Grupo: 44

13/12/19

Fuerza Bruta, Backtracking y Programación Dinámica

PROGRAMACIÓN III

AMUSQUÍVAR POPPE, JOSÉ MARÍA
JESWANI TEJWANI, PRASHANT
MARÍN REYES, IGNACIO JOSÉ

Índice

Información de la práctica

1. **Nombre y apellido de los miembros del grupo**

   José María Amusquívar Poppe
   Prashan Jeswani Tejwani
   Ignacio José Marín Reyes

2. **Horario de laboratorio asignado al grupo para defender las prácticas (Martes 12:30-14:30, Miércoles 10:30-12:30, Jueves 8:30-10:30, etc)**

   La defensa del trabajo se realizará en la hora de prácticas del grupo 44, grupo al que pertenecemos (Jueves 10:30-12:30).

3. **Lenguajes de programación elegidos.**

   Java, C++ y Python.

4. **Añade aquí cualquier comentario o descripción adicional que quieras hacer sobre esta práctica.**

   Tanto en Java como Python, los algoritmos de Memoization y Tabulation los hemos complementado para que obtenga qué elementos del vector ha cogido para formar los dos subvectores en el caso de que exista cuya partición. Para ver el contenido de los subvectores, estos se mostrarán con la opción -do (Display Output). Hemos aplicado la misma técnica que en el problema de la mochila o el ladrón.

Problema de partición

1. **Nombre del ejercicio elegido, enlace a la web del código y breve descripción.**

   El algoritmo elegido se llama "Partition Problem". Este algoritmo resuelve la búsqueda de dos subvectores cuya suma individual sea igual a la mitad de la suma total del vector.

   Un ejemplo de uso sería:

   For example, S = {3,1,1,2,2,1},
   We can partition S into two partitions each having sum 5.

   S_1 = {1,1,1,2}
   S_2 = {2,3}.

   Note that this solution is not unique. Below is another solution.

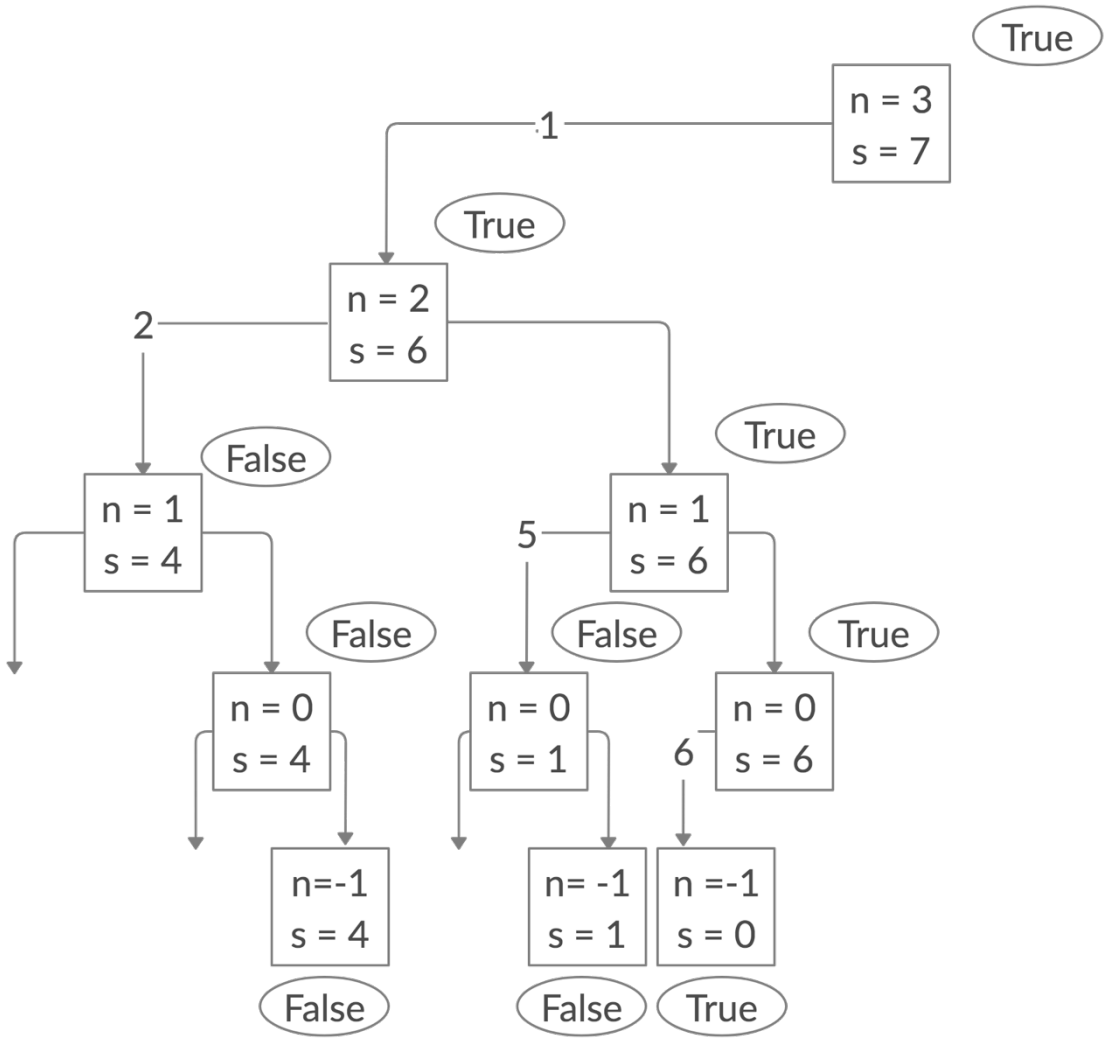
   S_1 = {3,1,1}
   S_2 = {2,2,1}

   URL informativo del algoritmo: https://www.techiedelight.com/partition-problem/

2. **Recurrencia que resuelve el problema mediante Fuerza Bruta y Programación Dinámica. Explica el significado de la recurrencia.**

