UNIVERSIDAD DE BUENOS AIRES

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Algoritmos y Estructuras de Datos III

Trabajo Práctico 1

9 de mayo de 2014

Integrante	LU	Correo electrónico
Contrufo, Maximiliano	336/12	
Vargas Telles, Matías	318/12	

Índice

1. Cambios realizados con respecto a la primera entrega

1.1. Problema 1

Se agregó el informe
Se modificó ligeramente el código para evitar analizar casos repetidos o innecesarios.

1.2. Problema 2

Se agregó el informe
Se modificó el código para no realizar un cociente haciendo un producto en su lugar para ordenar los elementos.

1.3. Problema 3

Se agregó el informe

2. Problema 1: Camiones sospechosos

2.1. Descripción del problema

Sintéticamente el problema planteado pide encontrar en complejidad estrictamente menor que $O(n^2)$ la máxima cantidad de camiones de una tira de n que un inspector puede supervisar en una cantidad d de días sabiendo qué días viene cada camión.

Para resolver este problema nuestra solución consiste en que para cada día en que llega un camión, se procede a chequear qué cantidad de camiones puede inspeccionar nuestro inspector si comenzara ese día. De estas cantidades se obtiene la mayor y se devuelve junto con el día en el cual fue obtenida. En caso de obtener más de una cantidad máxima de camiones inspeccionados, se devuelve cualquiera de ellas. Veamos entonces unos ejemplos:

En este caso el inspector fue contratado por 1 día, i.e. $d = 1$, vienen 2 camiones, $n = 2$, los días 6 y 7. De esta manera el input recibido se vería como:

```
1 2 6 7
```

Para resolver este ejemplo nos fijamos si nuestro inspector comienza el día 6 cuántos camiones puede revisar y lo mismo con el día 7. Como sólo puede inspeccionar por un día, cualquier día que empiece revisará un solo camión. Luego podemos devolver cualquier resultado. Nuestro output seria:

```
6 1
```

aunque también el siguiente output es aceptable:

```
7 1
```

Observemos otra situación similar:

Para este caso la entrada se ve así:

```
2 2 6 7
```

En esta instancia $d = 2$ y $n = 2$. Es directo ver que si el inspector comienza el día 6 logrará detener 2 camiones mientras que, si comenzara el 7, sólo podría detener al último de los vehículos.

El output correspondiente a esta situación es el siguiente:

```
6 2
```

Como último ejemplo, veamos qué sucede con más camiones:

y su input correspondiente:

```
2 4 9 6 6 7
```

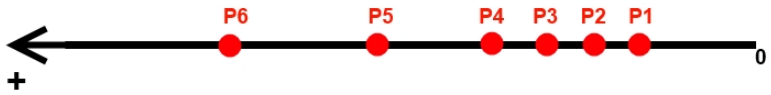
En este caso ($d = 2$, $n = 4$) podemos ver que no solo el input puede venir desordenado sino también que más de un camión puede llegar el mismo día. Dicho esto, una vez más recorremos todos los días de llegada para ver que, comenzando el día 6 podemos capturar 3 camiones mientras que si empezamos el 7 o el 9, sólo podremos detener uno. Por lo tanto, el output de esta instancia se ve de la siguiente manera:

```
6 3
```

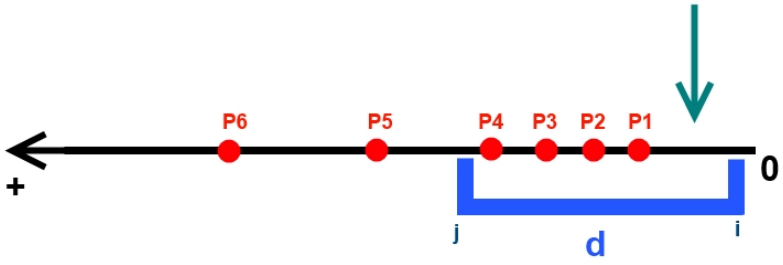
2.2. La solución

Para desarrollar nuestra idea para la resolución de este problema, creemos conveniente tratar por un momento otro problema cuyo resultado aplicaremos luego a nuestro enunciado original. El problema nuevo consiste en el siguiente enunciado:

> Dada una sucesión finita de puntos alineados sobre una recta horizontal (como se ve a continuación) y un intervalo fijo de longitud d, ubicar dicho intervalo sobre la recta de forma tal que contenga a la mayor cantidad de puntos posibles.



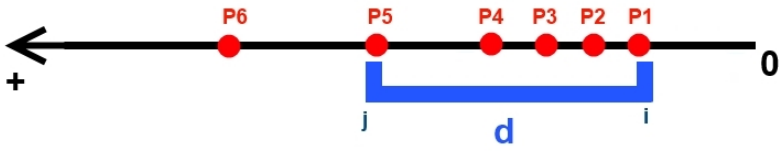
En principio, habría que chequear infinitas soluciones para encontrar la ubicación que resuelve este problema. Sin embargo, si prestamos atención a la siguiente imagen podremos esbozar una idea que nos servirá para evitar tener que pasar por todas las soluciones posibles.



Se puede ver que extremo i del intervalo de longitud d (el extremo más cercano al 0) tiene un espacio entre P1 y dicho extremo que no está siendo ocupado por ningún punto rojo. Este espacio "perdido"puede reutilizarse de forma tal que el extremo i coincida con P1 dejando así la posibilidad de incluir más puntos sin dejar de incluir a los puntos ya incluidos.

En particular, si colocamos un extremo del intervalo en cada uno de los posibles puntos y nos fijamos en qué cantidad de puntos quedan incluidos podemos obtener la mejor posición para el intervalo ya que, cualquier ubicación del extremo del intervalo entre dos puntos puede ser desplazada hasta el punto más cercano al mismo obteniendo así una mejor (o igual) utilización del espacio.

De esta forma logramos reducir la cantidad de posiciones a chequear y podemos resolver el problema de la siguiente manera:



Volviendo a nuestro problema del inspector y los camiones, podemos utilizar el resultado visto arriba y modelar los días en los que llegan los camiones como los puntos sobre la recta y a la longitud del intervalo como los días que puede el inspector realizar su labor.

De esta manera, haciendo que el funcionario comience a trabajar el día que llega un camión podemos ver a partir de qué día logra maximizar la cantidad de camiones que logra inspeccionar.

Recordando que la complejidad pedida debe ser estrictamente inferior a $O(n^2)$ siendo n la cantidad de camiones, tenemos que hacer algunas modificaciones ya que chequear linealmente cada día cuántos camiones puede revisar el inspector es exactamente $O(n^2)$ (pues hay que recorrer, por cada n, en el peor caso todo la tira de días nuevamente).

Para resolver este inconveniente decidimos ordenar previamente la lista de días de llegada para luego, de esta forma, por cada día de la misma, contar hasta cuál camión puede revisar el inspector haciendo uso del orden dado.

Esto se lleva a cabo simulando una búsqueda binaria que, en lugar de devolver verdadero o falso según la presencia o no del elemento buscado, retorna la posición del último camión que el inspector puede verificar para cada día de comienzo. Esta búsqueda se puede hacer en $O(log(n))$ permitiéndonos así resolver el problema planteado en $O(nlog(n))$.

En las siguientes secciones exhibiremos el pseudocódigo del algoritmo empleado (y su código fuente) junto con sus respectivas demostraciones de correctitud y complejidad.

2.3. Demostración de correctitud

Algorithm 1: Pseudocódigo para demostrar correctitud. d representa los días por los que fue contratado el inspector y n la cantidad de camiones

```
1  resultado ←0
2  sort(díasDeLlegada)
3  while haya día sin chequear do
4  |  día ←díaSinChequear()
5  |  parcial ←BusquedaBinariaEspecial(díasDeLlegada, día + d - 1, n - 1) - posición(día, díasDeLlegada) + 1
6  |  if parcial > resultado then
7  |  |  resultado ←parcial
8  |  |  díaInicio ←día
9  |  end
10 end
```

Para demostrar que nuestro código es correcto, i.e. devuelve un día de inicio en el cual se maximiza la cantidad de camiones inspeccionados, probaremos, en primer lugar, la idea introducida en la sección anterior. Recordemos:

> Dada una sucesión finita de puntos alineados sobre una recta horizontal y un intervalo fijo de longitud d, para hallar la ubicación del intervalo que maximice la cantidad de puntos dentro del mismo se debe colocar el intervalo de forma tal que uno de sus extremos coincida con uno de los puntos. Luego se procede a contar la cantidad de puntos que quedan dentro del intervalo para esta configuración. Si se repite para cada punto de la sucesión, podremos determinar qué ubicación del intervalo es la mejor.

Veamos qué sucede si un extremo del intervalo no coincide con uno de los puntos. Es decir, sea d_i un extremo del intervalo y P_i cualquier punto de la sucesión, $d_i \neq P_i \forall i \in \mathbb{N}$.

Sea n la cantidad de puntos de la sucesión, sabemos que hay k puntos dentro del intervalo. Si $k = n$, el problema está resuelto. Asumamos entonces que hay $n - k$ puntos fuera del intervalo.

Por la ubicación del intervalo, hay un i tal que la distancia de P_i a d_i es mínima y es mayor que cero. Llamemos a esta distancia h. Tenemos entonces que entre d_i y $d_i + h$ no hay ningún punto de la sucesión. Luego, podemos reemplazar, sin perder ningún punto de dentro del intervalo, d_i por $d_i + h$ (desplazando a su vez a d_j (el otro extremo) en h).

Si entre d_j y $d_j + h$ no había ningún punto de los $n - k$ que no estaban en el intervalo original, entonces el nuevo intervalo tendrá la misma cantidad de puntos. Sin embargo, si hubiera allí algún punto, el nuevo intervalo tendría más puntos que el original.

De esta forma queda probado que la mejor forma de posicionar el intervalo es haciéndolo de forma tal que coincida su extremo con alguno de los puntos. Aplicando esto a nuestro problema, podemos decir que la mejor manera de elegir un día para que empiece a trabajar el inspector es forzando que este día sea uno de los días en los que llegan camiones.

Dicho esto, falta probar que la función *BusquedaBinariaEspecial* devuelve efectivamente la posición en el vector de díasDeLlegada del último camión que puede revisar el inspector. Con este dato, calculando la diferencia de posición en el vector entre el día inicial y el día final (agregando 1 por el *ïndexamiento"* en 0)podremos saber la cantidad de camiones para cada día de comienzo del inspector y así, finamente, estaremos en condiciones de devolver el máximo de estos.

Veamos el pseudocódigo de la función en cuestión:

Algorithm 2: Pseudocódigo para demostración de correctitud

```
1  BusquedaBinariaEspecial(díasDeLlegada, día + d - 1, n - 1)
2  inicio←0
3  mid ←n
4  elem ←día + d - 1
5  while |inicio - mid| > 2 do
6  |  mid ←(n+inicio)/2
7  |  if díasDeLlegada[mid] >elem then
8  |  |  n←mid
9  |  else
10 |  |  inicio ←mid
11 |  end
12 end
13 retorna PosiciónDelMasCercanoPorAbajo(diasDeLlegada_{inicio..n}, elem)
```

El código utilizado es en esencia una búsqueda binaria que se detiene sólo cuando quedan 2 elementos (salvo en el caso trivial de 1 solo día).Como sabemos, la búsqueda binaria va *encerrando* el valor buscado a cada paso. Luego estamos seguros que uno de estos 2 valores restantes es el que necesitamos. Hay varios casos posibles. A saber:

- **Hay un elemento mayor y otro menor (o igual) al buscado:** En este caso el algoritmo devuelve la posición del elemento menor o igual ya que es el último camión que puede alcanzar el inspector con los días de su contrato
- **Ambos elementos son menores o iguales:** En este caso la función retorna la posición más grande entre ambos elementos.

Con el resultado de esta búsqueda, el resto del algoritmo incial procede a calcular la cantidad de camiones total del intervalo como la diferencia (más uno) entre las posiciones final e inicial del intervalo a evaluar (debido a que cada elemento del vector de días representa un camión).

Luego de esta prueba podemos entonces afirmar que nuestro algoritmo es correcto ya que busca entre todos los mejores intervalos posibles y devuelve aquel que contenga la mayor cantidad de camiones.

Nuestro algoritmo evita chequear valores de días repetidos puesto que siempre al ,estar ordenado el vector ,con ver el primer caso basta. Tampoco sigue chequeando si encontró una solución que supera a la cantidad de camiones que quedan disponibles. Estos ajustes no están presentes en el pseudocódigo pues es fácil ver que no afectan a la correctitud del algoritmo.

2.4. Demostración de complejidad

Algorithm 3: Pseudocódigo para demostrar complejidad.

```
1  n< −input
2  d< −input
3  díasDeLlegada< −input
4  resultado ←0
5  sort(díasDeLlegada) ▶Ordena de menor a mayor la entrada
6  while haya día sin chequear do
7  |  día ←díaSinChequear()
8  |  parcial ←BusquedaBinariaEspecial(díasDeLlegada, día + d - 1, n - 1) - posición(día, díasDeLlegada) + 1
9  |  if parcial > resultado then
10 |  |  resultado ←parcial
11 |  |  díaInicio ←día
12 |  end
13 end
14 print diaInicio
15 print resultado
```

La función sort tiene una complejidad $O(nlog(n))$ [1] Las asignaciones son $O(1)$ y la función *BusquedaBinariaEspecial* es una búsqueda binaria que llega hasta el final siempre, luego, es el peor caso de la misma que es $O(log(n))$ que se lleva a cabo, en el peor caso, n veces, una por cada día no chequeado.

Es decir, nuestro algoritmo tiene complejidad $O(nlog(n)$ que es estrictamente mejor que $O(n^2)$ como fue solicitado.

