DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

16 de abril de 2017

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2017

Autor	LU	Correo electrónico
Szperling, Sebastián Ariel	763/15	sszperling@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Descripción del problema

1.1. Descripción general

El problema propuesto pide que se encuentre la menor cantidad ."elementos sin pintar"posibles para una lista de números enteros siguiendo el siguiente criterio:

- Todos los elementos pueden solo estar pintados de azul, de rojo, o estar sin pintar (mutuamente exclusivos).
- Todos los elementos pintados de rojo deben estar en orden *estrictamente creciente*.
- Todos los elementos pintados de azul deben estar en orden *estrictamente decreciente*.

El problema, para una lista dada de números, se considera que tiene una resolución **óptima** si existe una manera de pintar todos los elementos de dicha lista ya sea de rojo o de azul, es decir, no queda ningún elemento sin pintar.

1.2. Ejemplos provistos por la cátedra

Para estos ejemplos, se denotará debajo de cada número con una letra si el mismo se pinta de **R**ojo, **A**zul o **N**o pintado.

1. Ejemplo con solución óptima

0	7	1	2	2	1	5	0
R	A	R	R	A	A	R	A

   Resultado: 0 (óptima)

   En este ejemplo se pueden encontrar subsecuencias que cumplan los criterios mencionados: una de elementos rojos (0 1 2 5) y otra de elementos azules (7 2 1 0).

2. Ejemplo sin solución óptima

3	11	0	1	3	5	2	4	1	0	9	3
N	A	R	R	R	A	A	R	A	A	R	N

   Resultado: 2

   En este ejemplo no existe un par de subsecuencias disjuntas que satisfagan todos los criterios, por lo que el primer y el último 3 son ignorados para maximizar la cantidad de elementos pintados.

1.3. Otros ejemplos

1. Ejemplo con único número

0	0	0	0	0	0	0	...
R	A	N	N	N	N	N	...

   Resultado: longitud de la lista - 2

   Para cualquier longitud de lista, si la misma solo contiene elementos idénticos, solo pueden pintarse 2 elementos (ya que los rojos y los azules deben ser estrictamente crecientes y decrecientes respectivamente).

   En general, sea l una lista cualquiera y sea $f(l)$ su solución, $0 \leq f(l) \leq max(|l| - 2, 0)$ (en otras palabras, si la lista tiene más de dos elementos, la solución siempre incluye por lo menos dos elementos pintados).

2. Ejemplo con un solo color

0	1	2	3	4	5	6	...
R	R	R	R	R	R	R	...

   Resultado: 0 (óptima)

   Para cualquier longitud de lista, si la misma es estrictamente creciente todos los elementos pueden pintarse del mismo color. El problema no requiere que haya elementos pintados de ambos colores.

   Esta regla aplica para cualquier secuencia l con elementos $l_1, l_2, ..., l_n$ tales que

   $$(\forall i, j \leftarrow i < j)(l_i < l_j)$$

   es decir, para toda lista estrictamente creciente. La regla vale para el caso opuesto (la lista es estrictamente decreciente), pintando de azul en lugar de rojo.

   Para este ejemplo también existe una cantidad de soluciones igual a la longitud de la lista en las que una de las posiciones está pintada del otro color (solo uno puede estar pintado de ese color).

2. Resolución por *Backtracking*

2.1. Descripción del algorítmo

La solución por Bactracking emplea recursividad en cada paso, y recorre el espacio de soluciones como si el mismo fuese un arbol en preorden.

A continuación se propone un algoritmo para las llamadas recursivas. Para el caso inicial, se toma $Indice = 0$ y $Res = 0$.

Data: Lista, la lista de números a pintar
Data: UltRojo, el valor del úlitmo rojo que pintamos (si existe)
Data: UltAzul, el valor del úlitmo azul que pintamos (si existe)
Data: Indice, la posición que estamos mirando ahora
Data: Res, la cantidad de elementos no pintados hasta ahora
if *Indice* == |*Lista*| **then**
  Llegamos al final de la lista, no hacer nada
else
  Calcular el mejor resultado sin pintar
  Llamada Recursiva: {Lista, UltRojo, UltAzul, Indice+1, Res+1}
  Guardar ese resultado como Res
  if *no existe UltRojo* ***or*** *UltRojo < Lista[Indice]* **then**
    Calcular el mejor resultado pintando de Rojo
    Llamada Recursiva: {Lista, Lista[Indice], UltAzul, Indice+1, Res}
    Si ese resultado es menor que Res, guardarlo como Res
  end if
  if *no existe UltAzul* ***or*** *UltAzul > Lista[Indice]* **then**
    Calcular el mejor resultado pintando de Azul
    Llamada Recursiva: {Lista, UltRojo, Lista[Indice], Indice+1, Res}
    Si ese resultado es menor que Res, guardarlo como Res
  end if
end if
Result: Res, la menor de todas las sub-soluciones

Cabe destacar que el orden en que se calculan los posibles resultados no es relevante, en tanto se conserve el menor de todos.

