Recursion

What is recursion

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration).

How to solve a problem

- Divide into small problems.
- Solve small problems.
- Use the result of small problems to solve the original problem.
- If the small problems are the same as original one, just scale is different. Then this is called recursion.

Recursion Properties

- Base case
  - simple scenario that does not need recursion to produce an answer.
- Recursion
  - a set of rules that reduce all other case toward the base case.

Tail Recursion

Tail recursion is a special kind of recursion where the recursive call is the ***very last*** thing in the function. It's a function that does not do anything at all after recursing.

```
int Factorial(int n) {
    if(n == 1) return 1;
    return n * Factorial(n-1);
}
```

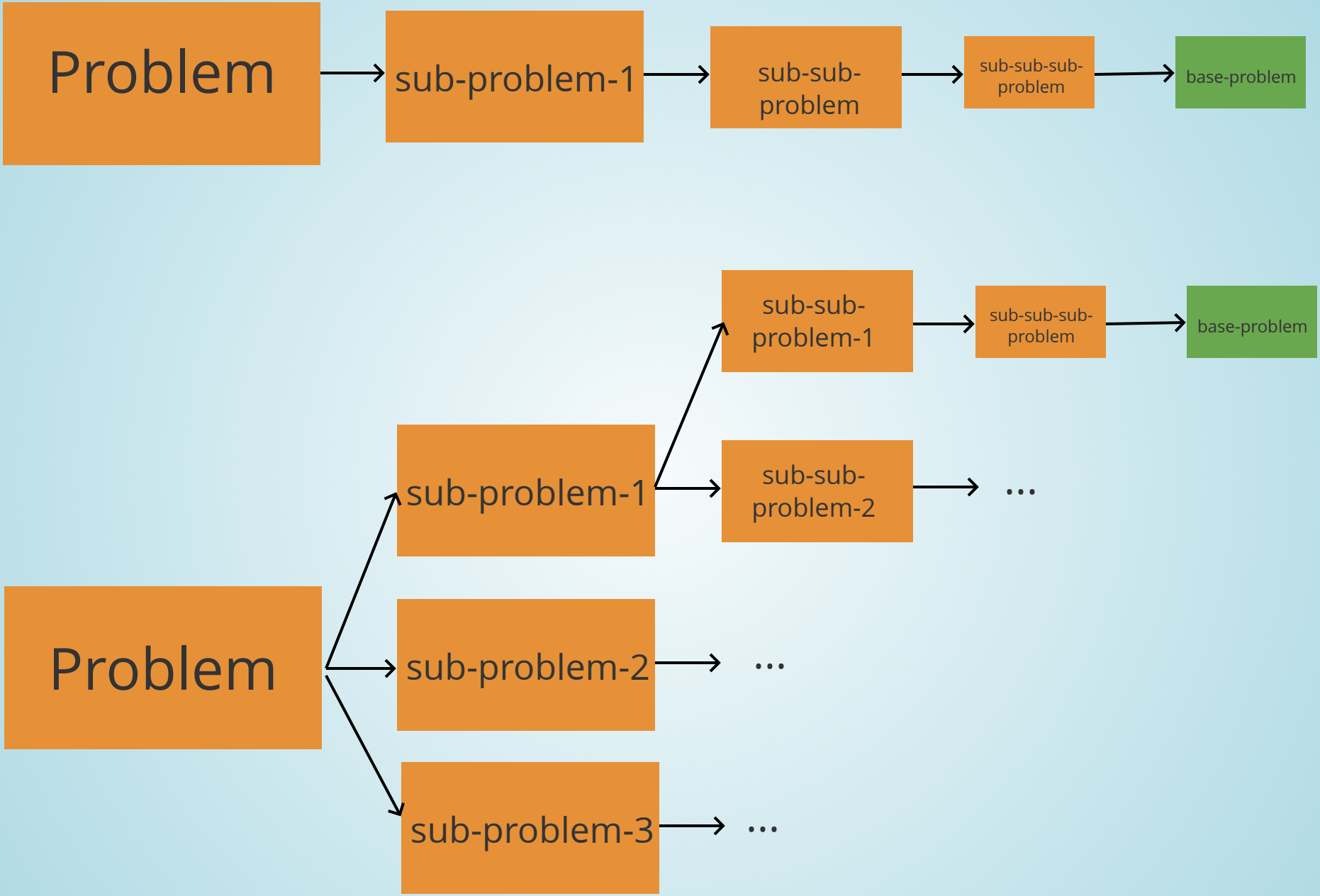
```
int Factorial(int n) {
    int sum = 1;
    for(int i = 1; i <= n; i ++) {
        sum = sum * i;
    }
    return sum;
}
```

