DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

11 de abril de 2014 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Chapresto, Matías	201/12	matiaschapresto@gmail.com
Dato, Nicolás	676/12	nico_dato@hotmail.com
Fattori, Ezequiel	280/11	ezequieltori@hotmail.com
Vileriño, Silvio	106/12	svilerino@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Introducción

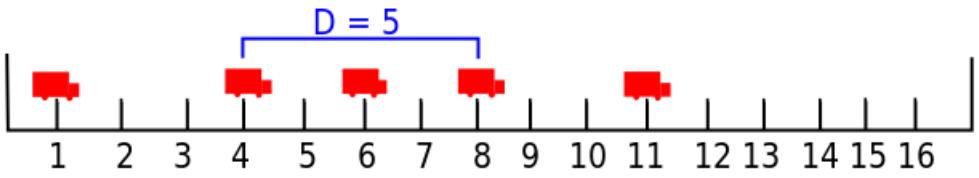
El objetivo de este informe es explicar los algoritmos utilizados para la resolución de los ejercicios presentados, dando ejemplos de como actuarían sobre ciertas entradas y cuales serían sus salidas asociadas. Asimismo se demostró la correctitud de los mismos, justificando porque resuelven el problema presentado en el ejercicio. Adicionalmente, se analizó la complejidad teórica asintótica de los algoritmos desarrollados, para luego contrastarse con las mediciones empíricas realizadas sobre un conjunto de tests aleatorios y algunos específicos como la determinación del mejor y peor caso. Luego se muestran los resultados obtenidos de las pruebas sobre los algoritmos y sus conclusiónes asociadas. Finalmente, hay dos apéndices en los cuales se adjuntan las partes importantes del código fuente y se explica como compilar, correr y ejecutar tests.

2. Ejercicio 1

2.1. Descripción del problema

En un puesto de control, se posee información sobre los días de llegada de un número n de camiones. Cada camión pasa una vez, y es posible que en un día determinado puedan pasar varios de ellos. Sabiendo esto, se quiere contratar a un inspector que los revise. Este inspector sólo puede ser contratado por una cantidad fija de D días consecutivos, por lo cual, para sacar mejor provecho del presupuesto, se desea contratar al inspector en un período en el cual pase la mayor cantidad de camiones. Se pide disenar e implementar un algoritmo que resuelva este problema en tiempo estrictamente menor a $O(n^2)$.

Consideremos el siguiente caso como ejemplo $\{n = 5$; días = 8, 6, 4, 11, 1; $D = 5\}$



En este caso el período del inspector que maximiza la cantidad de camiones es el [4,8] ya que en ningún otro intervalo llegan más de 3 camiones, por lo que esa debería ser el resultado de nuestro algoritmo.

2.2. Funcionamiento del algoritmo

En esta sección daremos una idea informal acerca del funcionamiento de nuestro algoritmo para luego dar paso a las demostraciones formales.

2.2.1. Preliminares

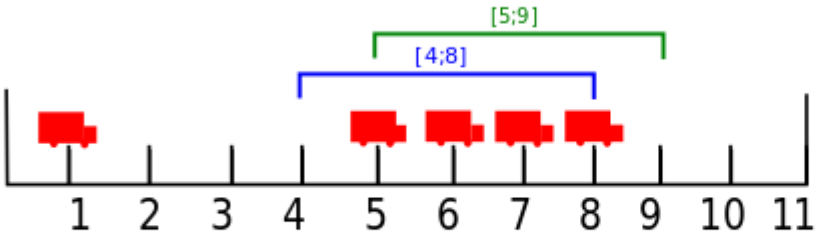
La entrada del algoritmo es de la forma D n $d1$ $d2$ . . .dn, donde D son los días de contratación del experto, n la cantidad de camiones, y $d1$, $d2$... dn los días en los que pasa cada camión, no necesariamente en orden cronológico.

Paso 1: El algoritmo ordena los días en los que vienen los camiones de manera de tener un vector calendario. Este vector de enteros va a representar una sucesión de días en los que viene al menos un camión. En caso de venir más de un camión el mismo día, ese día aparecerá tantas veces como camiones vengan.

Paso 2: Una vez hecho esto, el algoritmo transforma este vector de días en un vector de tuplas<día, ♯camiones>, donde ♯camiones es la suma acumulada de camiones que llegaron desde el día 1 hasta ese día inclusive. De esta forma, se agrupan los días repetidos, posibilitando las búsquedas binarias en el vector, y al tener las sumas acumuladas hasta cada día, se puede obtener en tiempo constante la cantidad de camiones que llegaron en un intervalo $[a..b]$, restando la suma acumulada del día anterior a a a la del día b.

2.2.2. Ciclo Principal

Antes de seguir, consideremos otro caso posible del problema, en el que hay varios posibles intervalos óptimos que maximicen la cantidad de camiones. $\{n = 5$, días = 5,1,8,7,6; $D = 5\}$



Vemos que en este caso hay dos intervalos que maximizan la cantidad de camiones (4): [4,8]; [5,9], llamḿosles a y b.

Según el enunciado en caso de haber varias soluciones óptimas debemos devolver cualquiera de ellas. El funcionamiento de nuestro algoritmo hace que devuelva una particular. Podemos ver que en el intervalo a, el primer camión llega el día 5. Con lo cual el experto es contratado el día 4 innecesariamente, ya que no llega ningún camión ese día. Si contratáramos al experto el día 5, podría revisar todos los mismos camiones que revisaba el intervalo a y tendría un día más (el día 9) para revisar camiones (aunque en este caso no venga ninguno más).

Siguiendo la línea de pensamiento, cualquier intervalo óptimo que maximice la cantidad de camiones, puede ser desplazado hacia la derecha de modo que el primer día en que es contratado el experto sea el primer día en el que viene un camión en el intervalo original. Todos los camiones que abarcaba el intervalo original van a ser abarcados por el intervalo desplazado, y todos los días del principio que no eran aprovechados pasan a ser parte del final del intervalo, dando la posibilidad de poder revisar más camiones. Por lo cual nuestro algoritmo ni siquiera consideraría la evaluación del intervalo a y devolvería como resultado el intervalo b.

Esto es clave para entender el funcionamiento y por lo tanto la correctitud de nuestro algoritmo, ya que éste se limita a considerar el subconjunto de intervalos de longitud D que comienzan en un día en el que viene un camión. El cardinal de este subconjunto, y por lo tanto la cantidad de intervalos que tiene que evaluar es n.

Paso 3: La evaluación de cada intervalo se produce de la siguiente forma: En cada iteración, se toma una de las tuplas<dia, $\sharp camiones >$, llamémosle i. La primer componente de la tupla (i_1) es el día en el que comienza el intervalo. A ese día, se le suma D- 1. Este número es último día que el experto revisa, llamémosle f.

Paso 4: A continuación se realiza una búsqueda binaria en el vector para poder obtener la suma acumulada de camiones hasta el día f. En caso de no existir este día en el vector se toma la tupla cuyo día es el menor más cercano a f (Ya que no vienen camiones entre este día y f, la suma acumulada de camiones hasta el día menor más cercano es igual a la de f).

Paso 5: Una vez obtenida la segunda tupla, se realiza la resta de: la suma acumulada hasta f - la suma acumulada hasta el día i -1. Esto nos da como resultado la cantidad de camiones que llegan en el intervalo [i_1 .. f], o en otras palabras la cantidad de camiones que revisa el experto si es contratado desde el dia i_1 hasta el f. Lo cual es lo que tenemos que maximizar.

Paso 6: Este resultado se guarda en una variable que se va reemplazando a medida que se itera por cada una de las n tuplas, de forma de encontrar el máximo.

A continuación se muestra un pseudocódigo de lo anteriormente explicado.

```
1: procedure MAXIMIZARCAMIONES(int diasInspector, int[] tablaDiaCantidad, int ♯camiones)
2:     Ordenar(tablaDiaCantidad)                                          ▷ Paso 1
3:     AgruparYAcumular(tablaDiaCantidad)                                 ▷ Paso 2
4:     int maximo ← −1
5:     int diaAContratar ← −1
6:     for i ← 0; i < tablaDiaCantidad.size; i++ do                       ▷ Ciclo Principal
7:         tupla < int, int > inicio ← tablaDiaCantidad[i]                ▷ Paso 3
8:         int diaFinal ← inicio.first + diasInspector − 1
9:         tupla < int, int > final ← tDC[BusquedaBinaria(tDC, i, diaFinal, tDC.size())]  ▷ Paso 4
10:        int♯camionesDelIntervalo ← final.second − tablaDiaCantidad[i − 1].second       ▷ Paso 5
11:        if ♯camionesDelIntervalo > maximo then                         ▷ Paso 6
12:            maximo ← camionesDelIntervalo
13:            diaAContratar ← inicio.first
14:        end if
15:    end for
16:    return < diaAContratar, maximo >
17: end procedure
```

Nota: en el paso 4 abreviamos tablaDiaCantidad por tDC para una mejor lectura.
Se puede ver una implementación del algoritmo en la sección 7.1.

2.3. Demostración de correctitud

Definamos algunos conceptos:

El algoritmo recibe n naturales, con posibilidad de repetidos, que representan los días en los que llegan los camiones. Podemos representar formalmente esto con un multiconjunto de naturales, implementado con un vector al que llamaremos *calendario*. Por cada día, vienen tantos camiones como cantidad de repeticiones tenga en el multiconjunto.

Podemos definir formalmente las funciones "$cantCamionesDia$", que dado un vector calendario y un dia devuelve la cantidad de camiones que llega ese día y "$cantCamionesIntervalo$" que dado un calendario y dos días devuelve la cantidad de camiones que llegan entre esos días.

$$cantCamionesDia(calendario, dia) = cantRepeticiones(calendario, dia)$$

Donde cantRepeticiones es una función canónica de los multiconjuntos que dado un natural y un multiconjunto devuelve su cantidad de apariciones en él.

Definimos intervalo a un par de naturales $< x, y >$ con $x \leq y$.

$$cantCamionesIntervalo(calendario, intervalo) = \sum_{i=intervalo_1}^{intervalo_2} (cantRepeticiones(calendario, i))$$

Esto representa la 'cantidad de camiones que llega en un intervalo'. Finalmente definamos la función principal $maximizarCamiones$, que es la formalización matemática del algoritmo. Los parámetros de la función son: el calendario y los dás que se contrata el inspector. Llamemos I_D a todos los intervalos posibles entre 1 y $max(calendario)$ de longitud D. Nótese que al ser naturales este es un conjunto finito.

$$maximizarCamiones(calendario, D) = i \in I_D / \forall x in I_D, cantCamionesIntervalo(calendario, x) \leq cantCamionesIntervalo(i)$$

2.3.1. Correctitud de la acumulación

Nuestro algoritmo manipula los días entrada, primero ordenándola (la correctitud de esto está asegurada por la documentación del lenguaje) y luego la transforma en un vector de tuplas, cuyas primeras componentes son los días anteriormente mencionados y las segundas componentes son la suma acumulada de camiones que llegan desde el día 1 hasta el día respectivo de su primera componente. Necesitamos probar que esta acumulación se realiza de manera correcta.

Definamos formalmente la función *acumDias* que toma un calendario y un día d y devuelve la suma acumulada de camiones que llegaron desde el día 1 hasta d.

$$acumDias(calendario, dia) = \sum_{i=1}^{dia}(cantCamionesDia(calendario, i))$$

Definido esto, vemos que podemos calcular la cantidad de camiones de un intervalo $< x, y >$ haciendo la resta:

$$acumDias(calendario, y) - acumDias(calendario, x - 1)$$

Probemos esto. Por definición de *acumDias*:

$$acumDias(calendario, y) - acumDias(calendario, x - 1) = \sum_{i=1}^{y}(cantCamionesDia(calendario, i) - \sum_{j=1}^{x-1}(cantCamionesDia(calendario, j)$$

Dado que es un intervalo, $x \leq$ entonces:

$$\sum_{i=1}^{y}(cantCamionesDia(calendario, i) = \sum_{i=1}^{x-1}(cantCamionesDia(calendario, i) + \sum_{i=x}^{y}(cantCamionesDia(calendario, i)$$

Si reeemplazamos en la anterior ecuación, nos queda que:

$$acumDias(calendario, y) - acumDias(calendario, x - 1) = \sum_{i=1}^{x-1}(cantCamionesDia(calendario, i) + \sum_{i=x}^{y}(cantCamionesDia(calendario, i) - \sum_{j=1}^{x-1}(cantCamionesDia(calendario, j)$$

Cancelando, nos queda:

$$\sum_{i=x}^{y}(cantCamionesDia(calendario, i) = cantCamionesIntervalo(calendario, < x, y >)$$

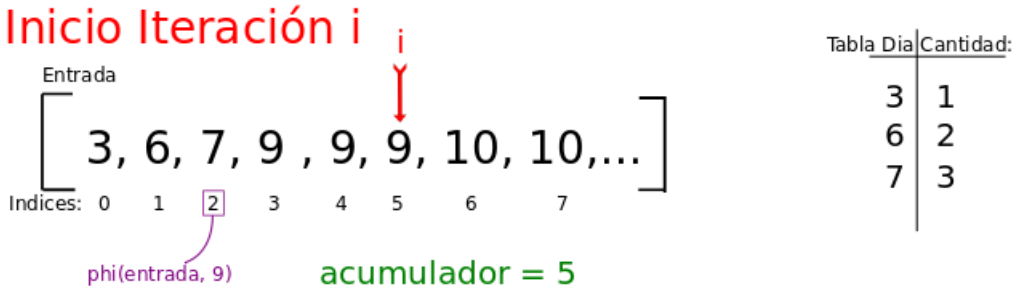
Por definición de $cantCamionesIntervalo$.

