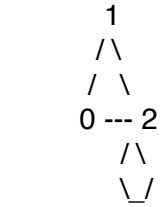
Chapter 7 Graph & Search

Clone Graph
Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.
OJ's undirected graph serialization:
Nodes are labeled uniquely.
We use # as a separator for each node, and , as a separator for node label and each neighbor of the node.
As an example, consider the serialized graph {0,1,2#1,2#2,2}.
The graph has a total of three nodes, and therefore contains three parts as separated by #.
First node is labeled as 0. Connect node 0 to both nodes 1 and 2.
Second node is labeled as 1. Connect node 1 to node 2.
Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle.
Visually, the graph looks like the following:



Answer:

```
/*
SOL 1
图的遍历有两种方式, BFS和DFS,
BFS来解本题, BFS需要使用queue来保存neighbors. 使用Map的主要意义在于充当BFS中Visited数组, 它也可以去环问题, 例如A--B有条边, 当处理完A的邻居node, 然后处理B节点邻居node时发现A已经处理过了. 处理就结束, 不会出现死循环. queue中放置的节点都是未处理neighbors的节点.
SOL 2
递归DFS来解决此题, 思路与上图一致, 但为了避免重复运算产生死循环. 当进入DFS时, 如果发现map中已经有了拷贝过的值, 直接退出即可.
题目虽然简单，但考虑递归的特性使程序简洁. 比如: 我们拷贝只拷贝根节点, 而子节点的拷贝由recursion来完成, 这样可以使程序更加简洁.
注意: 要先加入到map, 再调用rec, 否则会造成不断地反复拷贝而死循环.
*/
public class CloneGraphSol {
	// SOL 1 BFS
	public static UndirectedGraphNode cloneGraph1(UndirectedGraphNode node) {
		if (node == null) {
			return null;
		}

		HashMap<UndirectedGraphNode, UndirectedGraphNode> map = new
HashMap<UndirectedGraphNode, UndirectedGraphNode>();
		Queue<UndirectedGraphNode> q = new LinkedList<UndirectedGraphNode>();
		q.offer(node);
		UndirectedGraphNode nodeCopy = new UndirectedGraphNode(node.label);
		map.put(node, nodeCopy);
		while (!q.isEmpty()) {
```

```
                        UndirectedGraphNode cur = q.poll();
                        UndirectedGraphNode curCopy = map.get(cur);
                        for (UndirectedGraphNode child : cur.neighbors) {
                                if (map.containsKey(child)) {
                                        curCopy.neighbors.add(map.get(child));
                                } else {
                                        q.offer(child);
                                        UndirectedGraphNode childCopy = new
UndirectedGraphNode(child.label);
                                        curCopy.neighbors.add(childCopy);
                                        map.put(child, childCopy);
                                }
                        }
                }
                return map.get(node);
        }
        // SOL 2 DFS
        public static UndirectedGraphNode cloneGraph2(UndirectedGraphNode node) {
                if (node == null) {
                        return null;
                }
                return rec(node, new HashMap<UndirectedGraphNode,
UndirectedGraphNode>());
        }

        public static UndirectedGraphNode rec(UndirectedGraphNode node,
HashMap<UndirectedGraphNode, UndirectedGraphNode> map) {
                if (map.containsKey(node)) {
                        return map.get(node);
                }
                UndirectedGraphNode nodeCopy = new UndirectedGraphNode(node.label);
                map.put(node, nodeCopy);
                for (int i = 0; i < node.neighbors.size(); i++) {
                        nodeCopy.neighbors.add(rec(node.neighbors.get(i), map));
                }
                return nodeCopy;
        }

        public static class UndirectedGraphNode {
                int label;
                List<UndirectedGraphNode> neighbors;
                UndirectedGraphNode(int x) {
                        label = x;
                        neighbors = new ArrayList<UndirectedGraphNode>();
                }
        }

        public static void main(String[] args) {
                UndirectedGraphNode n1 = new UndirectedGraphNode(0);
```

```
                UndirectedGraphNode n2 = new UndirectedGraphNode(1);
                UndirectedGraphNode n3 = new UndirectedGraphNode(2);
                n1.neighbors.add(n2);
                n1.neighbors.add(n3);
                n2.neighbors.add(n3);
                n2.neighbors.add(n1);
                n3.neighbors.add(n3);
                n3.neighbors.add(n1);
                n3.neighbors.add(n2);
                UndirectedGraphNode rst = cloneGraph2(n1);
                HashMap<UndirectedGraphNode, Integer> hash = new
HashMap<UndirectedGraphNode, Integer>();
                Queue<UndirectedGraphNode> q = new LinkedList<UndirectedGraphNode>();
                q.offer(rst);
                hash.put(rst, rst.label);
                while (!q.isEmpty()) {
                        UndirectedGraphNode cur = q.poll();
                        for (UndirectedGraphNode child : cur.neighbors) {
                                if (!hash.containsKey(child)) {
                                        q.offer(child);
                                        hash.put(child, child.label);
                                } else {
                                        continue;
                                }
                        }
                }
                System.out.print(hash.values().toString());
        }
}
/*
 * output:
 * [0, 1, 2]
 */
```

Topological Sorting
Given an directed graph, a topological order of the graph nodes is defined as follow:
For each directed edge A -> B in graph, A must before B in the order list.
The first node in the order can be any node in the graph with no nodes direct to it.
Find any topological order for the given graph.
Example
For graph as follow:
The topological order can be:
[0, 1, 2, 3, 4, 5]
[0, 2, 3, 1, 5, 4]
Answer:

```
/*
入度和出度 每次找到入度为0的点, 去掉其边, 看剩下的点拓扑排序并不是唯一的
*/
```

```
public class TopSortSol {
	public static ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGraphNode>
graph) {
		ArrayList<DirectedGraphNode> rst = new ArrayList<DirectedGraphNode>();
		HashMap<DirectedGraphNode, Integer> map = new
HashMap<DirectedGraphNode, Integer>();
		for (DirectedGraphNode node : graph) {
			for (DirectedGraphNode child : node.neighbors) {
				if (map.containsKey(child)) {
					map.put(child, map.get(child) + 1);
				} else {
					map.put(child, 1);
				}
			}
		}
		Queue<DirectedGraphNode> q = new LinkedList<DirectedGraphNode>();
		for (DirectedGraphNode node : graph) {
			if (!map.containsKey(node)) {
				q.offer(node);
				rst.add(node);
			}
		}
		while (!q.isEmpty()) {
			DirectedGraphNode node = q.poll();
			for (DirectedGraphNode child : node.neighbors) {
				map.put(child, map.get(child) - 1);
				if (map.get(child) == 0) {
					rst.add(child);
					q.offer(child);
				}
			}
		}
		return rst;
	}

	public static class DirectedGraphNode {
		int label;
		ArrayList<DirectedGraphNode> neighbors;
		DirectedGraphNode(int x) {
			label = x;
			neighbors = new ArrayList<DirectedGraphNode>();
		}
	}

	public static void main(String[] args) {
		DirectedGraphNode n1 = new DirectedGraphNode(0);
		DirectedGraphNode n2 = new DirectedGraphNode(1);
		DirectedGraphNode n3 = new DirectedGraphNode(2);
		DirectedGraphNode n4 = new DirectedGraphNode(3);
```

```
                DirectedGraphNode n5 = new DirectedGraphNode(4);
                DirectedGraphNode n6 = new DirectedGraphNode(5);
                n1.neighbors.add(n2);
                n1.neighbors.add(n3);
                n1.neighbors.add(n4);
                n2.neighbors.add(n5);
                n3.neighbors.add(n5);
                n3.neighbors.add(n6);
                n4.neighbors.add(n5);
                n4.neighbors.add(n6);
                ArrayList<DirectedGraphNode> test = new ArrayList<DirectedGraphNode>();
                test.add(n1);
                test.add(n2);
                test.add(n3);
                test.add(n4);
                test.add(n5);
                test.add(n6);
                for (DirectedGraphNode node : test) {
                        System.out.print(node.label);
                }
                System.out.print("\n");
                ArrayList<DirectedGraphNode> rst = topSort(test);
                for (DirectedGraphNode node : rst) {
                        System.out.print(node.label);
                }
        }
}
/*
outputs:
012345
012345
*/
```

