LeetCode Summary

deepwaterooo

March 30, 2018

Contents

Chapter 1

LeetCode Summary - Part I

1.1 数组 Array

1.1.1 Remove Duplicates from Sorted Array

Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

For example,

Given input array A = [1,1,2],

Your function should return length = 2, and A is now [1,2].

1. 时间复杂度 O(n), 空间复杂度 O(1)

```
public int removeDuplicates(int[] a) {
    if (a.length == 0) return 0;
    int idx = 0;
    for (int i = 1; i < a.length; i++) {
        if (a[idx] != a[i])
            a[++idx] = a[i];
    }
    return idx + 1;
}
```

2. 使用 STL, 时间复杂度 O(n), 空间复杂度 O(1)

```
int removeDuplicates(int a[], int n) {
    return distance(a, unique(a, a + n));
}
```

3. 使用 STL, 时间复杂度 O(n), 空间复杂度 O(1)

```
int removeDuplicates(int A[], int n) {
    return removeDuplicates(A, A + n, A) - A;
}
template<typename InIt, typename OutIt>
OutIt removeDuplicates(InIt first, InIt last, OutIt output) {
    while (first != last) {
```

```
7           *output++ = *first;
8           first = find_if(first, last,
9                           bind1st(not_equal_to<int>(), *first));
10      }
11      return output;
12  }
```

1.1.2 Remove Element

Given an array and a value, remove all instances of that value in place and return the new length.

　　The order of elements can be changed. It doesn't matter what you leave beyond the new length.

1. 时间复杂度 O(n), 空间复杂度 O(1)

```
1  public int removeElement(int[] a, int elem) {
2      int idx = 0;
3      for (int i = 0; i < a.length; i++) {
4          if (a[i] != elem)
5              a[idx++] = a[i];
6      }
7      return idx;
8  }
```

2. 使用 remove(), 时间复杂度 O(n), 空间复杂度 O(1)

```
1  int removeElement(int A[], int n, int elem) {
2      return distance(A, remove(A, A+n, elem));
3  }
```

1.1.3 Plus One

1.1.4 Remove Duplicates from Sorted Array II

Follow up for "Remove Duplicates": What if duplicates are allowed at most twice?

　　For example,

　　Given sorted array A = [1,1,1,2,2,3],

　　Your function should return length = 5, and A is now [1,1,2,2,3].*** Rotate Image

　　加一个变量记录一下元素出现的次数即可。这题因为是已经排序的数组, 所以一个变量即可解决。如果是没有排序的数组, 则需要引入一个 hashmap 来记录出现次数。

1. 时间复杂度 O(n), 空间复杂度 O(1)

```
1  public int removeDuplicates(int[] a) {
2      if (a.length == 0) return 0;
3      int ocnt = 1;   // occurance count
4      int idx = 0;
5      for (int i = 1; i < a.length; i++) {
6          if (a[idx] == a[i]) {
7              if (ocnt < 2) {
8                  a[++idx] = a[i];
```

```
9                   ocnt++;
10               }
11           } else {
12               a[++idx] = a[i];
13               ocnt = 1;
14           }
15       }
16       return idx + 1;
17   }
```

2. 时间复杂度 O(n), 空间复杂度 O(1)

    下面是一个更简洁的版本。上面的代码略长, 不过扩展性好一些, 例如将 occur < 2 改为 occur < 3, 就变成了允许重复最多 3 次。

```
1   public int removeDuplicates(int[] a) {
2       //if (a.length == 0) return 0;
3       int idx = 0;
4       for (int i = 0; i < a.length; i++) {
5           if (i > 0 && i < a.length - 1 && a[i] == a[i - 1] && a[i] == a[i + 1])
6               continue;
7           a[idx++] = a[i];
8       }
9       return idx;
10  }
```

1.1.5 Set Matrix Zeroes

1.1.6 Search in Rotated Sorted Array II

Follow up for "Search in Rotated Sorted Array":

    What if duplicates are allowed?

    Would this affect the run-time complexity? How and why?

    Write a function to determine if a given target is in the array.

    允许重复元素, 则上一题中如果 A[m]>=A[l], 那么 [l,m] 为递增序列的假设就不能成立了, 比如 [1,3,1,1,1]。

    如果 A[m]>=A[l] 不能确定递增, 那就把它拆分成两个条件:

    • 若 A[m]>A[l], 则区间 [l,m] 一定递增

    • 若 A[m]==A[l] 确定不了, 那就 l++, 往下看一步即可。

```
1   public boolean search(int[] a, int target) {
2       int bgn = 0, end = a.length;
3       while (bgn != end) {
4           int mid = bgn + (end - bgn) / 2;
5           if (a[mid] == target) return true;
6           if (a[bgn] < a[mid]) {
7               if (a[bgn] <= target && target < a[mid])
8                   end = mid;
9               else
```

```
10              bgn = mid + 1;
11      } else if (a[bgn] > a[mid]) {
12          if (a[mid] < target && target <= a[end - 1])
13              bgn = mid + 1;
14          else
15              end = mid;
16      } else
17          // skip duplicated one, a[bgn] == a[mid]
18          bgn++;
19  }
20  return false;
21 }
```

- 3Sum

- 3Sum Closest

- 4Sum

1.1.7   Remove Element

- Next Permutation

1.1.8   Permutation Sequence

1.1.9   Valid Sudoku

1.1.10   Trapping Rain Water

1.1.11   Gray Code

Hard

1.1.12   Search in Rotated Sorted Array

Suppose a sorted array is rotated at some pivot unknown to you beforehand.
    (i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).
    You are given a target value to search. If found in the array return its index, otherwise return -1.
    You may assume no duplicate exists in the array.
    二分查找, 难度主要在于左右边界的确定。
    时间复杂度 O(log n), 空间复杂度 O(1).

```
1 public int search(int[] a, int target) {
2     int bgn = 0, end = a.length;
3     while (bgn < end) {
4         int mid = bgn + (end - bgn) / 2;
5         if (a[mid] == target) return mid;
6         if (a[bgn] <= a[mid]) {
7             if (a[bgn] <= target && target < a[mid])
8                 end = mid;
```

```
9              else
10                 bgn = mid + 1;
11         } else {
12             if (a[mid] < target && target <= a[end - 1])    //
13                 bgn = mid + 1;
14             else
15                 end = mid;
16         }
17     }
18     return -1;
19 }
```

1.1.13   Median of Two Sorted Arrays

There are two sorted arrays A and B of size m and n respectively. Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

Tags: Divide and Conquer, Array, Binary Search

这是一道非常经典的题。这题更通用的形式是, 给定两个已经排序好的数组, 找到两者所有元素中第 k 大的元素。

O(m + n) 的解法比较直观, 直接 merge 两个数组, 然后求第 k 大的元素。

不过我们仅仅需要第 k 大的元素, 是不需要"排序"这么复杂的操作的。可以用一个计数器, 记录当前已经找到第 m 大的元素了。同时我们使用两个指针 pA 和 pB, 分别指向 A 和 B 数组的第一个元素, 使用类似于 merge sort 的原理, 如果数组 A 当前元素小, 那么 pA++, 同时 m++; 如果数组 B 当前元素小, 那么 pB++, 同时 m++。最终当 m 等于 k 的时候, 就得到了我们的答案,O(k) 时间,O(1) 空间。但是, 当 k 很接近 m + n 的时候, 这个方法还是 O(m + n) 的。

有没有更好的方案呢? 我们可以考虑从 k 入手。如果我们每次都能够删除一个一定在第 k 大元素之前的元素, 那么我们需要进行 k 次。但是如果每次我们都删除一半呢? 由于 A 和 B 都是有序的, 我们应该充分利用这里面的信息, 类似于二分查找, 也是充分利用了"有序"。

假设 A 和 B 的元素个数都大于 k/2, 我们将 A 的第 k/2 个元素 (即 A[k/2-1]) 和 B 的第 k/2 个元素 (即 B[k/2-1]) 进行比较, 有以下三种情况 (为了简化这里先假设 k 为偶数, 所得到的结论对于 k 是奇数也是成立的):

- A[k/2-1] == B[k/2-1]
- A[k/2-1] > B[k/2-1]
- A[k/2-1] < B[k/2-1]

如果 A[k/2-1] < B[k/2-1], 意味着 A[0] 到 A[k/2-1] 的肯定在 A　B 的 top k 元素的范围内, 换句话说,A[k/2-1 不可能大于 A　B 的第 k 大元素。留给读者证明。

因此, 我们可以放心的删除 A 数组的这 k/2 个元素。同理, 当 A[k/2-1] > B[k/2-1] 时, 可以删除 B 数组的 k/2 个元素。

当 A[k/2-1] == B[k/2-1] 时, 说明找到了第 k 大的元素, 直接返回 A[k/2-1] 或 B[k/2-1] 即可。

因此, 我们可以写一个递归函数。那么函数什么时候应该终止呢?

- 当 A 或 B 是空时, 直接返回 B[k-1] 或 A[k-1];
- 当 k=1 是, 返回 min(A[0], B[0]);
- 当 A[k/2-1] == B[k/2-1] 时, 返回 A[k/2-1] 或 B[k/2-1]

```
1 public int helper(int [] a, int [] b, int k, int i, int j, int x, int y) {
2     int m = a.length;
3     int n = b.length;
4     if (j == i) return b[x + k];
```

```java
5        if (y == x) return a[i + k];
6        int mida = (i + j) / 2;
7        int midb = (x + y) / 2;
8        int lena = mida - i;
9        int lenb = midb - x;
10       if (b[midb] >= a[mida]) {
11           if (k <= lena + lenb)
12               return helper(a, b, k, i, j, x, midb);
13           else
14               return helper(a, b, k - lena - 1, mida + 1, j, x, y);
15       } else {
16           if (k <= lena + lenb) {
17               return helper(a, b, k, i, mida, x, y);
18           } else {
19               return helper(a, b, k - lenb - 1, i, j, midb + 1, y);
20           }
21       }
22   }
23
24   public double findMedianSortedArrays(int a[], int b[]) {
25       int m = a.length;
26       int n = b.length;
27       if ((m + n) % 2 == 0)
28           return (helper(a, b, (m + n) / 2, 0, m, 0, n) +
29                   helper(a, b, (m + n) / 2 - 1, 0, m, 0, n)) / 2.0;
30       else
31           return helper(a, b, (m + n) / 2, 0, m, 0, n);
32   }
```

1.1.14   Longest Consecutive Sequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example,

Given [100, 4, 200, 1, 3, 2],

The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.

Your algorithm should run in O(n) complexity.

如果允许 O(n log n) 的复杂度, 那么可以先排序, 可是本题要求 O(n)。由于序列里的元素是无序的, 又要求 O(n), 首先要想到用哈希表。

用一个哈希表 unordered_map<int, bool> used 记录每个元素是否使用, 对每个元素, 以该元素为中心, 往左右扩张, 直到不连续为止, 记录下最长的长度。

```java
1   public int longestConsecutive(int[] num) {
2       Map<Integer, Boolean> used = new HashMap<Integer, Boolean>();
3       int longest = 0;
4       for(int i : num) {
5           if (used.containsKey(i)) continue;
6           int length = 1;
7           used.put(i, true);
```

```
8           for (int j = i + 1; used.containsKey(j); j++) {
9               used.put(j, true);
10              ++length;
11          }
12          for (int j = i - 1; used.containsKey(j); j--) {
13              used.put(j, true);
14              ++length;
15          }
16          longest = Math.max(longest, length);
17      }
18      return longest;
19  }
```

OJ 超时。

```
1   public int longestConsecutive(int[] num) {
2       Map<Integer, Integer> used = new HashMap<Integer, Integer>();
3       int res = 0, tmp;
4       int left, right;
5       for(int i : num) {
6           if (!used.containsKey(i)) {
7               left = (used.containsKey(i - 1)) ? used.get(i - 1) : 0;
8               right = (used.containsKey(i + 1)) ? used.get(i + 1) : 0;
9               tmp = left + right + 1;
10              used.put(i, tmp);
11              res = Math.max(res, tmp);
12              used.put(i - left, tmp);
13              used.put(i + right, tmp);
14          } else continue;
15      }
16      return res;
17  }
```

简单地想一下，就知道前面一个遍历了所有数组中的无素，重复了已经 cnt 在最长片段里的元素，而这部分是可以跳过的。

1.1.15 Trapping Rain Water

1.1.16 others Covered

- Construct Binary Tree from Inorder and Postorder Traversal 1.4.17

- Construct Binary Tree from Preorder and Inorder Traversal 1.4.16

- Best Time to Buy and Sell Stock 2.7.2

- Best Time to Buy and Sell Stock II 2.7.3

- Largest Rectangle in Histogram 1.5.6

- Maximal Rectangle `Maximum Rectangle`

- Pascal's Triangle

- Pascal's Triangle II

- Merge Sorted Array 2.1.1

- Two Sum 1.6.3

- Two Sum II - Input array is sorted 3.1.13

- Spiral Matrix 3.6.7

- Spiral Matrix II 3.6.8

- Sort Colors 2.1.3

- Insert Interval 3.6.5

- Merge Intervals 3.6.6

1.1.17   Majority Element

1.1.18   Container With Most Water

1.1.19   Minimum Path Sum

- Search Insert Position 2.2.2

- Unique Paths

- Search for a Range 2.2.1

- Search a 2D Matrix 2.2.3

- Unique Paths II

1.1.20   Missing Ranges

1.1.21   Find Minimum in Rotated Sorted Array

1.1.22   Triangle

- Maximum Subarray `Maximum Subarray`

- Maximum Product Subarray 2.8.8

- Word Search 2.5.9

- Combination Sum 2.5.4

- Combination Sum II 2.5.5

- Subsets 2.3.1

- Subsets II

- Jump Game

1.1.23   Find Peak Element

1.1.24   Find Minimum in Rotated Sorted Array II

1.1.25   Jump Game II

- First Missing Positive

- Word Ladder II

- Best Time to Buy and Sell Stock III

- Roman to Integers `Roman to Integers`

- Integer to Roman

- Pow(x, n)

- Sqrt(x)

1.2   字符串 String

1.2.1   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.

时间复杂度 O(n), 空间复杂度 O(1).

```java
public boolean isPalindrome(String s) {
    int i = 0, j = s.length()-1;
    while (i < j) {
        while (i < j && !Character.isLetterOrDigit(s.charAt(i))) i++;
        while (i < j && !Character.isLetterOrDigit(s.charAt(j))) j--;
        if (Character.toLowerCase(s.charAt(i)) != Character.toLowerCase(s.charAt(j)))
            return false;
        i++;
        j--;
    }
    return true;
}
```

1.2.2 Implement strStr(), undone~

Implement strStr().

Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

The signature of the function had been updated to return the index instead of the pointer. If you still see your function signature returns a char * or String, please click the reload button to reset your code definition.

Tags: Two Pointers String

暴力算法的复杂度是 $O(m*n)$, 代码如下。更高效的的算法有 KMP 算法、Boyer-Mooer 算法和 Rabin-Karp 算法。面试中暴力算法足够了, 一定要写得没有 BUG。

1. 暴力解法, 时间复杂度 O(N*M), 空间复杂度 O(1)

```java
public int strStr(String haystack, String needle) {
    for (int i = 0; ; i++) {
        for (int j = 0; ; j++) {
            if (j == needle.length()) return i;
            if (i + j == haystack.length()) return -1;
            if (needle.charAt(j) != haystack.charAt(i + j)) break;
        }
    }
}
```

2. KMP, 时间复杂度 O(N+M), 空间复杂度 O(M)

1.2.3 String to Integer (atoi)

Implement atoi to convert a string to an integer.

Hint: Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.

Notes: It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.

spoilers alert… click to show requirements for atoi.

Requirements for atoi:

The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, INT_MAX (2147483647) or INT_MIN (-2147483648) is returned.

细节题。注意几个测试用例:

1. 不规则输入, 但是有效,"-3924x8fc","+ 413",

2. 无效格式,"++c", "++1"

3. 溢出数据,"2147483648"

时间复杂度 O(n), 空间复杂度 O(1).

```java
private static final int maxdt = Integer.MAX_VALUE / 10;

public int atoi(String str) {
    int i = 0, n = str.length();
    while (i < n && Character.isWhitespace(str.charAt(i))) i++;

    int sign = 1;
    if (i < n && str.charAt(i) == '+') i++;
    else if (i < n && str.charAt(i) == '-') {
        sign = -1;
        i++;
    }

    int val = 0;
    while (i < n && Character.isDigit(str.charAt(i))) {
        int dgt = Character.getNumericValue(str.charAt(i));
        if (val > maxdt || val == maxdt && dgt >= 8)
            return sign == 1 ? Integer.MAX_VALUE : Integer.MIN_VALUE;
        val = val * 10 + dgt;
        i++;
    }
    return sign * val;
}
```

时间复杂度 O(n), 空间复杂度 O(1).

```java
public int atoi(String str) {
    int num = 0;
    int sign = 1;
    int n = str.length();
    int i = 0;
    while (i < n && Character.isWhitespace(str.charAt(i))) i++;
    if (i < n && str.charAt(i) == '+') i++;
    else if (i < n && str.charAt(i) == '-') {
        sign = -1;
        i++;
    }
    for(; i < n; i++) {
        if (!Character.isDigit(str.charAt(i))) break;
        if (num > Integer.MAX_VALUE / 10
            || (num == Integer.MAX_VALUE / 10
                && (str.charAt(i) - '0' > Integer.MAX_VALUE % 10)))
            return sign == -1 ? Integer.MIN_VALUE : Integer.MAX_VALUE;
        num = num * 10 + str.charAt(i) - '0';
    }
```

```
20      return num * sign;
21  }
```

1.2.4   Add Binary

Given two binary strings, return their sum (also a binary string).
   For example,
   a = "11"
   b = "1"
   Return "100".
   时间复杂度 O(n), 空间复杂度 O(1)

```java
1   public String addBinary(String a, String b) {
2       StringBuffer res = new StringBuffer("");
3       int n = a.length() > b.length() ? a.length() : b.length();
4       a = new StringBuilder(a).reverse().toString();
5       b = new StringBuilder(b).reverse().toString();
6       int carry = 0;
7       for (int i = 0; i < n; i++) {
8           int aval = i < a.length() ? a.charAt(i) - '0' : 0;
9           int bval = i < b.length() ? b.charAt(i) - '0' : 0;
10          int value = (aval + bval + carry) % 2;
11          carry = (aval + bval + carry) / 2;
12          res.insert(0, (char)(value + '0'));
13      }
14      if (carry == 1)
15          res.insert(0, '1');
16      return res.toString();
17  }
```

1.2.5   Longest Common Prefix

Write a function to find the longest common prefix string amongst an array of strings.
   从位置 0 开始, 对每一个位置比较所有字符串, 直到遇到一个不匹配。

   1. 纵向扫描, 从位置 0 开始, 对每一个位置比较所有字符串, 直到遇到一个不匹配

      时间复杂度 O(n1+n2+…)

```java
1   public String longestCommonPrefix(String[] strs) {
2       if (strs.length == 0) return "";
3       for (int j = 0; j < strs[0].length(); j++) {
4           for (int i = 1; i < strs.length; i++) {
5               if (strs[i].length() < j+1 || strs[i].charAt(j) != strs[0].charAt(j))
6                   return strs[0].substring(0, j);
7           }
8       }
9       return strs[0];
10  }
```

2. 横向扫描, 每个字符串与第 0 个字符串, 从左到右比较, 直到遇到一个不匹配, 然后继续下一个字符串

时间复杂度 O(n1+n2+...)

1.2.6 Roman to Integer

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

从前往后扫描, 用一个临时变量记录分段数字。

如果当前比前一个大, 说明这一段的值应该是当前这个值减去上一个值。比如 IV = 5 − 1; 否则, 将当前值加入到结果中, 然后开始下一段记录。比如 VI = 5 + 1, II=1+1

```java
public int getInt(char x) {
    switch (x) {
    case 'I': return 1;
    case 'V': return 5;
    case 'X': return 10;
    case 'L': return 50;
    case 'C': return 100;
    case 'D': return 500;
    case 'M': return 1000;
    default: return 0;
    }
}

public int romanToInt(String s) {
    int result = getInt(s.charAt(0));
    int tmp, prev;
    for (int i = 1; i < s.length(); i++) {
        tmp = getInt(s.charAt(i));
        prev = getInt(s.charAt(i-1));
        result += prev < tmp ? tmp - 2 * prev : tmp;
    }
    return result;
}
```

1.2.7 Count and Say

The count-and-say sequence is the sequence of integers beginning as follows:

```
1, 11, 21, 1211, 111221, ...
1 is read off as "one 1" or 11.
11 is read off as "two 1s" or 21.
21 is read off as "one 2, then one 1" or 1211.
```

Given an integer n, generate the nth sequence.

Note: The sequence of integers will be represented as a string.

模拟。

时间复杂度 $O(n^2)$, 空间复杂度 $O(n)$

```cpp
string countAndSay(int n) {
    string s("1");
    while (--n)
        s = getNext(s);
    return s;
}


string getNext(const string &s) {
    stringstream ss;
    for (auto i = s.begin(); i != s.end(); ) {
        auto j = find_if(i, s.end(), bind1st(not_equal_to<char>(), *i));
        ss << distance(i, j) << *i;
        i = j;
    }
    return ss.str();
}
```

得优化自己的 java 代码。。

1.2.8   Length of Last Word

Given a string s consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string.

If the last word does not exist, return 0.

Note: A word is defined as a character sequence consists of non-space characters only.

For example,

Given s = "Hello World",

return 5.

细节实现题。

1. 用 STL: 时间复杂度 O(n), 空间复杂度 O(1)

```cpp
int lengthOfLastWord(const char *s) {
    const string str(s);
    auto first = find_if(str.rbegin(), str.rend(), ::isalpha);
    auto last = find_if_not(first, str.rend(), ::isalpha);
    return distance(first, last);
}
```

2. 顺序扫描, 记录每个 word 的长度, 时间复杂度 O(n), 空间复杂度 O(1)

```cpp
int lengthOfLastWord(const char *s) {
    int len = 0;
    while (*s) {
        if (*s++ != ' ')
            ++len;
        else if (*s && *s != ' ')
            len = 0;
    }
```

```
9          return len;
10    }
```

```
1    public int lengthOfLastWord(String s) {
2        if (s == null || s.length() == 0) return 0;
3        String [] tmp = s.split("\\s+");
4        if (tmp.length == 0) return 0;
5        return tmp[tmp.length-1].length();
6    }
```

1.2.9 Longest Palindromic Substring

Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.

最长回文子串, 非常经典的题。

思路一: 暴力枚举, 以每个元素为中间元素, 同时从左右出发, 复杂度 O(n 2)。

思路二: 记忆化搜索, 复杂度 O(n 2)。设 f[i][j] 表示 [i,j] 之间的最长回文子串, 递推方程如下:

```
1    f[i][j] =
2    if (i == j) S[i]
3    if (S[i] == S[j] && f[i+1][j-1] == S[i+1][j-1]) S[i][j]
4    else max(f[i+1][j-1], f[i][j-1], f[i+1][j])
```

思路三: 动规, 复杂度 O(n 2)。设状态为 f(i,j), 表示区间 [i,j] 是否为回文串, 则状态转移方程为

$$f(i, j) = \begin{cases} true, & i = j; \end{cases}$$

S[i] = S[j], j = i + 1
S[i] = S[j] and f (i + 1, j − 1) , j > i + 1
(fix this format one later)

思路四:Manacher's Algorithm, 复杂度 O(n)。详细解释见 http://leetcode.com/2011/11/longest-palindromic-substring-part-ii.html 。

1. 备忘录法, 会超时: 时间复杂度 $O(n^2)$, 空间复杂度 $O(n^2)$

2. 动规: 时间复杂度 $O(n^2)$, 空间复杂度 $O(n^2)$

```
1    public String longestPalindrome(String s) {
2        int n = s.length();
3        boolean [][] f = new boolean[n][n];
4        int maxLen = 1, start = 0;   // 最长回文子串的长度，起点
5        for (int i = 0; i < n; i++) {
6            f[i][i] = true;
7            for (int j = 0; j < i; j++) { // [j, i]
8                f[j][i] = (s.charAt(j) == s.charAt(i) && (i - j < 2 || f[j + 1][i - 1]));
9                if (f[j][i] && maxLen < (i - j + 1)) {
10                   maxLen = i - j + 1;
11                   start = j;
12               }
13           }
14       }
```

```
15          return s.substring(start, start + maxLen);
16    }
```

3. Manacher's Algorithm: 时间复杂度 O(n), 空间复杂度 O(n)

1.2.10   Integer to Roman

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

```
1  public String intToRoman(int num) {
2      int radix [] = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
3      String symbol [] = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"
4      StringBuffer res = new StringBuffer();
5      for (int i = 0; num > 0; i++) {
6          int cnt = num / radix[i];
7          num %= radix[i];
8          for (; cnt > 0 ; --cnt)
9              res.append(symbol[i]);
10      }
11      return res.toString();
12  }
```

感觉这个好像 hard code 稍微多了一点儿，如果面试官比较严格，就用下面这个吧。。。

```
1  public String intToRoman(int num) {
2      // index        0    1    2    3    4    5    6
3      char [] sym = {'I', 'V', 'X', 'L', 'C', 'D', 'M'};
4      // value:       1    5    10   50   100  500  1000
5      int scale = 1000;
6      StringBuffer result = new StringBuffer();
7      for (int i = 6; i >= 0; i -= 2) {
8          int digit = num / scale;
9          if (digit != 0) {
10              if (digit <= 3) {
11                  for (int j = 0; j < digit; j++)
12                      result.append(sym[i]);
13              } else if (digit == 4) {
14                  result.append(sym[i]);
15                  result.append(sym[i+1]);
16              } else if (digit == 5)
17                  result.append(sym[i+1]);
18              else if (digit > 5 && digit <= 8) {
19                  result.append(sym[i+1]);
20                  for (int j = 0; j < digit-5; j++)
21                      result.append(sym[i]);
22              } else { // == 9
23                  result.append(sym[i]);
24                  result.append(sym[i+2]);
```

```
25              }
26          }
27          num = num % scale;
28          scale /= 10;
29      }
30      return result.toString();
31  }
```

1.2.11   Anagrams

Given an array of strings, return all groups of strings that are anagrams.

Note: All inputs will be in lower-case.

Anagram(回文构词法) 是指打乱字母顺序从而得到新的单词, 比如"dormitory" 打乱字母顺序会变成"dirty room", "tea" 会变成"eat"。

回文构词法有一个特点: 单词里的字母的种类和数目没有改变, 只是改变了字母的排列顺序。因此, 将几个单词按照字母顺序排序后, 若它们相等, 则它们属于同一组 anagrams 。

时间复杂度 O(n), 空间复杂度 O(n)

```
1  vector<string> anagrams(vector<string> &strs) {
2      unordered_map<string, vector<string> > hash;
3      for (string s : strs) {
4          string tmp = s;
5          sort(tmp.begin(), tmp.end());
6          hash[tmp].push_back(s);
7      }
8      vector<string> result;
9      /*
10     for (auto key : hash)  {
11         if (key.second.size() > 1)
12             result.insert(result.end(), key.second.begin(), key.second.end());
13     } */
14     for (auto it = hash.cbegin(); it != hash.cend(); it++) {
15         if (it->second.size() > 1)
16             result.insert(result.end(), it->second.begin(), it->second.end());
17     }
18     return result;
19
20 }
```

Java 版:

```
1  public String sort(String a) {
2      if (a == "") return "";
3      char [] s = a.toCharArray();
4      Arrays.sort(s);
5      return new String(s);
6  }
7
```

```java
public List<String> anagrams(String[] strs) {
    List<String> list = new ArrayList<String>();  // result
    Map<String, List<Integer>> res = new HashMap<String, List<Integer>>();
    String [] sorted = new String[strs.length];
    List<Integer> one = new ArrayList<Integer>();
    for (int i = 0; i < strs.length; i++) {
        sorted[i] = sort(strs[i]);
        if (res.containsKey(sorted[i]))
            res.get(sorted[i]).add(i);
        else {
            one.add(i);
            res.put(sorted[i], one);
            one = new ArrayList<Integer>();
        }
    }
    for (String key : res.keySet())
        if (res.get(key).size() > 1)
            for (int i = 0; i < res.get(key).size(); i++)
                list.add(strs[res.get(key).get(i)]);
    return list;
}
```

1.2.12   Simplify Path

Given an absolute path for a file (Unix-style), simplify it.
For example,

```
path = "/home/", => "/home"
path = "/a/./b/../../c/", => "/c"
```

click to show corner cases.

Corner Cases:

Did you consider the case where path = "*..*"?

In this case, you should return "/".

Another corner case is the path might contain multiple slashes '/' together, such as "*home//foo*".

In this case, you should ignore redundant slashes and return "/home/foo".