En la implementación propuesta, UltRojo y UltAzul comienzan con el menor y mayor valor representable de entero respectivamente, a modo de simplificar la comparación.

2.2. Cota de complejidad

El algoritmo propuesto realiza hasta tres llamadas recursivas en cada paso (asumiendo que el próximo número se puede pintar de cualquier color). Esto siempre es cierto para los primeros dos elementos.

En el mejor caso, un paso cualquiera llama a solo una llamada recursiva (no se puede pintar el número).

En cualquier llamada recursiva, el tamaño del problema a resolver disminuye en 1 (se avanza/pinta un solo número).

La función de complejidad sería

$$T(BT(n)) = \begin{cases} 1, & \text{si n == 0} \\ 3\ \text{T(BT(n-1))} + \text{O(1)}, & \text{si no} \end{cases}$$

donde n es la cantidad de números restantes, es decir, $|Lista| - Indice$.

Como, en peor caso, el algorítmo realiza 3 llamadas recursivas n veces, la complejidad resultante es $\mathbf{O(3^n)}$, donde n es el largo de la lista de números de entrada (es decir, cuántos numeros contiene la lista). Este es el espacio de soluciones completo, ya que cualquiera de los n elemento puede tener uno de 3 estados (existen 3^n combinaciones distintas).

El caso ideal para el algoritmo es una lista de números idénticos, dando una complejidad en mejor caso de $\Omega(n^2)$, ya que solo se deben pintar 2 elementos de color distinto y hay n posiciones válidas para cada uno de ellos.

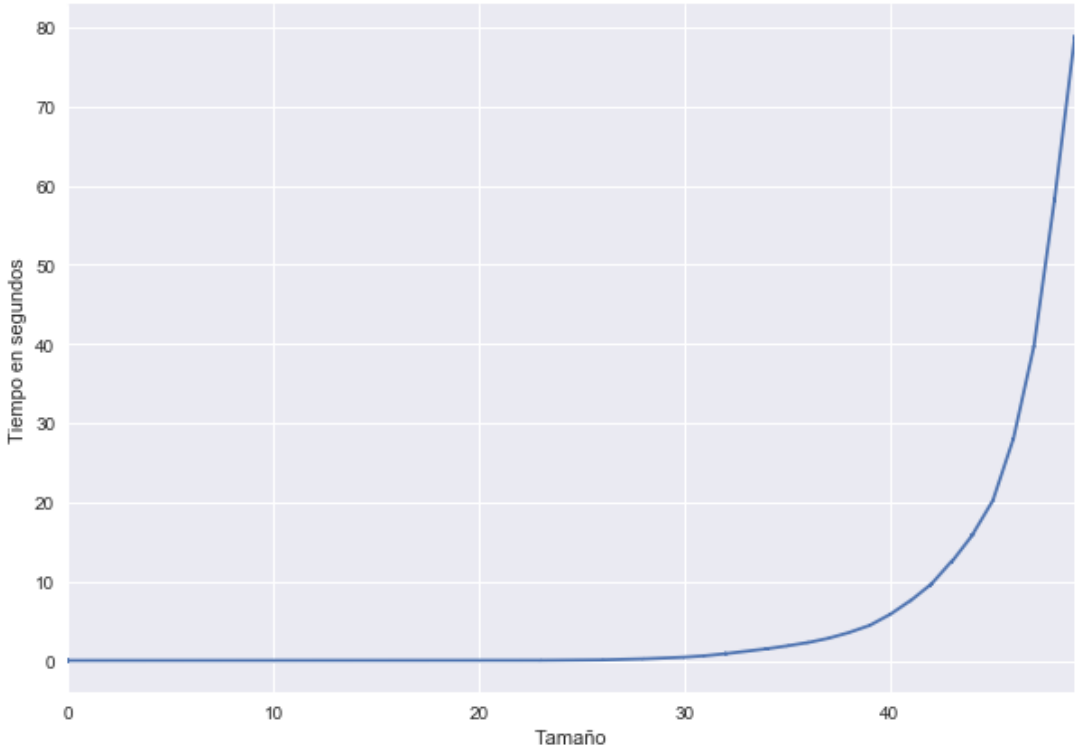
Otro caso que vale la pena destacar es que, si la lista es estrictamente creciente/decreciente, la complejidad es de orden $\Omega(n * 2^n)$, ya que un solo elemento puede ser pintado del color opuesto, y todos los demás pueden o no ser pintados o ser pintados del color ideal (rojo si es creciente o azul en caso opuesto).