2.5. Código fuente

```
typedef vector<int> Vec;
typedef pair<int, int> Coord;

int Bs(const Vec& vec, int n, int elem);
Coord resolver(const Vec& dias, int n, int D);


int main()
{
  int D;
  int n;
  Coord res;
  bool fin=false;
  double time;
  while(!fin) {
    cin >> D;
    if (D==0){
      fin=true; // sale del ciclo cuando encuentra el 0
    }
    else{
    cin >> n;
    Vec dias(n,0);
    for(int i = 0; i < n; ++i){
      cin >> dias[i];
      }
    sort(dias.begin(), dias.end());

    res =resolver(dias, n, D);
    cout<<res.second<<" ";
    cout<<res.first<<endl; //muestro resultado
    }
  }
```

[1] http://es.cppreference.com/w/cpp/algorithm/sort

```
33
34    return 0;
35  }
36    int Bs(const Vec& vec, int n, int elem){
37      int inicio=0;
38      int mid;
39      //bool end =false;
40      while(abs(inicio-n)>1){
41          mid=(n+inicio)/2;
42          if (vec[mid]>elem){
43            n=mid;
44
45          }
46          else{
47            inicio=mid;
48            }
49
50      }
51      if (n!=inicio){
52        if (vec[max(n,inicio)]>elem){
53          return min(n,inicio);
54        }
55        else{return max(n,inicio);}
56      }
57      else{return n;}
58
59    }
60
61
62  Coord resolver(const Vec& dias, int n, int D)
63  {
64    Coord mayor (0,0);
65    int parcial;
66    int ultimoProcesado=-1;
67    //sort(dias.begin(), dias.end());
68    for (int i=0; i<n, mayor.first <= n-1-i; i++){
69      if(ultimoProcesado!=dias[i]){
70      ultimoProcesado=dias[i];
71      parcial=Bs(dias, n-1, (dias[i]+D -1) ) - i + 1  ;
72      if (parcial>mayor.first){
73        mayor.first=parcial;
74        mayor.second=dias[i];
75        }
76      }
77    }
78    return mayor;
79  }
```

Código 1: Descriptive Caption Text

2.6. Tests

2.6.1. Tests de correctitud

Para mostrar la correctitud empírica del algoritmo seleccionamos una variedad de casos que muestran su adecuado funcionamiento. A saber:

- Caso con un camión y 1 día para el inspector.
- Caso con 5 camiones el mismo día y 1 día para el inspector. (Hace solo 1 vez la búsqueda.)
- Caso con 5 camiones en distintos días y 1 día para el inspector. (Hace 4 veces la búsqueda.)

- Caso con 5 camiones en distintos días y 5 días para el inspector. (Llega desde el primer día hasta el final)
- Caso con 10 camiones en distintos días y 2 días para el inspector.
- Caso con 10 camiones con algunos repetidos y 3 días para el inspector.

#	**Input**	**Output del programa**
0	1 1 1 1 5 1 1 1 1 1 1 5 5 2 3 4 1 5 5 1 2 3 4 5 2 10 1 3 5 7 13 11 9 15 17 18 3 10 1 1 1 5 5 5 12 12 12 18 0	1 1 1 5 1 1 1 5 17 2 1 3

2.6.2. Tests de complejidad

Para mostrar la complejidad de nuestra solución decidimos concentrarnos en la sección que realiza la "búsqueda binaria" adaptada dado que a la función *sort* y sus casos particulares serán analizados en el problema 2 del presente informe.

Dado que la cantidad de veces que hace la búsqueda depende del resultado parcial más grande que encontró y de la cantidad de elementos repetidos (porque no analiza los casos que ya analizó), decidimos enfocar la experimentación en las siguientes familias de casos:

- El algoritmo termina en la primer pasada vs El algoritmo hace *n-1* pasadas.
- La entrada no tiene repetidos vs La entrada tiene elementos repetidos.
- Entrada aleatoria

2.6.2.1. Casos de 1 pasada vs *n-1* pasadas Para esta familia de instancias generamos (con valores aleatorios) situaciones en las que la cantidad de días de contrato del inspector pudieran cubrir todos los camiones (y por lo tanto sólo hace 1 pasada) y casos en los que sólo pudiera cubrir 1 camión como solución óptima (razón por la cual debe ir hasta la posición *n-1*). Veamos los resultados:

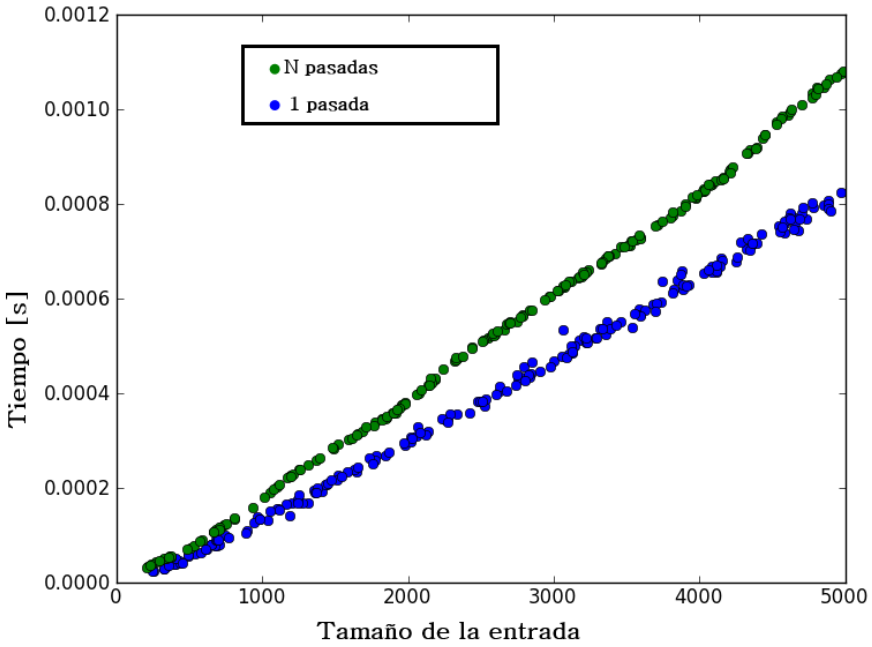


Figura 1: Gráfico comparativo de entrada en función de tiempo para 1 pasada vs n-1 pasadas

Para ver que efectivamente cumple con nuestra complejidad propuesta, dividimos los valores de tiempo de la entrada por $nLog(n)$ y graficamos:

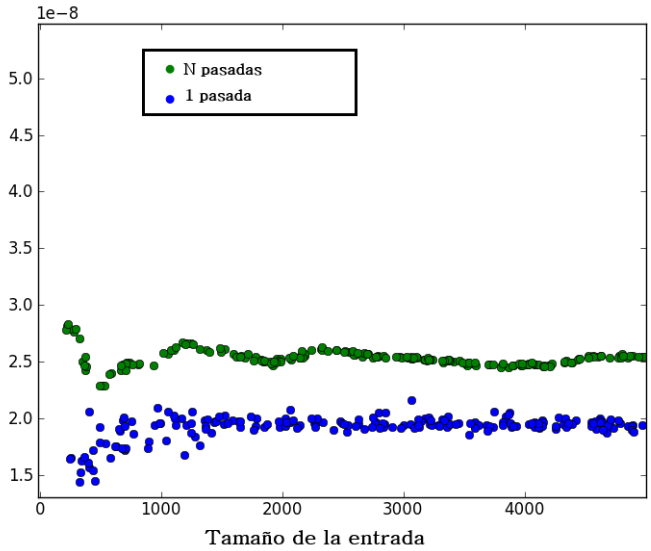


Figura 2: Gráfico de comparación empírica de la complejidad

Se puede ver que para ambas entradas, es posible, con cierto error acotable, establecer una constante de complejidad, es decir, que nuestra solución se ajusta en este caso a la complejidad calculada.

2.6.2.2. Casos sin repeticiones vs repetidos Para estas instancias construimos situaciones en las que los camiones vinieran en grupo en ciertos días (el vector tiene elementos repetidos) y otras en las que no haya más de un camión por día.

Veamos los resultados de esto:

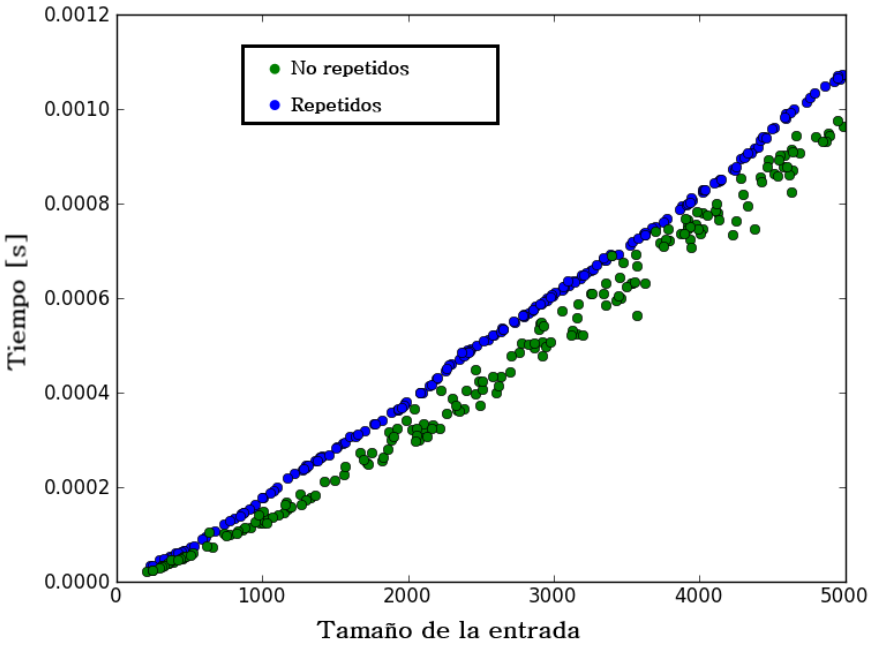


Figura 3: Gráfico comparativo de entrada en función de tiempo para repetidos vs no repetidos

Podemos ver que los casos repetidos varían su comportamiento mientras que los no repetidos parecen comportarse siempre de peor manera. Sin embargo esto no es siempre así. Veamos este otro caso (los colores se mantienen):

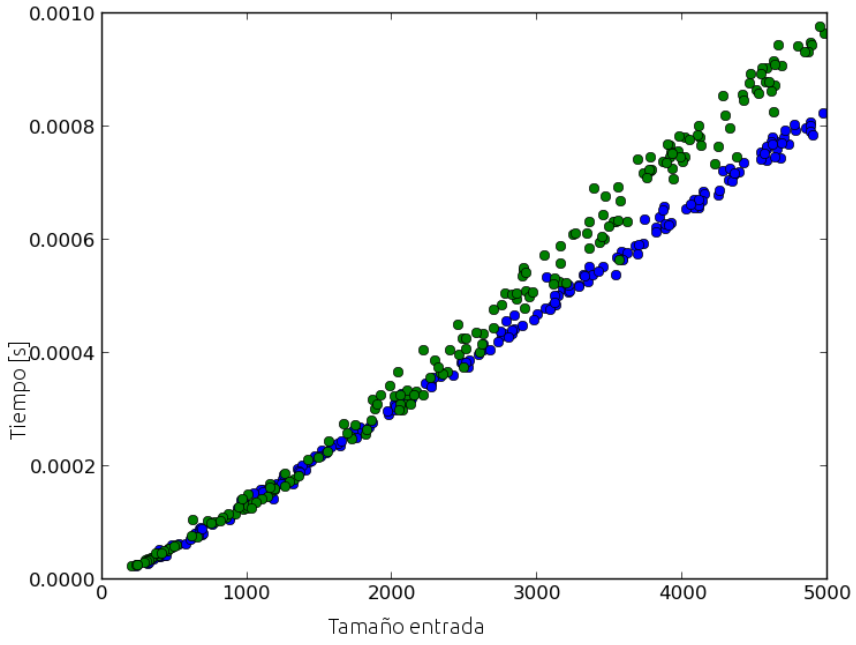


Figura 4: Gráfico comparativo de entrada en función de tiempo para repetidos vs no repetidos

Aquí podemos ver que los no repetidos son cota inferior de los repetidos.

Esto se debe a que no afecta tanto al tiempo la cantidad de las repeticiones sino su distribución. En el primer caso los elementos de la entrada fueron elegidos para estar separados por D días cada camión, razón por la cual el algoritmo hace todas las pasadas de la búsqueda binaria mientras que en el último caso todos los camiones llegan en distinto día pero todos los días son menores a D, luego, basta con una pasada. En el caso de los repetidos, sólo se controló la cantidad de elementos repetidos y a esto se debe la variación en la cantidad de pasadas.

Podemos ver entonces, si volvemos a dividir por los tiempos por $nlog(n)$ para el primer caso, que los elementos no repetidos muestran una constante más definida mientras que los repetidos varían notoriamente aunque es posible acotarlo.

