BACKTRACKING

CALCULAR TODAS LAS POSIBILIDADES

Hay problemas para los que no se conoce un algoritmo para su resolución o al menos, no cuentan con un algoritmo eficiente para calcular su solución. En estos casos, lo que nos queda es recurrir a una exploración directa de todas las posibilidades. Por ejemplo, recuerda cuando calculamos si un número *n* era primo, la primera solución era pura fuerza bruta, mirando desde 2 a n-1 si encontrábamos algún divisor para delatar a "n" como no primo. Luego fuimos generando un algoritmo más eficiente que cada vez exploraba menos posibilidades gracias a aplicar determinadas relaciones matemáticas que existen entre los números enteros.

CALCULAR TODAS LAS POSIBILIDADES COMBINANDO VARIOS VALORES Y SU REPRESENTACIÓN EN UN ÁRBOL DE POSIBILIDADES.

Cuando el cálculo de todas las posibilidades consiste en combinar valores realmente estamos generando "un árbol de posibilidades" que refleja la combinatoria. No es un árbol físico como el árbol de nodos que vimos anteriormente pero el árbol "está ahí" en base a cómo se exploran todas las posibilidades.

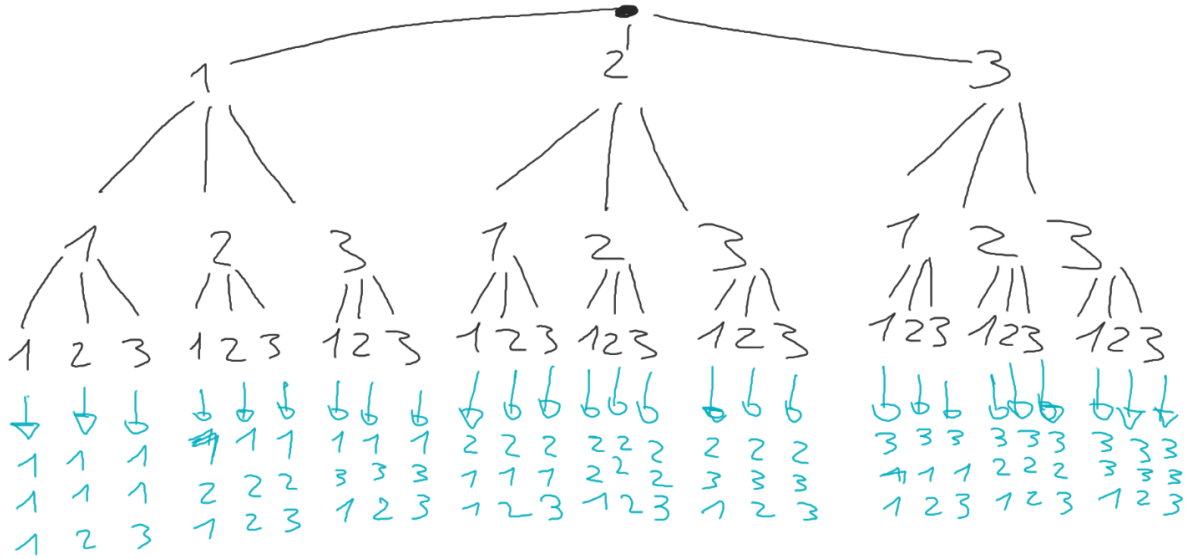
Recuerda el siguiente ejemplo para generar todas las combinaciones de longitud 3 para las cifras 1, 2 Y 3.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class App{


  static List<String> combinar(int longitud, List<String> numeros){
     List<String> result= new ArrayList<>();
     if(longitud==1){
        return numeros;
     }
     List<String> sublista= combinar(longitud-1,numeros);

     for(String numero:numeros){
        for(String numeroCombinado:sublista){
           result.add(numero+numeroCombinado);
        }
     }
     return result;
  }
  public static void main(String[] args) {
     List<String> numeros=Arrays.asList("1","2","3");

