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 utilizamos 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 de este trabajo práctico 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 cppreference[1], encontramos que su complejidad es $O(n*log(n))$ comparaciones, y a lo sumo $O(n*log(n))$ swaps. Con sólo esta informacion no podemos asegurar que el algoritmo en su totalidad tenga una complejidad temporal de $O(n*log(n))$ operaciones, por lo que buscamos que hace el algoritmo *std::sort* revisando el código de *algorithm.h*. Encontramos que, para casos con cantidad de elementos a ordenar menor a 64, hace un sort especial (el cual no nos interesa ya que queremos evaluar lo que pasa para n grande). En casos mas grandes, realiza *IntroSort*. *IntroSort* intenta ordenar usando *QuickSort*, si no lo resuelve en $n*log(n)$ pasos, usa HeapSort para garantizar $O(n*log(n))$ comparaciones.

En los algoritmos utilizados en el problema 1 y 2, guardamos los datos en *vector¡int¿* por lo que sabemos que la comparaciones y swaps son $O(1)$. Luego, en ambos algoritmos, usamos dicho algoritmo de sorting sobre el *vector<int>*. Por esto, cual podemos garantizar que ordenar los *vector<int>* tiene una complejidad de $O(n*log(n))$ operaciones.

2.3. Mediciones de performance

Para llevar a cabo mediciones de performance sobre las implementaciones desarrolladas, medimos el tiempo consumido para resolver instancias de sucesivos tamaños en función de un parámetro a definir según el caso. Procuramos 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 nos 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[2].

[1] http://en.cppreference.com/w/cpp/algorithm/sort
[2] 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 diferentes instancias del mismo tamaño), registramos como valor definitivo la mediana de la serie de valores. Escogimos este criterio en vez de, por ejemplo, tomar la media, ya que utilizar la mediana 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 3.3):

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 es $d \in \{d_1, d_2, ..., d_n\}$.*

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[3]:

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

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

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[4], fundamentado en la siguiente propiedad (demostración en la sección 3.3):