2.3. Gráfico de complejidad

El siguiente gráfico muestra la relación entre la cantidad de elementos y el tiempo que el algoritmo toma en resolver el problema. Para el mismo se utilizaron listas con contenido random, ignorando el resultado real de las mismas. A su vez, cada caso se corrió 20 veces para tratar de reducir el ruido generado por otras aplicaciones corriendo en el equipo.

Los tiempos fueron medidos con las utilidades de `std::chrono` de C++11, corriendo dentro de los equipos de los laboratorios del DC.

Como se puede apreciar, el tiempo crece muy rápidamente con respecto al tiempo. Originalmente las mediciones estaban en microsegundos ($seg \times 10^{-6}$), pero fueron escaladas a segundos para este caso. Casos de tamaño mayor a 50 no fueron testeados por el gran costo temporal asociado con esos tests.



3. Mejora del algoritmo propuesto (poda de resultados)

3.1. Descripción del algorítmo

Las podas apuntan a obviar el cálculo de soluciones que ya sabemos peores que otras ya calculadas:

- Si sabemos que en alguna rama encontramos una solución final de tamaño i y la rama actual tiene una solución parcial de $j \leq i$, podemos obviar el resto de la rama actual.
- Si sabemos que existe una rama con solución final 0, existe una solución óptima y no requerimos calcular nada más.

A continuación se propone un algoritmo para las llamadas recursivas. Para el caso inicial, se toma $Indice = 0$, $Actual = 0$ y $Mejor = |Lista|$.

La variable $Mejor$ es de entrada/salida, es decir, las llamadas recursivas pueden modificar el valor. Al final del algoritmo, la misma almacenará el mejor resultado.

```
Data: Lista, la lista de números a pintar
Data: UltRojo, el valor del úlitmo rojo que pintamos
Data: UltAzul, el valor del úlitmo azul que pintamos
Data: Indice, la posición que estamos mirando ahora
Data: Actual, la cantidad de elemntos no pintados hasta ahora
Data: Mejor, la mejor solución hasta ahora
if Actual < Mejor then
    if Indice == |Lista| then
        Llegamos al final de la lista
        Mejor es Actual
    else
        if no existe UltRojo or UltRojo < Lista[Indice] then
            Calcular el mejor resultado pintando de Rojo
            Llamada Recursiva: {Lista, Lista[Indice], UltAzul, Indice+1, Res, Mejor}
            La función modifica el valor de Mejor si corresponde
        end if
        if Mejor ≠ 0 then
            if no existe UltAzul or UltAzul > Lista[Indice] then
                Calcular el mejor resultado pintando de Azul
                Llamada Recursiva: {Lista, UltRojo, Lista[Indice], Indice+1, Res, Mejor}
                La función modifica el valor de Mejor si corresponde
            end if
            if Mejor ≠ 0 then
                Calcular el mejor resultado sin pintar
                Llamada Recursiva: {Lista, UltRojo, UltAzul, Indice+1, Res+1, Mejor}
                La función modifica el valor de Mejor si corresponde
            end if
        end if
    end if
end if
Result: Ninguno, pero Mejor puede ser modificado si se encuentra una mejor subsolución.
```

A diferencia del algoritmo de Backtracking básico, el orden en que se calculan los casos recursivos es importante, ya que podemos encontrar un caso óptimo y podar el resto del espacio de soluciones.

3.2. Cota de complejidad

El algoritmo propuesto, al igual que el algorítmo base de Backtracking, realiza hasta tres llamadas recursivas en cada paso. En mejor caso, no se realiza ninguna llamada recursiva (se realiza una poda).

En cualquier llamada recursiva, el tamaño del problema a resolver disminuye en 1 (se avanza/pinta un solo número).

La función de complejidad sería

$$T(BTMejorado(n)) = \begin{cases} 1, & \text{si n == 0} \\ 3\ \text{T(BTMejorado(n-1))}, & \text{si no} \end{cases}$$

donde n es la cantidad de números restantes, es decir, $|Lista| - Indice$.

Es decir, en órdenes de complejidad, BTMejorado (Backtracking con podas) es igual a BT (Backtracking sin podas).

