DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Técnicas de Diseño de Algoritmos

Viernes 11 de Abril de 2014

Algoritmos y Estructuras de Datos III
Entrega de TP

Grupo %NUMERO_DE_GRUPO%

Integrante	LU	Correo electrónico
Barrios, Leandro E.	404/11	ezequiel.barrios@gmail.com
Benegas, Gonzalo	958/12	gsbenegas@gmail.com
Duarte, Miguel	904/11	miguelfeliped@gmail.com
Niikado, Marina	711/07	mariniiik@yahoo.com.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

1.1. Objetivos

Mediante la realización de este trabajo práctico se pretende realizar una introducción a la implementación y el análisis de las técnicas algorítmicas básicas para resolución de problemas.

Se analizan las técnicas de *algoritmos golosos*, y de *backtracking*.

1.2. Pautas de trabajo

Se brindan tres problemas, escritos en términos coloquiales, en donde para cada uno de ellos se requiere **encontrar un algoritmo** que brinde una **solución particular**, acotado por una determinada **complejidad temporal**. El algoritmo debe ser **implementado** en un lenguaje de programación a elección. Los datos son proporcionados y deben ser devueltos bajo formatos específicos de *input* y de *output*.

Posteriormente se deben realizar análisis teóricos y empíricos tanto de la de **correctitud** como de la **complejidad temporal** para cada una de las soluciones propuestas.

1.3. Metodología utilizada

Para cada ejercicio, se brinda primeramente una **descripción** del problema planteado, a partir de la cual se realiza una **abstracción** hacia un **modelo formal**, que permite tener un **entendimiento preciso** de las pautas requeridas.

Se expone, cuando las hay, una **enumeración de las características** elementales del problema; estas son aquellas que permiten **encuadrarlo** dentro de una **familia de problemas** típicos.

Se desarrolla posteriormente un análisis del conjunto **universo de posibles soluciones**, caracterizando matemáticamente el concepto de **solución correcta** y, en los casos en que se solicita optimización, las condiciones que definen a la **solución particular** (o *mejor solución*) que se encuadra dentro de las pretenciones del problema.

Luego de caracterizar para todo conjunto posible de entradas «*cómo se compone el conjunto solución*» correspondiente, se desarrolla un **pseudocódigo** en el que se expone «*cómo llegar a ese conjunto*»[1].

Habiendo planteado la **hipótesis de resolución** se demuestra, de manera informal o mediante inferencias matemáticas según sea necesario, que el **algoritmo propuesto** realmente permite obtener la **solución correcta**[2].

Después de demostrar la **correctitud de la solución**, y su **optimalidad** en caso de existir varias soluciones correctas, se realiza un análisis teórico de la **complejidad temporal** en donde se estima el comportamiento del algoritmo en términos de tiempo. Este análisis en particular se realiza con el objetivo de obtener una *cota superior asintótica*[3].

Luego de calcular la **cota de complejidad temporal**, se realiza una **verificación** empírica, junto con una **exposición gráfica** de los resultados obtenidos, mediante la combinación de técnicas básicas de medición y análisis de datos.

[1] Una explicación coloquial, obviando detalles puramente implementativos: arquitectura, lenguaje, etc.

[2] En caso de existir más de una solución correcta, se demuestra que el algoritmo obtiene al menos una de ellas o, dicho de otro modo, para problemas de optimización, se demuestra que ninguna del resto de las soluciones correctas es mejor que la solución propuesta por nuestro algoritmo

[3] Aunque se mencionan, sobre todo en el caso del *Algoritmo de Backtracking*, algunas «familias de entrada» particulares bajo las cuales el algoritmo propuesto presenta un comportamiento mucho mejor al peor caso.

1.4. Herramientas utilizadas

Para la realización de este trabajo se utilizaron un conjunto de herramientas, las cuales se enumeran a continuación:

- `C++` como lenguaje de programación
  - `gcc` como compilador de C++
- `python` y `bash` para la realización de scripts
  - `python` para generar casos de prueba
  - `bash` para automatizar las mediciones
  - `python/matplotlib` para plotear los gráficos
- LaTeX para la redacción de este documento
- Se testeó Sistemas Operativos
  - `Debian GNU/Linux`
  - `Ubuntu`
  - `FreeBSD`, compilando a través de `gmake`
  - `Windows`, a través de `cygwin`

2. Desarrollo del TP

2.1. Problema 1: Camiones sospechosos

2.1.1. Descripción

desea contratar a un experto por D días consecutivos para que éste pueda detectar si algún camión de la

El inspector `Rick A. Lapazta`, legendario[4] empleado un puesto de control de camiones, es encomendado por la diosa Mor`fea` durante un revelador sueño a la tarea de investigar si la empresa *il Raviolin* está transportando `sustancias ilegales`. Luego de meses de investigación, y de arriesgar su pasiva e inspeccionista vida intrusionando en el archivo secreto de la empresa descubre que realmente cuenta con muy poca información: dispone de una lista en la anotó los datos importantes de cada camión, la cantidad de antenas de cada uno de ellos, y el día en que atravesarán el control. El inspector deduce posteriormente que la mayor o menor cantidad de antenas de un camión no debe ser un factor influyente al momento de transportar este tipo de sustancias, y que la mejor solución a sus problemas es contratar a un *experto*.

Como dispone de una cantidad acotada de dinero, y sabe que el tercerizado le cobrará una cantidad fija de dinero por cada día de trabajo el inspector busca, dado un rango de días de inspección, maximizar la cantidad de camiones inspeccionados, es decir que la mayor cantidad de camiones tienen que estar pasando por el puesto durante el período de contración del experto. El inspector conoce los días en que cada uno de los n camiones de la empresa estará pasando, pero desgraciadamente esta información puede o no estar dada en orden cronológico, nadie lo sabe. El inspector olvidó mencionarlo, pero el experto es de otro país, el reino de muy muy lejano, y la legislación de turismo de la zona es muy dura, por lo que este sólo dispone de permiso realizar un viaje al puesto de inspección.

El problema encomienda encontrar una solución al dilema del inspector, en donde recibiendo como **datos de entrada** la cantidad de días (D), la cantidad de camiones (n) y los días en que estos llegarán (d_1,d_2,...,d_n) se deberá idear un algoritmo que lo resuelva con una complejidad **estrictamente mejor que** $\mathcal{O}(n^2)$, siendo n la cantidad total de camiones. El resultado que se deberá obtener es el día inicial d_i en que convendría contratar al experto, y el total c de camiones que serán inspeccionados por el mismo.

Formato de entrada:

$$D\ n\ d1\ d2\ ...\ dn$$

Formato de salida:

$$d_i\ c$$

Si se contrata al experto por $D = 3$ días consecutivos y en total hay $n = 20$ camiones sospechosos $(c1, c2, ..., c20)$ que pasan por el puesto según se muestra a continuación:

Ejemplo 2.1.1.1.

```
Entrada:
3 20 10 10 9 4 1 1 1 1 1 2 6 6 6 4 5 5 5 5 5 9
```

[4]La leyenda dice que junto a su compañero Zepas A. Poraki, son invictos en no hacer muy bien su trabajo

Día	1	2	3	4	5	6	7	8	9	10
Camiones	$c5$	$c10$	$c14$	$c4$	$c15$	$c11$			$c3$	$c1$
	$c6$				$c16$	$c12$			$c20$	$c2$
	$c7$				$c17$	$c13$				
	$c8$				$c18$					
	$c9$				$c19$					

El primer día del período de contratación será $d = 4$ y habrán sido inspeccionados, durante los días `Día4`, `Día5` y `Día6`, un total de $c = 9$ camiones.

```
Salida:
4 9
```