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.

```
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 i = 0 ∨ d_{i-1} ≠ d_i then                          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.

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, y no se ejecuta n veces 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.

Buscar el máximo tiene una complejidad $O(n)$. Por esto, la etapa de ordenamiento (la cual tiene una complejidad de $O(n*log(n))$) domina la complejidad del algoritmo, permitiendo dar la cota $O(n*log(n))$ para la solución.

[4] 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 propuesto, es necesario probar esencialmente dos afirmaciones mencionadas en la sección 3.2. En primer lugar, debe justificarse la decisión de cuáles candidatos se eligen como posibles fechas de inicio del período de contratación; o, formalmente, se debe demostrar la Propiedad 1. Luego, es importante argumentar por qué el ciclo del algoritmo computa efectivamente la cantidad de elementos de la lista contenidos en el intervalo asociado a cada candidato.

3.4.1. Demostración de la Propiedad 1

Dada una instancia $< D,\ n,\ l = [d_1, d_2, ..., d_n] >$, se quiere probar que siempre existe un intervalo óptimo $[d, d + D)$ (en el sentido de que comprende la mayor cantidad posible de elementos de la lista l) cuyo límite inferior es $d \in \{d_1, d_2, ..., d_n\}$.

Como cualquier valor d natural genera un intervalo $[d, d + D)$ válido, el conjunto de posibles soluciones es no vacío. Como además cualquier solución posible contiene entre 0 y n elementos de la lista, necesariamente existe solución óptima. Sea d' tal que $[d', d' + D)$ es un intervalo óptimo.

- Si $(\nexists\ i : 1 \leq i \leq n)\ d' \leq d_i$, entonces el intervalo $[d', d' + D)$ no contiene ningún elemento de la lista. Esto es absurdo porque cualquier intervalo $[d_i, d_i + D)$ contiene al menos un elemento, y sería mejor que el óptimo.
- De lo contrario, se puede definir d^* como el menor elemento de la lista que es mayor o igual a d'; es decir: $d^* = min\ \{\ d_i : 1 \leq i \leq n \wedge d' \leq d_i\ \}$.

  Luego, $\forall\ i : 1 \leq i \leq n$:

  - Como d^* es el menor elemento que cumple la propiedad de ser mayor o igual que d':

    $$d' \leq d_i \rightarrow d^* \leq d_i$$

  - Como $d' \leq d^*$ por definición, se desprende que $d' + D \leq d^* + D$, y por lo tanto:

    $$(d_i < d' + D) \rightarrow (d_i < d^* + D)$$

  Entonces, vale que todo elemento contenido en el intervalo óptimo también está contenido en el intervalo $[d^*, d^* + D)$. Como $d^* \in \{d_1, d_2, ..., d_n\}$, queda demostrada la propiedad.

3.4.2. Correctitud del ciclo

En la inicialización previa al ciclo, se configura $(i, j) \leftarrow (1, 1)$ y por lo tanto, vale trivialmente el invariante:

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

El cuerpo del ciclo se ejecuta únicamente para el primer elemento de la lista, y para todos aquellos que no sean iguales al inmediatamente anterior. Como la lista está ordenada, esto es equivalente a evaluar únicamente la primera aparición de cada valor dentro de la lista, ignorando sus posibles repeticiones posteriores.

El ciclo interno tiene la única función de incrementar el valor de j. La guarda del mismo es:

$$(j \leq n) \wedge (d_i \leq d_j < d_i + D) \equiv (j \leq n) \wedge d_j \in [d_i, d_i + D)$$

Es decir, j jamás se incrementa si el j-ésimo elemento no pertenece al intervalo $[d_i, d_i + D)$. Esto mantiene el invariante, evitando que algún elemento con índice menor a j no pertenezca al intervalo del candidato en evaluación. Como además j se incrementa tantas veces como sea posible sin romper el invariante, esto garantiza que al terminar el ciclo interno la variable tendrá su valor máximo (cuando $j = n + 1$ ó $d_j \notin [d_i, d_i + D)$).

Luego, d_i es el primer elemento de la lista contenido en $[d_i, d_i + D)$ por no tener predecesor o por ser mayor que el mismo. Además, d_{j-1} es el último elemento de la lista contenido en $[d_i, d_i + D)$ por no tener sucesor, o porque lo garantiza el invariante en caso contrario. De ello se desprende que existen $(j-1) - i + 1 = j - i$ elementos de la lista contenidos en el intervalo, por lo cual es el valor que se utiliza para actualizar el estado de la solucion parcial.

Por último, es necesario probar que el invariante se mantiene al incrementar i al final del ciclo. Esto es equivalente a demostrar la propiedad 2.

3.4.3. Demostración de la Propiedad 2

Si el invariante de ciclo es válido para el par (i, j), entonces también lo es para $(i + 1, j)$. Formalmente:

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

De la verdad del antecedente, y dado que los elementos d_i están ordenados al comenzar el ciclo, se deducen las siguientes propiedades para todo k' tal que $i + 1 \leq k' < j$:

1. $d_{i+1} \leq d_{k'}$ por estar ordenados, ya que $i + 1 \leq k'$
2. $d_i \leq d_{i+1}$ por estar ordenados
3. $d_i + D \leq d_{i+1} + D$ se deduce de 2
4. $d_{k'} < d_i + D$ por el antecedente, ya que $d_{k'} \in [d_i, d_i + D)$
5. $d_{k'} < d_{i+1} + D$ se deduce de 3 y 4

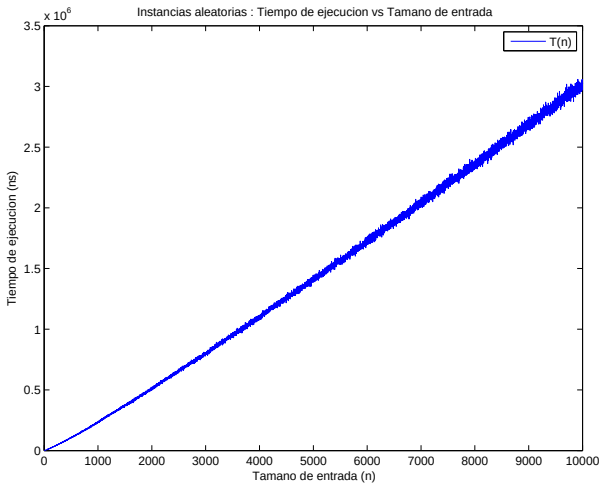
En base a 1 y 5, vale que $d_{k'} \in [d_{i+1}, d_{i+1} + D)$, y queda demostrada la propiedad.

3.5. Experimentación

Para experimentar, utilizamos una computadora con las siguientes características:

- Procesador:
- RAM:

La primer etapa de experimentación consistió en verificar empíricamente la cota de complejidad temporal obtenida teóricamente para el algoritmo completo. Estas primeras dos figuras nos permiten ver que efectivamente nuestro algoritmo es $O(n * log(n))$. Utilizamos entradas aleatorias, generadas pseudo-aleatoriamente que no necesariamente están ordenadas.



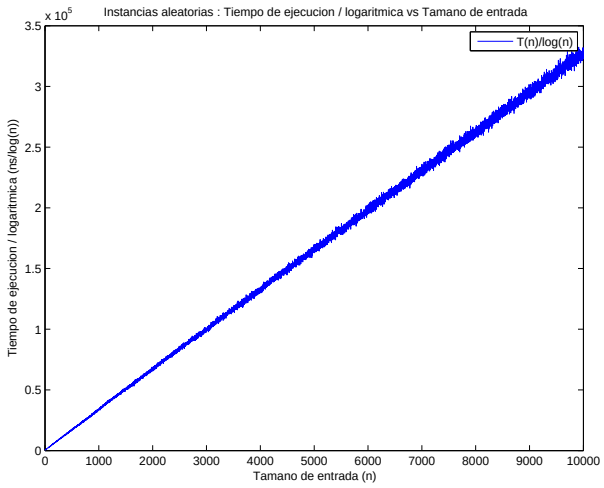


Figura 1: n ¡= 10000 *Figura 2: Idem, pero dividido por $log(n)$*

Como se puede observar en la figura 2, una vez que dividimos por *logn* a cada resultado obtenido por nuestras mediciones, éstos forman una recta. Es decir, $T(n)/log(n) \in O(n)$. Esto significa que la complejidad del algoritmo en su totalidad es de $O(n * log(n))$.

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 de C++, la experimentación subsiguiente se realizó sobre instancias donde la lista de entrada se encuentra ordenada, eliminando la etapa de ordenamiento. Esto permitió constatar si el ciclo final incurre efectivamente en un costo a lo sumo lineal, y poder medir el impacto de la ordenamiento en la complejidad temporal final.

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. Para hacer el gráfico mas interesante, comparamos casos donde todos los camiones llegan en días distintos contra casos donde actualiza seguido cuál es el día óptimo.

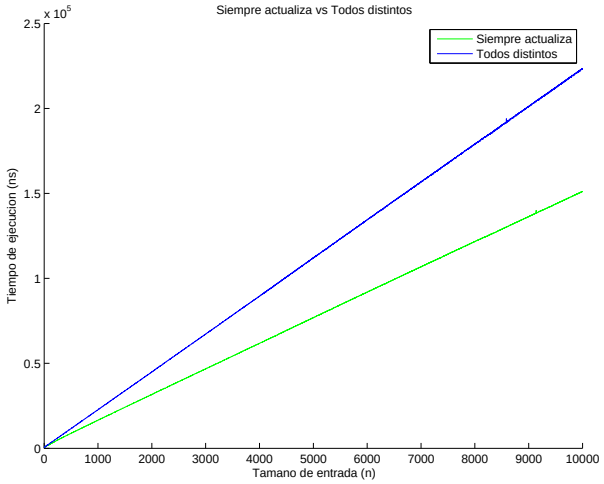


Figura 3: n ¡= 10000 *Figura 4: ACA PONEMOS ALGO???*

Curiosamente, "siempre actualiza" es mejor que "Todos distintos". Investigamos y llegamos a la conclusión que esto se debe a como creamos las instancias de entrada. Las creamos siguiendo la siguiente secuencia:

```
1,2,2,3,3,3,4,4,4,4 ...
```

Como tiene números repetidos, el algoritmo saltea varias iteraciones en la parte del siguiente *for*.

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

Por esta razón, termina siendo mejor que el de "Todos distintos".

3.6. Conclusión

Para concluir, nos parece importante destacar que la complejidad temporal de nuestra 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 debido a que los algoritmos de sorting son, a lo sumo, $O(n * log(n))$. Sin embargo, cabe destacar que en el caso en que los datos están ordenados al momento de leer, podemos mejorar la complejidad temporal de $O(n * log(n))$ a $O(n)$.

Por otro lado, sin contar la etapa de ordenamiento, realizamos a lo sumo $O(n)$ operaciones, ya que podríamos tener que recorrer el vector por lo menos dos veces: una para guardar los datos y otra para

saber cuál es el día óptimo para colocar al inspector. Por esto, podemos asegurar que por más que los datos vengan ordenados no vamos a poder mejorar la complejidad temporal de $O(n)$.

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

Lo primero que hicimos fue hallar que el orden óptimo en el que se deben armar las piezaspara minimizar la pérdida 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.

Lo que nuestro algoritmo hace es crear un vector y colocar 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)
```

Y el pseudo código de Calcular_perdida(V) es el siguiente:

```
procedure Calcular_perdida(V)
    total ← 0
```

$tiempo \leftarrow 0$
$i \leftarrow 0$
while $i < n$ **do**
  $tiempo \leftarrow tiempo + V[i].tiempo$
  $total \leftarrow total + V[i].perdida * tiempo$
  $i \leftarrow i + 1$
return total

4.3. Complejidad temporal

Nuestro algoritmo tiene una complejidad de $O(n * logn)$ 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 $O(n * logn)$.