很有实际价值的题目。

时间复杂度 O(n), 空间复杂度 O(n)

```java
public String simplifyPath(String path) {
    Stack<String> stack = new Stack<String>();
    Stack<String> f = new Stack<String>();
    String [] s = path.split("\\/");
    StringBuffer res = new StringBuffer();
    if (s.length == 0) return res.append('/').toString();
    for (int i = s.length - 1; i >= 0; i--)
        if (!s[i].equals("")) {
            stack.push(s[i]);
        }
```

```java
11      String tmp = "";
12      while (!stack.isEmpty()) {
13          tmp = stack.pop();
14          if (tmp.equals("."));
15          else if (tmp.equals("..")) {
16              if (!f.isEmpty())
17                  f.pop();
18          } else f.push(tmp);
19      }
20      if (!f.isEmpty()) {
21          while (!f.isEmpty()) {
22              tmp = f.pop();
23              res.insert(0, tmp);
24              res.insert(0, "/");
25          }
26      } else
27          res.append('/');
28      return res.toString();
29  }
```

c++ 版：需要明天再做一下。。

```cpp
1  string simplifyPath(string const& path) {
2      vector<string> dirs;
3      for (auto i = path.begin(); i != path.end() ; ) {
4          ++i;
5          auto j = find(i, path.end(), '/');
6          auto dir = string(i, j);
7          if (!dir.empty() && dir != ".") {
8              if (dir == "..") {
9                  if (!dirs.empty())
10                     dirs.pop_back();
11             } else
12                 dirs.push_back(dir);
13         }
14         i = j;
15     }
16     stringstream out;
17     if (dirs.empty()) out << "/";
18     else {
19         for (auto dir : dirs)
20             cout << '/' << dir;
21     }
22     return out.str();
23 }
24 /*
25   Input:^^I"/..."
26 Output:^^I""
```

```
27   Expected:^^I"/..."
28    */
```

Hard

1.2.13   Valid Number

Validate if a given string is numeric.

Some examples:

```
1   "0" => true
2   " 0.1 " => true
3   "abc" => false
4   "1 a" => false
5   "2e10" => true
```

Note: It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.

click to show spoilers.

Update (2014-12-06): New test cases had been added. Thanks unfounder's contribution.

细节实现题。

本题的功能与标准库中的 strtod() 功能类似。

1. 有限自动机: finite automata, 时间复杂度 O(n), 空间复杂度 O(n)

   这个还是等我改天再仔细研究吧。。。

2. 使用 strtod(): 偷懒, 直接用 strtod(), 时间复杂度 O(n)

```
1   bool isNumber (char const* s) {
2       char* endptr;
3       strtod (s, &endptr);
4       if (endptr == s) return false;
5       for (; *endptr; ++endptr)
6           if (!isspace (*endptr)) return false;
7       return true;
8   }
```

1.2.14   Edit Distance

1.2.15   Minimum Window Substring

1.2.16   Longest Substring with At Most Two Distinct Characters

1.2.17   Read N Characters Given Read4 II - Call multiple times

other Covered

- Longest Valid Parentheses 1.5.4

1.2.18   Valid Parentheses

1.2.19   Compare Version Numbers

1.2.20   Read N Characters Given Read4

1.2.21   One Edit Distance

1.2.22   Decode Ways

1.2.23   Longest Substring Without Repeating Characters

1.2.24   Reverse Words in a String

1.2.25   Interleaving String

- Multiply Strings

- Substring with Concatenation of All Words

- ZigZag Conversion

- Text Justification

- Restore IP Addresses

- Letter Combinations of a Phone Number

- Generate Parentheses

- Regular Expression Matching

- Wildcard Matching

- Scramble String

- Distinct Subsequences

- Word Ladder II 2.4.3

1.3   Linked List

1.3.1   Remove Duplicates from Sorted List

1.3.2   Remove Nth Node From End of List

Given a linked list, remove the nth node from the end of list and return its head.
  For example,
  Given linked list: 1->2->3->4->5, and n = 2.
  After removing the second node from the end, the linked list becomes 1->2->3->5. Note:
  Given n will always be valid.
  Try to do this in one pass.
  Tags: Linked List, Two Pointers

设两个指针 p, q, 让 q 先走 n 步, 然后 p 和 q 一起走, 直到 q 走到尾节点, 删除 p->next 即可。
时间复杂度 O(n), 空间复杂度 O(1).

```java
public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode dummy = new ListNode(-1);
    ListNode p = dummy;
    ListNode q = dummy;
    p.next = head;
    q.next = head;
    for (int i = 0; i < n; i++) {       // q 先走 n 步
        q = q.next;
    }
    while (q.next != null) {               // 一起走
        p = p.next;
        q = q.next;
    }
    ListNode tmp = p.next;
    p.next = tmp.next;
    return dummy.next;
}
```

1.3.3   Merge Two Sorted Lists

1.3.4   Intersection of Two Linked Lists

1.3.5   Add Two Numbers

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)

Output: 7 -> 0 -> 8

跟 Add Binary(见 §3.4) 很类似.

```java
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode head = new ListNode(-1);    // dummy 头节点
    int carry = 0;
    ListNode prev = head;
    for (ListNode pa = l1, pb = l2;
         pa != null || pb != null;
         pa = pa == null ? null : pa.next,
           pb = pb == null ? null : pb.next,
           prev = prev.next) {
        int aval = pa == null ? 0 : pa.val;
        int bval = pb == null ? 0 : pb.val;
        int value = (aval + bval + carry) % 10;
        carry = (aval + bval + carry) / 10;
        prev.next = new ListNode(value); // 尾插法
    }
    if (carry > 0)
```

```
17         prev.next = new ListNode(carry);
18     return head.next;
19 }
```

1.3.6 Reverse Linked List II

Reverse a linked list from position m to n. Do it in-place and in one-pass.

For example:

Given 1->2->3->4->5->NULL, m = 2 and n = 4,

return 1->4->3->2->5->NULL.

Note:

Given m, n satisfy the following condition:

1 m n length of list.

这题非常繁琐, 有很多边界检查,15 分钟内做到 bug free 很有难度!

1. 迭代版, 时间复杂度 O(n), 空间复杂度 O(1)

```
1 public ListNode reverseBetween(ListNode head, int m, int n) {
2     ListNode dummy = new ListNode(-1);
3     dummy.next = head;
4     ListNode prevM = null;
5     ListNode curr = dummy;
6     for (int i = 1; i <= n ; i++) {
7         if (i == m) prevM = curr;
8         if (i > m && i <= n) { // prev_m 和 curr 静止不动
9             curr.next = head.next;
10            head.next = prevM.next;
11            prevM.next = head;
12            head = curr;
13        }
14        curr = head;
15        head = head.next;
16    }
17    return dummy.next;
18 }
```

再仔细好好想一下这个代码 ~

1.3.7 Partition List

Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

For example,

Given 1->4->3->2->5->2 and x = 3,

return 1->2->2->4->3->5.

Tags: Linked List, Two Pointers

1. 时间复杂度 O(n), 空间复杂度 O(1)

```
1  public ListNode partition(ListNode head, int x) {
2      if (head == null) return head;
3      ListNode lftDummy = new ListNode(0);
4      ListNode ritDummy = new ListNode(0);
5      ListNode lftCurr = lftDummy; // 左头结点
6      ListNode ritCurr = ritDummy; // 右头结点
7
8      for (; head != null; head = head.next) {
9          if (head.val < x) {
10             lftCurr.next = head;
11             lftCurr = head;
12         } else {
13             ritCurr.next = head;
14             ritCurr = head;
15         }
16     }
17     lftCurr.next = ritDummy.next;
18     ritCurr.next = null; // 无限寻还
19     return lftDummy.next;
20 }
```

1.3.8 Remove Duplicates from Sorted List

Given a sorted linked list, delete all duplicates such that each element appear only once.

For example,

Given 1->1->2, return 1->2.

Given 1->1->2->3->3, return 1->2->3.

1. 递归版, 时间复杂度 O(n), 空间复杂度 O(1)

   c++ 版:

```
1  class Solution {
2  public:
3      ListNode *deleteDuplicates(ListNode *head) {
4          if (!head) return head;
5          ListNode dummy{head->val + 1, head}; // 值只要跟 head 不同即可
6          recur(&dummy, head);
7          return dummy.next;
8      }
9  private:
10     static void recur(ListNode *prev, ListNode *cur) {
11         if (cur == nullptr) return;
12         if (prev->val == cur->val) { // 删除 head
13             prev->next = cur->next;
14             delete cur;
15             recur(prev, prev->next);
16         } else {
17             recur(prev->next, cur->next);
```

```
18          }
19      }
20  };
```

Java 版:

```java
 1  public void recursion(ListNode prev, ListNode curr) {
 2      if (curr == null) return;
 3      if (prev.val == curr.val) {      // 删除 head
 4          prev.next = curr.next;
 5          recursion(prev, curr.next);
 6      } else {
 7          prev.next = curr;    // important for Java
 8          recursion(prev.next, curr.next);
 9      }
10  }
11
12  public ListNode deleteDuplicates(ListNode head) {
13      if (head == null) return head;
14      ListNode dummy = new ListNode(head.val + 1);
15      // 值只要跟 head 不同即可
16      recursion(dummy, head);
17      return dummy.next;
18  }
```

2. 迭代版, 时间复杂度 O(n), 空间复杂度 O(1)

```java
 1  public ListNode deleteDuplicates(ListNode head) {
 2      if (head == null) return head;
 3      for (ListNode prev = head, curr = head.next; curr != null; curr = curr.next) {
 4          if (prev.val == curr.val) {
 5              prev.next = curr.next;
 6          } else {
 7              prev = curr;
 8          }
 9      }
10      return head;
11  }
```

虽然觉得自己链表和树的原理掌握得还不错，可这两个版块都因为代码没有受过正规训练，所以代码奇差无比，看看别人写的。。。照葫芦画饼地写一遍，也算是一个提高的过程吧。。。

1.3.9 Remove Duplicates from Sorted List II

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

For example,

Given 1->2->3->3->4->4->5, return 1->2->5.

Given 1->1->1->2->3, return 2->3.

1. 递归版, 时间复杂度 O(n), 空间复杂度 O(1)

```java
public ListNode deleteDuplicates(ListNode head) {
    if (head == null || head.next == null) return head;
    ListNode curr = head.next;
    if (head.val == curr.val) {
        while (curr != null && head.val == curr.val)
            curr = curr.next;
        return deleteDuplicates(curr);
    } else {
        head.next = deleteDuplicates(head.next);
        return head;
    }
}
```

2. 迭代版, 时间复杂度 O(n), 空间复杂度 O(1)

```java
public ListNode deleteDuplicates(ListNode head) {
    if (head == null) return head;
    ListNode dummy = new ListNode(Integer.MIN_VALUE);
    dummy.next = head;
    ListNode prev = dummy;
    ListNode curr = head;
    while (curr != null) {
        boolean dup = false;
        while (curr.next != null && curr.val == curr.next.val) {
            dup = true;
            curr = curr.next;
        }
        if (dup) {    // 删除重复的最后一个元素
            curr = curr.next;
            continue;
        }
        prev.next = curr;
        prev = prev.next;
        curr = curr.next;
    }
    prev.next = curr;
    return dummy.next;
}
```

1.3.10   Rotate List

Given a list, rotate the list to the right by k places, where k is non-negative.

For example:

Given 1->2->3->4->5->NULL and k = 2,

return 4->5->1->2->3->NULL.

Tags: Linked List, Two Pointers

先遍历一遍, 得出链表长度 len, 注意 k 可能大于 len, 因此令 k% = len。将尾节点 next 指针指向首节点, 形成一个环, 接着往后跑 len − k 步, 从这里断开, 就是要求的结果了。(这个思路真是好。。。)

```java
public ListNode rotateRight(ListNode head, int n) {
    if (head == null || n == 0) return head;
    int len = 1;
    ListNode curr = head;
    while (curr.next != null) {     // 求长度
        len++;
        curr = curr.next;
    }
    n = len - n % len;
    curr.next = head;               // 首尾相连
    for (int step = 0; step < n; step++) {
        curr = curr.next;           // 接着往后跑
    }
    head = curr.next;               // 新的首节点
    curr.next = null;               // 断开环
    return head;
}
```

1.3.11 Swap Nodes in Pairs

Given a linked list, swap every two adjacent nodes and return its head.

For example,

Given 1->2->3->4, you should return the list as 2->1->4->3.

Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed.

1. 时间复杂度 O(n), 空间复杂度 O(1)

```java
public ListNode swapPairs(ListNode head) {
    if (head == null || head.next == null) return head;
    ListNode dummy = new ListNode(-1);
    dummy.next = head;
    for (ListNode prev = dummy, curr = prev.next, next = curr.next;
            next != null;
            prev = curr, curr = curr.next, next = curr != null ? curr.next : null) {
        prev.next = next;
        curr.next = next.next;
        next.next = curr;
    }
    return dummy.next;
}
```

2. 简洁版: 时间复杂度 O(n), 空间复杂度 O(1)

下面这种写法更简洁, 但题目规定了不准这样做。

```java
1  public ListNode swapPairs(ListNode head) {
2      ListNode p = head;
3      int tmp = 0;
4      while (p != null && p.next != null) {
5          tmp = p.val;
6          p.val = p.next.val;
7          p.next.val = tmp;
8          p = p.next.next;
9      }
10     return head;
11  }
```

1.3.12   Reorder List

Given a singly linked list L: L_0→L_1→···→L_{n-1}→L_n,
    reorder it to: L_0→L_n→L_1→L_{n-1}→L_2→L_{n-2}→···
    You must do this in-place without altering the nodes' values.
    For example,
    Given {1, 2, 3, 4}, reorder it to {1, 4, 2, 3}.
    题目规定要 in-place, 也就是说只能使用 O(1) 的空间。
    可以找到中间节点, 断开, 把后半截单链表 reverse 一下, 再合并两个单链表。
    时间复杂度 O(n), 空间复杂度 O(1)

```java
1  public ListNode reverse(ListNode head) {
2      if (head == null || head.next == null) return head;
3      ListNode prev = head;
4      for (ListNode curr = head.next, next = curr.next;
5            curr != null;
6            prev = curr, curr = next, next = next != null ? next.next : null) {
7          curr.next = prev;
8      }
9      head.next = null;
10     return prev;
11  }
12
13  public void reorderList(ListNode head) {
14      if (head == null || head.next == null) return;
15      ListNode slow = head;
16      ListNode fast = head;
17      ListNode prev = null;
18      while (fast != null && fast.next != null) {
19          prev = slow;
20          slow = slow.next;
21          fast = fast.next.next;
22      }
23      prev.next = null;      // cut at middle
24      slow = reverse(slow);
25      ListNode curr = head;
```

```
26      while (curr.next != null) {
27          ListNode tmp = curr.next;
28          curr.next = slow;
29          slow = slow.next;
30          curr.next.next = tmp;
31          curr = tmp;
32      }
33      curr.next = slow;
34  }
```

1.3.13   Convert Sorted List to Binary Search Tree

1.3.14   Linked List Cycle

Given a linked list, determine if it has a cycle in it.