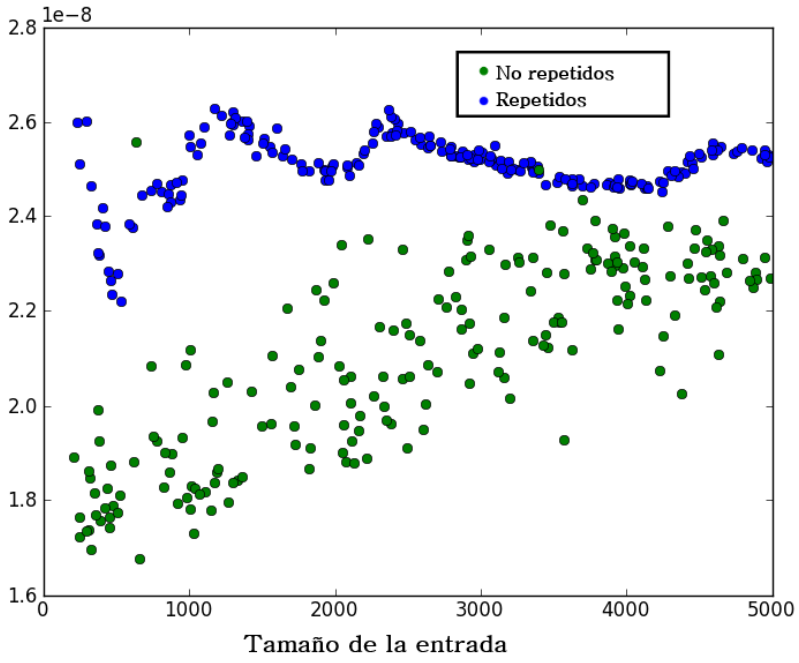


Figura 5: Gráfico de comparación empírica de la complejidad para repetidos vs no repetidos

2.6.2.3. Casos Random y comparación con peor y mejor caso En este caso las instancias fueron generadas de forma (pseudo)aleatoria en todos sus parámetros. Este conjunto de valores los usamos para comparar con el resto de las instacias. Veamos esta comparación:

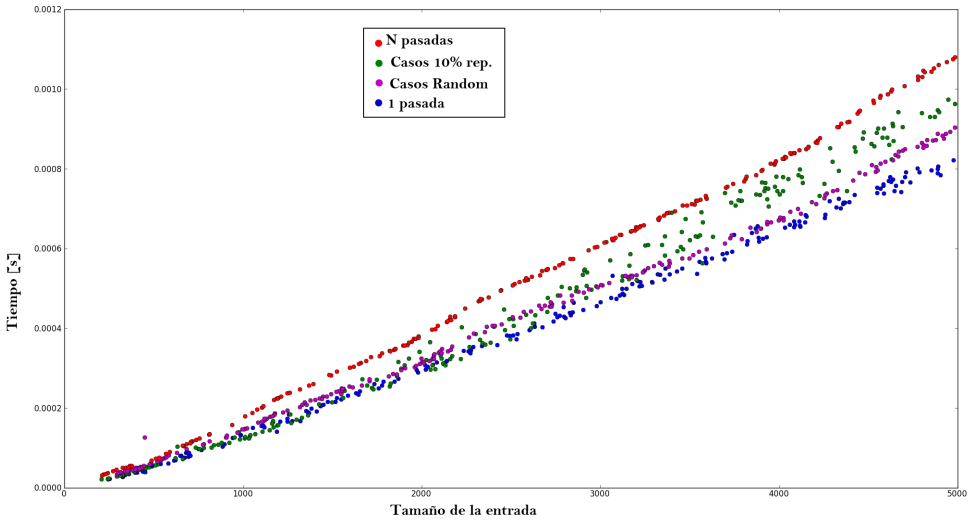


Figura 6: Gráfico de comparación entre random, mejor caso, peor caso y valores repetidos

Podemos ver entonces que las instancias aleatorias y con repeticiones se mantienen entre los límites de nuestro peor y mejor caso. Hay sólo un *outlier* para los casos randoms que sospechamos tiene que ver con el momento de la ejecución. Puede descartarse sin afectar nuestro análisis. Podemos determinar entonces en base a nuestra experimentación que lo que determina notoriamente el tiempo final por instancia (sin tener el cuenta el sort ya que todas las instancias están ïgualmente"desordenadas) es la cantidad de veces que haga la búsqueda nuestro algoritmo. Los mejores casos son aquellos donde en los primeros días ya se alcanza una solución adecuada mientras que si tiene que ir hasta los últimos días el tiempo de espera será más largo. La cantidad de camiones que vengan por día, es decir, los valores repetidos, no necesariamente mejoran la ejecución, aunque es posible ver casos en los que sea mejor que esto suceda.

2.7. Adicionales

En esta sección discutiremos los enuncados adicionales de este problema en contexto de la reentrega del presente informe.

2.7.1. ¿Cómo se modificaría el algoritmo si es posible partir los D días del inspector en 2 intervalos? ¿Cómo se modificaría la complejidad del mismo?

Dado un intervalo de D días, podemos construir $\frac{D}{2}$ combinaciones de intervalos distintos.

Para cada uno de ellas (contando la combinación D, vacío) habría que correr el algoritmo (quitando de los camiones a revisar por el segundo intervalo aquellos que registre el primer intervalo) y quedarse con la combinación que registre más camiones.

Es importante notar que para cada combinación, ejecutarlas en distinto orden altera el resultado obtenido, luego para cada una de ellas habría que ejecutar 2 veces el algoritmo (salvo para el caso extremo de intervalo D y vacío). Por ejemplo si tomamos la combinación (D-1, 1) y una entrada que tenga 10 camiones el primer día, d-1 días después 3 camiones un día y 2 al siguiente. Ejecutar primero el algoritmo para D-1 y luego para uno sacaría los 10 primeros camiones en la primera ejecución y cuando ejecute con 1 día tomaría los 3 camiones, pudiendo registrar 13 en total mientras que, ejecutando primero 1 día y después D-1, el algoritmo devuelve los 15 camiones como es esperable.

Por lo tanto habría que ejecutar D veces el algoritmo inicial modificando su complejidad hacíendola $O(Dnlog(n))$. Una modificación menor es que, al momento de devolver la salida, debe especificar los 2 días de inicio y cuánto tiempo implica cada uno.

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

3.1. Descripción del problema

El enunciado de este ejercicio nos plantea, en breves palabras. la necesidad de un joyero de armar un plan de trabajo. Dada una cantidad n de piezas, debemos indicar qué orden debe seguir este joyero para minimizar su pérdida. De cada pieza se sabe su número(i), los días necesarios para su realización (t_i) y, finalmente, cuánto dinero pierde por cada día que dedica al trabajo de esa pieza(d_i). Un dato importante es que el joyero, una vez que comienza con una pieza, no deja de trabajar con ella hasta terminarla ni tampoco puede trabajar con más de una a la vez. Se nos pide, a su vez, la pérdida total del joyero calculada como la suma sobre todas las piezas de su pérdida diaria tantas veces como días pasaron hasta su finalización. Todo el algoritmo debe implementarse en a lo sumo $O(n^2)$

La solución que proponemos consiste en ordenar la entrada de este problema (las n piezas) de menor a mayor según el cociente $\frac{DiasDeTrabajo_i}{PerdidaDiaria_i}$ de cada pieza p_i $\forall i$ $1 \leq i \leq n$, obteniendo así el costo mínimo.

Veamos entonces unos ejemplos

Tarda 2 días
Pierde 2 por día

Tarda 5 días
Pierde 10 por día

En este caso tenemos 2 joyas, la joya 1 para la que tarda 2 días y pierde 2 por dia que pasa y la joya 2 para la cual tarda 5 dias y pierde 10 por cada uno.

El input de esta situación entonces se ve asi:

```
2
2 2
10 5
```

Analizamos las posibles soluciones. Podemos o bien decirle al joyero que termine primero la joya 1 y luego la 2 o viceversa. Si hace primero la 1 perdera 2 por cada día que tarde, es decir 4 más 70, 10 por los 5 días de la joya 2 y los 2 que tardó en hacer la 1. En total perdería 74. Si lo hacemos al revés perdería 50 con la primera joya (5 días a razón de 10 de pérdida por día) más 14, 2 por cada uno de los 7 que pasaron hasta el final. En total, perdería 64. Nuestro output sería entonces:

```
2 1 64
```

Para reforzar la idea presentamos el siguiente ejemplo:

Tarda 2 días
Pierde 2 por día

Tarda 5 días
Pierde 10 por día

Tarda 100 días
Pierde 1 por día

En esta nueva instancia se agrega una pieza más. Su input es:

```
3
2 2
10 5
1 100
```

Ya conocemos el orden óptimo para las primeras dos, veamos dónde ubicar la tercera. Si la ubicamos primera, cada pieza subsiguiente pagará esos 100 días que tarda por su pérdida diaria agregando un costo enorme. Resulta entonces directo que el output será

```
2 1 3 171
```

en donde 171 es el costo final contra un posible 1374 de la combinación con la joya 3 al principio.

Habiendo analizado un par de instancias, pasemos a la siguiente sección en donde presentaremos detalladamente la solución propuesta.

3.2. La solución

Para resolver este problema optamos por un algoritmo goloso. Dicho tipo de algoritmo se caracteriza por tomar, en cada paso, la decisión óptima localmente, es decir, sin mirar el resto del problema. Para esta sección nos enfocaremos sólo en el ordenamiento de las piezas dado que el cálculo del costo es bastante directo.

A fin de dar una solución al enunciado propuesto, el algoritmo que implementamos ordena cada piezas ascendentemente según el cociente $\frac{DiasDeTrabajo_i}{PerdidaDiaria_i}$

Sobre la elección del criterio para ordenar, decidimos usar uno que mezcle ambas propiedades de las piezas ya que, tomando solo una es fácil construir ejemplos donde no se minimice la pérdida. A saber:

- Si tomamos en cuenta **solo la pérdida diaria** y ordenamos de mayor a menor (es decir, haciendo primero la pieza que pierde más) podemos construir un caso donde haya una joya que pierde 2 por dia pero tarda 1000 días y una que pierde 1 pero tarda 10 y, si calculamos el costo con este orden tendremos que perdería 3010 mientras que ordenando al revés saldria 2010, por lo que este criterio no garantiza buenos resultados.
- Un caso análogo puede hacerse si sólo se toma en cuenta **el tiempo de trabajo de la pieza**. Si ordenamos de menor a mayor según este criterio (para así evitar que los días de las piezas largas afecten a las más pequeñas) podemos armar un caso en donde una pieza tarde 1 pero pierda 1 por dia y otra tarde 5 pero pierda 100 por dia. Ordenando por días vemos que pierde 601 mientras que al revés sólo 506.

Decidimos, entonces, elegir el cociente de días de trabajo sobre pérdida diaria porque representa la idea intutiva que mientras más pierda por día menos día queremos que pasen y que mientras más tiempo tarde en hacerlo más tarde queremos que lo haga para así minimizar el costo de las piezas subsiguientes. La formalización de esta elección y su demostración, en la próxima sección.

3.3. Demostración de correctitud

Algorithm 4: Pseudocódigo para probar correctitud

```
1 print OrdenarPiezas(input, t_i/d_i)
2 print costo(input)
```

Para probar que este algoritmo es correcto basta con probar que el ordenamiento propuesto es óptimo para este caso (minimiza la pérdida del joyero).

Para ello probaremos que, dada una solución S (un ordenamiento de las piezas), si cambiamos de lugar 2 elementos desordenados según nuestro criterio, la nueva solución S$'$ es igual o mejor que la original. Supondremos entonces que sea i un índice válido en S ($1 \leq i \leq n$) S_i y S_{i+1} no están ordenados, es decir, $\frac{t_i}{d_i} > \frac{t_{i+1}}{d_{i+1}}$. Construimos entonces S$'$ como $\{S_{1..i-1}, S_{i+1}, S_i, S_{i+2...n}\}$ que son los mismos elementos de S con los elementos conflictivos cambiados de lugar.

Queremos ver que

$$Costo(S') - Costo(S) < 0 \tag{1}$$

En el contexto del problema, llamamos *Costo* al cálculo de la sumatoria sobre todas las piezas de la pérdida diaria de cada una tantas veces como días pasaron hasta el final de la misma. Recordemos que el problema pide la solución que tenga Costo mínimo. Para evitar notación innecesaria, llamemos a $Costo(S_{1...i})$ como C_1 y a $Costo(S_{i+2...n})$ como C_2. También llamemos T_{i-1} a la suma de los días de trabajo de cada pieza desde 1 hasta $i-1$. Luego, reemplazando en (1):

$$C_1 + d_{i+1}(T_{i-1} + t_{i+1}) + d_i(T_{i-1} + t_{i+1} + t_i) + C_2 - (C_1 + d_i(T_{i-1} + t_i) + d_{i+1}(T_{i-1} + t_i + t_{i+1}) + C_2) < 0 \quad (2)$$

Cancelamos términos repetidos.

$$-d_{i+1}(t_i) + d_i(t_{i+1}) < 0 \quad (3)$$

A partir de (3) deducimos: (3) $\iff d_i(t_{i+1}) < d_{i+1}(t_i) \iff \frac{t_{i+1}}{d_{i+1}} < \frac{t_i}{d_i}$

Llegamos a lo que habíamos asumido, es decir, que cualquier solución no ordenada puede ser mejorada intercambiando 2 elementos desordenados. Luego, es correcto afirmar que vale eq:costo para cualquier solución S.

Veamos que, en particular si tomo una solución óptima P, si asumimos que está desordenada, repitiendo el proceso anterior, llegaríamos al absurdo de poder mejorara una solución óptima. Esto se debe a asumir que la solución óptima no está ordenada.

Veamos también que si tomo una solución ordenada O y cambio 2 elementos O_i y O_{i+1} generando O$'$, si estos dos elementos tienen coeficientes distintos, O$'$ tendrá costo mayor que O pero si ambos son iguales, O$'$ estará ordenada y tendrá el mismo costo que O. Esto prueba que las soluciones ordenadas tienen todas el mismo costo.

Podemos decir entonces que como nuestro algoritmo ordena las piezas según el cociente planteado obtiene una solución óptima para este problema. Finalmente calcula el costo.

3.4. Demostración de complejidad

Algorithm 5: Pseudocódigo para demostrar complejidad

```
1 n ←input (leo la cantidad de piezas)
2 for i = 0 to n − 1 do
3 |   entrada[i].perdida←input
4 |   entrada[i].dias ←input
5 |   entrada[i].num ←i
6 end
7 sort(entrada, ordenarPiezasPorCociente) (ordeno las piezas por ti/di)
8 costo(entrada)
```

La primera asignación es $O(1)$.

Luego hay un ciclo que se ejecuta n veces de 3 asignaciones $O(1)$. En total O(n).

La función sort tiene una complejidad $O(nlog(n))$ que la garantiza su implementación (ver ej 1).

La función costo recorre la entrada y para pieza suma los días pasados hasta esa pieza y los que tarda la misma ($O(1)$ la suma y $O(1)$ para obtener el valor de los días que tarda) y multiplica este valor por la pérdida ($O(1)$ la multiplicación y lo mismo para conseguir la pérdida de la pieza). En total $O(n)$.