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. Para cada calcular la pérdida de cada pieza, tenemos que tener en cuenta el tiempo que pasó hasta ese momento sumar el tiempo que tarda en hacer la pieza (una suma), calcular la pérdida y sumarlo a la pérdida hasta ese momento (multiplicación y suma). Estas operaciones son $O(1)$ ya que trabajamos con ints.

Luego, por álgebra de ordenes O(1) + O(n) + O(n.log n) + O(1) + O(n) = $O(n * logn)$, como queríamos demostrar.

4.4. Demostración de correctitud

Para probar que nuestro algoritmo es correcto y que además es óptimo, lo que debemos demostrar es que el orden en que se fabrican las piezas que nosotros en principio supusimos como óptimo es realmente óptimo. Y lo otro que debemos demostrar es que el algoritmo Calcular_perdida calcule correctamente la perdida total del joyero por fabricar las joyas en el orden utilizado.

4.4.1. Demostración de la optimicidad del orden propuesto

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) $\Rightarrow$ 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 permutación 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 óptimo, se que C(R) $\leq$ C(R')

También 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 permutación 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 óptimo.

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' óptimo

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

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:

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

□

4.4.2. Demostración de correctitud del algoritmo Calcular_perdida

Para empezar recordemos el pseudocódigo de este algoritmo:

```
procedure CALCULAR_PERDIDA(V)
    total ← 0
    tiempo ← 0
    i ← 0
    while i < n do
        tiempo ← tiempo + V[i].tiempo
        total ← total + V[i].perdida * tiempo
        i ← i + 1
    return total
```

Supongamos que tenemos las piezas en el orden $V = [pieza_1..pieza_n]$, la pérdida total ocasionada por fabricar las piezas en ese orden es la siguiente:

$\sum_{i=1}^{n} (p_i \sum_{j=1}^{i} t_j)$

Donde p_i es la pérdida por día de la $i-esima$ joya y t_i es la cantidad de días que la $i-esima$ joya tarda en ser armada.

Lo que queremos demostrar es que este algoritmo cumple el siguiente invariante:

P(n) : en la $k-esima$ iteración, vale que total $= \sum_{i=1}^{k} (p_i \sum_{j=1}^{i} t_j)$

Vamos a demostrar esto por inducción.

- Caso base: Vale P(1) En la primera iteración del algoritmo vale que:

  $total = V[1].perdida * V[1].tiempo$

  Esto significa que total $= p_1 * t_1 = \sum_{i=1}^{1} (p_i \sum_{j=1}^{i} t_j)$

  Por lo tanto vale P(1).

- Paso Inductivo:

  Hipótesis Inductiva: P(n)

  Queremos ver que $P(n) \Rightarrow P(n+1)$

  Sabemos que en la iteración $n+1-esima$, la variable total tiene el siguiente valor:

  $total = total + V[n+1].perdida * tiempo$

  Por un lado sabemos que en nuestro algoritmo la variable *tiempo* empieza valiendo cero y va acumulando en cada iteración del ciclo el tiempo de cada pieza que vamos recorriendo. Así que podemos decir que en la $n+1-esima$ iteración la variable *tiempo* es igual a $\sum_{i=1}^{n+1} t_i$ (esto se podría demostrar por inducción).

  Asumiendo esto podemos ver que la variable total tiene el siguiente valor

  $total = total + V[n+1].perdida * tiempo = total + p_{n+1} * \sum_{j=1}^{n+1} t_j$

  Y por hipótesis inductiva:

  $total = \sum_{i=1}^{n} (p_i \sum_{j=1}^{i} t_j) + p_{n+1} * \sum_{j=1}^{n+1} t_j = \sum_{i=1}^{n+1} (p_i \sum_{j=1}^{i} t_j)$

□

4.5. Experimentación

Para experimentar, utilizamos una computadora con las siguientes características:

- Procesador:
- RAM:

Al igual que en el problema 1, la primer etapa de experimentación consistió en verificar empíricamente la cota de complejidad temporal obtenida teóricamente para el algoritmo completo. Estas primeras dos figuras nos permiten ver que efectivamente nuestro algoritmo es $O(n * log(n))$. Utilizamos entradas aleatorias, generadas pseudo-aleatoriamente que no necesariamente están ordenadas.