Follow up:

Can you solve it without using extra space?

Tags：Linked List，Two Pointers

最容易想到的方法是, 用一个哈希表 $unordered_{map}$<int, bool> visited, 记录每个元素是否被访问过, 一旦出现某个元素被重复访问, 说明存在环。空间复杂度 O(n), 时间复杂度 O(N)。

最好的方法是时间复杂度 O(n), 空间复杂度 O(1) 的。设置两个指针, 一个快一个慢, 快的指针每次走两步, 慢的指针每次走一步, 如果快指针和慢指针相遇, 则说明有环。参考 http://leetcode.com/2010/09/detecting-loop-in-singly-linked-list.html

1. 时间复杂度 O(n), 空间复杂度 O(1)

```java
1  public boolean hasCycle(ListNode head) {
2      // 设置两个指针，一个快一个
3      ListNode fast = head;
4      ListNode slow = head;
5      while (fast != null && fast.next != null) {
6          slow = slow.next;
7          fast = fast.next.next;
8          if (slow == fast) return true;
9      }
10     return false;
11 }
```

1.3.15   Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

Follow up:

Can you solve it without using extra space?

Tags: Linked List, Two Pointers

当 fast 与 slow 相遇时, slow 肯定没有遍历完链表, 而 fast 已经在环内循环了 n 圈 (1 n)。假设 slow 走了 s 步, 则 fast 走了 2s 步 (fast 步数还等于 s 加上在环上多转的 n 圈), 设环长为 r, 则:

2s = s + nr

s = nr

设整个链表长 L, 环入口点与相遇点距离为 a, 起点到环入口点的距离为 x, 则

x + a = nr = (n-1)r + r = (n - 1)r + L - x

x = (n - 1)r + (L - x - a)

L-x-a 为相遇点到环入口点的距离, 由此可知, 从链表头到环入口点等于 n - 1 圈内环 + 相遇点到环入口点, 于是我们可以从 head 开始另设一个指针 slow2, 两个慢指针每次前进一步, 它俩一定会在环入口点相遇。
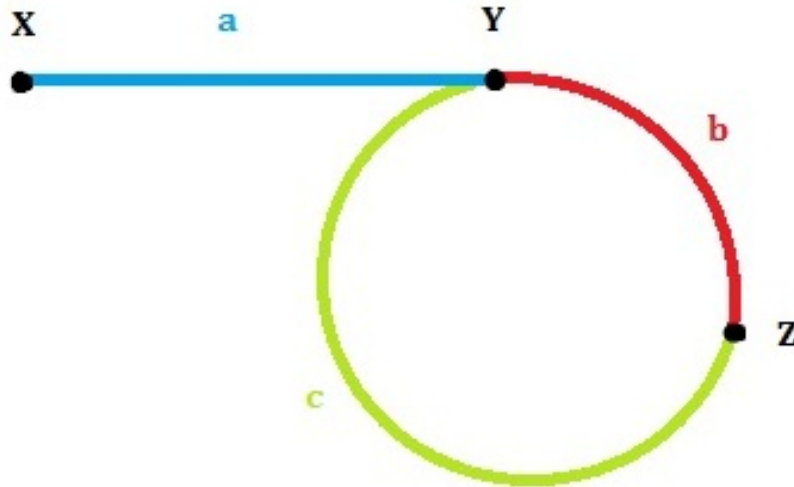
Figure 1.1: Linked List Cycle II

1. 方法一（网上都是这个答案）：

   第一次相遇后，让 slow,fast 继续走，记录到下次相遇时循环了几次。因为当 fast 第二次到达 Z 点时，fast 走了一圈，slow 走了半圈，而当 fast 第三次到达 Z 点时，fast 走了两圈，slow 走了一圈，正好还在 Z 点相遇。

   方法二：

   第一次相遇后，让 fast 停着不走了，slow 继续走，记录到下次相遇时循环了几次。

   方法三（最简单）：

   第一次相遇时 slow 走过的距离：a+b，fast 走过的距离：a+b+c+b。

   因为 fast 的速度是 slow 的两倍，所以 fast 走的距离是 slow 的两倍，有 2(a+b) = a+b+c+b，可以得到 a=c（这个结论很重要！）。

   我们发现 L=b+c=a+b，也就是说，从一开始到二者第一次相遇，循环的次数就等于环的长度。

2. 我们已经得到了结论 a=c，那么让两个指针分别从 X 和 Z 开始走，每次走一步，那么正好会在 Y 相遇！也就是环的第一个节点。

3. 在上一个问题的最后，将 c 段中 Y 点之前的那个节点与 Y 的链接切断即可。

4. 如何判断两个单链表是否有交点？先判断两个链表是否有环，如果一个有环一个没环，肯定不相交；如果两个都没有环，判断两个列表的尾部是否相等；如果两个都有环，判断一个链表上的 Z 点是否在另一个链表上。

如何找到第一个相交的节点? 求出两个链表的长度 L1,L2 (如果有环, 则将 Y 点当做尾节点来
算), 假设 L1<L2, 用两个指针分别从两个链表的头部开始走, 长度为 L2 的链表先走 (L2-L1)
步, 然后两个一起走, 直到二者相遇。

时间复杂度 O(n), 空间复杂度 O(1).

```java
public ListNode detectCycle(ListNode head) {
    ListNode fast = head;
    ListNode slow = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next; // 2X speed
        if (fast == slow) {
            ListNode curr = head;
            while (curr != slow) {
                curr = curr.next;
                slow = slow.next;
            }
            return curr;
        }
    }
    return null;
}
```

1.3.16   Reverse Nodes in k-Group

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.

If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.

You may not alter the values in the nodes, only nodes itself may be changed.

Only constant memory is allowed.

For example,

Given this linked list: 1->2->3->4->5

For k = 2, you should return: 2->1->4->3->5

For k = 3, you should return: 3->2->1->4->5

1. 递归版, 时间复杂度 O(n), 空间复杂度 O(1)

```java
public ListNode reverseKGroup(ListNode head, int k) {
    if (head == null || head.next == null || k < 2) return head;
    ListNode gNext = head;
    for (int i = 0; i < k; i++) {
        if (gNext != null) {
            gNext = gNext.next;
        } else return head;
    }
    // gNext is the head of next group
    // newGNext is the new head of next group after reversion
    ListNode newGNext = reverseKGroup(gNext, k);
    ListNode prev = null;
```

```
13        ListNode curr = head;
14        while (curr != gNext) {
15            ListNode next = curr.next;
16            curr.next = prev != null ? prev : newGNext;
17            prev = curr;
18            curr = next;
19        }
20        return prev;
21    }
```

2. 迭代版, 时间复杂度 O(n), 空间复杂度 O(1)

```
1    bug~~
```

1.3.17   Copy List with Random Pointer

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a deep copy of the list.

Tags: Hash Table, Linked List

```
1  public RandomListNode copyRandomList(RandomListNode head) {
2      if (head == null) return null;
3      for (RandomListNode curr = head; curr != null; ) {
4          RandomListNode node = new RandomListNode(curr.label);
5          node.next = curr.next;
6          curr.next = node;
7          curr = node.next;
8      }
9      for (RandomListNode curr = head; curr != null; ) {
10         if (curr.random != null)
11             curr.next.random = curr.random.next;
12         curr = curr.next.next;
13     }
14     // 分拆两个单链表
15     RandomListNode dummy = new RandomListNode(-1);
16     for (RandomListNode curr = head, newC = dummy; curr != null; ) {
17         newC.next = curr.next;
18         newC = newC.next;
19         curr.next = curr.next.next;
20         curr = curr.next;
21     }
22     return dummy.next;
23 }
```

改天去找一下 HashMap 的解法，或者自己写一个。
other Covered

- Sort List 2.1.4

- Insertion Sort List 2.1.6

- Merge k Sorted Lists `Merge k Sorted Lists`

1.4 树 **Binary Tree, Binary Search Tree**

Morris 专题

http://blog.ygui.me/algorithms/datastructure/2014-09/166

二叉树的 Morris 遍历可不使用用栈，在常量空间 O(1)、线性时间 O(n) 内实现二叉树的前中后序遍历，且遍历后不破坏二叉树的形状（中间过程允许改变其形状）。

Morris 遍历的基本原理是利用所有叶子结点的 right 指针，指向其后继结点，组成一个环，在第二次遍历到这个结点时，由于其左子树已经遍历完了，则访问该结点。

1.4.1 中序遍历 **(Inorder Traversal)**

中序遍历的算法步骤如下：

1. 如果当前节点的左孩子为空，则输出当前节点并将其右孩子作为当前节点。

2. 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点。

    A. 如果前驱节点的右孩子为空，将它的右孩子设置为当前节点。当前节点更新为当前节点的左孩子。

    B. 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空（恢复树的形状）。输出当前节点。当前节点更新为当前节点的右孩子。

3. 重复以上 1、2 直到当前节点为空。

    下图为每一步迭代的结果（从左至右，从上到下），cur 代表当前节点，深色节点表示该节点已输出。
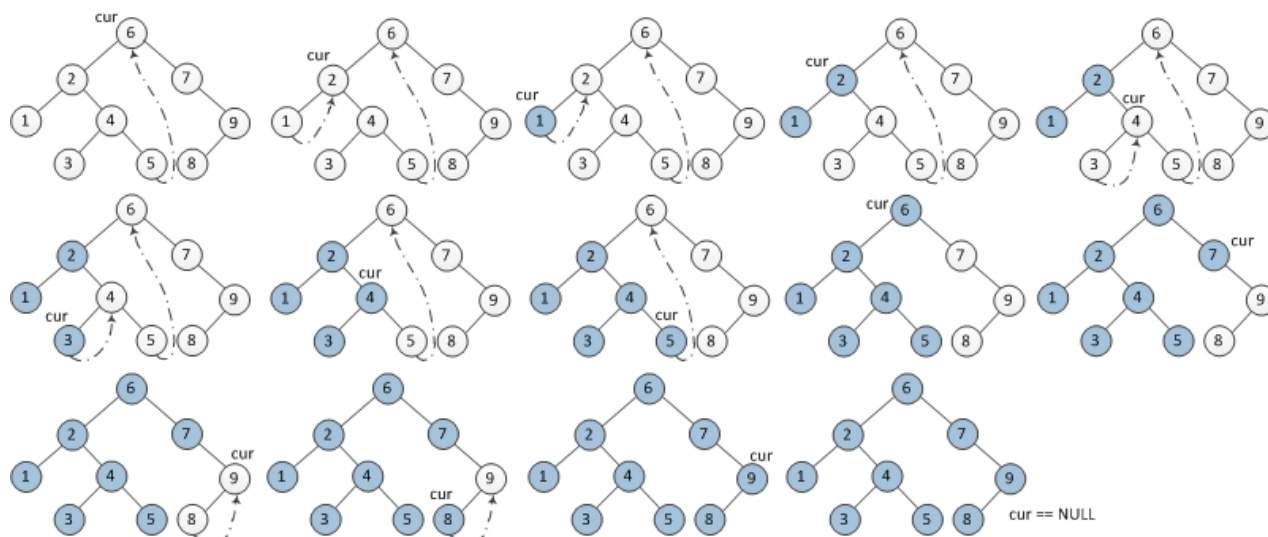
Figure 1.2: Morris 中序遍历

```
1  vector inorderTraversal(TreeNode *root) {
2      vector order;
3      for(TreeNode *now = root, *tmp; now;) {
4          if (now->left == NULL) {
5                  order.push_back(now->val);
6                  now = now->right;
7          } else {
8                  for (tmp = now->left;
9                       tmp->right != NULL && tmp->right != now;)
10                     tmp = tmp->right;
11                 if (tmp->right) {
12                     order.push_back(now->val);
13                     tmp->right = NULL;
14                     now = now->right;
15                 } else {
16                     tmp->right = now;
17                     now = now->left;
18                 }
19         }
20     }
21     return order;
22  }
```

1.4.2   先序遍历 (Preorder Traversal)

先序遍历与中序遍历相似，代码上只有一行不同，不同就在于输出的顺序。

1. 如果当前节点的左孩子为空，则输出当前节点并将其右孩子作为当前节点。

2. 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点。
   A. 如果前驱节点的右孩子为空，将它的右孩子设置为当前节点。输出当前节点并将当前节点更新为当前节点的左孩子。

   B. 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空。当前节点更新为当前节点的右孩子。

3. 重复以上 1、2 直到当前节点为空。

   下图为每一步迭代的结果（从左至右，从上到下），cur 代表当前节点，深色节点表示该节点已输出。

```
1  vector preorderTraversal(TreeNode *root) {
2  ^^Ivector order;
3  ^^Ifor (TreeNode *now = root, *tmp; now;) {
4  ^^I    if (now->left == NULL) {
5                  order.push_back(now->val);
6                  now = now->right;
7              } else {
8  ^^I        for (tmp = now->left; tmp->right != NULL && tmp->right != now;) {
9                  tmp = tmp->right;
```
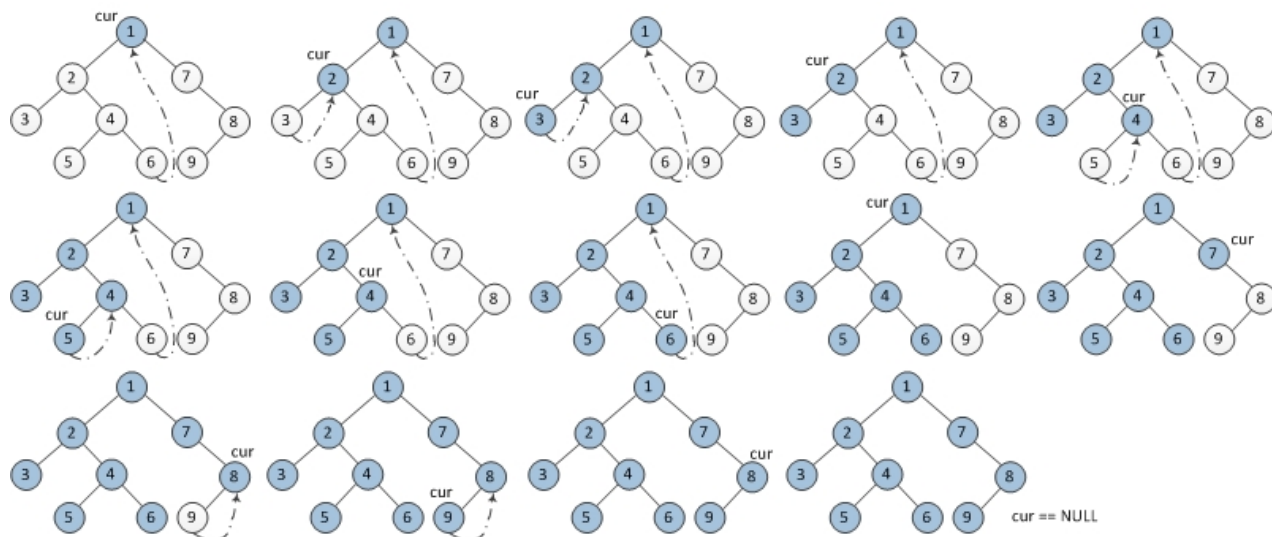
Figure 1.3: Morris 先序遍历

```
10  ^^I            if (tmp->right) {
11  ^^I^^I^^I^^Itmp->right = NULL;
12  ^^I^^I^^I^^Inow = now->right;
13  ^^I            } else {
14  ^^I^^I^^I^^Iorder.push_back(now->val);
15  ^^I^^I^^I^^Itmp->right = now;
16  ^^I^^I^^I^^Inow = now->left;
17  ^^I            }
18  ^^I        }
19  ^^I}
20  ^^Ireturn order;
21  }
```

