backtracking的三种基本类型：

回溯是一种穷举，但与brute force有一些区别，回溯带了两点脑子的，并不多，brute force一点也没带。

第一点脑子是回溯知道**回头**；相反如果是brute force,发现走不通立刻跳下山摔死，换第二条命从头换一条路走。

第二点脑子是回溯知道**剪枝**；如果有一条岔路上放了一坨屎，那这条路我们不走，就可以少走很多不必要走的路。

**还有一些爱混淆的概念：递归，回溯，DFS。**  
回溯是一种找路方法，搜索的时候**走不通就回头换路接着走**，**直到走通了或者发现此山根本不通。**  
DFS是一种开路策略，就是一条道先走到头，再往回走一步换一条路走到头，这也是回溯用到的策略。在树和图上回溯时人们叫它DFS。

递归是一种行为，回溯和递归如出一辙，都是一言不合就回到来时的路，所以一般回溯用递归实现；当然也可以不用，用栈stack。

**判断是否用backtracking：**

拿到一个问题，你感觉如果不穷举一下就没法知道答案，那就可以开始回溯了

**一般的问题有三种：**

1. Find a path to success 有没有解， 返回true/false
2. Find all paths to success 求所有解
   * 求所有解的个数
   * 求所有解的具体信息
3. Find the best path to success 求最优解

**把实际题目归类为backtrack的问题，需要考虑清楚：**

1. whether leaf node: 什么情况下会被判断为leaf node，也就是track终结；
2. whether goal node: 到达leaf的情况下，什么情况会被判定为goal – 为有效解。
3. 尤其是第二类/第三类问题：如何track中间状态，常用的是用个collection, 以及cur index.

第一类问题：有没有解/只用返回一个解

backtrack返回的是boolean, 可以用于判断是否找到，找到后直接返回，不再继续

后两类返回的都是void，因为需要全部走完

boolean backtrack(Node n) {

if (n is leaf node) {

if (n is goal node) return true;

return false;

} else {

for (each child of n) {

if (backtrack(child)) return true;

}

return false;

}

}

第二类问题：求所有解的信息

Collectio solve(Node n) {

*// handle corner case*

Collection res = new Collections;

backtrack(res, new Collection, n);

return res;

}

void backtrack(Collection res, Collection cur, Node n) {

if (n is leaf node) {

if (n is goal node) add cur to res, return;

else return;

} else {

for (each child of n) {

*// make changes to cur, which is related with node*

backtrack(child, cur, res);

*// remove changed that made beofre*

}

}

}

第三类问题： 求最优解

Optimize solve(Node n) {

Optimize best = initialize.

backtrack(n, best);

return best;

}

void backtrack(Node n, Optimize best) {

if (n is leaf node) {

if (n is goal node) update best result, return;

else return;

} else {

for (each child of n) {

backtrack(child, best);

}

}

}

77. 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:



**S1 Backtrack** – 第二类问题，求所有解

public List<List<Integer>> combine(int n, int k) {

List<List<Integer>> res = new ArrayList<>();

backtrack(res, new ArrayList<>(), k, 1, n);

return res;

}

// k是所需的数字，与list.size()对比，用于判断终止条件，list存放走到当前用到的数字

public void backtrack(List<List<Integer>> res, List<Integer> list, int k, int start, int end) {

if (list.size() == k) {

res.add(new ArrayList<>(list));

return;

}

for (int i = start;i <= end;i++) {

list.add(i);

backtrack(res, list, k, i + 1, n);

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

}

}

39. Combination Sum

Given a set of candidate numbers (C) (without duplicates) 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.