Permutations
Given a collection of 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].
Answer:

```
/*
求所有可能解用dfs, time O(n!)
rst 存所有的构造的排列 可行的方案 list 当前可行的可行解 num 可取的数目,
遍历num 看数曾经是否被选取, 若没被选取, 加到list,
dfs search遍历完一个, 进入下一个新的dfs,
取下一个数, 把之前那个数remove, 加入了一个元素, 尝试取下一个元素.
*/
public class permuteSol {
        public static List<List<Integer>> permute1(int[] nums) {
                List<List<Integer>> rst = new ArrayList<List<Integer>>();
```

```
		if (nums == null || nums.length == 0) {
			return rst;
		}

		List<Integer> list = new ArrayList<Integer>();
		helper1(rst, list, nums);
		return rst;
	}

	public static void helper1(List<List<Integer>> rst, List<Integer> list, int[] nums) {
		// list = new ArrayList<Integer>();
		if (list.size() == nums.length) {
			rst.add(new ArrayList<Integer>(list));
			return;
		}

		for (int i = 0; i < nums.length; i++) {
			if (list.contains(nums[i])) {
				continue;
			}
			list.add(nums[i]);
			helper1(rst, list, nums);
			list.remove(list.size() - 1);
		}
	}

	public static List<List<Integer>> permute2(int[] nums) {
		List<List<Integer>> rst = new ArrayList<List<Integer>>();
		if (nums == null || nums.length == 0) {
			return rst;
		}

		helper2(rst, nums, new LinkedHashMap<Integer, Integer>());
		return rst;
	}

	public static void helper2(List<List<Integer>> rst, int[] nums, LinkedHashMap<Integer,
Integer> set) {
		if (set.size() == nums.length) {
			List<Integer> list = new ArrayList<Integer>();
			for (Integer i : set.keySet()) {
				list.add(i);
			}
			rst.add(list);
			return;
		}

		int len = nums.length;
		for (int i = 0; i < len; i++) {
```

```
                        if (set.containsKey(nums[i])) {
                                continue;
                        }
                        set.put(nums[i], 0);
                        helper2(rst, nums, set);
                        set.remove(nums[i]);
                }
        }

        public static void main(String[] args) {
                int[] nums = {1, 2, 3};
                System.out.print(permute1(nums));
                System.out.print("\n");
                System.out.print(permute2(nums));
        }
}
```

Next Permutation
Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.
If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).
The replacement must be in-place, do not allocate extra memory.
Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.
1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1
Answer:

```
/*
用来计算排列组合的函数, 对序列{a, b, c}, 每一个元素都比后面的小, 按照字典序列, 固定a之后, a比bc都小, c比b大, 它的下一个序列即为{a, c, b}, 而{a, c, b}的上一个序列即为{a, b, c},
同理可以推出所有的六个序列为: {a, b, c}, {a, c, b}, {b, a, c}, {b, c, a}, {c, a, b}, {c, b, a}, 其中{a, b, c}没有上一个元素, {c, b, a}没有下一个元素,
*/
public class NextPermutationSol {
        public static void nextPermutation1(int[] nums) {
                if (nums == null) {
                        return;
                }

                int len = nums.length;
                int dropIndex = -1;
                for (int i = len - 1; i >= 0; i--) {
                        if (i != len - 1 && nums[i] < nums[i + 1]) {
                                dropIndex = i;
                                break;
                        }
                }
```

```
                if (dropIndex != -1) {
                        for (int i = len - 1; i >= 0; i--) {
                                if (nums[i] > nums[dropIndex]) {
                                        swap(nums, dropIndex, i);
                                        break;
                                }
                        }
                }

                int l = dropIndex + 1;
                int r = len - 1;
                while (l < r) {
                        swap(nums, l, r);
                        l++;
                        r--;
                }
        }

        public static void swap(int[] nums, int l, int r) {
                int tmp = nums[l];
                nums[l] = nums[r];
                nums[r] = tmp;
        }

        public static void main(String[] args) {
                int[] test = {1, 2, 4, 3, 2, 1};
                nextPermutation1(test);
                for (int i = 0; i < test.length; i++) {
                        System.out.print(test[i] + " ");
                }
        }
}
/*
 * outputs:
 * 1 3 1 2 2 4
 */
```

Permutations II
Given a collection of numbers that might contain duplicates, return all possible unique permutations.
For example,
[1,1,2] have the following unique permutations:
[1,1,2], [1,2,1], and [2,1,1].
*/
/*
SOL 1
For deal with the duplicate solution, we should sort it.
递归以及回溯

```
SOL 2
使用一个pre来记录。只取第一个可以取的位置
只取第一个可取的位置，因为别的位置取到的也没有区别
*/
/*
SOL 1
dfs()
  visit[0] = true
  list.add(1)
  dfs()
    visit[2] = true
    list.add(2)
    dfs()
    list.remove(2 - 1)
    visit[2] = false
   list.remove(0)
   visit[0] = false
   visit[1] = true
   list.add(1)
   dfs()
     visit[0] = true
     list.add(1)
     dfs()
       visit[2] = true
       list.add(2)
       dfs()
         rst.add(list) [[1, 1, 2]]
       list.remove(2)
       visit[2] = false
     list.remove(1)
     visit[0] = false
     visit[2] = true
     list.add(2)
     dfs()
       visit[0] = true
       list.add(1)
       dfs()
         rst.add(list) [[1, 1, 2] [1, 2, 1]]
    list.remove(0)
    visit[1] = false

    visit[2] = true
    list.add(2)
    dfs()
      visit[0] = true
      list.add(1)
      dfs()
      list.remove(2 - 1)
      visit[0] = false
```

```
        visit[1] = true
        list.add(1)
        dfs()
          visit[0] = true
          list.add(1)
          dfs()
            rst.add(list) [[1, 1, 2] [1, 2, 1] [2, 1, 1]]
          ...
        ...
      ...
*/
public class permuteUniqueSol {
        public static List<List<Integer>> permuteUnique(int[] nums) {
                if (nums == null) {
                        return null;
                }

                List<List<Integer>> rst = new ArrayList<List<Integer>>();
                List<Integer> list = new ArrayList<Integer>();
                boolean[] visit = new boolean[nums.length];
                Arrays.sort(nums);
                // SOL 1
                dfs1(nums, list, rst, visit);
                // SOL 2
                // dfs2(nums, list, rst, visit);
                return rst;
        }

        public static void dfs1(int[] nums, List<Integer> list, List<List<Integer>> rst, boolean[]
visit) {
                if (list.size() == nums.length) {
                        rst.add(new ArrayList<Integer>(list));
                        return;
                }

                for (int i = 0; i < nums.length; i++) {
                        if (visit[i] || (i != 0 && visit[i - 1] && nums[i] == nums[i - 1])) {
                                continue;
                        }

                        visit[i] = true;
                        list.add(nums[i]);
                        dfs1(nums, list, rst, visit);
                        list.remove(list.size() - 1);
                        visit[i] = false;
                }
        }
```

```
	public static void dfs2(int[] nums, List<Integer> list, List<List<Integer>> rst, boolean[]
visit) {
		if (list.size() == nums.length) {
			rst.add(new ArrayList<Integer>(list));
			return;
		}

		long pre = Long.MIN_VALUE;
		for (int i = 0; i < nums.length; i++) {
			int value = nums[i];
			if (visit[i] || pre == value) {
				continue;
			}
			pre = value;
			visit[i] = true;
			list.add(value);
			dfs2(nums, list, rst, visit);
			list.remove(list.size() - 1);
			visit[i] = false;
		}
	}

	public static void main(String[] args) {
		int[] nums = {1, 1, 2};
		System.out.print(permuteUnique(nums));
	}
}
```