Ejemplo 2.1.1.2.

```
Entrada:
3 20 1 2 7 1 1 1 1 1 1 1 7 7 7 1 4 4 4 4 4 7
```

Día	1	2	3	4	5	6	7
Camiones	$c5$			$c15$			$c11$
	$c6$			$c16$			$c12$
	$c7$			$c17$			$c13$
	$c8$			$c18$			$c3$
	$c9$			$c19$			$c20$
	$c10$						
	$c14$						
	$c4$						
	$c1$						
	$c2$						

Para este caso, el primer día del período de contratación será $d = 1$ y habrán sido inspeccionados, durante los días `Día1`, `Día2` y `Día3`, un total de $c = 10$ camiones.

```
Salida:
1 10
```

2.1.2. Hipótesis de resolución

Nos interesa saber cuál es el mejor día entre d_1 y d_n para contratar al experto. Desde el día elegido hasta el día $D - 1$, buscamos que la cantidad de camiones sea máxima. Primero, entonces, ordenamos en forma creciente el conjunto d_1,...,d_n dado. Sabemos que la solución pertenece a este conjunto. Tenemos a $P(x)$ como la suma de la cantidad de camiones que pasan desde el día x hasta el día $x + D - 1$. Nuestro algoritmo itera sobre cada elemento d_i del conjunto calculando $\mathrm{P}(d_i)$, y guardando el elemento e con $\mathrm{P}(e)$ máximo encontrado hasta el momento. Al final de las iteraciones, tenemos guardado el elemento e con $\mathrm{P}(e)$ máximo en todo el conjunto. Devolvemos e y $\mathrm{P}(e)$.

2.1.3. Justificación formal de correctitud

Consideremos los días de llegada ordenados en forma creciente: $d_1 \leq d_2 \leq \cdots \leq d_n$ son enteros positivos al igual que D.
Definimos $\overline{\mathrm{P(x)}}$, donde $x \in \mathbb{N}$, como $\sum_{i=1}^{n} I(d_i)_{[x,x+D-1]}$.
Buscamos d tal que $(\forall x)$ P(x) $\leq$ P(d).
Veamos que $d \leq d_n$:
P$(d_n) = 1$ pues $d_n \in [d_n, d_n + D - 1]$ pero $\forall x > d_n$, P(x)= 0 pues $d_i < x, \forall i \in 1...n$.
Luego, si $d > d_n, d$ no es óptimo.
Veamos que existe un d óptimo tal que $d = d_i$ para algún $i \in 1...n$.
Sea d' óptimo.
Sea $d = min_{i \in 1...n}\ d_i | d_i \geq d'$.
P(d)$\geq$ P(d') pues $\forall i$ tal que $d_i \in [d', d' + D - 1], d_i \in [d, d + D - 1]$ ya que $\nexists\ j$ tal que $d' \leq d_j < d$ y pues $d' + D - 1 \leq d + D - 1$.
Luego, hemos reducido el espacio de solución a $\{\ d_1, ..., d_n\}$.

2.1.4. Cota de complejidad temporal

El algoritmo utilizado es el siguiente:

Algoritmo 1 Algoritmo Camiones Sospechosos

Entrada:
$intervaloInspector \leftarrow$ dameIntervaloInspector ▷ *integer*
$cantidadDeCamiones \leftarrow$ dameCantidadCamiones ▷ *integer*
$fechasCamiones \leftarrow$ dameFechasCamiones ▷ $arreglo\langle integer\rangle$
Salida:
Fecha Óptima Inspector ▷ *integer*
Cantidad De Camiones Analizados ▷ *integer*

```
1: ordenar(fechasCamiones)                                   ▷ O(n. log n)
2: inicioInter ← 0                                           ▷ O(1)
3: finInter ← 0                                              ▷ O(1)
4: mDia ← 0                                                  ▷ O(1)
5: mCantidadCamiones ← 0                                     ▷ O(1)
6: mientras finInter < cantidadDeCamiones hacer              ▷ Como máximo n interaciones
7:     mientras DiferenciaMenorAInterInspector(inicioInter, finInter) hacer
8:         finInter ← finInter + 1
9:     fin mientras
10:    si CantCamionesEn(inicioInter, finInterl) < mCantidadCamiones entonces   ▷ O(1)
11:        mDia ← inicioInter                                ▷ O(1)
12:        mCantidadCamiones ← CantCamionesEn(inicioInter, finInter)   ▷ O(1)
13:    fin si
14:    inicioInter ← inicioInter + 1
15: fin mientras                                             ▷ ciclo O(n)
16: retornar mDia, mCantCamiones
17:
18: función CantCamionesEn(inicioInter, finInter)
19:    retornar finInter − inicioInter                       ▷ O(1)
20: fin función                                              ▷ final O(1)
21:
22: función DiferenciaMenorAInterInspector(inicioInter, finInter)
23:    diferencia ← fechasCamiones_finInter − fechasCamiones_inicioInter   ▷ O(1)
24:    retornar finInter < cantidadDeCamiones and diferencia < intervaloInspector   ▷ O(1)
25: fin función                                              ▷ final O(1)
```

Consideremos la complejidad del ciclo for. A lo largo del ciclo se van actualizando izq y

der, y se va guardando el maximo que lleva en si O(1). ¿Cuántas veces se actualizan izq y der? Pues, como van en forma creciente de 1 a n, exactamente n veces cada una. El ciclo for tiene complejidad O(2n) = O(n). La complejidad del algoritmo es finalmente O(n log n).

Como se puede ver el algorítmo tiene 2 partes bien diferenciadas: Primero tiene un ordenamiento de un arreglo unidimensional. Para eso se usa el algorítmo de ordenamiento que brinda a librería standard de *c++*. En la documentación de la misma se puede apreciar que su complejidad en el peor caso es de $\mathcal{O}(n.\log n)$[5] donde n es la distancia que hay entre el primer y el último elemento que se quieren ordenar. En este caso se necesita ordenar todo el arreglo, por lo que termina siendo $\mathcal{O}(n.\log n)$ con respecto al tamaño de la entrada.

Luego hay un bucle while. El bucle `mientras` se repite mientras que el fin del intervalo a analizar se encuentre dentro del arreglo de camiones.

Apenas se entra al ciclo `mientras` hay otro ciclo, el cuál tiene una complejidad de peor caso de $\mathcal{O}(n)$. Este ciclo interno se encarga de hacer avanzar el puntero *finInter* hasta que la diferencia entre de las fechas entre el principio del intervalo a analizar y el final del intervalo a analizar sea menor a *intervaloInspector*. El peor caso posible en este ciclo sería que la diferencia entre el la fecha mas próxima y la fecha mas lejana sea menor a intervalo inspector. En ese caso sale del bucle porque se pasó del máximo. Sim embargo si eso ocurre también se deja de cumplir la condición de la guarda externa, por lo tanto el bucle externo también termina.

Es decir, ambas guardas dependen de la misma comparación entre *finInter* y *cantCamiones*, por lo tanto cuando termine una va a terminar la otra. Por otra parte *finInter* siempre avanza. No hay ningún caso en el cuál retroceda. Esto significa que siempre va a recorrer exactamente *cantCamiones*, es decir que en el peor de los casos va a haber `n` iteraciones del ciclo. Como el resto de las operaciones son todas $\mathcal{O}(1)$ podemos concluir que el ciclo completo tiene una complejidad de $\mathcal{O}(n)$

De esta manera el algorítmo termina teniendo una complejidad de $\mathcal{O}(n.\log n + n) = \mathcal{O}(n.\log n)$. De esta manera la complejidad es estrictamente menor que $\mathcal{O}(n^2)$.

2.1.5. Verificación mediante casos de prueba

A continuación presentamos distintas instancias que sirven para verificar que el programa funciona correctamente.

Input: [D n d_1...d_n], Output: [d c]

D: cant. de días de contratación del experto
n: cant. total de camiones que pasan por el puesto
d_i con $1 \leq i \leq n$: días de llegada de los camiones
d: día inicial del período de contratación del experto
c: cant. de camiones inspeccionados por el experto

Sea L el intervalo de días en que llegan los camiones.
$L =$ (máx d_i − mín d_i +1) con $1 \leq i \leq n$.