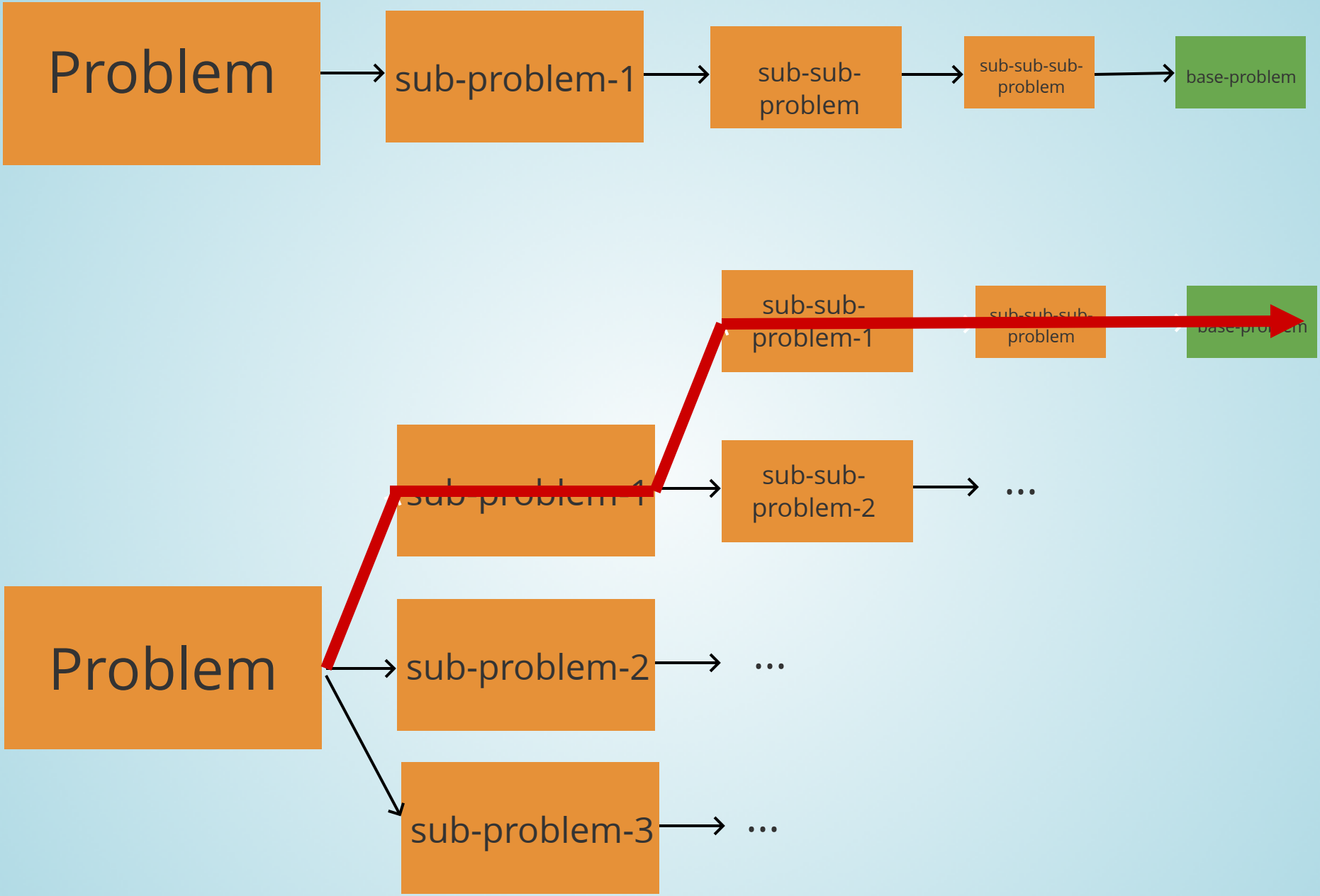
Code problem with recursion

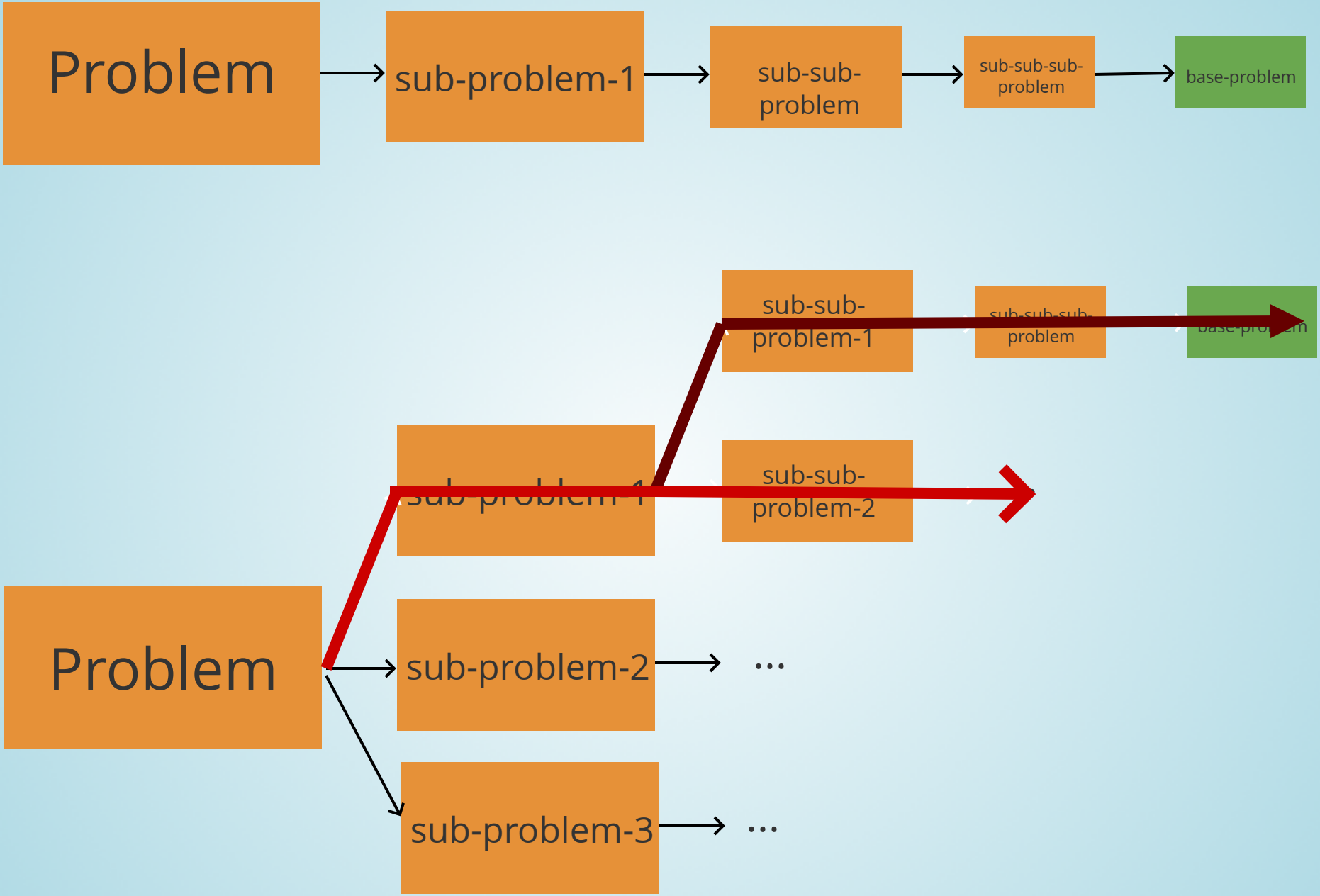
- Base case
- recursion rules
- **Represent the problem with coding function.**
  - **Define the essential parameters**
    - **Parameters that define the problem.**
    - **Parameters that store the temporary result or state.**
  - **Define the return value**

