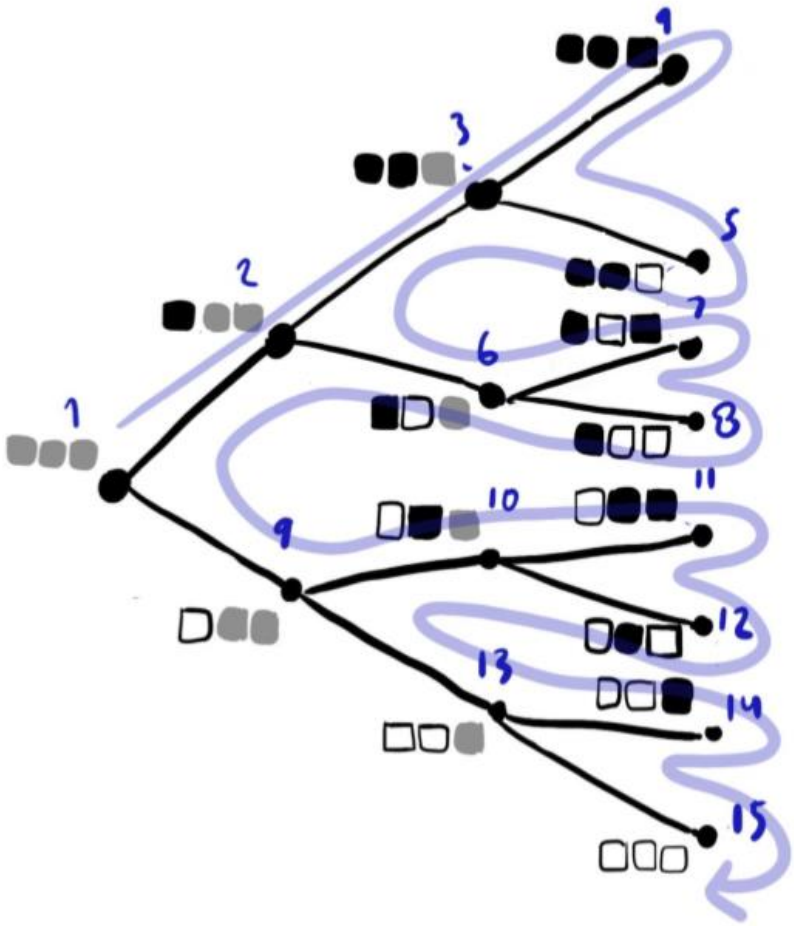
Recursive Backtrack Lab

This lab follows the structure of the lecture closely. The backtracking itself has already been implemented, but it will ask of you to fill in the three functions required to implement subset, permutation and path enumeration.

Refresher on backtracking

Recall that recursive backtracking works by tentatively extending a partial solution array with all promising candidate values until it forms a complete solution. Here, the array is initially empty (not a complete solution), so it considers its two options for the first slot (black and white), tries with black, and proceeds to test again.



There are many ways of implementing it. One way, as seen in the Pythonesque pseudo-code below, is to re-use the same array for every recursive call (an alternative way is to make a copy of the array for each recursive call).

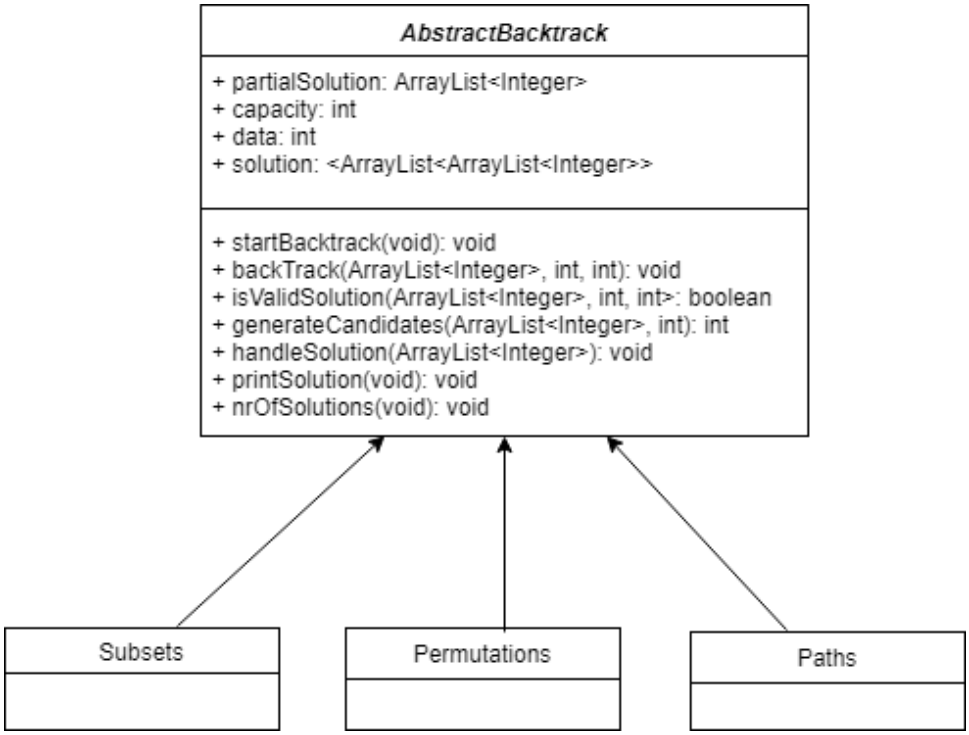
```
def backtrack( array, index):
      if completeSolution(array):
            handleSolution(array)
            return
      else:
            index += 1
            candidates = generateCandidates(A,index)
            for candidate in candidates:
                  array[index] = candidate
                  backtrack(array, index)
            return
```

Recall that the backtracking pattern has three replaceable components that can be adapted to output - for a set of consecutive integers - the possible subsets and permutations. For a graph, it can also be used to output all possible paths from a starting node to a destination node.

Problem	handleSolution	completeSolution	generateCandidates
Subsets	Print out	Are all slots filled?	{0,1}
Permutations	Print out	Have all objects been assigned?	The objects not already assigned
All paths to t	Print out	Is the last slot's node the target node?	The neighbours not already in path

The exercise

The backtracking algorithm has already been implemented in the abstract class **AbstractBacktrack** and the task is to implement the abstract methods **isValidSolution(), handleSolution()** and **generateCandidates()** for the concrete subclasses **Subsets**, **Permutations** and **Paths**.



Inside the **Main** class you can run the different problems through command line arguments. Args[0] can be "subsets" or "permutations" or "paths". For example:

- `Java Main subsets 3` should print out all subsets of integers [0,1,2]
- `Java Main permutations 5` should print out all permutations of [0,1,2,3,4]
- `Java Main paths 5` should print out all paths from node 1 to 5.

Note that for paths, I have hardcoded an undirected graph (below) using a map datastructure, Hashmap<Integer, ArrayList<Integer>>. Nodes are keys, neighbours are the value. Also note that nodes start at 1, not 0. Target node is specified via args[1]. So, for example, map.get(1) would return [6,3,2].

You will find more hints in the java files themselves. For guidance beyond lecture materials, I recommend "The Algorithm Design Manual" by Skiena.

Good luck! =)