Ahora nos queda probar que nuestro algoritmo calcula correctamente la acumulación durante el ciclo. Veamos el código.

```
1:  procedure AGRUPARYACUMULAR(int entrada[])
2:      vector < pair < int, int >> tablaDiaCantidad;
3:      int longuitud = entrada.size();
4:      tablaDiaCantidad.reserve(longuitud)
5:      int i = 0;
6:      int acumulador = 0;
7:      while i < longuitud do
8:          acumulador + +;
9:          if (entrada[i]! = entrada[i + 1])||(i == longuitud − 1) then
10:             pair < int, int > par = make_pair(entrada[i], acumulador);
11:             tablaDiaCantidad.push_back(par);
12:         end if
13:         i + +;
14:     end while
15: end procedure
```

El algoritmo reserva el vector de tuplas. Paso seguido declara el indice i para iterar hasta $entrada.size()$ con lo cual vemos que itera sobre todos los días. Se declara el entero $acumulador$ para guardar la suma acumulada de los días en el vector.

Veamos el ciclo. El invariante del ciclo que tenemos es que, en cada iteración, $acumulador = \sum_{j=1}^{i-1}(cantRepeticiones(entrada[0..i), entrada[j]))$, es decir contiene la cantidad de apariciones en el vector de entrada de todos los dás hasta el que estamos iterando, y $\forall x \in entrada[0..\phi(entrada[i])]\exists y \in tablaDiaCantidad, y_1 = x \wedge y_2 = \sum_{j=1}^{x}(cantCamionesDia(entrada, j) \wedge ordenado(tablaDiaCantidad)$, es decir, en cada iteracion, por cada día que esté en entrada hasta el anterior día a i en el que viene un camión (la funcion ϕ devuelve el índice anterior a la primera aparición de entrada[i]) tablaDiaCantidad va a tener una tupla, con ese día como primer componente y la acumulaciíon de camiones que llegaron como segunda componente, y está ordenado.



Suponemos que vale el invariante al comenzar el ciclo. En (7) se le suma 1 a acumulador. Podemos ver que en caso de no valer la guarda del if, no se realizan modificaciones a tablaDiaCantidad, y al final del ciclo (12) se le suma 1 a i. Por lo cual esto valida a acumulador (que suma 1 más al igual que el índice), y se reestablece la segunda parte del invariante, al ser $entrada[i] = entrada[i + 1]$, la función ϕ va a devolver el mismo número que en la iteración anterior, y tablaDiaCantidad no recibió modificaciones.

En caso de valer la guarda del if, significa que $(entrada[i]! = entrada[i+1])||(i == longuitud - 1)$. Si se da la primera condición, acumulador también vale al finalizar el ciclo ya que suma 1 junto con el índice. La segunda parte del invariante se reestablece, ya que se agrega la nueva tupla $< entrada[i], acumulador >$ a tablaDiaCantidad. La función ϕ en la próxima iteración va a devolver el valor de i actual, valiendo que $\forall x \in entrada[0..\phi(entrada[i])]$, existe una tupla en tablaDiaCantidad, en particular para $entrada[i]$ es la que acabamos de agregar. La segunda componente de la nueva tupla va a ser igual a $acumulador = \sum_{j=1}^{i-1}(cantRepeticiones(entrada[0..i), entrada[i-1]))$ lo que es igual trivialmente a $\sum_{j=1}^{x}(cantCamionesDia(entrada, j)$ por definición de $cantCamionesDia$ y porque al estar ordenada $entrada$, no va a haber más apariciones de el día de la nueva tupla en $entrada[i+1.,0]$.

En caso de valer la segunda condición, llegamos al elemento final de entrada por lo cual tenemos que agregarlo a tablaDiaCantidad, y al ser el elemento final ya no va a haber más apariciones de él, por lo tanto no va a haber más camiones que lleguen ese día, lo cual reestablece trivialmente los invariantes.

Con esto queda demostrado la correctitud del cálculo de este vector de tuplas, y sus propiedades.

2.3.2. Correctitud de la reducción del dominio

Es necesario demostrar que la reducción de los intervalos a analizar que realiza el algoritmo es correcta, es decir, que del conjunto de soluciones óptimas del problema, nuestro algoritmo va a poder devolver efectivamente una de ellas. Nuestro algoritmo solamente evalú aquellos intervalos de longitud D que comienzan con un día en el que viene al menos un camión. Es bastante intuitivo pensar que, cualquier intervalo $< x, y >$ puede ser "mejorado"si se le realiza un çorrimiento a derecha"hasta el primer día en el que venga un camión en el intervalo. De esta forma no se pierde ningun camión y se abre la posibilidad de que vengan más despues, ya que se aprovecha mejor la longitud del intervalo.

Dicho formalmente, sea c un natural tal que $x \leq c \leq y$ y $cantCamionesDia(calendario, c) \geq 1$ y $\forall i \in [x..y], cantCamionesDia(calendario, i) \geq 1 \Rightarrow c \leq i$, entonces podemos transformar al intervalo $< x, y >$ en el intervalo $< c, c + D >$ y asegurarnos que $cantCamionesIntervalo(calendario, < x, y >) \leq cantCamionesIntervalo(calendario, < c, c + D >)$.

Para demostrar que nuestro algoritmo realiza correctamente la reducción del dominio, basta probar que del conjunto de intervalos de longitud D $\subset I_D$ que son soluciones óptimas, esta reducción deja al menos uno (ya que se pide devolver sólo uno).

Necesitamos probar que para cualquier solución óptima, podemos çonstruirnos.ºtra solución óptima desplazándola a la derecha tal que la cantidad de camiones que lleguen en los dos intervalos sea la misma, y dado que el nuevo intervalo comienza en un día en el que viene el camión, va a ser evaluada por nuestro algoritmo.

Sea $\alpha =< x, y >$ una solución óptima, es decir,

$$cantCamionesIntervalo(calendario, \alpha) = maximizarCamiones(calendario, D) \wedge y - x = D$$

Sea c el natural antes descripto. Sea $\beta =< c, c+D >$. Como $\forall i \in [x..y], cantCamionesDia(calendario, i) \geq 1 \Rightarrow c \leq i$ entonces $i \in [c..c + D]$, ya que por transitividad $x \leq c \leq i \leq y \leq c + D$. Esto se traduce a que todo día i del intervalo α en el que viene un camión está pertenece al el intervalo β.

Como demostramos que dada una solución óptima cualquiera podemos construirnos otra tal que pase la misma cantidad de camiones y comience en un día en el que viene uno, falta demostrar que nuestro algoritmo efectivamente evalúa todos los intervalos posibles que comienzan en un día en el que viene un camión. Podemos verificar esto trivialmente viendo esta porción del ciclo principal código fuente.

```
1: for(inti = 0; i < tablaDiaCantidad.size(); + + i)
2:     pair < int, int > target = tablaDiaCantidad[i]
```

Vemos que el ciclo itera de 0 a la longitud del vector de tuplas, es decir, evalúa todos los intervalos por los que pasa un camión.

2.3.3. Correctitud del cálculo de los camiones de un intervalo

Vamos a demostrar ahora que efectivamente el cálculo de la cantidad de camiones en un intervalo dado es correcta. Para calcularla, nuestro algoritmo realiza la resta antes propuesta: $acumDias(calendario, intervalo_1)$ $acumDias(calendario, intervalo_2 - 1) =$

Veamos la secciíon del ciclo principal del código que se encarga de realizar esta tarea. Realizaremos la demostración sobre el pseudocódigo para abstraernos de los iteradores y demás inconveniencias del código fuente, pero es fácil comprobar que ambos códigos son equivalentes.

1: $tupla < int, int > inicio \leftarrow dias[i]$
2: $int\ diaFinal \leftarrow inicio.first + diasInspector - 1$
3: $tupla < int, int > final \leftarrow dias[BusquedaBinaria(dias, i, diaFinal, dias.size())]$
4: $int\sharp camionesDelIntervalo \leftarrow final.second - dias[i-1]$

Tomamos una tupla del vector, llamada *inicio* (1). En su primer componente $inicio_1$ tenemos el inicio del intervalo que queremos evaluar. Para tomar el intervalo de longitud D (diasInspector) que comienza en $inicio_1$, calculamos el final del intervalo como $diaFinal = inicio_1 + D - 1$ (2). Ya tenemos el intervalo a evaluar. Entonces, para ser correcto, nuestro algoritmo debería calcular $acumDias(entrada, diaFinal) - acumDias(entrada, inicio_1 - 1)$.

En (3), el algoritmo realiza una búsqueda binaria sobre *dias*, buscando la tupla cuya primer componente sea $diaFinal$. De esta forma, en su segunda componente tenemos la suma acumulada de camiones hasta el dia final del intervalo y podriamos realizar la resta propuesta. La funciíon $BusquedaBinaria$ (upperBound) nos garantiza el índice del primer elemento mayor a $diaFinal$, al restarle uno, obtenemos o bien el índice de la tupla cuyo primer componente es $diaFinal$, o el índice de la tupla cuyo primer componente es el mayor número menor a $diaFinal$ en caso de éste no existir. Llamamos a esta tupla $final$.

Finalmente (4), realizamos la resta $final_2 - dias[i-1]_2$. Gracias a la demostración anterior, esto podría traducirse como $\sum_{k=1}^{final_1}(cantCamionesDia(entrada, k) - \sum_{j=1}^{dias[i-1]_1}(cantCamionesDia(entrada, j)$. Esto es igual a $acumDias(entrada, diaFinal) - acumDias(entrada, inicio_1)$, por los siguientes argumentos:

Veamos ahora que $acumDias(entrada, diaFinal) = \sum_{k=1}^{final_1}(cantCamionesDia(entrada, k)$

Por definición $acumDias(entrada, diaFinal) = \sum_{j=1}^{diaFinal}(cantCamionesDia(entrada, j)$. Si $diaFinal$ se encontraba en el calendario, entonces la búsqueda binaria nos devolvió el índice de la tupla cuya primer componente es $diaFinal$, entonces $final_1 = diaFinal$, y vale la igualdad. En caso de no existir $diaFinal$ en el calendario, la búsqueda nos brinda el índice del primer elemento mayor a $diaFinal$, a lo cual restándole 1 obtenemos el mayor número menor a $diaFinal$. Esto significa que $\forall x \in [final_1..diaFinal], cantCamionesDia(entrada, x) = 0$. Con lo cual podemos extender la sumatoria desde $final_1$ hasta $diaFinal$, haciendo valer la igualdad.

Veamos que $acumDias(entrada, inicio_1 - 1) = \sum_{k=1}^{dias[i-1]}(cantCamionesDia(entrada, k)$. Por (1), $inicio_1 = dias[i]_1$. En caso de existir el día anterior a $inicio_1$, entonces $inicio_1 - 1 = dias[i-1]_1$ y vale la igualdad. En caso de no existir, $dias[i-1]_1$ es el mayor número de los menores a $inicio_1$. Esto significa que $\forall x \in [dias[i-1]_1..inicio_1 - 1], cantCamionesDia(entrada, x) = 0$ con lo cual puedo extender la sumatoria hasta $inicio_1 - 1$ y hacer avaler la igualdad.

Con esto demostramos que el algoritmo calcula correctamente la función que calcula la cantidad de camiones de un intervalo.

2.3.4. Correctitud del cálculo del máximo

Nos queda demostrar que es correcto el cálculo del máximo. Veamos la sección del código que calcula esto.

```
1: int maximo = −1;
2: int diaAContratar = −1;
3: for uint i = 0; i <dias.size(); ++i do
4:     pair < int, int > inicio = dias[i];
5:     ...procesamiento y busqueda....
6:     int cantCamionesDelIntervalo = (final.second − resta;
7:     if cantCamionesDelIntervalo > maximo then
8:         maximo = cantCamionesDelIntervalo;
9:         diaAContratar = dias[i].first;
10:    end if
11: end for
12: return par < diaAContratar, maximo >;
```

Podemos ver que en cada iteración, tenemos un condicional tal que si $cantCamionesDelIntervalo > maximo$ se reemplaza el máximo anterior por el nuevo y el dá contratar. Se itera hasta el final del arreglo, por lo que la elección del máximo es correcta.

2.4. Cota de complejidad

Lo primero que hace nuestro algoritmo es ordenar el vector de entrada. Según la documentación del lenguaje, esto se realiza en $O(n \log n)$. Acto seguido nuestra función $acumularYAcumular$ reserva un vector de longitud n ($O(n)$) y realiza n iteraciones sobre el vector de entrada, en la que puede verse que realiza todas operaciones de tiempo constante (la documentación del lenguaje nos asegura esta complejidad para $make\_pair$ y $push\_back$), por lo que el ciclo es de complejidad $O(n)$. Se declaran algunas variables (tiempo constante), y finalmente se realiza el ciclo principal, en el cual se recorre de 0 a dias.size(), realizando todas operaciones de tiempo constante excepto la $BusquedaBinaria$, cuya documentación (upperBound) nos asegura tiempo logarítmico. Esto nos da una complejidad de $O(dias.size() \log dias.size())$. Pero dado que $dias.size() \leq n$, podemos acotar por n lo que nos deja una complejidad para el algoritmo (omitiendo constantes) de $O(n \log n) + O(n) + O(n) + O(n \log n) = O(n \log n)$.

3. Ejercicio 2

3.1. Descripción del problema

Un joyero recibe un encargo de una serie de piezas de orfebrería para fabricar. Cada pieza posee una cantidad de días que tarda el joyero en fabricarla (sólo puede fabricar una al mismo tiempo), y un valor de depreciación, esto es, la cantidad de dinero que el joyero pierde diariamente mientras esa pieza no sea entregada. Se pide realizar un algoritmo que calcule el orden óptimo de fabricación de las piezas, esto es, el orden en el que el joyero debe fabricar cada una de forma de que se minimize la pérdida que sufre.

La entrada del algoritmo, por cada instancia son n lineas di ti, donde di representa el valor de depreciación y ti la cantidad de dias que tarda la fabricación de esa pieza.

