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 0111_2

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 en particular 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, además, 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** (o factibles), caracterizando matemáticamente el concepto de **solución correcta** y, en los casos en que se solicita **optimización**[1], se definen las condiciones que dan forma ya sea a todo el subconjunto de **soluciones óptimas** que se encuadran dentro de las pretenciones del problema, o a una **solución particular** dentro del mismo (la cual denominamos *mejor solución*).

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*»[2].

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**[3].

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*[4].

[1] Es decir, que la solución pertenezca al *subconjunto de soluciones que **maximicen** o **minimicen** una determinada función*

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

[3] 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

[4] 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.

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.

2. Instrucciones de uso

2.1. 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`

3. Desarrollo del TP

3.1. Problema 1: Camiones sospechosos

3.1.1. Descripción

Planteo del problema

El inspector `Rick A. Lapazta`, legendario[5] 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 Ravioli* 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.

Formato de los datos

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	**Formato de salida**
$D\ n\ d_1\ d_2\ ...\ d_n$	$d_i\ c$

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

Interpretación del problema (ejemplos)

Ejemplo 3.1.1.1.

Si se contratase al experto por $D = 3$ días consecutivos y en total hubiesen $n = 20$ camiones sospechosos $(c1, c2, ..., c20)$, se podría tener una entrada como la siguiente:

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

La cual se podría visualizar de la siguiente forma:

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

En este caso, como el primer día del período de contratación sería $d = 4$ y habrían sido inspeccionados un total de $C = 9$ camiones durante los días `Día4`, `Día5` y `Día6` el programa debería devolver la siguiente salida:

```
Salida:
4 9
```

Ejemplo 3.1.1.2.

Bajo las mismas condiciones del ejemplo anterior, también se podría tener una entrada como la siguiente:

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

La cual se podría visualizar de la siguiente forma:

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

En este otro caso, el primer día del período de contratación sería $d = 1$, en el cual habrían sido inspeccionados, durante los días `Día1`, `Día2` y `Día3`, un total de $c = 10$ camiones:

```
Salida:
1 10
```

3.1.2. Planteamiento de resolución

Caracterización de la solución

Notación. Sean:

- $D \in \mathbb{N}$ la cantidad de días que trabaja el inspector
- $n \in \mathbb{N}$ la cantidad total de camiones.
- $t \in \mathbb{N}$ la diferencia entre el último día y el primer día en que pasan camiones.
- $c_i \in \mathbb{N}$ el día en que pasará el "camión i". **Notar aquí la diferencia de notación con la notación de entrada**.
- $d_i \in \mathbb{Z}$ el "día i", dado por el intervalo determinado por d_1 = el primer día que pasa un camión, y d_t = el último día que pasa un camión. Aclaración: d_i puede ser negativo.

Proposición 3.1.2.1. *Representamos el conjunto* ***universo de soluciones*** *bajo la forma de $S \subseteq \mathbb{N}$ en donde, para cada solución, $s \in S$ representa al día en que se contrata al inspector. Esta representación es suficiente para generar todo el universo de soluciones únicas.*

Demostracion 3.1.2.2. Bajo las condiciones de una determinada entrada, asumir que este parámetro s basta para definir una **solución única** es equivalente a decir que **existe una función sobreyectiva** $\#camiones$: $S \to \mathbb{N}_0$ que, dado un determinado día s, devuelve la cantidad de camiones inspeccionados para esa solución.

Esto es así porque el inspector trabajará una **cantidad fija y predeterminda de días consecutivos** que depende exclusivamente de susodicha entrada, y además la cantidad total de camiones inspeccionados es la sumatoria de la cantidad de camiones inspeccionados en cada uno de los días en que el inspector trabaje.

Definición 3.1.2.1. Función Objetivo

Nos interesa elegir cuál es el mejor día entre d_1 y d_{t+1-D}.Siendo camionesEnElDia una función que dado un día devuelve el conjunto de camiones que circulan en el mismo, buscamos maximizar la siguiente **función objetivo** $f: S \to \mathbb{N}$:

$$f(s) = \sum_{i=k}^{k+D-1} \mid camionesEnElDia(d_i) \mid$$

la cual, además, resulta ser la función mencionada $\#camiones$ mencionada en 3.1.2.2.

Proposición 3.1.2.3. *Reducimos el intervalo de análisis al rango $[d_1,\ d_{t+1-D}] \subset S$, ya que el resto de las soluciones resultan indiferentes a los objetivos del problema.*

Demostracion 3.1.2.4. Ya que nos interesa obtener la solución que maximice la cantidad de camiones inspeccionados (función 3.1.2.1), y por el mismo motivo expuesto en la demostración anterior, podemos acotar el análisis del universo de soluciones al rango $[d_{1+1-D},\ d_t]$ debido a que, ya que d_1 es el día en que pasa el primer camión, y d_t es el día en que pasa el último camión, podemos asegurar que para cualquier solución **s** fuera de este rango $\#camiones(s) = 0$.

Más aún, podemos acotar el análisis al intervalo $[d_1,\ d_{t+1-D}]$, contenido en el intervalo anterior ya que, por las razones antes expuestas, podemos asegurar que:

$\#camiones$ valuada en $[d_{1+(1-D)},\ d_1)$ es menor o igual a $\#camiones(d_1)$

$\#camiones$ valuada en $(d_{t+(1-D)},\ d_t]$ es menor o igual a $\#camiones(d_{t+1-D})$.

Proposición 3.1.2.5. *Dado un intervalo de análisis, nos interesará inspeccionar sólamente los días en que haya transcurrido al menos un camión.*

Demostracion 3.1.2.6. Dado que buscamos maximizar la función objetivo (3.1.2.1) entonces, abreviando camionesEnElDia como cEED,

$$\begin{aligned}
f(s) &= \sum_{i=s}^{s+D-1} |cEED(d_i)| \\
&= |cEED(s)| + \sum_{i=s+1}^{s+D-1} |cEED(d_i)| \\
&= |cEED(s)| + \sum_{i=s+1}^{s+D-1} |cEED(d_i)| + |cEED(s+D-1+1)| - |cEED(s+D-1+1)| \\
&= |cEED(s)| + \sum_{i=s+1}^{s+D-1+1} |cEED(d_i)| - |cEED(s+D-1+1)| \\
&= |cEED(s)| + f(s+1) - |cEED(s+D-1+1)|
\end{aligned}$$

Pero si $| \; cEED(s) = 0 \; |$, y dado que cEED es no negativa, entonces se deduce que

$$\begin{aligned}
f(s) &= |cEED(s)| + f(s+1) - |cEED(s+D-1+1)| \\
&= 0 + f(s+1) - |cEED(s+D-1+1)| \leq f(s+1)
\end{aligned}$$

de lo cual se desprende que

$$s = 0 \Rightarrow f(s) \leq f(s+1)$$

y como buscamos maximizar f, podemos obviar las soluciones en que s=0.

Pseudocódigo

Primero, entonces, ordenamos en forma creciente el conjunto c_1,...,c_n. Sabemos que la solución pertenece a este conjunto. Según (3.1.2.1), tenemos a $f(s)$ como la sumatoria de la cantidad de camiones que pasan desde el día s hasta el día $s + D - 1$. Nuestro algoritmo itera sobre cada elemento c_i del conjunto, calculando para cada uno $f(c_i)$, y guardando en todo momento el elemento e tal que $f(e)$ es el máximo. Al final de las iteraciones, tenemos guardado el elemento e tal que $f(e)$ es máximo en todo el conjunto de soluciones. Devolvemos e y $f(e)$.

3.1.3. 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 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 $\{ \; d_1, ..., d_n \}$.

3.1.4. 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 $\mathcal{O}(n.\log n)$[6] 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

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

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 $\mathcal{O}(n^2)$.

3.1.5. 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
- $1 \leq$ 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.

3.1.6. Medición empírica de la performance

Como se mencionó en `Sección 3.1.4:Cota de complejidad temporal (página 10)`, 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.