     System.out.println("todas las combinaciones de  1, 2 y 3 con longitud 3");
     int longitud=3;
     List<String> result=combinar(longitud,numeros);
     for(String s:result){
        System.out.println(s);
     }



  }

}
```

SALIDA:

todas las combinaciones de 1, 2 y 3 con longitud 3
111
112
113
121
122
123
131
132
133
211
212
213
221
222
223
231
232
233
311
312
313
321
322
323
331
332
333

OBSERVA SU CORRESPONDIENTE ÁRBOL DE POSIBILIDADES



PODA DEL ÁRBOL DE POSIBILIDADES.

El cálculo exhaustivo de todas las posibilidades se puede mejorar (acortar) cuando a priori sabemos que la combinación de determinados valores no conducen a una solución. Gráficamente decimos que podemos "podar" el árbol de posibilidades.

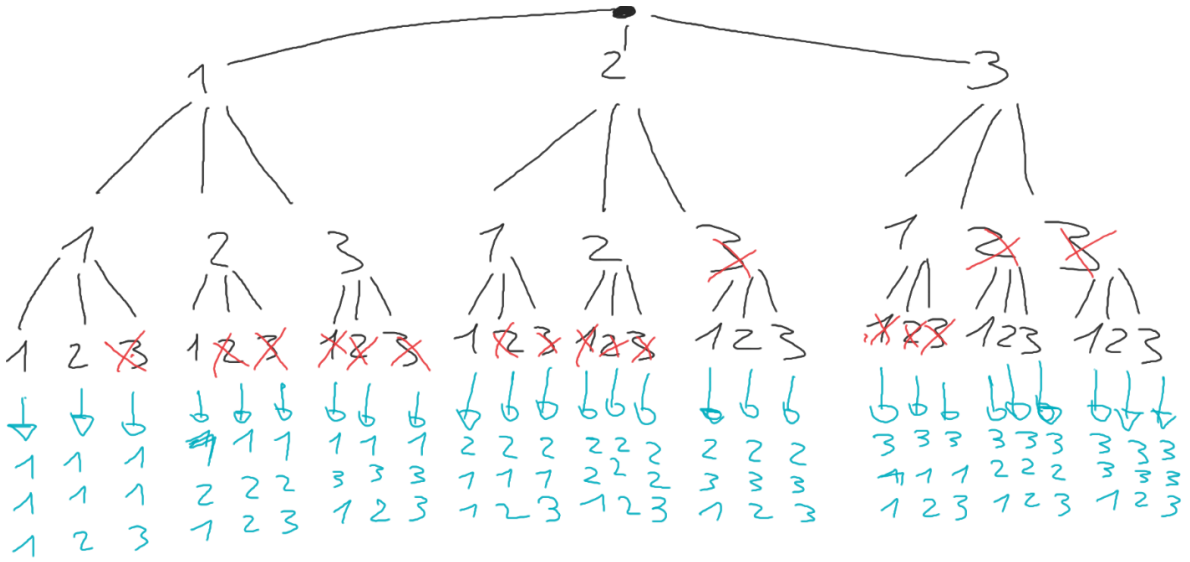
EJEMPLO: "PODAMOS" si la suma parcial o total de las cifras es > 5

```
import java.util.ArrayList;
```

```
import java.util.Arrays;
import java.util.List;
public class App{
 static int sumar(String cifra, String cifras){
    //llega a cifra un String que contiene una cifra y a cifras un String con varias cifras
    //por ejemplo "1"    y "31"
    int suma=Integer.parseInt(cifra);
    for(int i=0;i<cifras.length();i++){
       suma+=Integer.parseInt(""+cifras.charAt(i));
    }
    return suma;
 }
 static List<String> combinar(int longitud, List<String> numeros){
    List<String> result= new ArrayList<>();
    if(longitud==1){
       return numeros;
    }
    List<String> sublista= combinar(longitud-1,numeros);
     for(String numero:numeros){

       for(String numeroCombinado:sublista){
          //podar cuando suma más de 5
          if(sumar(numero,numeroCombinado)>=5){
             continue;//elimina llamada recursiva
          }
          result.add(numero+numeroCombinado);
       }
    }
    return result;
 }
 public static void main(String[] args) {
    List<String> numeros=Arrays.asList("1","2","3");