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]]

**S1. backtrack**

终止条件的判断依赖： all numbers are positive，所以一旦target为负就可以停止。

需要判断两种终止条件：找到合适的sum，或者此路不通，不继续走。

public List<List<Integer>> combinationSum(int[] candidates, int target) {

List<List<Integer>> res = new ArrayList<>();

dfs(res, new ArrayList<>(), candidates, target, 0);

return res;

}

// list:到目前为止加过的值，target:目标值；curIdx:目前走到哪

private void dfs(List<List<Integer>> res, List<Integer> list,

int[] candidates, int target, int curIdx) {

if (target < 0) { // 这条路走不通

return;

} else if (target == 0) { // 找到了合适的

res.add(new ArrayList<>(list));

return;

} else { // 因为每个可以选无限次，所以从curIdx开始

for (int i = curIdx;i < candidates.length;i++) {

list.add(candidates[i]);

dfs(res, list, candidates, target - candidates[i], i);

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

}

}

}

40. Combination Sum II

和上一道题的区别是：**Each number in C may only be used once in the combination.**

但是难点在于，C中本身可能含有重复的数字，要保证同一个大小的数字只出现一次，比如下面的例子：

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]]

需要保证[1,2,5]只出现一次，而不会因为有两个1而出现[1,2,5], [2,1,5]两次。

**S1. backtrack + sort**

有重复数字，但又要保证每个数字只出现一次：sort，跳过重复的。

先把输入的array排序，这样重复的数字会排在一起，

然后在handle的时候通过while的判断，来避免在同一个位置选两次一样的数值。

public List<List<Integer>> combinationSum2(int[] candidates, int target) {

List<List<Integer>> result = new ArrayList<>();

Arrays.sort(candidates);

dfs(result, new ArrayList<>(), candidates, target, 0, 0);

return result;

}

private void dfs(List<List<Integer>> result, List<Integer> list, int[] candidates, int target, int sum, int idx) {

if (sum > target) return;

if (sum == target) {

result.add(new ArrayList<>(list));

return;

}

int len = candidates.length;

for (int i = idx;i < len;i++) {

list.add(candidates[i]);

dfs(result, list, candidates, target, sum + candidates[i], i + 1);

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

// make sure the same value won’t appear at the same place. won’t be two [1,2,5]

while (i + 1 < len && candidates[i] == candidates[i + 1]) i++;

}

}

216. 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.

Input: k = 3, n = 7，Output: [[1,2,4]]

Input: k = 3, n = 9，Output: [[1,2,6], [1,3,5], [2,3,4]]

题目就变成了从1-9中选k个数字，组成n

public List<List<Integer>> combinationSum3(int k, int n) {

List<List<Integer>> result = new ArrayList<>();

dfs(result, new ArrayList<>(), k, n, 0, 1);

return result;

}

private void dfs(List<List<Integer>> result, List<Integer> list,

int k, int n, int sum, int idx) {

if (sum > n || n < 0) return;

if (sum == n && k == 0) {

result.add(new ArrayList<>(list));

return;

}

for (int i = idx;i < 10;i++) {

list.add(i);

dfs(result, list, k - 1, n, sum + i, i + 1);

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

}

}

377. Combination Sum IV

Given an integer array with **all positive numbers and no duplicates,** find the number of possible combinations that add up to a positive integer target.

nums = [1, 2, 3]，target = 4，The possible combination ways are:

(1, 1, 1, 1)

(1, 1, 2)

(1, 2, 1)

(1, 3)

(2, 1, 1)

(2, 2)

(3, 1)

Note that different sequences are counted as different combinations. Therefore the output is 7.

这道题的区别是：不同的顺序会当成不同的result，同样的数字，可以选无限次。

同样的数字，不同的排列组合顺序，会被看成不同的result

**S1 DP**

这其实是背包啊！

*// dp[i]: # of ways sum up to i*

public int combinationSum4(int[] nums, int target) {

int[] dp = new int[target + 1];

int n = nums.length;

dp[0] = 1;

for (int i = 1;i <= target;i++) {

for (int j = 0;j < n;j++) {

if (i-nums[j] >= 0) dp[i] += dp[i - nums[j]];

}

}

return dp[target];

}

78. Subsets

Given a set **of distinct integers**, nums, return **all possible subsets**.

Note: 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],

[]

每个数字都有：加，不加，这两种情况

题解：

这道题目的重点是，返回任意0-n个数目的组合，重点是组合，顺序不重要，顺序不同仍然算是一个。

所以不需要终止条件，直接加入res即可。

public List<List<Integer>> subsets(int[] nums) {

List<List<Integer>> res = new ArrayList<>();

if (nums == null) return res;

dfs(res, new ArrayList<>(), nums, 0);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> list,

int[] nums, int curIdx) {

res.add(new ArrayList<>(list));

for (int i = curIdx;i < nums.length;i++) {

list.add(nums[i]);

dfs(res, list, nums, i + 1);

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

}

}

90. Subsets II

Given a collection of integers that **might contain duplicates**, nums, return all possible subsets.