De los datos de entrada y salida sabemos que:

- $1 \leq$ D, n, d_i (con $1 \leq$ i $\leq$ n).
- $1 \leq$ c $\leq$ n

[5] http://www.cplusplus.com/reference/algorithm/sort/

- $1 \leq$ mín $d_i \leq$ d $\leq$ máx d_i

Entonces, podemos separar el conjunto de soluciones en los siguientes casos:

Caso	Condición	Ej.Input	Ej.Output
1	D=L, d=mín d_i, c=n	3 3 1 2 3	1 3
2	D>L, d=mín d_i, c=n	5 3 1 2 3	1 3
3	D<L, d=mín d_i, c<n	3 4 1 2 5 1	1 3
4	D<L, mín $d_i <$ d $<$ máx d_i, c<n	2 5 1 3 4 7 4	3 3
5	D= 1, d=máx d_i, c<n	1 4 1 5 5 5	5 3

Si c=n podemos deducir que sólo es posible que d=mín d_i y que D≥L (casos $1, 2$). De no ser así, quedarían camiones sin inspeccionar y eso estaría contradiciendo que c=n. En cambio si c<n, por lo que mencionamos anteriormente, sólo nos queda que el período de contratación del experto sea menor al intervalo L (casos $3, 4, 5$).

Estos 5 casos cubrirían todo el espacio de soluciones. Ejecutamos distintos ejemplos correspondientes a cada uno de ellos y obtuvimos la respuesta esperada. Por lo tanto, podemos concluir que el comportamiento del programa es correcto.

2.1.6. Medición empírica de la performance

Como se mencionó en `Sección Cota de complejidad temporal (página 7)`, el algoritmo elegido para la resolución del problema, está dividido en dos partes.

La primera parte del algoritmo corresponde al ordenamiento de un arreglo con las fechas en que los camiones estarán pasando, con una complejidad para el peor caso de $\mathcal{O}(n.\log n)$. Debido a que este ordenamiento está provisto por el *sort* de la *STL de C++*, lo asumimos como un caso de `caja negra`, es decir, no vale la pena realizar un análisis profundo del comportamiento empírico de esta parte del algoritmo, ya que no tenemos conocimiento ni control sobre su comportamiento interno, por lo cual no sería justo efectuar a priori un planteo teórico realista sobre mejores o peores casos.

En la segunda parte se corren dos ciclos anidados, en donde se decide desde qué día el inspector tendría que ser contratado para revisar la mayor cantidad de camiones posibles. Como se explicó en el análisis teórico, la complejidad es amortizada en n, por lo que en el peor caso esta sección pertenece a las familias de las $\mathcal{O}(n)$.

Ya que $\mathcal{O}(n) \subset \mathcal{O}(n.\log n)$, es intuitivo decir que la complejidad final del algoritmo estaría regida por el algoritmo de ordenamiento.

Para la elección de los casos de tests, tuvimos esto en cuenta y decidimos tomar los tiempos de ejecución del programa, variando el orden de las fechas de los camiones en el input.

- Orden ascendente de fechas
- Orden descendente de fechas
- Orden aleatorio de fechas

Comparamos, entonces, cada uno de estos 3 casos en un gráfico de *tiempo* vs n: cantidad de camiones.

Se puede observar que la cota teórica calculada es correcta.