   t(n, s) = t(n-1,s) : num[n] > s
   = t(n-1, s- num[n]) || t(n-1,s) : num[n] < s
   = False : n < 0
   = True : s = 0

   n= posición del vector
   s= suma de los elementos del subconjunto
   t(n-1,s)
   Significa que no incluye el elemento del vector en el subconjunto.
   t(n-1, s- num[n])
   Significa que incluye el elemento del vector en el subconjunto, y es necesario restar el valor del elemento del vector a s.
   Además, la condición num[n] > s, nos obliga a no incluir el elemento del vector en el subconjunto. Y la condición num[n] < s, nos fuerza a incluir el elemento en el subconjunto o a no incluirlo en el subconjunto.
   Los casos bases son:
   n < 0, el subproblema tiene 0 elementos del vector.
   s = 0, existen dos particiones con igual suma.



Por ejemplo, dado un vector = [6, 5, 2, 1].
num = [6, 5, 2, 1]
n = [0, 1, 2, 3]

La suma del vector es 6 + 5 + 2+ 1 = 14. Primero se comprueba que el número es par, 14 mod 2 = 0.
Si fuera impar, ya sabemos que no hay dos subconjuntos con igual suma, ya que la mitad de un número impar da un número real. Por ejemplo: 13/2=6,5.

Como es par, podría haber dos subconjuntos con igual suma, ya que 14/2=7. Cada subconjunto tendrá una suma igual a 7. Por tanto, debemos de buscar las combinaciones de subconjuntos que nos den una suma igual a 7.

En fuerza bruta comprobamos todas las combinaciones posibles de subconjuntos. En backtracking y programación dinámica hace una poda de sus ramas cuando num[n] > s. Es decir, no calcularía esas combinaciones de subconjuntos.

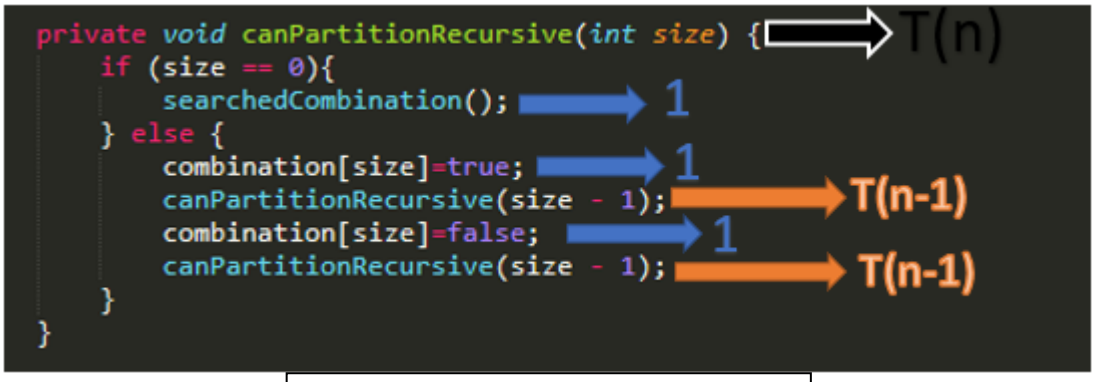
Para explicar el significado de t(n-1, s- num[n]) || t(n-1,s) y porque del uso de un or lógico debemos de realizar la siguiente traza. En nuestro ejemplo, si empezamos en la raíz del árbol, nos movemos a través del árbol superior y realizamos una traza de nuestra recurrencia, obtenemos el siguiente procedimiento.

Empezamos en t(3 , 7). Nos desplazamos a la izquierda.
Si num[3] < 7 => t(3-1, 7 – 1) = t(2, 6). Nos desplazamos a la izquierda.
Si num[2] < 6 => t(2-1, 6 - 2) = t(1, 4). Nos desplazamos a la derecha.
Si num[1] > 4 => t(1-1, 4) = t(0, 4). Nos desplazamos a la derecha.
Si num[0] > 4 => t(0-1, 4) = t(-1, 4). Llegamos al caso base.
Si n < 0 => Devolvemos False.

Subimos por la rama hasta llegar a t(2,6) devolviendo false en cada subproblema.
Repetimos el mismo proceso, hasta llegar al caso base de s=0, desde que devolvemos True, subimos por la rama devolviendo True, y cuando alcanzamos una intersección de la rama como en t(2,6), debido al t(n-1, s- num[n]) || t(n-1,s), obtenemos que FALSE or TRUE es TRUE. Por tanto, demostramos que debemos usar el operador lógico or.

3. **Fragmento de código fuente con el algoritmo implementado mediante la estrategia de Fuerza Bruta (recursivo y con iterador) en uno de los 3 lenguajes elegidos. Análisis asintótico.**

El análisis de fuerza bruta recursivo es parecido al juego de las torres de Hanoi, por lo que su recurrencia será también parecida, en este caso tenemos que la función de recurrencia es T(n). Esta función tiene un caso base, cuando size es igual a 0, qué entonces llama a una función externa. Y en la rama del "else" se tiene la recurrencia en sí, donde se evalúan todas las combinaciones posibles, restándole cada vez uno, de este modo se tiene que hay dos llamadas recursivas de tipo t(n-1) y otras dos asignaciones cuyo valor asignado será 1, al ser una operación esencial. De este modo se obtiene que la función de recurrencia es:



Análisis de la función de recurrencia.