    System.out.println("combinaciones de  1, 2 y 3 con longitud 3 y poda si suma >= 5" );
    int longitud=3;
    List<String> result=combinar(longitud,numeros);
    for(String s:result){
       System.out.println(s);
    }



 }
}
```

```
combinaciones de  1, 2 y 3 con longitud 3 y poda si suma >= 5
111
112
121
211
```

Ten en cuenta que haber generado todas las combinaciones, y luego recorrer todas las combinaciones mirando si su suma es > 5 sería ineficiente
OBSERVA EL EFECTO DE LA PODA EN EL ÁRBOL DE SOLUCIONES



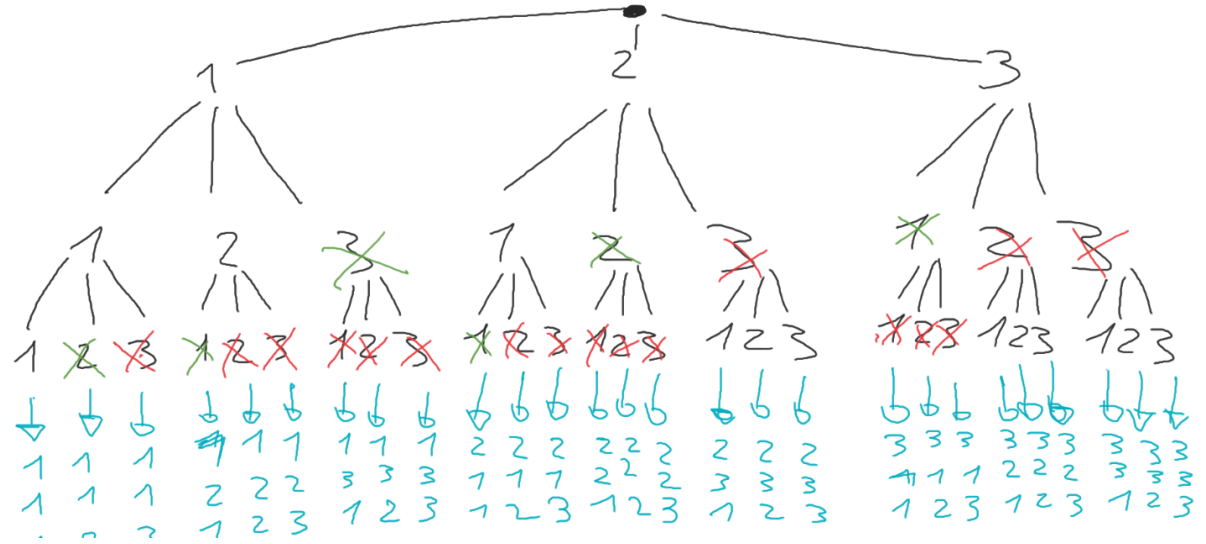
Observa en