1.4.3 后序遍历 (Postorder Traversal)

后序遍历比较复杂，它的思路是利用中序遍历，所以首先产生了一个假的根结点，其左子树为原来的二叉树，从假的根结点开始中序遍历它和中序遍历有所不同，在发现当前结点左子树为空时，不访问此结点 (后序遍历需要保证访问完右子树后才能访问根结点)，直接访问右子树。第二次遍历到某个结点时，将该结点左子树的最右路径反序输出即可。具体步骤如下:

1. 当前节点设置为临时节点 dump。

2. 如果当前节点的左孩子为空，则将其右孩子作为当前节点。

3. 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点。

    A. 如果前驱节点的右孩子为空，将它的右孩子设置为当前节点。当前节点更新为当前节点的左孩子。

    B. 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空。倒序输出从当前节点的左孩子到该前驱节点这条路径上的所有节点。当前节点更新为当前节点的右孩子。

4. 重复以上 2、3 直到当前节点为空。

下图为每一步迭代的结果（从左至右，从上到下），cur 代表当前节点，深色节点表示该节点已输出。
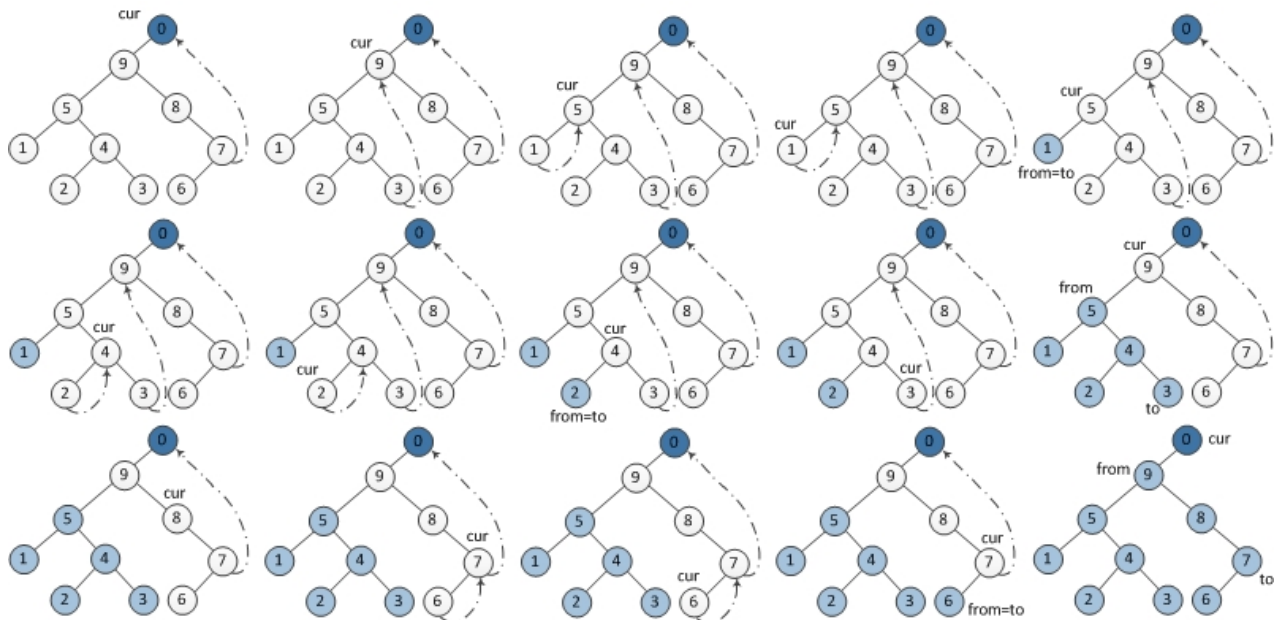
Figure 1.4: Morris 后序遍历

二叉树的遍历

树的遍历有两类: 深度优先遍历和宽度优先遍历。深度优先遍历又可分为两种: 先根 (次序) 遍历和后根 (次序) 遍历。

树的先根遍历是: 先访问树的根结点, 然后依次先根遍历根的各棵子树。树的先跟遍历的结果与对应二叉树 (孩子兄弟表示法) 的先序遍历的结果相同。

树的后根遍历是: 先依次后根遍历树根的各棵子树, 然后访问根结点。树的后跟遍历的结果与对应二叉树的中序遍历的结果相同。

二叉树的先根遍历有: 先序遍历 (root -> left -> right), root -> right -> left; 后根遍历有: 后序遍历 (left -> right -> root), right -> left -> root; 二叉树还有个一般的树没有的遍历次序, 中序遍历 (left -> root -> right)。

1.4.4　Binary Tree Preorder Traversal

Given a binary tree, return the preorder traversal of its nodes' values.

For example:

Given binary tree {1,#,2,3},

```
1    1
2     \
3      2
4     /
5    3
```

return [1,2,3].

Note: Recursive solution is trivial, could you do it iteratively?

用栈或 Morris 遍历

1. 栈: 使用栈, 时间复杂度 O(n), 空间复杂度 O(n)

```java
public List<Integer> preorderTraversal(TreeNode root) {
    List<Integer> res = new ArrayList<Integer>();
    if (root == null) return res;
    Stack<TreeNode> s = new Stack<TreeNode>();
    TreeNode curr = root;
    if (curr != null) s.push(curr);
    while (!s.isEmpty()) {
        curr = s.pop();
        res.add(curr.val);
        if (curr.right != null) s.push(curr.right);
        if (curr.left != null) s.push(curr.left);
    }
    return res;
}
```

2. Morris 先序遍历: Morris 先序遍历, 时间复杂度 O(n), 空间复杂度 O(1)

```java
public List<Integer> preorderTraversal(TreeNode root) {
    List<Integer> res = new ArrayList<Integer>();
    TreeNode curr = root;
    TreeNode prev = null;
    while (curr != null) {
        if (curr.left == null) {
            res.add(curr.val);
            prev = curr;        /* cur 刚刚被访问过 */
            curr = curr.right;
        } else {                        /* 查找前驱 */
            TreeNode node = curr.left;
            while (node.right != null && node.right != curr) {
                node = node.right;
            }
            if (node.right == null) {   /* 还没线索化，则建立线索 */
                res.add(curr.val);      /* 仅这一行的位置与中序不同 */
                node.right = curr;
                prev = curr;            /* cur 刚刚被访问过 */
                curr = curr.left;
            } else {                /* 已经线索化，则删除线索 */
                node.right = null;
                /* prev = cur; 不能有这句,cur 已经被访问 */
                curr = curr.right;
            }
        }
```

```
26        }
27        return res;
28    }
```

1.4.5   Binary Tree Inorder Traversal

Given a binary tree, return the inorder traversal of its nodes' values.

For example:

Given binary tree {1,#,2,3},

```
1     1
2      \
3       2
4      /
5     3
```

return [1,3,2].

Note: Recursive solution is trivial, could you do it iteratively?

1. 栈: 使用栈, 时间复杂度 O(n), 空间复杂度 O(n)

```
1   public List<Integer> inorderTraversal(TreeNode root) {
2       List<Integer> res = new ArrayList<Integer>();
3       TreeNode p = root;
4       Stack<TreeNode> s = new Stack<TreeNode>();
5       while (!s.isEmpty() || p != null) {
6           if (p != null) {
7               s.push(p);
8               p = p.left;
9           } else {
10              p = s.pop();
11              res.add(p.val);
12              p = p.right;
13          }
14      }
15      return res;
16  }
```

2. Morris 先序遍历: Morris 先序遍历, 时间复杂度 O(n), 空间复杂度 O(1)

```
1   public List<Integer> inorderTraversal(TreeNode root) {
2       List<Integer> res = new ArrayList<Integer>();
3       TreeNode prev = null;
4       TreeNode curr = root;
5       Stack<TreeNode> s = new Stack<TreeNode>();
6       while (curr != null) {
7           if (curr.left == null) {
8               res.add(curr.val);
9               prev = curr;
```

```
10                  curr = curr.right;
11              } else {
12                  /* 查找前驱 */
13                  TreeNode node = curr.left;
14                  while (node.right != null && node.right != curr) {
15                      node = node.right;
16                  }
17                  if (node.right == null) {  /* 还没线索化，则建立线索 */
18                      node.right = curr;
19                      /* prev = cur; 不能有这句,cur 还没有被访问 */
20                      curr = curr.left;
21                  } else {          /* 已经线索化，则访问节点，并删除线索 */
22                      res.add(curr.val);
23                      node.right = null;
24                      curr = curr.right;
25                  }
26              }
27          }
28      return res;
29  }
```

1.4.6 Binary Tree Postorder Traversal

Given a binary tree, return the postorder traversal of its nodes' values.

For example:

Given binary tree {1,#,2,3},

```
1     1
2      \
3       2
4      /
5     3
```

return [3,2,1].

Note: Recursive solution is trivial, could you do it iteratively?

1. 栈: 使用栈, 时间复杂度 O(n), 空间复杂度 O(n)

```
1   public List<Integer> postorderTraversal(TreeNode root) {
2       List<Integer> res = new ArrayList<Integer>();
3       /* curr，正在访问的结点；prev，刚刚访问过的结点 */
4       TreeNode prev = null;
5       TreeNode curr = root;
6       Stack<TreeNode> s = new Stack<TreeNode>();
7       do {
8           while (curr != null) {      /* 往左下走 */
9               s.push(curr);
10              curr = curr.left;
11          }
```

```
12            prev = null;
13            while (!s.isEmpty()) {
14                curr = s.pop();
15                /* 右孩子不存在或已被访问，访问之 */
16                if (curr.right == prev) {
17                    res.add(curr.val);
18                    prev = curr;  /* 保存刚访问过的结点 */
19                } else {
20                    /* 当前结点不能访问，需第二次进栈 */
21                    s.push(curr);
22                    curr = curr.right; /* 先处理右子树 */
23                    break;
24                }
25            }
26        } while (!s.isEmpty());
27        return res;
28    }
```

2. Morris 先序遍历：Morris 先序遍历, 时间复杂度 O(n), 空间复杂度 O(1)

```
1  // 逆转路径
2  public void reverse(TreeNode bgn, TreeNode end) {
3      TreeNode x = bgn;
4      TreeNode y = bgn.right;
5      TreeNode z = null;
6      if (bgn == end) return;
7      while (x != end) {
8          z = y.right;
9          y.right = x;
10         x = y;
11         y = z;
12     }
13 }
14
15 // 访问逆转后的路径上的所有结点
16 public void visit_reverse(TreeNode bgn, TreeNode end, List<Integer> res) {
17     TreeNode p = end;
18     reverse(bgn, end);
19     while (true) {
20         res.add(p.val);
21         if (p == bgn) break;
22         p = p.right;
23     }
24     reverse(end, bgn);
25 }
26
27 public List<Integer> postorderTraversal(TreeNode root) {
28     List<Integer> res = new ArrayList<Integer>();
```

```
29          TreeNode dummy = new TreeNode(-1);
30          TreeNode curr = null;
31          TreeNode prev = null;
32          dummy.left = root;
33          curr = dummy;
34          while (curr != null) {
35              if (curr.left == null) {
36                  prev = curr;                              /* 必须要有 */
37                  curr = curr.right;
38              } else {
39                  TreeNode node = curr.left;
40                  while (node.right != null && node.right != curr)
41                      node = node.right;
42                  if (node.right == null) { /* 还没线索化，则建立线索 */
43                      node.right = curr;
44                      prev = curr;                          /* 必须要有 */
45                      curr = curr.left;
46                  } else {        /* 已经线索化，则访问节点，并删除线索 */
47                      visit_reverse(curr.left, prev, res);
48                      prev.right = null;
49                      prev = curr;                          /* 必须要有 */
50                      curr = curr.right;
51                  }
52              }
53          }
54      return res;
55  }
```

c++ 版:

```
1   // LeetCode, Binary Tree Postorder Traversal
2   // Morris 后序遍历，时间复杂度 O(n)，空间复杂度 O(1)
3   class Solution {
4   public:
5       vector<int> postorderTraversal(TreeNode *root) {
6           vector<int> result;
7           TreeNode dummy(-1);
8           TreeNode *cur, *prev = nullptr;
9           std::function < void(const TreeNode*)>
10              visit = [&result](const TreeNode *node) {
11              result.push_back(node->val);
12          };
13
14          dummy.left = root;
15          cur = &dummy;
16          while (cur != nullptr) {
17              if (cur->left == nullptr) {
18                  prev = cur;                               /* 必须要有 */
```

```
19                      cur = cur->right;
20                  } else {
21                      TreeNode *node = cur->left;
22                      while (node->right != nullptr && node->right != cur)
23                          node = node->right;
24                      if (node->right == nullptr) { /* 还没线索化，则建立线索 */
25                          node->right = cur;
26                          prev = cur;                    /* 必须要有 */
27                          cur = cur->left;
28                      } else { /* 已经线索化，则访问节点，并删除线索 */
29                          visit_reverse(cur->left, prev, visit);
30                          prev->right = nullptr;
31                          prev = cur;                    /* 必须要有 */
32                          cur = cur->right;
33                      }
34                  }
35              }
36              return result;
37          }
38  private:
39      // 逆转路径
40      static void reverse(TreeNode *from, TreeNode *to) {
41          TreeNode *x = from, *y = from->right, *z;
42          if (from == to) return;
43          while (x != to) {
44              z = y->right;
45              y->right = x;
46              x = y;
47              y = z;
48          }
49      }
50      // 访问逆转后的路径上的所有结点
51      static void visit_reverse(TreeNode* from, TreeNode *to,
52                              std::function< void(const TreeNode*) > &visit) {
53          TreeNode *p = to;
54          reverse(from, to);
55          while (true) {
56              visit(p);
57              if (p == from)
58                  break;
59              p = p->right;
60          }
61          reverse(to, from);
62      }
63  };
```