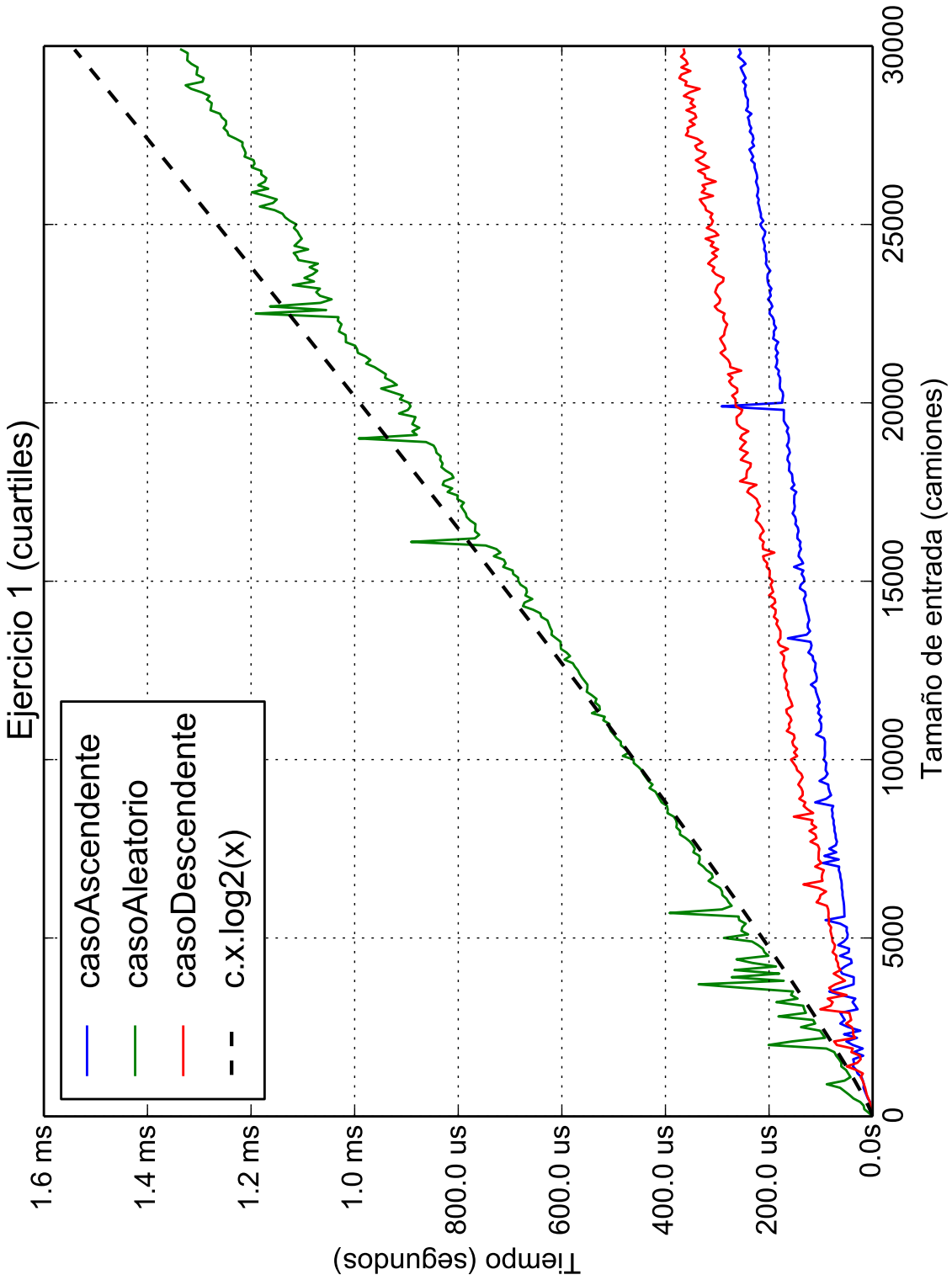

Figura 1: **Muestreo general del Ejercicio 1**

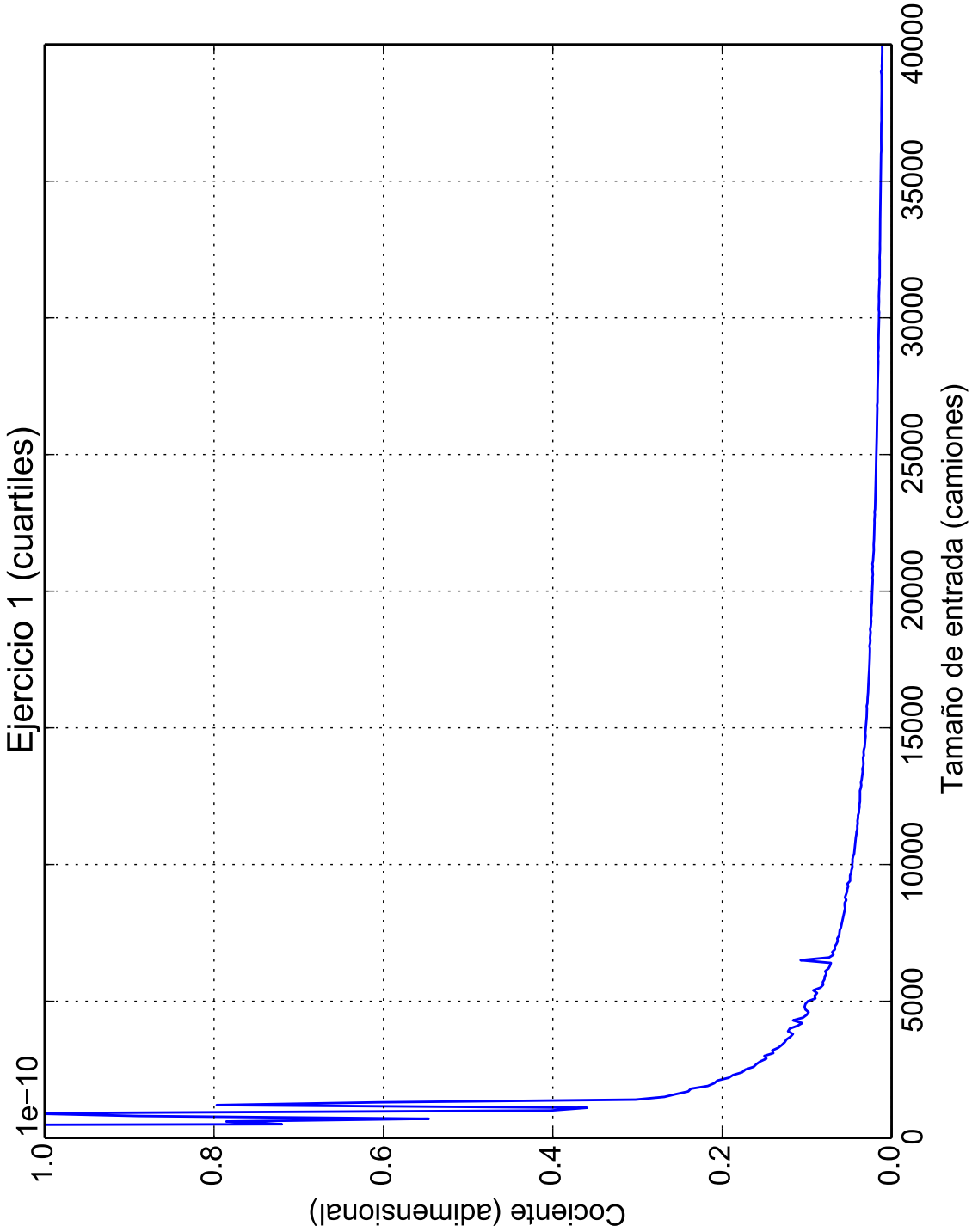

Figura 2: **Relación contra** n^2

Si bien como se explicó previamente y se plasmó en las figuras 1 y 2 la complejidad del algoritmo está dada por el ordenamiento. Sin embargo también resulta interesante analizar

otro factor que entra en juego en el rendimiento del algoritmo. La segunda parte del algoritmo tiene un mejor caso en el cual la diferencia entre los entre la mayor fecha y la menor fecha es menor o igual a *intervaloInspector*. Si eso ocurre la condición de la guarda interna se cumple en una primera iteración y la constante del ciclo se ve completamente amortizada. Sigue siendo lineal, pero la constante resulta muy baja. El peor caso sería el caso contrario: Cada fecha está a mas de *intervaloInspector* días de su vecina. Esto significaría que para iteración del bucle exterior el iterador del final del intervalo solo puede avanzar un día, por lo tanto para cada elemento hay que realizar toda la lógica de comparación y descarte.

Esto significa que esa segunda parte, la parte lineal, funciona muy bien cuando hay mucha densidad de datos y pierde eficacia cuando hay mucha dispersión de datos. Para contrastar esto lo que se hizo fue fijar un tamaño de entrada (5000 camiones) y un intervalo de inspector (200) y generar diferentes casos de prueba, en los cuales lo que cambia es la dispersión de los datos. Para eso lo que se hizo fue tomar intervalos de tamaño creciente y ubicar todos los datos de entrada en esos intervalos ditribuidos uniformemente. Al aumentar el tamaño del intervalo y distribuir los mismos datos en el intervalo lo que se obtiene es una menor densidad de datos. El intervalo se midió en manera porcentual con respecto a *intervaloInspector*. En la figura 3 se puede apreciar este experimento.

Es interesante además notar que mientras el tamaño del intervalo no llega al 100 % de *intervaloInspector* el tiempo de mantiene completamente constante. Luego aumenta de manera claramente lineal. Por otra parte se puede observar como la parte de ordenamiento se estabiliza rapidamente mientras que el tiempo total de ejecución aumenta debido a la parte lineal del algorítmo.

