BÚSQUEDA DE SUBCONJUNTOS MEDIANTE BACKTRACKING

Daniel Paredes Santamaría
Miguel Bayón Sanz

1. ÍNDICE

1. ÍNDICE

2. DESCRIPCIÓN DEL CÓDIGO

- **2.1. Instrucciones de compilación**
- **2.2. Instrucciones de ejecución**
- **2.3. Métodos utilizados**
- **2.4. Estructuras de datos utilizadas**

3. EXPLICACIÓN DEL ALGORITMO

- **3.1. Explicación de porqué usamos backtracking**

4. BATERÍA DE PRUEBAS

5. REFERENCIAS

2. DESCRIPCIÓN DEL CÓDIGO

2.1. Instrucciones de compilación:

-Mediante cmd:

Se presupone que estamos en la carpeta donde se ubica el archivo.

Ejecutamos:

```
unzip p2_migbayo_danpare.zip
javac p2_migbayo_danpare.java
java p2_migbayo_danpare
```

-Mediante entorno de desarrollo (por ejemplo, Eclipse):

En la pestaña *File* hacemos click en *Import*.
En el apartado *General* seleccionamos *Existing projects into workspaces* y *Next*.
En *Select Archive File* hacemos click en *Browse* y buscamos p2_migbayo_danpare.zip

Con esto ya tendríamos el programa en nuestro Eclipse, por lo tanto podremos hacer click con el botón derecho en el proyecto ---> *Run As* ---> *Java Application* y ya podríamos ejecutarlo.

2.2. Instrucciones de ejecución:

El programa pedirá por pantalla una serie de números del tamaño que el usuario quiera, indicando que esta serie termina cuando se introduzca un 0. Esta serie será introducida por teclado.

A continuación, se pedirá también por pantalla el número a buscar mediante sumas con los distintos subconjuntos.

El programa finalizará introduciendo las distintas soluciones en un fichero creado en la misma carpeta del proyecto. Estas soluciones serán las distintas combinaciones de números cuya suma será exactamente el valor introducido en último lugar.

2.3. Métodos utilizados:

- *main*():

Genera y rellena el vector de números con los valores introducidos por el usuario por teclado, además de añadir de la misma manera el número a buscar. Por último, realiza la llamada a *buscar*() para comenzar las comprobaciones.

- *buscar*():

Recibe como parámetros el vector principal y la suma buscada.

Con un bucle *for* recorre el vector principal:

Entra en un primer *if* para comprobar simplemente que ese primer número obtenido es el mismo que la suma que queremos buscar, en caso afirmativo lo toma como subconjunto solución y llama a imprimir para que la introduzca en el fichero.

En caso negativo obtenemos dos caminos, si ese primer elemento del subconjunto es mayor que la suma buscada simplemente acaba esa recursión y pasa al siguiente número del vector. Como último caso, el primer número del subconjunto es menor que la suma buscada, entonces llamamos a *SumaPosic*() pasándole como parámetros el primer elemento del subconjunto, el valor de ese elemento como suma parcial, el vector principal y la suma buscada.

- *sumaPosic*():

Recibe como parámetros la primera posición del subconjunto, la posición variable la cual evaluamos en esta iteración, la suma parcial de los elementos del subconjunto, el vector principal y la suma buscada.

Utilizando la primera posición recibida desde el método *buscar*(), este método recursivo comprueba posición por posición si la suma de la combinación pasada al mismo es menor, igual o mayor que el número buscado. Para ello realiza las mismas comprobaciones que *buscar*() con la diferencia de que, en caso de que la suma sea menor que el número buscado, se llamará a sí mismo cambiando la posición variable a la siguiente posición del vector.

- *imprimir*():

Recibe como parámetro el vector principal.

Contiene una variable booleana iniciada a *true*, solo usada para la estética del fichero. Recorre el vector de soluciones con un *for* imprimiendo aquellos elementos del vector principal que tengan el valor *true* en el vector de soluciones.