3.2. Formalización Matemática

Para formalizar, es análogo decir que se recibe una entrada compuesta por dos arrays no vacios de numeros enteros positivos de igual longitud, T y D. El problema consiste en encontrar la ordenacion (la misma aplicada a ambos arrays de entrada) que minimiza el valor de la siguiente funcion:

```
loop
    n ← long(T)
    ret ← 0
    for i = 0 to n-1 do
        a ← 0
        for j = 0 to i-1 do
            a ← a + T[j]
        end for
        ret ← ret + a.D[i]
    end for
    return ret
end loop
```

En otras palabras:

$$f(T, D) = \sum_{i=0}^{n-1} \left(D[i] \sum_{j=0}^{i-1} T[i] \right)$$

3.2.1. Ejemplos

Ejemplo 1. $T = [1, 1, 1, 1, 1]$, $D = [2, 2, 2, 2, 2]$, $f(T, D) = [1, 2, 3, 4, 5]$. *En este caso cualquier ordanacion vale.*

Ejemplo 2. $T = [1, 1]$, $D = [1, 2]$, $f(T, D) = [2, 1]$. *En este caso el mas costoso se hace primero.*

Ejemplo 3. $T = [3, 1, 2]$, $D = [1, 1, 1]$, $f(T, D) = [2, 3, 1]$. *En este caso se hacen primero los que llevan menos tiempo.*

Ejemplo 4. $T = [3, 3, 1]$, $D = [5, 4, 4]$, $f(T, D) = [3, 1, 2]$. *Un caso general.*

3.3. Ideas para la resolución

Se construye un array A de tuplas (i, a_i) con $a_i = T[i]/D[i]$. Luego se ordena el array A de menor a mayor por su segunda componente. Se devuelve el array A_1, la proyección de A en la primer componente.

```
loop
    n ← long(T)
    for i = 0 to n-1 do
```

```
        A[i] ← (i, T[i]/D[i])
    end for
    ordenar(A,2)                    ▷ Ordenar A por la componente 2. El sorting es O(n log n)
    for i = 0 to n-1 do
        ret[i] ← A[i].1             ▷ Me quedo con la primer componente
    end for
    return ret
end loop
```

Se puede ver una implementación del algoritmo en la sección 7.2.

3.3.1. Ejemplos

Ejemplo 5. $T = [3, 3, 1]$, $D = [5, 4, 4]$.
Antes del sorting $A = [(1, 3/5), (2, 3/4), (3, 1/4)]$.
Despues del sorting $A = [(3, 1/4), (1, 3/5), (2, 3/4)]$.
$Op(T, D) = [3, 1, 2]$.

3.4. Correctitud

3.4.1. Notación

Siendo t_i y d_i los tiempos y pérdidas por día de cada manufactura, se define $a_i = t_i/d_i$. La función "pérdida total" asocia a cada orden (permutación) de los objetos fabricados el dinero total perdido según la fórmula (1)

$$f(\tau) = \sum_{i=1}^{n}(d_{\tau(i)} \sum_{j=1}^{i} t_{\tau(i)}) \tag{1}$$

3.4.2. Demostración

Teorema 1. *Sea* $(i\ i+1)$ *una transposición. Entonces* $f((i\ i+1) \circ \tau) = f(\tau)$ *si* $a_{\tau(i)} = a_{\tau(i+1)}$, $f((i\ i+1) \circ \tau) > f(\tau)$ *si* $a_{\tau(i)} < a_{\tau(i+1)}$ *y* $f((i\ i+1) \circ \tau) < f(\tau)$ *si* $a_{\tau(i)} > a_{\tau(i+1)}$.

Demostración. Se puede ver por cálculo directo que $f((i\ i+1) \circ \tau) - f(\tau) = t_{\tau(i+1)}d_{\tau(i)} - t_{\tau(i)}d_{\tau(i+1)}$. Se cumple que

$$t_{\tau(i+1)}d_{\tau(i)} - t_{\tau(i)}d_{\tau(i+1)} > 0 \iff a_{\tau(i)} < a_{\tau(i+1)}$$
$$t_{\tau(i+1)}d_{\tau(i)} - t_{\tau(i)}d_{\tau(i+1)} < 0 \iff a_{\tau(i)} > a_{\tau(i+1)}$$
$$t_{\tau(i+1)}d_{\tau(i)} - t_{\tau(i)}d_{\tau(i+1)} = 0 \iff a_{\tau(i)} = a_{\tau(i+1)}$$

□

Teorema 2. *Sea* τ *cualquier permutación con la propiedad de que* $i \leq j \Rightarrow a_{\tau(i)} \leq a_{\tau(j)}$. *Entonces la función* f *alcanza su mínimo en* τ.

Demostración. Dado que el dominio de f es finito existe una permutación τ' en donde la función alcanza el mínimo.

Para tal permutación se cumple la propiedad citada en el enunciado, ya que de lo contrario existiría un índice i tal que $a_{\tau'(i)} > a_{\tau'(i+1)}$ lo cual por el teorema anterior implicaría que $f((i\ i+1) \circ \tau') < f(\tau')$ contradiciendo la minimalidad de τ'. Que $f(\tau) = f(\tau')$ se puede deducir de que como ambas permutaciones cumplen la propiedad, una se puede transformar en la otra por composición de transposiciones $(i\ i+1)$ con $a_{\tau(i)} = a_{\tau(i+1)}$, y por el teorema anterior la f se mantiene invariante en cada una de ellas. □

3.5. Correctitud del ciclo de acumulación

Nuestro código fuente calcula el monto total perdido mediante un ciclo lineal de atrás hacia adelante para mejorar la ejecución. Recordemos que dado un vector A de tuplas $< depreciacion, tiempo >$, el monto total perdido es:

$$Depreciacion(A) = \sum_{i=0}^{n-1}(i_2) * (\sum_{j=i}^{n-1}(j_1))$$

Veamos el ciclo:

$i = n - 1$
while $i \geq 0$ **do**
  $depreciaciones = depreciaciones + (piezas[i])_1$;
  $totalPerdido = totalPerdido + (piezas[i])_2 * depreciaciones$;
  $i - -$;
end while

Podemos ver que en cada iteración, *depreciaciones* contiene la sumatoria de las depreciaciones de las piezas de $[i..n-1]$.,o sea, $\sum_{j=i}^{n-1}(piezas[i]_1)$.Por otro lado $totalPerdido$ se suma en cada iteración a si mismo + $piezas[i]_2 * depreciaciones$. Con lo cual en cada iteración posee la sumatoria $\sum_{k=i}^{n-1}(piezas[i]_2 * \sum_{j=i}^{n-1}(piezas[i]_1))$. Como estamos hablando del mismo indice i, como $k = i$ podemos reemplazar en la segunda sumatoria $j = i$ por $j = k$, ya que esto es cierto en cada iteración. Cuando se llega al final del ciclo, $i = 0$ queda la fórmula de $Depreciacion(piezas)$.

3.6. Cota de complejidad

Siendo $n = long(T)$, el primer bucle tarda $O(n)$, el sorting $O(n\ log(n))$ (esto es un requerimiento) y el segundo bucle $O(n)$. El tiempo total es $O(n\ log(n))$.

4. Ejercicio 3

4.1. Descripción del problema

La entrada consta de un tablero de n x m casilleros y $k = n * m$ fichas cuadradas a colocar, estando cada uno de los 4 lados de las fichas identificado con un color. Se requiere colocar la mayor cantidad de fichas posibles en el tablero, tal que si 2 fichas son adyacentes, entonces el lado del cuadrado que tienen en común tiene el mismo color. Las fichas ya vienen con una orientación y no pueden ser rotadas. Ver la imagen 1 para un ejemplo de la entrada, y la imagen 2 para ver una solución posible donde se colocan todas las fichas. Se pide utilizar la técnica de *Backtracking* y elaborar podas para mejorar los tiempos de ejecución.

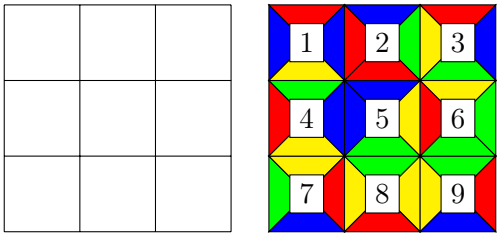


Figura 1: Un tablero de 3 x 3 con las fichas que se deben colocar en el tablero

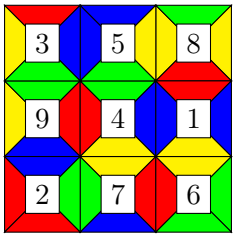


Figura 2: Solución colocando todas las fichas del problema de la figura 1

4.2. Ideas para la resolución

Como los casilleros del tablero pueden quedar vacíos si no se encuentra una ficha que pueda ser colocada, llamando $k = n * m$, n y m las filas y columnas del tablero, y i a la cantidad de casilleros que se dejan vacíos, para cada i se pueden elegir de $\binom{k}{i}$ formas diferentes esos i casilleros a dejar vacíos (porque de la cantidad de casilleros totales, se eligen i los cuales son indistinguibles), y para cada i, nos quedan $k - i$ casilleros donde colocar fichas, las cuales sí son distinguibles, y se pueden colocar de $k!/i!$ formas diferentes. Por lo tanto, la cantidad de combinaciones que se pueden hacer con el tablero incluyendo dejar casilleros libres, es (2)

$$\sum_{i=0}^{k} \binom{k}{i} \frac{k!}{i!} \tag{2}$$

$$= \sum_{i=0}^{k} \frac{k!}{i!(k-i)!} \frac{k!}{i!}$$

$$= \sum_{i=0}^{k} \frac{(k!)^2}{(i!)^2(k-i)!}$$

$$= (k!)^2 \sum_{i=0}^{k} \frac{1}{(i!)^2(k-i)!}$$

Pero para la resolución del problema sólo nos interesan las combinaciones válidas, la idea que se plantea es ir completando el tablero casillero a casillero, y en cada casillero intentar colocar todas las fichas tal que no hayan sido colocadas antes y que al colocarla siga siendo una combinación válida (que

coincidan los colores con las fichas adyacentes). Así se van analizando sólo las combinaciones válidas, que serán menor o igual que la cuenta anterior (4.3), y retornando la combinación que contenga la mayor cantidad de fichas colocadas.

Como el resultado será una sóla combinación que contenga la mayor cantdidad de fichas colocadas, se puede elaborar una *poda* para descartar combinaciones a medida que se están armando para no terminar de generarlas y analizarlas si se determina que aunque se sigan agregando fichas a la combinación en la que se está trabajando, no podrá contener más fichas colocadas que alguna otra combinación que ya se haya calculado. Supongamos que se está trabajando con un tablero de 3 x 3 (9 fichas), y ya se procesó una combinación que sólo se deja vacío un casillero (contiene 8 fichas colocadas), guardando esta posible solución como la que más fichas contiene; y actualmente se está procesando una posible solución que ya analizó 6 casilleros del tablero (quedando 3 aún por procesar), pero tuvo que dejar 2 casilleros vacíos (es decir, de los 6 casilleros 4 contienen fichas), si en los 3 casilleros restantes colocase 3 fichas (lo máximo que puede conseguir), esta combinación tendría 7 fichas colocadas (4 que ya había colocado mas 3 posibles fichas que puede colocar a continuación) y 2 casilleros vacíos, siendo menor a la candidad de 8 fichas de la solución guardad como máxima encontrada hasta el momento, por lo que se puede descartar la solución que se venía procesando porque nunca podría llegar a ser una solución al problema.

4.2.1. Algoritmo para la resolución propuesta

Como se describió en la sección (4.2), la resolución del problema se basa en 2 ideas, sólo analizar combinaciones válidas y descartar combinaciones si ya se calcula que no pueden superar a una combinación máxima ya encontrada. Se propone el algoritmo 1.

Algorithm 1 maximizarTablero

Require: *tablero*: el tablero donde se irán colocando las fichas
Require: *n, m*: filas y columnas del tablero
Require: *fichas_disponibles*: array con las fichas aun diponibles por poner
Require: *i_actual, j_actual*: la posición que se tiene que procesar
Require: *casilleros_calculados*: cantidad de casilleros que ya se procesaron
Require: *fichas_colocadas*: cantidad de fichas colocadas (no se cuentan los casilleros vacíos dejados)
Require: *tablero_optimo*: el optimo encontrado hasta el momento
Require: *fichas_maximas*: cantidad máxima de fichas que se pudieron colocar en el tablero_optimo
Ensure: La función va a dejar en *tablero_optimo* y en *fichas_maximas* la mejor combinación que pudo encontrar de fichas tal que las fichas adyacentes comparten el color en el borde que tienen en común, y es la combinación en la que más fichas se pudo colocar

```
1:  procedure MAXIMIZARTABLERO(tablero, n, m, fichas_disponibles, i_actual, j_actual, casilleros_calculados,
    fichas_colocadas, tablero_optimo, fichas_maximas)
2:      total_casilleros ← n * m
3:      n_fichas ← total_casilleros
4:      if casilleros_calculados = total_casilleros then
5:          if fichas_colocadas > fichas_maximas then
6:              copiarTablero(tablero_optimo, tablero)
7:              fichas_maximas ← fichas_colocadas
8:          end if
9:          return
10:     end if
11:     if total_casilleros − casilleros_calculados + fichas_colocadas <= fichas_maxima then
12:         return
13:     end if
14:     proximo_i ← i_actual
15:     proximo_j ← j_actual
```

```
16:     if i_actual = n − 1 then
17:         proximo_i ← 0
18:         proximo_j + +
19:     else
20:         proximo_i + +
21:     end if
22:     for f ∈ fichas_disponibles do
23:         if validaColocar(f, tablero, i_actual, j_actual) then
24:             tablero[i_actual][j_actual] = f
25:             if ¬f− > vacia then
26:                 fichas_nuevas ← fichas_disponibles − f
27:                 fichas_colocadas + +
28:             else
29:                 fichas_nuevas ← fichas_disponibles
30:             end if
31:             maximizarTablero(tablero, n, m, fichas_nuevas, i_proximo, j_proximo, casilleros_calculados
    + 1, fichas_colocadas, tablero_optimo, fichas_maximas)
32:             if ¬f− > vacia then
33:                 fichas_colocadas − −
34:             end if
35:         end if
36:     end for
37: end procedure

38: procedure BOOL VALIDACOLOCAR(ficha, tablero, i, j)
39:     if ficha− > vacia then
40:         return ← true
41:     end if
42:     if j > 0 then
43:         if ¬tablero[i][j − 1]− > vacia then
44:             if tablero[i][j − 1]− > derecha ≠ fichas− > izquierda then
45:                 return ← false
46:             end if
47:         end if
48:     end if
49:     if i > 0 then
50:         if ¬tablero[i − 1][j]− > vacia then
51:             if tablero[i − 1][j]− > abajo ≠ fichas− > arriba then
52:                 return ← false
53:             end if
54:         end if
55:     end if
56:     return ← true
57: end procedure
```

La función maximizarTablero (algoritmo 1), es recursiva y en la primera llamada debe llamarse con *tablero*, siendo una matriz de $n * m$, donde se colocarán las fichas. n y m son las filas y columnas del *tablero*. *fichas_ disponibles* tiene que contener todas las fichas que vinieron en la entrada y con el campo *vacía* en $false$, más una ficha con el campo *vacía* en *true* (para indicar que ese casillero se dejo vacío). *i_ actual*, *j_ actual*, *casilleros_ calculados* y *fichas_ colocadas* tienen que estar en 0. *tablero_ optimo* es donde se va a colocar el tablero con mayor cantidad de fichas colocadas, y en *fichas_ maximas* se va a colocar la cantidad de fichas en dicho tablero, *fichas_ maximas* debe estar en un valor menor a 0 cuando se llama al algoritmo.