Permutation Sequence
The set [1,2,3,…,n] contains a total of n! unique permutations.
By listing and labeling all of the permutations in order,
We get the following sequence (ie, for n = 3):
"123"
"132"
"213"
"231"
"312"
"321"
Given n and k, return the kth permutation sequence.
Note: Given n will be between 1 and 9 inclusive.
Answer:
/*
SOL 1
解答：
1. 以某一数字开头的排列有(n-1)! 个,
例如: 123, 132, 以1开头的是 2！个
2. 所以第一位数字就可以用 (k-1）/ (n-1)！来确定. 这里K-1的原因是, 序列号我们应从0开始计算, 否则在边界时无法计算,
3. 第二位数字. 假设前面取余后为m, 则第二位数字是第 m/(n-2)! 个未使用的数字,

4. 不断重复2, 3, 取余并且对(n-k)!进行除法, 直至计算完毕.
SOL 2
优化后, 使用链表来记录未使用的数字, 每用掉一个, 将它从链表中移除即可
SOL 3
在2解基础进一步优化, 使用for循环替代while循环, 更简洁
*/

```
public class PermuteSeqSol {
	public static String getPermutation1(int n, int k) {
		if (n == 0) {
			return "";
		}
		// n!
		int num = 1;
		for (int i = 1; i <= n; i++) {
			num *= i;
		}

		boolean[] use = new boolean[n];
		for (int i = 0; i < n; i++) {
			use[i] = false;
		}

		k = k - 1;
		StringBuilder sb = new StringBuilder();
		for (int i = 0; i < n; i++) {
			num = num / (n - i);
			int index = k / num;
			k = k % num;
			for (int j = 0; j < n; j++) {
				if (!use[j]) {
					if (index == 0) {
						sb.append((j + 1) + "");
						use[j] = true;
						break;
					}
					index--;
				}
			}
		}
		return sb.toString();
	}

	public static String getPermutation2(int n, int k) {
		LinkedList<Character> digits = new LinkedList<Character>();
		for (char i = '1'; i <= '0' + n; i++) {
			digits.add(i);
		}
		k = k - 1;
		StringBuilder sb = new StringBuilder();
```

```
                int num = 1;
                for (int i = 1; i <= n; i++) {
                        num *= i;
                }
                int cur = n;
                while (!digits.isEmpty()) {
                        num = num / cur;
                        cur--;
                        int digitIndex = k / num;
                        k = k % num;
                        sb.append(digits.get(digitIndex));
                        digits.remove(digitIndex);
                }
                return sb.toString();
        }

        public static String getPermutation3(int n, int k) {
                LinkedList<Character> digits = new LinkedList<Character>();
                for (char i = '1'; i <= '0' + n; i++) {
                        digits.add(i);
                }
                k = k - 1;
                StringBuilder sb = new StringBuilder();
                int num = 1;
                for (int i = 1; i <= n; i++) {
                        num *= i;
                }
                for (int i = n; i >= 1; i--) {
                        num = num / i;
                        int digitIndex = k / num;
                        k = k % num;
                        sb.append(digits.get(digitIndex));
                        digits.remove(digitIndex);
                }
                return sb.toString();
        }

        public static void main(String[] args) {
                System.out.print(getPermutation3(3, 5));
        }
}
```

Subsets
Given a set of distinct integers, nums, return all possible subsets.
Note:
Elements in a subset must be in non-descending order.
The solution set must not contain duplicate subsets.
For example,
If nums = [1,2,3], a solution is:

```
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

Answer:

```
/*
Basically our algorithm can't be faster than O(2^n) since we need to go through all possible combinations.
SOL 1 Recurstion
SOL 3 The idea is that all the numbers from 0 to 2^n are represented by unique bit strings of n bit width that can be translated into a subset.
e.g.
Nr Bits Combination
0  000  {}
1  001  {1}
2  010  {2}
3  011  {1, 2}
4  100  {3}
5  101  {1, 3}
6  110  {2, 3}
7  111  {1, 2, 3]
int numberOfTrailingZeros(int i) 给定一个int类型数据, 返回这个数据的二进制串中从最右边算起连续的“0”的总数量, 因为int类型的数据长度为32所以高位不足的地方会以“0”填充.
1的二进制表示为: 1最右边开始数起连续的0的个数为: 0
2的二进制表示为: 10最右边开始数起连续的0的个数为: 1
*/
public class SubsetsSol {
	public static List<List<Integer>> subsets1(int[] nums) {
		List<List<Integer>> rst = new ArrayList<List<Integer>>();
		if (nums == null) {
			return rst;
		}
		Arrays.sort(nums);
		dfs(nums, 0, new ArrayList<Integer>(), rst);
		return rst;
	}

	public static void dfs(int[] nums, int index, List<Integer> list, List<List<Integer>> rst) {
		// 先加一个空集合进去
		rst.add(new ArrayList<Integer>(list));
		for (int i = index; i < nums.length; i++) {
			list.add(nums[i]);
			dfs(nums, i + 1, list, rst);
```

```
                list.remove(list.size() - 1);
            }
        }

        public static List<List<Integer>> subsets2(int[] nums) {
            List<List<Integer>> rst = new ArrayList<List<Integer>>();
            if (nums == null) {
                return rst;
            }
            Arrays.sort(nums);
            return dfs2(nums, 0, new HashMap<Integer, List<List<Integer>>>());
        }

        public static List<List<Integer>> dfs2(int[] nums, int index, HashMap<Integer,
List<List<Integer>>> map) {
            int len = nums.length;
            if (map.containsKey(index)) {
                return map.get(index);
            }
            List<List<Integer>> rst = new ArrayList<List<Integer>>();
            List<Integer> pathTmp = new ArrayList<Integer>();
            rst.add(pathTmp);
            for (int i = index; i < len; i++) {
                List<List<Integer>> left = dfs2(nums, i + 1, map);
                for (List<Integer> list : left) {
                    pathTmp = new ArrayList<Integer>();
                    pathTmp.add(nums[i]);
                    pathTmp.addAll(list);
                    rst.add(pathTmp);
                }
            }
            map.put(index, rst);
            return rst;
        }

        public static List<List<Integer>> subsets3(int[] nums) {
            List<List<Integer>> rst = new ArrayList<List<Integer>>();
            if (nums == null || nums.length == 0) {
                return rst;
            }
            int len = nums.length;
            Arrays.sort(nums);
            long numOfSet = (long)Math.pow(2, len);
            for (int i = 0; i < numOfSet; i++) {
                long tmp = i;
                ArrayList<Integer> list = new ArrayList<Integer>();
                while (tmp != 0) {
                    int indexOfLast1 = Long.numberOfTrailingZeros(tmp);
                    list.add(nums[indexOfLast1]);
```

```
                        tmp ^= (1 << indexOfLast1);
                    }
                    rst.add(list);
                }
                return rst;
        }

        public static void main(String[] args) {
                int[] test = {1, 2, 3};
                System.out.print(subsets3(test));
        }
}
```

Subsets II
Given a list of numbers that may has duplicate numbers, return all possible subsets
Example
If S = [1,2,2], a solution is:

```
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

Note
Each element in a subset must be in non-descending order.
The ordering between two subsets is free.
The solution set must not contain duplicate subsets.
Answer:

```
public class Subsets2Sol {
        public static List<List<Integer>> subsetsWithDup(int[] nums) {
                List<List<Integer>> rst = new ArrayList<List<Integer>>();
                List<Integer> list = new ArrayList<Integer>();
                if (nums == null || nums.length == 0) {
                        return rst;
                }
                Arrays.sort(nums);
                helper(rst, list, nums, 0);
                return rst;
        }

        public static void helper(List<List<Integer>> rst, List<Integer> list, int[] nums, int pos) {
                rst.add(new ArrayList<Integer>(list));
                for (int i = pos; i < nums.length; i++) {
                        if (i != pos && nums[i] == nums[i - 1]) {
                                continue;
                        }
                        list.add(nums[i]);
```

```
                helper(rst, list, nums, i + 1);
                list.remove(list.size() - 1);
            }
        }

        public static void main(String[] args) {
            int[] test = {1, 2, 2};
            System.out.print(subsetsWithDup(test));
        }
}
```

N-Queens
Given an integer n, return all distinct solutions to the n-queens puzzle.
Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.
For example,
There exist two distinct solutions to the 4-queens puzzle:

```
[
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],
 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
```

Challenge
Can you do it without recursion?
Answer:

```
/*
SOL 1
cols: 存放每一行皇后的列值, 我们在每一行放一个皇后
DFS 某一行中所有的位置, 看是否可以放置一个皇后
若加新皇后在row = cols.size() - 1行的下一行的第col列, 则新皇后的行号和列号是:
(cols.size(), col)
但若棋盘上已有的某皇后位置, 比如(i, cols.get(i)), 与新皇后位置互为对角线, 则不可行
*/
public class NQueensSol {
    public static List<String[]> solveNQueens1(int n) {
        List<String[]> rst = new ArrayList<String[]>();
        List<Integer> cols = new ArrayList<Integer>();
        if (n <= 0) {
            return rst;
        }
        helper(n, cols, rst);
        return rst;
    }
```

```
public static String[] createSol(int n, List<Integer> cols) {
        String[] chessboard = new String[n];
        for (int i = 0; i < n; i++) {
                StringBuilder sb = new StringBuilder();
                for (int j = 0; j < n; j++) {
                        if (j == cols.get(i)) {
                                sb.append('Q');
                        } else {
                                sb.append('.');
                        }
                }
                chessboard[i] = sb.toString();
        }
        return chessboard;
}

private static boolean isValid(List<Integer> cols, int col) {
        for (int i = 0; i < cols.size(); i++) {
                if (col == cols.get(i)) {
                        return false;
                }
                if (cols.size() - i == Math.abs(col - cols.get(i))) {
                        return false;
                }
        }
        return true;
}

private static void helper(int n, List<Integer> cols, List<String[]> rst) {
        if (cols.size() == n) {
                rst.add(createSol(n, new ArrayList<Integer>(cols)));
                return;
        }
        for (int col = 0; col < n; col++) {
                if (!isValid(cols, col)) {
                        continue;
                }
                cols.add(col);
                helper(n, cols, rst);
                cols.remove(cols.size() - 1);
        }
}

public static List<String[]> solveNQueens2(int n) {
        List<String[]> ret = new ArrayList<String[]>();
        if (n == 0) {
                return ret;
        }
        StringBuilder[] list = new StringBuilder[n];
```

```
		for (int i = 0; i < n; i++) {
			list[i] = new StringBuilder();
			for (int j = 0; j < n; j++) {
				list[i].append('.');
			}
		}
		dfs(n, list, ret, 0);
		return ret;
	}

	public static void dfs(int n, StringBuilder[] list, List<String[]> ret, int index) {
		int rows = list.length;
		if (n == 0) {
			String[] strs = new String[rows];
			for (int i = 0; i < rows; i++) {
				strs[i] = list[i].toString();
			}
			ret.add(strs);
			return;
		}
		if (index >= rows * rows) {
			return;
		}
		for (int i = index; i < rows * rows; i++) {
			int row = i / rows;
			int col = i % rows;
			if (!canPut(list, row, col)) {
				continue;
			}
			list[row].setCharAt(col, 'Q');
			dfs(n - 1, list, ret, i + 1);
			list[row].setCharAt(col, '.');
		}
		return;
	}

	public static boolean canPut(StringBuilder[] list, int row, int col) {
		for (int i = 0; i < list.length; i++) {
			for (int j = 0; j < list.length; j++) {
				if (list[i].charAt(j) == 'Q' &&
					(i == row || j == col || Math.abs(i - row) ==
Math.abs(j - col))) {
					return false;
				}
			}
		}
		return true;
	}
```

```
        public static void main(String[] args) {
                List<String[]> list = solveNQueens2(4);
                for (String[] str : list) {
                        for (String s : str) {
                                System.out.println(s);
                        }
                        System.out.print("\n");
                }
        }
}
```

N-Queens II
Follow up for N-Queens problem.
Now, instead outputting board configurations, return the total number of distinct solutions.
Answer:

```
/*
The base case: 当最后一行, 皇后只有1种放法(就是不放)
当row == size时, 也就是row越界了, 这时意思是整个N-Queen都摆满了, 我们这时应返回解为1.
因为
你不需要任何动作, 也就是唯一的解是保持原状.
*/
public class Solution {
    public int totalNQueens(int n) {
        if (n == 0) {
            return 0;
        }

        return dfs(n, 0, new ArrayList<Integer>());
    }

    public int dfs(int n, int row, ArrayList<Integer> path) {
        if (row == n) {
            return 1;
        }
        int cnt = 0;
        for (int i = 0; i < n; i++) {
            if (!isValid(path, i)) {
                continue;
            }
            path.add(i);
            cnt += dfs(n, row + 1, path);
            path.remove(path.size() - 1);
        }
        return cnt;
    }

    public boolean isValid(ArrayList<Integer> path, int col) {
        int size = path.size();
        for (int i = 0; i < size; i++) {
```

```
            if (col == path.get(i)) {
                return false;
            }
            if (size - i == Math.abs(col - path.get(i))) {
                return false;
            }
        }
        return true;
    }
}
```