Luego la complejidad del algoritmo está dominada por la función sort, es decir, es $O(nlog(n))$

3.5. Código fuente

```
1
2  struct pieza{
3      int di;
4      int ti;
5      int num;
6  };
7
8  bool ordenarPares (pieza i, pieza j );
9  int costo(const vector< pieza> & vec, const int n);
10
11 int main(){
12
13   int n;
14   pieza temp;
15   bool fin=false;
```

```
  while(!fin) {
    cin >> n;
    if (n==0){
      fin=true; // sale del ciclo cuando encuentra el 0
    }
    else{
      vector < pieza > input (n);
      for(int i = 0; i < n; i++){
        cin >> input[i].di;
        cin >> input[i].ti;
        input[i].num=i+1;
        }

      sort(input.begin(), input.end(), ordenarPares);
      for(int i = 0; i < n; ++i){
        cout<<input[i].num <<" ";
      }
      cout<<costo(input, n)<<endl;

    }
  //
  }

  return 0;
}

bool ordenarPares (pieza i, pieza j ) {
    return ((j.di*i.ti)<(i.di*j.ti));
  }
int costo(const vector< pieza > & vec, const int n){
  int sum=0;
  int res=0;
  for(int i = 0; i < n; ++i){
    sum+=vec[i].ti;
    res+=sum*vec[i].di;
  }
  return res;
```

Código 2: Código fuente del problema 2

3.6. Tests

3.6.1. Test de correctitud

Para mostrar la correctitud empírica del algoritmo seleccionamos una variedad de casos que muestran su adecuado funcionamiento. A saber:

- Una sola joya.
- 2 joyas iguales
- 2 joyas ordenadas
- 2 joyas desordenadas
- 5 joyas no del todo desordenadas.

#	Input	Output del programa
0	1 2 3 2 2 3 2 3 2 1 1000 1000 1 2 1000 1 1 1000 5 999 1 2 3 2 3 1 1000 1000 1 0	1 6 1 2 18 1 2 2001 2 1 2001 5 1 2 3 4 4032

3.6.2. Test de complejidad

El test de complejidad de este algoritmo se reduce a ver la influencia de la entrada en el algoritmo de *sort* ya que el costo se calcula para todos en la misma cantidad de pasos (n). Para ello nos valdremos entonces de las siguientes familias de casos:

- Casos ordenados
- Casos desordenados
- Casos con la misma joya repetida
- Casos random.

3.6.2.1. Ordenados vs Desordenados Sobre la generación de estas instacias diremos que fueron al azar en cuanto a los números pero ordenadas (o desordenadas) posteriormente. Veamos los resultados:

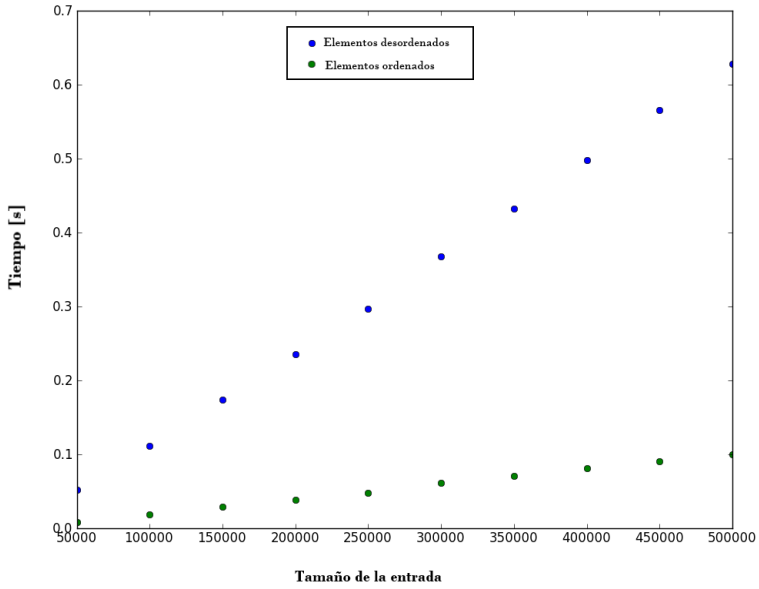


Figura 7: Gráfico comparativo de entrada en función de tiempo para ordenados vs no ordenados

Podemos ver en el gráfico que la función sort parece aprovechar el orden de la entrada resultando esto en un menor tiempo para la entrada ordenada.

Veamos que cumple con la complejidad propuesta diviendo los valores de tiempo por $nlog(n)$:

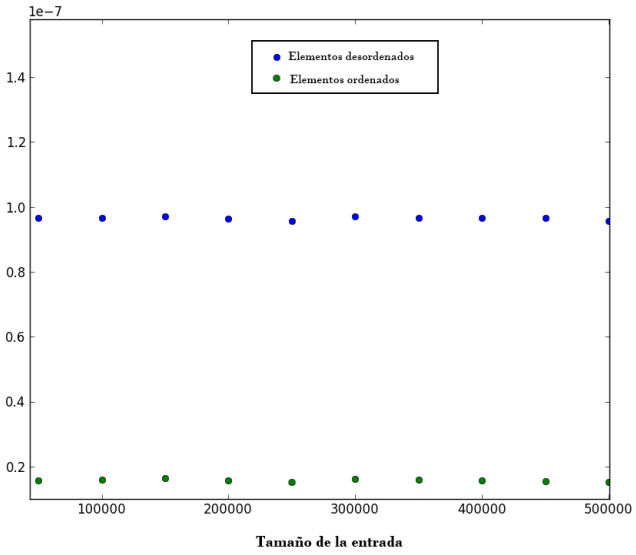


Figura 8: Gráfico de comparación empírica de la complejidad

Nuevamente obtenemos una constante por lo que podemos afirmar que nuestro algoritmo responde a la complejidad propuesta.

3.6.2.2. Casos Random y con la misma joya Hagamos una comparación entre una serie de instancias con la misma joya y una instancia completamente aleatoria.

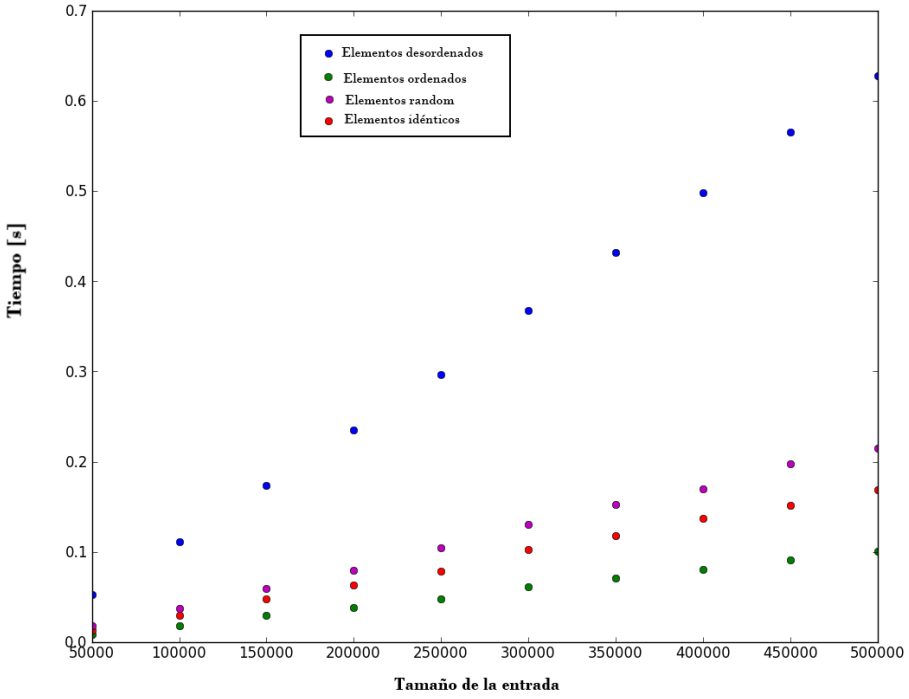


Figura 9: Gráfico comparativo de entrada en función de tiempo para ordenados vs no ordenados

Vemos en este caso que la instancia con las joyas idénticas, si bien está ordenada, tarda ligeramente más que para los casos ordenados estrictamente. Asumimos que esto se debe a que la función de ordenamiento devuelve *false* en caso de igualdad. Podemos observar también que estos 2 casos (random e iguales) están comprendidos entre los límites de mejor y peor caso de esta experimentación.

Si bien desconocemos la implementación interna de *sort*, podemos conjeturar que aprovecha el orden de las entradas y, luego, para este algoritmo, resulta conveniente pasar la entrada de forma ordenada siempre que sea posible. También pudimos ver que ordenamientos parciales también son aprovechados por la función sort. No mostramos esos datos en este informe pero están en los casos de test bajo el nombre "midOrd".

3.7. Adicionales

3.7.1. ¿Cómo modificar el algoritmo si a cada pieza se le agrega al final un tiempo de descanso? Dar una idea de la demostración.

Para modificar el algoritmo se debe ordenar según el cociente $\frac{t_i+r_i}{d_i}$. Siendo r_i el descanso asociado a la pieza iésima. Para la demostración, al momento de asumir que la solución S está desordenada se reemplaza el cociente por el nuevo cociente. Al desarrolar el costo vemos la siguiente expresión, en donde R_{i-1} es la suma de los descansos hasta la pieza i-1:

$C_1 + d_{i+1}(T_{i-1} + t_{i+1} + R_{i-1}) + d_i(T_{i-1} + t_{i+1} + t_i + R_{i-1} + r_{i+1}) + C_2 -$
$(C_1 + d_i(T_{i-1} + t_i + R_{i-1}) + d_{i+1}(T_{i-1} + t_i + t_{i+1} + R_{i-1} + r_i) + C_2) < 0$

y desarrolando nuevamente llegamos a :

$-d_{i+1}(t_i + r_i) + d_i(t_{i+1} + r_{i+1}) < 0$

que es de donde habíamos partido. Desde aquí en adelante la nueva demostración es análoga.

4. Problema 3: Rompecolores

4.1. Descripción del problema

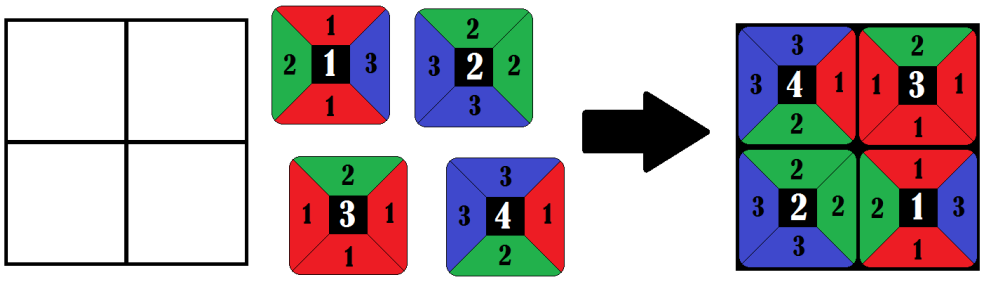
Se tiene un tablero rectangular (con casillas cuadradas) de $(n.m)$ casillas iguales. Se tienen también $(n.m)$ fichas, cada una del tamaño de una casilla y un número que la identifica. A su vez cada ficha está coloreada en sus cuatro bordes (superior, inferior, derecho e izquierdo), cada borde con un color (pueden repetirse). En total a lo sumo hay c colores. El problema consiste en encontrar una de las formas de colocar la mayor cantidad posible de fichas en el tablero, cumpliendo siempre que cuando dos fichas sean adyacentes, los colores de sus bordes en contacto coincidan. Además toda rotación de ficha (es decir que por ejemplo el borde que originalmente era el superior pase a ser el inferior o el de un costado) se encuentra terminantemente prohibida.

Para simplificar el problema, y dado que no interesa en realidad qué colores se utilicen sino las igualdades o desigualdades de colores entre bordes de piezas, estableceremos para la resolución del problema una erquivalencia entre los colores y números del 1 a c inclusive.

Por ejemplo, si $n = 2$, $m = 2$ y $c = 4$ y se tienen las siguientes 4 fichas coloreadas de la siguiente manera:

Pieza	Arriba	Izquierda	Derecha	Abajo
1	rojo(1)	verde(2)	azul(3)	rojo(1)
2	verde(2)	azul(3)	verde(2)	azul(3)
3	verde(2)	rojo(1)	rojo(1)	rojo(1)
4	azul(3)	azul(3)	rojo(1)	verde(2)

La representacion de las mismas y una solución óptima serían las siguientes:



Si por ejemplo la pieza 1 tuviera todos sus bordes color amarillo(4) (recordemos que $c = 4$), no habría forma correcta en cuanto a colores de colocar todas las piezas en el tablero y por lo tanto la solución óptima usaría solo tres piezas (por ejemplo con las piezas 4, 3 y 2, colocadas del mismo modo que en la solucion anterior, pero un espacio vacio donde antes estaba la pieza 1).

4.2. Formatos de Entrada-Salida

El formato de entrada (o input) del programa que resuelve el problema consta primero de una línea que contiene (en este orden) los valores de n, m y c; y luego una cantidad $(n.m)$ de líneas, representando cada una a una pieza (y en orden, es decir la primer línea a la pieza número 1, la segunda a la 2, etc) con los valores equivalentes a los colores de sus bordes superior, izquierdo, derecho e inferior (en ese orden). Por ejemplo, el input para el caso representado en la seccion anterior sería:

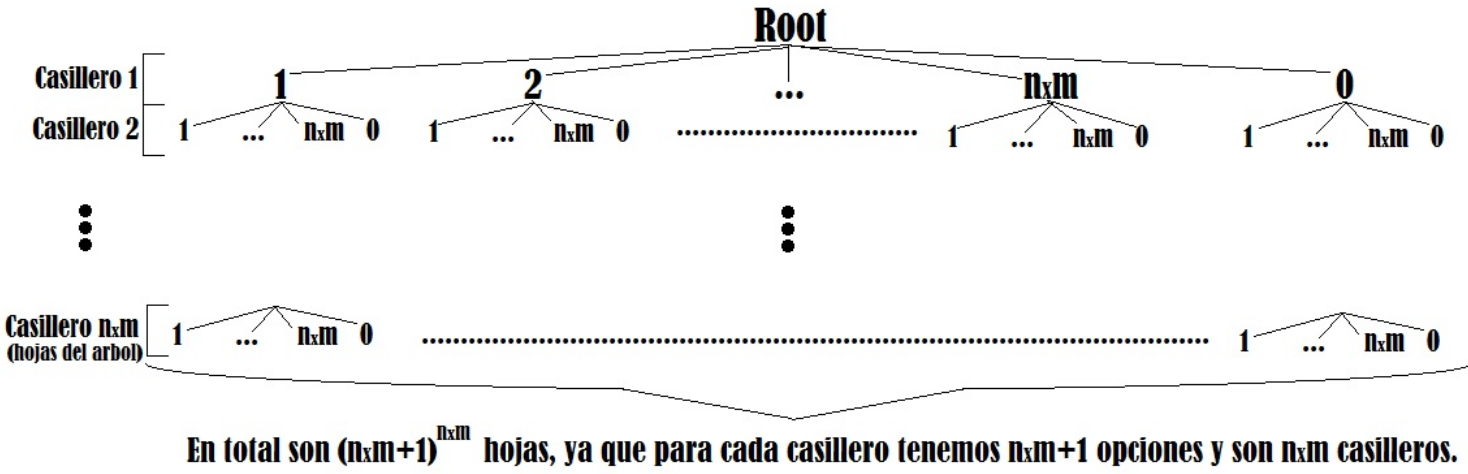
2 2 4
1 2 3 1
2 3 2 3
2 1 1 1
3 3 1 2

En cuanto al formato de salida (o output) del programa, consta de n líneas de m valores entre 0 y $(n.m)$ cada una. Cada valor es el número de la pieza colocada en el casillero correspondiente a ese valor (el valor j de la línea i representa el casillero de fila i y columna j del tablero). En caso de que el valor sea 0, representa que en ese casillero no se

colocó ninguna pieza. En nuestro ejemplo original, el output sería:
4 3
2 1

4.3. Resolución

Para la resolución de este problema planteamos primero un algoritmo que lo que hace es probar todas las formas de colocar las piezas en el tablero (considerando también la posibilidad de dejar casilleros vacíos y sin tener en cuenta que cada pieza se puede usar sólo una vez). Esas combinaciones se pueden resumir en el siguiente arbol:



Llamaremos *hermanos* a los nodos hijos del mismo padre y *piso del arbol* a la seccion del mismo que representa las posibilidades para el casillero correspondiente a ese piso, como indica la figura anterior.