El algoritmo (1) lo que realiza es llamarse recursivamente probando en cada casillero todas las fichas que esten disponibles para poner y que sea valido colocarlas.

Entre las líneas 4 y 10 se encuentra el caso base de la recursión, el cual sucede cuando ya se procesaron todos los casilleros, si la cantidad de fichas colocadas es mayor a la cantidad máxima que se había almacenado, entonces reemplaza el *tablero_optimo* con el tablero actual, y actualiza la variable *fichas_maximas*.

Luego en la línea 11 se encuentra la *poda* para cortar con la recursión antes de procesar todos los casilleros si aunque se completen todos los casilleros restantes con fichas, no superarán la cantidad máxima de fichas que ya se almacenó con otra combinación.

Entre las líneas 14 y 21 se calcula el próximo casillero que se deberá llamar en la recursión

Y por último entre las líneas 22 y 36 se buscan todas las fichas disponibles para colocar (línea 22) y que sea válida su colocación (línea 23), entonces si se cumplen dichas condiciones, coloca la ficha en el tablero (línea 24) y si no es la ficha que corresponde al casillero vacío (línea 25) la elmina de las fichas disponibles, y se llama recursivamente (lĺinea 31) con la nueva posición a analizar.

Tambiém se encuentra la función *validaColocar* como una función auxiliar que retornará *true* si colocar la *ficha* en el *tablero* en la posición i, j, pero tiene en cuenta que el tablero se va completando de izquierda a derecha y de abajo a izquierda (el $(0; 0)$ corresponde a arriba a la izquierda), y en base a esto sólo chequea 2 lados de la ficha a colocar y no los 4.

En la sección (4.3) se hablará sobre la justificación y correctitud del algoritmo.

Se puede ver una implementación del algoritmo en la sección 7.2.

4.2.2. Ejemplo de la resolución propuesta

Siguiendo los pasos del algoritmo de la sección 4.2.1, se procede a un ejemplo de la ejecución con un tablero de 2 x 2 y 4 fichas, y 3 colores (imagen 3, todas las imágenes se encuentran al final de la sección 4.2.2), agregando la ficha V para indicar dejar el casillero vacío

El algoritmo comienza en la posición $(0; 0)$ y comienza a iterar por todas las fichas disponibles, comenzando por la número 1, la cual es válida de colocar (ya que no tiene fichas al rededor) (imagen 4) y la coloca, llando recursivamente al algoritmo pasándole como posición el valor $(0; 1)$.

Ahora vuelve a recorrer la lista de fichas disponibles (la ficha 2, 3 y 4) y como la ficha 2 no es válida de colocar (el color no coincide) (imagen 5), entonces avanza a la siguiente ficha, la 3 y como es válida (comparte el color verde), la coloca (imagen 6) y se llama recursivamente ahora con la posición $(1; 0)$. Como descartó la ficha 2 para ésta posición con la ficha 1 colocada en la primera posición, ya descartó todos los casos que comienzen con la ficha 1 en la posición $(0; 0)$ y la ficha 2 en $(0; 1)$ (7 casos contando dejar casilleros vacíos)

Quedando 2 fichas disponibles, y estando posicionado en $(1; 0)$, la primer ficha disponible es la númeroro 2, la cual es válida de colocar (imagen 7) y llama recursivamente para posicionarse en $(1; 1)$ y la última ficha disponible también es válida y es colocada (imagen 8).

En la siguiente llamada recursiva, ya todos los casilleros fueron procesados por ende entra en el caso base del algoritmo (1), el cual como no tenía valor cargado para la cantidad máxima de fichas ya calculadas, guarda ésta opción como la mejor combinación hasta el momento con 4 fichas colocadas (imagen 9).

Luego de ésto, retorna de la recursión y vuelve a quedar con un casillero sin procesar y con la ficha 4 disponible y la 'ficha vacía' disponible, la ficha 4 la procesó en la iteración anterior, por lo que intenta con la ficha vacía (imagen 10), llama nuevamente a la recursión, pero como tiene 3 fichas colocadas en vez de 4 que se tiene guardado como mejor resultado hasta el momento, entonces retorna y descarta la combinación. Al haber procesado para el último casillero todas las posibilidades, vuelve a retornar de la recursión quedando 2 casilleros sin procesar y 2 fichas disponibles, la ficha 2 ya fue procesada por lo que intenta con la 4, la cuál no es válida de colocar, y termina dejando el casillero vacío y llamar recursivamente para el último casillero. En este caso entra en la condición de la *poda*, ya que tiene 2 casilleros con fichas, y queda 1 casillero por procesar, lo cual nunca podrá superar a las 4 fichas máximas que pudo poner anteriormente.

De a partir de ahora como 4 es lo máximo que se puede colocar, todas las demás combinaciones que se intenten van a entrar en la condición de la *poda* y el algoritmo va a terminar retornando como combinación óptima el tablero guardado de la imagen 9

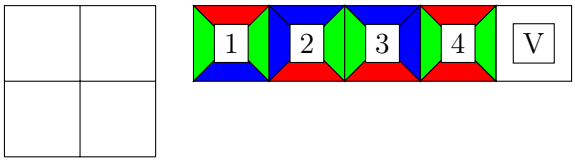


Figura 3: Tablero y fichas de entrada. La ficha V va a representar dejar el casillero vacío

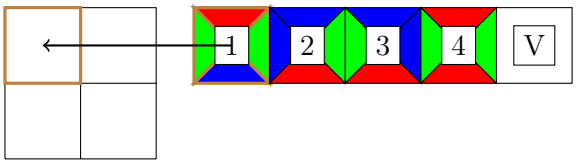


Figura 4: Comienza con la primer posición del tablero y la primera ficha disponible para colocar, siendo esta válida para ser colocada

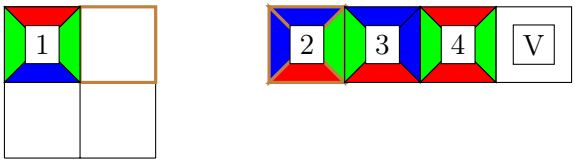


Figura 5: Se mueve a la segunda posición el tablero y analiza la primer ficha disponible, pero no es válida su colocación

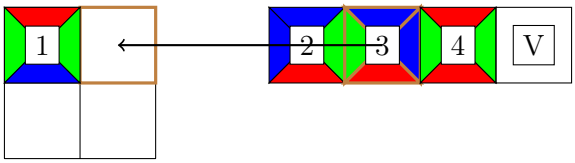


Figura 6: Ahora analiza la ficha número 3, la cual si es válida colocar

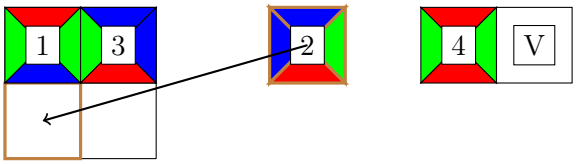


Figura 7: Avanza al siguiente casillero y vuelve a fijarse en la segunda ficha disponible para poner, como es válida, la coloca

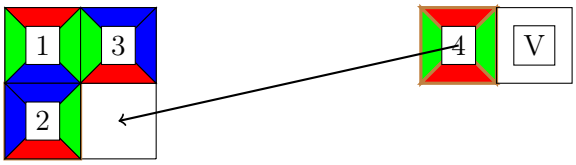


Figura 8: Avanza al último casillero y se fija en la última ficha disponible que queda, como es válida, la coloca

Figura 9: Al no quedar casillero sin procesar, se llega al caso base y guarda la combinación como la máxima encontrada

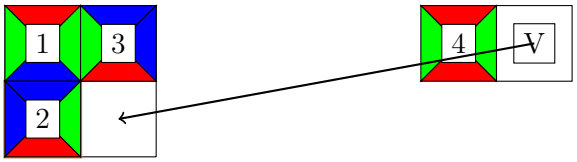


Figura 10: Habiendo retornado de la recursión y volviendo al paso de la imagen 8, la ficha que queda por colocar es la que representa al casillero vacío

4.3. Correctitud

Como se comentó en la sección de la idea de la resolución (4.2), tenemos $\sum_{i=0}^{k} \binom{k}{i} \frac{k!}{i!}$ (2) combinaciones diferentes del cubo, si el algoritmo las recorre todas y toma la máxima combinación válida, entonces el algoritmo es correcto, ya que no le quedaría ninguna combinación por validar. Pero el algoritmo no chequea todas las combinaciones, la idea es descartar las combinaciones en las que las fichas no están colocadas correctamente, y realiza una *poda*, por lo que hay que justificar si el algoritmo saca el máximo de un conjunto de soluciones posibles correctamente, y si dicho conjunto de posibles soluciones compuesto por todas las posibles combinaciones descartando las que contienen fichas adyacentes que no tienen el mismo color y quitando los que descarta la poda, contiene la solución máxima que se está buscando.

Sin la línea 23 del algoritmo (1) que valida la ficha a colocar, el algoritmo recorre todas las posiciones y en cada una recorre todas las fichas disponibles, y sólo saca la ficha de disponibildiad cuando la coloca, es decir, si una ficha no está disponible es porque ya fue colocada y las fichas no se pueden repetir.

En el descarte de las combinaciones que las fichas no están colocadas correctamente, la solución que se pide encontrar tiene que contener todas las fichas colocadas correctamente, tal que 2 fichas adyacentes tengan el lado en común con el mismo color, por lo que descartar las combinaciones no válidas no está descartando la solución que se pide encontrar. El algoritmo (1) recorre el tablero de izquierda a derecha y de arriba a abajo, y en cada posición recorre todas las fichas disponibles, la coloca si es válida, y continúa a la siguiente posición quitando como disponible la ficha colocada, cuando retorna de la recursión, la ficha que se había colocado se vuelve a poner como disponible y continúa con la próxima ficha disponible. Si la ficha no es válida, entonces descarta todas las combinaciones que comienzen con las fichas:

$$\{f_0, \ldots, f_i\} \tag{3}$$

siendo f_i la ficha no válida de colocar, y $\{f_0, \ldots, f_{i-1}\}$ las fichas colocadas en los pasos anteriores, y como todas las fichas en la solución que se quiere encontrar tienen que estar puestas correctamente, entonces ésta solución no puede comenzar con $\{f_0, \ldots, f_i\}$ por tener una ficha que no es válida de colocar en esa posición

En el caso de la *poda*, el algoritmo tiene la variable $fichas\_maximas$ que contiene la cantidad máxima de fichas que se pudieron colocar (y se tiene dicha combinación almacenada), la condición de la *poda* es (línea 11 del algoritmo 1)

if $total\_casilleros - casilleros\_calculados + fichas\_colocadas <= fichas\_maxima$ **then**
    return
end if

Siendo $total\_casilleros$ los casilleros del tablero, $casilleros\_calculados$ la cantidad de casilleros que ya se colocaron fichas o se dejaron vacío, y $fichas\_colocadas$ la cantidad de casilleros con fichas hasta el momento. Si dicha condición se cumple, entonces corta con la recursión y no continúa intentando. Lo que se quiere encontrar es para todos los tableros, obtener el que tiene mas fichas colocadas.

$$\forall(tablero) fichas\_colocadas(tablero\_optimo) >= fichas\_colocadas(tablero) \tag{4}$$

Para el tablero en particular en construcción que se está armando, los casilleros restantes por calcular son $total\_casilleros - casilleros\_calculados$, como buscamos el tablero con mayor cantidad de fichas, lo mejor que podemos obtener de un tablero en particular en construcción es que la totalidad de casilleros restantes contengan fichas, en ese caso a la cantidad de fichas ya colocadas, le sumamos los casilleros restantes

$$total\_casilleros - casilleros\_calculados + fichas\_colocadas \tag{5}$$

Si $fichas\_maximas >= total\_casilleros - casilleros\_calculados + fichas\_colocadas$, este tablero en construcción tendrá menor o igual de fichas que $fichas\_maximas$ porque para tener más fichas tendría que tener más casilleros, y como sólo nos interesa 1 solución, si es igual la podemos descartar una vez ya se encontró una. Entonces, la *poda* descarta combinaciones que son menores o iguales que el máximo, ya que si el máximo es el que está en *tablero_optimo*, descarta combinaciones con igual o menor cantidad de fichas, y si el máximo es otro aun no encontrado, entonces va a tener más fichas que *tablero_optimo* y más que que tablero en construcción (6), por lo que el tablero en construcción tampoco es solución

$$\begin{aligned} solucion\_aun\_no\_encontrada > fichas\_maximas >= \\ total\_casilleros - casilleros\_calculados + fichas\_colocadas \end{aligned} \tag{6}$$

Sabiendo que las combinaciones que se descartan no son soluciones correctas, el algoritmo para guardar el máximo realiza lo siguiente una vez que no quedan casilleros por procesar:

if $fichas\_colocadas > fichas\_maximas$ **then**
  copiarTablero(tablero_optimo, tablero)
  $fichas\_maximas \leftarrow fichas\_colocadas$
end if

Y por cada ficha que se coloca, realiza:

if $\neg f->vacia$ **then**
  $fichas\_nuevas \leftarrow fichas\_disponibles - f$
  $fichas\_colocadas++$
else
  $fichas\_nuevas \leftarrow fichas\_disponibles$
end if
maximizarTablero(tablero, n, m, fichas_nuevas, i_proximo, j_proximo, casilleros_calculados + 1, fichas_colocadas, tablero_optimo, fichas_maximas)

Por cada ficha que no sea dejar el casillero vacío, se incrementa la cantidad de fichas colocadas y se le pasa recursivamente a la función, si la llamada recursiva llega al caso base, $fichas\_colocadas$ tendrá la cantidad de fichas puestas, y por la condición $fichas\_colocadas > fichas\_maximas$, si se cumple significa que se encontró un tablero con más fichas, y se guarda en *tablero_optimo* y en $fichas\_maximas$ los valores correspondientes. Como se llama a la función con el valor $fichas\_maximas < 0$ (como está explicado en la sección 4.2.1), entonces con el primer tablero que calcule aunque sea dejar todo vacío obteniendo la cantidad mínima de fichas colocadas (0), ya va a cumlir que es mayor a $fichas\_maximas$ y va a colocar el tablero como posible solución.