一段代码写成这样，实在是太恐怖了。。。面试的时候怎么出得来? 还是把原理弄懂尽量不要写这种方法好了。。。

3. root, right, left –> Reverse

另一个比较 tricky 的方法就是先模拟 postorder （实际上是按 root, right, left 的顺序遍历的），再 reverse 得到的结果。

```java
public List<Integer> postorderTraversal0(TreeNode root) {
    // get preorder results first, no no no no, root, right, left
    List<Integer> res = new ArrayList<Integer>();
    if (root == null) {
        return res;
    }
    Stack<TreeNode> s = new Stack<TreeNode>();
    TreeNode curr = root;
    while (!s.isEmpty() || curr != null) {
        while (curr != null) {
            res.add(curr.val);
            s.push(curr);
            curr = curr.right;
        }
        if (!s.isEmpty()) {
            curr = s.pop();
            curr = curr.left;
        }
    }
    // then reverse the values
    List<Integer> result = new ArrayList<Integer>();
    for (int i = res.size() - 1; i >= 0; i--) {
        result.add(res.get(i));
    }
    return result;
}
```

1.4.7 Binary Tree Level Order Traversal

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example:

Given binary tree {3,9,20,#,#,15,7},

```
    3
   / \
  9  20
     / \
    15  7
```

return its level order traversal as:

```
[
  [3],
  [9,20],
```

```
4    [15,7]
5  ]
```

1. 递归版: 时间复杂度 O(n), 空间复杂度 O(n)

```java
1  public void traverse(TreeNode root, int level, List<List<Integer>> res) {
2      if (root == null) return;
3      if (level > res.size())
4          res.add(new ArrayList<Integer>());
5      res.get(level - 1).add(root.val);
6      traverse(root.left, level + 1, res);
7      traverse(root.right, level + 1, res);
8  }
9
10 public List<List<Integer>> levelOrder(TreeNode root) {
11     List<List<Integer>> res = new ArrayList<List<Integer>>();
12     traverse(root, 1, res);
13     return res;
14 }
```

2. 迭代版: 时间复杂度 O(n), 空间复杂度 O(1)

```java
1  public List<List<Integer>> levelOrder(TreeNode root) {
2      List<List<Integer>> res = new ArrayList<List<Integer>>();
3      if (root == null) return res;
4      Queue<TreeNode> curr = new LinkedList<TreeNode>();
5      Queue<TreeNode> next = new LinkedList<TreeNode>();
6      List<Integer> level = new ArrayList<Integer>();
7
8      curr.add(root);
9      while (!curr.isEmpty()) {
10         while (!curr.isEmpty()) {
11             TreeNode node = curr.poll();
12             level.add(node.val);
13             if (node.left != null) next.add(node.left);
14             if (node.right != null) next.add(node.right);
15         }
16         res.add(level);
17         //level.clear();    // not working
18         level = new ArrayList<Integer>();
19         curr.addAll(next);
20         next.clear();
21     }
22     return res;
23 }
```

1.4.8 Binary Tree Level Order Traversal II

Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example: Given binary tree {3,9,20,#,#,15,7},

```
1      3
2    /  \
3   9   20
4      /  \
5     15   7
```
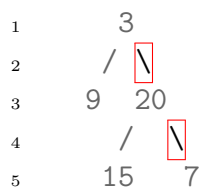
return its bottom-up level order traversal as:

```
1  [
2    [15,7],
3    [9,20],
4    [3]
5  ]
```

在上一题 (Binary Tree Level Order Traversal 见 1.4.7) 的基础上,reverse() 一下即可。

1. 递归版: 时间复杂度 O(n), 空间复杂度 O(n)

```java
1  public void traverse(TreeNode root, int level, List<List<Integer>> res) {
2      if (root == null) return;
3      if (level > res.size())
4          res.add(new ArrayList<Integer>());
5      res.get(level - 1).add(root.val);
6      traverse(root.left, level + 1, res);
7      traverse(root.right, level + 1, res);
8  }
9
10 public List<List<Integer>> levelOrderBottom(TreeNode root) {
11     List<List<Integer>> res = new ArrayList<List<Integer>>();
12     traverse(root, 1, res);
13     Collections.reverse(res);    // 比上一题多此一行
14     return res;
15 }
```

2. 迭代版: 时间复杂度 O(n), 空间复杂度 O(1)

```java
1  public List<List<Integer>> levelOrderBottom(TreeNode root) {
2      List<List<Integer>> res = new ArrayList<List<Integer>>();
3      if (root == null) return res;
4      Queue<TreeNode> curr = new LinkedList<TreeNode>();
5      Queue<TreeNode> next = new LinkedList<TreeNode>();
6      List<Integer> level = new ArrayList<Integer>();
7
8      curr.add(root);
9      while (!curr.isEmpty()) {
10         while (!curr.isEmpty()) {
11             TreeNode node = curr.poll();
12             level.add(node.val);
```

```
13                if (node.left != null) next.add(node.left);
14                if (node.right != null) next.add(node.right);
15            }
16            res.add(level);
17            level = new ArrayList<Integer>();
18            curr.addAll(next);
19            next.clear();
20        }
21        Collections.reverse(res);     // 比上一题多此一行
22        return res;
23    }
```

1.4.9   Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right,
then right to left for the next level and alternate between).

For example:

Given binary tree {3,9,20,#,#,15,7},

```
1       3
2      / \
3     9   20
4        /  \
5       15   7
```

return its zigzag level order traversal as:

```
1  [
2    [3],
3    [20,9],
4    [15,7]
5  ]
```

广度优先遍历, 用一个 bool 记录是从左到右还是从右到左, 每一层结束就翻转一下。

1. 递归版, 时间复杂度 O(n), 空间复杂度 O(n)

```
1  public void traverse(TreeNode root, int level,
2                        List<List<Integer>> res, boolean lefttoright) {
3      if (root == null) return;
4      if (level > res.size())
5          res.add(new ArrayList<Integer>());
6      if (lefttoright)
7          res.get(level - 1).add(root.val);
8      else
9          res.get(level - 1).add(0, root.val);
10     traverse(root.left, level + 1, res, !lefttoright);
11     traverse(root.right, level + 1, res, !lefttoright);
12 }
13
```

```
14    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
15        List<List<Integer>> res = new ArrayList<List<Integer>>();
16        traverse(root, 1, res, true);
17        return res;
18    }
```

2. 迭代版, 时间复杂度 O(n), 空间复杂度 O(n)

   广度优先遍历, 用一个 bool 记录是从左到右还是从右到左, 每一层结束就翻转一下。

```
1    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
2        List<List<Integer>> res = new ArrayList<List<Integer>>();
3        if (root == null) return res;
4        Queue<TreeNode> q = new LinkedList<TreeNode>();
5        boolean lefttoright = true;
6        List<Integer> level = new ArrayList<Integer>();
7        q.add(root);
8        q.add(null);        // level separator
9        while (!q.isEmpty()) {
10            TreeNode curr = q.poll();
11            if (curr != null) {
12                level.add(curr.val);
13                if (curr.left != null) q.add(curr.left);
14                if (curr.right != null) q.add(curr.right);
15            } else {
16                if (lefttoright) {
17                    res.add(level);
18                } else {
19                    Collections.reverse(level);
20                    res.add(level);
21                }
22                level = new ArrayList<Integer>();
23                lefttoright = !lefttoright;
24                if (q.size() > 0) q.add(null);
25            }
26        }
27        return res;
28    }
```

1.4.10 Recover Binary Search Tree

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note:

A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

O(n) 空间的解法是, 开一个指针数组, 中序遍历, 将节点指针依次存放到数组里, 然后寻找两处逆向的位置, 先从前往后找第一个逆序的位置, 然后从后往前找第二个逆序的位置, 交换这两个指针的值。

中序遍历一般需要用到栈, 空间也是 O(n) 的, 如何才能不使用栈? Morris 中序遍历。

1. 一个 inorder 的解法:

```java
public void inOrderTraverse(TreeNode root) {
    if (root == null) return;
    if (root.left != null)
        inOrderTraverse(root.left);

    if (prev.val < root.val) {
        prev = root;
    } else if (prev.val > root.val) {
        if (first == null) {
            first = prev;
            second = root;
        } else second = root;
        prev = root;
    }
    if (root.right != null)
        inOrderTraverse(root.right);
}

public void recoverTree(TreeNode root) {
    if (root == null || (root.left == null && root.right == null)) return;
    if (first == null || second == null) {
        TreeNode curr = root;
        if (curr.left != null)
            while (curr.left != null)
                curr = curr.left;
        prev = curr;
        inOrderTraverse(root);
    }
    int tmp = first.val;
    first.val = second.val;
    second.val = tmp;
}
```

2. Morris 中序遍历, 时间复杂度 O(n) , 空间复杂度 O(1)

```java
public void detect(List<TreeNode> res, TreeNode prev, TreeNode curr) {
    if (prev != null && prev.val > curr.val) {
        if (res.get(0) == null)
            res.add(0, prev);
        // 不能用 else, 例如 {0,1}, 会导致最后 swap 时 second 为 nullptr,
        // 会 Runtime Error
        res.add(1, curr);
    }
}

public void recoverTree(TreeNode root) {
```

```java
12        TreeNode prev = null;
13        TreeNode curr = root;
14        List<TreeNode> res = new ArrayList<TreeNode>();
15        res.add(null);
16
17        while (curr != null) {
18            if (curr.left == null) {
19                detect(res, prev, curr);
20                prev = curr;
21                curr = curr.right;
22            } else {
23                TreeNode node = curr.left;
24                while (node.right != null && node.right != curr) {
25                    node = node.right;
26                }
27                if (node.right == null) {
28                    node.right = curr;
29                    //prev = cur; 不能有这句！因为 cur 还没有被访问
30                    curr = curr.left;
31                } else {
32                    detect(res, prev, curr);
33                    node.right = null;
34                    prev = curr;
35                    curr = curr.right;
36                }
37            }
38        }
39        int tmp = res.get(0).val;
40        res.get(0).val = res.get(1).val;
41        res.get(1).val = tmp;
42    }
```

这个可以套成模板，可能写得不熟悉吧，现在写起来还是怎么写怎么别扭，不如原本朴素的 inorder traverse 来得自然。。。。

1.4.11   Same Tree

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

1. 递归版: 时间复杂度 O(n), 空间复杂度 O(logn)

```java
1  public boolean isSameTree(TreeNode p, TreeNode q) {
2      if (p == null && q == null) return true;   // 终止条件
3      if (p == null || q == null) return false;  // 剪枝
4      return p.val == q.val                      // 三方合并
5          && isSameTree(p.left, q.left)
```

```
6            && isSameTree(p.right, q.right);
7  }
```

2. 迭代版: 时间复杂度 O(n), 空间复杂度 O(logn)

```
1  public boolean isSameTree(TreeNode p, TreeNode q) {
2      Stack<TreeNode> s = new Stack<TreeNode>();
3      s.push(p);
4      s.push(q);
5      while (!s.isEmpty()) {
6          p = s.pop();
7          q = s.pop();
8          if (p == null && q == null) continue;
9          if (p == null || q == null) return false;
10         if (p.val != q.val) return false;
11         s.push(p.left);
12         s.push(q.left);
13         s.push(p.right);
14         s.push(q.right);
15     }
16     return true;
17 }
```

1.4.12 Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree is symmetric:

```
1      1
2     / \
3    2   2
4   / \ / \
5  3  4 4  3
```

But the following is not:

```
1      1
2     / \
3    2   2
4     \    \
5      3    3
```

Note:

Bonus points if you could solve it both recursively and iteratively.

1. 递归版: 时间复杂度 O(n), 空间复杂度 O(logn)

```
1  public boolean isSymTree(TreeNode left, TreeNode right) {
2      if (left == null && right == null) return true;
3      if (left == null || right == null) return false;
```

```java
4        return left.val == right.val
5            && isSymTree(left.left, right.right)
6            && isSymTree(left.right, right.left);
7   }
8
9   public boolean isSymmetric(TreeNode root) {
10       if (root == null ) return true;
11       return isSymTree(root.left, root.right);
12   }
```

2. 迭代版: 时间复杂度 O(n), 空间复杂度 O(logn)

```java
1   public boolean isSymmetric(TreeNode root) {
2       if (root == null) return true;
3       Stack<TreeNode> s = new Stack<TreeNode>();
4       s.push(root.left);
5       s.push(root.right);
6       while (!s.isEmpty()) {
7           TreeNode p = s.pop();
8           TreeNode q = s.pop();
9           if (p == null && q == null) continue;
10          if (p == null || q == null) return false;
11          if (p.val != q.val) return false;
12          s.push(p.left);
13          s.push(q.right);
14          s.push(p.right);
15          s.push(q.left);
16      }
17      return true;
18  }
```

1.4.13   Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

时间复杂度 O(n), 空间复杂度 O(logn)

```java
1   public int balanceHeight(TreeNode root) {
2       if (root == null) return 0;        // 终止条件
3       int left = balanceHeight(root.left);
4       int right = balanceHeight(root.right);
5       if (left < 0 || right < 0 || Math.abs(left - right) > 1)
6           return -1;                     // 剪枝
7       return Math.max(left, right) + 1;  // 三方合并
8   }
9
10  public boolean isBalanced(TreeNode root) {
```

```
11        return balanceHeight(root) >= 0;
12    }
```

1.4.14   Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.

For example, Given

```
1        1
2       / \
3      2   5
4     / \   \
5    3   4   6
```
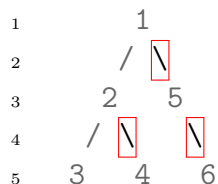
The flattened tree should look like:

```
1     1
2      \
3       2
4        \
5         3
6          \
7           4
8            \
9             5
10             \
11              6
```
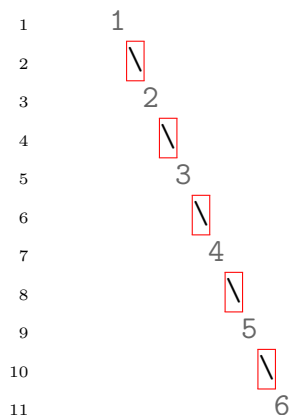
click to show hints.

Hints:

If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order traversal.