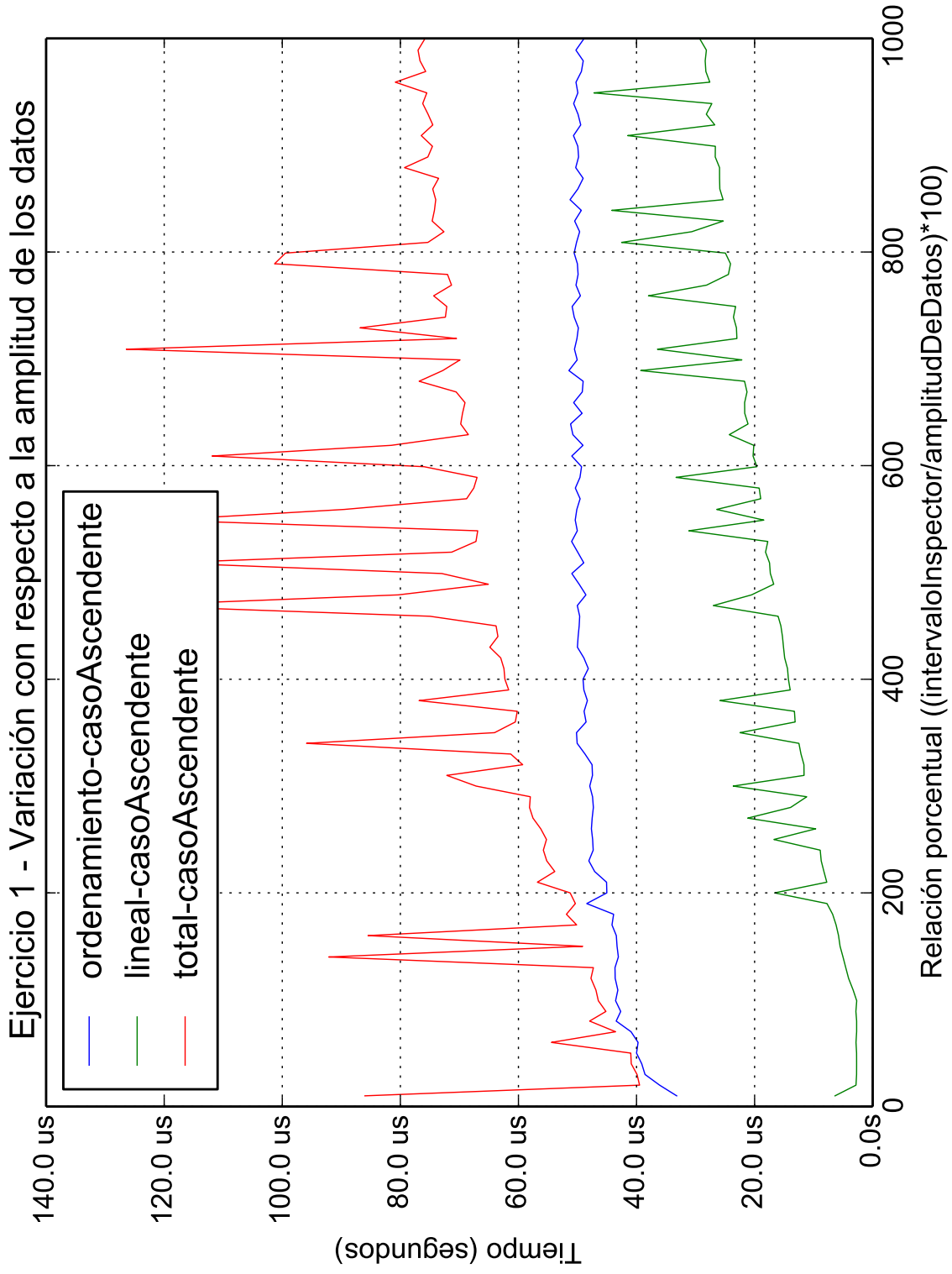

Figura 3: Tiempo de ejecución contra dispersión de los datos

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

2.2.1. Descripción

Dada una cantidad n de pizzas i de orfebrería, cada una de las cuales tienen una pérdida diaria de ganancia d_i y una cantidad de días t_i que lleva su fabricación, se quiere saber el orden en que el pizzero Franco Shar tendría que fabricar y vender las mismas para minimizar las pérdidas. Cada pizza no entregada pasadas las media hora dará como mínimo una pérdida de d_i por t_i, y a esto se le sumará d_i por la cantidad de días que Frank haya estado fabricando otras pizzas, ya que él sólo puede trabajar de a una pizza a la vez. El problema se deberá resolver con una complejidad temporal de $\mathcal{O}(n^2)$, siendo n la cantidad de pizzas a fabricar. En el resultado se mostrarán: P, el monto total perdido de la ganancia, y las pizzas $i1, ..., in$ ordenadas.

Ejemplo 2.2.1.1.

Frank tiene que entregar $n = 3$ piezas en total $(i : 1, 2, 3)$.

Los datos de cada pieza i son los siguientes:

Pieza	Pérdida diaria	# días de fabr.
1	1	3
2	2	3
3	3	3

La fabricación de las 3 piezas le llevará 9 días en total y durante estos días se irán sumando las pérdidas diarias de ganancia correspondiente a cada pieza sin terminar.

Día 1: $3 + 2 + 1$.

Día 2: $3 + 2 + 1$.

Día 3: $3 + 2 + 1 \rightarrow$ entrega pizza 3.

Día 4: $2 + 1$.

Día 5: $2 + 1$.

Día 6: $2 + 1 \rightarrow$ entrega pizza 2.

Día 7: 1

Día 8: 1

Día 9: $1 \rightarrow$ entrega pizza 1.

Como resultado se obtendría el orden $3, 2, 1$ y $P : 30$, el monto total perdido de la ganancia.

Ejemplo 2.2.1.2.

Frank tiene que entregar $n = 2$ piezas en total $(i : 1, 2)$. Los datos de cada pieza i son los siguientes:

Pieza	Pérdida diaria	# días de fabr.
1	1	3
2	3	1

La fabricación de las 2 piezas le llevará 4 días en total y durante estos días se irán sumando las pérdidas diarias de ganancia correspondiente a cada pieza sin terminar.

- Día1: $3 + 1 \rightarrow$ entrega pieza 2.
- Día2: 1.
- Día3: 1.
- Día4: $1 \rightarrow$ entrega pieza 1.

Como resultado se obtendría el orden $2, 1$ y $P : 7$, el monto total perdido de la ganancia.

2.2.2. Hipótesis de resolución

Tenemos n pizzas, cada una de los cuales cuenta con un valor de d, es lo que se va devaluando diariamente, y t, el tiempo que tomará su fabricación. Nos interesa minimizar la pérdida de la ganancia total, dando el mejor orden en que habría que ir fabricando las piezas $(1, ..., n)$.

Definimos un cociente π como d/t para cada uno de los n elementos. La relación entre d y t es la que nos determinará cuál es el orden óptimo de fabricación.

Contamos con un vector $v = (v_1, v_2, ..., v_n)$. Cada v_i representa la pieza i a fabricar, $i = 1, ..., n$.

Nuestro algoritmo ordena de forma decreciente por el cociente π, luego devolvemos un vector v óptimo.

Iteramos sobre v con el indice i, acumulando la pérdida del subvector v[1..i]. Al final de las iteraciones tenemos entonces la pérdida completa correspondiente al vector v.

2.2.3. Justificación formal de correctitud

Contamos con $n \in \mathbb{N}$, donde cada collar queda determinado por una `3-upla`: (p,d,t) donde
$p \in \{1, ..., n\}$ es el número de collar ($p_i \neq p_j$),
$d \in \mathbb{N}$ es la pérdida diaria de ganancia del collar, y
$t \in \mathbb{N}$ es el tiempo de fabricación (en días) del collar.
Se pide devolver el orden de fabricación de los collares que minimice la pérdida total.
El espacio de soluciones es luego, todos los vectores que son permutaciones de los primeros n naturales.

Sea $v = (v_1, v_2, ..., v_n)$ y entiéndase,
$d[V_i]$ como la segunda componente del collar que tiene $p = v_i$
$t[V_i]$ como la tercer componente del collar que tiene $p = v_i$.
Sea $C(V) = \sum_{i=1}^{n}((\sum_{j=1}^{i} t[v_i])d[v_i])$ la función que queremos minimizar sobre el espacio de soluciones. La solución v cumple $(\forall V')$ C(v) $\leq$ C(v').
Para cada $i \in 1, ...n$ consideramos $\pi(i) = \frac{d[v_i]}{t[v_i]}$.
Veamos que v cumple $(\forall i \in 1...n-1)\pi(i) \geq \pi(i+1)$.

Supongo que no, es decir, $(\exists X \in 1...n-1)\pi(X) < \pi(X+1)$.

Sea V'= $(v_1, v_2, ..., v_{X-1}, v_{X+1}, v_X, v_{X+2}, ..., v_n)$, es decir, se construye a partir de V con

las posiciones X e X+1 intercambiadas.
Veamos que C(v')<C(v).

$$C(v)-C(v')=\sum_{i=1}^{n}(d[v_i](\sum_{j=1}^{i}t[v_j]))-\sum_{i=1}^{n}(d[v'_i](\sum_{j=1}^{i}t[v'_j]))=$$

$$\underbrace{\sum_{i=1}^{x-1}(d[v_i]\sum_{j=1}^{i}t[v_j]-\sum_{i=1}^{x-1}d[v'_i]\sum_{j=1}^{i}t[v'_j]}_{0,\ \text{pues}\forall i\in 1...x-1, v_i=v'_i}+d[v_x]\sum_{j=1}^{x}t[v_j]-d[v'_x]\sum_{j=1}^{x}t[v'_j]+$$

$$d[v_{x+1}]\sum_{j=1}^{x+1}t[v_j]-d[v'_{x+1}]\sum_{j=1}^{x+1}t[v'_j]+\underbrace{\sum_{i=x+2}^{n}(d[v_i]\sum_{j=1}^{i}t[v_j]-\sum_{i=x+2}^{n}d[v'_i]\sum_{j=1}^{i}t[v'_j]}_{0,\ \text{pues}\forall i\in x+2...n, v_i=v'_i \text{y} \sum_{j=1}^{i}t[v_j]=\sum_{j=1}^{i}t[v'_j]}=$$

$$d[v_x](\sum_{j=1}^{x-1}t[v_j]+t[v_x])-d[v_{x+1}](\sum_{j=1}^{x-1}t[v_j]+t[v_{x+1}])+$$

$$d[v_{x+1}](\sum_{j=1}^{x-1}t[v_j]+t[v_x]+t[v_{x+1}])-d[v_x](\sum_{j=1}^{x-1}t[v_j]+t[v_{x+1}]+t[v_x])=$$

$$-d[v_x]t[v_{x+1}]+d[v_{x+1}]t[v_x]>0\Leftrightarrow\frac{d[v_{x+1}]}{t[v_{x+1}]}>\frac{d[v_x]}{t[v_x]}\Leftrightarrow$$

$\pi(x+1)>\pi(x)$ que asumimos como verdadero.
Abs! Pues v era óptimo. Luego, $(\forall i\in 1...n-1)\pi(i)\geq\pi(i+1)$.