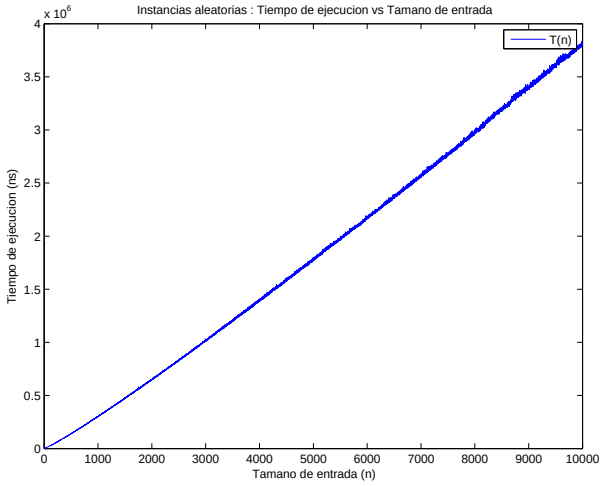


Figura 1: Caso $n = 10000$

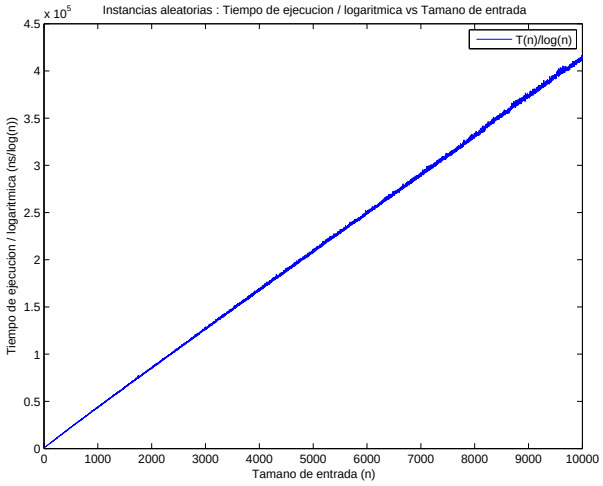


Figura 2: Idem, pero dividido por $log(n)$

Como se puede observar en la figura 2, una vez que dividimos por $log(n)$ a cada resultado obtenido por nuestras mediciones, éstos forman una recta. Es decir, $T(n)/log(n) \in O(n)$. Esto significa que la complejidad del algoritmo en su totalidad es de $O(n * log(n))$.

Según nuestro análisis, la complejidad temporal de la solución es dominada por la etapa de ordenamiento una vez más. Dado que el algoritmo utilizado para este fin es el mismo que el utilizado en el problema 1, de nuevo la experimentación subsiguiente se realizó sobre instancias donde la lista de entrada se encuentra ordenada, eliminando la etapa de ordenamiento. Esto permitió constatar si el resto del algoritmo 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, utilizamos entradas en las cuales las joyas estén ordenadas según el coeficiente anteriormente mencionado, y por esto no haga falta realizar el sorting.

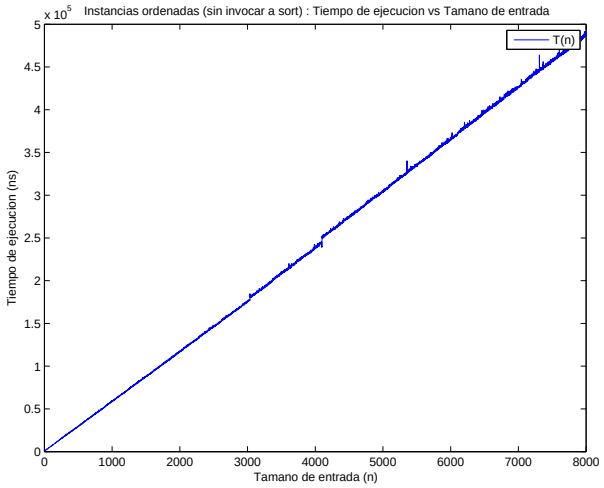


Figura 3: Caso $n = 8000$

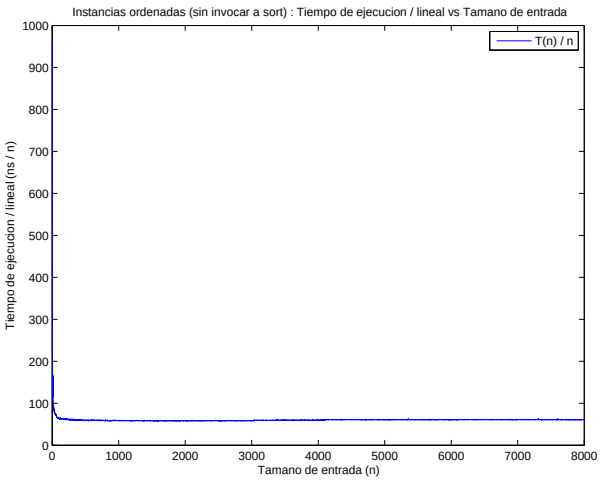


Figura 4: Idem, pero dividido por n

Como se puede observar en la figura 4, nuevamente al dividir por n a cada resultado obtenido por nuestras mediciones, éstos se agrupan en una constante. Es decir, $T(n)/n \in O(1)$. Esto significa que la complejidad del algoritmo en su totalidad es de $O(n)$.

