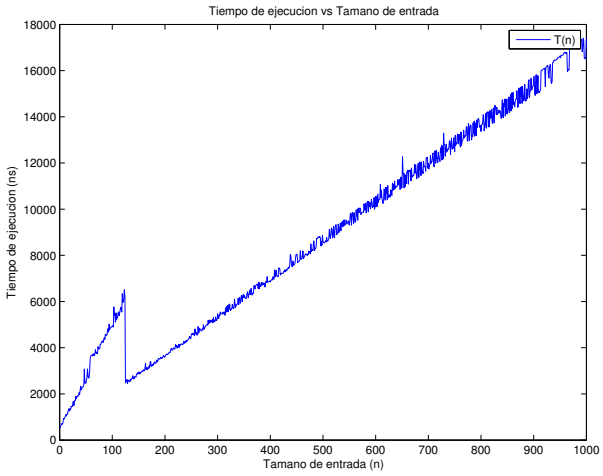

Figura 4: Idem, pero dividido por $log(n)$

Como se puede observar en las figuras 2 y 4, una vez que dividimos por *logn* nuestro algoritmo es una recta. Esto significa que la complejidad del algoritmo en su totalidad es de $O(n * log(n))$.

En los gráficos presentados arriba, se puede observar un pico alrededor de $n = 64$. Como hablamos anteriormente, esto se debe al algoritmo de sorting que realiza un sorting especial con n relativamente chico.

Según nuestro análisis, la complejidad temporal de la solución es dominada por la etapa de ordenamiento. Dado que el algoritmo utilizado para este fin pertenece a librerías estándares, la experimentación subsiguiente se realizó sobre instancias donde la lista de entrada se encuentra ordenada, eliminando la instrucción de ordenamiento. Esto permitió constatar si el ciclo final incurre efectivamente en un costo a lo sumo lineal, y verificar la preponderancia del ordenamiento como parte de la solución.

En las figuras siguientes podemos observar que, sin contar la etapa de ordenamiento, efectivamente nuestro algoritmo es $O(n)$. Para lograr esto, simplemente utilizamos entradas en las cuales las joyas estén ordenadas según el coeficiente anteriormente mencionado, y por esto no haga falta realizar el sorting.

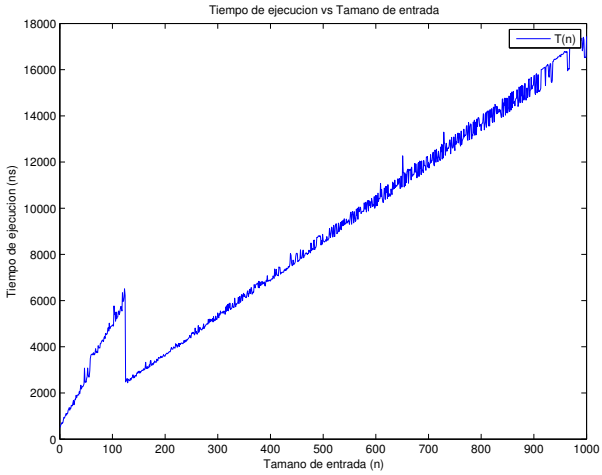

Figura 5: Caso n=1000, sin sorting

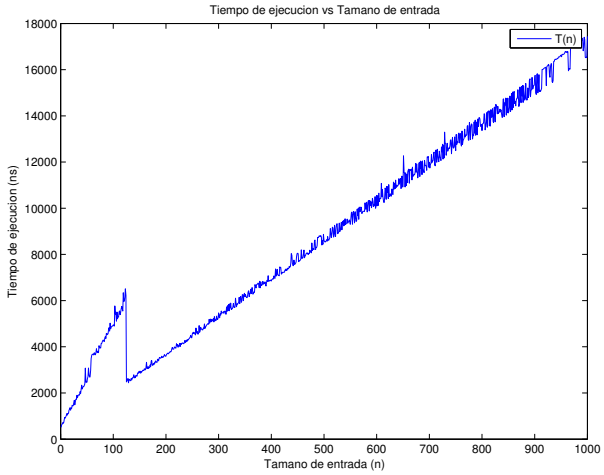

Figura 6: Idem, pero n=10000

Cabe aclarar que en el caso de este problema, al remover la etapa de ordenamiento no existe ni mejor caso ni peor caso (O mejor dicho, todos los casos son θ(n)). Esto se debe a que este algoritmo recorre una vez el vector para guardar los datos, y una vez más para calcular las pérdidas. Siempre realiza esto, sin importar los datos.