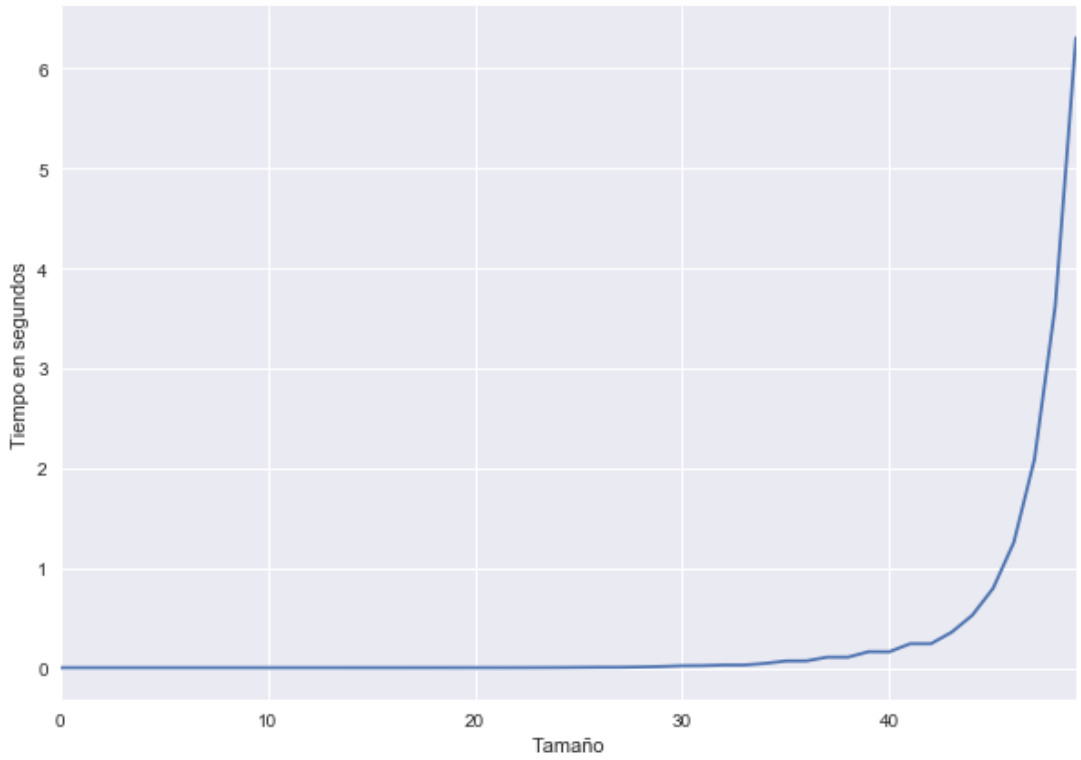
Esto no quiere decir que las podas son irrelevantes, las mismas pueden cortar el tiempo de procesamiento de manera importante, pero los cambios en complejidad son de orden menor a 3^n, por lo que esta sigue siendo la complejidad en peor caso.

El mejor caso para el algoritmo se da si la lista es estrictamente creciente. En este caso, se realiza solo una llamada recursiva por nivel, hallando un caso óptimo al llegar al final de la lista y podando el resto, por lo que la complejidad resultante es de $\Omega(n)$.

Este es un detalle implementativo, ya que en el algoritmo propuesto primero se realizan las llamdadas recursivas pintando de rojo de ser posible. De haberse utilizado un orden distinto, el mejor caso podría haberse dado con una entrada diferente.