Entonces, se vió que los casos que descarta no contienen a la solución o bien tienen la misma cantidad de fichas que alguna la solución encontrada, y el algoritmo siempre se queda con el máximo encontrado hasta el momento.

Como se recorre primero por columnas de izquierda a derecha y luego por filas de arriba a abajo, las fichas a colocar sólo pueden ser validadas contra 2 bordes de sus fichas adyacentes, arriba y/o izquierda, ya que no hay fichas aún colocadas a la derecha y abajo porque aún no se procesaron esos casilleros, si la ficha a colocar está en la primera fila, no hay que verificar el color de arriba, y si está en la primer columna, no hay que verificar el color izquierda. El tablero comienza vacío por lo que cada ficha que se coloca es validad, y al colocar una ficha, todas las anteriores están colocadas correctamente porque ya fueron validadas.

4.4. Cota de complejidad

El algoritmo es recursivo y contiene un ciclo donde recorre todas las fichas disponibles (lo implementamos en una lista, y es $O(1)$), y en cada ciclo chequea que la ficha sea válida ($O(1)$) de colocar y se llama recursivamente a la función, si no se dejó el casillero vacío entonces hay una ficha menos a recorrer en el ciclo de la llamada recursiva, por lo que nos queda que realiza:

$$\begin{aligned} n*m(n*m+1-j_1)(n*m+1-j_2)(\dots)(n*m+1-j_{n*m}) \\ = \prod_{i=0}^{n*m} (n*m+1-j_i) \end{aligned} \tag{7}$$

Siendo j_i la cantidad de fichas colocadas en el paso i de la recursión ($j_0 = 0$ porque es el primer paso y no se procesoó ningún casillero aún), y a las filas y columnas n m se le suma 1 para contar dejar el casillero vací. Pero para cada i, puede variar el valor de j_i, ya que si se está procesando el casillero i, se pudo haber llegado dejandose diferente cantidad de casilleros vacíos anteriormente, pero podemos calcular el peor caso dejar todos los casilleros vacíos siempre $(\forall i \in [1; n * m])j_i = 0$, maximizando el producto, o el mejor caso minimizando el producto, donde siempre se colocaron todas las fichas anteriores $(\forall i \in [1; n * m])j_i = i - 1$

Cuando encuentra un tablero que puede ser solución y realiza la copia, como se implementa en una matriz y se copian todos los valores, nos queda $O(n * m)$, lo cuál se realiza al llegar al final, pero por la condición de que se copia si supera la cantidad de fichas que se tienen guardadas (si es igual o menor no copia), entonces nos queda que como máximo va a copiarlo $n * m$ veces (que sería ir mejorando siempre la solución agregando 1 ficha), lo que nos queda $O((n * m)^2)$ y lo podemos sacar afuera como una suma aparte de toda la llamda recursiva. Si queremos ver la mayor cantidad de iteraciones, sería que se estén comparando todos las fichas contra todas, quedandonos

$$\begin{gathered} \prod_{i=0}^{n*m} (n * m + 1 - j_i) + (n * m)^2 \\ \prod_{i=0}^{n*m} (n * m + 1) + (n * m)^2 \\ (n * m + 1)^{n*m} + (n * m)^2 \end{gathered} \tag{8}$$

Quedando una complejidad de:

$$O((n * m)^{n*m}) \tag{9}$$

También podemos calcular que pasaría si logra colocar todas las fichas en la primera iteración del ciclo sobre las fichas disponibles, quedándonos un tablero con la máxima cantidad de fichas colocadas en $n * m$ llamadas recursivas, y ahora la poda cortará el resto de las recursiones, aunque en cada retorno de la recursión, se sigue intentando con todas las fichas que quedaban, pero no se ejecutan más recurciones por la poda, quedando

$$\sum_{i=1}^{n*m} 1 + \sum_{i=1}^{n*m} i + (n * m)$$

La primer sumatoria es el ingreso a la recursión encontrando una ficha válida en la primera iteración sobre las fichas disponibles, y la segunda es la vuelta de la recursión, que en la 'vuelta' continúa iterando las fichas que faltaban, y como en cada llamda se va quitando la ficha que se coloca, la iteración en cada llamada es sobre 1 ficha menos. Y por último el $n * m$ es la copia de la combinación máxima encontrada, quedandonos entonces

$$\begin{gathered} O(n * m) + O(\sum_{i=1}^{n*m} i) + O(n * m) \\ O(n * m) + O\left(\frac{n * m(n * m - 1)}{2}\right) + O(n * m) \\ O\left((n * m)^2\right) \end{gathered} \tag{10}$$

Entonces nos queda que el mejor caso es de complejidad cuadrática sobre la cantidad de casilleros del tablero (10) y el peor caso por acotado por $O(n * m)^{n*m}$ (9)

4.5. Comentarios y conclusiones

Dada la complejidad calculada (4.4), las mediciones realizadas (6.6) y el cálculo de la cantidad de combinaciones posibles (2), que la cantidad de operaciones que se realizan para chequear la gran cantidad de combinaciones es muy grande, aunque realizando la *poda* y descartando las combinaciones que no son válidas puede bajar considerablemente la cantidad de operaciones con determinadas entradas.

Se puede concluir que es un problema complejo que a no ser que se realize otro método en vez de buscar todas las posibilidades y sacar el máximo, con problemas de entrada pequeña, como puede ser un tablero de 5x5 (25 fichas), nos da que tendríamos un máximo de 29533262279404214911002168626 posibilidades para analizar y sacar el máximo.

5. Generación de casos de prueba

Para el testeo de los algoritmos y la medición de tiempos en función de la entrada, se programó una utilidad para generar los casos de prueba.

El programa recibe como parámetro el ejercicio del cual se quieren generar los casos, y las variables, como el tamaño de la entrada o en el ejercicio 1 la cantidad de días que estaba disponible el inspector.

Los números aleatorios que se generaron en los casos, se hizo con la función *random()* de C (http://linux.die.net/man/3/random) usando como semilla el tiempo en microsegundos.

Para el ejercicio 1, dada una cantidad n y la cantidad de días del inspector, genera n números aleatorioscorrespondiente a los camiones, éste valor se puede acotar para que vengan entre el día 0 y un día x - 1 como máximo quedándose con el resto de dividir el valor aleatorio de *random()* por x, y se utilizó x = n * 3 para que los camiones vengan en un rango de n * 3 dias. Para el ejercicio 2 se le indica la cantidad de piezas y el valor del tiempo y la pérdida es número aleatorio se le indica como máximo sea i + 20, siendo i el número del test a ejecutar. Y para el ejercicio 3, se le indica la cantidad de filas y columnas y de colores, y crea fichas con colores aeatorios.

Así se crearon muchos casos donde se fue incrementando el tamaño de la entrada y ejecutando varios tests de un mismo tamaño y varias veces el mismo test para obtener un promedio del tiempo que tarda en resolvero y descartar algunas imperfecciones que puedan surgir por el entorno donde se está midiendo los tiempos, como puede ser que justo se ejecute otra tarea.

6. Mediciones, tests y experimentos

6.1. Tests de correctitud

Para los ejercicios 1 y 2 se realizaron ademas de las mediciones, tests de correctitud, se detalla a continuación como fueron implementados:

- **Ejercicio 1:** Se programó un algoritmo mas simple en O(n^2) que asumimos como válido y se compararon todas las salidas de este algoritmo versus el algoritmo mas eficiente(y por lo tanto mas complejo).
- **Ejercicio 2:** Se constató que todas las salidas, estuvieran ordenadas según el criterio indicado en la demostración de correctitud del ejercicio 2.

6.2. Forma de medición

Con la ayuda de una macro creada especialmente para ser reutilizada en todas las porciones críticas de código que quisieramos medir, las mediciones fueron realizadas con la herramienta $high\_resolution\_clock$ provista por la libreria *chrono* de $C++$. Dicha macro provee una forma práctica de obtener la cantidad promedio de microsegundos consumidos por el código que se desea medir. Se le pasan como parámetros el código a ejecutar, la cantidad de veces que se repetirá el experimento para tomar el promedio y una referencia a la variable en la que se guardará el resultado de la medición.

6.3. Porciones críticas de código a medir de los ejercicios

Los ejercicios se basan en 3 secciones principales:

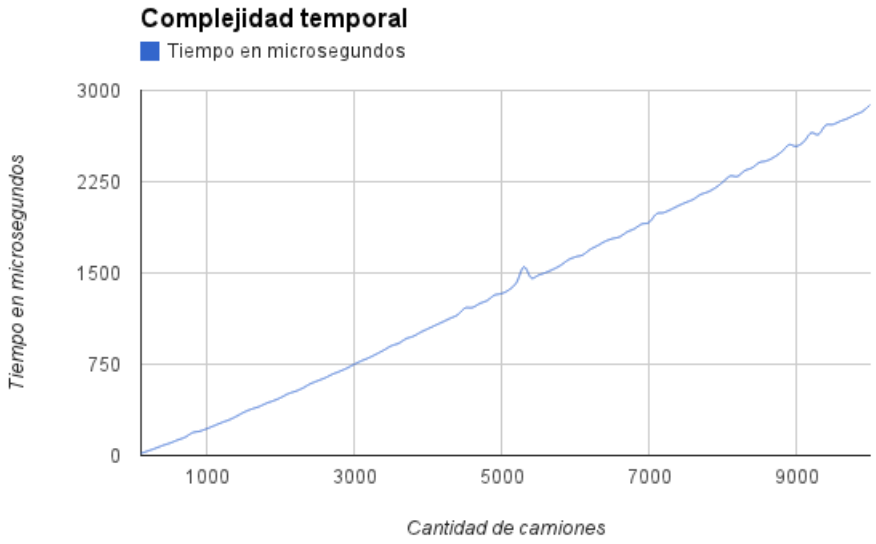
1. Lectura y decodificación de la entrada estandar.
2. Procesamiento y transformación de los datos de entrada para producir la salida
3. Codificación de la salida y escritura a salida estandar.

En particular lo que nos interesa medir a nosotros, es unicamente el inciso 2 de la lista anterior, no nos interesa tomar mediciones de entrada-salida, pues esto no es parte intrínseca del algoritmo. Tanto para los ejercicios 1 y 2 se realizaron 100 repeticiones para obtener el promedio de las mediciones y licuar posibles valores atípicos. Para el caso del ejercicio 3 este numero de repeticiones se vio disminuido a aproximadamente 10 repeticiones dada la complejidad mas alta del algoritmo.

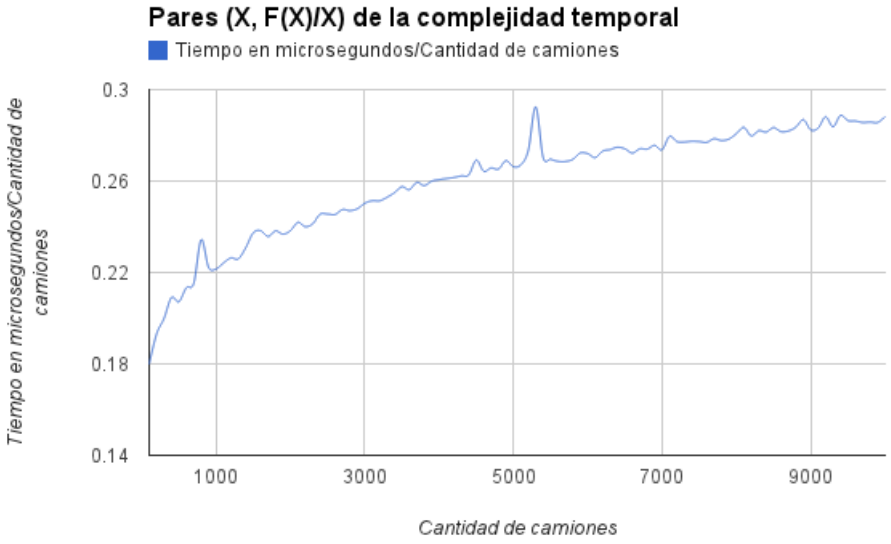
6.4. Ejercicio 1

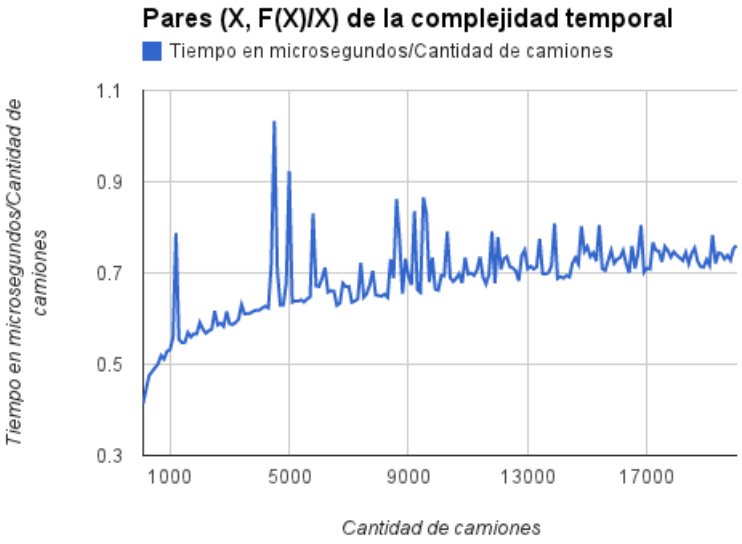
En este algoritmo, hemos determinado una cota de complejidad temporal asintótica teórica de O(n.log(n)).(Detalles en la sección complejidad de dicho ejercicio). La entrada ha sido expresado en n = #{ Cantidad de camiones } y sobre este numero se ha realizado la medición. Las mediciones de los experimentos que hemos realizado se condicen con esta cota teórica, a continuación se explicará como fue obtenida esta conclusión.