Sabemos pues v óptimo $\Rightarrow$ v ordenado no ascendentemente por π.
Sabemos que existe $v*$ óptima, luego existte $v*$ óptima y ordenada.
Veamos que v ordenada $\Rightarrow$ v óptima, es decir que no existe v ordenada y no óptima.

C(v)=C(v*) pues uno siempre puede ir intercambiando dos posiciones x y $x+1$ que mantengan el ordenamiento (es decir $\pi(x)=\pi(x+1)$) las veces que sea necesario hasta que $v=v*$ y luego, C(v)=C(v*) y por lo tanto, v óptima.

Veamos que se pueden intercambiar x y $x+1$:
$v=(v_1,...,v_x,v_{x+1},...,v_n)$
$v'=(v_1,...,v_{x+1},v_x,...,v_n)$

Desarrollando como hecho previamente,
$C(v)-C(v')=-d[v_x]t[v_{x+1}]+d[v_{x+1}]t[v_x]=0\Leftrightarrow$
$\frac{d[v_{x+1}]}{t[v_{x+1}]}=\frac{d[v_x]}{t[v_x]}\Leftrightarrow\pi(x+1)=\pi(x)$ que asumimos verdadero.

Luego, v ordenado $\Rightarrow$ v óptimo.

2.2.4. Cota de complejidad temporal

Como se puede ver en el pseudo código la complejidad está dada por un Ordenamiento. El ordenamiento se realiza utilizando un criterio específico de comparación que consiste en comparar el cociente entre la devaluación diaria de una pieza y el tiempo que toma fabricar esa pieza. La comparación entonces tiene una complejidad de $\mathcal{O}(1)$, por lo que un ordenamiento basado en comparaciones tiene complejidad de $\mathcal{O}(n.\log n)$.

Algoritmo 2 La joya del Río de la plata

Entrada:
cantidadDePiezas ← dameCantidadDePiezas ▷ *integer*
listaDePiezas ← dameListaDePiezas ▷ *vector < Pieza >*
Salida:
Orden Óptimo Piezas ▷ *vector < Pieza >*

```
1: sumaTotal ← 0                                                          ▷ O(1)
2: tiempoAcum ← 0                                                         ▷ O(1)
3: ordenar(listaDePiezas, compararPiezasMayorQue )                        ▷ O(n. log n)
4: para cada pieza en listaDePiezas hacer                                 ▷ n iteraciones
5:     tiempoAcum ← tiempoAcum + pieza.tiempo                             ▷ O(1)
6:     sumaTotal ← sumaTotal + pieza.devaluación ∗ tiempoAcum             ▷ O(1)
7: fin para                                                               ▷ cicloO(n)
8: retornar listaDePiezas, sumaTotal
9:

10: función compararPiezasMayorQue(pieza1, pieza2)                        ▷ O(1)
11:     si (pieza1.devaluación/pieza1.tiempo) > (pieza2.devaluación/pieza2.tiempo entonces   ▷ O(1)
12:         retornar 1                                                    ▷ O(1)
13:     sino
14:         retornar 0                                                    ▷ O(1)
15:     fin si
16: fin función                                                           ▷ final O(1)
17:

18: Definición Pieza:
19:  integer  tiempo                                  ▷ tiempo de fabricación
20:  integer  devaluación                             ▷ cantidad de dinero perdido por unidad de tiempo
```

Luego hay un ciclo **para cada** que itera una por cada elemento en *listaDePiezas*, por lo tanto el ciclo se realiza *cantidadDePiezas* veces. Cada iteración del ciclo realiza sólo acciones $\mathcal{O}(1)$, por lo tanto el ciclo en si es $\mathcal{O}(cantidadDePiezas)$. La cantidad de piezas es proporcional al tamaño de la entrada, por lo tanto el ciclo es $\mathcal{O}(n)$.

En conclusión el algorítmo tiene una complejidad temporal $\mathcal{O}(n + n.\log n)$=$\mathcal{O}(n.\log n)$

2.2.5. Verificación mediante casos de prueba

A continuación presentamos distintas instancias que sirven para verificar que el programa funciona correctamente:

Input	
n	
d_1	t_1
⋮	⋮
d_n	t_n

Output	
$i_1 \ldots i_n$	P

n: cant. total de piezas
d_i: cant. de pérdida diaria de la pieza i
t_i: cant. de días de fabricación $(1 \leq i \leq n)$
$i_1 \ldots i_n$: n piezas ordenadas
P: monto total perdido de la ganancia

Podemos separar el conjunto de soluciones en los siguientes casos:

1. $\frac{d_i}{t_i}$ es igual $\forall\ i$.

Input
3
2 1
4 2
8 4

Output	
1 2 3	70
1 3 2	70
⋮	⋮
3 2 1	70

Cualquier permutación de las piezas $1, 2, 3$ es una solución válida para este caso. $P = 70$ en los 6 posibles outputs. Esto es porque el cociente entre la pérdida diaria y el tiempo de fabricación es igual para las 3 piezas, $\frac{d_i}{t_i} = 2, \forall i = 1, 2, 3$.

2. $\exists\ j$ tal que $\frac{d_i}{t_i} \neq \frac{d_j}{t_j}$, j $\neq$ i, con j, i $= 1, ..., n$.

   - Fijo t_i y varío d_i, $\forall\ i$. Entonces el orden va a estar determinado por la pérdida diaria d_i.

Input
5
1 2
2 2
3 2
5 2
4 2

Output	
4 5 3 2 1	70

Pieza i	d_i
1	1
2	2
3	3
4	5
5	4

   - Fijo d_i y varío t_i $\forall\ i$. El orden depende del cociente entre d_i y t_i.

Input
3
2 1
2 2
2 3

Output	
1 2 3	20

Pieza i	$\frac{d_i}{t_i}$
1	2
2	1
3	0,67

   - Varío d_i y t_i $\forall\ i$ pero sin que todos los $\frac{d_i}{t_i}$ sean iguales. El orden depende del cociente entre d_i y t_i.

Input
3
3 1
1 2
2 3

Output	
1 3 2	14

Pieza i	$\frac{d_i}{t_i}$
1	3
2	0,5
3	0,67

Estos casos cubrirían todo el espacio de soluciones. Ejecutamos distintos ejemplos correspondientes a cada uno de ellos y obtuvimos la respuesta esperada. Por lo tanto, podemos concluir que el comportamiento del programa es correcto.

2.2.6. Medición empírica de la performance

El algoritmo consta de dos partes, una de ordenamiento al principio, con complejidad $\mathcal{O}(cantPiezas.\log cantPiezas)$ y una segunda parte que es sencillamente un recorrido por todas las piezas donde para cada pieza se hace una multiplicación y una suma, por lo que claramente es lineal con respecto al tamaño de la entrada, pero no solo en el peor caso sino que en cualquier caso. De esta manera el algorítmo pertenece a los de complejidad $\mathcal{O}(n.\log n)$.