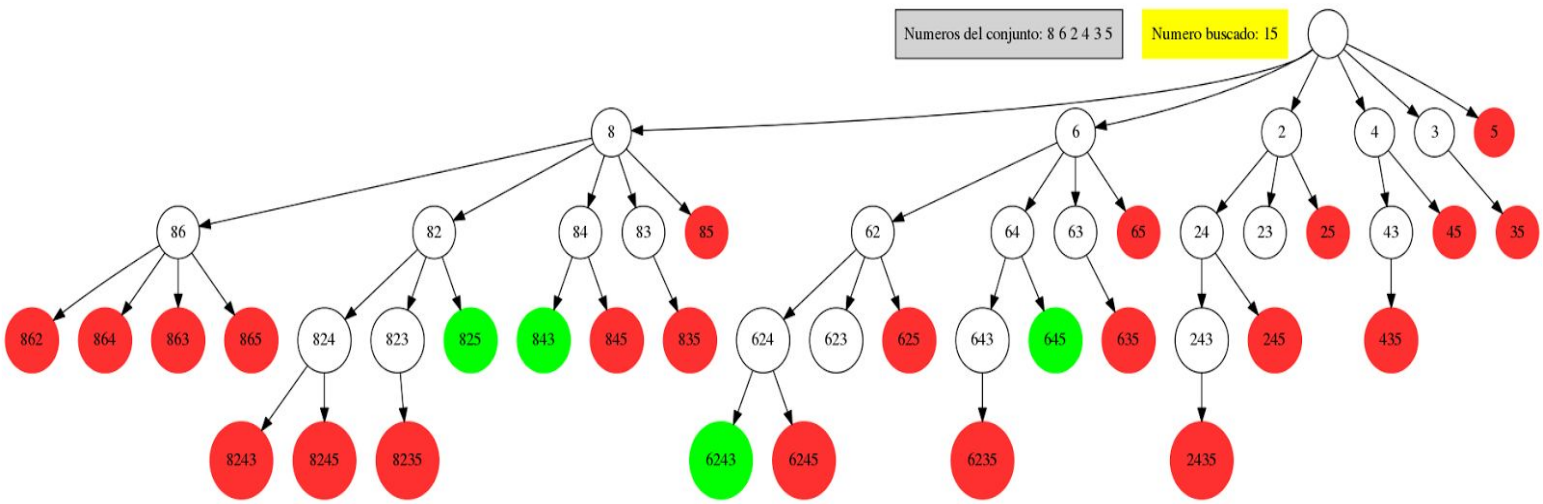
2.4. Estructuras de datos utilizadas:

Para nuestro programa hemos usado un *ArrayList* de enteros (*princ*) para introducir uno a uno los números que iba eligiendo el usuario, seleccionando cada una de sus entradas y pasando la posición como parámetro en los distintos métodos que lo necesitaban.

También tenemos como variable global un vector de booleanos soluciones el cual toma todas las posiciones del vector principal, siendo valor *true* si forman parte de un subconjunto de soluciones.

Finalmente usamos un fichero para sacar las soluciones del algoritmo.

3. EXPLICACIÓN DEL ALGORITMO



Para la explicación del algoritmo la forma más sencilla de explicarlo sería mediante un ejemplo, por lo tanto cogeremos este ejemplo propuesto en el enunciado de la práctica.

Recorreremos este árbol con el algoritmo de *búsqueda en profundidad* (ver detalles en las referencias).

Empezamos por el subelemento 8:

1. El programa comprobará si ese 8 es el número buscado (en este ejemplo, 15).
2. Como no es así pasaría a comprobar si es mayor, que tampoco es el caso.
3. Si no se ha dado ninguno de los casos anteriores el programa cogerá el siguiente elemento del vector y lo sumará (el nudo 86 quiere decir 8+6) y así sucesivamente hasta que se dé el caso 1 o el caso 2.

Cada vez que se produce el caso 1 o el caso 2 se realiza un backtracking, es decir, se vuelve al nodo anterior para continuar por otra rama (en el caso 1 el nodo se guardaría en el vector de soluciones y en el caso 2 no) y así sucesivamente.

Tras ver cómo funciona nuestro algoritmo podemos sacar los siguientes **casos base**:

a) Se encuentra el número buscado
b) Se sobrepasa el número buscado
c) Se termina de recorrer todo el vector para ese subconjunto

En el resto de casos, lo que llamamos **caso general** o **caso recursivo**, el número buscado es mayor aún que la suma parcial y volvemos a realizar el algoritmo añadiendo el siguiente número.

3.1 Explicación de porqué usamos backtracking

En cuanto planteamos la práctica propusimos rápidamente una solución al ejemplo dado en ella, nos dimos cuenta de que una forma muy eficiente podría ser el backtracking, ir cogiendo cada número y formar con él todas las combinaciones posibles, así paso a paso hasta conseguir tener todas las soluciones posibles para el conjunto de números introducido.

