Algoritmos y Estructuras de Datos III

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Abril 2017

Trabajo Práctico 1

Alumno	LU	Correo electrónico
Seijo, Jonathan Adrián	592/15	jon.seijo@gmail.com

Índice

1. Introducción

1.1. Explicación del problema

Dada una secuencia A de números, se quieren pintar cada uno de ellos con rojo, azul o dejarlos sin pintar. Una aclaración importante es que los elementos de A no pueden modificarse, ni tampoco cambiarse su orden inicial. Lo unico que puede hacerse con ellos es colorearlos (o no).

Para que una secuencia de colores se considere **válida** es necesario que se cumplan ciertas condiciones:

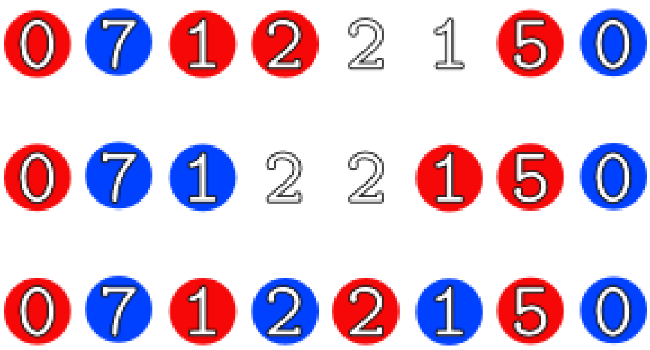
1. Todos los elementos de color rojo están ordenados por valor de forma estrictamente creciente
2. Todos los elementos de color azul están ordenados por valor de forma estrictamente decreciente

(Estrictamente significa que no hay numeros consecutivos iguales)

Las secuencias de colores válidas pueden tener diferentes cantidades de elementos sin pintar.
El objetivo del problema es encontrar la **mínima cantidad de elementos sin pintar** de todas las secuencias válidas que pueden formarse a partir de A.

1.2. Ejemplo

Supongamos que $A = [0, 7, 1, 2, 2, 1, 5, 0]$. Veamos *algunas* de las posibles secuencias de colores válidas:



Consideremos los colores del tercer caso para ver que es una secuencia válida.

1. **Rojos:** [0, 1, 2, 5] (estrictamente crecientes)
2. **Azules:** [7, 2, 1, 0] (estrictamente decrecientes)

Podemos ver que diferentes formas de pintar de rojo y azul nos obligan a dejar algunos elementos sin pintar para que la secuencia sea válida. En el caso de este ejemplo la **mínima cantidad de elementos sin pintar** que puede obtenerse de A es **0**, como puede verse en la tercer combinación.

2. Backtracking

2.1. Solución naive

Llamo A a la secuencia de números que quiero pintar, y n a la cantidad de elementos en A. De todas las secuencias válidas de colores que puedo formar quiero saber cual es la mínima cantidad de elementos que puedo dejar sin pintar.

Una forma natural de pensar la solución es la siguiente: genero todas las formas de pintar posibles, y veo cual es el mínimo sin pintar que puede usarse para las secuencias que son válidas. Esa es la idea central detrás de ambos algoritmos de backtracking. Veamos entonces una posible implementación, la forma *naive*.

El primer elemento puede ser Rojo, Azul o Ninguno. Dado el color del primero, el segundo elemento puede también ser Rojo, Azul o Ninguno. Fijados el primero y el segundo, el tercero puede ser tomar cualquiera de las tres posibilidades, y así siguiendo.

Una vez fijos los colores de los n elementos, reviso si la secuencia de colores que se formó es válida. (Esto es, que los elementos rojos estén ordenados crecientemente y los azules decrecientemente, ambos de forma estricta).

Si la secuencia formada era válida, entonces cuento la cantidad de elementos sin pintar, y devuelvo ese número. La respuesta final es se consigue tomando el mínimo de todos los mínimos.