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.

En este problema tuvimos dificultades similares a las del ejercicio 1 en cuanto a determinar cuál es el peor y el mejor caso del algoritmo, ya que usamos un algoritmo de ordenamiento de las librerías de C++ y como éste usa en general el algoritmo de *QuickSort*(en realidad usa *IntroSort* ya explicado anteriormente), depende del azar el hecho de que ordenar un vector tarde más o menos comparaciones (siempre manteniéndose en el orden de complejidad temporal de $O(n * logn)$).

Pero además en este ejercicio que un caso sea peor o mejor depende exclusivamente de cuanto tarde el algoritmo en ordenar la entrada, ya que luego de esto sólo recorre el vector para calcular la pérdida total y devuelve dicho vector, esto siempre toma la misma cantidad de pasos, por lo tanto no nos es posible determinar *a priori* si un caso va a resultar mejor o peor para este algoritmo.

4.6. Conclusión

Para concluir, nos parece importante destacar que la complejidad temporal de nuestra 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 debido a que los algoritmos de sorting son, a lo sumo, $O(n * log(n))$. 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 $O(n)$.

Por otro lado, sin contar la etapa de ordenamiento, realizamos a lo sumo $O(n)$ operaciones, ya que recorremos el vector por lo menos dos veces: una para guardar los datos y otra para saber cuál es la pérdida total. Por esto, podemos asegurar que por más que los datos vengan ordenados no vamos a poder mejorar la complejidad temporal de $O(n)$.

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 solución (que pondría todas las fichas en el tablero) de esta instancia del problema sería:

1	3
4	2

5.2. Desarrollo de la solución

Para resolver el problema implementamos el algortimo presentado debajo. Cabe aclarar que en ResolverBacktracking, SolucionOptima y SolucionParcial son **in/out**, es decir, pasados por referencia y por eso esta función no contiene un **return**

```
Tipo de dato Pieza es Tupla ⟨ id : entero, inferior : entero, superior : entero, izquierda : entero, derecha
: entero ⟩
Tipo de dato Tablero es Tupla ⟨ cantFichas : entero, casillas : Matriz⟨Pieza⟩ ⟩
procedure RompecolorES(n, m, c, ⟨p_1, ..., p_{n*m}⟩)
    SolucionParcial, SolucionOptima← nuevo Tablero
    ResolverBacktracking(SolucionOptima,SolucionParcial,0,0,...)
    return SolucionOptima


procedure ResolverBacktracking(SolucionOptima,SolucionParcial,i,j,...)
    for Cada Ficha do
        if EstaDisponible ∧ esCompatible then
            Agrego ficha a SolucionParcial
            if SolucionParcial mejor que SolucionOptima then
                SolucionOptima = SolucionParcial
            if !Podo then
                ResolverBacktracking en siguiente posicion
            Remuevo ficha
    Agrego ficha blanca a SolucionParcial
    ResolverBacktracking en siguiente posicion
```

Donde:

• EstaDisponible devuelve si la ficha no fue usada todavia (verdadero), o si ya fue agregada al tablero (falso).

• esCompatible devuelve verdadero si al agregar la ficha en la posicion en la cual se esta trabajando en el tablero SolucionParcial, no se genera ninguna contradiccion.

• Podo devuelve verdadero si a travez de alguna de nuetras podas podemos asegurar que en esa rama no se encuentra ninguna solucion mejor a SolucionOptima.

Implementamos tres posibles funciones Podo(), la primera es la mas simple que solo devuelve falso cuando se llega al final del tablero. Si bien no poda, la utilizamos como corte en el caso en que encuentre una solución donde las fichas llenan todo el tablero, por lo que seguir recorriendo es inútil.

La segunda opción se fija si la cantidad de fichas que puse en el tablero más la cantidad de lugares del tablero que todavía no recorrí, es menor que la cantidad de fichas de SolucionOptima. Si es menor sabemos que no hay ninguna forma que por esa rama se llegue a una solución mejor que SolucionOptima y por esto, la descartamos.