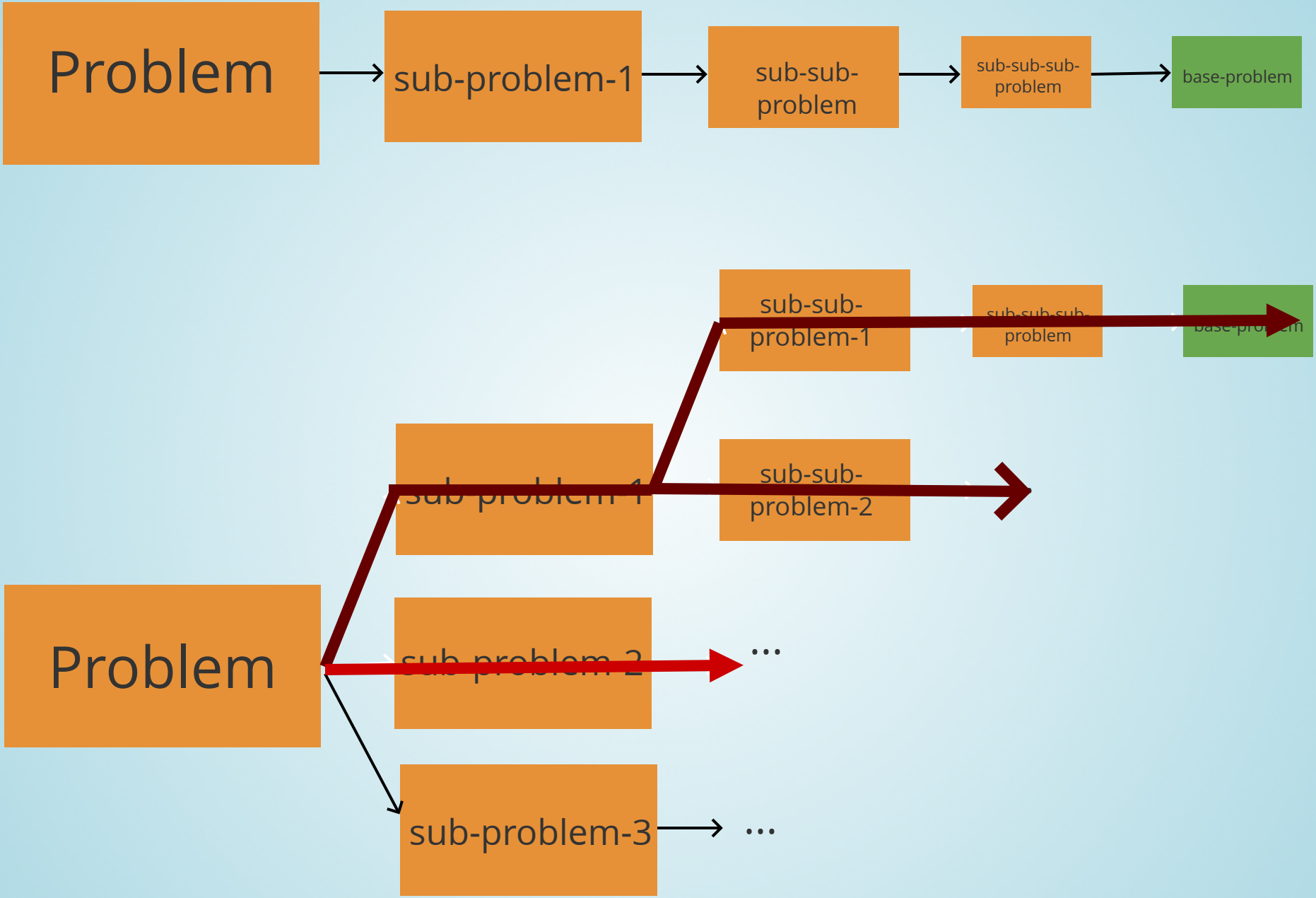
Classical Recursion Problem

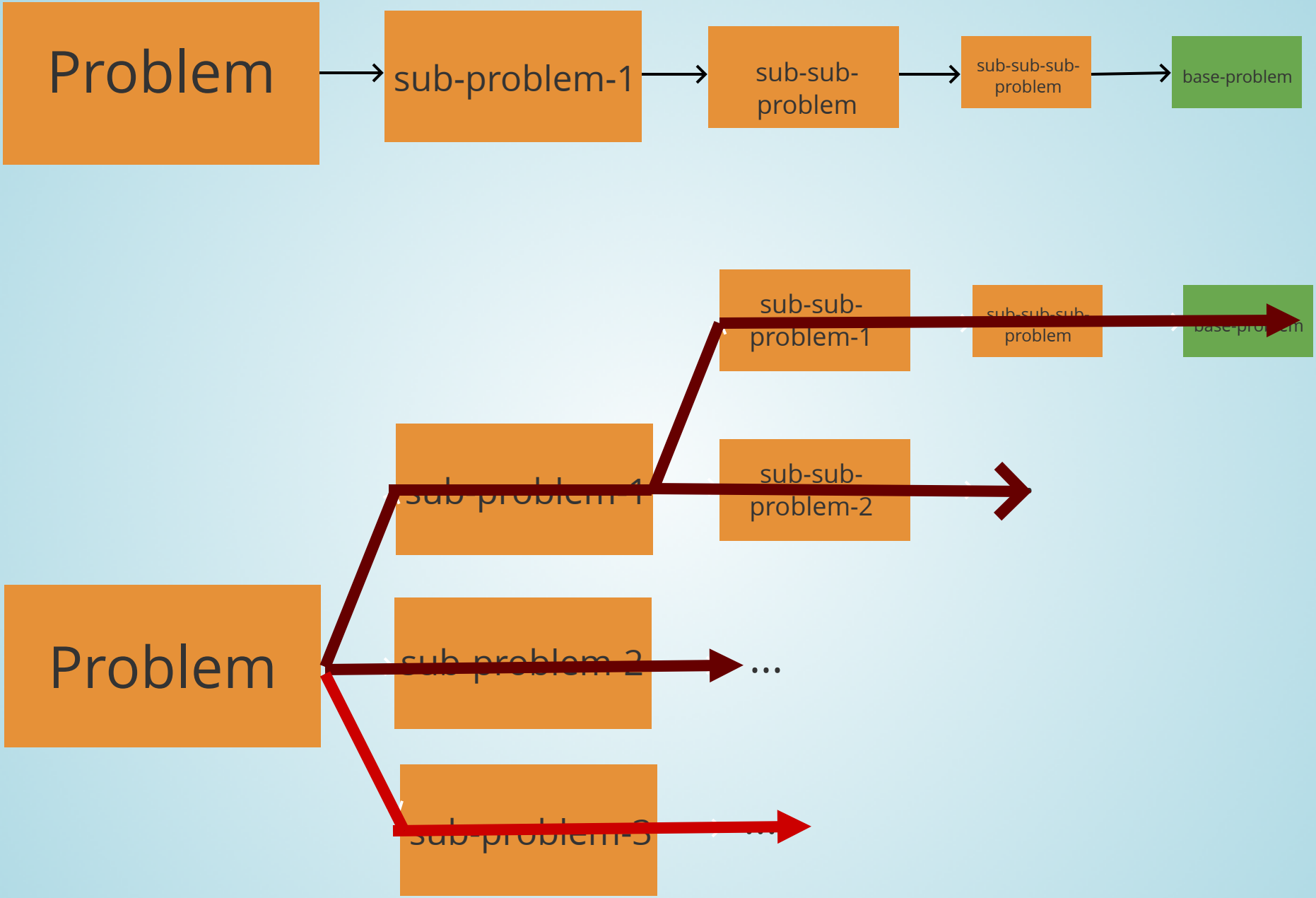
- Fibonacci Number
  - Can be solved by DP, which is better
- Climbing Stairs
  - Same as Fibonacci
- Merge Sort
  - We will talk more about that in Sort
- Towers of Hanoi
- Binary Search
  - This is a tail Recursion

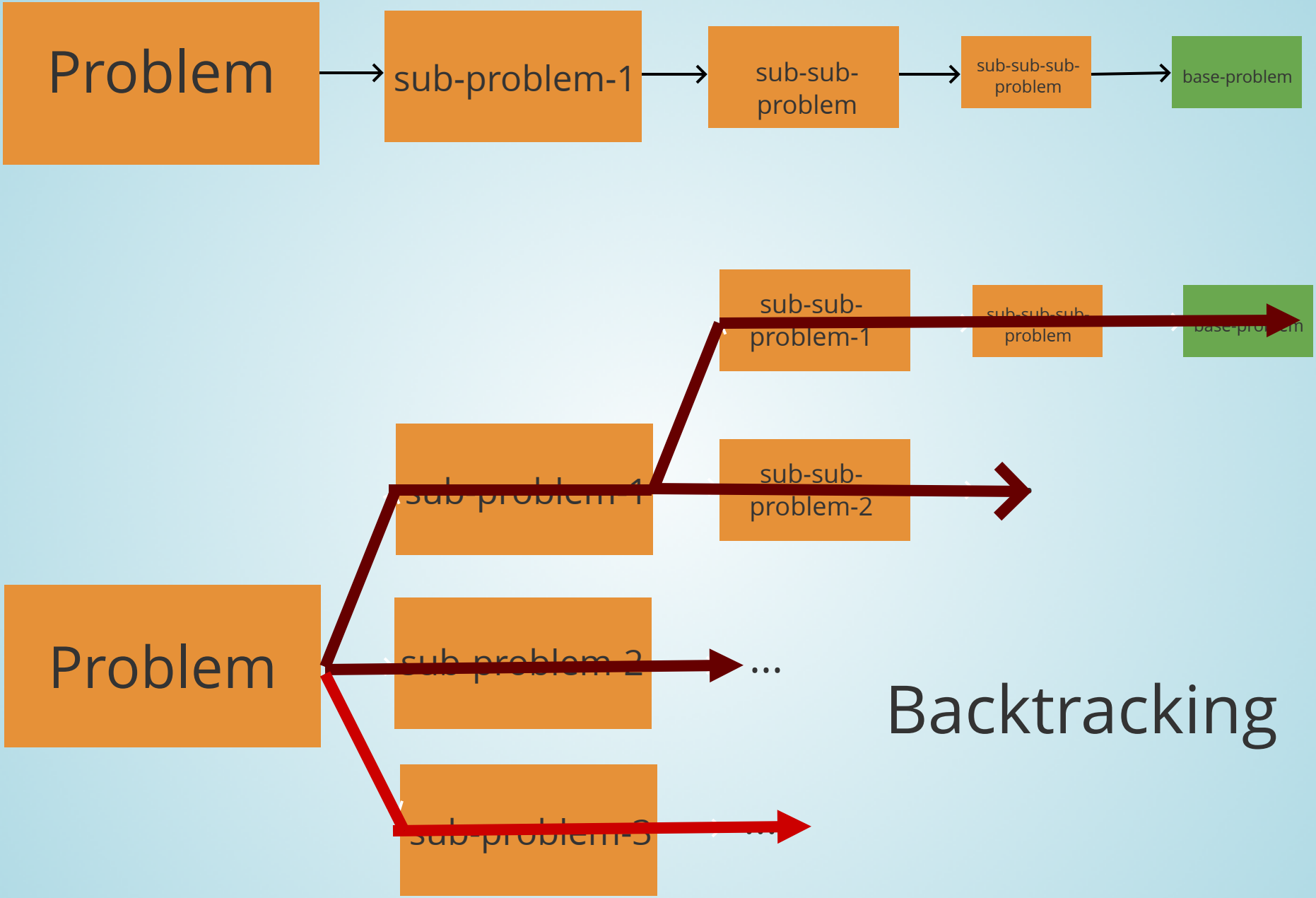












Gray Code

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given n = 2, return [0,1,3,2]

This is a typical problem with only one sub-problem

You need to find the pattern inside of it.