Como detalle de implementación, en caso de que la secuencia formada no sea válida, devuelvo un valor infinito para que no afecte al valor mínimo solución. Esta solución existe porque no pintar ningún elemento de ningún color es siempre una solucion válida **finita**

2.2. Pseudocódigo

```
procedure BACKTRACK(secuencia(Colores) colores, int actual)
    if actual = n then
        if EsValido(colores) then
            return CantSinPintar(colores)
        else
            return ∞
    else
        colores[actual] ← Rojo
        minimoConRojo ← backtrack(colores, actual + 1)

        colores[actual] ← Azul
        minimoConAzul ← backtrack(colores, actual + 1)

        colores[actual] ← Ninguno
        minimoSinPintar ← backtrack(colores, actual + 1)

        return Min(minimoConRojo, minimoConAzul, minimoSinPintar)
```

Auxiliares:

```
procedure ESVALIDA(secuencia(Colores) colores)
    bool rojoValido ← EsCreciente(DameRojos(colores))        ▷ O(n)
    bool azulValido ← EsDecreciente(DameAzules(colores))     ▷ O(n)
    return (rojoValido ∧ azulValido)
```

procedure CANTSINPINTAR(secuencia(Colores) *colores*)
    return Tamaño(DameSinPintar(*colores*)) ▷ $O(n)$

Para resolver el problema original, llamo a la función con los siguientes parámetros:

procedure RESOLVER NAIVE(secuencia(int) A)
    backtrack($secuenciaColoresVacia(n)$, 0)

2.3. Complejidad

El algoritmo presentado visita todas las posibles combinaciones de colores. Cada uno de los elementos tiene tres posibilidades, y como hay n elementos, la cantidad de combinaciones posibles es 3^n. Por lo tanto, visitar todas las posibilidades es $O(3^n)$.

Además, para cada combinación, se revisa en $O(n)$ si es una secuencia valida o no. Por lo tanto, la complejidad total del algoritmo es $O(n * 3^n)$

Otra forma de verlo es pensando en el arbol de recursión que se va formando al llamar a la función. Cada nivel representa al elemento iésimo de A, y cada nodo representa el color del elemento. De todo nodo se desprenden tres posibilidades hasta llegar al nivel n. Al llegar a una hoja, se decide si la secuencia *hasta esa hoja* es válida, en tiempo lineal.

Sabiendo que en un árbol ternario el nivel **i** tiene 3^i nodos, y sabiendo que el arbol tiene n niveles, la cantidad de nodos que se visitan es:

$$\sum_{i=0}^{n} 3^i = \frac{3^{n+1}}{2} = O(3^n)$$

El costo de las visitas de nodos no es el total, pues para cada una de las hojas se verifica si la secuencia obtenida es válida o no. Las hojas se encuentran en el útimo nivel n, entonces el árbol de recursión tiene 3^n hojas, donde cada hoja tiene costo $O(n)$. El costo de operar en las hojas entonces es $3^n * O(n) = O(n * 3^n)$

El costo **total** es la suma entre visitar todos los nodos y operar en las hojas, es decir:

$$O(3^n) + O(n * 3^n) = O(n * 3^n)$$

2.4. Solución con poda

El algoritmo anterior funciona, pero puede mejorarse si tenemos en cuenta algunas observaciones:

1. Pueden filtrarse las secuencias válidas a medida que vamos construyendo la secuencia de colores.
2. Recorriendo de izquierda a derecha, lo único que se necesita para saber si es válido pintar cierto elemento de algun color, es ver el valor del último elemento antes del actual que fue pintado de ese mismo color.
3. Puede conocerse la cantidad de elementos sin pintar a medida que se va pintando la secuencia.
4. La cantidad de elementos sin pintar **no** aumenta si se pinta rojo o azul.
5. Si la cantidad de elementos sin pintar es mayor al mínimo entontrado, entonces no vale la pena seguir explorando esa secuencia, pues la cantidad de elementos sin pintar no puede disminuir.