La idea planteada en principio es recorrer las hojas de izquierda a derecha a traves del arbol mismo, es decir a traves de un camino (en particular, un camino simple). Esto nos permite considerar todas las posibles combinaciones manteniendo un control sobre la cantidad de espacios utilizados en cada combinacion, es decir en cada hoja (a medida que recorremos el arbol vamos actualizando una variable que los cuenta en función de si bajamos o subimos un piso cuyo valor era 0) y así poder finalmente devolver una combinacion tal que la cantidad de espacios utilizados sea mínima, es decir una combinación óptima.

Es claro que esto no se adapta a lo que estamos queriendo representar, ya que en nuestro problema hay algunas restricciones más a tener en cuenta. La adaptación del modelo se realizó mediante **podas**.

4.3.1. Podas

Al recorrer un árbol, efectuar una poda significa evitar recorrer algunos sub-árboles siguiendo cierto criterio.

4.3.1.1. Podas para correctitud

El algoritmo sin podas recorre todas las combinaciones de fichas y espacios vacios, permitiendo la utilizacion de cada ficha en más de un casillero. Esto en realidad no se adapta a nuestro problema ya que en el juego las fichas no pueden repetirse, y además las piezas adyacentes deben cumplir la condición de colores. Para que nuestro algoritmo considere solamente las soluciones que cumplen con estas condiciones (es decir, las soluciones correctas para el problema) efectuamos dos podas:

La primera implica mantener el control piso-a-piso de qué piezas ya fueron elegidas en pisos anteriores, de modo de sólo recorrer subárboles que mantengan esta la condición de haber utilizado como máximo una vez cada pieza (notese que el valor 0 no representa a una pieza sino al espacio vacio).

La segunda implica chequear, antes de empezar a recorrer un subárbol, si el hacerlo nos lleva indefectiblemente a un tablero incorrecto en cuanto a colores o no, para en funcion de eso entrar a ese subárbol o podarlo.

De ese modo logramos recorrer todas las hojas exceptuando las que no sirven como solución de nuestro problema, es

decir recorremos todas las que representan tableros correctos. Y como hacíamos antes, entre los tableros representados por cada hoja que recorremos solo guardamos el óptimo, es decir el que deja menos espacios vacios, que es justamente lo que tenemos que devolver como solución.

4.3.1.2. Podas para performance

En busca de la solución óptima entre las correctas, podemos efectuar algunas podas extra para mejorar la performance. Establecimos en total dos podas de este tipo:

La primera es la poda de mejor caso teórico. Esta poda consiste en dejar de recorrer el árbol en cuanto se encuentra una solución que no utiliza ni un espacio vacío (ya que esta solución es insuperable, y por lo tanto es al menos una de las óptimas). Es decir, si el camino desde Root hasta la hoja que estamos mirando en algún momento contiene a valores todos mayores a 0, entonces devolvemos el tablero representado por esa hoja y terminamos, es decir podamos todo subárbol que no hayamos recorrido todavía.

La segunda poda consiste en chequear, antes de entrar en un subárbol cuya raíz vale 0, si esto puede llevarme a una solución que supere a la mejor que habia encontrado hasta entonces o no. Como ya dijimos, una solución es mejor que otra si deja menos espacios vacíos en el tablero. Si antes de entrar a este subárbol con raiz=0 ya habia elegido el 0 en k pisos, el entrar al subárbol me llevaría a soluciones de por lo menos k+1 espacios vacíos. En caso de que ya tuviera algun caso anterior con sólo k espacios vacíos, no tiene sentido recorrer este subárbol y por lo tanto lo podamos.

4.4. Demostracion de Correctitud

Todo algoritmo que cumpla con lo descripto en la sección anterior (dejando en principio de lado las podas de performance) resuelve efectivamente el problema dado ya que, como fue explicado anteriormente, recorre absolutamente todas las posibles soluciones y guarda la mejor encontrada hasta el momento. Cuando termina de recorrer todas esas posibles soluciones, la mejor encontrada hasta el momento es la óptima, que es la que el problema requiere devolver. Dada la descripción de las podas de performance queda claro que no evitan recorrer ninguna solución que pueda superar a la mejor encontrada hasta el momento de la poda.

El modo en el cual se pueden llevar a cabo algunas secciones de la descripción de la solución puede no ser trivial. Es por esto que explicamos en esta sección un poco más detalladamente como se podrían implementar algunas cuestiones, para mostrar que no sólo el algoritmo terórico es correcto, sino que efectivamente hay por lo menos una implementación que se apega al mismo y es por lo tanto también correcta:

El tablero se puede representar como una matriz de nxm enteros, o lo que es computacionalmente equivalente, un arreglo de nxm enteros. El valor k de este arreglo representa al valor elegido en el piso k del árbol. Lo que hacemos entonces para recorrer el arbol es completar el arreglo en orden (ascendente en cuanto al indexado del arreglo) cuando bajamos pisos del arbol, y reiniciar posiciones del arreglo en el orden contrario cuando subimos pisos del árbol. Los valores con los que vamos completando no son arbitrarios, sino que la primera vez que se completa una posición se la completa con 1 y las sucesivas modificaciones que se efectuen sobre esta posición cumplen con el orden dado en el árbol para los hermanos. Cuando una posición se reinicia, vuelve al estado inicial (es decir la proxima vez que se la complete, vuelve a completarse con 1).

Esto nos permite mantener el control sobre la cantidad de espacios vacios utilizados, aumentando en una unidad la variable que los cuenta cada vez que escribimos un 0 en el arreglo, y decrementándola en una unidad cada vez que reiniciamos un elemento que valía 0 (notemos que si valía 0 no se lo puede modificar por otro valor sin reiniciarlo primero).

Respecto a la poda de no-repetición de piezas, se puede implementar manteniendo un arreglo actualizado de nxm elementos tal que cada uno representa a una pieza. Cada elemento tiene un valor que determina si esa pieza ya fue usada en alguna posición del tablero o todavía está libre. De este modo antes de completar una posición del arreglo-tablero con un valor, corroboramos en este arreglo de piezas si la pieza correspondiente está o no libre. Si estaba usada efectuamos la poda, es decir nos salteamos este valor para el elemento del arreglo-tablero y probamos con el hermano siguiente. Notemos que siempre podremos colocar un espacio vacio, de modo que siempre hay algun hermano siguiente libre, aunque sea el 0. Si estaba libre completamos el arreglo-tablero con ese valor y en el arreglo de piezas cambiamos esa pieza para marcar que pasa a estar usada. Por el contrario, antes de reiniciar un elemento del arreglo-tablero vamos al arreglo de piezas y marcamos la pieza correspondiente como libre.

Respecto a la poda de correctitud de colores, se puede tener una funcion que recorra todo el arreglo-tablero pero en forma matricial (el primer grupo de m elementos corresponde a la primer fila de la matriz-tablero, el segundo grupo

a la segunda fila, etc) y corrobore para cada par de elementos vecinos, que las piezas que representan esos elementos tengan la igualdad de color en los bordes necesarios. Chequeando el tablero entero con esta funcion cada vez que se completa un elemento del arreglo-tablero (y reemplazando ese valor por el hermano siguiente en caso de corroborar incorrectitud de colores) evitamos recorrer los subárboles incorrectos en cuanto a color. Notemos que si mantenemos el tablero correcto de este modo, sólo la ultima posicion completada del arreglo-tablero puede generar un inconveniente, y al modificarlo no es necesario reiniciar los subsiguientes, ya que no estaban todavía iniciados. Notemos también que el espacio vacío nunca genera conflicto de colores, de modo que siempre hay un hermano siguiente con el que se pueda completar un elemento del arreglo-tablero sin incurrir en conflicto de colores.

4.5. Complejidad

El análisis de complejidad del algoritmo fue efectuado sin tener en cuenta las podas de optimización. Sí fueron tenidas en cuenta las podas de correctitud ya que sin ellas el algoritmo no cumple con las condiciones exigidas por el problema.

Siendo $N = nxm$ la cantidad de fichas, de casilleros del tablero y entonces también de pisos del árbol, y K la cantidad de hojas del árbol que no repiten fichas (recordemos que sí pueden repetir ceros, ya que eso no implica repetir fichas) ni tienen conflicto de colores, la complejidad temporal del algoritmo es a lo sumo $O(N^3.K)$.

Esto se debe a que cada bajada de piso se hace en $O(N^2)$ en el peor caso porque habría que recorrer todas las fichas lo cual tiene un costo de $O(N)$ (ya que tenemos que encontrar una que este libre y encaje en cuanto a colores, o poner un 0 en caso de no haber encontrado ninguna) y **para cada ficha** hay que comprobar si está libre o usada ($O(1)$) y si encaja en cuanto a colores ($O(N)$) *(recordemos que nuestra funcion para chequear la correctitud de colores siempre chequea la correctitud del tablero entero, no de la pieza que estoy colocando en ese paso. Esto podria optimizarse a* $O(1)$ *justamente chequeando en cada paso solamente la correctitud entre la pieza que agrego y sus vecinos anteriores (ya que no hay vecinos posteriores), pero en esta implementacion no buscamos optimizar complejidad).* Luego de bajar, en caso de haber colocado una ficha hay que marcarla como usada ($O(1)$) o actualizar la variable de espacios vacíos en caso de no haber colocado ficha ($O(1)$). Las subidas se hacen en $O(1)$ ya que sólo hay que liberar la ficha correspondiente que es $O(1)$ o actualizar la variable que cuenta los espacios vacios que es también $O(1)$.

A su vez, el camino minimo entre dos hojas es, en el peor caso, subir hasta Root desde el piso de las hojas y volver a bajar. Esto implica N subidas y N bajadas, lo cual nos da una complejidad de $O(N^3)$. Para cada hoja hay además que comprobar si es mejor solución que la que veníamos considerando como la mejor ($O(1)$) y en caso de que lo sea, guardarnos ese tablero completo ($O(N)$).

Considerando en el peor caso que para cada hoja hay que hacer ese camino y guardar el tablero completo, por cada hoja que recorramos tendremos una complejidad de $O(N^3)$. En particular, sabemos que recorremos K hojas, de modo que la complejidad total del algoritmo es $O(N^3.K)$. *Notemos que con la optimización del chequeo de colores planteado anteriormente, esta complejidad se reduciría a* $O(N^2.K)$ *ya que cada bajada costaria* $O(N)$ *en vez de* $O(N^2)$.

Respecto a K, como ya dijimos representa la cantidad de soluciones correctas (no necesariamente óptimas) para el tablero. Para simplificar el análisis supondremos que todas las fichas están pintadas del mismo color, de modo que nunca tendremos conflicto de colores y así la cantidad de hojas correctas para un N dado son máximas, es decir el peor caso de colores para nuestro algoritmo sin podas de optimización. En este caso, va a haber hojas correctas con una sola pieza colocada (y el resto del tablero vacio), con dos piezas, con tres, hasta con N piezas colocadas. A su vez, suponiendo que se usan I de las N piezas, hay $\binom{N}{I}$ formas de elegirlas. Y para cada una de esas formas, hay que elegir en qué casilleros del tablero ponerlas (es decir elegir I entre los N casilleros, hay $\binom{N}{I}$ formas de elegirlos). A su vez, fijadas las piezas a utilizar y los casilleros donde colocarlas, hay que establecer en que orden se las colocara en esos casilleros (es decir si la primer pieza ira al primer casillero a llenar, o al segundo a llenar, etc). Hay en total $I!$ formas de ordenarlas.

Es decir, por cada I entre 0 y N hay $\binom{N}{I}^2 I!$ hojas. Si sumamos las hojas para todos los I entre 0 y N obtenemos $\sum_{i=0}^{N} \binom{N}{I}^2 I!$. En particular, con $I = N$ se da el mayor o igual de los términos de esa sumatoria (esto es facil de ver ya que por cada posible tablero (es decir hoja) para un I fijo tal que $I < N$, podemos generar al menos un tablero de $I = N$ reemplazando todos los 0's por piezas (en algún orden)). De modo que podemos acotar la sumatoria del siguiente modo: $\sum_{i=0}^{N} \binom{N}{I}^2 I! = 1 + \sum_{i=1}^{N} \binom{N}{I}^2 I! \leq 1 + \sum_{i=1}^{N} \binom{N}{N}^2 N! = 1 + N\binom{N}{N}^2 N! = 1 + N.N!$

En resumen, podemos afirmar que $K \leq 1 + N.N!$, por lo que podemos expresar la complejidad de nuestro algoritmo en función de N como $O(N^4.N!)$ (*Recordemos, reducible a* $O(N^3 N!)$ *con la optimizacion de la funcion que chequea colores*).