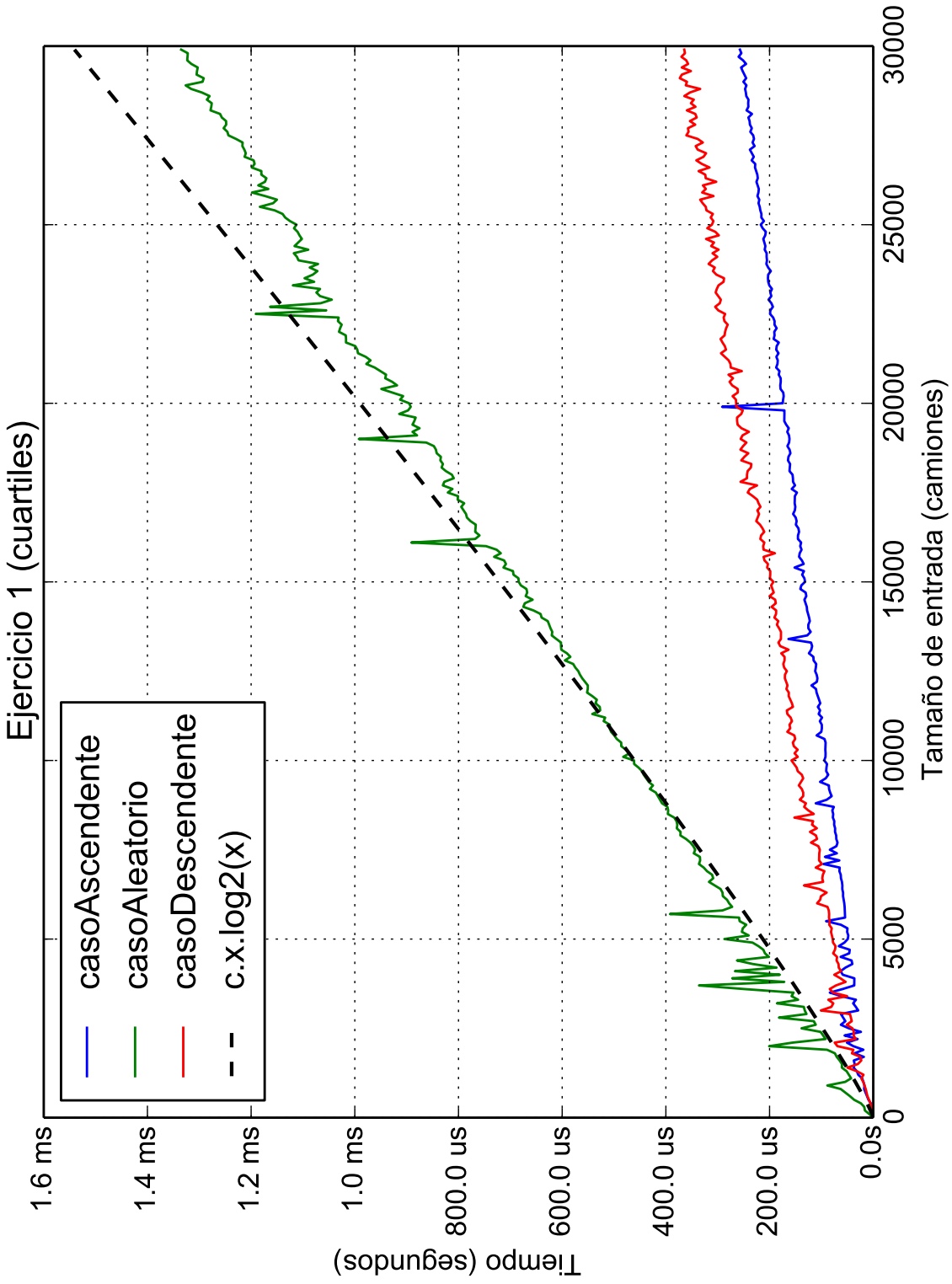

Figura 1: **Muestreo general del Ejercicio 1**

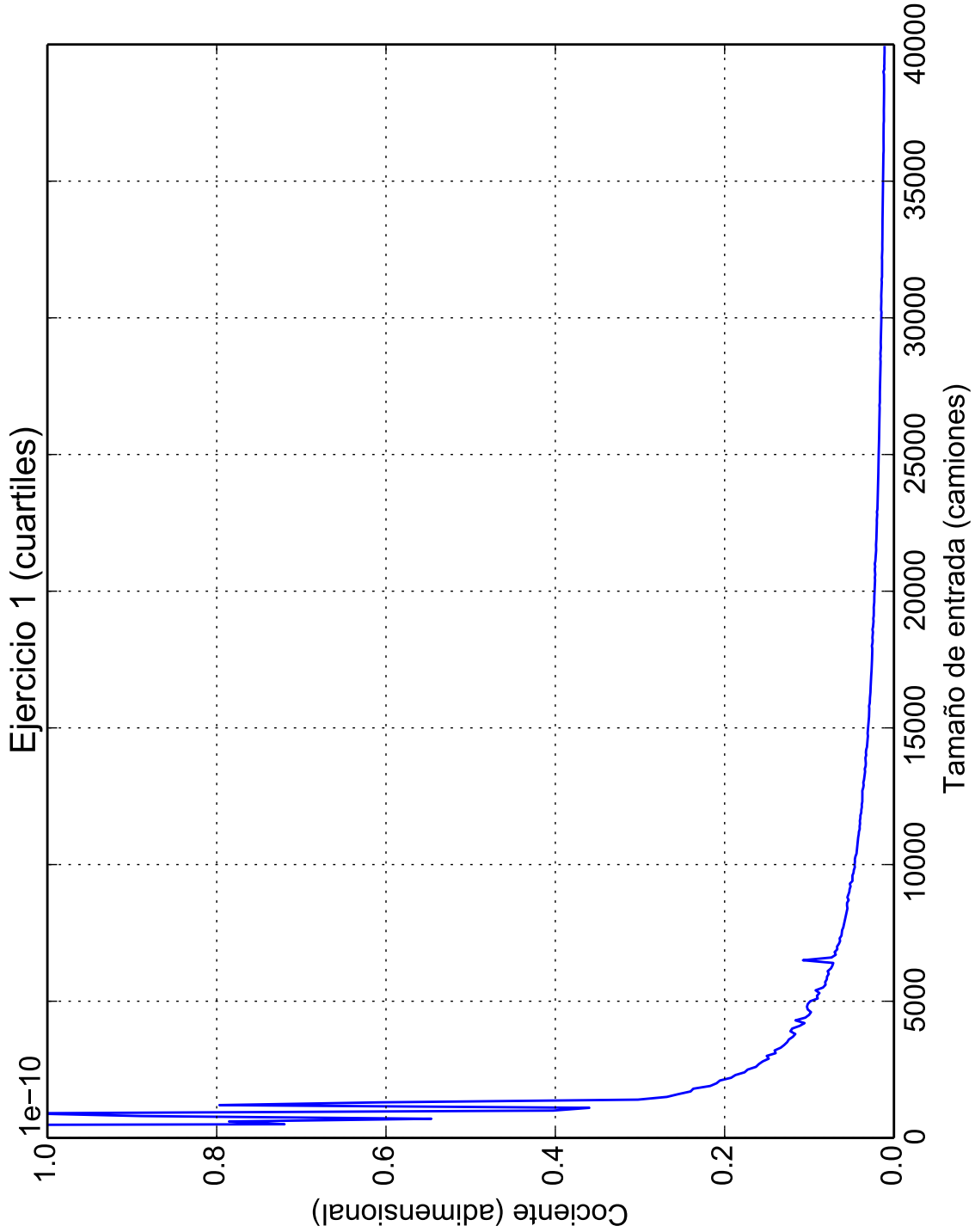

Figura 2: **Relación contra** n^2

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.

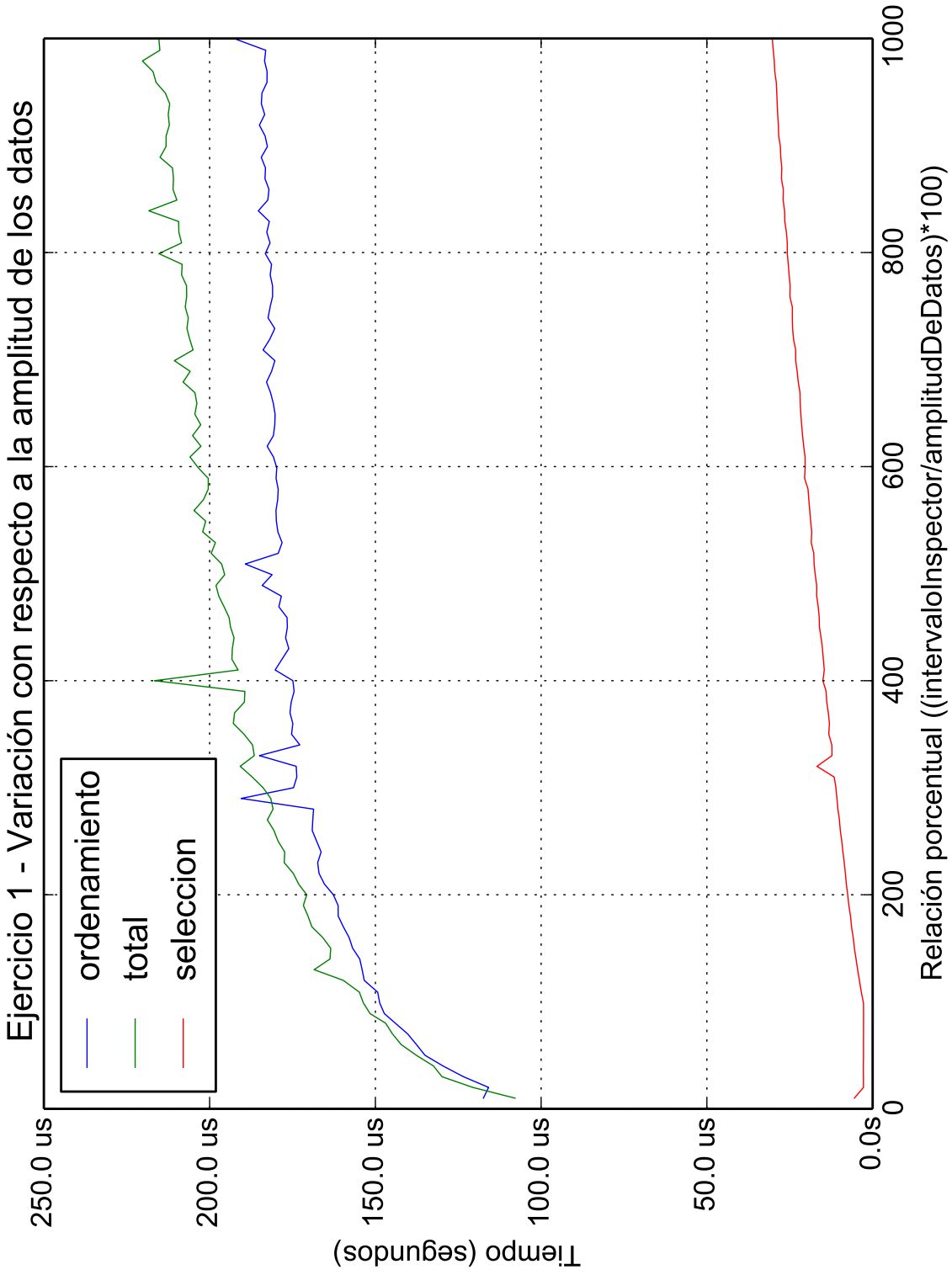

Figura 3: Tiempo de ejecución contra dispersión de los datos

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

3.2.1. 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 3.2.1.1.

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 3.2.1.2.

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.

3.2.2. 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.

3.2.3. 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 $\pi(i) = \dfrac{d[v_i]}{t[v_i]}$.
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.

3.2.4. 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                                                               ▷ ciclo O(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 $\mathcal{O}(n + n.\log n)$=$\mathcal{O}(n.\log n)$

3.2.5. 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
$\vdots$	$\vdots$
d_n	t_n

Output	
$i_1 \dots i_n$	P

n: cant. total de piezas
d_i: cant. de pérdida diaria de la pieza i
t_i: cant. de días de fabricación $(1 \leq i \leq n)$
$i_1 \dots i_n$: 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.

3.2.6. 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 $\mathcal{O}(n.\log n)$

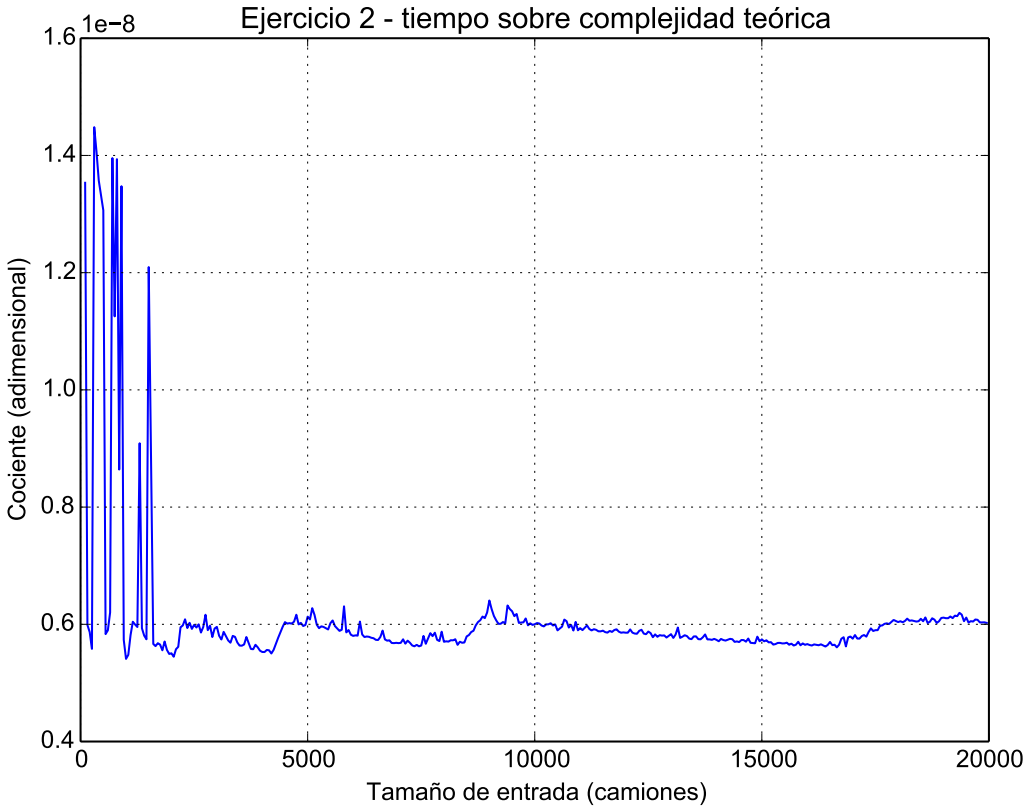

Figura 4

3.3. Problema 3: Rompecolores

3.3.1. Descripción

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

Consideraciones

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

Ejemplos

Ejemplo 3.3.1.1. 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

Ejemplo 3.3.1.2. 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**

Algoritmos de backtracking (ideas y análisis general)

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[7], estableciendo un paralelismo entre «el universo de soluciones», y los nodos de un **grafo arbol n-ario**[8], 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*). Además puede, según el caso, ser interpretado como un árbol en donde cada nodo representa **una solución o sub-solución no necesariamente "válida"** (o "bien formada"), y en donde sólamente **las hojas del árbol** serían *soluciones "correctamente formadas"* (aunque no necesariamente factibles).

Podemos separar también el tipo de problema en dos casos: los **problemas de decisión**, para los que es necesario que la solución cumpla ciertas condiciones particulares determinados por la denominada **función de selección**, y los **problemas de optimización**, los cuales son derivados de los anteriores, y en donde se desea obtener la **"mejor"** de las soluciones válidas[9] en base a una **función objetivo**, entendiendo a esta última como «*la función que establece cuándo una solución es mejor que otra*».

Pequeño ejemplo sobre las ideas anteriores

Un ejemplo concreto de los párrafos anteriores sería un problema en el que se nos pidiese encontrar una lista de 10 números del 1 al 20, tal que la lista sea estrictamente creciente. Tendríamos entonces un problema de decisión, en el que la función de selección estaría determinada por las condiciones *"que la lista sea estrictamente creciente"*, *"que el número esté entre 1 y 20"*, y *"que el largo de la lista sea 10"*, es decir, en donde tomaríamos como soluciones factibles a todas aquellas en que «**lista[i]** $<$ **lista[i+1] y además** $1 \leq$ **lista[i]** ≤ 20**, y además** $|$**lista**$| = 20$»[10]. Si se agregara, además, la condición "quiero de todas las listas posibles la que maximice la sumatoria de cada uno de sus componentes", me encontraría entonces con un **problema de optimización**.