Por otro lado, como utilizamos el sort de la STL, no podemos asegurar si un caso en particular va a ser un "mejor caso", o un "peor caso". Esto se debe a que, como lo explicamos anteriormente, el sorting utiliza un *intro sort* que usa primordialmente un *quick sort*. Como ya sabemos, el *quick sort* depende del azar si es un buen caso o un peor caso y por esto no se puede saber *a priori* si un caso va a ser un "mejor caso", o un "peor caso".

Por estas razones, no incluimos gráficos extra en los cuales hablamos de "peor caso" y "mejor caso".

4.6. Conclusión

Praesent et rutrum tortor. Proin sit amet mi blandit, posuere justo eu, feugiat lacus. Nam luctus auctor condimentum. Sed porttitor auctor neque, eget condimentum arcu venenatis vel. In aliquam nibh ut quam vulputate, et euismod augue iaculis. Morbi blandit a dolor sit amet suscipit. Praesent consectetur purus ac elit sagittis, nec aliquam augue lobortis. Nam non sem leo. Pellentesque fringilla, enim et commodo lacinia, urna risus condimentum sapien, eget scelerisque eros enim sed ipsum. Pellentesque euismod consectetur cursus. Vivamus lobortis justo ante, cursus posuere nibh egestas in. Nunc tempor mollis sapien, et dapibus erat. Etiam id sagittis mauris, laoreet fermentum turpis. Donec enim nunc, rutrum pretium quam non, malesuada interdum libero. Praesent sed sagittis nisl, sit amet eleifend lorem.

Vestibulum sit amet suscipit leo, et egestas velit. Quisque vitae volutpat felis, id venenatis mauris. Donec bibendum lacinia tristique. Curabitur quis dictum velit. Etiam eget arcu condimentum, placerat ante et, hendrerit turpis. Pellentesque nec molestie magna, id facilisis tortor. Nam mollis, nunc in pulvinar lobortis, mi nibh volutpat nibh, non fermentum nisi diam ac lorem.

Aenean odio lorem, congue id aliquam at, semper in ligula. Sed auctor neque eget est pulvinar suscipit. Aenean molestie lorem ut viverra volutpat. Nulla vitae augue nec lorem hendrerit eleifend. Pellentesque ut malesuada ante, a facilisis nulla. Nam eleifend vel dolor non mollis. Fusce consequat sit amet nulla nec consequat.

Praesent ultrices sem non elit semper, ut semper arcu tempor. Duis elementum eros sed massa facilisis, id convallis orci pharetra. Mauris euismod suscipit turpis. Nulla quis nisi eu mauris dapibus viverra. Donec nulla urna, eleifend ac odio sit amet, sodales pretium velit. Nunc hendrerit quam neque, aliquet condimentum nisl hendrerit eget. Etiam pharetra hendrerit nisl at sodales. Vivamus at tortor metus. Donec ultrices turpis libero, ac sollicitudin tortor tincidunt sed. Nam molestie dui dignissim, imperdiet elit ac, dignissim diam. Mauris iaculis nisl lectus, accumsan mattis felis iaculis in. Nam elit diam, tristique in ornare ac, condimentum sed odio. Praesent et gravida est. Donec nec justo id neque gravida iaculis ac eu orci. Vestibulum volutpat est in tellus volutpat tincidunt. Sed et est lacus.

5. Problema 3: Rompecolores

5.1. Descripción del problema

Este problema: plantea la siguiente situación, se tiene un tablero de dimensiones $n * m$ y la misma cantidad de piezas y el objetivo es insertar la mayor cantidad posible de piezas en dicho tablero. Las piezas son cuadradas y tienen un color en el lado superior, uno en el lado inferior, uno en el lado derecho y uno en el lado izquierdo. Una pieza se puede colocar adyacente a otra sólo si sus lados coinciden (por ejemplo si una pieza tiene el lado izquierdo de color rojo, a su izquierda sólo se puede colocar una pieza cuyo lado derecho también sea rojo). La otra variable que tiene este problema es la cantidad de colores posibles.