Palindrome Partitioning
Given a string s, partition s such that every substring of the partition is a palindrome.
Return all possible palindrome partitioning of s.
For example, given s = "aab",
Return
```
  [
    ["aa","b"],
    ["a","a","b"]
  ]
```
Answer:
```
/*
time O(2^N)
*/
public class partitionSol {
        // SOL 1 DFS
        public static List<List<String>> partition1(String s) {
                List<List<String>> ret = new ArrayList<List<String>>();
                if (s == null) {
                        return ret;
                }
                dfs1(s, 0, new ArrayList<String>(), ret);
                return ret;
        }

        public static void dfs1(String s, int index, List<String> path, List<List<String>> ret) {
                int len = s.length();
                if (index == len) {
                        ret.add(new ArrayList<String>(path));
                        return;
                }

                for (int i = index; i < len; i++) {
                        String sub = s.substring(index, i + 1);
                        if (!isPalindrome1(sub)) {
                                continue;
                        }
                        path.add(sub);
                        dfs1(s, i + 1, path, ret);
```

```
                path.remove(path.size() - 1);
            }
        }

        public static boolean isPalindrome1(String s) {
            int len = s.length();
            int left = 0;
            int right = len - 1;
            while (left < right) {
                if (s.charAt(left) != s.charAt(right)) {
                    return false;
                }
                left++;
                right--;
            }
            return true;
        }
        // SOL 2 DFS + HashMap
        public static List<List<String>> partition2(String s) {
            List<List<String>> ret = new ArrayList<List<String>>();
            if (s == null) {
                return ret;
            }
            dfs2(s, 0, new ArrayList<String>(), ret, new HashMap<String, Boolean>());
            return ret;
        }

        public static void dfs2(String s, int index, List<String> path, List<List<String>> ret,
HashMap<String, Boolean> map) {
            int len = s.length();
            if (index == len) {
                ret.add(new ArrayList<String>(path));
                return;
            }
            for (int i = index; i < len; i++) {
                String sub = s.substring(index, i + 1);
                if (!isPalindrome2(sub, map)) {
                    continue;
                }
                path.add(sub);
                dfs2(s, i + 1, path, ret, map);
                path.remove(path.size() - 1);
            }
        }

        public static boolean isPalindrome2(String s, HashMap<String, Boolean> map) {
            int len = s.length();
            int left = 0;
            int right = len - 1;
```

```
                if (map.get(s) != null) {
                        return map.get(s);
                }
                // map.put(s, true);
                while (left < right) {
                        if (s.charAt(left) != s.charAt(right)) {
                                map.put(s, false);
                                return false;
                        }
                        left++;
                        right--;
                }
                return true;
        }
        // SOL 3 DP + DFS
    public static List<List<String>> partition3(String s) {
        List<List<String>> ret = new ArrayList<List<String>>();
        if (s == null) {
                return ret;
        }
        int len = s.length();
        boolean[][] D = new boolean[len][len];
        for (int j = 0; j < len; j++) {
                for (int i = 0; i <= j; i++) {
                        D[i][j] = s.charAt(i) == s.charAt(j) && (j - i <= 2 || D[i + 1][j - 1]);
                }
        }
        dfs3(s, 0, new ArrayList<String>(), ret, D);
        return ret;
    }

    public static void dfs3(String s, int index, List<String> path, List<List<String>> ret, boolean[][]
D) {
        int len = s.length();
        if (index == len) {
                ret.add(new ArrayList<String>(path));
                return;
        }
        for (int i = index; i < len; i++) {
                String sub = s.substring(index, i + 1);
                if (!D[index][i]) {
                        continue;
                }
                path.add(sub);
                dfs3(s, i + 1, path, ret, D);
                path.remove(path.size() - 1);
        }
    }
```

```
        public static void main(String[] args) {
                String test = "abb";
                System.out.print(partition3(test));
        }
}
```

Palindrome Partitioning II
Given a string s, partition s such that every substring of the partition is a palindrome.
Return the minimum cuts needed for a palindrome partitioning of s.
For example, given s = "aab",
Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.
Answer:

```
/*
使用DP来解决:
1. D[i]表示前i个字符切为回文需要的切割数
2. P[i][j]: S.sub(i-j) is a palindrome.
3. 递推公式: D[i] = Math.min(D[i], D[j] + 1), 0 <= j <= i - 1), 并且要判断P[j][i - 1]是不是回文.
4. 注意D[0] = -1的用意, 它是指当整个字符串判断出是回文是, 因为会D[0] + 1其实应该是结果为0
(没有任何切割), 所以, 应把D[0]设置为-1
有个转移函数之后, 一个问题出现了, 就是如何判断[i,j]是否是回文? 每次都从i到j比较一遍? 太浪费
了, 这里也是一个DP问题.
定义函数
P[i][j] = true if [i, j]为回文
那么
P[i][j] = str[i] == str[j] && P[i + 1][j - 1].
The worst case is cut character one by one.
*/
public class Solution {
    public int minCut(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }
        int len = s.length();
        int[] D = new int[len + 1];
        D[0] = -1;
        boolean[][] P = new boolean[len][len];
        for (int i = 1; i <= len; i++) {
            D[i] = i - 1;
            for (int j = 0; j <= i - 1; j++) {
                P[j][i - 1] = false;
                if (s.charAt(j) == s.charAt(i - 1) && (i - 1 - j <= 2 || P[j + 1][i - 2])) {
                    P[j][i - 1] = true;
                    D[i] = Math.min(D[i], D[j] + 1);
                }
            }
        }
        return D[len];
    }
}
```

Valid Palindrome
Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.
For example,
"A man, a plan, a canal: Panama" is a palindrome.
"race a car" is not a palindrome.
Note:
Have you consider that the string might be empty? This is a good question to ask during an interview.
For the purpose of this problem, we define empty string as valid palindrome.
Answer:

```
/*
O(n) runtime, O(1) space
*/
public class isPalindromeSol {
	// SOL 1 Iterator
	public static boolean isPalindrome1(String s) {
		if (s == null) {
			return false;
		}
		int len = s.length();
		int left = 0;
		int right = len - 1;
		String sNew = s.toLowerCase();
		while (left < right) {
			while (left < right && !isNumChar(sNew.charAt(left))) {
				left++;
			}
			while (left < right && !isNumChar(sNew.charAt(right))) {
				right--;
			}
			if (sNew.charAt(left) != sNew.charAt(right)) {
				return false;
			}
			left++;
			right--;
		}
		return true;
	}

	public static boolean isNumChar(char c) {
		if (c <= '9' && c >= '0' || c <= 'z' && c >= 'a' || c <= 'Z' && c >= 'A') {
			return true;
		}
		return false;
	}

	public static boolean isPalindrome2(String s) {
		if (s == null) {
```

```
                        return false;
                }
                int len = s.length();
                boolean ret = true;
                int left = 0;
                int right = len - 1;
                String sNew = s.toLowerCase();
                while (left < right) {
                        if (!Character.isLetterOrDigit(sNew.charAt(left))) {
                                left++;
                        } else if (!Character.isLetterOrDigit(sNew.charAt(right))) {
                                right--;
                        } else if (sNew.charAt(left) != sNew.charAt(right)) {
                                return false;
                        } else {
                                left++;
                                right--;
                        }
                }
                return true;
        }

        public static void main(String[] args) {
                String s = "A man, a plan, a canal: Panama";
                System.out.print(isPalindrome2(s));
        }
}
```

Palindrome Number
Determine whether an integer is a palindrome. Do this without extra space.
Some hints:
Could negative integers be palindromes? (ie, -1)
If you are thinking of converting the integer to string, note the restriction of using extra space.
You could also try reversing an integer. However, if you have solved the problem "Reverse Integer", you know that the reversed integer might overflow. How would you handle such case?
There is a more generic way of solving this problem.
Answer:

```
public class Solution {
    public boolean isPalindrome(int x) {
        if (x < 0) {
            return false;
        }
        return x == reverse(x);
    }

    private int reverse(int x) {
        int rst = 0;
        while (x != 0) {
            rst = rst * 10 + x % 10;
```

```
            x = x / 10;
        }
        return rst;
    }
}
```

Shortest Palindrome
Given a string S, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.
For example:
Given "aacecaaa", return "aaacecaaa".
Given "abcd", return "dcbabcd".
Answer:

```
/*
O(N^2) runtime (each isPalindrome is O(N), done N times)
substring(int beginIndex)
*/
public class Solution {
        public static String shortestPalindrome(String s) {
        if (isPalindrome(s)) {
            return s;
        }
        int i = s.length();
        while (i > 0) {
            String sub1 = s.substring(0, i--);
            if (isPalindrome(sub1)) {
                break;
            }
        }
        i++;
        String sub2 = s.substring(i);
        StringBuilder sb = new StringBuilder();
        for (int j = sub2.length() - 1; j >= 0; j--) {
            sb.append(sub2.charAt(j));
        }
        for (int j = 0; j < s.length(); j++) {
            sb.append(s.charAt(j));
        }
        return sb.toString();
    }

    private static boolean isPalindrome(String s) {
        int i = 0;
        int j = s.length() - 1;
        while (i < j) {
            if (s.charAt(i++) != s.charAt(j--)) {
                return false;
            }
        }
```

```
        return true;
    }

    public static void main(String[] args) {
        String test = "bcba";
        System.out.print(shortestPalindrome(test));
    }
}
```