En este ejemplo, entonces, una **solución factible** sería cualquiera que cumpla con los criterios de selección, mientras que una **solución correctamente formada**, es decir, aquellas que podrían pertenecer a las hojas de un posible arbol / "universo de soluciones" (tal y como se menciona en el párrafo introductorio), sería simplemente "una lista de 10 números"[11].

De esta forma, para el ejemplo anterior, se podría establecer un algoritmo en que cada nodo interno del árbol de soluciones representase la **subestructura** de una o más soluciones, sin llegar a ser una solución, más concretamente, se podría asumir que el nodo raíz del arbol

[7] Quizás la oración no está expresada de la mejor forma; al decir "se puede concebir como..." se quiere dar a entender que no se está dando la *definición estricta* de backtracking, sino más bien una *interpretación* de la misma. Se dio por sobreentendido el hecho de que en su sentido más básico es "una técnica de resolución de problemas"; así, representando al "universo de soluciones" como un grafo, **y en el contexto del párrafo**, creemos que la idea que se intenta exponer es válida.

[8] O incluso un grafo conexo con ciclos, en el caso de un problema muy mal modelado.

[9] Es decir, las que cumplan con el criterio de selección. Tener en cuenta que esto último no implica necesariamente la existencia de funciones inválidas, sino que puede darse el caso en que el criterio de selección admita que todas las soluciones a un problema determinado son válidas, y por consiguiente simplemente se esté buscando obtener la mejor de ellas.

[10] Se cometen abusos varios de notación para no ahondar en detalles innecesarios.

[11] Esto último es debatible si se considera el tamaño como un criterio de selección; en todo caso habría que formalizar aún más y no viene al caso.

es una lista vacía, y que cada hijo es una lista que contiene a los elementos de su padre, y cuyo tamaño es superior al anterior en 1, de forma tal que es simple ver que: *cada nodo comparte una subestructura con sus nodos-hermanos heredada desde su nodo-padre, y sólamente las hojas del arbol son soluciones.*

Análisis general

Siguiendo la línea de pensamiento anterior al ejemplo, teniendo en cuenta todas las ideas ya expuestas, calcular el gráfo completo del universo de soluciones (factibles y no factibles) para este problema tendría una complejidad de $\mathcal{O}(n^n)$, *o incluso infinita* si no se estableciese una asbtracción idónea al momento de transformar el universo de soluciones en un grafo. Además, incluso luego de establecer una abstracción en la que el grafo no considerase, por ejemplo, soluciones imposibles, el tamaño del grafo sigue siendo muy grande[12]. Por esta razón, para realizar esta tarea, se utiliza el concepto de *grafo implícito*, mediante el cual se establece matemáticamente la composición del mismo, sin la necesidad de expresar cada una de sus componentes, **sino a través de una descripción formal 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.

Para asegurar la correctitud según los criterios expuestos en el párrafo anterior, es importante establecer una *vecindad* tal que el universo de soluciones tenga una subestructura adecuada para la realización de "podas", es decir, que dados dos $nodos - hijos$ derivados de un mismo $nodo - padre$, se puedan encontrar en estos características en común derivadas directamente de su padre; más concretamente, el nodo padre debe representar una solución o subsolución para la cual las funciones de selección y objetivo valúen de una forma idónea con relación a sus hijos. Por ejemplo, si en un ejercicio de optimización se lograse establecer un grafo de soluciones de tal forma que dado un nodo padre sus nodos derivados tengan una valuación menor o igual en la función objetivo, y se buscase la solución que la maximizara, sería fácil implementar entonces una poda que, en el momento en que una subsolución alcance un valor menor al deseado, *corte* la rama de soluciones determinada por el susodicho nodo.

Para este algoritmo en particular, el problema planteado es de **optimización**, ya que se desea obtener una solución válida (debe cumplir ciertos requisitos, como que cada pieza coincida en sus colores con sus aledañas), pero que además maximice la **función objetivo**, definiendo a esta última como, dado un tablero determinado, la sumatoria de los casilleros que contienen ficha.

Definición 3.3.1.1. Sean T el conjunto de todos los posibles tableros, definimos la función auxiliar $t.lleno : \mathbb{N}_{\geq 1} \to [0, 1]$, tal que

$$\begin{cases} t.lleno(i) = 1 & \text{si T[i] contiene una pieza} \\ t.lleno(i) = 0 & \text{de lo contario} \end{cases}$$

La **función objetivo** es, dado un tablero $t \in T$, $f : T \to \mathbb{N}$, de la forma

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

[12] De orden factorial, como se expone luego.

3.3.2. Planteamiento de resolución

Idea general

La idea es aplicar el algoritmo de backtracking según las consideraciones expuestas anteriormente. Para ello, recorreremos el tablero en un orden determinado (de izquierda a derecha, de arriba a abajo), evaluando en cada paso la ficha correspondiente a la posición donde se encuentra el recorrido: de tal forma que en cada paso se evalúe la posición siguiente a la que se evaluó en el paso anterior.

En cada paso del algoritmo se colocará una ficha, y luego se llamará recursivamente a la función, probando así las fichas posteriores. Es importante tener en cuenta que en cada llamada recursiva el algoritmo probará con M fichas, y se llamará recursivamente para cada una de ellas.

Es fácil ver que, *en su versión más básica*, este comportamiento lleva a tener una complejidad del orden de $\mathcal{O}\left(M^N\right)$, siendo $N = n * m$ la cantidad de posiciones del tablero, y M la cantidad de piezas, ya que se realizan N iteraciones (una por cada posición del tablero), y en cada iteración se prueban M piezas, lo cual resulta en $\overbrace{M * M * M * ... * M}^{N\ veces} = M^N$. Hay que tener en cuenta que esta forma de recorrer el universo de soluciones está analizando también las soluciones no factibles, es decir las que resultan imposibles, tal como "colocar la misma pieza en todos los casilleros".

Poda: Función de Selección

La primer poda posible resulta totalmente trivial, ya que se trata de probar en cada iteración sólamente las piezas que no se hayan colocado ya en el tablero. Aunque esta poda es simple, y no es más que una aplicación incompleta de las restricciones impuestas por la función de selección, se la menciona, ya que de esta forma se reduce la complejidad a $\mathcal{O}\left(\overbrace{(M) * (M-1) * ... * (M-(N-1)) * (M-(N))}^{\text{N multiplicaciones, restando una pieza en cada una}}\right)$, es decir $\mathcal{O}\left(\frac{M!}{(N-1)!}\right) \subseteq \mathcal{O}(M!)$. Simplificando, diremos que esta cota reduce la complejidad del orden exponencial $\mathcal{O}\left(M^N\right)$ al orden factorial $\mathcal{O}(M!)$.

Sobre la misma poda anterior, se puede implementar una mejora, que es probar en cada iteración sólamente las piezas cuyos colores sean coherentes con los colores de las piezas que ya se encuentren colocadas en el tablero. Así, si una posición tiene una pieza a la izquierda cuyo color derecho es verde, yo debería poder colocar en esa posición sólamente las piezas cuyo color izquierdo es verde. Esta última poda, tal y como está planteada, en realidad se trata básicamente de **reducir el recorrido del universo de soluciones a "las soluciones factibles"**, es decir, las que generan un tablero final válido. Conceptualmente no es distinta a la poda mencionada en el párrafo anterior, sino que es más bien una extensión de la misma, ya que las dos se encargan de analizar en cada iteración la correctitud del tablero en términos de la **función de selección**. De hecho, esta poda se encarga de **seleccionar sólamente las fichas que cumplan *estrictamente* con la función de selección**, mientras que la anterior sólo aplica la restricción de forma parcial.

Es imposible, sin embargo, establecer *a priori* una caracterización precisa del orden de complejidad del algoritmo bajo las condiciones de esta nueva poda, ya que la misma depende

no solo del tamaño de entrada, sino del contenido de la entrada. Puede darse, por ejemplo, el caso de una entrada en la que todas las piezas tengan el mismo color en todos sus lados, lo cual significaría que habría que probar todas las piezas en todas las posiciones, lo cual conllevaría una complejidad de $\mathcal{O}(M!)$. También puede darse el caso en que la disposición de la entrada sea de forma tal que luego de colocar la primer pieza, sólo exista en cada paso una pieza posible, con lo cual la complejidad sería del orden de $\mathcal{O}\left(M * 1^{N-1}\right)$, es decir $\mathcal{O}(M)$, lineal sobre la cantidad de piezas. Por la razón expuesta en este párrafo diremos que, a pesar de que según la disposición de la entrada la misma puede ser mucho menor, la complejidad en el peor caso está igualmente acotada por $\mathcal{O}(M!)$.

Por todo lo antes expuesto, además, podemos asegurar que al final del recorrido (es decir, luego de evaluar todas las hojas del árbol) se habrán evaluado *estrictamente* todas las **soluciones factibles**, ya que el algoritmo habrá recorrido en cada paso cada posición del tablero, **evaluando las piezas que cumplan estrictamente con el criterio de selección**.

Poda: Función Objetivo

Como se expuso en párrafos anteriores, la solución brindada por el algoritmo debe no solo cumplir con las condiciones de una determinada función de selección, sino también maximizar cierta función objetivo (*ref: 3.3.1.1*), la cual depende de la cantidad de piezas dispuestas en el tablero o, inversamente, de la cantidad de "agujeros", es decir, posiciones en que no exista ninguna pieza.

Dado que las soluciones comparten una subestructura, en donde tomando un determinado nodo d[13] del arbol de soluciones podemos asegurar que «para todas las hojas para las que exista un camino simple hacia ese nodo que contenga a los hijos de ese nodo»[14] la solución representada por esas hojas contiene a todas las piezas que ya se encuentran determinadas en el nodo d, es válido afirmar que si una deteriminada subsolución contiene cierta cantidad de "agujeros", entonces todas sus soluciones derivadas también van a contener esos mismos "agujeros". Así, podemos decir que *la cantidad de agujeros de una hoja es siempre mayor o igual a la cantidad de agujeros de su padre*.

Nota. Antes de continuar, vale hacer una aclaración ligada a la implementación, y es que a efectos de abstraer este concepto, y mejorar la claridad del algoritmo se consideró al "agujero" o "casillero libre" como otra pieza, representada por la constante `PIEZA_VACÍA` (es decir, independientemente del *tipo* con el que se elija representar a la pieza, nos referiremos a esta pieza como `PIEZA_VACÍA`).

A la luz de las consideraciones anteriores surge esta poda: el objetivo es **cortar una *rama***[15] en el momento en que se encuentra una subsolución que, por su cantidad de piezas vacías, **no es candidata a maximizar la función objetivo** (y ya que no lo es la subsolución, por las consideraciones ya expuestas, se puede asumir que tampoco lo serán las soluciones derivadas de ella).

[13] Notación: llamémoslo «subsolución»
[14] O coloquialmente: que exista una "relación de parentezco" con ese nodo
[15] Es decir, una subsolución junto con todas sus soluciones derivadas

Nota. A efectos de simplificar la explicación, procedemos a mencionar primero una versión alternativa de esta poda, la cual quizás resulte más intuitiva. Dada la recursión una determinada posición n, la cual define una subsolución T_n, es fácil ver que si T_n contiene p `piezas no-vacías`, siendo N la cantidad de posiciones, se cumple que:

- Las soluciones derivadas de T_n contienen al menos p piezas.
- Las soluciones derivadas de T_n contienen a lo sumo $p+(N-n)$ piezas.

De esta forma, si uno va guardando la cantidad máxima de piezas encontradas, uno puede cortar una rama cuando sabe que cualquier solución derivada de la misma contendrá una cantidad menor.