2.5. Pseudocódigo

Siguiendo la idea principal del algoritmo anterior, pero teniendo en cuenta las nuevas ideas, podemos mejorar nuestro algoritmo:

```
procedure BACKTRACK(int actual, int ultRojo, int ultAzul, int cantSinPintar)
    if actual = n then
        minimoTotal ← cantSinPintar
        return minimoTotal
    else
        if (ultRojo = n) ∨ (A[actual] > A[ultRojo]) then
            minRojo ← backtrack(actual + 1, actual, ultAzul, cantSinPintar)

        if (ultAzul = n) ∨ (A[actual] < A[ultAzul]) then
            minAzul ← backtrack(actual + 1, ultRojo, actual, cantSinPintar)

        if cantSinPintar + 1 < minimoTotal then
            minSinPintar ← backtrack(actual + 1, ultRojo, ultAzul, cantSinPintar + 1)

        return Min(minimoConRojo, minimoConAzul, minimoSinPintar)
```

Para resolver el problema original, llamo a la funcion con los siguientes parámetros:

```
procedure RESOLVER PODA(secuencia(int) A)
    backtrack(0, n, n, 0)
```

Como detalle de implementación, cuando el último rojo (o azul) es n, significa que no hay ningún rojo (o azul) aún, porque los índices del arreglo comienzan en cero.

2.6. Complejidad con poda

Haciendo un análisis similar al del algoritmo naive, pensando en el arbol de recursión, tenemos que como cota superior se visitan todos los nodos del árbol. La validez de la secuencia se comprueba a medida que se desciende en la recursión, pues no se entra al siguiente paso si la propiedad de los colores no se cumple. Además, se va contando la cantidad de elementos sin pintar a medida que se eligen colores, por lo que tampoco se usa tiempo en eso.

Como todo el costo se encuentra en la cantidad de nodos que se visita, y como la cantidad máxima de nodos es $O(3^n)$, el costo de obtener la respuesta usando este algoritmo con poda en peor caso es $O(3^n)$, mejorando la complejidad del anterior.

3. Programación Dinámica

3.1. Solución bottomup

Generar todas las combinaciones es muy caro computacionalmente. Se puede conseguir algo mejor si pensamos el problema desde otro ángulo.

Supongamos que para todo (i, j) ya tenemos el valor mínimo de elementos sin pintar dado que el i-ésimo es el último rojo y el j-ésimo es el último azul. De este modo, lo único que tendríamos que hacer es tomar el mínimo de todas las combinaciones de los últimos rojos y azules. Esto es así porque de todas las posibles soluciones para un último rojo y último azul fijos, hay una combinación que es óptima, y es ésta la que nos interesa obtener.

Entonces nuestro problema se reduce a: suponiedo que la secuencia tiene el último rojo y el último azul en posiciones fijas, ¿cuál es la mínima cantidad de elementos sin pintar que podemos obtener?

Llamo DP$[i][ur][ua]$ a la mínima cantidad sin pintar hasta $A[i]$ siendo que el elemento en ur es el último rojo y el elemento en ua es el último azul. (Si ur o ua es n, represento que no hay rojo o azul)

La idea es ir llenando la matriz DP en orden, para encontrar los valores minimos hasta llegar a $i = n$.

En los casos base donde i es 0, lleno la matriz con ceros.

Veamos ahora DP$[i][ur][ua]$ para el elemento en la posición i (dados un ur y un ua) existen 3 casos

- **No lo pinto**: En este caso, la cantidad de elementos sin pintar **aumenta en uno** con respecto al mínimo hasta $i - 1$. Es decir, la solución es igual a 1+DP$[i-1][ur][ua]$

- **Pinto i de rojo**:

1. Si i es el último rojo, o si es posible que i sea color rojo (pues no rompe la propiedad) entonces la solución es la misma que la solución hasta $i - 1$ siendo que i es el último rojo. Es decir, es igual a DP$[i-1][i][ua]$
2. Si estoy en la rama donde no hay rojo ($ultRojo = n$), o si estoy en la rama donde el último azul era i, entonces no puede ser que i sea rojo, por lo que no hay solución. (Devuelvo ∞)
3. Si no era posible que i sea rojo, entonces no hay solucion para este caso. (Devuelvo ∞)

- **Pinto i de azul**: Análogo al caso de pintar con rojo

Cuando termina la iteración, DP$[i][ur][ua]$ contiene el mínimo de los tres casos. En todos los casos, sé que el óptimo anterior que utilizo para calcular la solución actual ya está calculado porque los voy calculando en orden.