Para contrastar esta idea de manera empírica lo que se hizo fue analizar el resultado del siguiente cociente:

$$\frac{tiempoDeEjecución}{n.\log n}$$

La figura 4 muestra el resultado de de este cociente para diferentes tamaños de la entrada. Se puede apreciar que para tamaños de entrada grandes la curva se estabiliza muy cerca de una constante. Este resultado reafirma la hipótesis de que el algorítmo pertenece a los de complejidad $\mathcal{O}(n.\log n)$

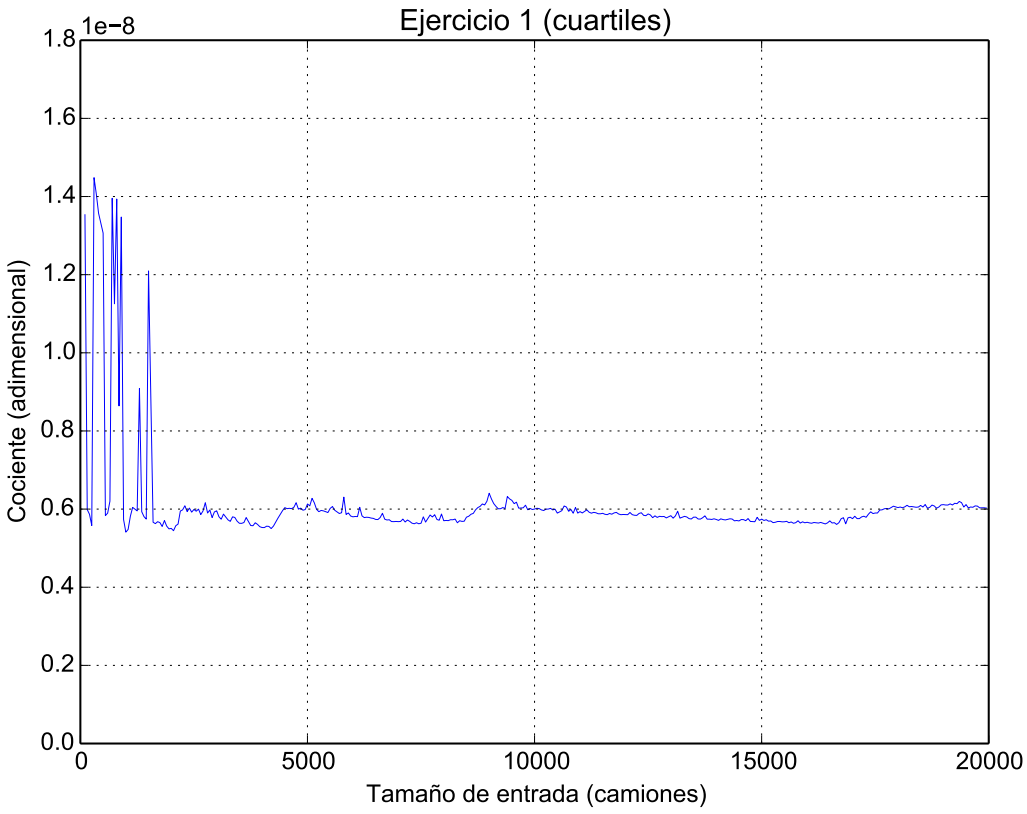

Figura 4

2.3. Problema 3: Rompecolores

2.3.1. Descripción

Este problema consiste en ubicar en un tablero la mayor cantidad de piezas posibles siguiendo ciertas reglas.

- El tablero contiene $n \times m$ casilleros cuadrados, n filas y m columnas.
- Piezas existentes: $1, ..., n \times m$. (Cantidad total de piezas: $n \times m$).
- Una pieza es cuadrada y puede tener de 1 a 4 colores distintos. A cada lado (sup, izq, der, inf) le corresponde un color.
- Las piezas no se pueden rotar.
- Colores posibles: $1, ..., c$ (c entero positivo). Como una pieza puede tener como mucho 4 colores distintos y son $n \times m$ piezas en total, $1 \leq c \leq (4 \times n \times m)$.
- 2 piezas pueden ubicarse en casilleros adyacentes sólo si sus lados adyacentes tienen el mismo color. Podría ocurrir que no sea posible llenar completamente el tablero con las piezas existentes.
- El contenido final de una casilla podría ser $1, ..., n \times m$, si se pudo colocar alguna ficha, o 0 si quedara vacía.
- Cantidad mínima de piezas que se pueden colocar en el tablero: $(n \times m)/2$. (Se intercalan las piezas en el tablero).

  Para un tablero de 3×3 y uno de 2×2, suponiendo un caso donde ninguna ficha puede colocarse adyacente a otra, una de las posibles soluciones sería la siguiente:

  En el tablero de 3×3 se podrían colocar las fichas $1, 2, 3, 4$ y 5, y en el de 2×2, las fichas 1 y 2.

1	0	2
0	3	0
4	0	5

1	0
0	2

El problema se deberá resolver utilizando la técnica de *Backtracking* eligiendo algunas podas para mejorar los tiempos de ejecución del programa.

Ejemplo 2.3.1.1. Se tiene un tablero de 2×2, los colores $1, 2, 3$ y las piezas **1**,**2**,**3**,**4** :

	1	
3	**1**	2
	2	

	3	
2	**2**	2
	1	

	3	
1	**2**	3
	2	

	1	
1	**4**	2
	2	

En este caso, la cantidad máxima de piezas que se pueden colocar en el tablero es 3. Entonces las posibles soluciones serían:

1	**2**
0	**4**

0	**2**
3	**1**

2.3.2. Hipótesis de resolución

El objetivo del problema es diseñar un algoritmo utilizando la técnica de **backtracking**. El algoritmo de **backtracking** se puede concebir como una técnica recursiva de recorrido de grafos. En este caso, establecemos un paralelismo entre «el universo de soluciones», y los nodos de un **grafo arbol** $n-ario$, en donde cada nodo representa una solución posible, y n representa la cantidad máxima de soluciones distintas que pueden desprenderse a partir de realizar un cambio determinado en la solución anterior (*vecindad*).

Calcular todo el gráfo tendría una complejidad de $\mathcal{O}(n^n)$, *o incluso infinita* si no se estableciese una asbtracción correcta al momento de transformar el universo de soluciones en un grafo. Por esa razón, para realizar esta tarea, se utiliza el concepto de *grafo implícito*, mediante el cual se establece formalmente la composición del mismo, sin la necesidad de expresar cada una de sus componentes, **a través de una descripción matemática de sus nodos y aristas**, en donde **cada nodo es una solución**, y **cada arista es la transformación de una solución a una solución distinta dentro de su *vecindad*** de manera que, dado un caso de prueba particular, sea posible recrearlo en el momento mismo en que se realiza el recorrido del grafo.

Podemos separar el tipo de problema en dos casos: **problemas de decisión**, para los que es necesario que la solución cumpla ciertas condiciones particulares, y **problemas de optimización**, en donde se desea obtener la mejor de las soluciones válidas en base a una **función objetivo**, entendiendo a esta última como «la función que establece cuándo una solución es mejor que otra». Para este algoritmo en particular, el problema planteado es de **optimización**, ya que se desea obtener la solución válida (debe cumplir ciertos requisitos) que maximice la función objetivo, definiéndola como la sumatoria de los casilleros completos de un tablero determinado.

Formalizando

Notación. Sea π el conjunto formado por todas las 4-uplas de la forma

$$p \in \pi \Leftrightarrow \{(\forall\ a,b,c,d \in \mathbb{N})\ \exists p = [a,b,c,d]\}$$

decimos que $p \in \pi$ es una pieza.