Definición 3.3.2.1. Dada una determinada subsolución T_n, en donde n representa a la posición sobre la cual se encuentra la recursión, decimos que la misma contiene una «*cantidad máxima posible de piezas vacías*», y esta es la cantidad de piezas vacías que el tablero tendría en el peor caso. En otras palabras, **la cantidad máxima de piezas vacías de un tablero T_n es la cantidad de piezas vacías que ya contiene la subsolución T_n sumada a todas las piezas que aún le faltan por poner en las sucesivas $T_{n+...}$.** Se asume para ello que el peor caso posible es un tablero en donde todas las piezas $\in \{n+1, ..., N\}$, siguientes a la posición actual resulten incompatibles y, por lo tanto, vacías. La cantidad de piezas de ese intervalo es, entonces, $N - n$.

$$cantidadMaximaAgujeros(T_n) = cantidadAgujeros(T_n) + (N - n)$$

Implementación de la poda

En un principio, podemos asumir que toda combinación posible de fichas es **candidata a ser mejor solución**.

En cada paso recursivo, podemos analizar cuánto es el valor de `cantidadMaximaAgujeros(`T_n`)` para el tablero T_n actual de ese paso, y contrastarlo contra la **menor cantidad máxima de agujeros encontrada hasta el momento**, la cual desde el punto de vista implementativo es una variable ajena a la recursión (por ejemplo, una variable global, o un parámetro de la propia recursión). Siempre que se encuentre una «cantidad máxima de agujeros» menor a la global, se actualizará el valor de la segunda. Para ello, se inicializará esta última variable con el valor del peor caso posible ($N \times M$), es decir el de un tablero en donde todas las piezas sean vacías.

Como dijimos anteriormente, **dada una subsolución** S, podemos asegurar que todas sus soluciones derivadas tendrán al menos la misma cantidad de piezas vacías. Utilizando esta última consideración, dada una subsolución con k_S piezas vacías, y sea X_{min} la `menor cantidad máxima de agujeros encontrada hasta el momento`, de tal forma que $X_{min} <= k_S$, podemos afirmar que la cantidad de `piezas no vacías` de cualquier solución derivada de esa **subsolución** k_S no es en particular **mayor** a la cantidad de `piezas no vacías` del resto de las soluciones, ya que por lo antes expuesto sabemos que dada la existencia de X_{min} existe al menos una subsolución T tal que su cantidad máxima de piezas vacías es exactamente $k_T = X_{min} <= k_S$, lo cual es equivalente a afirmar que la ***cantidad mínima de piezas no vacías de cualquier solución derivada de T*** es $N - k_T = N - X_{min}$, y dado que $k_T <= k_S$ se deduce que $(-K_T) >= (-kS)$, de lo que finalmente se desprende que $T - K_T >= T - k_S$ es decir la *cantidad mínima de piezas no vacías de cualquier solución derivada* de T es mayor o igual a la *cantidad mínima de piezas no vacías de cualquier solución derivada* de S.

De todo lo anterior surge que mediante esta poda se logra descartar ciertas soluciones, cuando se está seguro de que las mismas no maximizan la función objetivo o, en caso de hacerlo, de que ya se encontró[16] al menos una solución que la maximice de igual o mejor forma.

Corolario 3.3.2.1. *Dado un tablero de T* ***posiciones****, y T* ***piezas cualesquiera****, sin tomar a consideración sus colores, podemos asegurar que la* ***cantidad mínima de piezas*** *que podemos colocar en el mismo es de* $\lceil T/2 \rceil$*, de forma tal que las piezas se distribuyan tal y como los casilleros de un mismo color en un* `Tablero de Ajedrez`*. Esto es así debido a que esta distribución nos asegura que en las posiciones indicadas ninguna pieza tiene contacto inmediato con cualquier otra pieza, por lo que no existe riesgo de incompatibilidad. De forma recíproca, podemos también asegurar que la* ***menor cantidad máxima de agujeros*** *que el tablero puede llegar a contener es de a lo sumo* $\lfloor T/2 \rfloor$*. Estos valores pueden ser utilizados, entonces, como inicialización de las variables globales mencionadas en la explicación de la poda.*

Esta misma observación nos permite, también, afirmar que dado un tablero cualquiera existe al menos una solución factible.

3.3.3. Justificación formal de correctitud

Se demostró la correctitud de este algoritmo y sus podas en la propia descripión del mismo: por un lado afirmamos que la **correctitud de un algoritmo genérico de Backtracking**, en su sentido más básico, **resulta trivial**, amparandonos en la propia definición de **Backtracking**, siendo que el algoritmo "*analiza el universo de soluciones*". Luego, durante la descripción de las dos podas implementadas, se demostró que, en el caso de la poda relacionada con la «Función de Selección» se estaban eliminando las soluciones no factibles, las cuales iban a ser de todas formas descartadas por el algoritmo durante el recorrido de las soluciones, y que en el caso de la poda relacionada con la «Función Objetivo» se estaban eliminando todas aquellas soluciones para las que dadas las condiciones de la subestructura implementada se podía preveer que no iban a cumplir con el objetivo de maximizar la función requerida.

3.3.4. Cota de complejidad temporal

La complejidad de este algoritmo pertenece a la familia de $\mathcal{O}(M!)$, en donde M es la cantidad de piezas (contando la pieza vacía). Esto está demostrado en la sección `3.3.2:Planteamiento de resolución` (página 26), en el apartado de la poda relacionada con la función de selección.

3.3.5. Verificación mediante casos de prueba

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

[16] o al menos, implícitamente, se tiene certeza de su existencia

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

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

    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	Output	Output
2 2 2	1 0	0 1
1 2 1 2	0 2	2 0
1 2 1 2		
1 2 1 2		
1 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).

- 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.

3.3.6. Medición empírica de la performance

Para realizar las pruebas de tiempo se nos presentaron varias complicaciones. Dado que la complejidad del algoritmo implementado es exponencial, el tiempo de ejecución se vuelve inmenso ante tamaños de entrada relativamente pequeños. Por ejemplo, dado un tamaño de entrada de $n = 4$, $m = 4$, $c = 10$, tenemos una cantidad de fichas $M = (n * m) + 1 = 17$, y dado que la complejidad del algoritmo es $\mathcal{O}(M!)$, esto es equivalente a afirmar que existe una constante k para la cual a partir de un n_0 nuestro algoritmo está acotado superiormente por $k * (M!)$. Puesto que decimos que este $M!$ en realidad surge de la cantidad de soluciones analizadas, si imaginásemos que el procesador demorase un ciclo de clock en calcular una solución[17], averiguar todo el universo de soluciones bajo las condiciones expuestas en este ejemplo tardaría a lo sumo 17! ciclos de clock o, suponiendo un procesador de 3GHz[18], unos 110419 segundos (aproximadamente media hora).

Debido a lo expuesto anteriormente, nos resultó casi imposible medir el comportamiento del algoritmo para tamaños de tablero muy grandes. Se realizó una primer medición comparando el rendimiento del algoritmo con todas sus podas, contra el algoritmo sin la "poda objetivo", y el mismo sin ambas podas.

[17] Es el mínimo valor que le podemos asignar sin involucrarnos demasiado con la arquitectura/microarquitectura del procesador

[18] Es decir, $3 * 2^30 ciclos por segundo$.