Input:

$t(n) = 2\,t(n-1) + 1 \quad | \quad t(0) = 1$

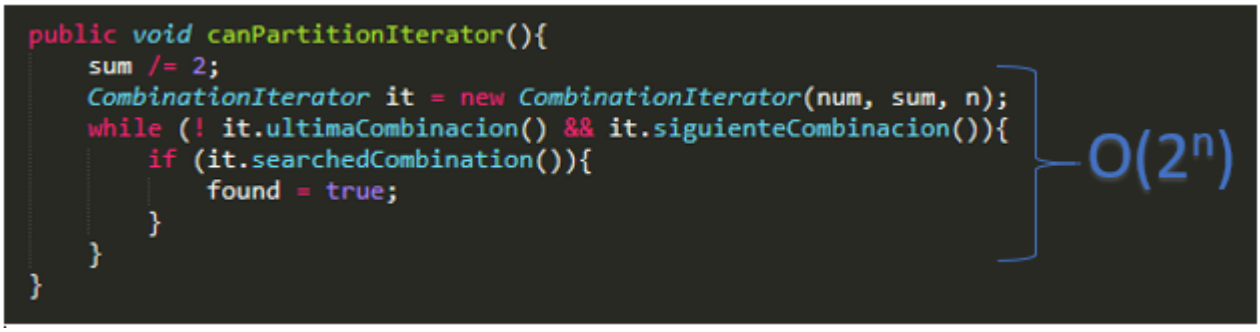
Recurrence equation solution:

$t(n) = 2^{n+1} - 1$

Solución de la recurrencia. Cálculo obtenido desde la página WolframAlpha.

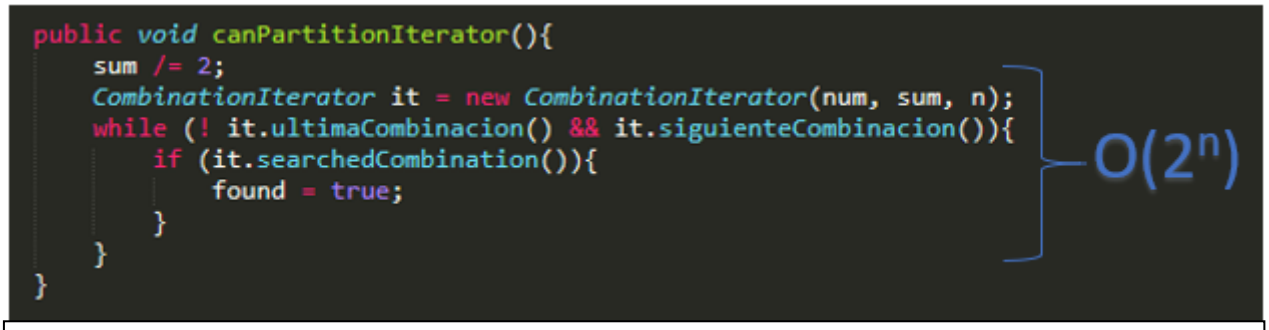
Para el análisis de este método iterativo con orden 2^n. En este caso, este orden se debe a que se trata del algoritmo de fuerza bruta, cuyo objetivo es recorrer todo el árbol de combinaciones, lo que quiere decir que para un vector de n elementos, se realizarán 2^n combinaciones, llenando el vector de 0s o 1s, según si lo excluye o lo incluye en la suma. Por tanto, este algoritmo terminará de ejecutarse cuando no existan más combinaciones posibles, qué en cuyo caso retornará un false en el método "siguienteCombinacion()", con lo que se consigue que el algoritmo salga del bucle pero no termine la ejecución del programa.

Al igual que BackTracking, Fuera Bruta utiliza este método que itera todas las posibles combinaciones, la diferencia principal recae en el método "siguienteCombinación()" de la clase Iterador que implementa BackTracking (isValid()), que evita que itere combinaciones imposibles para el problema planteado.



4 Fragmento de código fuente con el algoritmo implementado mediante la estrategia de Vuelta Atrás (Backtracking) (recursivo y con iterador) en uno de los 3 lenguajes elegidos.

Backtracking Iterador



Al igual que el algoritmo de fuerza bruta, éste también utiliza el mismo bucle "while", con las mismas condiciones de salidas. Excepto que este algoritmo se encarga de descartar aquellas combinaciones imposibles (poda), esto lo realiza usando el método isValidCombination().

```
private boolean isValidCombination() {
    return (totalValues() <= sum);
}
```

Método que comprueba que la suma de los elementos escogidos es menor o igual que la suma requerida.

```java
public boolean nextCombination() {
    for (int i = 0; i < combinacion.length; i++) {
        if (combinacion[i] == valormaximo) {
            combinacion[i] = valorminimo;
        } else {
            combinacion[i] += 1;
            if (isValidCombination()){
                return true;
            } else {
                combinacion[i] = valorminimo;
            }
        }
    }
    return false;
}
```

El coste del algoritmo es 2^n debido a que realiza todas las combinaciones posibles de subconjuntos que existen en el input vector. Además, debemos de tener en cuenta que realiza un número menor de combinaciones que fuerza bruta, debido a que poda la rama del árbol. Esto lo realiza si el método isValidCombination() devuelve falso, es decir, la combinación no es válida, por tanto, puede podar o quitar todas las siguientes combinaciones que estarían en la misma rama. Solamente poda la rama si la suma de los valores es mayor que sum, es decir, no es necesario comprobar si las siguientes combinaciones son válidas, ya sabemos que no serán válidas.