4.6. Codigo Fuente

```
#include <iostream>

#define usado 1
#define libre 0

using namespace std;

void sigPieza(int* tablero, int n, int m, int i, int j);
bool backTrack(int* tablero, int n, int m, int& i, int& j, int& espUtil);
bool check(int* tablero, int n, int m, int piezas[][5]);
void coutMatriz(int* matriz, int ancho, int alto);

int main(){
  bool podaNoMejor = false; //esta poda hace que no recorra subarboles que ya tienen mas espacios utilizados que el mejor caso
  bool podaOptimo = false; //esto hace que si encuentra un caso en el que ubica todas las fichas, corte porque ya es optimo.

  int n, m, c;
  cin >> n;
  cin >> m;
  cin >> c;

  int espUtil = 0;
  int mejorCaso = n*m;
  int tablero[n*m];
  int mejorTablero[n*m];
  for(int i=0; i<n; i++){
    for(int j=0; j<m; j++){
      tablero[i*m+j] = 0; //inicializo todo en 0. Tener en cuenta que esto no significa que elegi dejar un espacio vacio, sino que todavia no tome ninguna decision para esa posicion. En gral, solo tome decisiones para casilleros iguales o anteriores al ij actual de cada momento.
    }
  }

  int piezas[n*m+1][5]; //el "+1" es porque voy a usar el espacio vacio como una "pieza"

  for(int i=1; i<n*m+1; i++){
    for(int j=1; j<=4; j++){
      cin>>piezas[i][j];
    }
    piezas[i][0] = libre; //booleano que dice si esta usado o libre (0=libre, 1=usado);
  }
  piezas[0][0] = libre; //el espacio vacio va a estar siempre libre.


  for(int i=0; i<n; i++){
    for(int j=0; j<m; j++){

      sigPieza(tablero, n, m, i, j); //pongo la pieza siguiente a la que estaba (si no habia nada, pongo la 1era)


      while(!(check(tablero, n, m, piezas) && piezas[tablero[i*m+j]][0]==libre)){ //me fijo si la que intente poner estaba usada, o no encajaba en cuanto a colores, en cuyo caso digo "no, esa no! pongo la siguiente!"(?) hasta que llego a la ultima y en ese caso pongo el espacio vacio (un 0) que siempre encaja y esta libre.
        sigPieza(tablero, n, m, i, j);
      }

```

```
53
54      if(tablero[i*m+j]!=0){ //osea si entro bien una pieza, no un espacio vacio
55        piezas[tablero[i*m+j]][0]=usado; //marco la que acabo de usar.
56      }else{
57        espUtil++; //actualizo la variable
58
59        /*este if que viene ahora es la poda*/ //si se quiere habilitar/deshabilitar la poda
              , se usa el bool podaNoMejor
60        if(espUtil>=mejorCaso && podaNoMejor){  //en este caso ya tengo un mejor caso que el
               que puedo llegar a obtener en esta rama del arbol que tome recien al poner el
              espacio vacio en ij, asi que la podo osea backtrack ahora en vez de llegar hasta
               las hojas:
61
62          if(backTrack(tablero, n, m, i, j, espUtil)==1) { //paso i, j y espUtil por
                referencia para que tenga impacto aca afuera. Si la funcion devuelve 1 es que
                ya se recorrio todo, entonces muestro el mejor y termino. Si no, sigo
                recorriendo.
63            goto termine;
64          }
65
66          piezas[tablero[i*m+j]][0] = libre; //libero la que estoy modificando, porque
               justamente la estoy dejando de usar para modificarla por otra
67
68          if(j!=0 || i==0) j--; //ahora vuelvo una posicion atras porque asi cuando vuelvo a
                entrar al for me re-posiciona en la que estaba recien, osea la que me queria
               replantear.
69          else if(i!=0){
70              j=m-1;
71              i--;
72          }
73        } //termino la poda
74
75      }
76
77      if(i==n-1 && j==m-1){ //osea si termine todo el tablero
78        if(espUtil<=mejorCaso) { //ver si actualizo el mejor caso.
79          mejorCaso = espUtil;
80          for(int k=0; k<m*n; k++) mejorTablero[k] = tablero[k]; //me copio todo el tablero
81
82          /*segunda poda, si encontre el mejor caso teorico corto*/ //se habilita/
                deshabilita con podaOptimo
83          if (mejorCaso==0 && podaOptimo) { //si entro aca es porque encontre un caso ideal,
                 entraron todas las piezas, no sigo probando nada
84            goto termine;
85          };
86        }
87
88        if(backTrack(tablero, n, m, i, j, espUtil)==1) { //paso i, j y espUtil por
              referencia para que tenga impacto aca afuera. Si la funcion devuelve 1 es que ya
               se recorrio todo, entonces muestro el mejor y termino. Si no, sigo recorriendo.
89          goto termine;
90        }
91
92        piezas[tablero[i*m+j]][0] = libre; //libero la que estoy modificando, porque
               justamente la estoy dejando de usar para modificarla por otra
93
94        if(j!=0 || i==0) j--;
95        else if(i!=0){
96          j=m-1;
97          i--;
98        }//tengo que volver a hacer esto porque al salir, el "for" me va a incrementar una
               posicion de nuevo, entonces justo antes de eso resto una para que al
               incrementarla caiga de nuevo en la que acabo de chequear que me quiero
```

```
            replantear.
99
100       }
101
102     }
103
104   }
105   termine:
106   coutMatriz(mejorTablero, m, n); //muestro el resultado
107   return 0;
108
109 }
110
111 void sigPieza(int* tablero, int n, int m, int i, int j){
112   if (tablero[i*m+j]<n*m) tablero[i*m+j]++;
113   else{
114     tablero[i*m+j]=0;
115   }
116 }
117
118 bool backTrack(int* tablero, int n, int m, int& i, int& j, int& espUtil){ //cuando salgo de
    aca, i y j corresponden a la posicion que me quiero replantear (osea, incluyendo la
    posicion (i,j) que me pasaron y yendo para atras, la primera que no es un espacio vacio)
119   while(tablero[i*m+j]==0){ //me fijo si donde estoy mirando habia un cero, porque en ese
      caso ya no puedo probar nuevas opciones para esta posicion excepto que cambie antes
      algo de una posicion anterior. La primera vez va a entrar seguro, porque justamente
      acabo de poner un 0 y eso es lo que me determino que no voy a poder ya mejorar el
      mejor caso, pero el while hace que vaya tanto para atras como haga falta.
120     if(j!=0) j--;
121     else if(i!=0){
122       j=m-1;
123       i--;
124     }else{ //si siendo el primer casillero==0 ya no puedo encontrar una solucion mejor que
        la que tenia antes, significa que la que tenia antes no usaba ningun espacio vacio y
         por lo tanto es el tablero optimo.
125       return 1; //devuelvo 1 si el backtrack me dio como resultado que ya no hay mas
          posibles tableros que recorrer, osea que tengo que imprimir el mejor que obtuve
          hasta entonces
126     }
127     espUtil--; //este espacio ya no lo cuento como utilizado porque me lo voy a replantear
        asi que disminuyo la variable.
128   } //cuando salgo de aca, i y j corresponden a la posicion que me quiero replantear (osea,
      incluyendo la posicion (i,j) que me pasaron y yendo para atras, la primera que no es
      un espacio vacio)
129   return 0;
130 }
131
132 void coutMatriz(int* matriz, int ancho, int alto){
133     for(int i=0; i<alto; i++){
134         for(int j=0; j<ancho; j++){
135             cout << matriz[i*ancho+j];
136             if(j<ancho-1) cout <<" ";
137             else cout << endl;
138         }
139     }
140 }
141
142 bool check(int* tablero, int n, int m, int piezas[][5]){ //para chequear primero recorro las
     filas, desde el segundo elemento hasta el ultimo inclusive, mirando si entra en
    conflicto con el anterior de su fila. Despues recorro todo de nuevo pero por columnas
    desde el segundo hasta el ultimo de cada columna inclusive, mirandi si entra en
    conflicto con el anterior de su columna. Si recorri todo y no entontre conflicto,
    devuelvo true; si encuentro algun conflicto devuelvo false en el momento.
```

```
143
144   for(int i=0; i<n; i++){ //recorro todas las filas
145       for(int j=1; j<m; j++){ //desde el segundo elemento hasta el ultimo de cada fila
146
147         if(tablero[i*m+j]!=0 && tablero[i*m+j-1]!=0){ //si de los 2 que voy a comparar
              alguno es 0, osea espacio vacio, no hay nada que comparar entonces no entro.
148
149           if(piezas[tablero[i*m+j]][2]!=piezas[tablero[i*m+j-1]][3]) return false; //el que
                estoy mirando entra en conflicto con el de su izquierda
150
151         }
152
153       }
154   }
155
156
157   for(int j=0; j<m; j++){ //recorro todas las columnas
158       for(int i=1; i<n; i++){ //desde el segundo elemento hasta el ultimo de cada columna
159
160         if(tablero[i*m+j]!=0 && tablero[(i-1)*m+j]!=0){ //si de los 2 que voy a comparar
              alguno es 0, osea espacio vacio, no hay nada que comparar entonces no entro.
161
162           if(piezas[tablero[i*m+j]][1]!=piezas[tablero[(i-1)*m+j]][4]) return false; //el
                que estoy mirando entra en conflicto con el de arriba
163
164         }
165
166       }
167   }
168
169   return true;
170 }
```

Código 3: Descriptive Caption Text

4.7. Casos Borde

Para el análisis de casos borde utilizaremos el algoritmo sin podas de performance. Vamos a tener en cuenta dos aspectos: el tamaño del tablero y la distribución de los colores en las fichas. El número c (recordemos, la cantidad total de colores diferentes con que pueden estar pintadas las piezas) no interesa ya que el algoritmo no lo utiliza en absoluto (como preconcidición suponemos que vale que el conjunto de colores utilizados tiene un cardinal igual o menor a c).

Respecto al tamaño del tablero, veremos dos casos:

El primero es cuando $n = 0 \vee m = 0$. En este caso $n.m = 0$ entonces el árbol planteado inicialmente tiene sólo a Root, de modo que no hay hojas (excluyendo a Root, claro, ya que su piso no representa ningún casillero). Claro que desde el punto de vista del juego Rompecolores este tablero no tiene sentido, pero vemos que en particular nuestro algoritmo funciona también para este caso, devolviendo un tablero sin casilleros, y por lo tanto obviamente sin piezas. Esto es, la única solución (y por lo tanto la óptima) para este problema trivial.

input1:	**input2:**	**input3:**	**output(único):**
0 0 1	0 2 1	2 0 1	(vacío)

El segundo es cuando $n = 1$ y $m = 1$. En este caso el árbol tiene sólo dos hojas. De nuevo, desde el punto de vista del juego esto no presenta ningun interés pero al probar el algoritmo para este caso vemos que devuelve un tablero de un sólo casillero con la única ficha disponible. Esto es, nuevamente, la solución óptima para este problema trivial (aunque menos trivial que el caso $n = 0 \vee m = 0$).

input:	**output:**
1 1 5 2 4 5 1	1

Para el resto de valores de n y m el modelo expresado anteriormente se adapta sin inconvenientes. Cabe resaltar que, como se expresa en el modelo del árbol, a nuestro algoritmo no le interesa demasiado cuánto vale n y cuánto vale m sino mas bien cuánto vale el producto entre ambos. Es necesario igual tener ambos valores por separado para poder establecer la vecindad entre casilleros, pero sólo para esto (que a su vez es necesario para la corroboración de la correctitud de colores).

Respecto a la distribución de colores, analizaremos dos casos:

El primero es cuando todas las fichas estan pintadas con el mismo color. En este caso no puede existir conflicto de colores entre dos fichas adyacentes. Este caso es el más beneficioso para quien quiera colocar el mayor número de fichas, ya que se podrian colocar todas (en cualquier orden) sin problema alguno. Vemos que efectivamente nuestro algoritmo lo hace:

input:	**output:**
3 3 5	9 8 7
1 1 1 1	6 5 4
1 1 1 1	3 2 1
1 1 1 1	
1 1 1 1	
1 1 1 1	
1 1 1 1	
1 1 1 1	
1 1 1 1	
1 1 1 1	

El segundo caso es cuando todas las fichas estan pintadas completamente con colores diferentes. Es decir, cada ficha con el mismo color en sus cuatro bordes, pero todas las fichas con colores diferentes de modo que no pueda haber dos fichas adyacentes sin generar conflicto de color. Este es el caso menos conveniente en cuanto a colocar muchas fichas en el tablero, y como tal establece una cota inferior (no estricta) para la solución óptima de cualquier distribución de colores en un tablero de dimensiones fijas. Vemos que nuestro algoritmo devuelve una solución óptima, colocando la mayor cantidad de fichas posibles sin colocar dos adyacentes (analizaremos dos opciones: n y m pares, y n y m impares).

n y m pares:

input:	**output:**
4 4 16	0 16 0 15
1 1 1 1	14 0 13 0
2 2 2 2	0 12 0 11
3 3 3 3	10 0 9 0
4 4 4 4	
5 5 5 5	
6 6 6 6	
7 7 7 7	
8 8 8 8	
9 9 9 9	
10 10 10 10	
11 11 11 11	
12 12 12 12	
13 13 13 13	
14 14 14 14	
15 15 15 15	
16 16 16 16	

n y m impares:

input:	output:
3 3 9 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 6 6 6 6 7 7 7 7 8 8 8 8 9 9 9 9	9 0 8 0 7 0 6 0 5

4.8. Experimentación

Para la experimentación medimos tiempos de ejecución de nuestro algoritmo (implementado según los lineamientos esbozados en la sección de correctitud) en función del tamaño del tablero. Como vimos anteriormente, en cuanto a cantidad de operaciones (que tiene una relación directa con el tiempo de ejecución) lo que importa es el producto entre n y m y no los valores particulares de esas variables. Es por esto que mantendremos m constante para cada caso, y calcularemos los tiempos en función de n, alterando asi el producto sólo con modificaciones en una de las variables de entrada. Otra cuestión a resaltar es que, para cada caso (y en particular para cada n) se midieron tiempos para una cantidad k de instancias y esos tiempos luego fueron promediados para contemplar variaciones en los tiempos adjudicables a factores externos (uso del cpu de otras tareas, por ejemplo). Medimos esos tiempos para diferentes casos:

Caso 1: $m = 1, c = (n.m), k = 50, n$ *variable*. Piezas coloreadas cada una en sus 4 bordes con el color correspondiente a su numeración. Es decir, conflicto de colores entre cualquier par de vecinos. **Mejor caso** en cuanto a complejidad, ya que el numero de hojas correctas es mínimo respecto al tablero.

n	**tiempo(s) Sin podas**	**tiempo(s) Con podas**
1	1.8e-07	6e-08
2	8.8e-07	6.2e-07
3	3.34e-06	1.28e-06
4	1.596e-05	8.88e-06
5	7.074e-05	1.794e-05
6	0.00021244	0.0001661
7	0.00075498	0.00016516
8	0.00404342	0.00162756
9	0.0211439	0.00298698

Para ver si esta experimentacion se adapta o no a la complejidad propuesta, decidimos dividir los tiempos de ejecución sin podas por algunas funciones de N (recordemos, $N = (n.m)$ pero en particular como $m = 1$ en este caso $N = n$); entre ellas la que propusimos como cota de complejidad ($O(N^4.N!)$). El siguiente gráfico muestra el cociente entre los tiempos y las distintas funciones propuestas. Si una función efectivamente es cota de complejidad temporal del algoritmo, asintóticamente el cociente entre el tiempo de ejecución y esa función debería ser constante.