La solución total del problema, es el mínimo valor hasta $i = n$ considerando todas las combinaciones de ur y ua.

3.2. Pseudocódigo bottomup

---

procedure RESOLVER BOTTOMUP(secuencia(int) A)
    Matriz3 $DP \leftarrow$ Matriz3(n) ▷ Creo Matrix de tres dimesiones de tamaño n
    DP[0][ur][ua] $\leftarrow 0$ ▷ Lleno con 0 cuando $i = 0$

    for $ultRojo \in [0..n]$ **do**
        for $ultAzul \in [0..n]$ **do**
            for $i \in [1..n]$ **do**

                $minNada \leftarrow$ BottomupNada(i, ur, ua)
                $minRojo \leftarrow$ BottomupRojo(i, ur, ua)
                $minAzul \leftarrow$ BottomupAzul(i, ur, ua)

                DP[i][ur][ua] $\leftarrow$ Min($minNada$, $minRojo$, $minAzul$)

    return Min(DP[n][ur][ua] $\forall ur, ua$)

---

procedure BOTTOMUPNADA(int $actual$, int ur, int ua)
    return 1+ DP[$actual - 1$][ur][ua];

---

procedure BOTTOMUPROJO(int $actual$, int ur, int ua)
    bool $esUltimoRojo \leftarrow ((actual = ur) \wedge (actual \neq ua))$
    bool $cumplePropiedad \leftarrow ((actual < ur) \wedge (A[actual] < A[ur]))$

    if $esUltimoRojo \vee cumplePropiedad$ **then**
        return DP[$actual - 1$][$actual$][ua]
    else
        return ∞

---

procedure BOTTOMUPAZUL(int $actual$, int ur, int ua)
    bool $esUltimoAzul \leftarrow ((actual = ua) \wedge (actual \neq ur))$
    bool $cumplePropiedad \leftarrow ((actual < ua) \wedge (A[actual] > A[ua]))$

    if $esUltimoAzul \vee cumplePropiedad$ **then**
        return DP[$actual - 1$][ur][$actual$]
    else
        return ∞

3.3. Complejidad

Crear la Matriz de tres dimensiones de tamaño n, cuesta $O(n^3)$. Llenar los casos bases cuesta $O(n^2)$. Luego aparecen tres ciclos anidados, cada uno de ellos de tamaño n. En el cuerpo se ejecutan solo $O(1)$ comparaciones, asignaciones y accesos a arreglo. El costo de los tres ciclos principales es $O(n^3)$ Se devuelve el mínimo de las n^2 combinaciones de ur y ua, en $O(n^2)$.

El costo total es:

$$O(n^3) + O(n^2) + O(n^3) + O(n^2) = O(n^3)$$

3.4. Solución topdown

Muy similar al enfoque BottomUp, pero calculando de forma recursiva. En vez de comenzar desde $i = 0$ hacia adelante, comienzo desde $i = n$ hacia atrás.

Solución(i, ur, ua) devuelve la mínima cantidad sin pintar que pueden usarse hasta i, siendo ur el último rojo y siendo ua el último azul. Las soluciones se van a ir calculando recursivamente (pues cada solución necesita de la anterior) y voy almacenando los resultados en la matriz DP.

De esta forma, los resultados son calculados una única vez. La respuesta final es el mínimo de todas las combinaciones de ur y ua para $i = n$.