1. 递归版 1: 时间复杂度 O(n), 空间复杂度 O(logn)

```
1   public void flatten(TreeNode root) {
2       if (root == null) return;      // 终止条件
3       flatten(root.left);
4       flatten(root.right);
5       if (root.left == null) return;
6       // 三方合并，将左子树所形成的链表插入到 root 和 root->right 之间
7       TreeNode p = root.left;
8       while (p.right != null)
9           p = p.right;      //寻找左链表最后一个节点
10      p.right = root.right;
11      root.right = root.left;
12      root.left = null;
13  }
```

2. 递归版 2: 时间复杂度 O(n), 空间复杂度 O(logn)

```
1  // 把 root 所代表树变成链表后，tail 跟在该链表后面
2  public TreeNode flatten(TreeNode root, TreeNode tail) {
3      if (root == null) return tail;
4      root.right = flatten(root.left, flatten(root.right, tail));
5      root.left = null;
6      return root;
7  }
8
9  public void flatten(TreeNode root) {
10     flatten(root, null);
11 }
```

3. 迭代版: 时间复杂度 O(n), 空间复杂度 O(logn)

```
1  public void flatten(TreeNode root) {
2      if (root == null) return;
3      Stack<TreeNode> s = new Stack<TreeNode>();
4      s.push(root);
5      while (!s.isEmpty()) {
6          TreeNode p = s.pop();
7          if (p.right != null)
8              s.push(p.right);
9          if (p.left != null)
10             s.push(p.left);
11         p.left = null;
12         if (!s.isEmpty())
13             p.right = s.peek();
14     }
15 }
```

1.4.15   Populating Next Right Pointers in Each Node II

Follow up for problem "Populating Next Right Pointers in Each Node".

What if the given tree could be any binary tree? Would your previous solution still work?

Note:

You may only use constant extra space.

For example,

Given the following binary tree,

```
1        1
2      /   \
3     2     3
4    / \     \
5   4   5     7
```

After calling your function, the tree should look like:

```
1     1 -> NULL
2    /   \
```
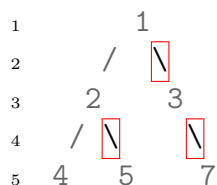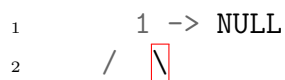
```
3      2 -> 3 -> NULL
4    / \       \
5   4-> 5 -> 7 -> NULL
```

　　要处理一个节点, 可能需要最右边的兄弟节点, 首先想到用广搜。但广搜不是常数空间的, 本题要求常数空间。

　　注意, 这题的代码原封不动, 也可以解决 Populating Next Right Pointers in Each Node I.

1. 递归版: 时间复杂度 O(n), 空间复杂度 O(1)

```java
1  public void connect(TreeLinkNode root) {
2      if (root == null) return;
3      TreeLinkNode dummy = new TreeLinkNode(-1);
4      for (TreeLinkNode curr = root, prev = dummy;
5           curr != null;
6           curr = curr.next) {
7          if (curr.left != null) {
8              prev.next = curr.left;
9              prev = prev.next;
10         }
11         if (curr.right != null) {
12             prev.next = curr.right;
13             prev = prev.next;
14         }
15     }
16     connect(dummy.next);
17 }
```

2. 迭代版: 时间复杂度 O(n), 空间复杂度 O(1)

```java
1  public void connect(TreeLinkNode root) {
2      while (root != null) {
3          TreeLinkNode next = null;
4          TreeLinkNode prev = null;
5          for (; root != null;
6               root = root.next) {
7              if (next == null)
8                  next = root.left != null ? root.left : root.right;
9              if (root.left != null) {
10                 if (prev != null)
11                     prev.next = root.left;
12                 prev = root.left;
13             }
14             if (root.right != null) {
15                 if (prev != null)
16                     prev.next = root.right;
17                 prev = root.right;
18             }
19         }
```

```
20        root = next;
21    }
22 }
```

二叉树的构建

1.4.16 Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

Note:

You may assume that duplicates do not exist in the tree.

1. 递归, 时间复杂度 O(n), 空间复杂度 O(logn)

    c++ 版别人写得比较精致的代码:

```cpp
1  class Solution {
2  public:
3      TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
4          return buildTree(begin(preorder), end(preorder),
5                           begin(inorder), end(inorder));
6      }
7      template<typename InputIterator>
8      TreeNode* buildTree(InputIterator pre_first, InputIterator pre_last,
9                          InputIterator in_first, InputIterator in_last) {
10         if (pre_first == pre_last) return nullptr;
11         if (in_first == in_last) return nullptr;
12         auto root = new TreeNode(*pre_first);
13         auto inRootPos = find(in_first, in_last, *pre_first);
14         auto leftSize = distance(in_first, inRootPos);
15         root->left = buildTree(next(pre_first), next(pre_first, leftSize + 1),
16                                in_first, next(in_first, leftSize));
17         root->right = buildTree(next(pre_first, leftSize + 1), pre_last,
18                                 next(inRootPos), in_last);
19         return root;
20     }
21 };
```

```java
1  public int getIdx(int [] a, int val) {
2      int idx = 0;
3      for (int i = 0; i < a.length; i++) {
4          if (a[i] == val) {
5              idx = i;
6              break;
7          }
8      }
9      return idx;
10 }
11
```

```
12  public TreeNode helper(int [] pre, int [] in, int idx, int bgn, int end) {
13      if (bgn > end) return null;
14      TreeNode root = new TreeNode(pre[idx]);
15      if (bgn == end) return root;
16
17      int inEnd = getIdx(in, pre[idx]);
18      root.left = helper(pre, in, idx + 1, bgn, inEnd - 1);
19      root.right = helper(pre, in, idx + (inEnd - bgn) + 1, inEnd + 1, end);
20      return root;
21  }
22
23  public TreeNode buildTree(int[] preorder, int[] inorder) {
24      return helper(preorder, inorder, 0, 0, inorder.length - 1);
25  }
```

1.4.17   Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

Note:

You may assume that duplicates do not exist in the tree.

1. 递归, 时间复杂度 O(n), 空间复杂度 O(logn)

```
1  class Solution {
2  public:
3      TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
4          return buildTree(begin(inorder), end(inorder),
5                           begin(postorder), end(postorder));
6      }
7      template<typename BidiIt>
8      TreeNode* buildTree(BidiIt in_first, BidiIt in_last,
9                          BidiIt post_first, BidiIt post_last) {
10         if (in_first ==in_last) return nullptr;
11         if (post_first == post_last) return nullptr;
12         const auto val = *prev(post_last);
13         TreeNode* root = new TreeNode(val);
14         auto in_root_pos = find(in_first, in_last, val);
15         auto left_size = distance(in_first, in_root_pos);
16         auto post_left_last = next(post_first, left_size);
17         root->left = buildTree(in_first, in_root_pos, post_first, post_left_last);
18         root->right = buildTree(next(in_root_pos), in_last,
19                                 post_left_last, prev(post_last));
20         return root;
21     }
22 };
```

```
1  public int getIdx(int [] a, int val) {
2      int idx = 0;
```

```java
3        for (int i = 0; i < a.length; i++) {
4            if (a[i] == val) {
5                idx = i;
6                break;
7            }
8        }
9        return idx;
10   }
11
12   public TreeNode helper(int [] in, int [] post, int idx, int bgn, int end) {
13       if (bgn > end) return null;
14       TreeNode root = new TreeNode(post[idx]);
15       if (bgn == end) return root;
16       int inEnd = getIdx(in, post[idx]);
17       root.left = helper(in, post, idx - (end - inEnd) - 1, bgn, inEnd - 1);
18       root.right = helper(in, post, idx - 1, inEnd + 1, end);
19       return root;
20   }
21
22   public TreeNode buildTree(int[] inorder, int[] postorder) {
23       return helper(inorder, postorder, postorder.length - 1, 0, inorder.length - 1);
24   }
```

二叉树查找

1.4.18 Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than the node's key.

- The right subtree of a node contains only nodes with keys greater than the node's key.

- Both the left and right subtrees must also be binary search trees.

    时间复杂度 O(n), 空间复杂度 O(log(n))

c++ 版:

```cpp
1  class Solution {
2  public:
3      bool isValidBST(TreeNode* root) {
4          return isValidBST(root, INT_MIN, INT_MAX);
5      }
6      bool isValidBST(TreeNode* root, int lower, int upper) {
7          if (root == nullptr) return true;
8          if (root->val == INT_MIN && root->left)
9              return false;
10         if(root->val == INT_MAX && root->right)
```

```
11             return false;
12         return root->val >= lower && root->val <= upper
13             && isValidBST(root->left, lower, root->val -1)
14             && isValidBST(root->right, root->val + 1, upper);
15     }
16 };
```

Java 版:

```
1  private boolean isValidBST(TreeNode root, int lower, int upper) {
2      if (root == null) return true;
3      if (root.val == Integer.MIN_VALUE && root.left != null)
4          return false;
5      if(root.val == Integer.MAX_VALUE && root.right != null)
6          return false;
7      return root.val >= lower && root.val <= upper
8          && isValidBST(root.left, lower, root.val - 1)
9          && isValidBST(root.right, root.val + 1, upper);
10 }
11
12 public boolean isValidBST(TreeNode root) {
13     return isValidBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
14 }
```

自己写的繁琐版的代码: inorder traversal 的

```
1  public void inOrderTraversal(TreeNode root, List<Integer> res) {
2      if (root == null) return;
3      if (root.left != null)
4          inOrderTraversal(root.left, res);
5      res.add(root.val);
6      if (root.right != null)
7          inOrderTraversal(root.right, res);
8  }
9
10 public boolean isValidBST(TreeNode root) {
11     if (root == null || (root.left == null && root.right == null)) return true;
12     List<Integer> res = new ArrayList<Integer>();
13     inOrderTraversal(root, res);
14     for (int i = 1; i < res.size(); i++) {
15         if (res.get(i) <= res.get(i - 1)) return false;
16     }
17     return true;
18 }
```

1.4.19   Convert Sorted Array to Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.
二分法。

1. 分治法, 时间复杂度 O(n), 空间复杂度 O(logn)

```cpp
class Solution {
public:
    TreeNode* sortedArrayToBST (vector<int>& num) {
        return sortedArrayToBST(num.begin(), num.end());
    }
    template<typename RandomAccessIterator>
    TreeNode* sortedArrayToBST (RandomAccessIterator first,
                                RandomAccessIterator last) {
        const auto length = distance(first, last);
        if (length <= 0) return nullptr;    // 终止条件
        // 三方合并
        auto mid = first + length / 2;
        TreeNode* root = new TreeNode (*mid);
        root->left = sortedArrayToBST(first, mid);
        root->right = sortedArrayToBST(mid + 1, last);
        return root;
    }
};
```

Java 版:

```java
public TreeNode createMyTree(int [] num, int bgn, int end) {
    TreeNode root = null;
    if (bgn == end) {
        root = new TreeNode(num[end]);
        return root;
    }
    if (bgn < end) {
        int mid = bgn + (end - bgn) / 2;
        root = new TreeNode(num[mid]);
        root.left = createMyTree(num, bgn, mid - 1);
        root.right = createMyTree(num, mid + 1, end);
        return root;
    }
    return null;
}

public TreeNode sortedArrayToBST(int[] num) {
    if (num == null || num.length == 0) return null;
    TreeNode root = createMyTree(num, 0, num.length - 1);
    return root;
}
```

1.4.20   Convert Sorted List to Binary Search Tree: bug.....

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

这题与上一题类似, 但是单链表不能随机访问, 而自顶向下的二分法必须需要 RandomAccessIterator, 因此前面的方法不适用本题。

存在一种自底向上 (bottom-up) 的方法, 见
http://leetcode.com/2010/11/convert-sorted-list-to-balanced-binary.html

1. 分治法: 自项向下

   分治法, 类似于 Convert Sorted Array to Binary Search Tree, 自顶向下, 复杂度 O(n log n)。

   自顶向下, 时间复杂度 O(n^2), 空间复杂度 O(logn)

```cpp
class Solution {
public:
    int listLength (ListNode* node) {
        int n = 0;
        while(node) {
            ++n;
            node = node->next;
        }
        return n;
    }
    ListNode* nth_node (ListNode* node, int n) {
        while (--n)
            node = node->next;
        return node;
    }
    TreeNode* sortedListToBST (ListNode* head) {
        return sortedListToBST (head, listLength(head));
    }
    TreeNode* sortedListToBST (ListNode* head, int len) {
        if (len == 0) return nullptr;
        if (len == 1) return new TreeNode (head->val);
        TreeNode* root = new TreeNode(nth_node (head, len / 2 + 1)->val);
        root->left = sortedListToBST(head, len / 2);
        root->right = sortedListToBST(nth_node (head, len / 2 + 2),
                                      (len - 1) / 2);
        return root;
    }
};
```

   Java:

```java
public int listLength(ListNode head) {
    int n = 0;
    while (head != null) {
        ++n;
        head = head.next;
    }
    return n;
}
```

```java
 9
10  ListNode nth_node(ListNode node, int n) {
11      while (--n > 0)
12          node = node.next;
13      return node;
14  }
15
16  public TreeNode sortListToBST(ListNode head, int len) {
17      if (len == 0) return null;
18      if (len == 1) return new TreeNode(head.val);
19      TreeNode root = new TreeNode(nth_node(head, len / 2 + 1).val);
20      root.left = sortListToBST(head, len / 2);
21      root.right = sortListToBST(nth_node(head, len / 2 + 2), (len - 1) / 2);
22      return root;
23  }
24
25  public TreeNode sortedListToBST(ListNode head) {
26      return sortListToBST(head, listLength(head));
27  }
```

换一种方式，如果我只走一遍 list, 那可以把数据存在一个数组中，方便 O（1）时间获得数据。自已写的乱代码。

```java
 1  public TreeNode createMyTree(ListNode [] num, int bgn, int end) {
 2      TreeNode root = null;
 3      if (bgn == end) {
 4          root = new TreeNode(num[end].val);
 5          return root;
 6      }
 7      if (bgn < end) {
 8          int mid = bgn + (end - bgn) / 2;
 9          root = new TreeNode(num[mid].val);
10          root.left = createMyTree(num, bgn, mid - 1);
11          root.right = createMyTree(num, mid + 1, end);
12          return root;
13      }
14      return null;
15  }
16
17  public TreeNode sortedListToBST(ListNode head) {
18      TreeNode root = null;
19      if (head == null) return null;
20      else if (head.next == null) {
21          root = new TreeNode(head.val);
22          return root;
23      }
24      ListNode [] num = new ListNode[100000]; // 10,0000 works, 50000 doesn't
25      ListNode curr = head;
```

```
26        int idx = 0;
27        while (curr != null) {
28            num[idx++] = curr;
29            curr = curr.next;
30        }
31        root = createMyTree(num, 0, idx - 1);
32        return root;
33    }
```

2. 自底向上: bottom-up, 时间复杂度 O(n), 空间复杂度 O(logn), bug.....