```
for(String numeroCombinado:sublista){

            if(sumar(numero,numeroCombinado)>=5){
                continue;
            }
            result.add(numero+numeroCombinado);
        }
```

que el continue implica que no se hace el add, con lo cual esa combinación no se incorpora al resultado, que por recursividad

Es un árbol pequeño y poco profundo pero en un ejemplo que implique la combinatoria de muchos valores la poda aporta una gran mejoría.

si podamos para >=4 en verde vemos las nuevas podas que no había con >=5



QUE ES BACKTRACKING

El ejemplo anterior realmente utiliza backtracking. Veámos ahora un poco más formalmente que es backtracking.

Es una técnica de búsqueda de soluciones con las siguientes características:

- En el problema a resolver hay una serie de variables que pueden tomar diversos valores. Una o varias de las combinaciones de dichos valores es la solución. Ejemplo: camino solución de un laberinto. La solución no tiene porque ser única puede haber varias combinaciones que son solución. Ejemplo: un laberinto con varios caminos solución

- La base de esta técnica es la fuerza bruta que genera todas las combinaciones posibles. Las combinaciones posibles se pueden representar como un árbol de posibilidades. A dicho árbol se les suele incorporar "poda" para evitar calcular combinaciones que ya se sabe a priori que no conducen a la solución.
- La generación del árbol de posibilidades se suele hacer con recursividad lo que genera el típico "recorrido/generación en profundidad" de un árbol como el típico pre-orden
- Se llama backtracking "retroceso" por su forma de avanzar en el árbol de posibilidades ya que si llega a un nodo a partir del que al usar una función de poda se sabe que no hay solución, retrocede al padre de este nodo y sigue desde ahí explorando posibilidades. Pero realmente la técnica se refiere en su conjunto a todas las características enunciadas en su conjunto: búsqueda exhaustiva+poda+recorrido en profundidad.
- Se usa backtracking en muchos ámbitos de la programación: cálculo de expresiones regulares, tareas de reconocimiento de texto, sintaxis de lenguajes regulares, juegos, IA, …

EJERCICIO: LABERINTO CON TODOS LOS CAMINOS POSIBLES

En la primera evaluación y sólo manejando Strings y una matriz escribimos un algoritmo de laberinto "El ratón y el queso". Este algoritmo termina al encontrar la primera solución. Queremos ahora modificar ese algoritmo de forma que me devuelva todas las soluciones posibles.

Si no quiero emplear estructuras de almacenamiento para almacenar soluciones de forma que el código sea más simple lo que puedo hacer es:

1. No parar después de encontrar la primera solución, seguir buscando hasta agotar combinaciones con posibilidades de ser solución.
2. Cada vez que llegue al queso, imprimo la solución.

Podemos seguir el siguiente pseudocódigo:

```
buscarSolucion(char[][] laberinto, int i, int j, String sol): //ahora al avanzar por un camino se va grabando la solución en un String
    nueva coordenada a estudiar i,j
    Se para(return) cuando:
        - i y/o j fuera de limites
```

- hay pared #
- llegue a una celda en exploración que marco con ".", para evitar ciclos
- se llega al queso y entonces antes de hacer return se imprime string

```
    si llego aquí no pasé por un return y sigo explorando
    laberinto[i][j]='.';//  marco esta celda en  exploración, para  evitar  ciclos

     sol=sol+coordenada;

     //miramos caminos a través de sus posibles 4 vecinos
     //NOVEDAD IMPORTANTE ¡sin return!, ya que si no sólo habría una solución
      buscarSolucion(laberinto,i-1,j,sol);//por el norte
      buscarSolucion(laberinto,i+1,j,sol); //por el  sur
      buscarSolucion(laberinto,i,j+1,sol); //por el  este
      buscarSolucion(laberinto,i,j-1,sol);//por el  oeste


      //NOVEDAD IMPORTANTE se cambia de "." en exploración a de nuevo libre para explorar"0"
      //esto permite que quizá esta celda forme parte de otras soluciones
      laberinto[i][j]='0';
      return;
   }
```

De tal forma que

```
public static void main(String[] args) {
      char[][] laberinto = {
            { 'R', '0', '0' },
            { '#', '0', '0' },
            { '#', 'Q', '#' }

      };
      String sol = "";
      buscarSolucion(laberinto, 0, 0, sol);
   }
```

Origina la siguiente salida

```
(0,0)(0,1)(1,1)(2,1)
(0,0)(0,1)(0,2)(1,2)(1,1)(2,1)
```

Observa cómo se imprime más de una solución y como hay celdas que forma parte de varias soluciones.

Recuerda que para hacer breve y sencillo el algoritmo imprimimos un String solución cada vez que llegamos al queso pero no tiene porque hacerse así necesariamente.

Este ejercicio lo tienes disponible para probar en coderunner
Una vez que hagas el código, con la salida anterior está bien que dibujes el árbol solución con boli y papel