Combination Sum
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.
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.
For example, given candidate set 2,3,6,7 and target 7,
A solution set is:
[7]
[2, 2, 3]
Answer:

```
/*
经典递归模板.
i的起始值是跟排列的最主要区别. 因为与顺序无关, 所以我们必须只能是升序, 也就是说下一个取值只能是i本身或是i的下一个.
但是排列的话, 可以再取前同的. 1, 2 与2 1是不同的排列, 但是同一个组合.
注意, 最后的参数是i, 不是index.
*/
public class Solution {
    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> ret = new ArrayList<List<Integer>>();
        if (candidates == null || candidates.length == 0) {
            return ret;
        }
        Arrays.sort(candidates);
        dfs(candidates, target, new ArrayList<Integer>(), ret, 0);
        return ret;
    }

    public void dfs(int[] candidates, int target, List<Integer> path, List<List<Integer>> ret, int index) {
        if (target < 0) {
            return;
        }
        if (target == 0) {
            ret.add(new ArrayList<Integer>(path));
            return;
```

```
        }
        for (int i = index; i < candidates.length; i++) {
            int num = candidates[i];
            path.add(num);
            dfs(candidates, target - num, path, ret, i);
            path.remove(path.size() - 1);
        }

    }
}
```

Combination Sum 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.
Each number in C may only be used once in the combination.
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.
For example, given candidate set 10,1,2,7,6,1,5 and target 8,
A solution set is:
[1, 7]
[1, 2, 5]
[2, 6]
[1, 1, 6]
Answer:

```
/*
每次只取第一个, 例如123334, 到了333这里, 我们第一次只取第1个3, 因为我们选任何一个3是对组合来说是一个解. 所以只第一次取就好了.
*/
public class Solution {
    public List<List<Integer>> combinationSum2(int[] candidates, int target) {
        List<List<Integer>> rst = new ArrayList<List<Integer>>();
        if (candidates == null || candidates.length == 0) {
            return rst;
        }
        Arrays.sort(candidates);
        dfs1(candidates, target, new ArrayList<Integer>(), rst, 0);
        return rst;
    }

    public void dfs1(int[] candidates, int target, List<Integer> path, List<List<Integer>> rst, int index) {
        if (target == 0) {
            rst.add(new ArrayList<Integer>(path));
            return;
        }
        if (target < 0) {
```

```
            return;
        }
        int pre = -1;
        for (int i = index; i < candidates.length; i++) {
            int num = candidates[i];
            if (num == pre) {
                continue;
            }
            pre = num;
            path.add(num);
            dfs1(candidates, target - num, path, rst, i + 1);
            path.remove(path.size() - 1);
        }
    }
    /*
    public void dfs2(int[] candidates, int target, List<Integer> path, List<List<Integer>> rst, int
index) {
        if (target == 0) {
            rst.add(new ArrayList<Integer>(path));
            return;
        }
        if (target < 0) {
            return;
        }
        for (int i = index; i < candidates.length; i++) {
            int num = candidates[i];
            if (i != index && num == candidates[i - 1]) {
                continue;
            }
            path.add(num);
            dfs2(candidates, target - num, path, rst, i + 1);
            path.remove(path.size() - 1);
        }
    }
    */
}
```

Letter Combinations of a Phone Number
Given a digit string, return all possible letter combinations that the number could represent.
A mapping of digit to letters (just like on the telephone buttons) is given below.
1 o_o 2 abc 3 def
4 ghi 5 jkl 6 mno
7 pqrs 8 tuv 9 wxyz
Input:Digit string "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
Note:
Although the above answer is in lexicographical order, your answer could be in any order you want.
Answer:

```
public class Solution {
    // SOL 1 DFS Backtracking
    public List<String> letterCombinations1(String digits) {
        List<String> rst = new ArrayList<String>();
        String[] map = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
        if (digits.length() != 0) {
            dfs1(digits, new StringBuilder(), rst, map, 0);
        }
        return rst;
    }

    public void dfs1(String digits, StringBuilder sb, List<String> rst, String[] map, int index) {
        int len = digits.length();
        if (index == len) {
            rst.add(sb.toString());
            return;
        }
        String s = map[digits.charAt(index) - '0'];
        for (int i = 0; i < s.length(); i++) {
            sb.append(s.charAt(i));
            dfs1(digits, sb, rst, map, index + 1);
            sb.deleteCharAt(sb.length() - 1);
        }
    }

    public List<String> letterCombinations2(String digits) {
        List<String> rst = new ArrayList<String>();
        Map<Character, char[]> map = new HashMap<Character, char[]>();
        map.put('0', new char[] {});
        map.put('1', new char[] {});
        map.put('2', new char[] {'a', 'b', 'c'});
        map.put('3', new char[] {'d', 'e', 'f'});
        map.put('4', new char[] {'g', 'h', 'i'});
        map.put('5', new char[] {'j', 'k', 'l'});
        map.put('6', new char[] {'m', 'n', 'o'});
        map.put('7', new char[] {'p', 'q', 'r', 's'});
        map.put('8', new char[] {'t', 'u', 'v'});
        map.put('9', new char[] {'w', 'x', 'y', 'z'});
        StringBuilder sb = new StringBuilder();
        if (digits.length() != 0) {
            dfs2(map, digits, sb, rst, 0);
        }
        return rst;
    }

    private void dfs2(Map<Character, char[]> map, String digits, StringBuilder sb, List<String> rst,
int index) {
        int len = digits.length();
        if (index == len) {
```

```
            rst.add(sb.toString());
            return;
        }
        char[] s = map.get(digits.charAt(index));
        for (int i = 0; i < s.length; i++) {
            sb.append(s[i]);
            dfs2(map, digits, sb, rst, index + 1);
            sb.deleteCharAt(sb.length() - 1);
        }
    }
}
```

Combinations
Given two integers n and k, return all possible combinations of k numbers out of 1 ... n.
For example,
If n = 4 and k = 2, a solution is:

```
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

Answer:

```
/*
这是一道典型的模板题. 以下是模板写法.
1. 必须时刻注意, for循环里是i + 1不是index + 1. 我们当前取过后, 应该是处理下一个位置.
2. start应该是从1开始. 这个是这题的特别情况, 一般idnex是0开始.
3. combination的题目, 注意因为没有顺序, 所以为了避免重复的一些解, 我们只考虑递增的解.
*/
public class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> rst = new ArrayList<List<Integer>>();
        if (k == 0) {
            rst.add(new ArrayList<Integer>());
            return rst;
        }
        dfs(n, k, new ArrayList<Integer>(), rst, 1);
        return rst;
    }

    public void dfs(int n, int k, List<Integer> path, List<List<Integer>> rst, int start) {
        if (path.size() == k) {
            rst.add(new ArrayList<Integer>(path));
            return;
        }
        for (int i = start; i <= n; i++) {
            path.add(i);
```

```
            dfs(n, k, path, rst, i + 1);
            path.remove(path.size() - 1);
        }
    }
}
```

Combination Sum III
Find all possible combinations of k numbers that add up to a number n, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.
Ensure that numbers within the set are sorted in ascending order.
Example 1:
Input: k = 3, n = 7
Output:
[[1,2,4]]
Example 2:
Input: k = 3, n = 9
Output:
[[1,2,6], [1,3,5], [2,3,4]]
Answer:

```
public class Solution {
    public List<List<Integer>> combinationSum3(int k, int n) {
        List<List<Integer>> rst = new ArrayList<List<Integer>>();
        List<Integer> path = new ArrayList<Integer>();
        dfs(rst, 1, 0, k, n, path);
        return rst;
    }

    private void dfs(List<List<Integer>> rst, int start, int sum, int k, int n, List<Integer> path) {
        if (sum > n || path.size() > k) {
            return;
        }
        if (path.size() == k && sum == n) {
            rst.add(new ArrayList<Integer>(path));
        } else {
            for (int i = start; i <= 9; i++) {
                path.add(i);
                dfs(rst, i + 1, sum + i, k, n, path);
                path.remove(path.size() - 1);
            }
        }
    }
}
```

Word Ladder
Given two words (beginWord and endWord), and a dictionary, find the length of shortest transformation sequence from beginWord to endWord, such that:
Only one letter can be changed at a time
Each intermediate word must exist in the dictionary
For example,

Given:
start = "hit"
end = "cog"
dict = ["hot","dot","dog","lot","log"]
As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",
return its length 5.
Note:
Return 0 if there is no such transformation sequence.
All words have the same length.
All words contain only lowercase alphabetic characters.
Answer:

```
/*
经典的BFS题目
想象一下, 这个变换过程是一个树, 每一层是当前所有的变换结果, 下一层又是上一层的字符串的所有的变换结果.
e.g.
HIT
AIT, BIT, CIT, DIT...
HAT, HBT, HCT, HDT...
HIA, HIB, HIC, HID...
HIT可以有这么多种变换方式, 而AIT, BIT本身也可以以相同的方式展开, 这就形成了一个相当大的树
*/
/*
public StringBuilder(String str)
Constructs a string builder initialized to the contents of the specified string.
*/
public class Solution {
    public int ladderLength(String beginWord, String endWord, Set<String> wordDict) {
        if (beginWord == null || endWord == null || wordDict == null) {
            return 0;
        }

        Queue<String> q = new LinkedList<String>();
        q.offer(beginWord);
        HashSet<String> set = new HashSet<String>();
        set.add(beginWord);
        int level = 1;
        while (!q.isEmpty()) {
            int size = q.size();
            level++;
            for (int i = 0; i < size; i++) {
                String s = q.poll();
                int len = s.length();
                for (int j = 0; j < len; j++) {
                    StringBuilder sb = new StringBuilder(s);
                    for (char c = 'a'; c <= 'z'; c++) {
                        sb.setCharAt(j, c);
                        String tmp = sb.toString();
```

```
                    if (tmp.equals(endWord)) {
                        return level;
                    }
                    if (set.contains(tmp) || !wordDict.contains(tmp)) {
                        continue;
                    }
                    set.add(tmp);
                    q.offer(tmp);
                }
            }
        }
        return 0;
    }
}
```

Word Ladder II
Given two words (start and end), and a dictionary, find all shortest transformation sequence(s) from start to end, such that:
Only one letter can be changed at a time
Each intermediate word must exist in the dictionary
For example,
Given:
start = "hit"
end = "cog"
dict = ["hot","dot","dog","lot","log"]
Return

```
 [
   ["hit","hot","dot","dog","cog"],
   ["hit","hot","lot","log","cog"]
 ]
```

Note:
All words have the same length.
All words contain only lowercase alphabetic characters.
Answer:

```
public class WordLadder2Sol {
        public static List<List<String>> findLadders1(String start, String end, Set<String> dict) {
                List<List<String>> ret = new ArrayList<List<String>>();
                if (start == null || end == null || dict == null) {
                        return ret;
                }
                HashMap<String, List<List<String>>> map = new HashMap<String,
List<List<String>>>();
                HashMap<String, List<List<String>>> mapTmp = new HashMap<String,
List<List<String>>>();
                Queue<String> q = new LinkedList<String>();
                q.offer(start);
                List<List<String>> listStart = new ArrayList<List<String>>();
                List<String> path = new ArrayList<String>();
```

```
                path.add(start);
                listStart.add(path);
                map.put(start, listStart);
                while (!q.isEmpty()) {
                        int size = q.size();
                        for (int i = 0; i < size; i++) {
                                String s = q.poll();
                                int len = s.length();
                                for (int j = 0; j < len; j++) {
                                        StringBuilder sb = new StringBuilder(s);
                                        for (char c = 'a'; c <= 'z'; c++) {
                                                sb.setCharAt(j, c);
                                                String tmp = sb.toString();
                                                if ((!dict.contains(tmp) && !tmp.equals(end)) ||
map.containsKey(tmp)) {
                                                        continue;
                                                }
                                                List<List<String>> pre = map.get(s);
                                                List<List<String>> curList = mapTmp.get(tmp);
                                                if (curList == null) {
                                                        curList = new ArrayList<List<String>>();
                                                        q.offer(tmp);
                                                        mapTmp.put(tmp, curList);
                                                }
                                                for (List<String> strList : pre) {
                                                        List<String> strListNew = new
ArrayList<String>(strList);
                                                        strListNew.add(tmp);
                                                        curList.add(strListNew);
                                                }
                                        }
                                }
                        }
                        if (mapTmp.containsKey(end)) {
                                return mapTmp.get(end);
                        }
                        map.putAll(mapTmp);
                }
                return ret;
        }
        /*
        SOL 2
        bfs: find the shortest length, dfs: find all the shortest solutions
        distance 每个结点到起始结点的距离
        */
        public static List<List<String>> findLadders2(String start, String end, Set<String> dict) {
                List<List<String>> ret = new ArrayList<List<String>>();
                HashMap<String, List<String>> map = new HashMap<String, List<String>>();
                HashMap<String, Integer> distance = new HashMap<String, Integer>();
```

```
                dict.add(start);
                dict.add(end);
                bfs(map, distance, start, end, dict);
                List<String> path = new ArrayList<String>();
                dfs(map, distance, end, start, path, ret);
                return ret;
        }

        private static void bfs(HashMap<String, List<String>> map, HashMap<String, Integer>
distance, String start, String end, Set<String> dict) {
                Queue<String> q = new LinkedList<String>();
                q.offer(start);
                distance.put(start, 0);
                for (String s : dict) {
                        map.put(s, new ArrayList<String>());
                }
                while (!q.isEmpty()) {
                        String cur = q.poll();
                        List<String> nextList = expand(cur, dict);
                        for (String next : nextList) {
                                map.get(next).add(cur);
                                if (!distance.containsKey(next)) {
                                        distance.put(next, distance.get(cur) + 1);
                                        q.offer(next);
                                }
                        }
                }
        }

        private static void dfs(HashMap<String, List<String>> map, HashMap<String, Integer>
distance, String cur, String start, List<String> path, List<List<String>> ret) {
                path.add(cur);
                if (cur.equals(start)) {
                        Collections.reverse(path);
                        ret.add(new ArrayList<String>(path));
                        Collections.reverse(path);
                } else {
                        for (String next : map.get(cur)) {
                                if (distance.containsKey(next) && distance.get(cur) ==
distance.get(next) + 1) {
                                        dfs(map, distance, next, start, path, ret);
                                }
                        }
                }
                path.remove(path.size() - 1);
        }

        private static List<String> expand(String cur, Set<String> dict) {
                List<String> expansion = new ArrayList<String>();
```