Gray Code

The key is to find the relationship between each sub-problem

```
public List<Integer> grayCode(int n) {
    List<Integer> result = new ArrayList<>();
    helper(n, result);
    return result;
}
public void helper(int n, List<Integer> result) {
    if (n == 0) {
       result.add(0);
       return;
    }
    helper(n-1, result);
    int size = result.size();
    int k = 1 << (n - 1);
    for (int i = size - 1; i >= 0; i --) {
        result.add(result.get(i) + k);
    }
    return;
}
```

Gray Code

And this is actually also a Tail Recursion. We actually could get rid of the recursion code.

```
public List<Integer> grayCode(int n) {
    List<Integer> result = new ArrayList<>();
    result.add(0);
    for(int i = 0; i < n; i ++) {
        int k = 1 << i;
        int size = result.size();
        for(int j = size - 1; j >= 0; j --) {
            result.add(result.get(j) + k);
        }
    }
    return result;
}
```

0-1 Knapsack

Given a knapsack which can hold ***s*** pounds of items, and a set of items with weight ***w1, w2, ... wn***. Return whether we can pick specific items so that their total weight ***s.***

Example Input:

s = 20;

w = [14, 8, 7, 5, 3];

Example Output:

True;

Given an item, just two options:

- pick it, the problem become (s-w[i], w - w[i])
- not pick it, the problem become (s, w - w[i])

0-1 Knapsack

Given a knapsack which can hold ***s*** pounds of items, and a set of items with weight ***w1, w2, ... wn***. Return whether we can pick specific items so that their total weight ***s.***

```
public static boolean knapsack(int s, int[] weights, int index) {
    if (s == 0) {
      return true;
    }
    if (s < 0 || index >= weights.length) {
      return false;
    }
    return knapsack(s - weights[index], weights, index+1) ||
      knapsack(s, weights, index+1);
}
```

Maze

Given a maze and a start point and a target point, return whether the target can be reached.

Example Input:

Start Point: (0, 0); Target Point (5, 5);

Maze: char[][] = {

```
  {'.', 'X', '.', '.', '.', 'X'},
  {'.', '.', '.', 'X', '.', 'X'},
  {'X', 'X', '.', 'X', '.', '.'},
  {'.', 'X', 'X', 'X', '.', 'X'},
  {'.', '.', '.', '.', '.', 'X'},
  {'.', '.', '.', '.', '.', '.'}
}
```

Example Output: True

Maze

Given a maze and a start point and a target point, return whether the target can be reached.

Example Input:

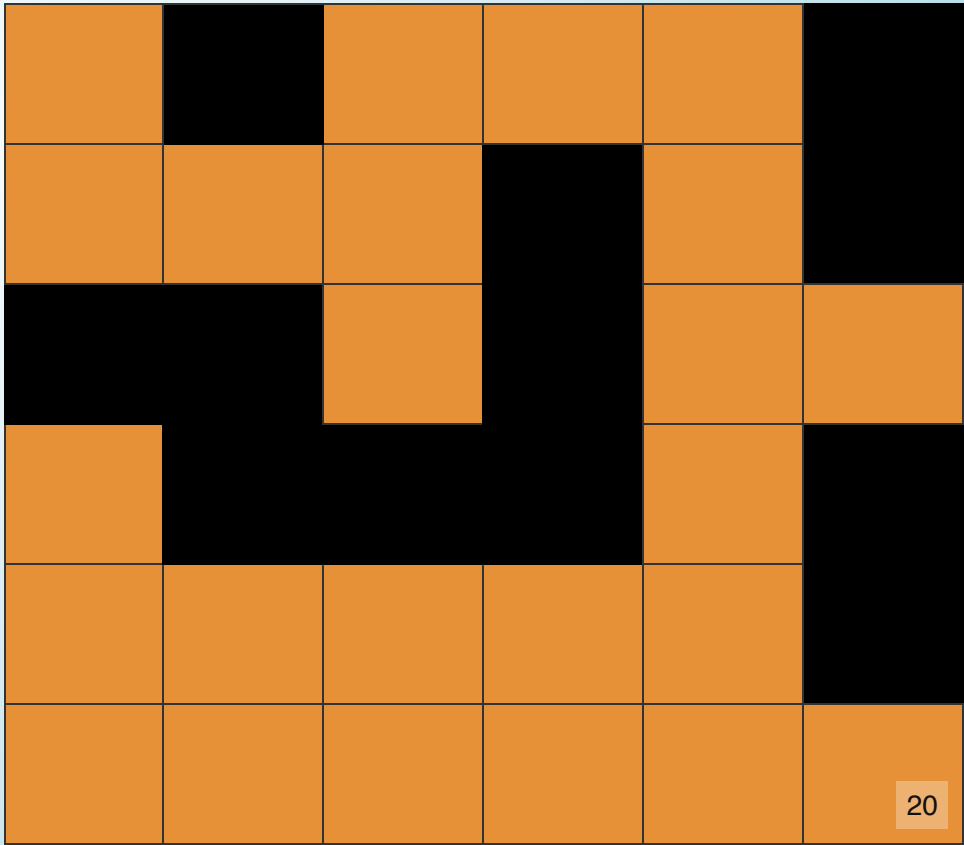
Start Point: (0, 0); Target Point (5, 5);

Maze: char[][] = {

```
  {'.', 'X', '.', '.', '.', 'X'},
  {'.', '.', '.', 'X', '.', 'X'},
  {'X', 'X', '.', 'X', '.', '.'},
  {'.', 'X', 'X', 'X', '.', 'X'},
  {'.', '.', '.', '.', '.', 'X'},
  {'.', '.', '.', '.', '.', '.'}
```