3.5. Pseudocódigo topdown

```
procedure RESOLVER TOPDOWN(secuencia(int) A)
    Matriz3 DP ← Matriz3(n, −1)                    ▷ Matriz de 3 dimensiones, llena con −1
    for ultRojo ∈ [0..n] do
        for ultAzul ∈ [0..n] do
            minSinPintar ← Min(minSinPintar, Solución(n − 1, ultRojo, ultAzul))
    return minSinPintar
```

```
procedure SOLUCIÓN(int actual, int ultRojo, int ultAzul)
    if actual = −1 then return 0
    if DP[actual][ultRojo][ultAzul] ≠ −1 then return DP[actual][ultRojo][ultAzul]

    minRojo ← TopdownCasoRojo(actual, ultRojo, ultAzul)
    minAzul ← TopdownCasoAzul(actual, ultRojo, ultAzul)
    minSinPintar ← TopdownCasoSinPintar(actual, ultRojo, ultAzul)

    return Min(minRojo, minAzul, minSinPintar)
```

```
procedure TOPDOWNCASOROJO(int actual, int ultRojo, int ultAzul)
        ▷ Si no hay rojo o si el actual es azul, entonces no puedo considerar que el actual sea rojo
    if (ultRojo = n) ∨ (actual = ultAzul) then
        return ∞
    else                    ▷ Si soy el último rojo, ó si puedo serlo porque cumplo la propiedad:
        if (actual = ultRojo) ∨ (i < ultRojo ∧ A[i] < A[ultRojo]) then
            return Solución(actual − 1, actual, ultAzul)
        else
            return ∞
```

procedure TopdownCasoAzul(int $actual$, int $ultRojo$, int $ultAzul$)
    ▷ Si no hay azul o si el actual es rojo, entonces no puedo considerar que el actual sea azul
    if $(ultAzul = n) \vee (actual = ultRojo)$ **then**
        return ∞
    else ▷ Si soy el último azul, ó si puedo serlo porque cumplo la propiedad:
        if $(actual = ultAzul) \vee (i < ultAzul \wedge A[i] > A[ultAzul])$ **then**
            return Solución($actual - 1$, $ultAzul$, $actual$)
        else
            return ∞

procedure TopdownCasoSinPintar(int $actual$, int $ultRojo$, int $ultAzul$)
    ▷ No puede pasar que el actual sea rojo o azul, pero que lo quiera dejar sin pintar
    if $(actual = ultRojo) \vee (actual = ultAzul)$ **then**
        return ∞
    else
        return 1+ Solución($actual - 1$, $ultRojo$, $ultAzul$)

3.6. Complejidad

Se quiere calcular la complejidad de *Resolver TopDown*.

Al principio se crea una matriz de tres dimensiones llena con -1, de tamaño n por cada lado. Esto tiene un costo de $O(n^3)$.

Para calcular Solución(i, ur, ua) puede verse que se llama recursivamente a otras instancias de *Solución*. Esto hace parecer que la complejidad total es muy grande, pero veamos que no es así considerando el problema desde una perspectiva mas alta.

Notemos que cada *Solución* es calculada una única vez. Esto es así porque vamos guardando las soluciones en una matriz, por lo que la próxima vez que quiera esa solución particular, ya voy a tenerlo en $O(1)$. Entonces solo considero el costo de las soluciones únicas.

Supongamos que para una instancia particular se realizan m llamados recursivos, *parecería* entonces que resolver una instancia cuesta m. Pero si miramos con atención, el algoritmo esta guardando las respuestas a los m subproblemas, por lo que en realidad estamos calculando m soluciones con m llamados recursivos. Además de los llamados recursivos, las otras operaciones son $O(1)$ comparaciones y asignaciones. Así, el costo de calcular m soluciones es $O(m)$. Vimos entonces que *Solución* tiene un costo amortizado.

La cantidad máxima de diferentes combinaciones de parámetros es n^3. El costo de calcular las n^3 Soluciones es $O(n^3)$ si consideramos las observaciones anteriores.

También podemos pensar que, al igual que en *BottomUp*, lo que estamos haciendo es llenar la matriz DP usando en $O(1)$ los valores ya calculados, solo que esta vez de forma recursiva. Hay n^3 lugares que llenar, por lo que el costo de llenar es $O(n^3)$.

Al final del algoritmo, devolvemos el máximo en $O(n^2)$. El costo total es entonces:

$$O(n^3) + O(n^3) + O(n^2) = O(n^3)$$

4. Experimentación

4.1. Aclaraciones

En todos lo experimentos, para cada tamaño de secuencia se corrió el programa 50 veces, guardando el tiempo de cada ejecución. Para elegir un valor representativo de la muestra tomo la **media** de cada tamaño. (También llamado *promedio*).