Note: The solution set must not contain duplicate subsets.

For example, If nums = [1,2,2], a solution is:

[2],

[1],

[1,2,2],

[2,2],

[1,2],

[]

也就是重复的只会被加入一次，不会加入多个[2]

**S1 backtrack**

挺简单的，和前面的combination sum, permutation II 一样，先sort, 然后在for循环里面加while，把一样的跳过。

public List<List<Integer>> subsetsWithDup(int[] nums) {

List<List<Integer>> res = new ArrayList<>();

if (nums == null) return res;

Arrays.sort(nums);

dfs(res, new ArrayList<>(), nums, 0);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> list, int[] nums, int curIdx) {

res.add(new ArrayList<>(list));

for (int i = curIdx;i < nums.length;i++) {

list.add(nums[i]);

dfs(res, list, nums, i + 1);

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

while (i < nums.length - 1 && nums[i] == nums[i + 1]) i++;

}

}

Permutations 类，subset类与combination类的区别：

给一个输入的array：

permutation返回array中数字不同的排列组合，要用上所有数字；

subset返回array中所有数字都有取，不取两种情况的所有排列组合，用上的数字从0-所有。

combination返回array中数字满足某个条件的所有解。

- backtrack的终止条件不同，

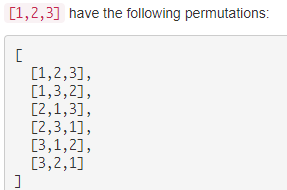
- combination的终止条件是len == k, permutation是len == nums.length;

并且因为对于 combination，k < nums.length，所以需要传入index，表示计算到哪里，

而permutation不用，因为每个有效的list都会包含nums中的所有数字。

46. Permutations

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



**S1 非常经典的backtrack**

第二类，求所有解，没有重复

public List<List<Integer>> permute(int[] nums) {

List<List<Integer>> list = new ArrayList<>();

if (nums == null || nums.length == 0) return list;

helper(list, new ArrayList<>(), nums);

return list;

}

private void helper(List<List<Integer>> list, List<Integer> cur, int[] nums) {

if (cur.size() == nums.length) {

list.add(new ArrayList<>(cur));

return;

}

int len = nums.length;

for (int i = 0;i < len;i++) {

if (cur.contains(nums[i])) continue;

cur.add(nums[i]);

helper(list, cur, nums);

cur.remove(cur.size() - 1);

}

}

**S2. backtrack 插空**

每次把curIdx对应的数字插入到所有可能的对应位置。

public List<List<Integer>> permute(int[] nums) {

List<List<Integer>> res = new ArrayList<>();

backtrack(res, new ArrayList<>(), nums, 0);

return res;

}

private void backtrack(List<List<Integer>> res, List<Integer> cur, int[] nums, int curIdx) {

if (curIdx == nums.length) {

res.add(new ArrayList<>(cur));

return;

}

int size = cur.size();

for (int i = 0;i <= size;i++) {

if (i != size) cur.add(i, nums[curIdx]);

else cur.add(nums[curIdx]);

backtrack(res, cur, nums, curIdx + 1);

cur.remove(i);

}

}

47. 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],

[2,1,1]

**S1. Backtrack 处理重复情况**

这道题和上一道题的区别就在于有重复数字存在，需要确定[1,1,2]不会出现两次。

如果按照上一道题的做法，只要index不同，就会被当做不同的情况处理，所以会出现两次。

所以我们可以先sort，然后在for循环跳过所有相同的值。

public List<List<Integer>> permuteUnique(int[] nums) {

List<List<Integer>> list = new ArrayList<>();

Arrays.sort(nums);

backtrack(list, new ArrayList<>(), nums, new boolean[nums.length]);

return list;

}

public void backtrack(List<List<Integer>> list, List<Integer> cur, int[] nums, boolean[] used) {

if (cur.size() == nums.length) {

list.add(new ArrayList<>(cur));

} else {

int n = nums.length;

for (int i = 0;i < n;i++) {

if (used[i]) continue;

cur.add(nums[i]);

used[i] = true;

backtrack(list, cur, nums, used);

cur.remove(cur.size() - 1);

used[i] = false;

while (i < n - 1 && nums[i + 1] == nums[i]) i++;

*// skip all the same value, so there won't be same value*

*// appear at this position.*

*// there is one additional i++ in for loop.*

}

}

}

526. Beautiful Arrangement

Suppose you have N integers from 1 to N. We define a beautiful arrangement as an array that is constructed by these N numbers successfully if one of the following is true for the ith position (1 <= i <= N) in this array:

- The number at the ith position is divisible by i.