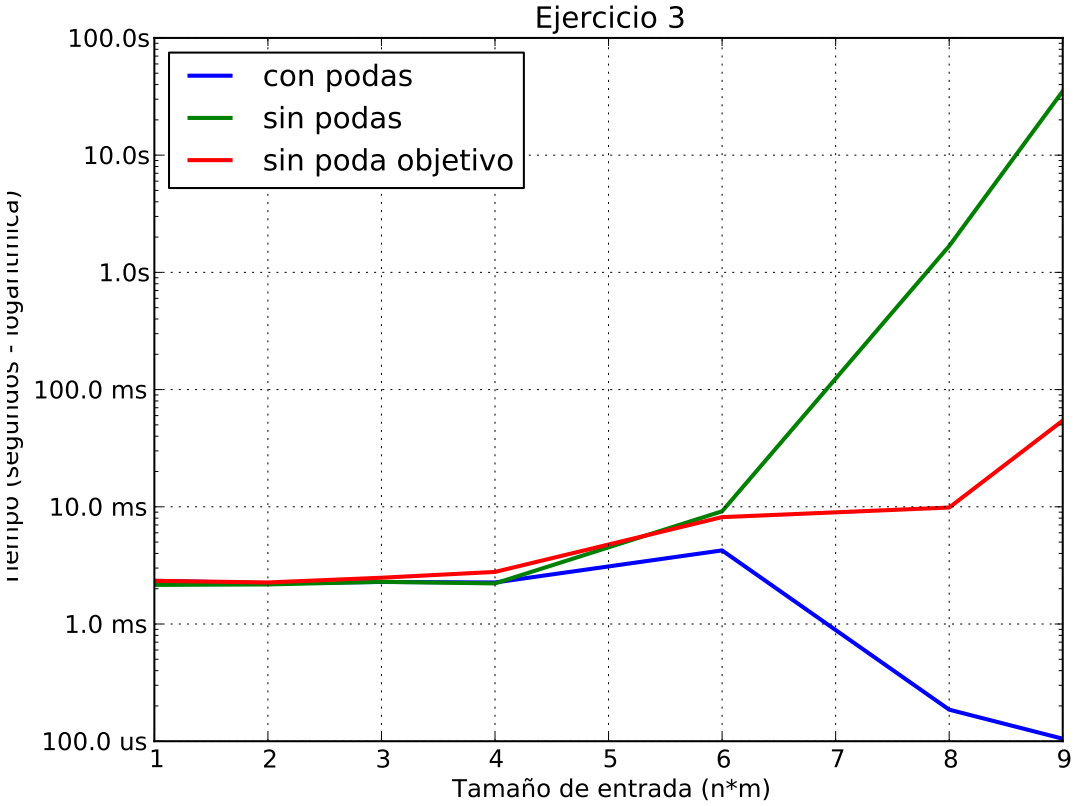

Figura 5: Comparación de las podas

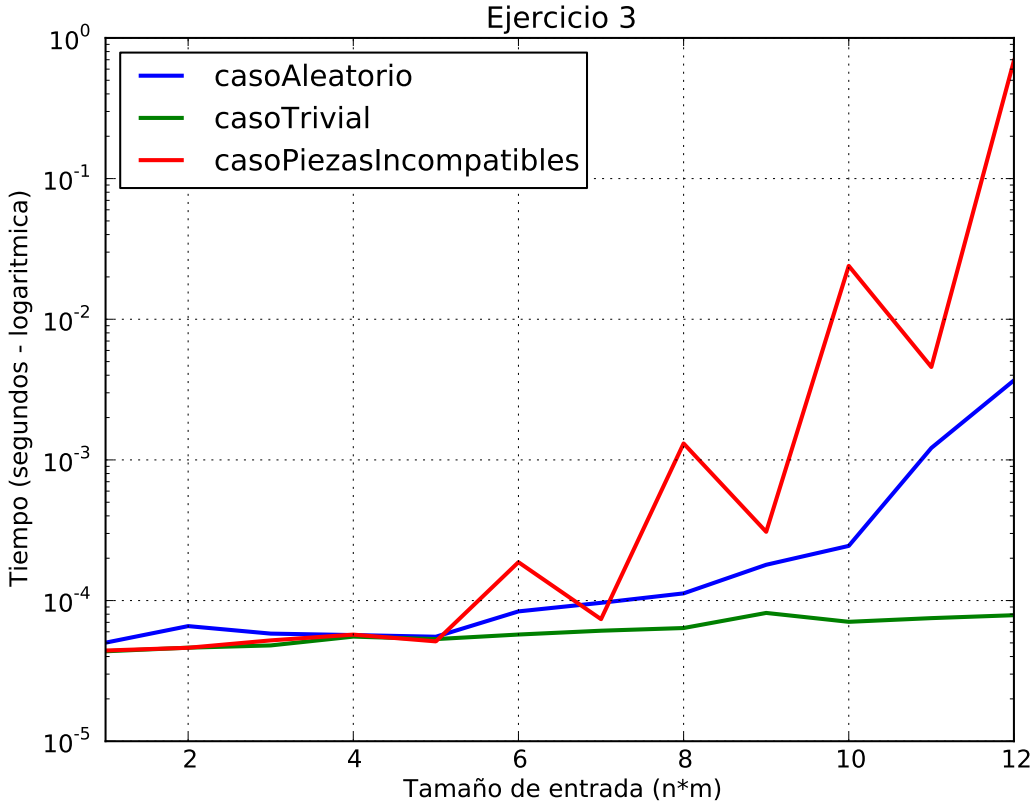

Figura 6: Comparación de la complejidad según el tipo de entrada

[FIXME] Explicar esto

4. Apéndices

4.1. Código Fuente (resumen)

4.1.1. Problema 3: Rompecolores

Listing 1: ej3.cpp

```
1
2  #include "ej3.h"
3  #include "Tablero.h"
4  #include "IndiceDePiezas.h"
5
6  Tablero& backtrack( Tablero&, IndiceDePiezas&, uint32_t );
7  void imprimeTablero( Tablero& );
8
9  int main( int argc, char** argv )
10 {
11   // Parseo los parametros con que fue llamado el ejecutable
12   ParserDeParametros parser( argc, argv );
13   // Esta clase representa un caso de prueba, y lo toma desde el input que
         le provee el parser
14   TestCaseEj3 testcase ( parser.dameInput() );
15
16   // Itero sobre los distintos casos de prueba hasta obtener un testcase
         nulo
17   while ( testcase.tomarDatos() != false )
18   {
19     // Obtengo los parametros del testcase.
20     uint32_t cantidadDeFilas = testcase.dameCantidadDeFilas();
21     uint32_t cantidadDeColumnas = testcase.dameCantidadDeColumnas();
22     uint32_t cantidadDeColores = testcase.dameCantidadDeColores();
23     vector<TestCaseEj3::Pieza>& listaDePiezas = testcase.dameListaDePiezas
           ();
24     // Inicializo el tablero (estructura auxiliar)
25     Tablero tablero( cantidadDeFilas, cantidadDeColumnas, listaDePiezas );
26     // Inicializo el indice de piezas (estructura auxiliar)
27     IndiceDePiezas indiceDePiezas( cantidadDeColores, listaDePiezas,
           tablero );
28     //Obtengo el mejor tablero a traves de backtracking
29     Tablero& mejorTablero = backtrack( tablero, indiceDePiezas, 0 );
30
31     // Devuelvo el resultado con el formato solicitado
32
33     for ( uint32_t columna = 0; columna < mejorTablero.cantidadDeColumnas;
           columna++ )
34     {
35       for ( uint32_t fila = 0; fila < mejorTablero.cantidadDeFilas; fila++
             )
36       {
37         uint32_t posicion = ( columna * mejorTablero.cantidadDeFilas ) +
               fila;
38         uint32_t pieza = mejorTablero[ posicion ];
39         parser.dameOutput() << pieza << " ";
40       }
41
42       parser.dameOutput() << endl;
43     }
44
45     delete &mejorTablero;
46   }
```

```
47
48    return 0;
49  }
50
51  Tablero& backtrack( Tablero& t, IndiceDePiezas& ip, uint32_t posicion )
52  {
53    DEBUG_ENTER; _C( "Entrando a recursion en posicion: " << posicion + 1 );
54    Tablero* mejorTablero = NULL;
55
56    #ifndef SINPODAOBJETIVO
57    if ( t.yaEncontreUnTableroMejor( posicion ) )
58    {
59      _C( "PODANDO EN POS " << posicion <<" PORQUE EXISTE UN TABLERO mejor
          QUE CUALQUIERA DE ESTA RAMA" );
60      return *mejorTablero;
61    }
62    #endif
63
64
65    IteradorIndiceDePiezas& it = ip.dameIterador( posicion );
66    _C("Obtenido iterador al indice de piezas");
67
68    // Me fijo si estoy antes de la ultima posicion
69    if ( posicion < t.cantidadDePosiciones - 1 )
70    {
71      while ( it.hayPiezasPosibles() )
72      {
73        _C( "Pieza disponible: " << *it );
74        // Si es asi, entonces llamo a backtrack para cada pieza posible
75        t.ponerPiezaEnPosicion( *it, posicion );
76        ip.marcarPiezaUtilizada( it );
77        // Hago recursion en el backtracking
78        Tablero* otroTablero = NULL;
79        otroTablero = &( backtrack( t, ip, posicion + 1 ) );
80        _C( "VOLVIENDO A POSICION " << posicion );
81
82        if ( !mejorTablero )
83        {
84          mejorTablero = otroTablero;
85        }
86        else
87        {
88          if ( !otroTablero )
89          {
90            ip.marcarPiezaDisponible ( it );
91            // Si la recursion me devolvio un puntero a nulo termino el BT
92            break;
93          }
94          else
95          {
96            if ( *mejorTablero < *otroTablero )
97            {
98              delete mejorTablero;
99              mejorTablero = otroTablero;
100           }
101           else
102           {
103             delete otroTablero;
104           }
105         }
106       }
107
108       ip.marcarPiezaDisponible ( it );
```

```
109         it++;
110       }
111     }
112     else
113     {
114       // (si estoy en la ultima posicion del tablero)
115       // Intento colocar la ultima pieza
116       if ( it.hayPiezasPosibles() )
117       {
118         t.ponerPiezaEnPosicion( *it, posicion );
119       }
120 
121       // Creo una copia del tablero final
122       mejorTablero = new Tablero( t );
123       // Pongo una pieza transparente en el lugar donde puse (o no) una pieza
124       t.ponerPiezaEnPosicion( TestCaseEj3::PIEZA_VACIA, posicion );
125     }
126 
127     delete (&it);
128     _C( "Saliendo de recursion en posicion: " << posicion + 1 ); DEBUG_ENTER;
129     return *mejorTablero;
130 }
131 
132 /**
133  * Dado un tablero, lo imprime a consola (stderr).
134  */
135 void imprimeTablero( Tablero& t )
136 {
137     for ( uint32_t y = 0; y < t.cantidadDeFilas; y++ )
138     {
139       for ( uint32_t x = 0; x < t.cantidadDeColumnas; x++ )
140       {
141         uint32_t posicion = y * t.cantidadDeColumnas + x;
142         cerr << t[posicion] << " ";
143       }
144 
145       cerr << endl;
146     }
147 
148     cerr << endl;
149 }
```

Listing 2: Tablero.h

```
1
2  #ifndef __TABLERO_H__
3  #define __TABLERO_H__
4  #include "ej3.h"
5
6  class Tablero
7  {
8  private:
9    vector<TestCaseEj3::Pieza>& _listaDePiezas;
10   vector<uint32_t> _piezasEnElTablero;
11   uint32_t _cantidadDePosicionesVacias;
12   uint32_t _mejorCantidadDePosicionesVacias;
13
14   // Poda
15   uint32_t _mejorCantidadDePosicionesVaciasPosible;
16   void _calcularMejorCantidadDePiezasPosible();
17   uint32_t _ultimaPosicionAgregada;
18 public:
19   Tablero( uint32_t p_filas, uint32_t p_columnas, vector<TestCaseEj3::Pieza
         >& );
20   /**
21    * Operador binario, indica si un tablero es menor a otro
22    */
23   inline bool operator< ( Tablero& t )
24   {
25     return this->_cantidadDePosicionesVacias > t.
           _cantidadDePosicionesVacias;
26   }
27   /**
28    * Permite acceder a la pieza ubicada en una posicion determinada
29    */
30   const inline uint32_t& operator[] ( uint32_t posicion ) const
31   {
32     return this->_piezasEnElTablero[posicion];
33   }
34   /**
35    * Dada una posicion en el tablero,
36    * devuelve la pieza que se encuentra a la izquierda
37    */
38   const uint32_t dameLaPiezaDeIzquierdaDePosicion( uint32_t );
39   /**
40    * Dada una posicion en el tablero,
41    * devuelve la pieza que se encuentra arriba
42    */
43   const uint32_t dameLaPiezaDeArribaDePosicion( uint32_t );
44
45
46   /**
47    * Coloca una pieza en la posicion indicada
48    */
49   void ponerPiezaEnPosicion( uint32_t, uint32_t );
50   const uint32_t cantidadDeFilas;
51   const uint32_t cantidadDeColumnas;
52   const uint32_t cantidadDePosiciones;
53   inline const uint32_t cantidadDePosicionesLlenas( void )
54   {
55     return this->cantidadDePosiciones - this->_cantidadDePosicionesVacias;
56   };
57
58   bool yaEncontreElMejorTableroPosible( void );
```

```
59    bool yaEncontreUnTableroMejor( uint32_t );
60    void imprimeTablero();
61  private:
62
63  };
64
65  #endif
```

Listing 3: Tablero.cpp

```
1
2  #include "Tablero.h"
3
4  Tablero::Tablero( uint32_t p_filas, uint32_t p_columnas, vector<TestCaseEj3
       ::Pieza>& listaDePiezas )
5    :
6    _listaDePiezas( listaDePiezas ),
7    _cantidadDePosicionesVacias( p_filas* p_columnas ),
8    _mejorCantidadDePosicionesVacias( _cantidadDePosicionesVacias/2 + 1 ),
9    cantidadDeFilas( p_filas ),
10   cantidadDeColumnas( p_columnas ),
11   cantidadDePosiciones( p_filas* p_columnas )
12 {
13   this->_piezasEnElTablero.assign( p_filas * p_columnas, 0 );
14   // Poda
15   this->_calcularMejorCantidadDePiezasPosible();
16 }
17 const uint32_t Tablero::dameLaPiezaDeArribaDePosicion( uint32_t posicion )
18 {
19   bool noEsPrimeraFila = ( posicion >= this->cantidadDeColumnas );
20
21   if ( noEsPrimeraFila )
22   {
23     _C( "La pieza arriba de la posicion " << posicion + 1 << " es: " <<
24         this->_piezasEnElTablero[posicion - this->cantidadDeColumnas] );
25     return this->_piezasEnElTablero[posicion - this->cantidadDeColumnas];
26   }
27   else
28   {
29     _C( "La posicion " << posicion + 1 << " contiene una pieza vacia a
           arriba" );
30     return TestCaseEj3::PIEZA_VACIA;
31   }
32 }
33 const uint32_t Tablero::dameLaPiezaDeIzquierdaDePosicion( uint32_t posicion
       )
34 {
35   bool noEsPrimeraColumna = ( posicion % this->cantidadDeFilas != 0 );
36
37   if ( noEsPrimeraColumna )
38   {
39     _C( "La pieza izquierda de la posicion " << posicion + 1 << " es: " <<
           this->_piezasEnElTablero[posicion - 1] );
40     return this->_piezasEnElTablero[posicion - 1];
41   }
42   else
43   {
44     _C( "La posicion " << posicion + 1 << " contiene una pieza vacia a su
           izquierda" );
45     return TestCaseEj3::PIEZA_VACIA;
46   }
47 }
48
49 void Tablero::ponerPiezaEnPosicion( uint32_t pieza, uint32_t posicion )
50 {
51   _C( "Poniendo pieza: " << pieza << " en posicion: " << posicion + 1 );
52
53   if ( this->_piezasEnElTablero[posicion] == TestCaseEj3::PIEZA_VACIA )
54   {
55     if ( pieza != TestCaseEj3::PIEZA_VACIA )
```

```
56      {
57        this->_cantidadDePosicionesVacias--;
58        this->_piezasEnElTablero[posicion] = pieza;
59
60        if ( this->_cantidadDePosicionesVacias < this->
              _mejorCantidadDePosicionesVacias )
61        {
62          _C( "Se encontro un nuevo mejor tablero, con " << this->
                _cantidadDePosicionesVacias << " posiciones vacias, la mejor
                era " <<
63               this->_mejorCantidadDePosicionesVacias << "." );
64          this->_mejorCantidadDePosicionesVacias = this->
                _cantidadDePosicionesVacias;
65        }
66      }
67    }
68    else
69    {
70      if ( pieza == TestCaseEj3::PIEZA_VACIA )
71      {
72        this->_cantidadDePosicionesVacias++;
73      }
74
75      this->_piezasEnElTablero[posicion] = pieza;
76    }
77
78  #ifdef DEBUG
79    this->imprimeTablero();
80  #else
81
82  #endif
83    this->_ultimaPosicionAgregada = posicion;
84  }
85
86  bool Tablero::yaEncontreElMejorTableroPosible( void )
87  {
88    return _mejorCantidadDePosicionesVacias <= this->
          _mejorCantidadDePosicionesVaciasPosible;
89  }
90  bool Tablero::yaEncontreUnTableroMejor ( uint32_t posicion )
91  {
92    DEBUG_INT(this->_mejorCantidadDePosicionesVacias);
93    DEBUG_INT(this->cantidadDePosiciones);
94    DEBUG_INT(posicion);
95    DEBUG_INT(this->_mejorCantidadDePosicionesVacias + this->
          cantidadDePosiciones - posicion);
96    DEBUG_INT(_cantidadDePosicionesVacias);
97    return this->_mejorCantidadDePosicionesVacias + this->
          cantidadDePosiciones - posicion
98    <= _cantidadDePosicionesVacias;
99  }
100 void Tablero::imprimeTablero()
101 {
102   for ( uint32_t y = 0; y < this->cantidadDeFilas; y++ )
103   {
104     for ( uint32_t x = 0; x < this->cantidadDeColumnas; x++ )
105     {
106       uint32_t posicion = y * this->cantidadDeColumnas + x;
107       cerr << Tablero::operator[]( posicion ) << " ";
108     }
109
110     cerr << endl;
111   }
```

```
112 }
113 void Tablero::_calcularMejorCantidadDePiezasPosible()
114 {
115   _mejorCantidadDePosicionesVaciasPosible = 0;
116   return;
117
118   multiset<uint32_t> coloresIzquierda;
119   multiset<uint32_t> coloresDerecha;
120   multiset<uint32_t> coloresArriba;
121   multiset<uint32_t> coloresAbajo;
122
123   for ( uint32_t i = 1; i < this->_listaDePiezas.size(); i++ )
124   {
125     coloresIzquierda.insert( this->_listaDePiezas[i].colorIzquierda );
126     coloresDerecha.insert( this->_listaDePiezas[i].colorDerecha );
127     coloresArriba.insert( this->_listaDePiezas[i].colorArriba );
128     coloresAbajo.insert( this->_listaDePiezas[i].colorAbajo );
129   }
130
131   multiset<uint32_t>::iterator it, it2;
132
133   for ( it = coloresIzquierda.begin(); it != coloresIzquierda.end(); it++ )
134   {
135     it2 = coloresDerecha.find( *it );
136
137     if ( it2 != coloresDerecha.end() )
138     {
139       coloresDerecha.erase( it2 );
140     }
141   }
142
143   for ( it = coloresDerecha.begin(); it != coloresDerecha.end(); it++ )
144   {
145     it2 = coloresIzquierda.find( *it );
146
147     if ( it2 != coloresIzquierda.end() )
148     {
149       coloresIzquierda.erase( it2 );
150     }
151   }
152
153   for ( it = coloresArriba.begin(); it != coloresArriba.end(); it++ )
154   {
155     it2 = coloresAbajo.find( *it );
156
157     if ( it2 != coloresAbajo.end() )
158     {
159       coloresAbajo.erase( it2 );
160     }
161   }
162
163   for ( it = coloresAbajo.begin(); it != coloresAbajo.end(); it++ )
164   {
165     it2 = coloresArriba.find( *it );
166
167     if ( it2 != coloresArriba.end() )
168     {
169       coloresArriba.erase( it2 );
170     }
171   }
172
173   uint32_t piezasInfumables = max( coloresDerecha.size(), max(
        coloresIzquierda.size(), max( coloresArriba.size(), coloresAbajo.size
```

```
        () ) ) );
174     _C( "Atencion: Hay como minimo " << piezasInfumables << " piezas
            incompatibles en este tablero" );
175     this->_mejorCantidadDePosicionesVaciasPosible = piezasInfumables / 2;
176     this->_mejorCantidadDePosicionesVaciasPosible = 0;
177     _C( "Atencion: Se estima un minimo de " << this->
            _mejorCantidadDePosicionesVaciasPosible << " posiciones en blanco
            para este tablero." );
178
179 }
```