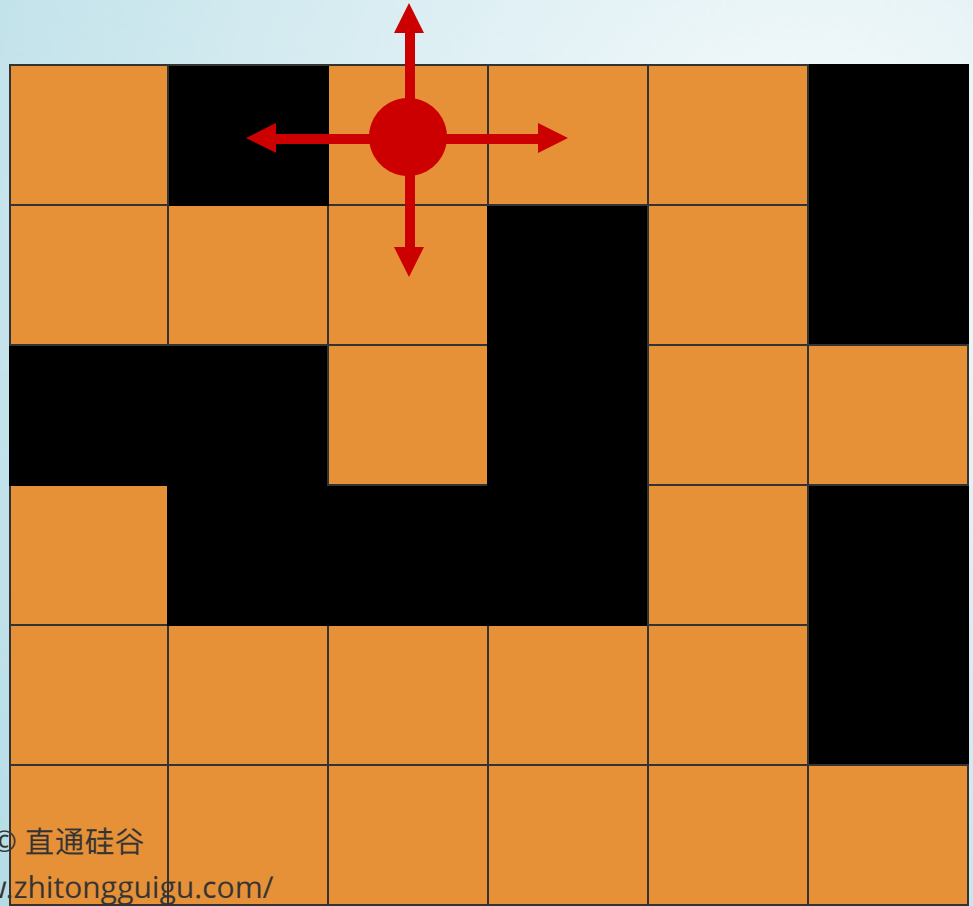
}

Example Output: True



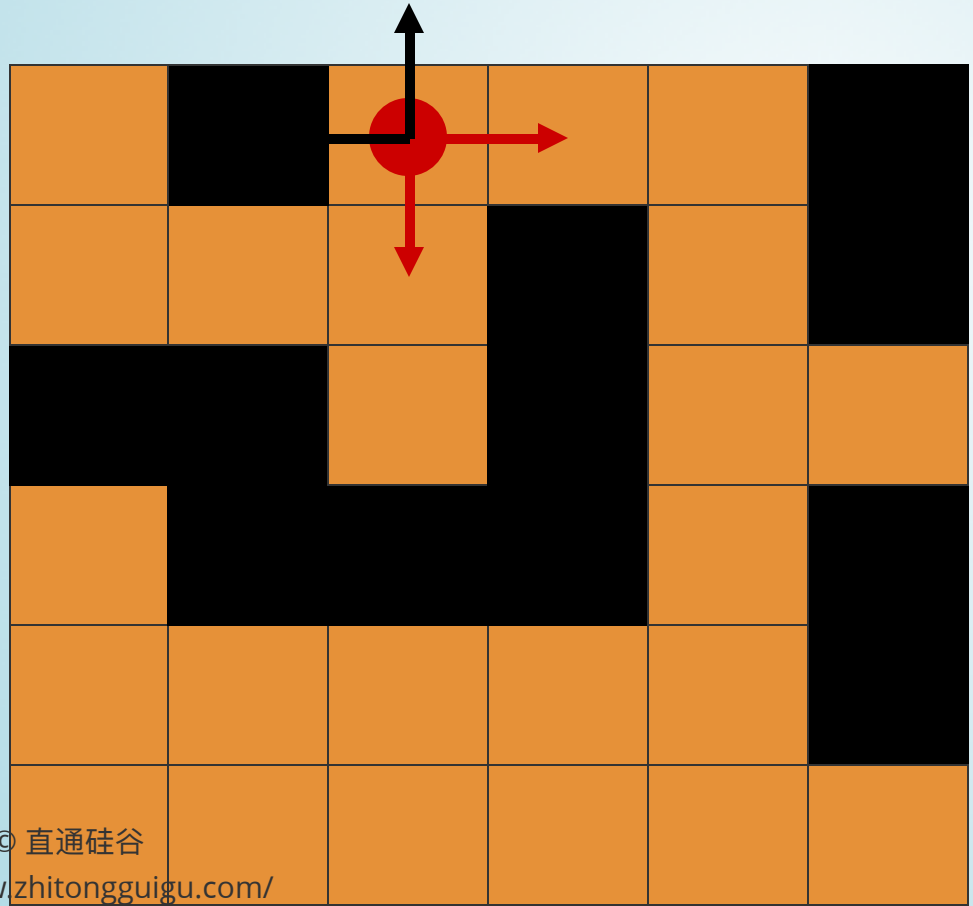
Maze

Given a maze and a start point and a target point, return whether the target can be reached.



Maze

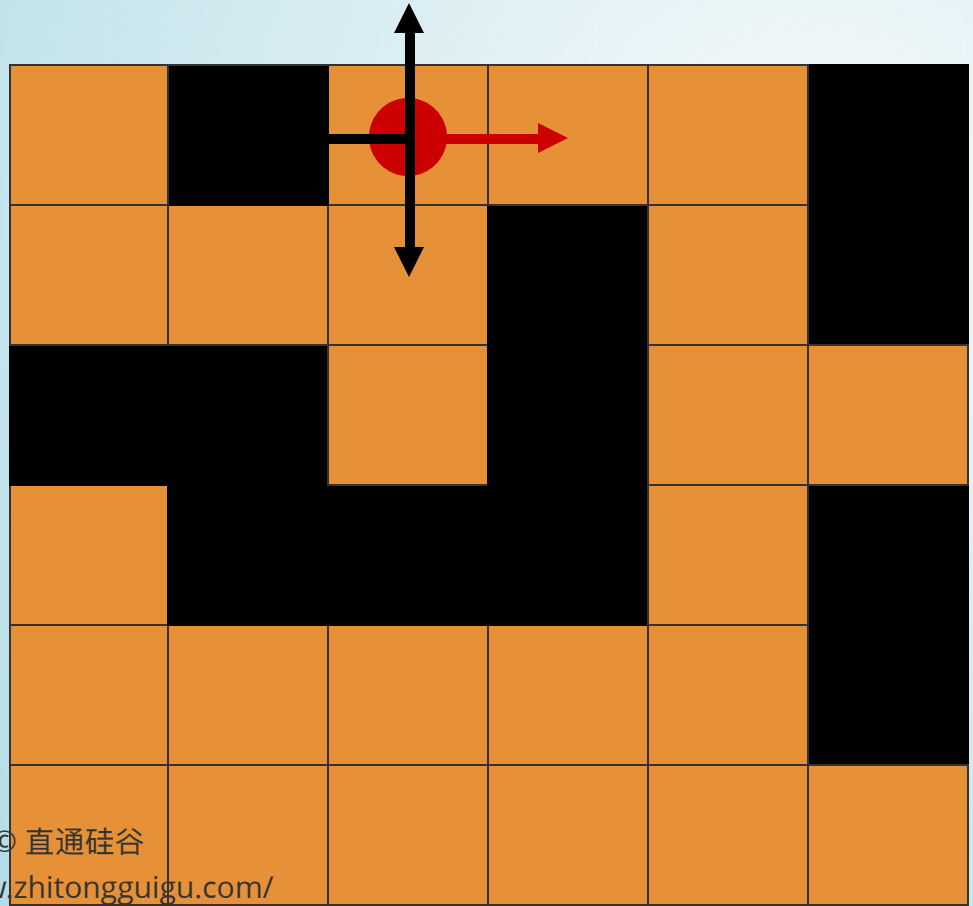
Given a maze and a start point and a target point, return whether the target can be reached.



- Out of Bound
- Wall

Maze

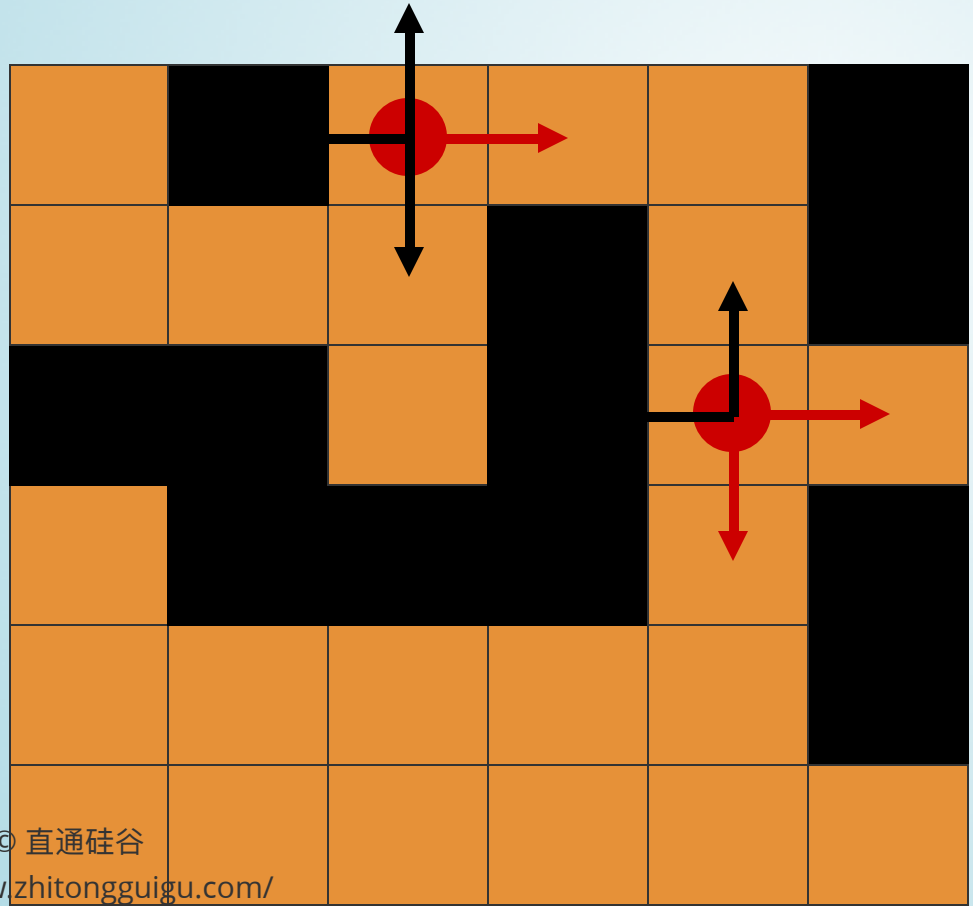
Given a maze and a start point and a target point, return whether the target can be reached.