La tercer poda que realizamos incluye a la anterior pero realiza algo extra: En las posiciones que todavía no recorrimos pero ya sabemos que tienen una restricción (como por ejemplo, que necesitan el color 1 en superior), queremos saber cuantas fichas tenemos disponible para esos lugares. Para esto contamos para cada color, cuantos de estos lugares necesitan una ficha con ese color en la posición superior. Luego recorremos las fichas disponibles y restamos 1 al nuemero calculado anteriormente para ese color.

Si sumamos todos los contadores de color que no nos quedaron negativos, conseguimos el numero de estos casilleros a los que no le vamos a poder agregar fichas. Luego, el numero de fichas posibles calculado en la opción 2 le restamos lo que acabamos de calcular obteniendo una cota mas ajustada de la cantidad de fichas que puede llegar a tener esta rama del árbol.

Estas podas nos ayudan a descartar casos para no tener que observarlos. En la etapa de experimentación vamos a evaluar si el uso de estas podas es favorable o no. Esto se debe a que por más que se pode una rama (o algunas ramas), la poda utilizada puede llegar a ser más costosa que recorrer dicha rama (o ramas).

5.3. Complejidad temporal

Para calcular la complejidad vamos a tener en cuenta el peor caso, en el cual

$T(n,n) = T(n-1,n) + n * T(n-1,n-1)$

$T(n,n) = T(1,n) + n * n * T(1,n-1) + \binom{n}{2} * \binom{n}{2} * T(1,n-2) + \binom{n}{3}^2 * T(1,n-3) + \ldots + \binom{n}{i}^2 * T(1,n-i) + \ldots + T(1,1)$

Si todo esta bien:

$T(n,n) = \sum_{j=0}^{n} T(1,j) * \binom{n}{j}^2$

Donde $T(1,j) \in O(j)$

5.4. Demostración de correctitud

Para este problema, decidimos enfocar la demostración de correctitud de otra manera. Vamos a probar que nuestro algoritmo, sin contar las podas, recorre todas las soluciones posibles. Luego, vamos a probar que las podas utilizadas no remueven soluciones mejores que la mejor solución que encontramos hasta ese momento.

Vamos a utilizar el mismo pseudo-código que se encuentra en la sección de Desarrollo. Para facilitar la lectura, decidimos escribirlo de nuevo en esta sección. Cabe recordar que SolucionOptima y SolucionParcial son **in/out**, es decir, pasados por referencia y por eso esta función no contiene un **return**

```
procedure ResolverBacktracking(SolucionOptima,SolucionParcial,i,j,...)
    for Cada Ficha do
        if EstaDisponible ∧ esCompatible then
            Agrego ficha a SolucionParcial
            if SolucionParcial mejor que SolucionOptima then
                SolucionOptima = SolucionParcial
            if !Podo then       // Esta guarda es siempre true para la primer parte de esta demostración
                ResolverBacktracking en siguiente posicion
            Remuevo ficha
```

Agrego ficha blanca a SolucionParcial
ResolverBacktracking en siguiente posicion

En un primer lugar vamos a demostrar que si no utilizamos ninguna poda recorremos todas las soluciones, y por consiguiente vamos a recorrer una solución óptima.

Nuestro algoritmo llamado ResolverBacktracking, revisa si es posible colocar una pieza en la casilla de la fila i y la columna j (De ahora en mas casilla$_{i,j}$). Para hacer esto, revisa una a una las piezas para observar si es posible colocarla. En el caso en que pueda colocarla, revisa si la solución parcial (la solución que tenía más la pieza colocada) que encontró es mejor que la óptima que había encontrado hasta ese momento. Si es mejor, significa que encontró una mejor solución, y por eso actualiza SolucionOptima.

Sin embargo, puede ser que no sea posible colocar alguna de las piezas que me faltan en la casilla$_{i,j}$ debido a restricciones acerca de como se colocaron las otras piezas. En cuyo caso, colocamos una pieza blanca (es decir, ninguna pieza o un espacio en blanco) y seguimos con la siguiente casilla.

Realizamos este procedimiento para cada casilla, y para cada casilla recorremos para cada pieza. De esta manera, estamos efectivamente recorriendo todos las combinaciones de disposiciones de piezas posibles que sean compatibles.

Ahora vamos a pasar a demostrar que ninguna de nuestras podas, descartan soluciones mejores a las que ya encontramos.

La primera es la más simple que sólo devuelve falso cuando se llega al final del tablero. Si bien no poda en el sentido estricto, la utilizamos como corte en el caso en que encuentre una solución donde las fichas llenan todo el tablero, por lo que seguir recorriendo es inútil.