Listing 4: IndiceDePiezas.h

```
1
2  #ifndef __INDICEDEPIEZAS_H__
3  #define __INDICEDEPIEZAS_H__
4  #include "ej3.h"
5  #include "Tablero.h"
6
7  class IteradorIndiceDePiezas;
8  class IndiceDePiezas
9  {
10   friend class IteradorIndiceDePiezas;
11 private:
12   Tablero& _t;
13   vector<TestCaseEj3::Pieza>& _listaDePiezas;
14   vector<IteradorIndiceDePiezas*> _iteradores;
15   typedef vector<uint32_t> listaDePiezas;
16   vector< vector< stack< listaDePiezas > > > _indiceDeDosColores;
17   vector<bool> _indicePiezasDisponibles;
18   stack< listaDePiezas > _indiceSecuencial;
19   void _imprimirIndiceDeDosColores();
20
21 public:
22   IndiceDePiezas( uint32_t, vector<TestCaseEj3::Pieza>&, Tablero& );
23   ~IndiceDePiezas();
24   IteradorIndiceDePiezas& dameIterador( uint32_t );
25   bool puedeColorarPiezaEnPosicion( uint32_t, uint32_t );
26   void marcarPiezaUtilizada( IteradorIndiceDePiezas& );
27   void marcarPiezaDisponible( IteradorIndiceDePiezas& );
28
29 };
30
31 #include "IteradorIndiceDePiezas.h"
32
33 #endif
```

Listing 5: IndiceDePiezas.cpp

```
1
2  #include "IndiceDePiezas.h"
3
4  IndiceDePiezas::IndiceDePiezas( uint32_t p_cantidadDeColores, vector<
       TestCaseEj3::Pieza>& p_listaDePiezas, Tablero& t ):
5    _t( t ),
6    _listaDePiezas( p_listaDePiezas ),
7    _indiceDeDosColores(
8      p_cantidadDeColores + 1,
9      vector< stack< listaDePiezas > >(
10       p_cantidadDeColores + 1,
11       stack< listaDePiezas >(
12         deque<listaDePiezas>( 1, listaDePiezas( 0 ) )
13       )
14     )
15   ),
16   _indicePiezasDisponibles(
17     p_listaDePiezas.size(), true
18   ),
19   _indiceSecuencial(
20     deque<listaDePiezas>( 1, listaDePiezas( 0 ) )
21   )
22 {
23   for ( uint32_t i = 1; i < this->_listaDePiezas.size(); i++ )
24   {
25     TestCaseEj3::Pieza pieza = this->_listaDePiezas[i];
26     uint32_t colorIzquierda = pieza.colorIzquierda;
27     uint32_t colorArriba = pieza.colorArriba;
28     _indiceDeDosColores[colorIzquierda][colorArriba].top().push_back( i );
29     _indiceSecuencial.top().push_back( i );
30   }
31   _imprimirIndiceDeDosColores();
32 }
33
34 void IndiceDePiezas::_imprimirIndiceDeDosColores(){
35   for ( uint32_t i = 1; i<_indiceDeDosColores.size(); i++){
36     for ( uint32_t j = 1; j<_indiceDeDosColores.size(); j++){
37       //_C("i="<<i<<", j="<<j<<" .size = "<< _indiceDeDosColores[i][j].top
           ().size());
38       for ( vector<uint32_t>::iterator it = _indiceDeDosColores[i][j].top()
           .begin(); it!=_indiceDeDosColores[i][j].top().end(); it++ )
39         _C("IC["<<i<<"]["<<j<<"] = "<<*it);
40     }
41   }
42 }
43
44
45 IteradorIndiceDePiezas& IndiceDePiezas::dameIterador( uint32_t posicion )
46 {
47   IteradorIndiceDePiezas* it = NULL;
48   // Obtengo las piezas de la izquierda y de arriba
49   uint32_t piezaIzquierda = _t.dameLaPiezaDeIzquierdaDePosicion( posicion )
       ;
50   uint32_t piezaArriba = _t.dameLaPiezaDeArribaDePosicion( posicion );
51
52   //it = new IteradorIndiceDePiezas( posicion );
53   #ifndef SINPODAOBJETIVO
54   bool arribaVacio = (piezaArriba == TestCaseEj3::PIEZA_VACIA);
55   bool izquierdaVacio = (piezaIzquierda == TestCaseEj3::PIEZA_VACIA);
56   if ( arribaVacio || izquierdaVacio )
```

```
57    {
58      _C("Creando iterador secuencial");
59      it = new IteradorSecuencial( *this, posicion );
60    }
61    else
62    {
63      _C("Creando iterador por colores");
64      it = new IteradorColores( *this, posicion );
65    }
66    #else
67    _C("Creando iterador secuencial");
68    it = new IteradorSecuencial( *this, posicion );
69    #endif
70
71    return *it;
72  }
73  void IndiceDePiezas::marcarPiezaUtilizada( IteradorIndiceDePiezas& it )
74  {
75    _C( "Marcando como utilizada la pieza: " << *it );
76
77    if ( *it == TestCaseEj3::PIEZA_VACIA )
78    {
79      _C( "La pieza es vacia" );
80      it.utilizarPiezaTransparente();
81    }
82    else
83    {
84
85    _imprimirIndiceDeDosColores();
86
87      // Marco la pieza como no disponible
88      this->_indicePiezasDisponibles[*it] = false;
89      // Obtengo la pieza del listado de piezas
90      TestCaseEj3::Pieza pieza = this->_listaDePiezas[*it];
91      _C("Pusheo una copia del indice para esos dos colores");
92      this->_indiceDeDosColores[pieza.colorIzquierda][pieza.colorArriba].push
            (
93        this->_indiceDeDosColores[pieza.colorIzquierda][pieza.colorArriba].
            top()
94      );
95      _C("Borro el elemento de la nueva lista");
96      this->_indiceDeDosColores[pieza.colorIzquierda][pieza.colorArriba].top
            ().erase(
97        lower_bound( this->_indiceDeDosColores[pieza.colorIzquierda][pieza.
            colorArriba].top().begin(),
98                     this->_indiceDeDosColores[pieza.colorIzquierda][pieza.
                         colorArriba].top().end(), *it )
99      );
100     _C("Pusheo una copia del indice secuencial");
101     this->_indiceSecuencial.push(
102       this->_indiceSecuencial.top()
103     );
104     _C("Borro el elemento del indice secuencial");
105     this->_indiceSecuencial.top().erase(
106       lower_bound( this->_indiceSecuencial.top().begin(),
107         this->_indiceSecuencial.top().end(), *it)
108     );
109     DEBUG_INT(this->_indiceSecuencial.top().size());
110   }
111 }
112 void IndiceDePiezas::marcarPiezaDisponible( IteradorIndiceDePiezas& it )
113 {
114   if ( *it != TestCaseEj3::PIEZA_VACIA )
```

```
115     {
116       _C( "Marcando como disponible la pieza: " << *it );
117       // Marco la pieza como disponible
118       this->_indicePiezasDisponibles[*it] = true;
119       // Obtengo la pieza del listado de piezas
120       TestCaseEj3::Pieza pieza = this->_listaDePiezas[*it];
121       this->_indiceDeDosColores[pieza.colorIzquierda][pieza.colorArriba].pop
              ();
122       this->_indiceSecuencial.pop();
123     }
124   }
125
126   bool IndiceDePiezas::puedeColorarPiezaEnPosicion( uint32_t i_pieza,
          uint32_t posicion )
127   {
128     TestCaseEj3::Pieza pieza = this->_listaDePiezas[i_pieza];
129     uint32_t piezaArriba = this->_t.dameLaPiezaDeArribaDePosicion( posicion )
            ;
130     bool arriba = piezaArriba == TestCaseEj3::PIEZA_VACIA || pieza.
            colorArriba == this->_listaDePiezas[piezaArriba].colorAbajo;
131     uint32_t piezaIzquierda = this->_t.dameLaPiezaDeIzquierdaDePosicion(
            posicion );
132     bool izquierda = piezaIzquierda == TestCaseEj3::PIEZA_VACIA || pieza.
            colorIzquierda == this->_listaDePiezas[piezaIzquierda].colorDerecha;
133     return arriba && izquierda;
134   }
```