- Out of Bound
- Wall
- Visited
  - (1, 2) -> (2, 2) -> (1, 2) -> (2, 2) -> ...

Maze

Given a maze and a start point and a target point, return whether the target can be reached.



Maze

Given a maze and a start point and a target point, return whether the target can be reached.

```
public static boolean solveMaze(char[][] maze,
                                int startX, int startY, int targetX, int targetY,
                                boolean[][] visited) {
    if (startX < 0 || startX >= maze.length ||
        startY < 0 || startY >= maze[0].length ||
        maze[startX][startY] == 'X' || visited[startX][startY]) {
      return false;
    }
    if (startX == targetX && startY == targetY) {
      return true;
    }
    visited[startX][startY] = true;
    if (solveMaze(maze, startX + 1, startY, targetX, targetY, visited) ||
        solveMaze(maze, startX, startY + 1, targetX, targetY, visited) ||
        solveMaze(maze, startX - 1, startY, targetX, targetY, visited) ||
        solveMaze(maze, startX, startY - 1, targetX, targetY, visited)) {
      return true;
    }
    return false;
}
```

Maze

Given a maze and a start point and a target point, return whether the target can be reached.

```
public static boolean solveMaze(char[][] maze,
                                int startX, int startY, int targetX, int targetY) {

    if (startX < 0 || startX >= maze.length ||
        startY < 0 || startY >= maze[0].length ||
        maze[startX][startY] == 'X') {
      return false;
    }
    if (startX == targetX && startY == targetY) {
      return true;
    }
    maze[startX][startY] = 'X';
    if (solveMaze(maze, startX + 1, startY, targetX, targetY) ||
        solveMaze(maze, startX, startY + 1, targetX, targetY) ||
        solveMaze(maze, startX - 1, startY, targetX, targetY) ||
        solveMaze(maze, startX, startY - 1, targetX, targetY)) {
      return true;
    }
    return false;
}
```

Maze

Given a maze and a start point and a target point, return whether the target can be reached.

```
public static boolean solveMaze(char[][] maze,
                                int startX, int startY, int targetX, int targetY) {

    if (startX < 0 || startX >= maze.length ||
        startY < 0 || startY >= maze[0].length ||
        maze[startX][startY] == 'X') {
      return false;
    }
    if (startX == targetX && startY == targetY) {
      return true;
    }
    maze[startX][startY] = 'X';
    int[] dx = {1, 0, -1, 0};
    int[] dy = {0, 1, 0, -1};
    for (int i = 0; i < 4; i++) {
      if (solveMaze(maze, startX + dx[i], startY + dy[i], targetX, targetY)) {
        return true;
      }
    }
    return false;
}
```

Maze

Given a maze and a start point and a target point, return whether the target can be reached.

```
public static boolean solveMaze(char[][] maze,
                                int startX, int startY, int targetX, int targetY) {

    if (startX == targetX && startY == targetY) {
      return true;
    }
    maze[startX][startY] = 'X';
    int[] dx = {1, 0, -1, 0};
    int[] dy = {0, 1, 0, -1};
    for (int i = 0; i < 4; i++) {
      int newX = startX + dx[i], newY = startY + dy[i];
      if (newX < 0 || newX >= maze.length || newY < 0 || newY >= maze.length ||
          maze[newX][newY] == 'X') {
        continue;
      }
      if (solveMaze(maze, newX, newY, targetX, targetY)) {
        return true;
      }
    }
    return false;
}
```

Maze

Given a maze and a start point and a target point, print out the path to reach the target.

```
public static boolean solveMaze(char[][] maze,
                                int startX, int startY, int targetX, int targetY,
                                String path) {
    if (startX < 0 || startX >= maze.length ||
        startY < 0 || startY >= maze[0].length ||
        maze[startX][startY] == 'X') {
      return false;
    }
    if (startX == targetX && startY == targetY) {
      System.out.println(path);
      return true;
    }
    maze[startX][startY] = 'X';
    int[] dx = {1, 0, -1, 0};
    int[] dy = {0, 1, 0, -1};
    char[] direction = {'D', 'R', 'U', 'L'};
    for (int i = 0; i < 4; i++) {
      String newPath = path + direction[i] + " ";
      if (solveMaze(maze, startX+dx[i], startY+dy[i], targetX, targetY, newPath)) {
        return true;
      }
    }
    return false;
}
```

Maze

Given a maze and a start point and a target point, return the path to reach the target.

```
public static boolean solveMaze(char[][] maze,
                                int startX, int startY, int targetX, int targetY,
                                ArrayList<Character> path) {
    if (startX < 0 || startX >= maze.length ||
        startY < 0 || startY >= maze[0].length ||
        maze[startX][startY] == 'X') {
      return false;
    }
    if (startX == targetX && startY == targetY) {
      return true;
    }
    maze[startX][startY] = 'X';
    int[] dx = {1, 0, -1, 0};
    int[] dy = {0, 1, 0, -1};
    char[] direction = {'D', 'R', 'U', 'L'};
    for (int i = 0; i < 4; i++) {
      path.add(direction[i]);
      if (solveMaze(maze, startX+dx[i], startY+dy[i], targetX, targetY, path)) {
        return true;
      }
      path.remove(path.size()-1);
    }
    return false;
}
```

Backtracking Summary

- Backtrack = try, iterate, traverse, etc.
- Keep trying (in the search space) until
  - Solution is found
  - No more meaningful methods to try (no more search space)
- Level-N problem -> M * Level-(N-1) subproblem
  - Keep states the same when entering subproblem except shared fields.

Knapsack

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

All candidate numbers are unique.

The same repeated number may be chosen from C unlimited number of times.

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).
- The solution set must not contain duplicate combinations.

Example Input: [7], [2, 3, 6, 7]

Example Output: [[2, 2, 3], [7]]

Knapsack

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

All candidate numbers are unique.

The same repeated number may be chosen from C unlimited number of times.

Define the problem as (C, i, T), then given a number

- Pick => (C, i, T-C[i])
- Not Pick => (C, i+1, T)

Knapsack

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

```
public static ArrayList<ArrayList<Integer>> knapsack(int[] candidates,
                                                     int target) {
    Arrays.sort(candidates);
    ArrayList<ArrayList<Integer>> results = new ArrayList<ArrayList<Integer>>();
    ArrayList<Integer> cur = new ArrayList<Integer>();
    knapsack(candidates, 0, target, results, cur);
    return results;
}
public static void knapsack(int[] candidates, int index, int target,
                            ArrayList<ArrayList<Integer>> results,
                            ArrayList<Integer> cur) {
    if (target < 0 || (target != 0 && index == candidates.length)) {
        return;
    }
    if (target == 0) {
        results.add(new ArrayList<Integer>(cur));
        return;
    }
    cur.add(candidates[index]);
    knapsack(candidates, index, target-candidates[index], results, cur);
    cur.remove(cur.size()-1);
    knapsack(candidates, index+1, target, results, cur);
}
```

Knapsack

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

All candidate numbers are unique.