Por ejemplo supongamos que tenemos un tablero de $2 * 2$ y que hay 4 colores disponibles: Amarillo, Azul, Rojo y Verde. Supongamos, además, que contamos con las siguientes piezas:

Pieza	Izq	Der	Sup	Inf
1	Amarillo	Azul	Rojo	Verde
2	Rojo	Azul	Verde	Amarillo
3	Azul	Amarillo	Azul	Verde
4	Amarillo	Rojo	Verde	Azul

Una posible soluciòn (que pondrìa todas las fichas en el tablero) de esta instancia del problema sería:

1	3
4	2

5.2. Desarrollo de la solución

Proin accumsan erat dignissim elit posuere, sed facilisis nisi semper. Cras suscipit sapien sed neque consectetur, ac pellentesque erat tristique. Interdum et malesuada fames ac ante ipsum primis in faucibus. Mauris lectus dolor, mattis eget ante sed, gravida blandit ante. Phasellus in bibendum lectus. Sed ut lectus hendrerit, convallis erat sed, feugiat nisl. Donec porttitor volutpat tortor, ac convallis mauris volutpat id. Nam a dictum ante. Maecenas ultrices eu elit in tincidunt. Morbi pellentesque varius ante nec scelerisque. Suspendisse potenti.

Nullam sem felis, consequat quis dui quis, rutrum gravida felis. Aenean et felis et dolor convallis elementum. Etiam neque lorem, ullamcorper vitae pulvinar id, molestie id elit. Morbi urna sapien, porttitor sit amet lorem sit amet, consequat convallis velit. Vestibulum auctor sapien ac ullamcorper volutpat. Nulla ut tellus at nibh luctus congue. Nam luctus feugiat feugiat.

Aenean et turpis nec quam volutpat lacinia eu eget nibh. Cras id velit laoreet, sollicitudin nulla nec, volutpat ipsum. Nulla sagittis lacus eget nibh porttitor venenatis. Phasellus eu sem in purus suscipit accumsan. Cras tristique justo ligula, ultrices consectetur nisl tempor vitae. Cras nibh diam, aliquet tincidunt lobortis non, eleifend vel quam. In eleifend pretium volutpat. Mauris vulputate dapibus nibh non rhoncus. Sed ut ipsum leo. Cras posuere congue purus id porta. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

5.3. Complejidad temporal

Ut ultrices nisi magna, a viverra nulla feugiat id. Vestibulum mollis magna nec augue volutpat auctor. Phasellus posuere leo in urna dapibus elementum vehicula nec odio. Nulla laoreet felis id est ultricies tempus. Nullam congue ipsum vel leo euismod laoreet. Donec gravida id justo ut lobortis. Curabitur id luctus neque. Fusce ullamcorper ligula quis nisl mattis pellentesque. Aenean dapibus odio turpis, vitae porta eros tempus quis. Praesent volutpat dui non molestie aliquam. Sed ullamcorper venenatis enim at aliquam. Integer nec orci leo.

Maecenas fermentum laoreet ultricies. Fusce ultrices lorem a lacus congue pellentesque. Cras vestibulum lectus sed purus imperdiet, sit amet tempus tellus vulputate. Nullam eu elementum nunc. Nulla eu magna commodo nunc tempor luctus. Nulla facilisi. Maecenas in quam laoreet leo sodales lacinia. Quisque

facilisis, enim sit amet vehicula adipiscing, massa sapien mollis ipsum, mattis fringilla diam lorem ut mauris. Nunc vel elit imperdiet, aliquet arcu non, posuere urna. Quisque sem enim, iaculis non nibh et, fermentum lobortis ligula. Aliquam purus libero, convallis a tincidunt sit amet, rutrum sed velit. Vestibulum luctus, sem ac vulputate accumsan, nibh arcu vulputate erat, ac mattis arcu arcu vitae odio. Phasellus et feugiat ante. Nam in placerat sem. Mauris tempor augue eu erat hendrerit, vel suscipit libero cursus.