Es simple demostrar que esta poda no descarta soluciones mejores a la ya obtenida, ya que ésta sólo corta cuando llega al final del tablero, es decir, cuando coloca todas las piezas. Como logró colocar todas las piezas, no existe otra disposición que coloque mas piezas que llenar el tablero. Por esto, no poda soluciones mejores a la que ya encontró (en este caso, llenar el tablero) ya que no existe una mejor solución que rellenar todo el tablero.

La segunda opción se fija si la cantidad de fichas que puso en el tablero mas la cantidad de lugares del tablero que todavía no recorrió, es menor que la cantidad de fichas de SolucionOptima.

Esta poda también es simple de demostrar. Supongo que tenemos un tablero t_1 donde logramos colocar k_1 piezas y tenemos k_2 casillas para colocar piezas. Por esto, la mayor cantidad de piezas que podríamos colocar es $k_3 = k_1 + k_2$. Si nuestra solución óptima tiene k_4, donde $k_4 \geq k_3$, entonces es imposible que el tablero t_1 (ni todas sus posibilidades de colocar piezas en las casillas restantes) puedan tener más piezas que nuestra solución óptima. Por esto, esta poda no descarta alguna mejor solución que la óptima hasta ese momento.

Finalmente, la tercer poda que utilizamos incluye a la anterior pero realiza algo extra: Queremos saber cuántas fichas tenemos disponibles en las posiciones que todavía no recorrimos pero que ya sabemos que tienen una restricción (como por ejemplo, que necesitan el color 1 en lado superior).

Esta última poda se prueba de la misma manera que la anterior. Tiene un extra que es que realiza una cota menos burda acerca de la cantidad de casillas que es posible llenar. Podemos demostrar de una manera similar que no poda soluciones mejores a la óptima encontrada hasta ese momento.

Supongo que tenemos un tablero t_1 donde logramos colocar k_1 piezas y tenemos k_2 casillas para colocar piezas, pero sin embargo solo puedo colocar a lo sumo k_5 piezas debido a las restricciones. Por esto, ahora la mayor cantidad de piezas que podríamos colocar no es $k_3 = k_1 + k_2$, sino que es $k_6 = k_1 + k_5$. Si nuestra solución óptima tiene k_4, donde $k_4 \geq k_6$, entonces es imposible que el tablero t_1 (ni todas sus posibilidades de colocar piezas en las casillas restantes) pueda tener más piezas que nuestra solución óptima. Por esto, esta poda no poda alguna mejor solución que la óptima hasta ese momento.

De esta manera, probamos que nuestro tablero recorre todas las soluciones, y las podas que utiliza no podan soluciones mejores a la solución óptima encontrada hasta ese momento.

5.5. Experimentación

Nota: Como vamos a hablar mucho de las distintas podas vamos a denominarlas de la siguiente manera:

- Primer poda: Terminar cuando lleno el tablero.
- Segunda poda: Si aunque ponga piezas en los casilleros que me queda revisar, no llego a alcanzar la cantidad de piezas de la solución óptima encontrada a ese momento, corto.
- Tercer poda: Similar a la segunda poda, pero además revisa hasta m casillas para adelante teniendo en cuenta las restricciones ya existentes en el tablero.

BLA BLA BLA

GRAFICOS

BLA BLA BLA

En el gráfico ACA HAY QUE PONER EL NUMERO CORRECTO, se puede observar claramente que la tercer poda logra podar mas casos, pero no logra compensar el costo adicional de $O(m)$. Dicha poda podría servir en otro problema en el que cual cada llamada recursiva tenga un mayor costo.

5.6. Conclusión

Para concluir, nos parece importante recalcar el impacto de las podas en este problema. Es simple realizar un algoritmo que recorra todas las soluciones y así encuentre la óptima pero esto incurriría en un algoritmo muy tosco y lento. Las podas nos ayudan a determinar si seguir por una rama del árbol de soluciones del problema nos llevará a una solución que valga la pena (con que valga la pena nos referimos a que la solución a la que nos lleve sea mejor que la que ya tengamos) y de no ser así podemos descartar dicha rama ahorrándonos varios casos.

Habiendo dicho eso, por mas que supongamos que no hay problemas en la implementación de las podas (es decir, no haya *bugs*), puede ser que utilizar una cierta poda puede no llegar a ser beneficioso. Puede llegar a ocurrir que, si bien una poda nos ayude a descartar muchas ramas del árbol de soluciones posibles, dicha poda sea muy ineficiente en términos de complejidad temporal y por consiguiente, el algoritmo final termine siendo más lento que una solución mas *naïf*.

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;
}
```