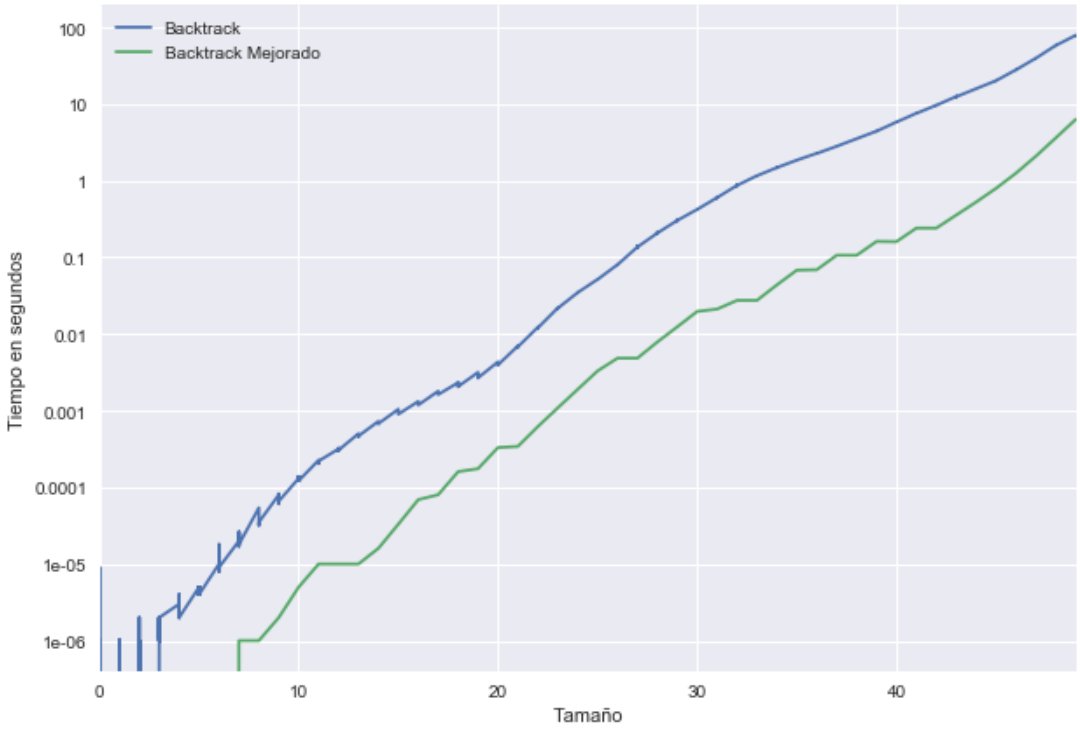
3.3. Gráfico de complejidad

El siguiente gráfico utiliza el mismo criterio que el anterior. Si bien la diferencia no es visualmente notoria, debe notarse que los valores del eje vertical son mucho menores (alrededor de 1/10 de los valores equivalentes sin poda).

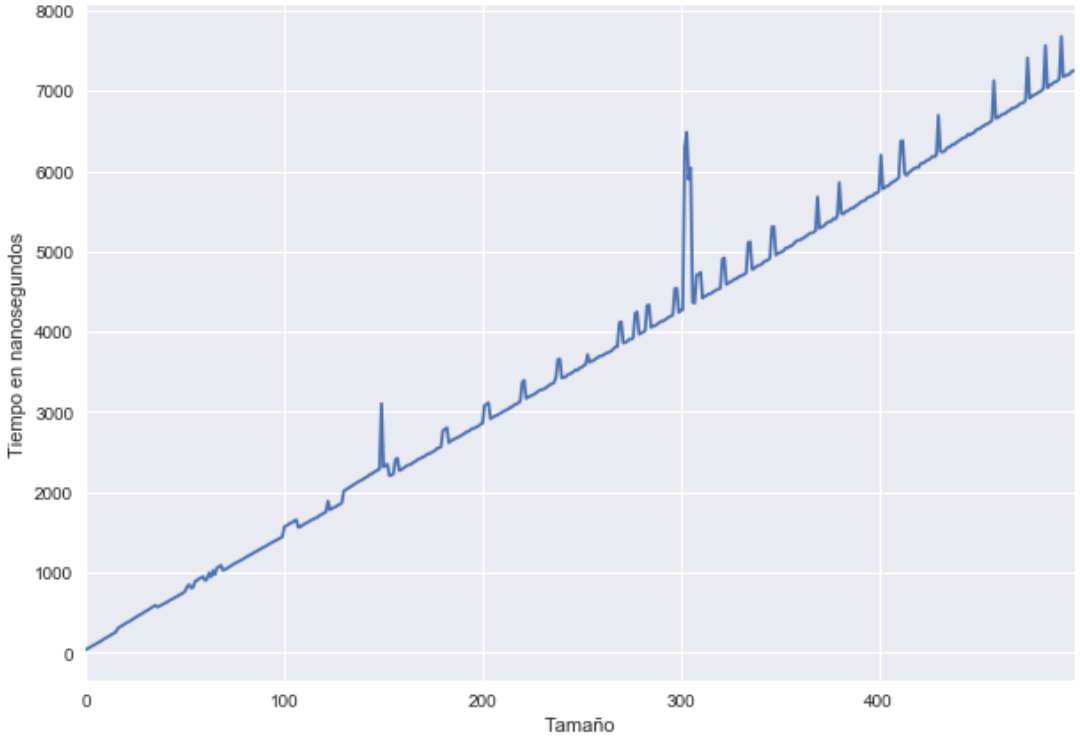


Se agrega también el siguiente gráfico en el que se muestran ambos algoritmos, utilizando una escala logarítmica para poder comparar mejor los órdenes de complejidad. El algoritmo mejorado es irregular debido a que las listas testeadas son aleatorias y pueden tener mejor o peor comportamiento frente a las podas definidas.