5.4. Demostración de correctitud

Aliquam pulvinar orci eget quam pellentesque imperdiet. Cras auctor purus sit amet facilisis auctor. Nulla auctor orci ac ligula aliquet sagittis. Sed ultrices ligula mattis, tempor eros eu, convallis urna. Aenean tincidunt facilisis nibh id aliquam. Sed quis lectus sit amet enim adipiscing varius. Phasellus tempus feugiat elit, non volutpat metus commodo vel. Suspendisse bibendum ante sed faucibus sagittis. Vivamus et purus id massa dapibus faucibus. In at lobortis elit. Ut ut malesuada nunc. Phasellus adipiscing dui vitae lorem convallis ultricies. Cras condimentum pulvinar malesuada.

Sed semper malesuada libero eget egestas. Morbi tincidunt purus non justo posuere consequat. Donec lobortis lorem sit amet est commodo, a lobortis justo accumsan. Morbi sodales risus vel scelerisque consequat. Nullam sed nisl et mi euismod tempus ac id enim. Curabitur a rutrum lacus. Maecenas nisi enim, faucibus id volutpat sit amet, auctor adipiscing justo. Suspendisse mattis ullamcorper libero in ultricies. Proin eget metus sed odio pellentesque lobortis in in nibh. In sit amet ullamcorper tellus, eget feugiat nulla. Donec congue egestas neque ac mollis. Nunc hendrerit, justo et commodo blandit, est nibh aliquam nibh, a sodales odio ligula sed odio. Duis quis arcu tempus, egestas diam eget, facilisis lorem. Maecenas vitae nisi eget leo porta varius. Nunc eu quam tincidunt, varius tellus nec, cursus tellus. Donec in enim rhoncus ligula fermentum iaculis vel facilisis ante.

5.5. Experimentación

Nulla bibendum massa purus, quis pharetra lorem venenatis sed. Pellentesque eu imperdiet sem. Integer euismod urna non odio gravida, eu bibendum nisl posuere. Quisque faucibus rhoncus ipsum eget tempor. Cras nec nibh mauris. Integer volutpat mauris et tincidunt condimentum. Donec ante velit, elementum a euismod rutrum, faucibus nec arcu. Nulla et eros tempus, hendrerit mauris sit amet, dictum velit. Fusce at odio sed massa aliquam lobortis.

Proin in enim vitae diam semper euismod. Suspendisse malesuada venenatis dictum. In hac habitasse platea dictumst. Nunc orci elit, eleifend nec rhoncus vel, fringilla ut arcu. Duis in rutrum justo. Mauris eget consectetur elit, id tincidunt eros. Pellentesque ullamcorper ut dui quis imperdiet. Fusce egestas egestas diam eget imperdiet.

Mauris tincidunt egestas ipsum et laoreet. Donec non orci faucibus, lacinia neque in, pretium est. Vivamus pellentesque mollis massa, in ultrices justo imperdiet non. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In mattis urna lorem, sit amet dignissim lorem consectetur in. Ut porta, dolor lobortis convallis fermentum, massa dui varius dui, sed lacinia lacus mi at diam. Integer eget lacinia odio. Etiam fermentum velit sed nibh cursus, quis mattis odio condimentum. Nam in porttitor purus, sed ultricies metus. Nam eros velit, molestie ac mi porta, posuere dignissim lorem. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Pellentesque elementum nunc eu nisi luctus dictum. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Duis et dictum eros, a varius ante.

5.6. Conclusión

Praesent et rutrum tortor. Proin sit amet mi blandit, posuere justo eu, feugiat lacus. Nam luctus auctor condimentum. Sed porttitor auctor neque, eget condimentum arcu venenatis vel. In aliquam nibh ut quam vulputate, et euismod augue iaculis. Morbi blandit a dolor sit amet suscipit. Praesent consectetur purus ac elit sagittis, nec aliquam augue lobortis. Nam non sem leo. Pellentesque fringilla, enim et commodo lacinia, urna risus condimentum sapien, eget scelerisque eros enim sed ipsum. Pellentesque euismod consectetur cursus. Vivamus lobortis justo ante, cursus posuere nibh egestas in. Nunc tempor mollis

sapien, et dapibus erat. Etiam id sagittis mauris, laoreet fermentum turpis. Donec enim nunc, rutrum pretium quam non, malesuada interdum libero. Praesent sed sagittis nisl, sit amet eleifend lorem.