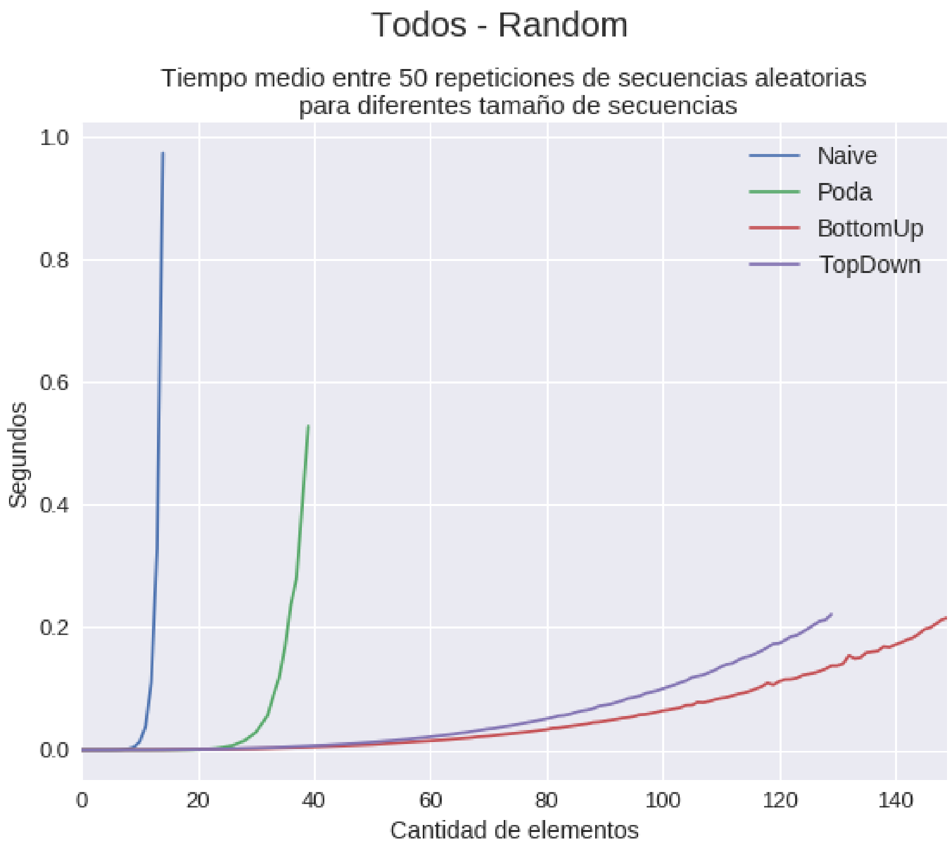
La generación de secuencias aleatorias de tamaño n fue hecho en Python3 usando:

```
[random.randrange(cota_superior) for _ in range(n)]
```

donde *cota_superior* es un número lo suficientemente grande para que no condicione la muestra. Para las muestras graficadas, el valor usado como cota es 1000000.

4.2. Aleatorias

Podemos observar una clara diferencia entre los algoritmos de backtracking y los algoritmos de programación dinámica. Esto es lo esperado si tenemos en cuenta que en nuestro análisis original concluímos que los primeros tenían complejidad **exponencial**, mientras que los segundos tenían complejidad **polinomial**.



Era esperable que el algoritmo con *poda* sea mucho mas eficiente que el algoritmo *naive*, pero veamos mas de cerca que pasa con los de programación dinámica:

Para instancias de tamaño ≤ 40 *BottomUp* y *TopDown* se comportan de manera muy similar, pero para tamaños mas grandes las curvas comienzan a separarse.