Aquí observamos un gráfico de los pares $(x, f(x))$ no demasiado claro respecto a que funcion representa, aunque parece lineal, mas que linearítmico, probamos variando la cantidad de elementos pero no obtuvimos mejora. Creemos se debe a que el logaritmo crece muy lentamente, lo cual se confirma mas adelante en el texto con nuevos experimentos.

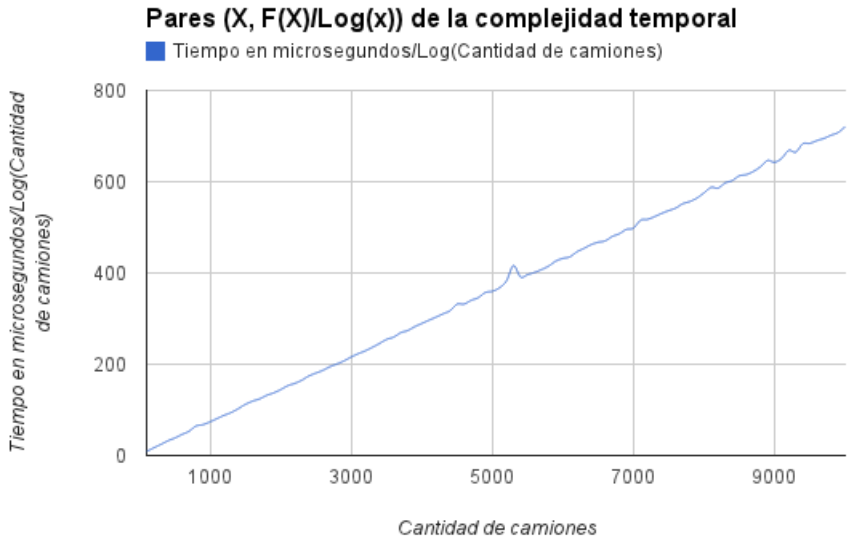


Luego del primer grafico obtenido, decidimos graficar los pares $(x, \frac{f(x)}{x})$ con el objetivo de obtener un gráfico consistente con una funcion logaritmica, el resultado es el esperado, se puede observar un comportamiento logarítmico en el crecimiento de la funcion graficada. Notemos ademas, la imagen de la funcion resultante, esta acotada en el intervalo $[0{,}18 \ldots 0{,}30]$, con lo cual, una funcion lineal multiplicada por valores en este intervalo, no se verá alterada en exceso, explicando el grafico original obtenido en la medición.

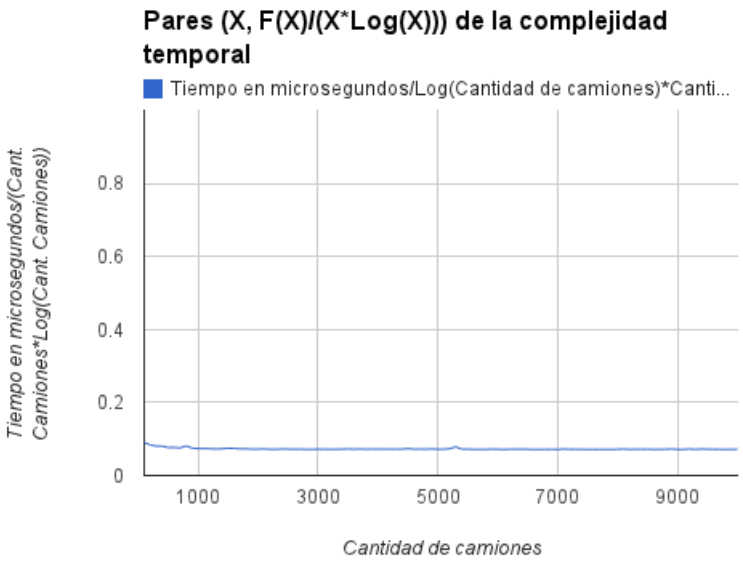




Como medida adicional, tambien graficamos los pares $(x, \frac{f(x)}{Log(x)})$ para tener una mayor certeza acerca de la funcion inicial. Podemos observar una funcion que es una escala de la función original, lo cual tiene sentido, dado el intervalo acotado del logaritmo que acompaña la funcion lineal multiplicando en la expresión **junto a una constante positiva menor que uno**. Probamos en dos casos, uno con una gran amplitud en en el intervalo de generacion de numeros aleatorios para los dias de la entrada, y otro con una amplitud más limitada, el resultado fue el mismo en todos los gráficos salvo en el logaritmico, lo cual puede explicarse por la busqueda binaria que se realiza sobre el conjunto de tuplas ordenadas (Dia, Cantidad de camiones hasta ese dia), lo cual tiene una complejidad O(k.logk) donde k = #{Cantidad de dias distintos en la entrada}



Finalmente graficamos los pares $(x, \frac{f(x)}{x*Log(x)})$ para constatar que se obtendría una constante, vamos que en realidad la función fluctua, pero dentro de un intervalo muy pequeño [0,07 . . . 0,09], lo cual consideramos es aceptable para asegurar que es una constante.



Como conclusión podemos asegurar que el algoritmo tiene realmente la complejidad teórica esperada.

6.4.1. Caso Promedio

Para el caso promedio simplemente probamos varias intancias generadas aleatoriamente

6.4.2. Mejor Caso

Luego de analizar nuestro algoritmo consideramos que el mejor caso posible se da cuando el vector de entrada posee todos sus elementos iguales, es decir, todos los camiones llegan el mismo día. El ordenamiento que realiza nuestro algoritmo al comenzar se llama *introsort*, de la librería de C++, que comienza con quicksort, cambia a heapsort cuando la cantidad de llamadas recursivas excede $2 * log_2(n)$. En un array de elementos iguales, la perfomance del Quicksort va a ser $(nlogn)$, ya que el pivote que va a tomar en cada una de las iteraciones va a ser exactamente la mediana, causando que los dos índices del ciclo se unan exactamente en el medio, dividiendo los dos subproblemas en exactamente la mitad de tamano.

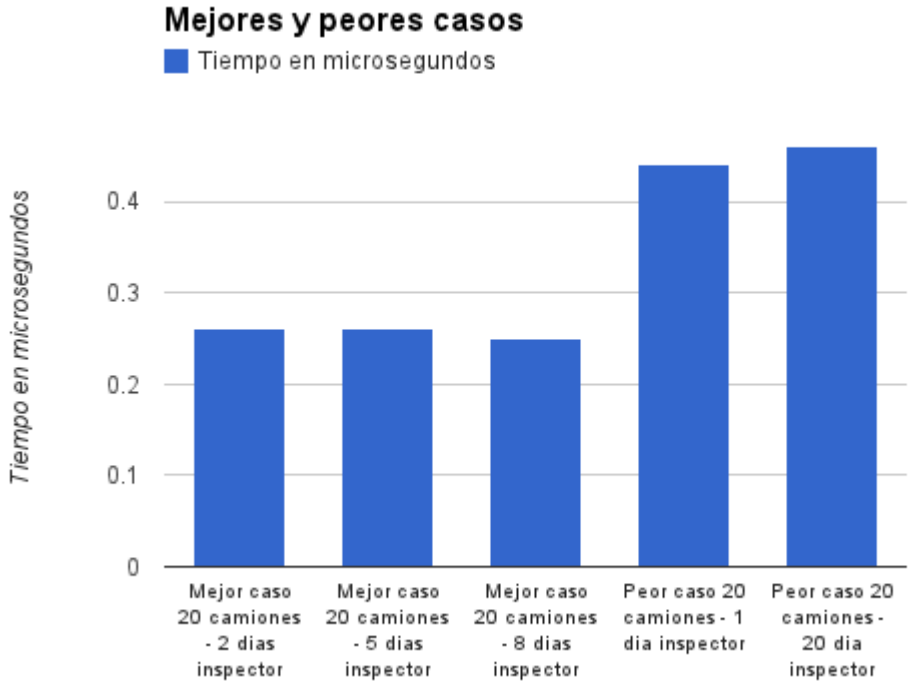
Vemos drásticamente una mejora en la perfomance en la segunda parte del algoritmo. En la función que agrupa y acumula, al ser todos los elementos idénticos, el vector tablaDiaCantidad va a tener un solo elemento, la tupla $< unicoelemento, longitud\_entrada >$. Por lo cual el ciclo principal del algoritmo va a realizar sólo una iteración hasta encontrar el intervalo óptimo, que es el único posible, el que comienza con el único día en que llegan todos los camiones.

6.4.3. Peor Caso

No pudimos encontrar la implementación del introsort, por lo cual no sabemos qué' método utiliza para tomar el pivote. Por lo cual no sabemos cuál es el peor caso de este ordenamiento. De todas formas al tener complejidad $O(nlogn)$ para cualquier caso posible, la disposición de los enteros del vector de entrada no influye demasiado. Busquemos peor caso de la segunda parte del algoritmo. Acto seguido del ordenamiento nuestro algoritmo recorre 1 vez el vector de entrada agrupando repetidos y acumulando. Esto es siempre linea, va a iterar una vez por cada elemento siempre, pero el peor caso posible del algoritmo debe tener todos los días de entrada distintos, así esta reducción efectivamente no reduce nada, y el vector resultante tablaDiaCantidad sigue siendo de longitud n.

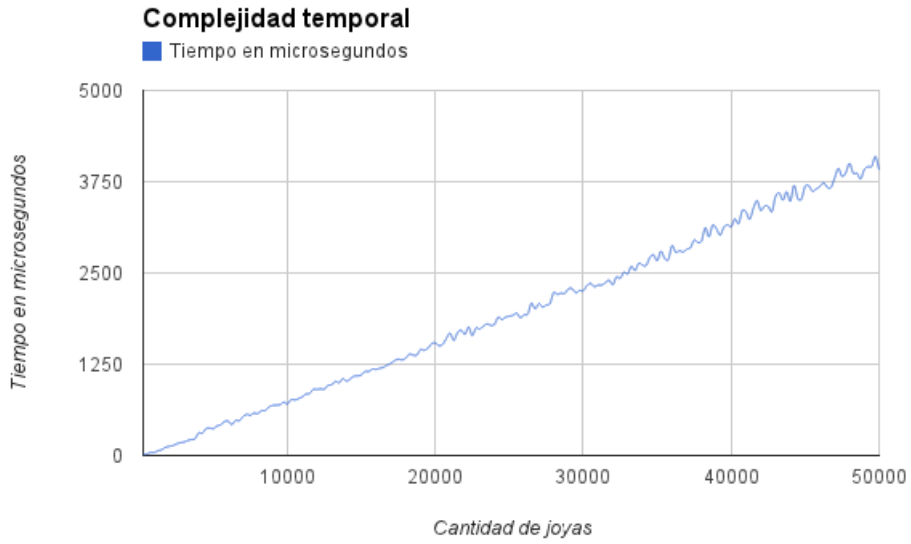
El ciclo principal de nuestro algoritmo recorre una vez el vector tablaDiaCantidad. Por cada iteración realiza una búsqueda binaria. El peor caso de la búsqueda binaria es cuando el número a buscar no se encuentra en el arreglo. Con lo cual, para cada iteración, si para cada día *inicio*, el día $inicio + D - 1$ no se encuentra en el arreglo nos aseguraremos de que la búsqueda realiza la mayor cantidad de pasos posibles. Además, si $D = 1$, el día a buscar queda en el extremo izquierdo del intervalo a buscar, y

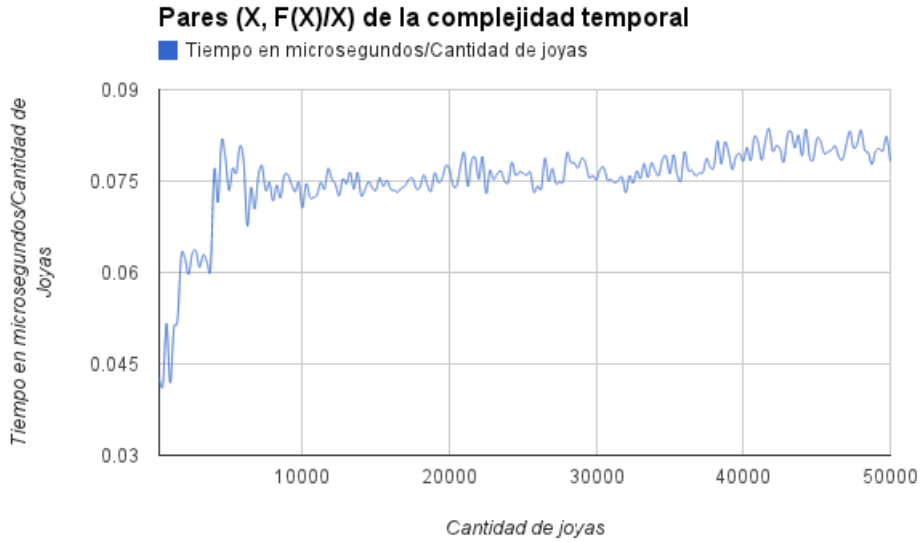
si $D = n$, queda siempre en el extremo derecho, lo cual también son los peores casos de la búsqueda binaria. Por lo cual vamos a testear dos tipos de casos. En ambos la la distribución de días es uniforme, en el primer tipo $D = 1$ y para cada día *inicio*, el día $inicio + D - 1$ nunca está. En el segundo tipo de caso D va a ser igual a n, con lo cual $inicio + D - 1$ siempre va a caer afuera del arreglo.