Como evidencia el gráfico logarítmico, si bien la mejora con las podas es notoria (como se mencionó antes, aproximadamente 1/10 del tiempo), ambos algorítmos crecen en órdenes similares.



Como nota final, al analizar el mejor caso del algoritmo, el tiempo de procesado de estas era casi despreciable ya que la complejidad es lineal al tamaño de la lista. Por este motivo, no es práctico comparar estos casos, pero si se incluye un gráfico ilustrativo de dicho caso, medido con listas de enteros consecutivos entre 1 y $|Lista|$. Notese que el tiempo está medido en nanosegundos por el requerimiento de precisión extrema, y las mediciones tienen ruido por procesamiento ajeno al problema.



4. Resolución por *Programación Dinámica*

4.1. Descripción del algorítmo

La solución por Programación Dinámica utiliza memoización: casos menores se guardan en una memoria interna y son reutilizados para calcular casos mayores.

En el algoritmo que se propone, la memoria toma forma de arreglo de dos dimensiones, donde cada dimensión expresa la posición en la lista donde se encuentra el último elemento pintado de uno u otro color. Al calcular un casillero, se toma el color más avanzado y se recorren los anteriores para encontrar la mejor subsolución.

Se debe considerar que no se calcula ninguno de los valores correspondientes a la diagonal de este arreglo, es decir, posiciones que representarían que un mismo elemento está pintado de ambos colores, ya que esto no es válido.

Data: Lista, la lista de números a pintar
Data: Memoria, el arreglo de dos dimensiones con soluciones previas
Data: Rojo, la posición del rojo que estamos calculando ahora
Data: Azul, la posición del azul que estamos calculando ahora
if *Memoria[Rojo][Azul] no fue calculado todavía* **then**
  if *Rojo == Azul == |Lista|* **then**
    Estamos en el caso base (nada pintado)
    Guardamos en Memoria[Rojo][Azul] —Lista— (todos sin pintar) y lo marcamos como calculado
  else if *Azul == |Lista|* ***or*** *Rojo > Azul* **then**
    Buscamos el mejor valor para los rojos anteriores.
    for $i \leftarrow 0$ **to** *Rojo - 1* **do**
      if $i \neq$ *Azul* ***and*** *Lista[i] ¡Lista[Rojo]* **then**
        Llamada recursiva: {Lista, Memoria, i, Azul}
      end if
    end for
    Llamadas recursiva: {Lista, Memoria, |*Lista*|, Azul} (caso sin pintar rojos)
    Sea Min el menor resultado de todas las llamadas recursivas que realizamos
    Guardamos en Memoria[Rojo][Azul] Min - 1 (fue pintado Rojo) y lo marcamos como calculado
  else
    Buscamos el mejor valor para los azules anteriores.
    for $i \leftarrow 0$ **to** *Azul - 1* **do**
      if $i \neq$ *Rojo* ***and*** *Lista[i] ¿Lista[Azul]* **then**
        Llamada recursiva: {Lista, Memoria, Rojo, i}
      end if
    end for
    Llamada recursiva: {Lista, Memoria, Rojo, |*Lista*|} (caso sin pintar azules)
    Sea Min el menor resultado de todas las llamadas recursivas que realizamos
    Guardamos en Memoria[Rojo][Azul] Min - 1 (fue pintado Azul) y lo marcamos como calculado
  end if
end if
Result: Memoria[Rojo][Azul], la mejor subsolución con Rojo y Azul como últimos elementos pintados de cada color (se garantiza que esa memoria fue calculada)

En la implementación provista, la memoria se rellena de forma *bottom-up*, es decir los casos más básicos primero, pero el orden no altera el producto ya que en cada paso la llamada recursiva asegura que cualquier posición que se necesite esté calculada, y solo se calcula una vez.

A su vez, cada dimensión de la memoria tiene de un casillero de largo extra para considerar el caso en que no se han pintado todavía casilleros de ese color. La memoria almacena si ya fue calculada y el mejor resultado posible para la misma.