- i is divisible by the number at the ith position.

Now given N, **how many** beautiful arrangements can you construct?

**S1. Backtrack**

需要求的是count，只是这里的限制条件，在针对能否走向下一步的限制条件改变了，不仅有是否visited before, 还添加了 pos % i == 0 || i % pos == 0

public int countArrangement(int N) {

int[] count = new int[]{0};

backtrack(N, 1, new boolean[N + 1], count);

return count[0];

}

// used: index using: 1-N，表示1-N对应的数字是否已被使用

private void backtrack(int N, int pos, boolean[] used, int[] count) {

if (pos > N) {

count[0]++;

return;

}

for (int i = 1;i <= N;i++) {

if (used[i] == false && (pos % i == 0 || i % pos == 0)) {

used[i] = true;

backtrack(N, pos + 1, used, count);

used[i] = false;

}

}

}

22. Generate Parentheses

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

For example, given n = 3, a solution set is:



**S1 Backtrack**

最外面的两个一定是(, ), 不需要考虑，

终止条件和是否合理的判断条件，是左括号和右括号剩余需要出现的次数。

实际上需要handle的是n-1个左括号和n-1个右括号，并且**在过程中需要保证左括号的个数>= 右括号**，

避免出现”())(()” 这样的情况。

public List<String> generateParenthesis(int n) {

List<String> res = new ArrayList<>();

backtrack(res, new StringBuilder("("), n - 1, n - 1);

return res;

}

*// left, right: # of parenthesis haven't been inserted yet.*

private void backtrack(List<String> res, StringBuilder sb,

int left, int right) {

if (left == 0 && right == 0) {

res.add(sb.toString() + ")");

return;

}

int len = sb.length();

if (left != 0) {

backtrack(res, sb.append("("), left - 1, right);

}

sb.setLength(len); // Set to previous condition

*// the condition here is IMPORTANT! Can't be "right != 0"*

*// That may leads to "())(()" condition, to avoid this, need to make sure left <= right*

if (left <= right) {

backtrack(res, sb.append(")"), left, right - 1);

}

}

17. 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.



Input:Digit string "23" Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

**S1 Backtrack**

这道题其实可以看作变种的combination，每个数字和一些char对应，在找够字符之后就停止，返回。

public List<String> letterCombinations(String digits) {

List<String> result = new ArrayList<String>();

if (digits == null || digits.isEmpty()) {

return result;

}

String[] chars = new String[]{"","","abc","def","ghi","jkl","mno",

"pqrs","tuv","wxyz"};

dfs(result, digits, new StringBuilder(), chars);

return result;

}

private void dfs(List<String> result, String digits, StringBuilder sb, String[] chars) {

if (sb.length() == digits.length()) {

result.add(sb.toString());

return;

}

int digit = digits.charAt(sb.length()) - '0';

char[] letters = chars[digit].toCharArray();

for (char c : letters) {

sb.append(c);

dfs(result, digits, sb, chars);

sb.deleteCharAt(sb.length() - 1);

}

}

425. Word Squares

Given a set of words (without duplicates), find all word squares you can build from them.

A sequence of words forms a valid word square if the kth row and column read the exact same string, where 0 ≤ k < max(numRows, numColumns).

For example, the word sequence ["ball","area","lead","lady"] forms a word square because each word reads the same both horizontally and vertically.



638. Shopping Offers

Input: [2,5], [[3,0,5],[1,2,10]], [3,2]

Output: 14

There are two kinds of items, A and B. Their prices are $2 and $5 respectively.

In special offer 1, you can pay $5 for 3A and 0B

In special offer 2, you can pay $10 for 1A and 2B.