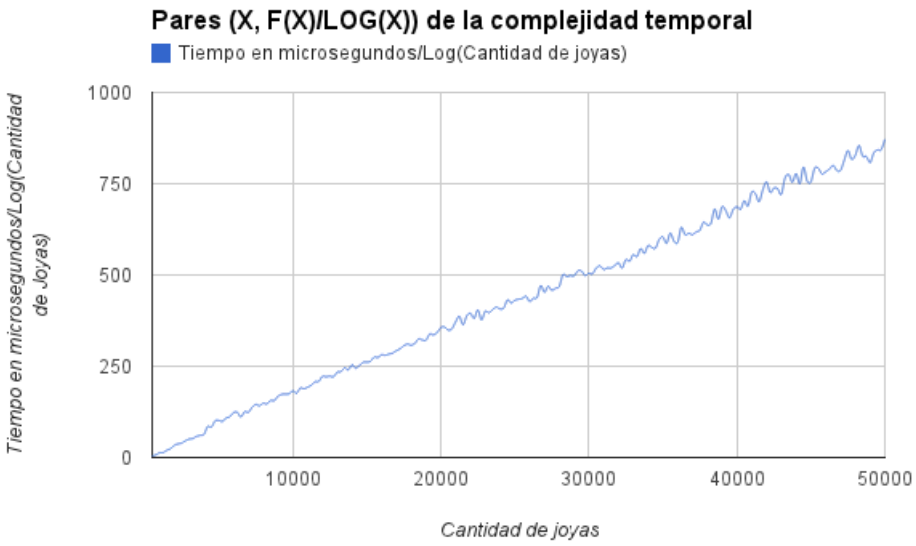


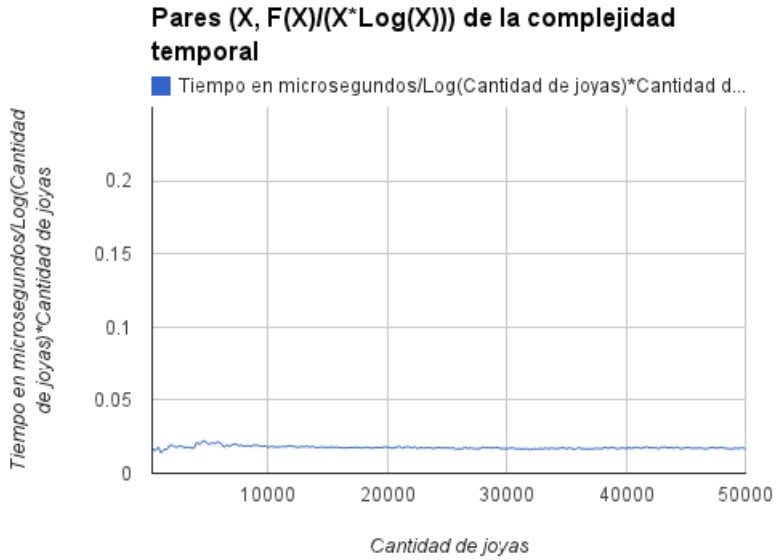
6.5. Ejercicio 2

Para el algoritmo del ejercicio 2, tambien se observó una complejidad asintótica teórica O(n.log(n)). La medición de la entrada fue sobre n = #{ Cantidad de joyas pendientes }. Se realizó un razonamiento análogo al ejercicio anterior para constatar con seguridad que se condicen las mediciones empíricas con la cota teórica. A continuación se indican las mediciones, nuevamente graficamos la funcion original obtenida de las mediciones, dividimos sucesivamente por n, logn y finalmente nlogn obteniendo nuevamente las mismas conclusiones que en el ejercicio anterior, aunque esta vez, la función logaritmica obtenida en los pares $(x, \frac{f(x)}{x})$, es mas irregular y la funcion constante esta mas acotada dentro de un intervalo mas ajustado. Finalmente este ejercicio tambien posee una complejidad que condice con la hallada teoricamente.
Podemos indicar además que el peor y mejor caso de este algoritmo depende fuertemente de la implementación del algoritmo de ordenamiento de la librería de $C++$ ya que el resto del algoritmo es lineal y muy simple como para interferir con casos particulares.


Complejidad temporal
Tiempo en microsegundos
5000
3750
2500
1250
0
Tiempo en microsegundos
10000
20000
30000
40000
50000
Cantidad de joyas


Pares (X, F(X)/X) de la complejidad temporal
Tiempo en microsegundos/Cantidad de joyas
0.09
0.075
0.06
0.045
0.03
Tiempo en microsegundos/Cantidad de Joyas
10000
20000
30000
40000
50000
Cantidad de joyas





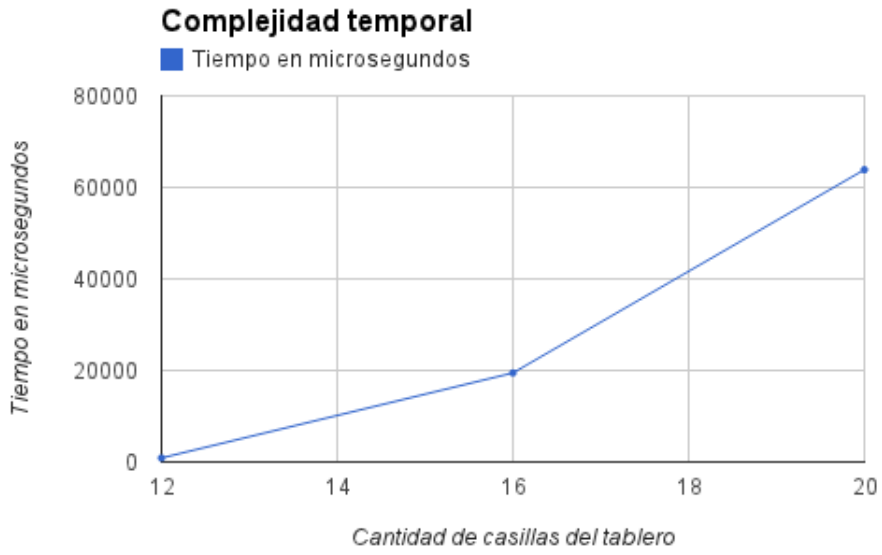
6.6. Ejercicio 3

Con el caso del ejercicio 3, como la complejidad es muy grande, no se pudieron realizar muchos casos de prueba con valores grandes de la entrada.

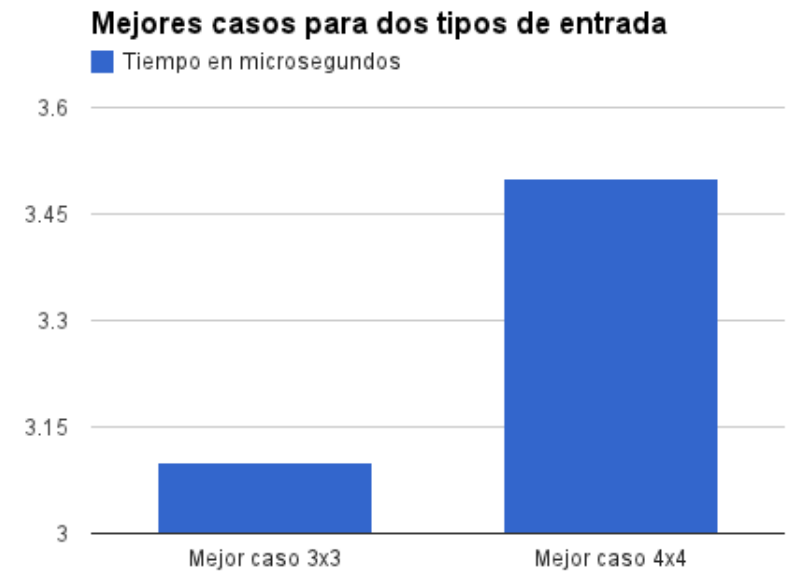
Se planteó como mejor caso el caso en que todas las fichas estén ordenadas de la forma que deben ser colocadas (o bien que todas las fichas tengan todos los lados del mismo color), por lo que encuentra el máximo en $n * m$ como se explicó en la sección 4.4, y se testeó dicho caso.

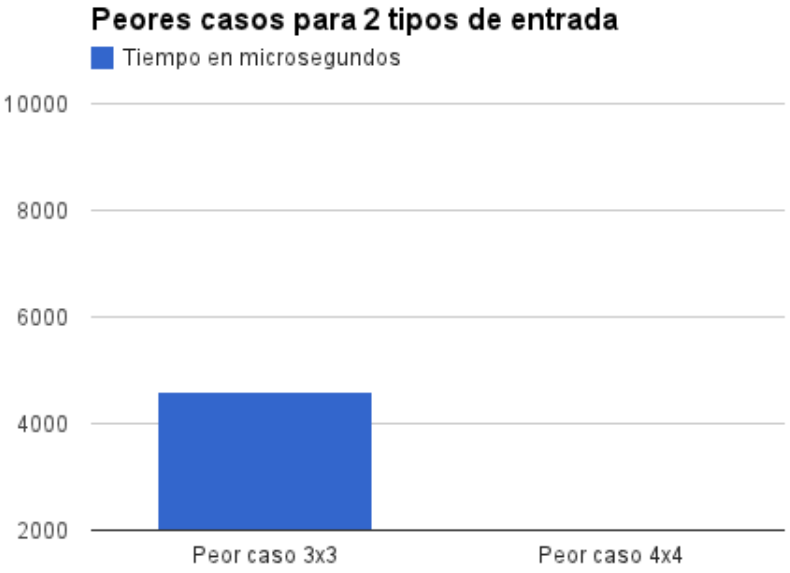
También se planteó como peor caso el que la poda no se realize y que pruebe la mayor cantidad de combinaciones posibles. Para lograr la mayor cantidad de combinaciones válidas todas las fichas deben poder colocarse de todas las maneras posibles entre sí, pero si ésto ocurriese, sería el mejor caso porque encuentra el máximo valor posible de fichas a colocar en la primera iteración, entonces para minimizar la cantidad de combinaciones que se descartan, se puede poner que sólo una ficha no se pueda poner junto con las otras, por ejemplo, que todas las fichas sean totalmente verdes menos 1 que sea roja. en éste caso el primer tablero que encuentre tendrá $n * m - 1$ fichas colocadas, por la poda, todas las demas combinaciones que tengan 1 o más casillero vacíos serán descartadas, por lo que tenemos en total $(n * m - 1)!$ combinaciones a probar.

Gráfico con el tamanño de la entrada siendo la cantidad de casilleros del tablero, y el tiempo en microsegundos, se separó en 2 gráficos para que se aprecien los valores con entrada más chica:

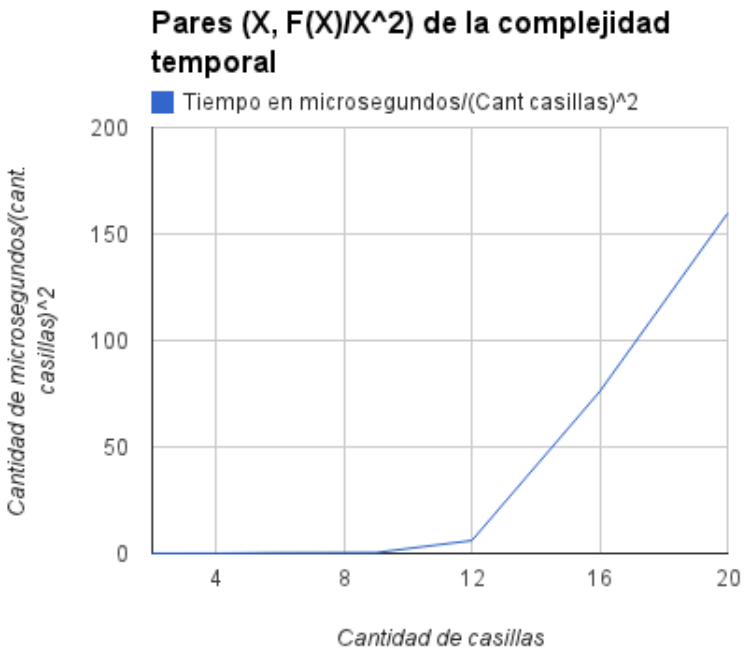


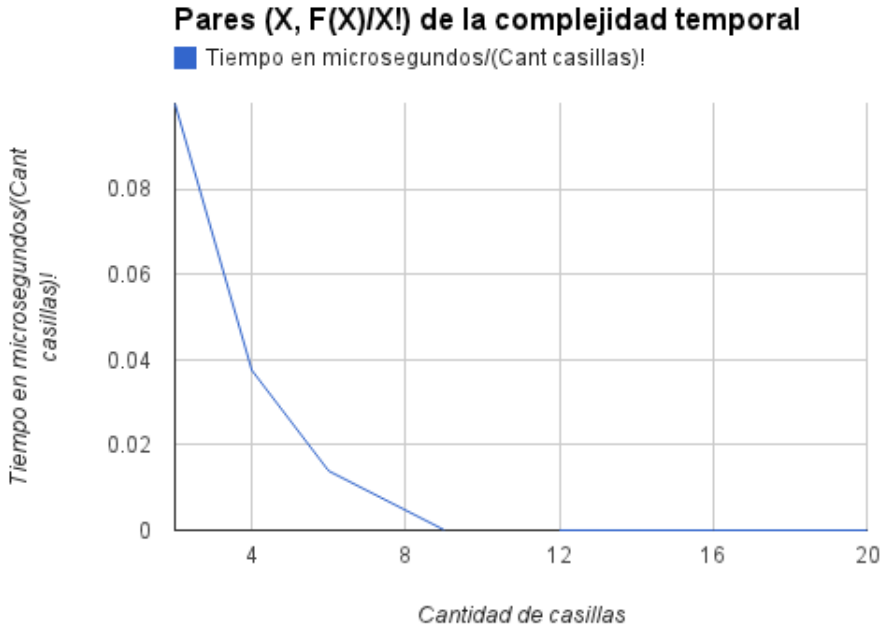
Luego se midieron los peores y mejores caso para tableros de 3x3 y 4x4 casilleros, para el peor caso de 4x4 cortamos la ejecución porque tomó mas de una hora y nos parecio demasiado.





Al no poder correr una cantidad muy variada de casos con diferentes tamaños de la entrada, no podemos asegurar mucho sobre los gráficos, de todas formas al tiempo que tardó cada caso, se lo dividió por $(n * m)^2$ y por $(n * m)!$ para ver si la curva se llevaba a una recta





Como se ve, al dividir por la cuadrática al menos para los primeros valores la curva aún está creciendo, con lo cual esta no es una cota válida.Asimismo cuando se divide por el factorial la curva decrece y tiende a cero, y dado que la funcion original de complejidad es creciente, podemos asegurar que la complejidad está acotada por debajo por O($(n * m)^2$) y por encima por O((n*m)!) .