Además, usando backtracking en cuanto comprobamos que un número ya es igual al número buscado o es mayor que éste corta esa iteración y pasa a la siguiente automáticamente, haciendo que el algoritmo sea muy eficiente.

4. BATERÍA DE PRUEBAS

A continuación realizaremos unas cuantas pruebas en nuestro programa para comprobar su buen funcionamiento o que el programa se acaba cuando debe acabarse en caso de introducir algún dato erróneo.

1. Secuencia normal sencilla:
Números: 1 2 3 4 5 Número buscado: 10

Salida obtenida en el fichero:

1, 2, 3, 4

1, 4, 5

2, 3, 5

El primer ejemplo es una secuencia sencilla en la cual el fichero devuelve una serie de subconjuntos solución de forma correcta.

2. Poniendo un número buscado erróneo, en este caso negativo:
Números: 1 2 3 4 5 Número buscado: -10

Salida obtenida en el fichero: (Vacío)

En este ejemplo hemos introducido un número buscado erróneo, por lo que el programa no encuentra ningún número. El programa finaliza y el fichero queda vacío.

3. Poniendo un numero del subconjunto erróneo, en este caso negativo:
Números: 1 2 3 4 5 -1 Número buscado: Ni siquiera deja introducir la suma por introducir un número negativo

Salida obtenida en el fichero: (Vacío)

Como podemos comprobar, en cuanto introducimos un dato erróneo en la entrada de números el programa acaba y el fichero queda vacío.

4. Secuencia sencilla en desorden:

Números: 4 2 1 3 5 Número buscado: 10

Salida obtenida en el fichero:

4, 2, 1, 3

4, 1, 5

2, 3, 5

Como podemos comprobar, el resultado es el mismo que en el ejemplo 1 pero influye el orden en el que nosotros introducimos los datos.

5. Secuencia sencilla en orden inverso:

Números: 5 4 3 2 1 Número buscado: 10

Salida obtenida en el fichero:

5, 4, 1

5, 3, 2

4, 3, 2, 1

Exactamente igual que el anterior y el ejemplo 1, la salida toma el orden introducido.

6. Secuencia sencilla con número(s) repetido(s):

Números: 1 1 2 3 4 5 Número buscado: 10

Salida obtenida en el fichero:

1, 1, 3, 5

1, 2, 3, 4

1, 4, 5

1, 2, 3, 4

1, 4, 5

2, 3, 5

En este caso toma cada número repetido como otro número independiente, es decir, aunque el 1 esté repetido dos veces el programa lo toma como si tuviese identidades distintas, consiguiendo así un mayor número de relaciones posibles.

7. Secuencia más compleja:

Números: 8 9 5 6 4 3 7 1 2 10 Número buscado: 18

Salida obtenida en el fichero:

8, 9, 1

8, 5, 4, 1

8, 5, 3, 2

8, 6, 4

8, 6, 3, 1

8, 4, 3, 1, 2

8, 3, 7

8, 7, 1, 2

8, 10

9, 5, 4

9, 5, 3, 1

9, 6, 3

9, 6, 1, 2

9, 4, 3, 2

9, 7, 2

5, 6, 4, 3

5, 6, 4, 1, 2

5, 6, 7

5, 4, 7, 2

5, 3, 7, 1, 2

5, 3, 10

5, 1, 2, 10

6, 4, 7, 1

6, 3, 7, 2

6, 2, 10

4, 3, 1, 10

7, 1, 1

Como podemos observar a medida que aumentamos un poco los números de entrada el tamaño de la salida aumenta exponencialmente, pero de esta forma se visualiza perfectamente el backtracking utilizado.

8. Lista de números vacía:

Números: ninguno. Número buscado: 5

Salida obtenida en el fichero: (vacío).

Este caso está cubierto por el caso base en que se acaba de recorrer la lista: puesto que no hay números que comprobar, el programa actúa como al terminar la lista y termina su ejecución.

5. REFERENCIAS

[1] Búsqueda en profundidad. (2016, 31 de julio). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 18:14, noviembre 9, 2016 desde https://es.wikipedia.org/w/index.php?title=B%C3%BAsqueda_en_profundidad&oldid=92624038

[2] Cesar Gonzalez Ferreras(Dpto. Informática), “Tema 2 - Diseño de Algoritmos”, Curso 2012/13 desde https://aulas.inf.uva.es/pluginfile.php/27848/mod_resource/content/1/tema2.pdf