Vestibulum sit amet suscipit leo, et egestas velit. Quisque vitae volutpat felis, id venenatis mauris. Donec bibendum lacinia tristique. Curabitur quis dictum velit. Etiam eget arcu condimentum, placerat ante et, hendrerit turpis. Pellentesque nec molestie magna, id facilisis tortor. Nam mollis, nunc in pulvinar lobortis, mi nibh volutpat nibh, non fermentum nisi diam ac lorem.

Aenean odio lorem, congue id aliquam at, semper in ligula. Sed auctor neque eget est pulvinar suscipit. Aenean molestie lorem ut viverra volutpat. Nulla vitae augue nec lorem hendrerit eleifend. Pellentesque ut malesuada ante, a facilisis nulla. Nam eleifend vel dolor non mollis. Fusce consequat sit amet nulla nec consequat.

Praesent ultrices sem non elit semper, ut semper arcu tempor. Duis elementum eros sed massa facilisis, id convallis orci pharetra. Mauris euismod suscipit turpis. Nulla quis nisi eu mauris dapibus viverra. Donec nulla urna, eleifend ac odio sit amet, sodales pretium velit. Nunc hendrerit quam neque, aliquet condimentum nisl hendrerit eget. Etiam pharetra hendrerit nisl at sodales. Vivamus at tortor metus. Donec ultrices turpis libero, ac sollicitudin tortor tincidunt sed. Nam molestie dui dignissim, imperdiet elit ac, dignissim diam. Mauris iaculis nisl lectus, accumsan mattis felis iaculis in. Nam elit diam, tristique in ornare ac, condimentum sed odio. Praesent et gravida est. Donec nec justo id neque gravida iaculis ac eu orci. Vestibulum volutpat est in tellus volutpat tincidunt. Sed et est lacus.

Apéndices

A. Código fuente del problema 1

```
Salida Problema1::resolver(const Entrada &e)
{
  Salida s;

  vector<int> dias(e.dias);
  sort(dias.begin(), dias.end());

  s.dia_inicial = -1;
  s.max_cant_camiones = -1;
  int i = 0, j = 0, cant_camiones;
  for (; i < e.cant_dias; ++i)
  {
    if (0 < i && dias[i - 1] == dias[i]) continue;

    for (; (j < e.cant_dias) && (dias[j] - dias[i] < e.cant_dias_inspeccion); ++j)
      ;

    cant_camiones = j - i;
    if (s.max_cant_camiones < cant_camiones)
    {
      s.max_cant_camiones = cant_camiones;
      s.dia_inicial = dias[i];
    }
  }

  return s;
}
```

B. Código fuente del problema 2

```
struct Pieza
{
  Pieza(int indice, int perdida, int tiempo)
    : indice(indice), perdida(perdida), tiempo(tiempo) {}
  static bool comparar_piezas(const Pieza& lhs, const Pieza& rhs)
  {
    return lhs.tiempo * rhs.perdida < rhs.tiempo * lhs.perdida;
  }

  int indice;
  int perdida;
  int tiempo;
};

Salida Problema2::resolver(const Entrada &e)
{
  Salida s;
  s.piezas = vector<Pieza>(e.piezas);
  s.perdida_total = 0;

  sort(s.piezas.begin(), s.piezas.end(), Pieza::comparar_piezas);

  int acum_tiempo = 0;
  for (vector<Pieza>::const_iterator i = s.piezas.begin(); i != s.piezas.end(); ++i)
  {
    acum_tiempo += i->tiempo;
    s.perdida_total += acum_tiempo * i->perdida;
  }

  return s;
}
```