7. Apéndice: Código fuente relevante

El lenguaje elegido para la resolución de los problemas fue $C++$ dada su alta velocidad de la ejecución respecto a otros lenguajes como Java y el manejo directo de memoria a bajo nivel nos liberan de posibles problemas de mediciones que podría traernos por ejemplo el garbage collector.

7.1. Ejercicio 1

```
/**
    PRE:    (diasInspector>0)

    POST:   res_1 = dia inicial conveniente de contratacion
            res_2 = cantidad de camiones capturados para revision en dicho periodo

    COMPL:  O(n.log(n))
**/
pair<int, int> solveInstance(int diasInspector, vector<int> entrada){
    //Ordeno los dias de la entrada de menor a mayor
    //la STL me garaniza que este sorting es O(n.log(n))
    sort(entrada.begin(), entrada.end());

    //agrupar repetidos y armar tuplas (dia, suma de camiones acumulada hasta ese dia inclusive
        )
    //Sea n = |entrada|
    vector<pair<int, int>> tablaDiaCantidad;//O(1)
    int longuitud = entrada.size();//O(1)
    tablaDiaCantidad.reserve(longuitud);//O(n)
    int i=0;//O(1)
    int apariciones = 0;//O(1)
    while(i < longuitud) {//O(n)
        apariciones++;//O(1)
        if((entrada[i] != entrada[i+1]) || (i==longuitud-1)){
            pair<int, int> par = make_pair(entrada[i], apariciones);//O(1)
            tablaDiaCantidad.push_back(par);//O(1) - no pueden ocurrir
            //realocamientos pues reserve  n elementos, y en el peor caso,
            //no hay dias repetidos y quedan n entradas en la tabla esta.
        }
        i++;//O(1)
    }

    //buscar ventanas entre las cuales revisa el inspector computando cuantos camiones
    //revisa y tomando la ventana que mas camiones revise
    int maximo=-1;
    int idxComienzoIntervaloOptimo=-1;
    for (uint i = 0; i < tablaDiaCantidad.size(); ++i)
    {
        pair<int, int> target = tablaDiaCantidad[i];
        target.first += (diasInspector-1);
        //la busqueda se reduce al intervalo [i..n] pues como D>0, y tablaDiaCantidad esta
            ordenada
        //por dia, i-esimo dia + D va a estar mas adelante en el arreglo que i-esimo dia.
        vector<pair<int, int> >::iterator upperBound = upper_bound (tablaDiaCantidad.begin() +
            i,
        tablaDiaCantidad.end(), target.first, compare_pair_number);//O(log n)

        tablaDiaCantidad.end(), target.first, compare_pair_number);//O(log n)
        pair<int, int> res = tablaDiaCantidad[(upperBound - tablaDiaCantidad.begin() - 1)];

        int resta=0;
        //supongo que vino el primer dia. no se le resta nada a la suma acumulada
        if(i>0){//si no vino el primer dia, corrijo la resta
            //asumo i>0(no puede ser negativo)
            resta = tablaDiaCantidad[i-1].second;
        }
        int cantCamionesDelIntervalo = (res.second-resta);
        if(cantCamionesDelIntervalo>maximo){
            maximo=cantCamionesDelIntervalo;
            idxComienzoIntervaloOptimo = tablaDiaCantidad[i].first;
        }
    }
    return make_pair(idxComienzoIntervaloOptimo, maximo);
}
// ------------------------------------------------------------------------------------ //

```

```
64 /**
65     PRE:     True
66 
67     POST:    (n < p_1)
68 
69     COMPL:   O(1)
70 **/
71 bool compare_pair_number(int n, pair<int, int> const& p)
72 {
73     return (n < p.first);
74 }
```

7.2. Ejercicio 2

```
bool compare_function (tuple<int, int, int> e1, tuple<int, int, int> e2){
    //tupla < depreciacion, tiempo, posicion en la entrada >
    return (get<1>(e1)*get<0>(e2)) > (get<0>(e1)*get<1>(e2));
}

void solveInstance(vector<tuple<int, int, int>>& piezas,
    uint cantidadPiezas, uint& totalPerdido){
    sort(piezas.begin(), piezas.end(), compare_function); // ordenar segun ti/di
    // recorro el vector de adelante para atras para hacer la acumulacion de perdidas
    int b = cantidadPiezas - 1;
    //suma de las depreciaciones a sumar
    uint depreciaciones = 0;
    /*
        Para un calculo de 3 piezas seria = t1*(d1+d2+d3) + t2*(d2+d3) +t3*(d3).
        En depreciaciones voy acumulando los "d"
    */
    while(b >= 0)
    {
        depreciaciones = depreciaciones + get<0>(piezas[b]);
        totalPerdido = totalPerdido + get<1>(piezas[b])*depreciaciones;
        b--;
    }
}
```

7.3. Ejercicio 3

```c
typedef struct Ficha {
    int numero; //-1 = espacio vacio
    int arriba;
    int abajo;
    int izquierda;
    int derecha;
} Ficha;

struct Lista {
    void *nodo;
    struct Lista *anterior;
    struct Lista *siguiente;
};

typedef struct Lista Lista;

Lista *agregar_a_lista(Lista *l, Lista *nuevo)
{
    if(!l){
        return nuevo;
    }

    nuevo->anterior = l;
    nuevo->siguiente = l->siguiente;
    if(l->siguiente){
        l->siguiente->anterior = nuevo;
    }
    l->siguiente = nuevo;
    return nuevo;
}

Lista *agregar_a_lista_atras(Lista *l, Lista *nuevo)
{
    if(!l){
        return nuevo;
    }

    nuevo->anterior = l->anterior;
    nuevo->siguiente = l;
    if(l->anterior){
        l->anterior->siguiente = nuevo;
    }
    l->anterior = nuevo;
    return nuevo;
}

Lista *obtener_siguiente(Lista *l)
{
    if(!l)
        return NULL;

    return l->siguiente;
}

Lista *obtener_anterior(Lista *l)
{
    if(!l)
        return NULL;

    return l->anterior;
}

Lista *sacar_elemento_de_lista(Lista *l)
{
    if(!l)
        return NULL;
    if(l->anterior){
        l->anterior->siguiente = l->siguiente;
    }
    if(l->siguiente){
        l->siguiente->anterior = l->anterior;
    }
    return l;
}

```

```
76 void *obtener_nodo(Lista *l)
77 {
78     if(!l)
79         return NULL;
80     return l->nodo;
81 }
82
83 void copiar(Ficha ***out, Ficha ***in, int n, int m)
84 {
85     int i, j;
86     for(j = 0; j < m; j++){
87         for(i = 0; i < n; i++){
88             out[i][j] = in[i][j];
89         }
90     }
91 }
92
93 int valida_colocar(Ficha *ficha, Ficha ***tablero, int i, int j)
94 {
95     if(ficha->numero >= 0){ //si es < 0 significa que se quiere dejar el casillero vacio, por
          lo que siempre es valido de colocar
96         if(j > 0){ //si no estamos en la primer columna, entonces chequeo el color izquierdo de
               la ficha
97             if(tablero[i][j - 1]->numero >= 0){
98                 if(tablero[i][j - 1]->derecha != ficha->izquierda){
99                     return 0;
100                }
101            }
102        }
103        if(i > 0){ //si no estamos en la primera fila, chequeo el color de arriba de la ficha
104            if(tablero[i - 1][j]->numero >= 0){
105                if(tablero[i - 1][j]->abajo != ficha->arriba){
106                    return 0;
107                }
108            }
109        }
110    }
111    return 1;
112 }
113
114 void resolver(Ficha ***tablero_optimo, int *fichas_maxima, Ficha ***tablero, int n_tablero, int
       m_tablero, Ficha *fichas, Lista *fichas_disponibles, int i_t, int j_t, int
      casilleros_tablero, int fichas_en_tablero)
115 {
116     int proximo_i, proximo_j;
117     int total_casilleros;
118     total_casilleros = n_tablero * m_tablero;
119     Lista *lista, *lista_anterior = NULL, *lista_siguiente = NULL;
120     Ficha *ficha;
121
122     if(casilleros_tablero == total_casilleros){
123         if(fichas_en_tablero > *fichas_maxima){ //solo si supera la cantidad maxima de fichas
              encontradas entonces reemplazo el tablero viejo con el nuevo
124             *fichas_maxima = fichas_en_tablero;
125             copiar(tablero_optimo, tablero, n_tablero, m_tablero); //O(n_tablero * m_tablero)
126         }
127         return;
128     }
129     if(total_casilleros - casilleros_tablero + fichas_en_tablero <= *fichas_maxima){ //la poda
          para cortar con la recursion si ya se calcula que no podra superar al tablero_optimo
130         return;
131     }
132     proximo_i = i_t; //calcula cual es la siguiente posicion que llama a la recursion
133     proximo_j = j_t;
134     if(j_t == m_tablero - 1){
135         proximo_j = 0;
136         proximo_i++;
137     }
138     else{
139         proximo_j++;
140     }
141     for(lista = fichas_disponibles; lista != NULL; lista = obtener_siguiente(lista)){ //(O(
          longitud de la lista)
142         ficha = (Ficha *)obtener_nodo(lista); //O(1)
143         if(valida_colocar(ficha, tablero, i_t, j_t)){ //O(1)
144             tablero[i_t][j_t] = ficha;
```

```
145             if(ficha->numero >= 0){ //si el numero es >= 0, es una ficha con colores, por lo
                    que la tengo que quitar de la listaa de fichas disponibles al llamar a la
                    recursion
146                 lista_siguiente = obtener_siguiente(lista); //O(1)
147                 lista_anterior = obtener_anterior(lista); //O(1)
148                 sacar_elemento_de_lista(lista); //O(1)
149                 fichas_en_tablero++;
150             }
151             else //si el numero es < 0, significa que se llego a la ultima 'ficha', que
                    representa dejar el casillero vacio, la cual siempre esta disponible y no hay
                    que quitarla de disponibilidad
152                 lista_anterior = fichas_disponibles; //le paso la lista entera de fichas, desde
                        el comienzo
153             resolver(tablero_optimo, fichas_maxima, tablero, n_tablero, m_tablero, fichas, !
                    lista_anterior ? lista_siguiente : fichas_disponibles, proximo_i, proximo_j,
                    casilleros_tablero + 1, fichas_en_tablero);
154             if(ficha->numero >= 0){ //si se saco la ficha de la lista de disponible, entonces
                    la tengo que volver a agregar en el mismo lugar donde estaba
155                 if(lista_anterior){ //si el elemento que saque tenia un anterior, se lo agrego
                        despues del anterior
156                     lista = agregar_a_lista(lista_anterior, lista); //O(1)
157                 }
158                 else if(lista_siguiente){ //sino (si saque el primer elemento), lo agrego atras
                        del siguiente elemento del que saque
159                     lista = agregar_a_lista_atras(lista_siguiente, lista); //O(1)
160                 }
161                 fichas_en_tablero--;
162             }
163         }
164     }
165 }
```

8. Apéndice: Entregable e instrucciones de compilacion y testing

8.1. Estructura de directorios

En esta sección se explicará brevemente como esta organizado el codigo, los tests, los resultados y el informe en la estructura en la carpeta. La estructura de directorios es la siguiente:

- **Raiz:** Aquí se encuentra el Makefile que engloba la compilación y generación de archivos necesaria para el tp.
- **ej1:** Contiene el código fuente del ejercicio correspondiente junto a su Makefile, estos cuentan con targets 'all' y 'clean' para compilación individual de este módulo.
- **ej2:** Contiene el código fuente del ejercicio correspondiente junto a su Makefile, estos cuentan con targets 'all' y 'clean' para compilación individual de este módulo.
- **ej3:** Contiene el código fuente del ejercicio correspondiente junto a su Makefile, estos cuentan con targets 'all' y 'clean' para compilación individual de este módulo.
- **common**:
  - **common/plotting:** Contiene scripts y código en python para realizar gráficos automatizados sobre los resultados. **Estos gráficos podrían no ser correctos pues se abandonó el uso de estos luego de decidirnos a utilizar graficos de google docs por cuestiones de mejor integración para trabajar en equipo. La documentación de la generación de gráficos es meramente informativa.**
  - **common/random_testcase_builder**: Aquí se encuentra una aplicación capaz de generar entradas aleatorias usando un generador de numeros aleatorios con distribución uniforme para los 3 ejercicios del trabajo respetando el formato pedido en el enunciado.
  - **common/testcases:** Aqui se encuentran los casos de prueba aleatorios generados por el programa automatico de generación de tests, para cada ejercicio, y además sus salidas asociadas y su tiempo insumido en microsegundos, acompañado por el numero de repeticiones utilizado para calcular el promedio de las ejecuciones.
  - **common/testing:** En esta carpeta se encuentran los scripts encargados de generar los casos de prueba y ejecutarlos, en el script de ejecución además se realiza una transformacion y intercalación de los datos de los tiempos consumidos para generar un archivo plot.out conteniendo en cada linea descripcion sobre los parametros de entrada, tiempo consumido y cantidad de repeticiones para el cálculo del promedio.
  - **common/timing:** En esta carpeta se encuentra la macro para tomar tiempos utilizada para la experimentación.

8.2. Compilación

Basta ejecutar make <target>sobre el directorio raíz eligiendo alguno de los targets mencionados a continuación.

- **clean:** Elimina todos los binarios, el informe compilado, archivos auxiliares, los tests aleatorios y sus resultados y los gráficos generados.
- **all:** Compila todos los ejercicios, el generador de casos aleatorios, crea los tests aleatorios, y compila el informe en latex a pdf.
- **run-tests:** Depende de all y luego ejecuta todos los tests aleatorios generados.
- **graphics:** Depende de run-tests y luego genera gráficos sobre los resultados. **Estos gráficos podrían no ser correctos pues se abandonó el uso de estos luego de decidirnos a utilizar graficos de google docs por cuestiones de mejor integración para trabajar en equipo. La documentación de la generación de gráficos es meramente informativa.**