```
                for (int i = 0; i < cur.length(); i++) {
                        for (char c = 'a'; c <= 'z'; c++) {
                                if (c != cur.charAt(i)) {
                                        String expanded = cur.substring(0, i) + c + cur.substring(i +
1);
                                        if (dict.contains(expanded)) {
                                                expansion.add(expanded);
                                        }
                                }
                        }
                }
                return expansion;
        }

        public static void main(String[] args) {
                String start = "hit";
                String end = "cog";
                Set<String> dict = new HashSet<String>();
                dict.add("hot");
                dict.add("dot");
                dict.add("dog");
                dict.add("lot");
                dict.add("log");
                System.out.print(findLadders2(start, end, dict));
        }
}
```

Word Search
Given a 2D board and a word, find if the word exists in the grid.
The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.
For example,
Given board =

```
[
  ["ABCE"],
  ["SFCS"],
  ["ADEE"]
]
```

word = "ABCCED", -> returns true,
word = "SEE", -> returns true,
word = "ABCB", -> returns false.
Answer:

```
/*
SOL 1
recursition and backtracking
m X n is board size, k is word size.
m * n * 4^(k-1) runtime
recuision最深是k层, recursive部分空间复杂度应该是O(k) + O(m * n) (visit array)
```

SOL 2
在进入DFS后, 把访问过的节点置为'#', 访问完毕之后再置回来即可, 可以省掉O(m * n)的空间复杂度
*/

```
public class WordSearchSol {
	public static boolean exist1(char[][] board, String word) {
		if (board == null || word == null || board.length == 0 || board[0].length == 0) {
			return false;
		}
		int rows = board.length;
		int cols = board[0].length;
		boolean[][] visit = new boolean[rows][cols];
		for (int i = 0; i < rows; i++) {
			for (int j = 0; j < cols; j++) {
				if (dfs1(board, i, j, word, 0, visit)) {
					return true;
				}
			}
		}
		return false;
	}

	public static boolean dfs1(char[][] board, int i, int j, String word, int wordIndex, boolean[][] visit) {
		int rows = board.length;
		int cols = board[0].length;
		int len = word.length();
		if (wordIndex >= len) {
			return true;
		}
		if (i < 0 || i >= rows || j < 0 || j >= cols) {
			return false;
		}
		if (word.charAt(wordIndex) != board[i][j]) {
			return false;
		}
		if (visit[i][j] == true) {
			return false;
		}
		visit[i][j] = true;
		if (dfs1(board, i + 1, j, word, wordIndex + 1, visit)) {
			return true;
		}
		if (dfs1(board, i - 1, j, word, wordIndex + 1, visit)) {
			return true;
		}
		if (dfs1(board, i, j - 1, word, wordIndex + 1, visit)) {
			return true;
		}
```

```
		if (dfs1(board, i, j + 1, word, wordIndex + 1, visit)) {
			return true;
		}
		visit[i][j] = false;
		return false;
	}

	public static boolean exist2(char[][] board, String word) {
		if (board == null || board.length == 0 || board[0].length == 0) {
			return false;
		}
		int rows = board.length;
		int cols = board[0].length;
		for (int i = 0; i < rows; i++) {
			for (int j = 0; j < cols; j++) {
				if (dfs2(board, word, i, j, 0)) {
					return true;
				}
			}
		}
		return false;
	}

	public static boolean dfs2(char[][] board, String word, int i, int j, int index) {
		int len = word.length();
		if (index >= len) {
			return true;
		}
		if (i < 0 || i >= board.length || j < 0 || j >= board.length) {
			return false;
		}
		if (word.charAt(index) != board[i][j]) {
			return false;
		}
		char tmp = board[i][j];
		board[i][j] = '#';
		boolean ret = dfs2(board, word, i + 1, j, index + 1)
				|| dfs2(board, word, i - 1, j, index + 1)
				|| dfs2(board, word, i, j + 1, index + 1)
				|| dfs2(board, word, i, j - 1, index + 1);
		board[i][j] = tmp;
		return ret;

	}

	public static void main(String[] args) {
		char[][] board = {{'A', 'B', 'C', 'E'}, {'S', 'F', 'C', 'S'}, {'A', 'D', 'E', 'E'}};
		String word = "ABCB";
		System.out.print(exist2(board, word));
```

```
        }
}
```

Word Search II
Given a 2D board and a list of words from the dictionary, find all words in the board.
Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.
For example,
Given words = ["oath","pea","eat","rain"] and board =

```
[
  ['o','a','a','n'],
  ['e','t','a','e'],
  ['i','h','k','r'],
  ['i','f','l','v']
]
```

Return ["eat","oath"].
Note:
You may assume that all inputs are consist of lowercase letters a-z.
You would need to optimize your backtracking to pass the larger test. Could you stop backtracking earlier?
If the current candidate does not exist in all words' prefix, you could stop backtracking immediately. What kind of data structure could answer such query efficiently? Does a hash table work? Why or why not? How about a Trie? If you would like to learn how to implement a basic trie, please work on this problem: Implement Trie (Prefix Tree) first.
Answer:

```
public class WordSearch2Sol {
    public static List<String> findWords(char[][] board, String[] words) {
        List<String> rst = new ArrayList<String>();
        Trie trie = new Trie();
        for (String word : words) {
            trie.insert(word);
        }
        boolean[][] visit = new boolean[board.length][board[0].length];
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if (trie.root == null) {
                    return rst;
                }
                search(board, i, j, rst, visit, trie.root, trie);
            }
        }
        return rst;
    }

    private static void search(char[][] board, int i, int j, List<String> rst, boolean[][] visit, TrieNode
node, Trie trie) {
        if (node == null) {
            return;
```

```
        }
        int c = board[i][j] - 'a';
        TrieNode nextNode = node.children[c];
        if (nextNode == null) {
            return;
        }
        if (nextNode.have) {
            rst.add(nextNode.word);
            trie.remove(nextNode.word);
        }
        int row = board.length;
        int col = board[0].length;
        visit[i][j] = true;
        if (i + 1 < row && !visit[i + 1][j]) {
            search(board, i + 1, j, rst, visit, nextNode, trie);
        }
        if (i - 1 >= 0 && !visit[i - 1][j]) {
            search(board, i - 1, j, rst, visit, nextNode, trie);
        }
        if (j + 1 < col && !visit[i][j + 1]) {
            search(board, i, j + 1, rst, visit, nextNode, trie);
        }
        if (j - 1 >= 0 && !visit[i][j - 1]) {
            search(board, i, j - 1, rst, visit, nextNode, trie);
        }
        visit[i][j] = false;
    }

    public static class TrieNode {
        boolean have;
        TrieNode[] children;
        String word;
        public TrieNode() {
            have = false;
            children = new TrieNode[26];
        }
    }

    public static class Trie {
        private TrieNode root;

        public Trie() {
            root = new TrieNode();
        }

        public void insert(String word) {
            TrieNode cur = root;
            int len = word.length();
            for (int i = 0; i < len; i++) {
```

```
            int c = word.charAt(i) - 'a';
            if (cur.children[c] == null) {
                cur.children[c] = new TrieNode();
            }
            cur = cur.children[c];
        }
        cur.have = true;
        cur.word = word;
    }

    public void remove(String word) {
        TrieNode cur = root;
        int len = word.length();
        for (int i = 0; i < len; i++) {
            int c = word.charAt(i) - 'a';
            TrieNode parent = cur;
            cur = cur.children[c];
        }
        cur.have = false;
        cur.word = null;
    }
}

public static void main(String[] args) {
        String[] words = {"oath", "pea", "eat", "rain"};
        char[][] board = {
                                    {'o','a','a','n'},
                                    {'e','t','a','e'},
                                    {'i','h','k','r'},
                                    {'i','f','l','v'}
                                };
        System.out.print(findWords(board, words));
    }
}
```