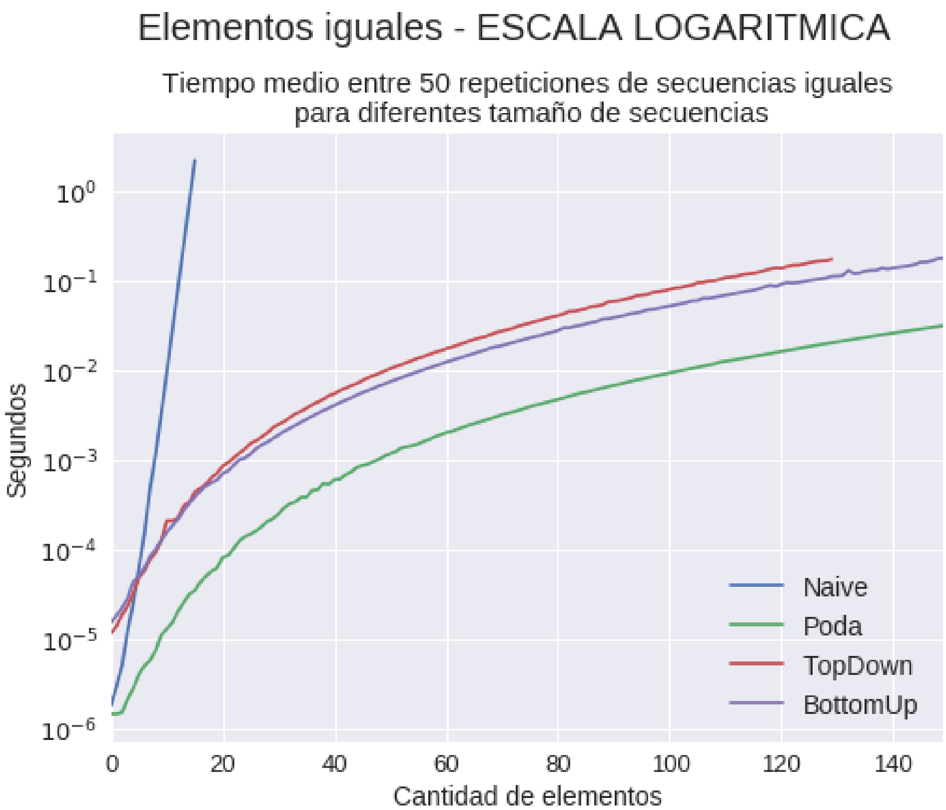
En el análisis teórico, tanto *TopDown* como *BottomUp* tenían la misma complejidad. Lo que podemos deducir de estas mediciones es que *TopDown* tiene una constante oculta mas elevada. Esto es razonable ya que en esta técnica se realizan llamados recursivos y en *BottomUp* no.

4.3. Elementos iguales

En una secuencia donde todos los elementos son iguales, como máximo pueden pintarse un elemento de rojo y otro de azul, pero no más que esto ya que no es posible encontrar secuencias *estrictamente* crecientes o decrecientes. Que sean todos iguales es un buen caso, ya que en rasgos generales las técnicas que revisan la validez de la secuencia a medida que la construyen solo podrían "entrar" en la rama donde los elementos no se pintan.

El algoritmo de *backtracking naive* tiene tiempos tan malos que no permiten ver con claridad las diferencias entre las otras técnicas, por lo que el gráfico está en **escala logarítmica** en y.

En el gráfico podemos ver cómo el *backtracking con poda* es el que produce los resultados mas eficientes. Esto es razonable si consideramos la observación del primer párrafo, y si tenemos en cuenta que los algoritmos de *programación dinámica* siempre crean una matriz de tres dimensiones de tamaño n^3, sin importar el contenido de la secuencia.



4.4. Elementos crecientes

En una secuencia donde los elementos son *estrictamente crecientes* la solución óptima se consigue pintando todos los elementos del mismo color, obteniendo el mínimo sin pintar en 0.

El algoritmo *naive* de backtracking sigue produciendo los peores resultados, ya que siempre revisa todas las combinaciones posibles. Al ser tiempos tan malos, no permiten ver las diferencias con las demás técnicas. Es por esto que el gráfico lo muestro en **escala logarítmica** en eje y.

Los algoritmos que usan *programación dinámica* siguen necesitando $O(n^3)$ tiempo para crear la matriz donde van a guardarse los datos.

(Es posible que las irregularidades de la curva se deban a que los tiempos sean demasiado pequeños para ser medidos con precisión)