The same repeated number may be chosen from C unlimited number of times.

Iterate all numbers to decide whether to pick the current number.

<=>

Which one of the numbers should I pick as the smallest one in the current set.

Knapsack

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

```
public ArrayList<ArrayList<Integer>> knapsack(int[] candidates, int target) {
    Arrays.sort(candidates);
    ArrayList<ArrayList<Integer>> results = new ArrayList<ArrayList<Integer>>();
    ArrayList<Integer> cur = new ArrayList<Integer>();
    knapsack(candidates, 0, target, results, cur);
    return results;
}
public void knapsack(int[] candidates, int index, int target,
                     ArrayList<ArrayList<Integer>> results,
                     ArrayList<Integer> cur) {
    if (target < 0) {
        return;
    }
    if (target == 0) {
        results.add(new ArrayList<Integer>(cur));
        return;
    }
    for (int i = index; i < candidates.length; i++) {
        cur.add(candidates[i]);
        knapsack(candidates, i, target-candidates[i], results, cur);
        cur.remove(cur.size()-1);
    }
    return;
}
```

Knapsack II

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

Candidate numbers may contain duplicate.

Each number in C may only be used once in the combination.

```
public List<List<Integer>> knapsack(int[] candidates, int target) {
    Arrays.sort(candidates);
    List<List<Integer>> results = new ArrayList<List<Integer>>();
    List<Integer> cur = new ArrayList<Integer>();
    knapsack(candidates, 0, target, results, cur);
    return results;
}

public void knapsack(int[] candidates, int index, int target,
                     List<List<Integer>> results, List<Integer> cur) {
    if (target < 0)  return;
    if (target == 0) {
        results.add(new ArrayList<Integer>(cur));
        return;
    }
    for (int i = index; i < candidates.length; i++) {
        cur.add(candidates[i]);
        knapsack(candidates, i+1, target-candidates[i], results, cur);
        cur.remove(cur.size()-1);
        while (i < candidates.length-1 && candidates[i] == candidates[i+1]) i++;
    }
    return;
}
```

0-1 Knapsack II

Given a knapsack which can hold ***s*** pounds of items, and a set of items with weight ***w1, w2, ... wn***. Try to put items into the pack as many as possible, return the largest weight we can get in the knapsack**.**

```
public static int knapsack(int s, int[] weights, int index) {
    if (s == 0 || index == weights.length) {
      return 0;
    }
    if (weights[index] > s) {
      return knapsack(s, weights, index+1);
    }
    return Math.max(knapsack(s, weights, index+1),
                    weights[index] + knapsack(s-weights[index], weights, index+1));
}
```

Generate Parenthesis

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example:

Given n = 3,

return ["((()))", "(()())", "(())()", "()(())", "()()()"]

Generate Parenthesis

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example:

Given n = 3,

return ["((()))", "(()())", "(())()", "()(())", "()()()"]

- n parenthesis => n left + n right => 2*n positions.
- Each time, one left or right should be chosen.
  - We can always pick left.
  - If there are more right remained, we can put right.

Generate Parenthesis

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

```
public ArrayList<String> generateParenthesis(int n) {
    ArrayList<String> results = new ArrayList<>();
    generateParenthesis(results, "", n, n);
    return results;
}

public void generateParenthesis(ArrayList<String> results, String prefix,
                                 int left, int right) {
    if (left == 0 && right == 0) {
        results.add(prefix);
        return;
    }
    if (left > 0) {
        generateParenthesis(results, prefix+"(", left-1, right);
    }
    if (left < right) {
        generateParenthesis(results, prefix+")", left, right-1);
    }
}
```

Permutation (Leetcode 46)

Given a collection of distinct numbers, return all possible permutations.

For example,
[1,2,3] have the following permutations:
[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1].

```
public List<List<Integer>> permute(int[] nums) {
    // Implement this method.
}
```

Permutation (Leetcode 46)

Given a collection of distinct numbers, return all possible permutations.

```
public ArrayList<ArrayList<Integer>> permute(int[] num) {
    ArrayList<Integer> numList = new ArrayList<Integer>();
    Arrays.sort(num);
    for (int i = 0; i < num.length; i++)
        numList.add(num[i]);
    return permute(new ArrayList<Integer>(), numList);
}

public ArrayList<ArrayList<Integer>> permute(ArrayList<Integer> cur,
                                             ArrayList<Integer> num) {
    ArrayList<ArrayList<Integer>> results =
         new ArrayList<ArrayList<Integer>>();
    if (num.size() == 0) {
        results.add(cur);
        return results;
    }
    for (int i = 0; i < num.size(); i++) {
        ArrayList<Integer> newCur = new ArrayList<Integer>(cur);
        newCur.add(num.get(i));
        ArrayList<Integer> newNum = new ArrayList<Integer>(num);
        newNum.remove(i);
        results.addAll(permute(newCur, newNum));
    }
    return results;
}
```

Permutation (Leetcode 46)

Given a collection of distinct numbers, return all possible permutations.

```
public List<List<Integer>> permute(int[] num) {
    Arrays.sort(num);
    return permute(new ArrayList<Integer>(), num);
}

public List<List<Integer>> permute(ArrayList<Integer> cur, int[] num) {
    List<List<Integer>> results = new ArrayList<List<Integer>>();
    if (cur.size() == num.length) {
        results.add(new ArrayList<Integer>(cur));
        return results;
    }
    for (int i = 0; i < num.length; i++) {
        if (cur.contains(num[i])) {
            continue;
        }
        cur.add(num[i]);
        results.addAll(permute(cur, num));
        cur.remove(cur.size() - 1);
    }
    return results;
}
```

Permutation (Leetcode 46)

Given a collection of distinct numbers, return all possible permutations.

```
public List<List<Integer>> permute(int[] num) {
    List<List<Integer>> results = new ArrayList<List<Integer>>();
    Arrays.sort(num);
    permute(results, new ArrayList<Integer>(), num);
    return results;
}

public void permute(List<List<Integer>> results,
                    ArrayList<Integer> cur, int[] num) {
    if (cur.size() == num.length) {
        results.add(new ArrayList<Integer>(cur));
        return;
    }
    for (int i = 0; i < num.length; i++) {
        if (cur.contains(num[i])) {
            continue;
        }
        cur.add(num[i]);
        permute(results, cur, num);
        cur.remove(cur.size() - 1);
    }
}
```

Permutation (Leetcode 46)

Given a collection of distinct numbers, return all possible permutations.

```
public List<List<Integer>> permute(int[] num) {
    List<List<Integer>> results = new ArrayList<List<Integer>>();
    Arrays.sort(num);
    permute(results, num, 0);
    return results;
}

public void permute(List<List<Integer>> results, int[] num, int index) {
    if (index == num.length) {
        ArrayList<Integer> result = new ArrayList<>();
        for (int i = 0; i < num.length; i++) {
            result.add(num[i]);
        }
        results.add(result);
        return;
    }
    for (int i = index; i < num.length; i++) {
        swap(num, index, i);
        permute(results, num, index + 1);
        swap(num, index, i);
    }
}

public void swap(int[] num, int i, int j) {
    int temp = num[i];
    num[i] = num[j];
    num[j] = temp;
}
```

Permutation II (Leetcode 47)

Given a collection of ~~distinct~~ numbers, return all possible permutations.

```
public List<List<Integer>> permuteUnique(int[] num) {
    ArrayList<Integer> numList = new ArrayList<>();
    Arrays.sort(num);
    for (int i = 0; i < num.length; i++)
        numList.add(num[i]);
    return permute(new ArrayList<Integer>(), numList);
}

public List<List<Integer>> permute(
    ArrayList<Integer> cur, ArrayList<Integer> num) {

    List<List<Integer>> results = new ArrayList<>();
    if (num.size() == 0) {
        results.add(cur);
        return results;
    }
    for (int i = 0; i < num.size(); i++) {
        if (i > 0 && num.get(i) == num.get(i-1))
            continue;
        ArrayList<Integer> newCur = new ArrayList<>(cur);
        newCur.add(num.get(i));
        ArrayList<Integer> newNum = new ArrayList<>(num);
        newNum.remove(i);
        results.addAll(permute(newCur, newNum));
    }
    return results;
}
```

Permutation II (Leetcode 47)

Given a collection of ~~distinct~~ numbers, return all possible permutations.

```
public List<List<Integer>> permuteUnique(int[] num) {
    Arrays.sort(num);
    boolean[] visited = new boolean[num.length];
    for (int i = 0; i < visited.length; i++) {
        visited[i] = false;
    }
    return permute(new ArrayList<Integer>(), num, visited);
}

public List<List<Integer>> permute(ArrayList<Integer> cur, int[] num, boolean[] visited) {
    List<List<Integer>> results = new ArrayList<>();
    if (cur.size() == num.length) {
        results.add(new ArrayList<Integer>(cur));
        return results;
    }
    for (int i = 0; i < num.length; i++) {
        if (visited[i] || (i > 0 && num[i] == num[i-1] && !visited[i-1])) {
            continue;
        }
        visited[i] = true;
        cur.add(num[i]);
        results.addAll(permute(cur, num, visited));
        cur.remove(cur.size() - 1);
        visited[i] = false;
    }
    return results;
}
```

Combination

Given a collection of distinct numbers, return all possible combinations.