You need to buy 3A and 2B, so you may pay $10 for 1A and 2B (special offer #2), and $4 for 2A.

You could use any of special offers as many times as you want.

**S1.Backtracking**

遍历每一个special offer，更新needs, 尝试是否可以使用，如果可以则用recursion, 调用shoppingOffers并传入新的needs，

每次recursion都尝试“采用special offer”和“不采用”的两种形式，并选取min.

time: exponential. 假设有m个不同的item，n个不同的special offer，n\*n\*n....

worst when all offers valid.

public int shoppingOffers(List<Integer> price, List<List<Integer>> special, List<Integer> needs) {

int result = Integer.MAX\_VALUE;

for(int i = 0; i < special.size(); i++) {

// 1.check offer能否使用

List<Integer> offer = special.get(i);

boolean valid = true;

for(int j = 0; j < needs.size(); j++) { *// needs-offer for each item*

int remain = needs.get(j) - offer.get(j);

needs.set(j, remain);

if(remain < 0) valid = false; *// items > needs*

}

// 2.使用offer, backtrack

if(valid) result = Math.min(result,

shoppingOffers(price, special, needs) + offer.get(needs.size()));

// 3. reset，重设为使用offer之前的值

for(int j = 0; j < needs.size(); j++) { *// reset the needs*

needs.set(j, needs.get(j) + offer.get(j));

}

}

int nonOfferPrice = 0;

for(int i = 0; i < needs.size(); i++) {

nonOfferPrice += price.get(i) \* needs.get(i);

}

return Math.min(result, nonOfferPrice);

}

**Pattern Matching**

10. Regular Expression Matching

Implement regular expression matching with support for '.' and '\*'.

'.' Matches **any single character**.

'\*' Matches **zero or more of the preceding element.**

**S1 DP**

注意\* match的是0+ 上一个字符。

dp[i][j]: s.substring(0, i)与p.substring(0, j) 是否match。

p[j] 为常规字母：if s[i]==p[j] && dp[i-1][j-1]==true: true

p[j] 为'.':if dp[i-1][j-1]==true:match

p[j] 为'\*': 看前一个是不是'.':

前一个是'.' : 这个'\*'有可能match一个或多个：dp[i-1][j]

有可能不match，只有上一个'.'match：dp[i][j-1]

有可能它和'.'都不match任何字符：dp[i][j-2]

前一个是正常字符：'\*'位置能否匹配取决于dp[i][j-2]

public boolean isMatch(String s, String p) {

if (s == null || p == null) return false;

int m = s.length(), n = p.length();

boolean[][] dp = new boolean[m + 1][n + 1];

dp[0][0] = true;

for (int i = 2;i <= n;i++) { // 初始化是为了在第一个char为\*时，可以正常匹配

if (p.charAt(i-1) == '\*') dp[0][i] = dp[0][i-2];

}

for (int i = 1;i <= m;i++) {

for(int j = 1;j <= n;j++) {

char chars = s.charAt(i-1);

char charp = p.charAt(j-1);

if (charp == '.' || chars == charp) {

dp[i][j] = dp[i-1][j-1];

} else if (charp == '\*') {

char prevp = p.charAt(j - 2);

*// 这里prevp==chars 和 prevp=='.'的效果是一样的，都是前一个匹配*

if (prevp == '.' || prevp == chars) {

dp[i][j] = dp[i-1][j] || dp[i][j-2] || dp[i][j-1];

} else { *// 前一个不match，但是由于'\*'的存在,还有可能是true,要看j-2的表现*

dp[i][j] = dp[i][j-2];

}

}

}

}

return dp[m][n];

}

**S2 Recursion**

理论和前面是一样的，只是采用recursion的方法来写

public boolean isMatch(String s, String p) {

if (s == null || p == null) return false;

if (p.length() == 0) return s.length() == 0;

int slen = s.length(), plen = p.length();

if (plen == 1 || p.charAt(1) != '\*') {

if (slen == 0) return false;

if (p.charAt(0) != s.charAt(0) && p.charAt(0) != '.')

return false;

return isMatch(s.substring(1), p.substring(1));

}

*// when second char is '\*'*

*// first 2 chars in p don't match anything*

if (isMatch(s, p.substring(2))) return true;

*// '\*' match 1+ preceding chars*

int i = 0;

while (i < slen && (s.charAt(i) == p.charAt(0) || p.charAt(0) == '.')) {

if (isMatch(s.substring(i + 1), p.substring(2))) return true;

i++;

}

return false;

}

44. Wildcard Matching

Implement wildcard pattern matching with support for '?' and '\*'.