C. Código fuente del problema 3

```
struct Pieza
{
  Pieza(int indice, int sup, int izq, int der, int inf)
    : indice(indice), sup(sup), izq(izq), der(der), inf(inf) {}
  int indice;
  int sup;
  int izq;
  int der;
  int inf;
};

struct Tablero
{
  Tablero(int n, int m) : cantPiezas(0), casilleros(n, vector<int>(m, 0)) {}

  int cantPiezas;
  vector<vector<int> > casilleros;
};

Salida Problema3::resolver(const Entrada &e)
{
  Tablero solucionParcial(e.n, e.m), solucionOptima(e.n, e.m);
  vector<bool> piezaDisponible(e.n * e.m, true);

  resolverBacktracking(0, 0, solucionParcial, solucionOptima, piezaDisponible, e);

  Salida s;
  s.casilleros = solucionOptima.casilleros;
  return s;
}

void resolverBacktracking(int i, int j, Tablero &solucionParcial,
  Tablero &solucionOptima, vector<bool> &piezaDisponible, const Entrada &e)
{
  int sig_i, sig_j;
  calcularSiguientePos(sig_i, sig_j, i, j, e);

  for (int indicePieza = 1; indicePieza <= e.n * e.m; ++indicePieza)
  {
    if (!piezaDisponible[indicePieza - 1]) continue;
    if (!esCompatible(indicePieza, i, j, solucionParcial, e)) continue;

    colocarPieza(indicePieza, i, j, solucionParcial, piezaDisponible);

    if (solucionOptima.cantPiezas < solucionParcial.cantPiezas)
      solucionOptima = solucionParcial;

    if (llamarRecursivamente(sig_i, sig_j, solucionParcial, solucionOptima, piezaDisponible, e))
      resolverBacktracking(sig_i, sig_j, solucionParcial, solucionOptima, piezaDisponible, e);

    removerPieza(indicePieza, i, j, solucionParcial, piezaDisponible);
  }

  if (llamarRecursivamente(sig_i, sig_j, solucionParcial, solucionOptima, piezaDisponible, e))
    resolverBacktracking(sig_i, sig_j, solucionParcial, solucionOptima, piezaDisponible, e);
}
```

```
bool llamarRecursivamente(int i, int j, Tablero &solucionParcial,
  Tablero &solucionOptima, vector<bool> &piezaDisponible, const Entrada &e)
{
  if (solucionOptima.cantPiezas == e.n * e.m)
    return false;

  if (e.n <= i || e.m <= j)
    return false;

  int espaciosRestantes = e.n * e.m - (i * e.m + j);
  vector<int> coloresNecesarios(e.c, 0);
  int f, c, indice;
  f = (i == 0) ? 0 : i - 1;
  c = (i == 0) ? 0 : j;
  for (; f != i && c != j ; calcularSiguientePos(f, c, f, c, e))
  {
    indice = solucionParcial.casilleros[f][c];
    if (indice != 0)
      ++coloresNecesarios[e.piezas[indice - 1].inf - 1];
  }

  for (int h = 0; h < e.n * e.m; ++h)
    if (piezaDisponible[h])
      --coloresNecesarios[e.piezas[h - 1].sup - 1];

  for (int h = 0; h < e.c; ++h)
    if (coloresNecesarios[h] > 0)
      espaciosRestantes -= coloresNecesarios[h];

  if (solucionParcial.cantPiezas + espaciosRestantes <= solucionOptima.cantPiezas)
    return false;

  return true;
}

void calcularSiguientePos(int &sig_i, int &sig_j, int i, int j, const Entrada &e)
{
  sig_i = i + (j + 1) / e.m;
  sig_j = (j + 1) % e.m;
}

bool esCompatible(int indicePieza, int i, int j, Tablero &solucionParcial, const Entrada &e)
{
  const Pieza &pieza = e.piezas[indicePieza - 1];
  if (0 < j)
  {
    int indicePiezaIzq = solucionParcial.casilleros[i][j - 1];
    if (0 < indicePiezaIzq)
      if (pieza.izq != e.piezas[indicePiezaIzq - 1].der)
        return false;
  }
  if (0 < i)
  {
    int indicePiezaSup = solucionParcial.casilleros[i - 1][j];
    if (0 < indicePiezaSup)
      if (pieza.sup != e.piezas[indicePiezaSup - 1].inf)
        return false;
```

```
  }

  return true;
}

void colocarPieza(int indicePieza, int i, int j,
  Tablero &tablero, vector<bool> &piezaDisponible)
{
  tablero.casilleros[i][j] = indicePieza;
  ++tablero.cantPiezas;
  piezaDisponible[indicePieza - 1] = false;
}

void removerPieza(int indicePieza, int i, int j,
  Tablero &tablero, vector<bool> &piezaDisponible)
{
  tablero.casilleros[i][j] = 0;
  --tablero.cantPiezas;
  piezaDisponible[indicePieza - 1] = true;
}
```