For example,
[2, 6, 8] have the following permutations:
[], [2], [6], [8], [2, 6], [2, 8], [6, 8], [2, 6, 8].

```
public List<List<Integer>> combine(int[] nums) {
        // Implement this method.
 }
```

Combination

Given a collection of distinct numbers, return all possible combinations.

```
public static List<List<Integer>> combine(int[] nums) {
    List<List<Integer>> results = new ArrayList<>();
    combination(results, nums, 0, new ArrayList<Integer>());
    return results;
}

public static void combination(List<List<Integer>> results,
                               int[] nums, int index, ArrayList<Integer> items) {
    if (index == nums.length) {
      results.add(items);
      return;
    }
    ArrayList<Integer> newItems1 = new ArrayList<Integer>(items);
    combination(results, nums, index+1, newItems1);

    ArrayList<Integer> newItems2 = new ArrayList<Integer>(items);
    newItems2.add(nums[index]);
    combination(results, nums, index+1, newItems2);
  }
```

Combination

Given a collection of distinct numbers, return all possible combinations.

```
public static List<List<Integer>> combine(int[] nums) {
    List<List<Integer>> results = new ArrayList<>();
    combination(results, nums, 0, new ArrayList<Integer>());
    return results;
}

public static void combination(List<List<Integer>> results,
                               int[] nums, int index, ArrayList<Integer> items) {
    if (index == nums.length) {
      results.add(new ArrayList<Integer>(items));
      return;
    }
    combination(results, nums, index+1, items);

    items.add(nums[index]);
    combination(results, nums, index+1, items);
    items.remove(items.size()-1);
  }
```

Combination

Given a collection of distinct numbers, return all possible combinations.

```
public static List<List<Integer>> combine(int[] nums) {
    List<List<Integer>> results = new ArrayList<>();
    combination(results, nums, 0, new ArrayList<Integer>());
    return results;
}

public static void combination(List<List<Integer>> results,
                               int[] nums, int index,
                               ArrayList<Integer> items) {
    if (index == nums.length) {
      results.add(new ArrayList<Integer>(items));
      return;
    }
    for (int i = index; i < nums.length; i++) {
      items.add(nums[i]);
      combination(results, nums, i+1, items);
      items.remove(items.size()-1);
    }
}
```

The empty set is missing.

-There will be at least one element in the "items"

Combination

Given a collection of distinct numbers, return all possible combinations.

```
public static List<List<Integer>> combine(int[] nums) {
    List<List<Integer>> results = new ArrayList<>();
    combination(results, nums, 0, new ArrayList<Integer>());
    return results;
}

public static void combination(List<List<Integer>> results,
                               int[] nums, int index,
                               ArrayList<Integer> items) {
    if (index == nums.length) {
      results.add(new ArrayList<Integer>(items));
      return;
    }
    for (int i = index; i <= nums.length; i++) {
      if (i == nums.length) {
        combination(results, nums, i, items);
        return;
      }
      items.add(nums[i]);
      combination(results, nums, i+1, items);
      items.remove(items.size()-1);
    }
}
```

This adds an empty set.

Combination

Given a collection of distinct numbers, return all possible combinations.

```
public static List<List<Integer>> combine(int[] nums) {
    List<List<Integer>> results = new ArrayList<>();
    combination(results, nums, 0, new ArrayList<Integer>());
    return results;
}

public static void combination(List<List<Integer>> results,
                               int[] nums, int index,
                               ArrayList<Integer> items) {
    if (index == nums.length) {
      results.add(new ArrayList<Integer>(items));
      return;
    }
    for (int i = index; i < nums.length; i++) {
      items.add(nums[i]);
      combination(results, nums, i+1, items);
      items.remove(items.size()-1);
    }
    combination(results, nums, nums.length, items);
}
```

This is the same as above

Lucky Numbers

888 is a lucky number. And for each American phone number, we can actually add some operators to make it become 888. For example:

phone number is 7765332111, you will have

7/7*65*3+3*21*11 = 888

776+5+3*32+11*1 = 888

...

We want to get a full list of all the operation equations that can get a certain lucky number. The interface will be

List<String> luckyNumbers(String num, int target)

Lucky Numbers

We want to get a full list of all the operation equations that can get a certain lucky number. The interface will be

List<String> luckyNumbers(String num, int target)

Additional information:

String num will always be 10 digits since it is a phone number.

we can have "0" but we cannot have "05", "032".

If a number cannot be divided, you cannot use it. For example, 55/2 is not allowed, you need to make sure the division can always be an integer result.

0 cannot be divided.

Lucky Numbers

We need to use recursion to solve the problem

Since we want to output the result. There are things we need to pay attention:

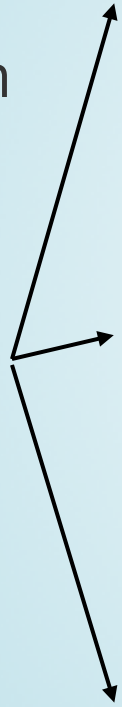
1. The first number does not have operators in front.
2. Use long in case you get the number overflow
3. * and / has higher priority than + and -. How do you handle that?
4. remind 0

Lucky Numbers

```
public List<String> luckyNumbers(String num, int target) {
    List<String> result = new ArrayList<>();
    recursion(num, target, "", 0, 0, 0, result);
    return result;
}
public void recursion(String num, int target, String temp, int pos, long current, long last, List<
    if (pos == num.length()) {
        if (current == target) {
            result.add(temp);
        }
        return;
    }
    for (int i = pos; i < num.length(); i ++) {
        if (num.charAt(pos) == '0' && i != pos) break;
        String m = num.substring(pos, i + 1);
        long n = Long.valueOf(m);
        if (pos == 0) {
            recursion(num, target, temp + m, i + 1, n, n, result);
        } else {
            recursion(num, target, temp + "+" + m, i + 1, current + n, n, result);
            recursion(num, target, temp + "-" + m, i + 1, current - n, -n, result);
            recursion(num, target, temp + "*" + m, i + 1, current - last + last * n, last * n, res
            if (n != 0 && last % n == 0) {
                recursion(num, target, temp + "/" + m, i + 1, current - last + last / n, last / n,
            }
        }
    }
}
```

Summary

Recursion



subproblem
-> problem

Factorial,
Sum of LinkedList,
Remove Linked List Element, etc.

subproblem-1 ||
subproblem-2 ||
subproblem-N
-> problem

Maze,
Sudoku,
Most DFS problems, etc.

subproblem-1 &&
subproblem-2 &&
subproblem-N
-> problem

Knapsack
Permutations,
Combinations,
Eight Queen,
kSum, etc.

Summary

- Recursion is a strategy.
- Always try to split a problem into sub-problems and solve the sub-problem first.
  - If the solution is the same, then you can call the same method.

Homework

Homework (Required)

Letter Combinations of a Phone Number

Subsets

N-Queens

Palindrome Partitioning

Homework (Optional)

Subsets II

Sudoku Solver

Restore IP Address

Word Search