'?' Matches any single character.

'\*' Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

**S1 DP**

这道题的重点是怎么分析问题，和怎么构造dp，dp[i][j]代表什么比较好，

两个string从小到大生长，每一步进行match的判断，dp[i][j] 表示s.substring[0, i], p.substring[0, j]是否match

接下来的问题是怎么识别所有情况，通过p[j]判断：

p[j] 为常规字母：if s[i]==p[j] && dp[i-1][j-1]==true: true

p[j] 为'?':if dp[i-1][j-1]==true:match

p[j] 为'\*': '\*'在这里具体match什么，有三种情况：

match当前这个字符，同'?': if dp[i-1][j-1]==true: true

不match任何字符：if dp[i][j-1]== true: true 忽略p[j]

match 多个字符：看dp[i-1][j]

注意初始化，不然s = “”, p = “\*” 这样的match不过去

public boolean isMatch(String s, String p) {

if (s == null || p == null) return false;

int m = s.length(), n = p.length();

boolean[][] dp = new boolean[m + 1][n + 1];

dp[0][0] = true;

for (int i = 1;i <= n;i++) { *// if s = "", p="\*\*\*"*

if (p.charAt(i-1) == '\*') {

dp[0][i] = true;

} else {

break;

}

}

for (int i = 1;i <= m;i++) {

for (int j = 1;j <= n;j++) {

*// 还是需要注意j-1, 对字符串的position总是比array的小1*

if (s.charAt(i-1) == p.charAt(j-1) || p.charAt(j-1) == '?') {

dp[i][j] = dp[i-1][j-1];

} else if (p.charAt(j-1) == '\*') {

dp[i][j] = dp[i-1][j] || dp[i][j-1] || dp[i-1][j-1];

}

}

}

return dp[m][n];

}

291. Word Pattern II

Given a pattern and a string str, find if str follows the same pattern.

Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty substring in str. Examples:

pattern = "abab", str = "redblueredblue" should return true.

pattern = "aaaa", str = "asdasdasdasd" should return true.

pattern = "aabb", str = "xyzabcxzyabc" should return false.

Notes: You may assume both pattern and str contains only lowercase letters.

**S Backtracking / DFS**

和之前的word pattern一样，<character, string>必须uniquely match to each other

这题比之前的题目的复杂在于，str没有进行划分。char从哪map到哪，需要尝试。

*/\*\**

*\* Use set: make sure one string won't match several patterns,*

*\* and 1 pattern, 1 substring should be uniquely match to each other*

*\*/*

public boolean wordPatternMatch(String pattern, String str) {

Map<Character, String> map = new HashMap<>();

Set<String> matched = new HashSet<>();

return isMatch(pattern, str, 0, 0, map, matched);

}

*// i, j: current index of pat and str.*

private boolean isMatch(String pat, String str, int i, int j, Map<Character, String> map, Set<String> matched) {

if (i == pat.length() && j == str.length()) return true;

if (i >= pat.length() || j >= str.length()) return false;

char curChar = pat.charAt(i);

if (map.containsKey(curChar)) {

String s = map.get(curChar);

if (!str.startsWith(s, j)) return false;

return isMatch(pat, str, i + 1, j + s.length(), map, matched);

}

*// the pattern char doesn't exist in the map,*

*// try to match from the beginning*

for (int k = j;k < str.length();k++) {

String tmp = str.substring(j, k + 1);

if (matched.contains(tmp)) continue;*// substring exist in set*

map.put(curChar, tmp);

matched.add(tmp);

if (isMatch(pat, str, i + 1, k + 1, map, matched)) return true;

map.remove(curChar);

matched.remove(tmp);

}

return false;

}

267. Palindrome Permutation II

Given a string s, return all the palindromic permutations (without duplicates) of it. Return an empty list if no palindromic permutation could be form.