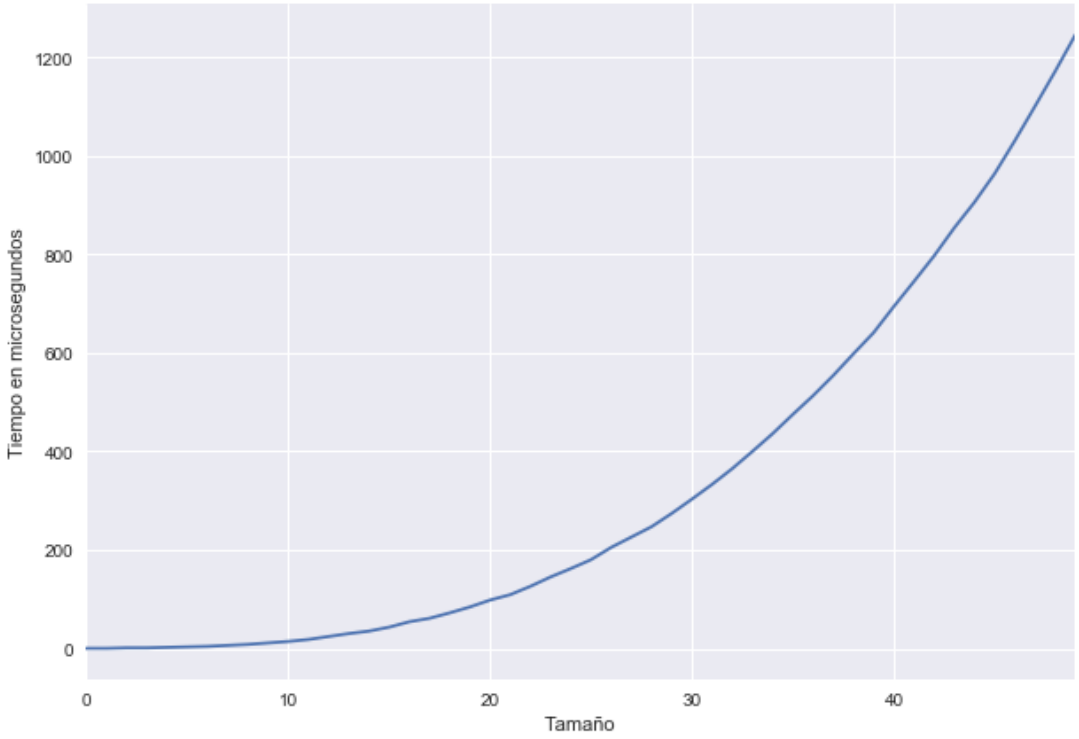
4.2. Cota de complejidad

Este algoritmo realiza, en peor caso, n llamadas recursivas por paso. Sin embargo, como cada resultado se memoiza, cada llamada recursiva cuesta O(1) amortizado.

Como el algoritmo rellena toda la memoria (salvo las posiciones inválidas), y el tamaño de la misma es de n^2 (siendo n el largo de la lista), el orden de complejidad del algoritmo es $O(n^3)$.