Backtracking recursivo

```java
    private boolean canPartitionRecursive(int n, int s)
        if (s <= 0) return true;

        if(n < 0) return false;

        if(s < num[n]){
            return canPartitionRecursive(n-1, s);
        }else{
            return (canPartitionRecursive(n-1, s) ||
            canPartitionRecursive(n-1, s-num[n]));
        }
    }
}
```

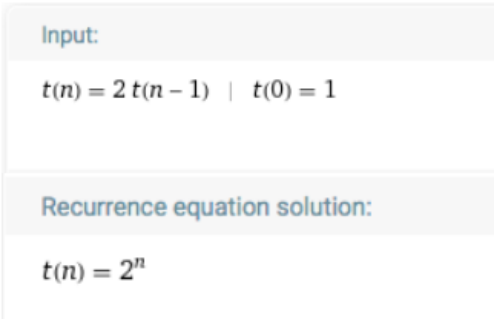
T(n)

1

T(n-1)

T(n-1)

Análisis de la función de recurrencia.



Solución de la recurrencia. Cálculo obtenido desde la página WolframAlpha.

Usando un análisis de la función de recurrencia y la página web WolframAlpha obtenemos t(n)= 2^n. Es decir, el algoritmo superior tiene un orden de 2^n.

5 Fragmento de código fuente con el algoritmo implementado mediante Programación Dinámica (memoization y tabulation) en uno de los 3 lenguajes elegidos. Análisis asintótico.

Memoization

```
private boolean canPartitionRecursive(int i, int s) {
    if(map[i][s] != null){
        return map[i][s];
    }

    if (s <= 0) return true;

    if (i == 0) return false;

    boolean result;
    if(s < num[i-1]){
        result = canPartitionRecursive(i-1, s);
    }else{
        result = canPartitionRecursive(i-1, s) ||
                 canPartitionRecursive(i-1, s-num[i-1]);
    }

    map[i][s] = result;

    return result;
}
```

El algoritmo de memoization tiene un orden de i*s, siendo i el total de números y s la suma del total de los números. Esto es debido a que el algoritmo realiza lo mismo que el algoritmo de bactracking recursivo anterior, la diferencia radica en que los subproblemas que se repiten no los vuelve a calcular a través de la recurrencia, los calcula solo una vez y los guarda en la matriz. Si el valor no está en la matriz realiza la recursividad para calcular el valor y almacenarlo en la matriz, en cambio, si ese subproblema del árbol ya está en la matriz, porque se hizo anteriormente, no vuelve a realizar el cálculo, sino que lo obtiene de la matriz. Por tanto, el coste va a ser esos accesos posibles a la matriz, y esos accesos posibles a la matriz serán equivalentes a i*s, valores obtenidos de las dimensiones de la matriz.

Tabulation

```
public void canPartitionTab(){
    sum /= 2;
    dp = new boolean[n + 1][sum + 1];

    for(int i=0; i <= n; i++) dp[i][0] = true;

    for(int s=1; s <= sum ; s++) dp[0][s] = false;

    for(int i=1; i <= n; i++) {
        for(int s=1; s <= sum; s++) {

            if (s < num[i-1]) {
                dp[i][s] = dp[i-1][s];
            }else{
                dp[i][s] = dp[i-1][s] || dp[i-1][s-num[i-1]];
            }
        }
    }
    found = dp[n][sum];
}
```

El algoritmo de tabulation tiene un orden de i*s, siendo i el total de números y s la suma del total de los números. Esto es debido a que tabulation realiza con el uso de la recurrencia todos los subproblemas posibles, empezando desde los casos base, para poder a través de la recurrencia y los casos calculados anteriormente, calcular todos los demás subproblemas hasta conseguir el último subproblema que coincide con la esquina inferior derecha de la matriz, donde tendremos un booleano que nos indica si es posible particionar el input vector en dos subconjuntos con igual suma.

6 Coste empírico de ejecución del algoritmo en Fuerza Bruta, Backtracking y Programación Dinámica en uno de los 3 lenguajes elegidos.

Los datos recolectados abajo están en C++.

N	FuerzaBrutaIterador	FuerzaBrutaRecursivo	BacktrackingIterador
4	0	0	0
6	0	0	0
8	0	0	0
10	0	0	0
12	0	0	0
14	0,001	0	0,001
16	0,008	0,003	0,009
18	0,034	0,014	0,035
20	0,143	0,06	0,152
22	0,702	0,264	0,643
24	2,585	1,225	2,68
26	10,727	4,659	11,668
28	46,874	19,073	45,208
30	188,983	82,031	192,509
32		350,652	

Debido al cambio de escala entre algoritmos, todas las tablas solicitadas no pueden estar conjuntas. Por ello se separa las tablas de programación dinámica de los señalados arriba.