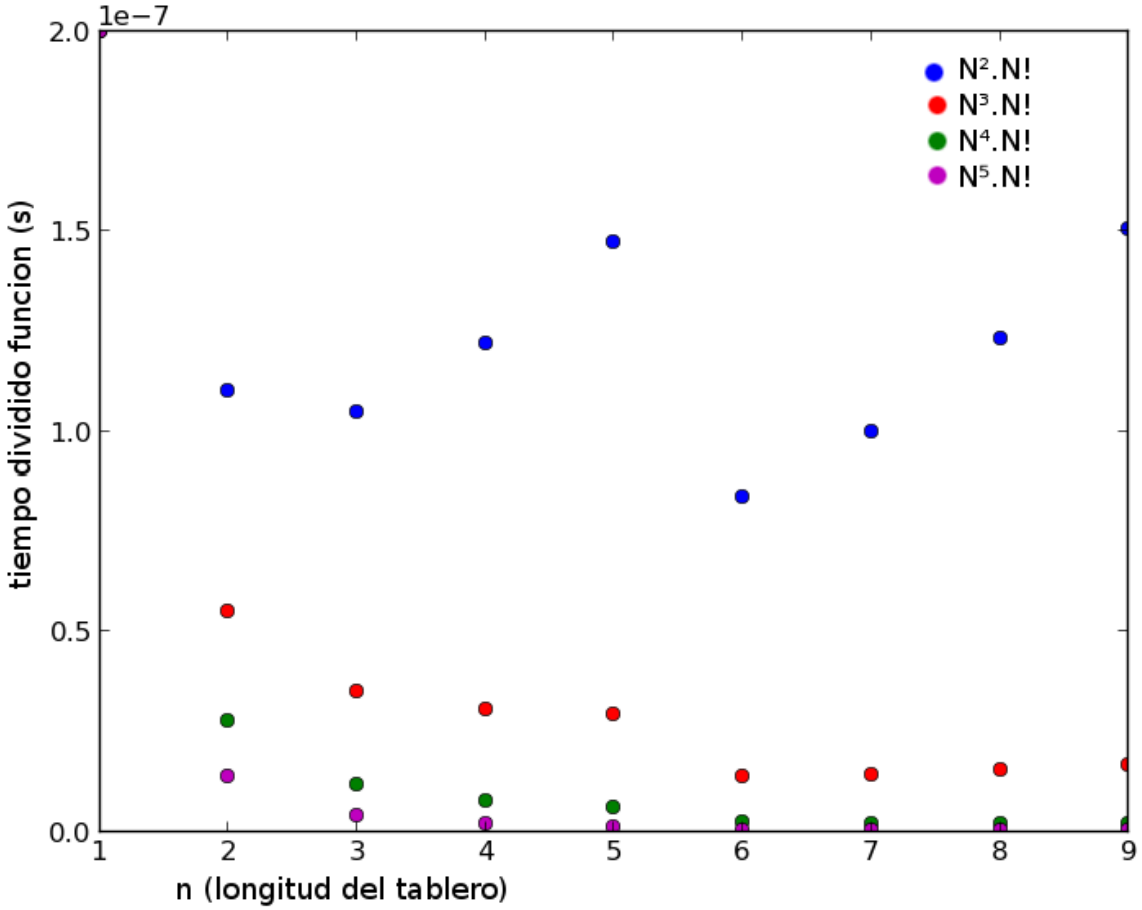


Figura 10: Grafico de analisis de complejidad para **caso 1**

En el gráfico no se llega a apreciar ninguna tendencia constante clara para ninguna función. Podemos apreciar, si, que por ejemplo para la función $N^2.N!$ el comportamiento del gráfico pareciera indicar que no va a tender a una constante. Para el resto de las funciones podría pensarse, segun el gráfico, que eventualmente pueden dejar de variar su valor pero dados los tiempos necesarios para la experimentacion (con $n > 9$ los tiempos crecían abruptamente) no nos fue posible comprobarlo.

Por otra parte analizamos también la diferencia de tiempos que genera la utilización de las podas de performance propuestas. Para apreciar mejor esta diferencia, la graficamos con una escala logarítmica ya que con escala lineal el crecimiento abrupto de los valores dificultaba su apreciación:

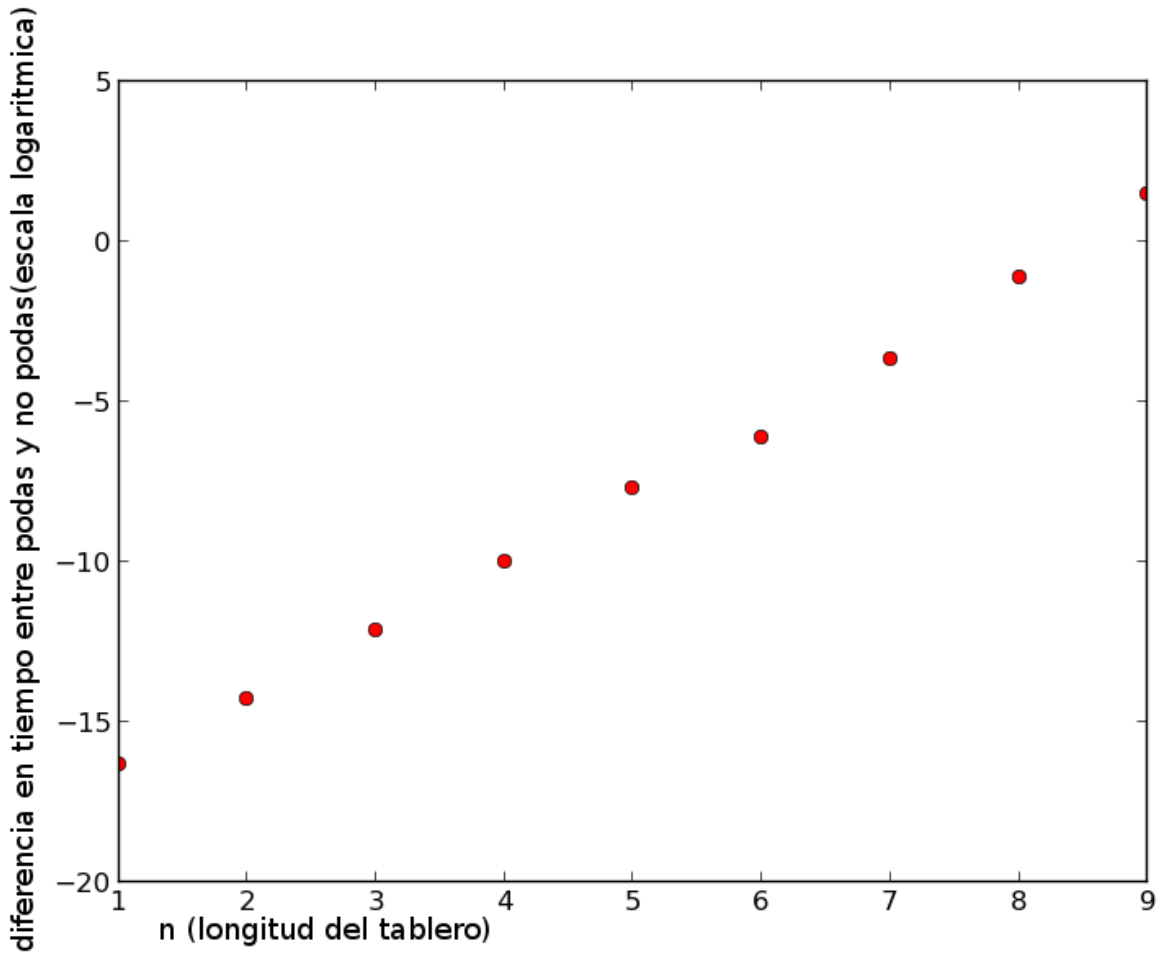


Figura 11: Grafico comparativo de tiempos de ejecucion para **caso 1**

Vemos en el gráfico una tendencia creciente en la diferencia de tiempos, lo que era de esperar dado que a mayor tamaño del tablero mayor cantidad de hojas a recorrer y por lo tanto mayor cantidad (y tamaño) de subárboles que se pueden llegar a podar.

Caso 2: $m = 1, c = 4, k = 50, n$ *variable*. Piezas coloreadas de manera aleatoria con los 4 colores. Elegimos $c = 4$ para permitir que una misma pieza tenga un color diferente en cada borde. Si c fuera más grande, el coloreo de manera aleatoria se acercaría más al caso anterior.

n	tiempo(s) Sin podas	tiempo(s) Con podas
1	2e-07	4e-08
2	9.2e-07	4.6e-07
3	4.26e-06	1.62e-06
4	2.252e-05	4.42e-06
5	0.00012814	1.098e-05
6	0.0003973	3.372e-05
7	0.00219192	0.00010034
8	0.01422	0.00020124
9	0.104553	0.00054804

Repetimos el gráfico para analisis de complejidad en este caso, que consideramos el caso promedio al haber coloreado de manera aleatoria las piezas.

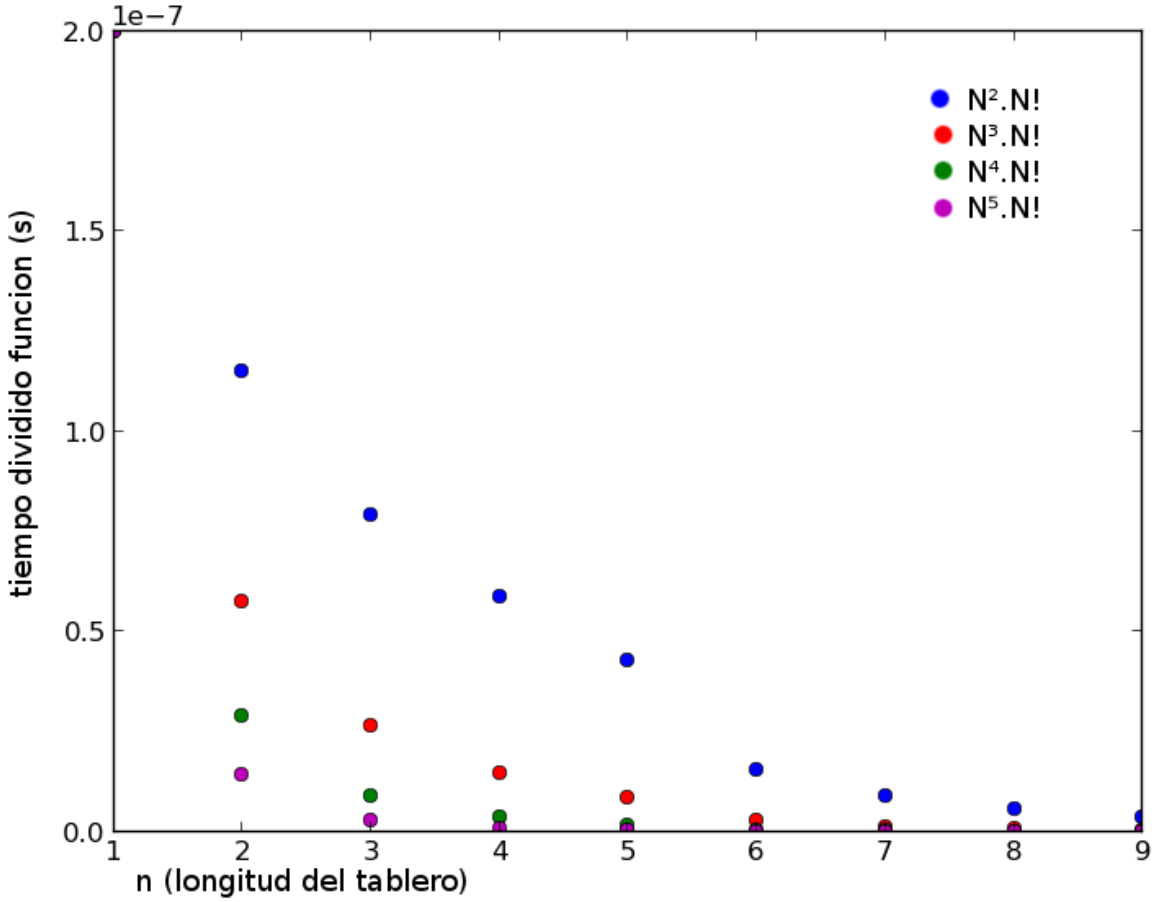


Figura 12: Grafico de analisis de complejidad para **caso 2**

En este caso a partir del grafico para analisis de complejidad no podriamos descartar la funcion $N^2.N!$ ya que nada parece indicar que asintoticamente no vaya a tender a una constante. Respecto al resto de las funciones, el análisis es análogo al grafico de complejidades anterior.