Listing 6: IndiceDeColores.h

```
1
2  #ifndef __INDICEDECOLORES_H__
3  #define __INDICEDECOLORES_H__
4  #include "ej3.h"
5
6  class IndiceDeColores
7  {
8  private:
9    struct piezaIndexada
10   {
11     piezaIndexada (uint32_t ip, uint32_t id) : indiceEnListaDePiezas(ip),
           indiceEnListaDisponibles(id) {};
12     uint32_t indiceEnListaDePiezas;
13     uint32_t indiceEnListaDisponibles;
14   };
15   struct listaDisponibles
16   {
17     listaDisponibles( ) : principio(0){ };
18     vector< piezaIndexada > lista;
19     uint32_t principio;
20   };
21   vector<TestCaseEj3::Pieza> &listaDePiezas;
22   vector< vector< listaDisponibles > > v;
23   inline uint32_t colorDerecha (uint32_t indice)
24   {
25     return this->listaDePiezas[indice].colorDerecha - 1;
26   }
27   inline uint32_t colorAbajo (uint32_t indice)
28   {
29     return this->listaDePiezas[indice].colorAbajo - 1;
30   }
31 public:
32   struct Iterador{
33     IndiceDeColores& _ic;
34     inline void quitarPieza()
35     {
36       (this->_ic.*_quitarPieza)( *this );
37     }
38     vector< vector< listaDisponibles > >& _v;
39     vector< piezaIndexada >* _v_res;
40     uint32_t _indice;
41     bool _primeraColumna;
42     bool _primeraFila;
43     uint32_t _colorDerechaActual;
44     uint32_t _colorAbajoActual;
45     bool _piezaTransparenteUtilizada;
46     void (Iterador::*_avanza)();
47     bool (Iterador::*_hayPosibles)();
48     void (IndiceDeColores::*_quitarPieza)( IndiceDeColores::Iterador & );
49     /**
50      * Inicializa el iterador
51      */
52     inline void _avanzaIteradorNormal() {
53       this->_indice++;
54     }
55     inline void _avanzaIteradorPrimerCasillero()
56     {
57       _C("_avanzaIteradorPrimerCasillero();");
58       this->_avanzaIteradorNormal();
59       if ( !this->_hayPosiblesNormal() )
```

```
60        {
61          _C("No hay posibles en este color " << this->_colorAbajoActual);
62          listaDisponibles l;
63          while(true)
64          {
65            this->_colorAbajoActual++;
66            if (this->_colorAbajoActual >= this->_v.size())
67            {
68              this->_colorAbajoActual = 0;
69              this->_colorDerechaActual++;
70            }
71            l = _v[this->_colorDerechaActual][this->_colorAbajoActual];
72            if ((l.lista.size() - l.principio) > 0)
73            {
74              _C("Encontrado color disponible: derecha:" << this->
                    _colorDerechaActual << ", abajo: " << this->
                    _colorAbajoActual);
75              break;
76            }
77          }
78          this->_v_res = &(l.lista);
79          this->_indice = l.principio;
80        }
81      }
82      inline void _avanzaIteradorPrimeraColumna(){
83        _C("_avanzaIteradorPrimeraColumna();");
84        this->_avanzaIteradorNormal();
85        if ( !this->_hayPosiblesNormal() )
86        {
87          _C("No hay posibles en este color " << this->_colorAbajoActual);
88          listaDisponibles l;
89          while(true)
90          {
91            this->_colorDerechaActual++;
92             l = _v[this->_colorDerechaActual][this->_colorAbajoActual];
93            if ((l.lista.size() - l.principio) > 0)
94            {
95              _C("Encontrado color disponible: derecha:" << this->
                    _colorDerechaActual << ", abajo: " << this->
                    _colorAbajoActual);
96              break;
97            }
98          }
99          this->_v_res = &(l.lista);
100         this->_indice = l.principio;
101       }
102     }
103     inline void _avanzaIteradorPrimeraFila()
104     {
105       _C("_avanzaIteradorPrimeraFila();");
106       this->_avanzaIteradorNormal();
107       if ( !this->_hayPosiblesNormal() )
108       {
109         _C("No hay posibles en este color " << this->_colorAbajoActual);
110         listaDisponibles l;
111         while(true)
112         {
113           this->_colorAbajoActual++;
114            l = _v[this->_colorDerechaActual][this->_colorAbajoActual];
115           if ((l.lista.size() - l.principio) > 0)
116           {
117             _C("Encontrado color disponible: derecha:" << this->
                    _colorDerechaActual << ", abajo: " << this->
```

```
                _colorAbajoActual);
118               break;
119             }
120           }
121           this->_v_res = &(l.lista);
122           this->_indice = l.principio;
123         }
124       }
125       inline bool _hayPosiblesNormal()
126       {
127         _C("Verificando hayPosiblesNormal. indice: "<< this->_indice <<",
              this->_v_res->size=" << this->_v_res->size());
128         DEBUG_BOOL(this->_indice < this->_v_res->size());
129         return (this->_indice < this->_v_res->size());
130       }
131       inline bool _hayPosiblesPrimeraColumna()
132       {
133         if (this->_hayPosiblesNormal())
134         {
135           return true;
136         }
137         _C("hayPosiblesPrimeraColumna: Debe cambiar de color");
138         DEBUG_INT(this->_colorDerechaActual);
139         for(uint32_t i = this->_colorDerechaActual+1;i<this->_v.size();i++)
140         {
141           listaDisponibles l = _v[i][this->_colorAbajoActual];
142           if ((l.lista.size() - l.principio) > 0)
143           {
144             return true;
145           }
146         }
147         return false;
148       }
149       inline bool _hayPosiblesPrimeraFila()
150       {
151         if (this->_hayPosiblesNormal())
152         {
153           return true;
154         }
155         _C("HayPosiblesPrimeraFila: Debe cambiar de color");
156         DEBUG_INT(this->_colorAbajoActual);
157         for(uint32_t i = this->_colorAbajoActual+1;i<this->_v.size();i++)
158         {
159           listaDisponibles l = _v[this->_colorDerechaActual][i];
160           if ((l.lista.size() - l.principio) > 0)
161           {
162             return true;
163           }
164         }
165         return false;
166       }
167       inline bool _hayPosiblesPrimerCasillero()
168       {
169         if (this->_hayPosiblesNormal())
170         {
171           return true;
172         }
173         for(uint32_t i = this->_colorAbajoActual; i<this->_v.size(); i++)
174         {
175           for(uint32_t j = this->_colorDerechaActual+1; j<this->_v.size(); j
                ++)
176           {
177             listaDisponibles l = _v[j][i];
```

```
178           if ((l.lista.size() - l.principio) > 0)
179           {
180             return true;
181           }
182         }
183       }
184       return false;
185     }
186     Iterador( IndiceDeColores& p_ic, uint32_t p_i, uint32_t p_a ) :
187       _ic(p_ic),
188       _v(p_ic.v),
189       _primeraColumna(p_i==TestCaseEj3::PIEZA_VACIA),
190       _primeraFila(p_a==TestCaseEj3::PIEZA_VACIA),
191       _piezaTransparenteUtilizada(false)
192       {
193         // Si es primera columna pero no primera fila
194         if(this->_primeraColumna && !this->_primeraFila)
195         {
196           _C("IC::IT() El casillero es primera columna pero no es primera
                fila");
197           this->_avanza = &Iterador::_avanzaIteradorPrimeraColumna;
198           this->_hayPosibles = &Iterador::_hayPosiblesPrimeraColumna;
199           this->_colorDerechaActual = 0;
200           this->_colorAbajoActual = _ic.colorAbajo(p_a);
201         }
202         // Si es primera fila pero no primera columna
203         if(this->_primeraFila && !this->_primeraColumna)
204         {
205           _C("IC::IT() El casillero es primera fila pero no es primera
                columna");
206           this->_avanza = &Iterador::_avanzaIteradorPrimeraFila;
207           this->_hayPosibles = &Iterador::_hayPosiblesPrimeraFila;
208           this->_colorDerechaActual = _ic.colorDerecha(p_i);
209           this->_colorAbajoActual = 0;
210         }
211         // Si no es ni primera columna ni primera fila
212         if(!this->_primeraFila && !this->_primeraColumna)
213         {
214           _C("IC::IT() El casillero no es primera columna ni primera fila")
                ;
215           this->_avanza = &Iterador::_avanzaIteradorNormal;
216           this->_hayPosibles = &Iterador::_hayPosiblesNormal;
217           this->_colorDerechaActual = _ic.colorDerecha(p_i);
218           this->_colorAbajoActual = _ic.colorAbajo(p_a);
219         }
220         // Si es el primer casillero (1ra columna y 1ra fila a la vez)
221         if(this->_primeraColumna && this->_primeraFila)
222         {
223           _C("IC::IT() El casillero es primera columna y primera fila (1er
                casillero)");
224           this->_avanza = &Iterador::_avanzaIteradorPrimerCasillero;
225           this->_hayPosibles = &Iterador::_hayPosiblesPrimerCasillero;
226           this->_quitarPieza = &IndiceDeColores::_quitarPieza;
227           this->_colorDerechaActual = 0;
228           this->_colorAbajoActual = 0;
229           //this->_indice = 1;
230         }
231         else
232         {
233           this->_quitarPieza = &IndiceDeColores::_quitarPieza;
234         }
235         this->_v_res = &(this->_v[_colorDerechaActual][_colorAbajoActual].
              lista);
```

```
236         this->_indice = this->_v[this->_colorDerechaActual][this->
                _colorAbajoActual].principio;
237         _C("IC::IT() colorDerechaActual= "<< this->_colorDerechaActual <<",
                 colorAbajoActual= " << this->_colorAbajoActual << ", indice= "
                 << this->_indice);
238         if(this->_v_res->size() == 0)
239         {
240           _C("IC::IT(); avanzando hasta colorDerechaActual= "<< this->
                  _colorDerechaActual <<", colorAbajoActual= " << this->
                  _colorAbajoActual << ", indice= " << this->_indice);
241           (this->*_avanza)();
242         }
243       }
244     /**
245      * Describe si hay piezas disponibles en el iterador
246      */
247     bool hayPiezasPosibles( void )
248     {
249       if ((this->*_hayPosibles)())
250       {
251         _C("hayPosibles = true");
252         return true;
253       }
254       _C("hayPosibles = false, utilizando PIEZA TRANSPARENTE");
255       return !this->_piezaTransparenteUtilizada;
256     }
257     /**
258      * Avanza el iterador.
259      * El usuario debe revisar si hayPiezasPosibles() primero.
260      */
261     inline Iterador& operator++( int )
262     {
263       if (!((this->*_hayPosibles)())){
264         _C("Avanzando el iterador. Ya no quedan piezas. (pieza transparente
                ).");
265         this->_piezaTransparenteUtilizada = true;
266         return *this;
267       }
268       _C("Avanzando el iterador normalmente");
269       (this->*_avanza)();
270       return *this;
271     }
272     /**
273      * Devuelve el contenido del iterador
274      */
275     inline uint32_t operator*( void )
276     {
277       if (!((this->*_hayPosibles)()))
278       {
279         return TestCaseEj3::PIEZA_VACIA;
280       }
281       DEBUG_INT(this->_indice);
282       DEBUG_INT((*this->_v_res).size());
283       return (*this->_v_res)[this->_indice].indiceEnListaDePiezas;
284
285     }
286   };
287   IndiceDeColores( uint32_t p_cantidadDeColores, vector<TestCaseEj3::Pieza>
          &p_listaDePiezas )
288   : listaDePiezas(p_listaDePiezas), v(p_cantidadDeColores, vector<
        listaDisponibles>(3, listaDisponibles()) ){
289     //this->_v_pieza_indexada.push_back(IndiceDeColores::piezaIndexada(0,0)
          );
```

```
290     // Agrego las piezas al indice de piezas por color, me salteo pieza
            vacia
291     for (uint32_t i = 1; i<this->listaDePiezas.size(); i++){
292       TestCaseEj3::Pieza pieza = this->listaDePiezas[i];
293       uint32_t colorDerecha = this->colorDerecha(i);
294       uint32_t colorAbajo = this->colorAbajo(i);
295       v[colorDerecha][colorAbajo].lista.push_back(IndiceDeColores::
              piezaIndexada(i,v[colorDerecha][colorAbajo].lista.size()));
296       //this->_v_pieza_indexada.push_back(IndiceDeColores::piezaIndexada(i,
              v[colorDerecha][colorAbajo].lista.size()));
297     }
298
299   };
300
301     /**
302      * Dado un ID de pieza, lo quita del indice
303      */
304     void _quitarPieza ( Iterador &it )
305     {
306       uint32_t colorDerecha = this->colorDerecha(*it);// listaDePiezas[*it
              ].colorDerecha;
307       uint32_t colorAbajo = this->colorAbajo(*it);// listaDePiezas[*it].
              colorAbajo;
308       listaDisponibles& listaDisp = v[colorDerecha][colorAbajo];
309       DEBUG_INT(it._indice);
310       DEBUG_INT((*it._v_res)[it._indice].indiceEnListaDisponibles);
311       DEBUG_INT(listaDisp.principio);
312       // Me aseguro que el elemento que quiero "borrar" quede al principio
313       if ( (*it._v_res)[it._indice].indiceEnListaDisponibles != listaDisp.
              principio )
314       {
315         // Si no esta al principio, swapeo
316         uint32_t bak = listaDisp.lista[it._indice].indiceEnListaDePiezas;
317         listaDisp.lista[it._indice].indiceEnListaDePiezas = listaDisp.lista
                [listaDisp.principio].indiceEnListaDePiezas;
318         listaDisp.lista[listaDisp.principio].indiceEnListaDePiezas = bak;
319       }
320       // Aumento el "principio" de la lista
321       (listaDisp.principio)++;
322     }
323
324     void restaurarPieza ( IndiceDeColores::Iterador& it )
325     {
326       uint32_t colorDerecha = this->colorDerecha(*it);// listaDePiezas[*it
              ].colorDerecha;
327       uint32_t colorAbajo = this->colorAbajo(*it);// listaDePiezas[*it].
              colorAbajo;
328       listaDisponibles& listaDisp = v[colorDerecha][colorAbajo];
329       // Disminuyo el "principio" de la lista
330       (listaDisp.principio)--;
331       // Si no estaba al principio, restauro el elemento a su posicion
              original
332       if (it._indice != listaDisp.principio)
333       {
334         uint32_t bak = listaDisp.lista[it._indice].indiceEnListaDePiezas;
335         listaDisp.lista[it._indice] = listaDisp.lista[listaDisp.principio];
336         listaDisp.lista[listaDisp.principio].indiceEnListaDePiezas = bak;
337       }
338     }
339   /**
340    * Dadas las piezas de la izquierda y de arriba devuelve una lista
341    * con los ID de las posibles piezas que se pueden agregar
342    */
```

```
343     Iterador &damePiezasPosibles( uint32_t piezaIzquierda, uint32_t
            piezaArriba)
344     {
345       if(piezaIzquierda != TestCaseEj3::PIEZA_VACIA && piezaArriba !=
              TestCaseEj3::PIEZA_VACIA){ _C("IC::IT damePiezasPosibles("<<
              piezaIzquierda <<","<< piezaArriba <<") -> con piezas no nulas");}
346       if(piezaIzquierda == TestCaseEj3::PIEZA_VACIA && piezaArriba !=
              TestCaseEj3::PIEZA_VACIA){ _C("IC::IT damePiezasPosibles("<<
              piezaIzquierda <<","<< piezaArriba <<") -> con pieza izquierda nula
              ");}
347       if(piezaIzquierda != TestCaseEj3::PIEZA_VACIA && piezaArriba ==
              TestCaseEj3::PIEZA_VACIA){ _C("IC::IT damePiezasPosibles("<<
              piezaIzquierda <<","<< piezaArriba <<") -> con pieza arriba nula")
              ;}
348       if(piezaIzquierda == TestCaseEj3::PIEZA_VACIA && piezaArriba ==
              TestCaseEj3::PIEZA_VACIA){ _C("IC::IT damePiezasPosibles("<<
              piezaIzquierda <<","<< piezaArriba <<") -> con ambas piezas nulas")
              ;}
349       Iterador *it = new Iterador( *this, piezaIzquierda, piezaArriba );
350       return *it;
351     }
352
353   private:
354
355   };
356
357   #endif
```