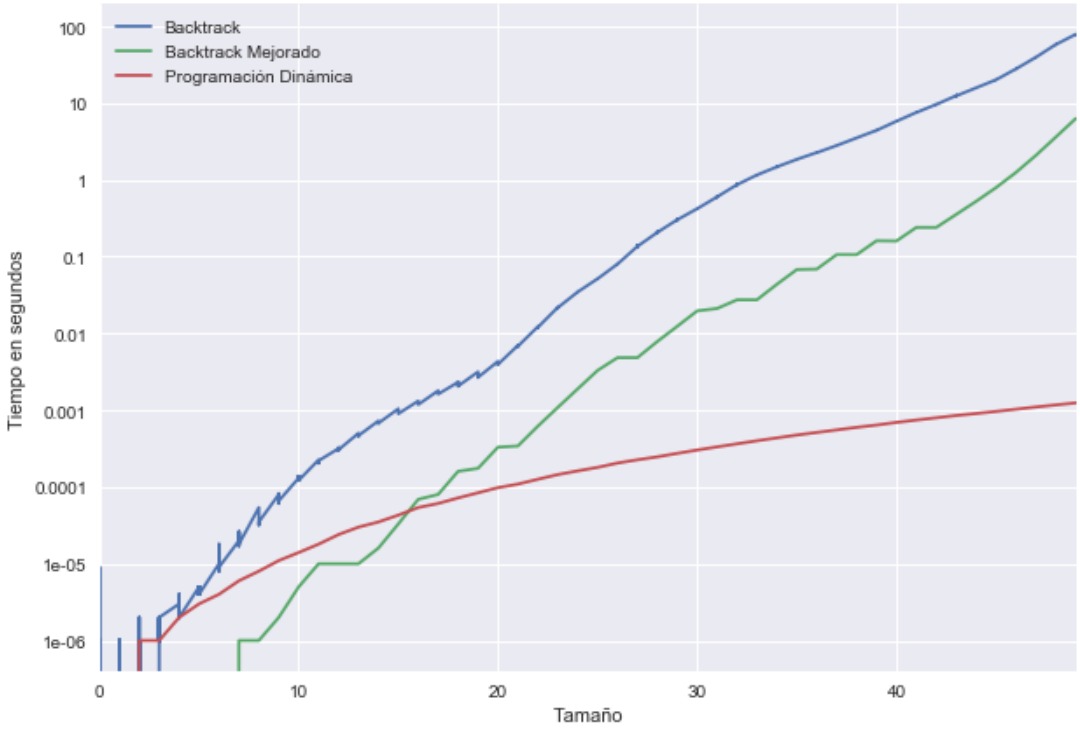
4.3. Gráfico de complejidad

El siguiente gráfico, si bien fue medido de la misma manera que los otros, utiliza una escala en microsegundos. Tanto la forma de la figura, más gradual que en los gráficos anteriores, como la necesidad de usar una unidad de medida más precisa dejan en claro que este algoritmo es mucho más eficiente.



A modo de comparación se agrega también el siguiente gráfico en el que se muestran los tres algoritmos utilizando la escala logarítmica, y utilizando nuevamente segundos para el eje Y.

En este caso se puede notar la diferencia en orden de complejidad comparado con los algoritmos basados en Backtracking: mientras que aquellos se asemejan levemente a una recta lineal en escala logarítmica (por su complejidad exponencial), este algoritmo es polinómico y por ende describe una curva logarítmica.



5. Apéndices

5.1. Apéndice I: generación de datos

Para poder analizar las complejidades de los algoritmos propuestos, se utilizaron las siguientes herramientas para generar mediciones y graficar datos:

- Un generador de listas aleatorias basadas en `std::rand()` (parte de la biblioteca estándar de C++).

  for *i* ← *0* **to** *n - 1* **do**
  | Lista[i] ← std::rand()
  end for

  Cada una de estas listas se guardo en una memoria temporal para poder probar los tres algoritmos con el mismo input.

- Un generador de listas secuenciales para el mejor caso de Backtracking con podas

  for *i* ← *0* **to** *n - 1* **do**
  | Lista[i] ← i()
  end for

  Este generador se usa en partiuclar para el ejercicio 2. Si bien las entradas fueron probadas en otros algoritmos, solo el Backtracking con podas puede sacar mayor provecho de esta entrada, y tiene un impacto relevante y fácil de medir en su performance.

- Mediciones de tiempo con `std::chrono` (parte de la biblioteca estándar de C++11).

  Sea Mejor el mejor tiempo medido (todavía no inicializado) **for** *i* ← *0* **to** *Repeticiones - 1* **do**
  | Comienzo ← `std::chrono::high_resolution_clock::now()`
  | Invocar Resolver(lista) (correspondiente al algoritmo)
  | Fin ← `std::chrono::high_resolution_clock::now()`
  | Actual ← `std::chrono::duration_cast<std::chrono::microseconds>(Fin - Comienzo).count()`
  | **if** *Mejor no está inicializado* ***or*** *Actual* < *Mejor* **then**
  | | Mejor ← Actual
  | **end if**
  end for

  Cada algoritmo fue probado varias veces con cada entrada, conservando solo el valor de tiempo menor para reducir el ruido por procesos ajenos al problema.

  La unidad de medición preferida fue microsegundos (`std::chrono::microseconds`, $seg \times 10^{-6}$), pero también se utilizó nanosegundos en algunas ocasiones, y los gráficos fueron escalados a segundos cuando correspondía por la dimensión de las variables.

- Graficado con `matplotlib.pyplot` y `pandas` (con Python y Jupyter Notebook)

  Se utilizaron los DataFrames de Pandas para el manejo de datos (guardados en `.csv`) y el graficado, en conjunto con matplotlib. Por el escalado a segundos, algunos detalles de los ejes se manejaron a mano con matplotlib.

5.2. Apéndice II: herramientas de compilación y testing

Durante el desarrollo se utilizaron las siguientes herramientas:

- CMake

  Se decidió utilizar CMake para la compilación por su simplicidad y compatibilidad con otras herramientras. Junto con el código se provee el archivo `CMakeLists.txt` para compilar el mismo.

- Google Test

  Para generar tests unitarios con datos reutilizables se usó Google Test. Dichos archivos eran importados por otro `CMakeLists.txt` y no están incluídos en la presente entrega del trabajo práctico.

- Namespace `Utils`

  Dentro de `Utils.h` se definieron 2 funciones útiles para el trabajo práctico: una de ellas crea un vector desde el stdin, para ser usado en los tres ejercicios; la otra es una función de logging que fue utilizada al programar para detectar errores y ver otros detalles del proceso.

  La función `log` sigue estando incluída en los algoritmos, pero su funcionalidad se encuentra apagada por un `#define` y no debería generar ningún costo adicional (ya que usa printf por detrás y no genera el output salvo que sea necesario).