N	Memoization
1000	0,042
2000	0,229
3000	0,552
4000	0,919
5000	1,542
6000	2,309
7000	3,3
8000	4,355
9000	5,322
10000	6,82
11000	8,637
12000	9,928
13000	11,687
14000	14,55
15000	17,069
16000	19,083
17000	21,711
18000	25,166
19000	28,874
20000	33,604
21000	39,358
22000	43,101
23000	55,277
24000	60,339

N	Tabulation
1000	0,014
6000	0,586
11000	2,259
16000	4,077
21000	7,072
26000	10,39
31000	14,512
36000	18,69
41000	24,893
46000	33,347
51000	39,279
56000	49,277
61000	59,166
66000	70,483

7 Copia de pantalla que muestre el uso del programa desde consola activando cada uno de los algoritmos implementados con las opciones que muestran el tiempo consumido en la ejecución del programa y leen los datos de entrada desde un fichero (en uno de los lenguajes elegidos).

Se incluye un fichero de texto “usage.txt” dentro del archivo comprimido entregado, en éste podrá ver todos los comandos usados para compilar, ejecutar o empaquetar cada uno de los proyectos presentados.

Además, se desarrollaron dos scripts “.bat” en Windows para automatizar estas tareas, de este modo, simplemente con ejecutar dichos scripts, se puede obtener los tiempos de ejecución deseados. La decisión de separar los dos scripts es sencilla, uno se encarga de compilar, y empaquetar en el caso de JAVA, todos los proyectos (6 por cada lenguaje de programación) reunirlos en una carpeta común (excepto Python), y el otro de obtener los tiempos de ejecución, por lo que el segundo no puede existir sin el primero.
Dado la necesidad de incluir todas las clases a la hora de compilar, se utilizó un vector que contiene los distintos nombres necesarios para tal acción.

```
for /l %%x in (0,1,5) do (

    call cd CPP/%%projectsNamei[%%x]%%

    if %%x == 0 (
        call g++ -o ../../Executables/%%exeNames[%%x]%%.exe %%projectsName[%%x]%%.cpp BackTrackingIt.cpp ParserArgs.cpp ReadFile.cpp IteratorCombinations.cpp
    ) else if %%x == 1 (
        call g++ -o ../../Executables/%%exeNames[%%x]%%.exe %%projectsName[%%x]%%.cpp BackTrackingR.cpp ParserArgs.cpp ReadFile.cpp
    ) else if %%x == 2 (
        call g++ -o ../../Executables/%%exeNames[%%x]%%.exe %%projectsName[%%x]%%.cpp BruteForceIt.cpp ParserArgs.cpp ReadFile.cpp IteratorCombinations.cpp
    ) else if %%x == 3 (
        call g++ -o ../../Executables/%%exeNames[%%x]%%.exe %%projectsName[%%x]%%.cpp BruteForceR.cpp ParserArgs.cpp ReadFile.cpp
    ) else if %%x == 4 (
        call g++ -o ../../Executables/%%exeNames[%%x]%%.exe %%projectsName[%%x]%%.cpp Memoization.cpp ParserArgs.cpp ReadFile.cpp
    ) else if %%x == 5 (
        call g++ -o ../../Executables/%%exeNames[%%x]%%.exe %%projectsName[%%x]%%.cpp Tabulation.cpp ParserArgs.cpp ReadFile.cpp
    )

    call cd ../../JAVA/%%projectsName[%%x]%%/src

    call javac org/apache/commons/cli/*.java %%projectsNamei[%%x]%%/*.java
    call jar -cfe ../../../Executables/%%exeNames[%%x]%%.jar %%projectsNamei[%%x]%%.%%projectsName[%%x]%% %%projectsNamei[%%x]%%/*.class org/apache/commons/cli/

    call cd ../../../
)
```

```
set projectsName[0]=PartitionProblemBackTrackingIterator
set projectsName[1]=PartitionProblemBackTrackingRecursive
set projectsName[2]=PartitionProblemBruteForceIterator
set projectsName[3]=PartitionProblemBruteForceRecursive
set projectsName[4]=PartitionProblemMemoization
set projectsName[5]=PartitionProblemTabulation

set projectsNamei[0]=partitionproblembacktrackingiterator
set projectsNamei[1]=partitionproblembacktrackingrecursive
set projectsNamei[2]=partitionproblembruteforceiterator
set projectsNamei[3]=partitionproblembruteforcerecursive
set projectsNamei[4]=partitionproblemmemoization
set projectsNamei[5]=partitionproblemtabulation

set exeNames[0]=pbtI
set exeNames[1]=pbtR
set exeNames[2]=pbfI
set exeNames[3]=pbfR
set exeNames[4]=pm
set exeNames[5]=pt
```