Listing 7: IteradorIndiceDePiezas.h

```
1
2  #include "IndiceDePiezas.h"
3  class IteradorIndiceDePiezas
4  {
5  private:
6    bool _piezaTransparenteUtilizada;
7    bool _utilizarPiezaTransparente;
8  protected:
9    vector<bool>& _indicePiezasDisponibles;
10   vector<uint32_t>& _indiceSecuencial;
11   vector<uint32_t>* _indiceColores;
12   IndiceDePiezas::listaDePiezas* _v;
13   IndiceDePiezas::listaDePiezas::iterator _v_it;
14   //IteradorIndiceDePiezas ( IteradorIndiceDePiezas::*avanzar )( int );
15   IndiceDePiezas& _ip;
16   uint32_t _posicion;
17 public:
18   IteradorIndiceDePiezas( IndiceDePiezas&, uint32_t );
19   virtual ~IteradorIndiceDePiezas() = 0;
20   virtual bool hayPiezasPosibles() = 0;
21   virtual IteradorIndiceDePiezas& operator++( int );
22   virtual uint32_t operator*();
23   void utilizarPiezaTransparente();
24 };
25
26 #include "IteradorSecuencial.h"
27 #include "IteradorColores.h"
28 #endif
```

Listing 8: IteradorIndiceDePiezas.cpp

```
1
2  #include "IteradorIndiceDePiezas.h"
3
4  IteradorIndiceDePiezas::IteradorIndiceDePiezas( IndiceDePiezas& ip,
       uint32_t posicion )
5    : _indicePiezasDisponibles( ip._indicePiezasDisponibles ),
6      _indiceSecuencial ( ip._indiceSecuencial.top() ),
7
8      _ip( ip ),
9      _posicion( posicion )
10 {
11   uint32_t piezaIzquierda = ip._t.dameLaPiezaDeIzquierdaDePosicion(
         posicion );
12   uint32_t piezaArriba = ip._t.dameLaPiezaDeArribaDePosicion( posicion );
13   if(piezaIzquierda!=TestCaseEj3::PIEZA_VACIA && piezaArriba !=TestCaseEj3
         ::PIEZA_VACIA)
14   {
15
16     this->_indiceColores = &(ip._indiceDeDosColores
17       [ip._listaDePiezas[piezaIzquierda].colorDerecha]
18       [ip._listaDePiezas[piezaArriba].colorAbajo]
19       .top());
20   }
21   this->_piezaTransparenteUtilizada = false;
22   this->_utilizarPiezaTransparente = false;
23   _C("Inicializado IteradorIndiceDePiezas");
24 }
25
26
27 bool IteradorIndiceDePiezas::hayPiezasPosibles()
28 {
29   return !_piezaTransparenteUtilizada;
30 }
31 IteradorIndiceDePiezas& IteradorIndiceDePiezas::operator++( int )
32 {
33   _C( "IteradorIndiceDePiezas::operator++" );
34   this->_utilizarPiezaTransparente = true;
35   return *this;
36 }
37 uint32_t IteradorIndiceDePiezas::operator*()
38 {
39   if ( this->_utilizarPiezaTransparente )
40   {
41     _C( "Utilizando pieza transparente" );
42     return TestCaseEj3::PIEZA_VACIA;
43   }
44   else
45   {
46     return *( this->_v_it );
47   }
48 }
49 void IteradorIndiceDePiezas::utilizarPiezaTransparente()
50 {
51   this->_piezaTransparenteUtilizada = true;
52 }
```

4.2. Informe de Modificaciones

Se realizaron las siguientes modificaciones:

- Se cambiaron los nombres de las secciones "Hipótesis de Resolución" por "Planteamiento de Resolución"; el contenido sigue siendo el mismo.

`Sección 1:Introducción (página 3)`:

- Correcciones y aclaraciones mínimas en el texto.
- Se creó **`Sección 2:Instrucciones de uso (página 5)`** en donde se aclaran detalles relativos al código y/o scripts provistos en el paquete de entrega, y se movió el apartado de "Herramientas Utilizadas" a **`Subsección 2.1:Herramientas utilizadas (página 5)`**.

`Subsección 3.3:Problema 3: Rompecolores (página 34)`:

- Se realizaron las correcciones marcadas por el profesor en **`3.1.1:Descripción`** (página 6).
- Para la primer parte[19] del contenido que figuraba en la sección "Hipótesis de Resolución", se agregaron aclaraciones a algunas de las correciones hechas, se eliminó/reformuló por completo la abstracción formal matemática[20]. Además, se trasladó todo el contenido de estos párrafos a **`3.1.1:Descripción`** (página 6), ya que consideramos que el análisis realizado en esos párrafos es más propio del modelado/descripción del problema; es decir que es independiente de la resolución que hayamos ideado.
- El texto que figuraba en la segunda parte del contenido de la sección "Hipótesis de Resolución" se eliminó/reformuló por completo, haciendo un análisis y descripción más detallados de los objetivos del algoritmo y las podas implementadas. No se hizo demasiado hincapié en los detalles estrictamente implementativos, sino en las nociones o ideas generales que inspiraron el código.
- El código del problema 3 fue reescrito por completo, utilizando una versión con clases, permitiendo así realizar diversas abstracciones que permitieron (a nuestro gusto) una mejor organización y comprensión del algoritmo. Por ejemplo, se implementó la clase IndiceDePiezas, la cual contiene todo el comportamiento relativo al índice con el que se eligen las piezas que serán utilizadas, y la clase abstracta IteradorIndiceDePiezas, cuyas clases heredadas (IteradorSecuencial, IteradorColores) son instanciadas por el Índice de Piezas, y contienen un comportamiento común a ambas, y un comportamiento propio que depende de la posición del tablero en que se encuentre el iterador.

[19] Consta del análisis del tipo de problema brindado, del universo de soluciones posible, y la formalización de la función objetivo.

[20] A causa de la notación utilizada, que era incorrecta, y de que llegamos a la conclusión de que no era necesario ni deseable ese nivel de "formalidad" en la notación, cuando las mismas ideas pueden bien ser explicadas, sin perder su formalidad, en lenguaje coloquial.

4.3. Ejercicios Adicionales (reentrega)

Problema 3:

1. ¿Cómo debería modificarse el algoritmo implementado para que el mismo funcione en el RompeColores 2.0?

No se debería modificar de manera sustancial. Ahora hay que chequear que al poner una ficha en la última columna, su color derecho concuerde con el color izquierdo de la primera ficha de la fila. Además, al poner una ficha en la última fila, su color inferior debe concordar con el color superior de la primera ficha de la columna. Nuestro algoritmo determinaba las fichas posibles para éstas posiciones solamente de acuerdo a su color superior e izquierdo. Ahora, antes de colocar la ficha, hay que satisfacer unas condiciones extra. Podemos iterar sobre la lista de piezas que ibamos a colocar ahi en Rompecolores1.0, y preguntar si satisfacen o no la condición.

2. ¿Cómo afecta el nuevo esquema a las podas implementadas en la primera versión?

Con respecto a nuestra optimización de utilizar PiezasPorColores, se podrían mantener otras estructuras que de acuerdo a 3 colores "izquierda, arriba, derecha", o "izquierda, arriba, abajo", nos devuelva todas las piezas que tienen esos colores en tales bordes. De esta forma podemos saber rápidamente que fichas se pueden poner cuando estamos en una posición de la última columna o última fila. Implicaría un mayor costo temporal y espacial, pero asintóticamente seríá el mismo costo que la versión anterior. La poda de cortar la rama cuando, aún llenando todas las posiciones que me quedan por recorrer, no llegamos a superar al mejor cubrimiento de tablero que encontramos, seguría exactamente igual. Lo mismo ocurre con la poda de terminar la ejecución cuando ya encontramos una solución que cubre todo el tablero.

3. ¿Pueden proponer nuevas podas para este nuevo esquema que no podrían implementarse en el esquema anterior?

Hay ciertas podas que resultan mucho más intuitivas con el nuevo esquema. Intentemos acotar la cantidad de casillas que cubre la solución óptima, y de esta forma parar la ejecución cuando encontramos una solución que llega a la cota. Por ejemplo, para cada ficha que tiene el color azul arriba, si queremos colocarla en el tablero debería existir una ficha con el color azul abajo. La idea es contar, para cada color x de 1 a c, la diferencia entre la cantidad de fichas que tienen arriba x y la cantidad de fichas que tienen abajo x, lo mismo con la diferencia de fichas que lo tienen a la izquierda y a la derecha. Si tomamos "p" la mayor de estas diferencias entre todos los colores, sabemos que hay por lo menos p fichas que no vamos a poder colocar. Luego si llegamos a una solución que coloque n - p fichas, ya podemos parar la ejecución.