Definición 2.3.2.1. Dado el orden de piezas relativo a la entrada del problema, definimos el conjunto ordenado

$$P = \{p_1, ..., p_n\}$$

como el conjunto de todas las piezas posibles, de forma tal que

$$\{i \geq 1\} \wedge \{(\forall i < n)\ p_i < p_{i+1} \Leftrightarrow \text{“}\textbf{pieza i}\text{” } \textit{es anterior a } \text{“}\textbf{pieza i+1}\text{”}\}$$

Notación. $P[i] = P_i$

Definición 2.3.2.2. Sea $T \in \theta$ la sucesión finita de la forma

$$T = \{p_1, ..., p_n : \{i_{\geq 1} \in \mathbb{N}\} \wedge \{(p_i \in \pi) \vee (p_i = \text{“}\texttt{vacio}\text{”})\}\},$$

T representa a un tablero, y definimos a cada $T_i = p_i$ como «el valor de la pieza ubicada en la posición i del tablero, o “`vacío`” en el caso de que no hubiese ninguna»

Notación. $T[i] = T_i$, $\theta =$ «conjunto de todos los posibles tableros»

Definición 2.3.2.3. sea $t.lleno : \mathbb{N}_{\geq 1} \to \mathbb{N}$, tal que

$$\begin{cases} t.lleno(i) = verdadero \text{ si } T[i] \neq \texttt{vacio} \\ t.lleno(i) = falso \text{ de lo contario} \end{cases}$$

Definición 2.3.2.4. La función objetivo es, dado un tablero T, $f : \theta \to \mathbb{N}$, de la forma

$$f(T) := \sum_{i=1}^{i \leq casilleros} t.lleno\Big(T\,(i)\Big)$$

La idea base es ir llenando el tablero en cierto orden, probando colocar en cada casillero piezas que no rompan las reglas, y analizando la cantidad total de piezas del tablero que hayan sido cubiertas. Se debe guardar de alguna forma la combinación de piezas que haya cubierto más casilleros. Consideramos dejar un casillero libre como colocar la pieza representada por el 0. En este problema una solución candidata o rama es una sucesión de piezas (que puede o no incluir el 0), que van cubriendo el tablero. Una primera decisión que se tomo fue guardar una estructura piezasPorColores que para cada par de colores (x, y) guarda una lista con todos las piezas que tienen como color superior e izquierdo x e y, respectivamente. Vamos recorriendo el tablero en cada fila de izquierda a derecha, desde la fila superior hasta la inferior. De este modo una pieza es legal en una posición si su color izquierdo es igual al color derecho de la pieza a su izquierda (de existir), y su color superior es igual al color inferior de la pieza ubicada arriba (de existir). Podemos obtener las piezas candidatas que cumplen con esa restricción de colores en piezasDeColores. De este modo nos aseguramos que todas las piezas por las que nos vamos ramificando mantienen las restricciones del tablero. Evitamos asi tener que iterar sobre todas las piezas verificando si cumplen la restricción. Una poda trivial que se implementó fue dejar de buscar soluciones cuando encontramos una solución que cubre todo el tablero. Al comenzar la búsqueda sabemos que por lo menos vamos a poder llenar la mitad de los casilleros del tablero. Ésto siempre se puede conseguir si visualizamos nuestro tablero como uno de ajedrez y ponemos las piezas en los casilleros negros. Ningun par de piezas compartiría lado. Vamos llevando cuenta de los huecos que dejamos en el tablero. Si hemos dejado tantos huecos como para que, aun llenando correctamente lo que queda del tablero, no alcanza para superar la mejor solución encontrada hasta el momento, abandonamos la rama.

2.3.3. Justificación formal de correctitud

[Vale por una demostración]

2.3.4. Cota de complejidad temporal

[Vale por un análisis teórico]

2.3.5. Verificación mediante casos de prueba

A continuación presentamos distintas instancias que sirven para verificar que el programa funciona correctamente.

Input			
n	m	c	
sup_1	izq_1	der_1	inf_1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
sup_{nxm}	izq_{nxm}	der_{nxm}	inf_{nxm}

Output		
x_1	$\dots$	x_m
$\vdots$		$\vdots$
x_n	$\dots$	x_{nxm}

- n: #filas.
- m: #columnas.
- c: #colores.
- sup_i, izq_i, der_i, inf_i: colores (entre 1 y c) de los lados de la pieza i.
- x_i: número de la pieza en la casilla i del tablero. ("0" si no hay ninguna pieza).

Separamos en casos y mostramos ejemplos para cada uno:

- Todas las piezas son iguales:

  - El tablero se completa.

Input
2 2 1
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1

Output
1 2
3 4

Input
2 2 2
1 2 2 1
1 2 2 1
1 2 2 1
1 2 2 1

Output
1 2
3 4

    En estos ejemplos el tablero se completaría colocando las 4 piezas de cualquier forma. No hay restricciones.

  - El tablero tiene la mínima cantidad de piezas ($(n \times m)/2$).

Input
2 2 2
1 2 1 2
1 2 1 2
1 2 1 2
1 2 1 2

Output
1 0
0 2

Output
0 1
2 0

En este caso sólo se pueden colocar las piezas intercaladas. Se podrían tomar 2 piezas cualesquiera para la solución (Ej: piezas 3 y 4).

- Todas las piezas son diferentes:

  - El tablero se completa.

Input	Output
3 3 8	9 2 3
8 2 6 2	4 5 6
1 1 2 4	7 8 1
1 2 7 5	
3 4 4 6	
4 4 3 7	
5 3 6 8	
6 4 1 2	
7 1 2 7	
8 8 1 3	

  - El tablero tiene la mínima cantidad de piezas ($(n \times m)/2$).

Input	Output	Output
2 2 3	1 0	0 1
2 3 2 2	0 2	2 0
1 1 2 2		
1 3 2 2		
2 1 2 2		

    En este caso sólo se pueden colocar las piezas intercaladas. Se podrían tomar 2 piezas cualesquiera para la solución (Ej: piezas 3 y 4).

  - El tablero no se completa pero se llena más que el mínimo posible.

Input	Output
2 2 3	1 2
1 2 1 1	0 3
1 1 1 2	
2 2 3 2	
3 2 3 3	

- Existe alguna pieza diferente al resto:

  - El tablero se completa.

Input	Output
2 2 2	1 3
2 2 1 2	2 4
2 2 1 2	
2 1 2 2	
2 1 2 2	

- El tablero tiene la mínima cantidad de piezas $((n \times m)/2)$.

Input	Output	Output
2 2 3	1 0	0 1
2 2 1 2	0 4	2 0
3 3 3 1		
3 2 1 1		
2 2 1 2		

  En este caso sólo se pueden colocar las piezas intercaladas. Se podrían tomar 2 piezas cualesquiera para la solución (Ej: piezas 3 y 4).

- El tablero no se completa pero se llena más que el mínimo posible.

Input	Output
2 2 3	1 3
2 2 1 2	0 4
3 2 3 1	
3 1 3 2	
2 2 1 2	

Ejecutamos el programa con los distintos ejemplos y se llegó a la solución esperada. Por lo tanto, podemos concluir que el comportamiento del programa es correcto.

2.3.6. Medición empírica de la performance

Para resolver este problema utilizamos la técnica de *Backtracking*. Nos interesa encontrar la forma en que se podrían colocar las piezas dadas, tal que se llegue a cubrir la mayor parte del tablero siguiendo ciertas reglas.
Con esta técnica se estarían probando todas las combinaciones posibles hasta encontrar una solución. Sin embargo, esto podría demorar muchísimo, por lo que agregamos algunas podas para no tener que recorrer todas las ramas y poder mejorar así, la performance del programa.
Una de las podas a analizar es la que lleva cuenta de las casillas que quedaron libres en el tablero, y verifica que si aún llenando las casillas restantes, no se llega a una solución mejor que la que ya se tenía, no continúa por esa rama.
Para comprobar que esta poda realmente funciona, medimos los tiempos ejecutando el algoritmo con ella y lo comparamos con los tiempos del mismo sin la poda. Durante la medición se fue variando la cantidad de piezas totales pasadas como input.
En el gráfico se pueden observar los distintos tiempos obtenidos según se haya utilizado la poda o no. Además, se ve que la complejidad exponencial calculada se cumple.

3. Apéndices

3.1. Código Fuente (resumen)

3.2. Bibliografía

Referencias