For example:

Given s = "aabb", return ["abba", "baab"].

Given s = "abc", return [].

**Number of Ways**

351. Android Unlock Patterns

Given an Android 3x3 key lock screen and two integers m and n, where 1 ≤ m ≤ n ≤ 9, count the total number of unlock patterns of the Android lock screen, which consist of minimum of m keys and maximum n keys.

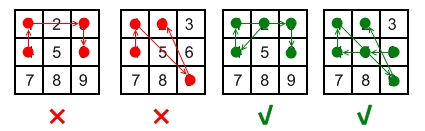
Rules for a valid pattern:

1.Each pattern must connect at least m keys and at most n keys.

2.All the keys must be distinct.

3.If the line connecting two consecutive keys in the pattern passes through any other keys, the other keys must have previously selected in the pattern. No jumps through non selected key is allowed.

4.The order of keys used matters.



**S1.DFS**

需要保存两个信息：有哪些点是不直接相连的，并且从点a到b会经过哪个点。

便于判断路径能否走通。

用jump[i][j]保存从i到j需要穿过的点。

visited 判断各个数字是否已经走过，num判断当前在哪个数字上，len表示# of keys connect.

public int numberOfPatterns(int m, int n) {

boolean[] visited = new boolean[10];

int[][] jump = new int[10][10];

jump[1][3] = jump[3][1] = 2;

jump[4][6] = jump[6][4] = 5;

jump[7][9] = jump[9][7] = 8;

jump[1][7] = jump[7][1] = 4;

jump[2][8] = jump[8][2] = 5;

jump[3][9] = jump[9][3] = 6;

jump[1][9] = jump[9][1] = 5;

jump[3][7] = jump[7][3] = 5;

*// 1, 3, 7, 9 are the same, so just check 1 and \*4*

*// 2, 4, 6, 8 are the same as well, just check 2 and \*4*

*// then check 5*

return 4\*dfs(jump, visited, 1, 1, m, n) + 4\*dfs(jump, visited, 2, 1, m, n) + dfs(jump, visited, 5, 1, m, n);

}

*//num is current number, len is current length.*

int dfs(int[][] jump, boolean[] visited, int num, int len, int m, int n) {

int count = 0;

if (len >= m) count++;

if (len >= n) return count;

visited[num] = true;

len++;

for (int i = 1;i <= 9;i++) {

if (!visited[i] && (jump[num][i] == 0 || visited[jump[num][i]])) {

count += dfs(jump, visited, i, len, m, n);

}

}

visited[num] = false;

return count;

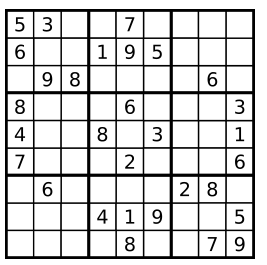
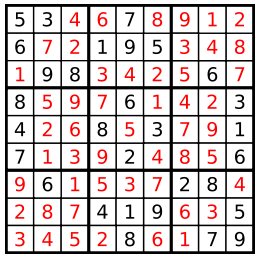
}

37. Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells.

Empty cells are indicated by the character '.'.

You may assume that there will be only one unique solution.

**S1 DFS**

虽然是求解数独，但本质上还是backtrack的第一种情况：求是否有解/只求一种解。

所以用dfs作为helper function，返回的是boolean, 一旦为true可以立即返回。

求解数独要考虑的情况：

对于每一行，每一列，每个block，数字1-9都只能出现一次。

所以用boolean[9][9] row, col, blk来表示每行，每列和每个block中已经出现过的数字，出现标true

分为两个步骤：

1. 根据目前matrix中的情况，更新row, col, blk的值，
2. 分为81个小格子，对于每个目前为空的格子都一个个试，试所有可能的数字。

public void solveSudoku(char[][] board) {

boolean[][] row = new boolean[9][9];

boolean[][] col = new boolean[9][9];

boolean[][] blk = new boolean[9][9];

for (int i = 0;i < 9;i++) {

for (int j = 0;j < 9;j++) {

if (board[i][j] == '.') continue;

int cur = board[i][j] - '1';

row[i][cur] = col[j][cur] = blk[i/3 + (j/3)\*3][cur] = true;