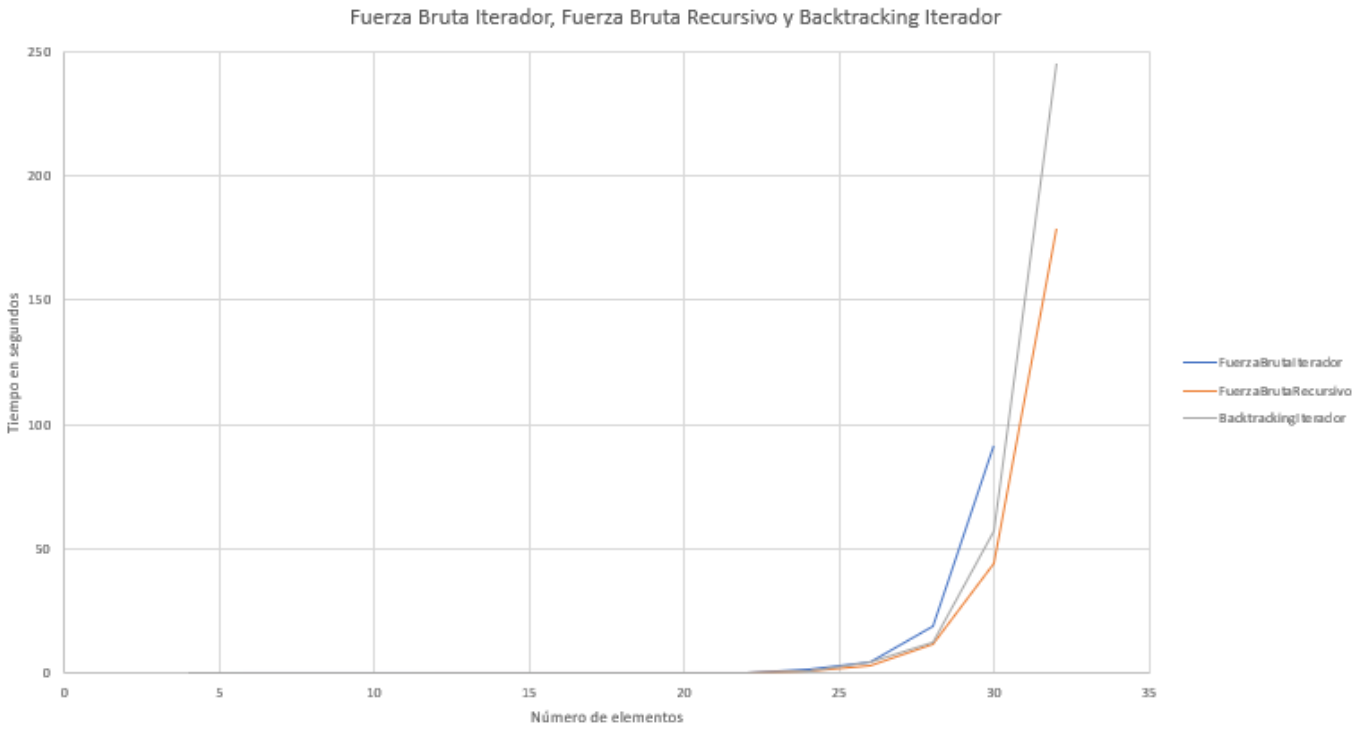
El primer vector almacena los nombres de los Main de Java y C++. El segundo almacena los nombres de los paquetes de JAVA. Y el tercero almacena los los nombres de los ejecutables obtenidos, tanto en C++ como en JAVA.
Ambos vectores están en el mismo orden, para evitar confusiones.

El segundo script es el que se encarga de obtener los tiempos de ejecución de cada algoritmo, usando los paquetes “.jar” y “.exe” que se obtuvieron con el script anterior. La salida de este script serán 6 ficheros, uno para cada algoritmo que incluye el tiempo para los tres lenguajes de programación. Y es en este dónde se puede ver lo que se pide.

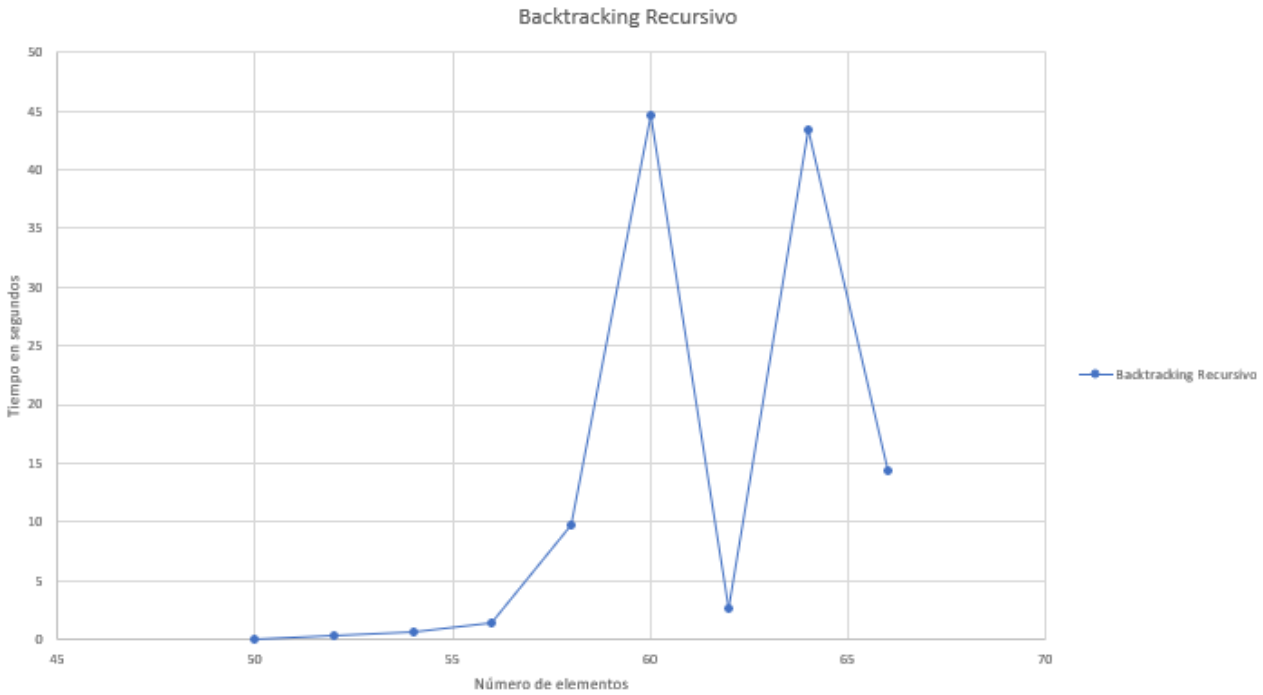
```
for /l %%x in (%min%, %jump%, %max%) do (
    cd ../FileGenerator/src
    call java -jar filesout.jar -n %%x -s -max 10


    cd ../../PYTHON
    echo Tiempo PYTHON >> ../Executables/times/resultadoM.txt
    python PartitionProblemMemoization.py -f ../fichVect.txt -dt >> ../Executables/times/resultadoM.txt

    cd ../Executables
    echo Tiempo JAVA >> times/resultadoM.txt
    java -jar pm.jar -f ../fichVect.txt -dt >> times/resultadoM.txt

    echo Tiempo C++ >> times/resultadoM.txt
    pm.exe -f ../fichVect.txt -t >> times/resultadoM.txt
    echo. >> times/resultadoM.txt
```