Hicimos tambien para este caso el grafico comparativo de los tiempos de ejecucion del algoritmo con y sin podas. Notamos nuevamente una tendencia creciente adjudicable al mismo factor.

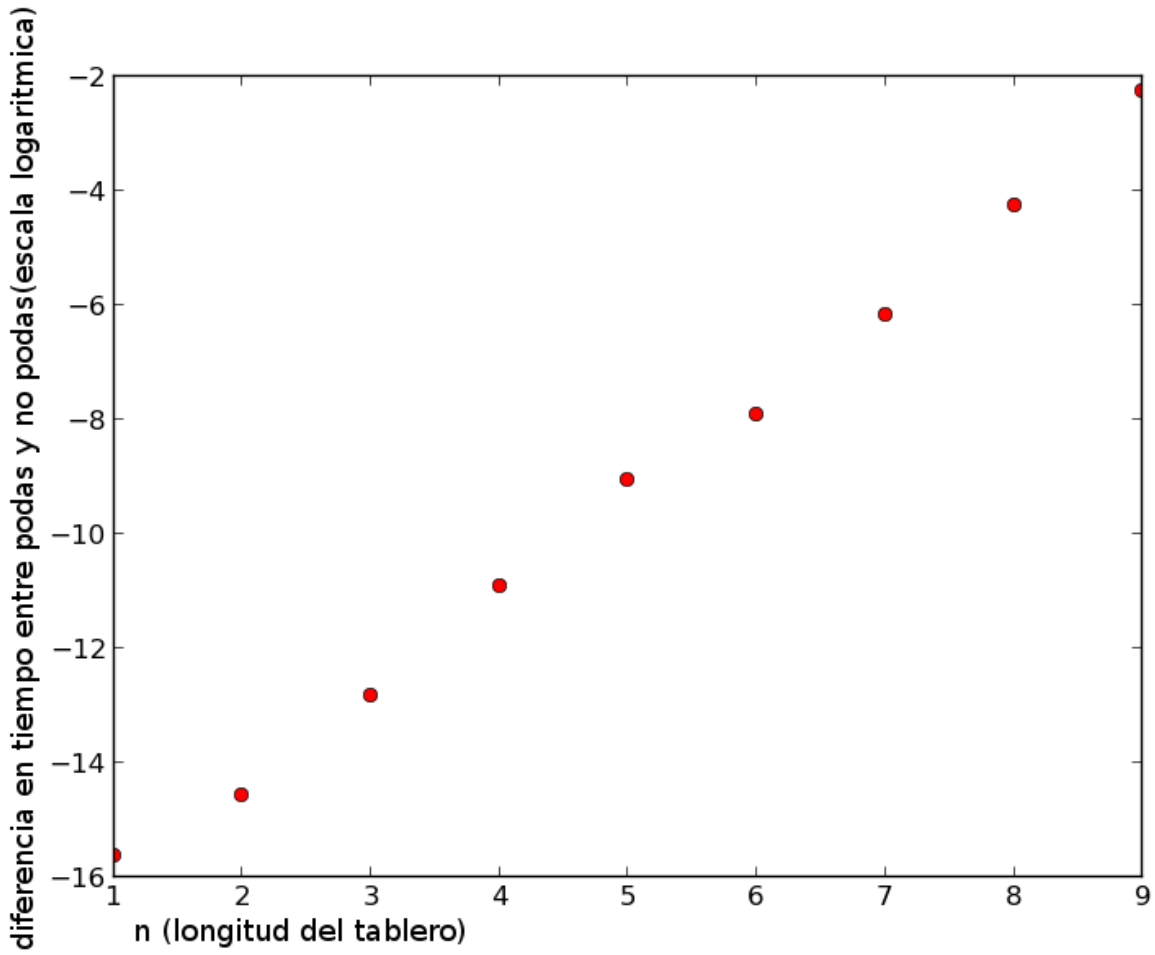


Figura 13: Grafico comparativo de tiempos de ejecucion para **caso 2**

Caso 3: $m = 1, c = 1, k = 50, n$ *variable*. Pîezas coloreadas todas con el mismo color (el único que se puede utilizar ya que $c = 1$). No puede haber conflicto de colores entre ningún par de vecinos. **Peor caso** en cuanto a complejidad, ya que el numero de hojas correctas es máximo respecto al tablero.

n	tiempo(s) Sin podas	tiempo(s) Con podas
1	2e-07	1.2e-07
2	8.8e-07	2.6e-07
3	5.66e-06	4.4e-07
4	4.686e-05	1e-06
5	0.0004412	1.58e-06
6	0.00216316	2.68e-06
7	0.0246436	4.08e-06
8	0.317332	5.92e-06
9	4.42317	7.46e-06

En este caso en el grafico de analisis de complejidad volvemos a notar que la funcion $N^2.N!$ parece dar señales de no tender nunca a una constante. Nuevamente para las otras funciones no encontramos prueba suficiente en este gráfico para afirmar, con un grado de certeza razonable, que ninguna tiende a una constante ni que no lo hacen. Otra vez, para tener ese grado de certeza tendríamos que experimentar con muchos más valores de n y esto no nos es posible por el tiempo que llevaría hacerlo. Igualmente el análisis teórico se mantiene y por eso nuestra hipótesis sigue siendo que la complejidad temporal es $O(N^4.N!)$ ya que ninguna experimentación pudo contradecirla.

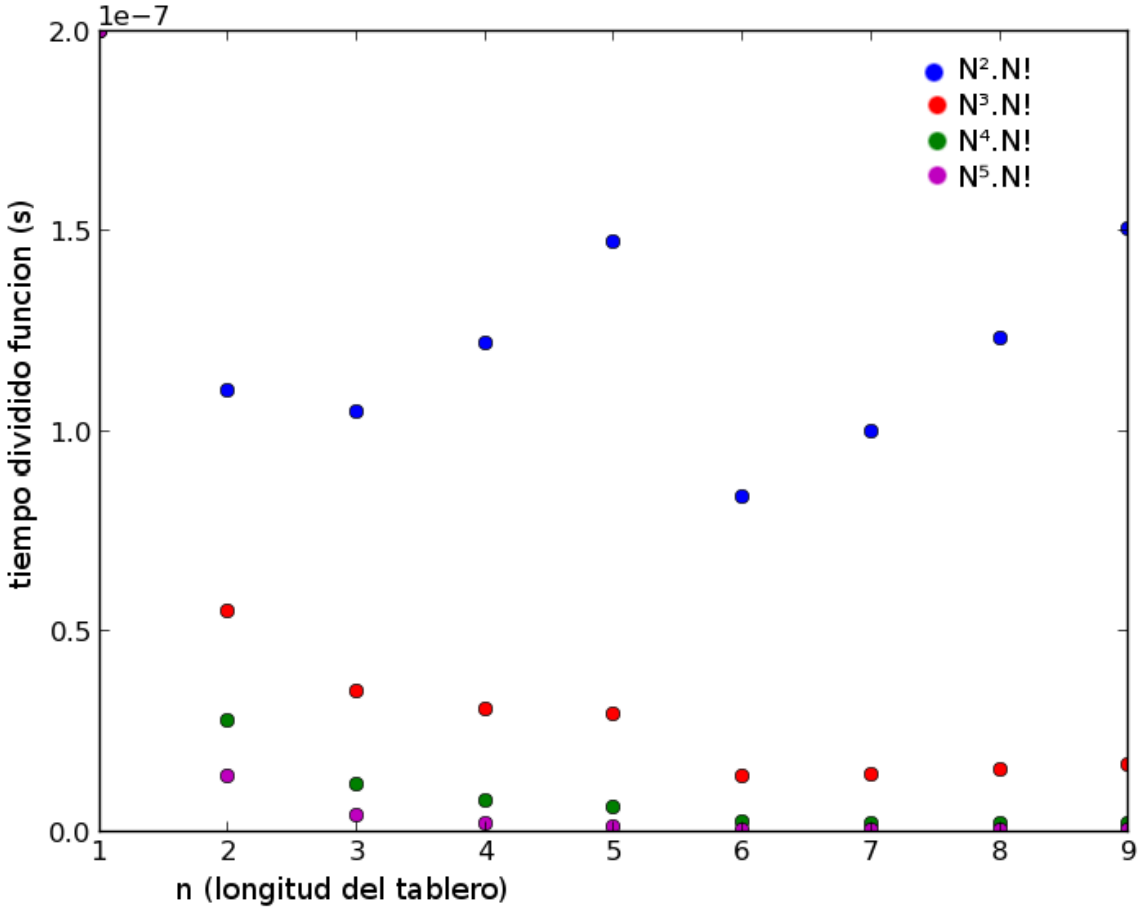


Figura 14: Grafico de analisis de complejidad para **caso 3**

Por último, respecto a la mejora de performance que aportan las podas al algoritmo, nuevamente vemos una mejora considerable (tener en cuenta que la diferencia de tiempos se graficó en escala logarítmica, de modo que una linealidad en el gráfico se corresponde con un crecimiento exponencial de la diferencia). En términos absolutos esta diferencia es apreciable en la tabla de tiempos presentada al comienzo de cada caso.

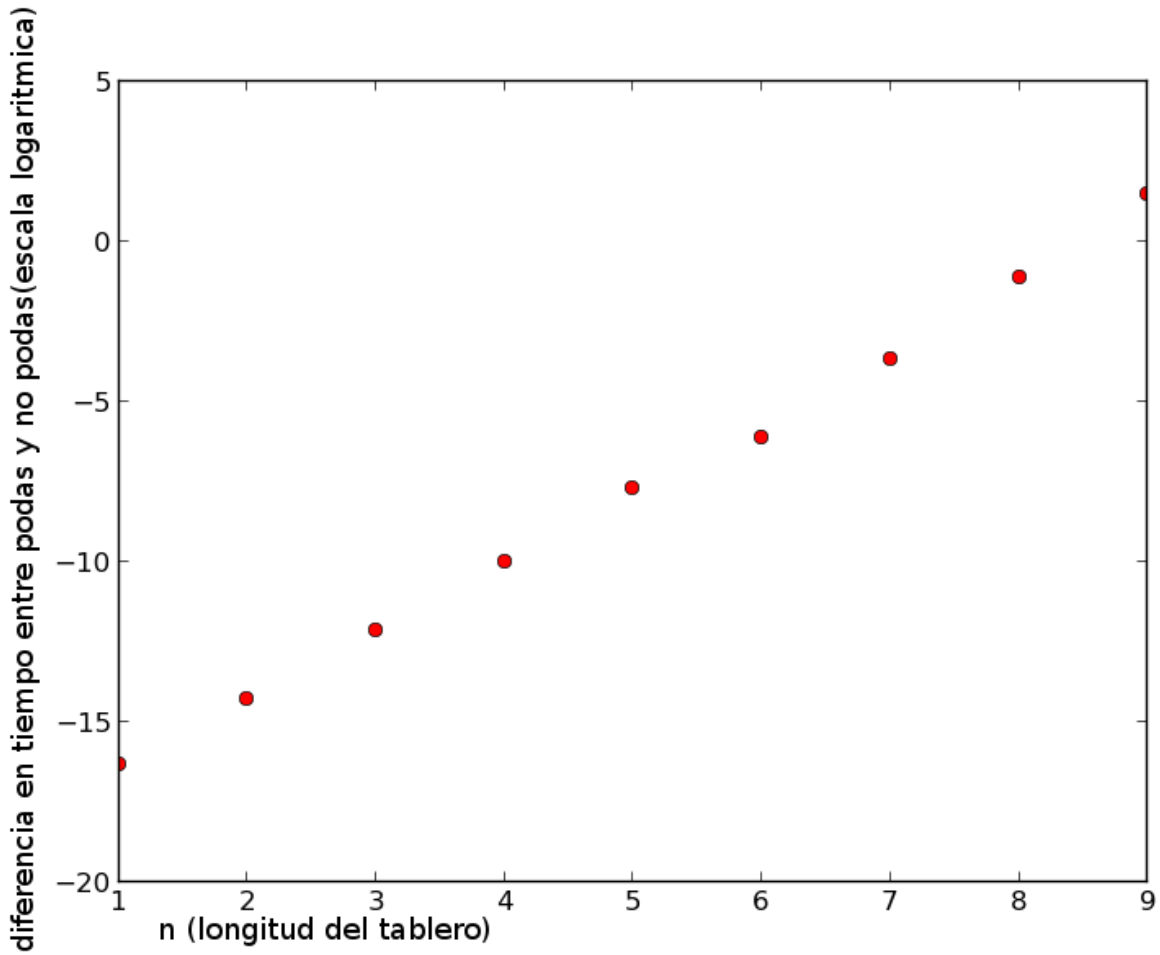


Figura 15: Grafico comparativo de tiempos de ejecucion para **caso 3**

4.9. Conclusiones

Como conclusion podemos afirmar que el algoritmo efectivamente resuelve el problema pero de una manera muy poco eficiente. Tal es así que para tableros de más de 9 casilleros el tiempo que el algoritmo requería (en un caso promedio) para resolver el problema fue tal que nos significó un impedimento para el análisis empirico de complejidad. Este tipo de comportamiento es propio de la técnica de **backtracking** Por otra parte, la implementación de podas produjo mejoras considerables en el rendimiento, lo cual combinado con la mala eficiencia del algoritmo en sí nos indica que, de usar esta técnica para resolución de problemas reales, resulta clave la utilización lo más exhaustiva posible de las podas.

4.10. Adicionales

El caso en el que el tablero es de forma toroidal se puede modelar con el caso en el que el tablero es rectangular, que es el que resolvimos en primer lugar. Este modelado consistiria en considerar las piezas colocadas en la primer fila del tablero vecinas de las colocadas en la última fila (el borde superior de la primer fila con el borde inferior de la última) y las piezas de la primer columna, vecinas de las de la última columna (borde izquierdo de la primer columna, con borde derecho de la última). De esta manera el algoritmo de resolución se mantiene tal cual, excepto por la comprobación de correctitud de colores que tendría que tener en cuenta las nuevas vecindades previamente mencionadas.