}

}

dfs(board, row, col, blk, 0);

}

private boolean dfs(char[][] board, boolean[][] row, boolean[][] col, boolean[][] blk, int idx) {

*// check leaf condition:*

if (idx == 81) return true;

int x = idx / 9; // row index

int y = idx % 9; // col index

int blockIdx = x/3 + (y/3)\*3; // block index

*// already processed before:*

if (board[x][y] != '.') return dfs(board, row, col, blk, idx + 1);

*// try all the possible solutions:*

for (char c = '1';c <= '9';c++) {

int i = c - '1';

*// this number already exists in col/column/block*

if (row[x][i] || col[y][i] || blk[blockIdx][i]) continue;

*// set it to new value, check if it works*

board[x][y] = c;

row[x][i] = col[y][i] = blk[blockIdx][i] = true;

if (dfs(board, row, col, blk, idx + 1)) return true;

*// set the original value back*

board[x][y] = '.';

row[x][i] = col[y][i] = blk[blockIdx][i] = false;

}

return false;

}

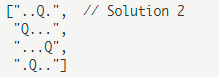
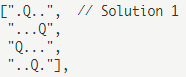
51. N-Queens

The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other.

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:



**S1 Backtrack**

本质上是backtrack，并且是第二类问题，求所有解的具体信息，这里有两个重点

1. 怎么把题目转换成backtracking的基本形式；
2. 如何判断某一步是否是valid，如果可以判断，就只在valid的情况下继续往下走。

1：根据题意，每一行只能有一个Q，所以用int[] queens来表示从第0行到n-1行每行的queue所在的列的index。

所以从第0行开始填queues, 最后一位填满的时候，就相当于leaf node

2：利用index trick – queues array已经可以保证每行只有一个Queue, 还需要检查每列，以及斜向是否有queen。

public List<List<String>> solveNQueens(int n) {

List<List<String>> res = new ArrayList<>();

int[] queens = new int[n];

if (n <= 0) return res;

dfs(res, queens, n, 0);

return res;

}

private void dfs(List<List<String>> res, int[] queens, int n, int cur) {

if (cur == n) { *// reached last line*

List<String> list = new ArrayList<>();

for (int num:queens) {

StringBuilder sb = new StringBuilder();

for (int j = 0;j < num;j++) {

sb.append('.');

}

sb.append('Q');

for (int j = num + 1;j < n;j++) {

sb.append('.');

}

list.add(sb.toString());

}

res.add(list);

} else {

for (int i = 0;i < n;i++) {

queens[cur] = i;

if (isValid(queens, cur)) {

dfs(res, queens, n, cur + 1);

}

// 这里不需要reset, 是因为只有在valid的情况下才会进入下一个位置对应的DFS

// when not valid, 下一个循环queens[cur]会直接被更新为下一个值，所以不需要reset

}

}

}

boolean isValid(int[] queens, int pos) {

for (int i = 0;i < pos;i++) {

if (queens[i] == queens[pos] || queens[i] - queens[pos] == i - pos || i + queens[i] == pos + queens[pos]) {

return false;

}

}

return true;

}

52. N-Queens II

Follow up for N-Queens problem.

Now, instead outputting board configurations, **return the total number of distinct solutions.**

**S1. Backtrack**

事实上这个followup比原版的要简单，

不需要存储List, 只需要传入一个int[] count, 并且更新count[0]即可。

用数组的形式是因为可以传递并更改。

public class Solution {

public int totalNQueens(int n) {

if (n <= 0) return 0;

int[] queens = new int[n];

int[] count = new int[]{0};

dfs(queens, n, 0, count);

return count[0];

}

private void dfs(int[] queens, int n, int cur, int[] count) {

if (cur == n) {

count[0]++;

} else {

for (int i = 0;i < n;i++) {

queens[cur] = i;

if (isValid(queens, cur)) {

dfs(queens, n, cur + 1, count);

}

}

}

}

boolean isValid(int[] queens, int pos) {

int len = queens.length;

for (int i = 0;i < pos;i++) {

if (queens[i] == queens[pos]

|| i - pos == queens[i] - queens[pos]

|| queens[i] + i == queens[pos] + pos)

return false;

}

return true;

}

}