8. **[Opcional] Gráfico que muestra el coste empírico de ejecución en los 3 lenguajes procesando un fichero con los datos de entrada.**

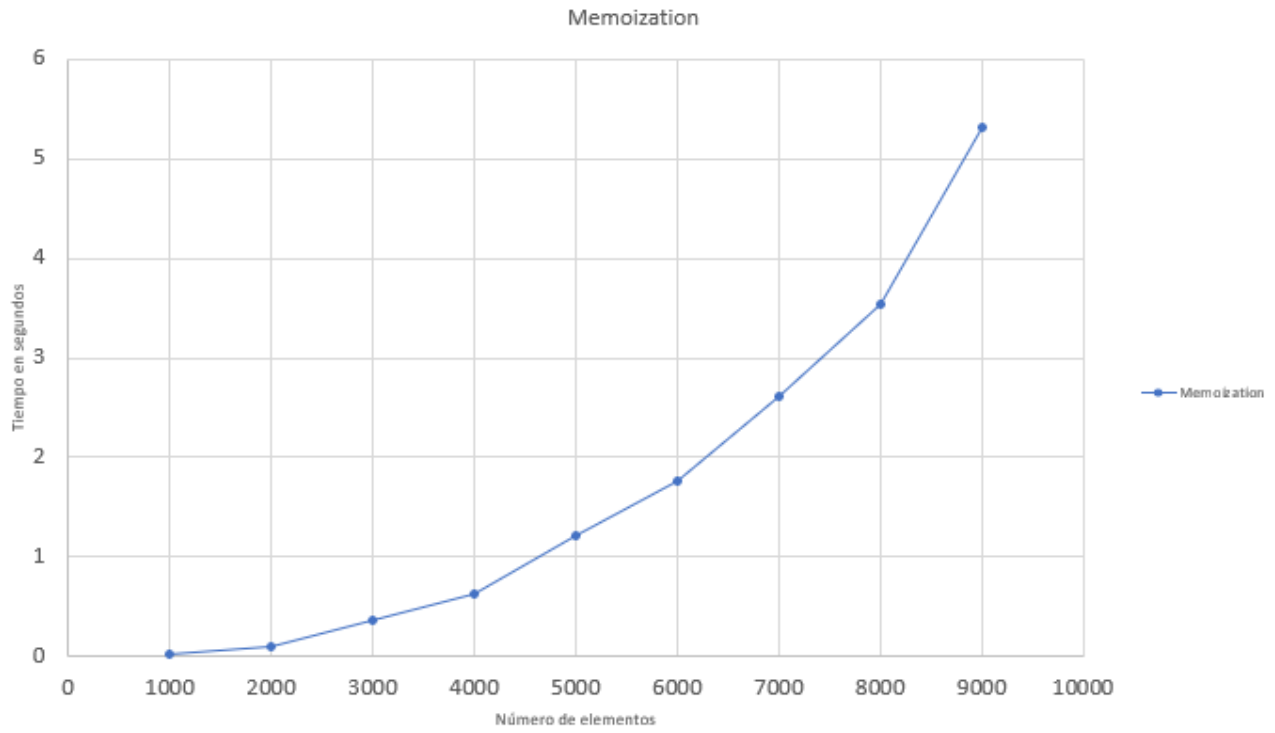
Java

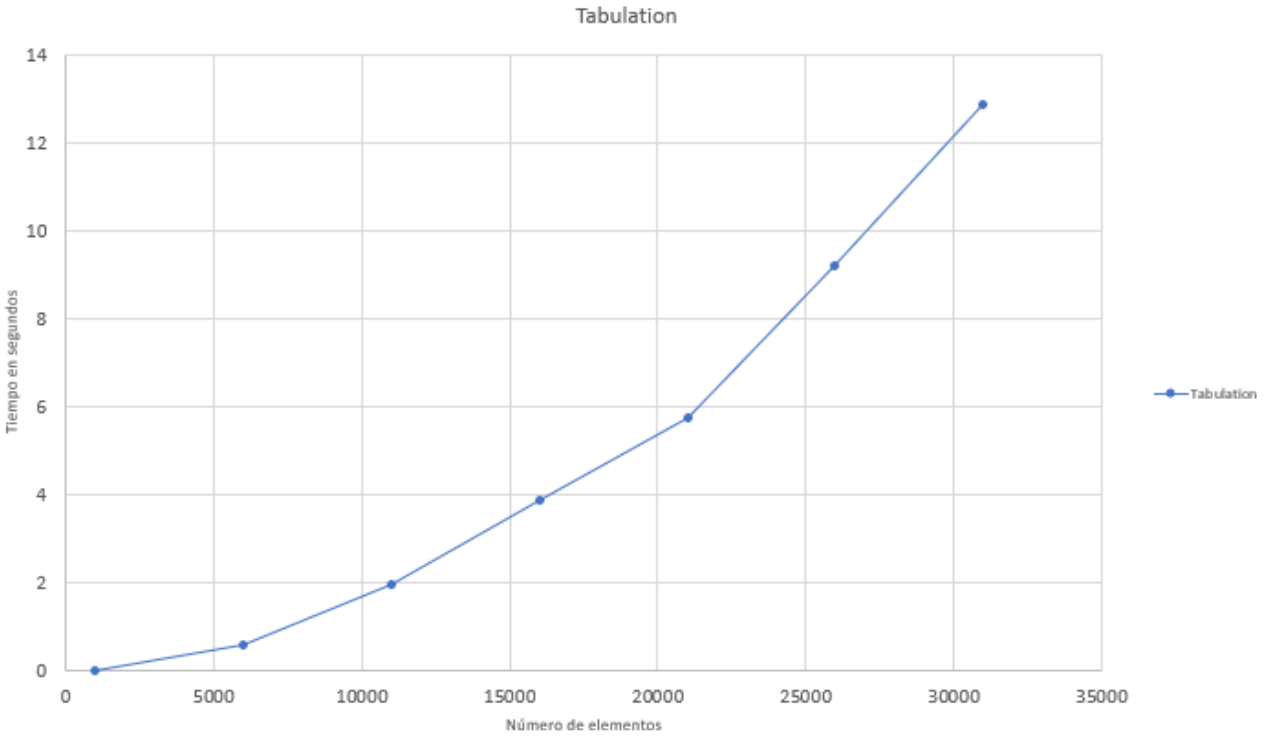


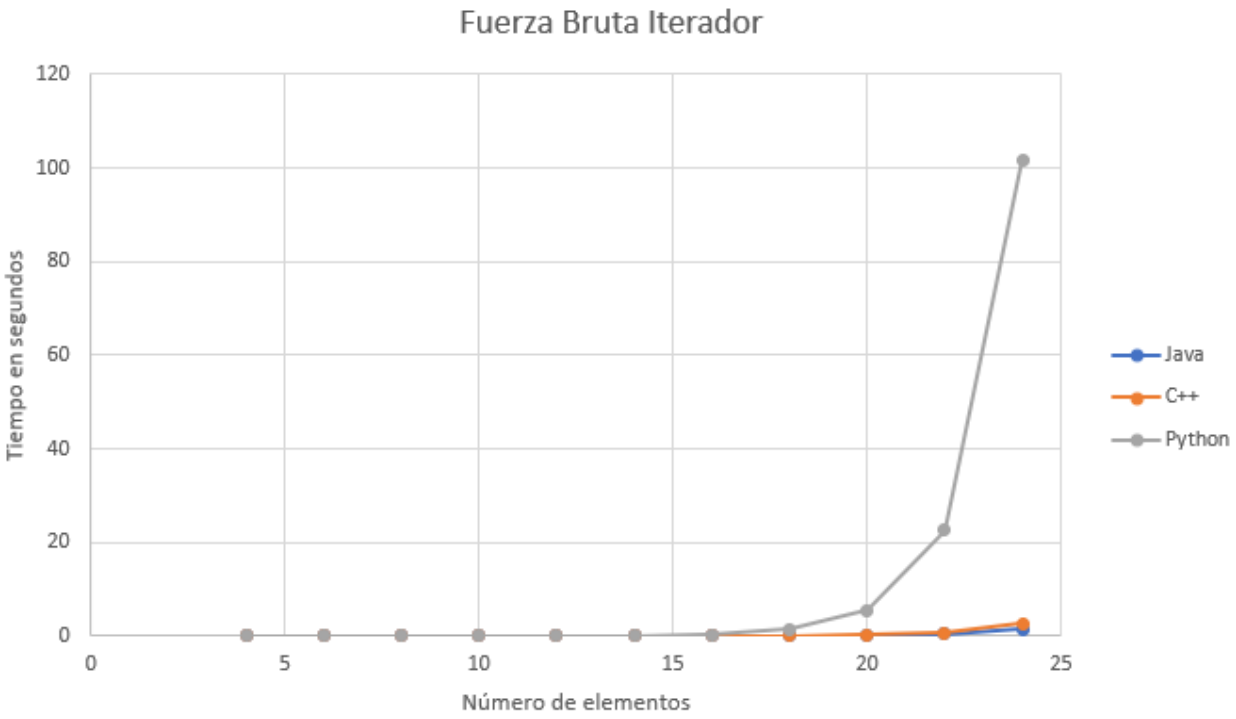
Acorde a lo redactado con anterioridad se puede comprobar que, efectivamente, el orden de este algoritmo en lo métodos de Fuerza Bruta (iterador y recursivo) y Backtracking iterador es 2^n. Se puede observar este comportamiento con la gráfica que tiene una tendencia exponencial.



Entendemos que los picos de la gráfica son debido a que existen diferentes casos, el mejor caso, el caso promedio o el peor caso, el cual tendrán un orden de 2^n en el peor caso.







Aquí se puede observar la comparación de eficiencia entre lenguajes de programación usando el mismo algoritmo (Fuerza Bruta con Iterador). Como es evidente Python sufre bastante con estos problemas respecto a lenguajes de programación como JAVA o C++. Y de igual manera, aquí se puede observar como el algoritmo con PYTHON encuentra su asíntota de manera repentina respecto a los otros dos qué aún no muestran tendencias exponenciales.