   c++ 版:

```
1  class Solution {
2  public:
3      TreeNode *sortedListToBST(ListNode *head) {
4          int len = 0;
5          ListNode *p = head;
6          while (p) {
7              len++;
8              p = p->next;
9          }
10          return sortedListToBST(head, 0, len - 1);
11      }
12  private:
13      TreeNode* sortedListToBST(ListNode*& list, int start, int end) {
14          if (start > end) return nullptr;
15          int mid = start + (end - start) / 2;
16          TreeNode *leftChild = sortedListToBST(list, start, mid - 1);
17          TreeNode *parent = new TreeNode(list->val);
18          parent->left = leftChild;
19          list = list->next;
20          parent->right = sortedListToBST(list, mid + 1, end);
21          return parent;
22      }
23  };
```

   Java 版:

```
1  // I don't understand where I got wrong yet....
2  public TreeNode sortListToBST(ListNode head, int bgn ,int end) {
3      if (bgn > end) return null;
4      int mid = bgn + (end - bgn) / 2;
5      TreeNode left = sortListToBST(head, bgn, mid - 1);
6      TreeNode root = new TreeNode(head.val);
7      root.left = left;
8      head = head.next;      // attention !
9      root.right = sortListToBST(head, mid + 1, end);
10     return root;
```

```
11  }
12
13  public TreeNode sortedListToBST(ListNode head) {
14      int len = 0;
15      ListNode p = head;
16      while (p != null) {
17          len++;
18          p = p.next;
19      }
20      return sortListToBST(head, 0, len - 1);
21  }
```

二叉树递归二叉树是一个递归的数据结构, 因此是一个用来考察递归思维能力的绝佳数据结构。递归一定是深搜 (见"深搜与递归的区别"), 由于在二叉树上, 递归的味道更浓些, 因此本节用"二叉树的递归"作为标题, 而不是"二叉树的深搜", 尽管本节所有的算法都属于深搜。

二叉树的先序、中序、后序遍历都可以看做是 DFS, 此外还有其他顺序的深度优先遍历, 共有 3! = 6 种。其他 3 种顺序是 root -> right -> left,right -> root -> left, right -> left -> root。

1.4.21 Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

1. 递归版: 时间复杂度 O(n), 空间复杂度 O(log(n))

```
1  public int minDepth(TreeNode root, boolean hasBrother) {
2      if (root == null) return hasBrother ? Integer.MAX_VALUE : 0;
3      return 1 + Math.min(minDepth(root.left, root.right != null),
4                          minDepth(root.right, root.left != null));
5  }
6
7  public int minDepth(TreeNode root) {
8      return minDepth(root, false);
9  }
```

2. 迭代版: 时间复杂度 O(n), 空间复杂度 O(log(n))

```
1  public class Tnd {
2      TreeNode tn;
3      int dep;
4      public Tnd() {
5          tn = null;
6          dep = 0;
7      }
8      public Tnd(TreeNode x, int y) {
9          tn = x;
10         dep = y;
```

```java
11          }
12      }
13
14      public int minDepth(TreeNode root) {
15          if (root == null) return 0;
16          int res = Integer.MAX_VALUE;
17          Stack<Tnd> s = new Stack<Tnd>();
18          s.push(new Tnd(root, 1));
19          while (!s.isEmpty()) {
20              Tnd tmp = s.pop();
21              TreeNode curr = tmp.tn;
22              int depth = tmp.dep;
23              if (curr.left == null && curr.right == null)
24                  res = Math.min(res, depth);
25              if (curr.left != null && res > depth) // 深度控制，剪枝
26                  s.push(new Tnd(curr.left, depth + 1));
27              if (curr.right != null && res > depth) // 深度控制，剪枝
28                  s.push(new Tnd(curr.right, depth + 1));
29          }
30          return res;
31      }
```

1.4.22   Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

时间复杂度 O(n), 空间复杂度 O(logn).

```java
1  public int maxDepth(TreeNode root) {
2      if (root == null) return 0;
3      return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
4  }
```

1.4.23   Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example:

Given the below binary tree and sum = 22,

```
1            5
2           / \
3          4   8
4         /   / \
5        11  13  4
6       /  \      \
7      7    2      1
```
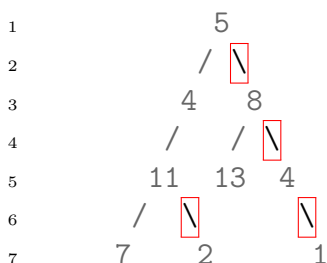
return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

题目只要求返回 true 或者 false, 因此不需要记录路径。

由于只需要求出一个结果, 因此, 当左、右任意一棵子树求到了满意结果, 都可以及时 return。

由于题目没有说节点的数据一定是正整数, 必须要走到叶子节点才能判断, 因此中途没法剪枝, 只能进行朴素深搜。

时间复杂度 O(n), 空间复杂度 O(log(n))

```java
public boolean hasPathSum(TreeNode root, int sum) {
    if (root == null) return false;
    if (root.left == null && root.right == null)
        return sum == root.val;
    return hasPathSum(root.left, sum - root.val)
        || hasPathSum(root.right, sum - root.val);
}
```

1.4.24 Path Sum II

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example:

Given the below binary tree and sum = 22,

```
          5
         / \
        4   8
       /   / \
      11  13  4
     / \     / \
    7   2   5   1
```

return

```
[
    [5,4,11,2],
    [5,8,4,5]
]
```
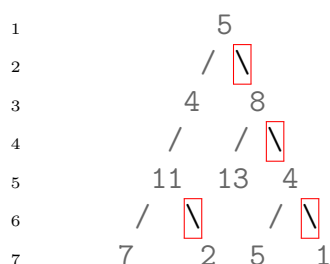
跟上一题相比, 本题是求路径本身。且要求出所有结果, 左子树求到了满意结果, 不能 return, 要接着求右子树。

时间复杂度 O(n), 空间复杂度 O(log(n))

```java
public void pathSum(TreeNode root, int gap, List<Integer> curr,
                    List<List<Integer>> res) {
    if (root == null) return;
    curr.add(root.val);
    if (root.left == null && root.right == null)
        if (root.val == gap)
            res.add(curr);
    pathSum(root.left, gap - root.val, new ArrayList<Integer>(curr), res);
    pathSum(root.right, gap - root.val, new ArrayList<Integer>(curr), res);
}
```

```java
11
12  public List<List<Integer>> pathSum(TreeNode root, int sum) {
13      List<List<Integer>> res = new ArrayList<List<Integer>>();
14      List<Integer> curr = new ArrayList<Integer>();
15      pathSum(root, sum, curr, res);
16      return res;
17  }
```

c++ 版:

```cpp
1   class Solution {
2   public:
3       vector<vector<int> > pathSum(TreeNode *root, int sum) {
4           vector<vector<int> > result;
5           vector<int> cur; // 中间结果
6           pathSum(root, sum, cur, result);
7           return result;
8       }
9   private:
10      void pathSum(TreeNode *root, int gap, vector<int> &cur,
11                   vector<vector<int> > &result) {
12          if (root == nullptr) return;
13          cur.push_back(root->val);
14          if (root->left == nullptr && root->right == nullptr) { // leaf
15              if (gap == root->val)
16                  result.push_back(cur);
17          }
18          pathSum(root->left, gap - root->val, cur, result);
19          pathSum(root->right, gap - root->val, cur, result);
20          cur.pop_back();
21      }
22  };
```

两个版本对于取不取当前值的控制略为不同。

1.4.25   Populating Next Right Pointers in Each Node

Given a binary tree

```java
1   public class TreeLinkNode {
2       int val;
3       TreeLinkNode left, right, next;
4       TreeLinkNode(int x) {
5           val = x;
6       }
7   }
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

Note:

You may only use constant extra space.

You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,

Given the following perfect binary tree,
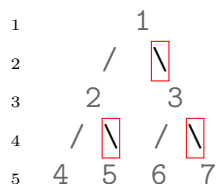
```
1        1
2      /   \
3     2     3
4    / \   / \
5   4   5  6   7
```

After calling your function, the tree should look like:

```
1        1 -> NULL
2      /   \
3     2 ->  3 -> NULL
4    / \   / \
5   4->5->6->7 -> NULL
```
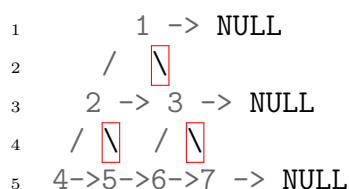
时间复杂度 O(n), 空间复杂度 O(log(n))

```java
1  public void connect(TreeLinkNode root, TreeLinkNode sibling) {
2      if (root == null) return;
3      else root.next = sibling;
4      connect(root.left, root.right);
5      if (sibling != null)
6          connect(root.right, sibling.left);
7      else
8          connect(root.right, null);
9  }
10
11 public void connect(TreeLinkNode root) {
12     connect(root, null);
13 }
```

1.4.26 Sum Root to Leaf Numbers

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path 1->2->3 which represents the number 123.

Find the total sum of all root-to-leaf numbers.

For example,

```
1        1
2      / \
3     2   3
```
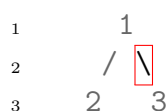
The root-to-leaf path 1->2 represents the number 12.

The root-to-leaf path 1->3 represents the number 13.

Return the sum = 12 + 13 = 25.

时间复杂度 O(n), 空间复杂度 O(log(n))

```
1   public int dfs(TreeNode root, int sum) {
2       if (root == null) return 0;
3       if (root.left == null && root.right == null)
4           return sum * 10 + root.val;
5       return dfs(root.left, sum * 10 + root.val)
6           + dfs(root.right, sum * 10 + root.val);
7   }
8
9   public int sumNumbers(TreeNode root) {
10      return dfs(root, 0);
11  }
```

1.4.27   Binary Tree Upside Down

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

For example:

Given a binary tree {1, 2, 3, 4, 5},

```
1        1
2       / \
3      2   3
4     / \
5    4   5
```
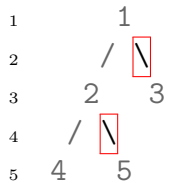
return the root of the binary tree [4, 5, 2, #, #, 3, 1].

```
1        4
2       / \
3      5   2
4         / \
5        3   1
```
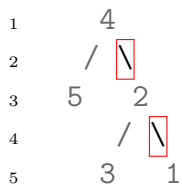
1. Recursive:

```
1   TreeNode* newRoot = NULL;
2   bool found = false;
3
4   void recursive(TreeNode* root) {
5       if (root->left == NULL) return;
6       recursive(root->left);
7       if (root->left->left == NULL) {
8           if (!found) {
9               newRoot = root->left;
10              found = true;
11          }
12          root->left->left = root->right;
13          root->right = NULL;
```

```
14          root->left->right = root;
15          root->left = NULL;
16      }
17  }
18
19  TreeNode *upsideDownBinaryTree(TreeNode *root) {
20      if (root == NULL) return NULL;
21      if (root->left == NULL) return root;
22      recursive(root);
23      return newRoot;
24  }
```

didn't realize it's c++, java later....

2. Iterative:

1.4.28   Binary Search Tree Iterator

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling next() will return the next smallest number in the BST.

Note: next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

```
1  /**
2   * Definition for binary tree
3   * public class TreeNode {
4   *     int val;
5   *     TreeNode left;
6   *     TreeNode right;
7   *     TreeNode(int x) { val = x; }
8   * }
9   */
10  public class BSTIterator {
11      private Stack<TreeNode> stack = new Stack<TreeNode>();
12      public BSTIterator(TreeNode root) {
13          pushAll(root);
14      }
15      /** @return whether we have a next smallest number */
16      public boolean hasNext() {
17          return !stack.isEmpty();
18      }
19      /** @return the next smallest number */
20      public int next() {
21          TreeNode tmpNode = stack.pop();
22          pushAll(tmpNode.right);
23          return tmpNode.val;
24      }
25      private void pushAll(TreeNode node) {
```

```
26          for(; node != null; stack.push(node), node = node.left);
27      }
28  }
29  /**
30   * Your BSTIterator will be called like this:
31   * BSTIterator i = new BSTIterator(root);
32   * while (i.hasNext()) v[f()] = i.next();
33   */
```

1.4.29   Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

The path may start and end at any node in the tree.

For example:

Given the below binary tree,

```
1      1
2    /  \
3   2    3
```

Return 6.

这题很难, 路径可以从任意节点开始, 到任意节点结束。

可以利用"最大连续子序列和"问题的思路, 见第 Maximum Sum SubArray `Maximum Sum SubArray`。如果说 Array 只有一个方向的话, 那么 Binary Tree 其实只是左、右两个方向而已, 我们需要比较两个方向上的值。

不过,Array 可以从头到尾遍历, 那么 Binary Tree 怎么办呢, 我们可以采用 Binary Tree 最常用的 dfs 来进行遍历。先算出左右子树的结果 L 和 R, 如果 L 大于 0, 那么对后续结果是有利的, 我们加上 L, 如果 R 大于 0, 对后续结果也是有利的, 继续加上 R。

时间复杂度 O(n), 空间复杂度 O(log(n))

```
1   private int res;
2
3   public int dfs(TreeNode root) {
4       if (root == null) return 0;
5       int left = dfs(root.left);
6       int right = dfs(root.right);
7       int sum = root.val;
8       if (left > 0) sum += left;
9       if (right > 0) sum += right;
10      res = Math.max(res, sum);
11      return Math.max(right, left) > 0 ? Math.max(right, left) + root.val : root.val;
12  }
13
14  public int maxPathSum(TreeNode root) {
15      res = Integer.MIN_VALUE;
16      dfs(root);
17      return res;
18  }
```

注意, 最后 return 的时候, 只返回一个方向上的值, 为什么? 这是因为在递归中, 只能向父节点返回, 不可能存在 L -> root -> R 的路径, 只可能是 L -> root 或 R -> root。

1.4.30 others Covered

- Unique Binary Search Tree 2.8.5

- Unique Binary Search Tree II 2.8.6

1.5 栈 Stack

1.5.1 Valid Parentheses

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.

时间复杂度 O(n), 空间复杂度 O(n)。

1.5.2 Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, *, /. Each operand may be an integer or another expression.

Some examples:

```
1    ["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9
2    ["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6
```

1.5.3 Simplify Path

Given an absolute path for a file (Unix-style), simplify it.

For example,

```
1    path = "/home/", => "/home"
2    path = "/a/./b/../../c/", => "/c"
```

click to show corner cases.
Corner Cases:

```
1    Did you consider the case where path = "/../"?
2    In this case, you should return "/".
3    Another corner case is the path might contain multiple slashes '/' together, such as "/home//
4    In this case, you should ignore redundant slashes and return "/home/foo".
```

Hard

1.5.4 Longest Valid Parentheses

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2.

Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

1. 使用栈: 时间复杂度 O(n), 空间复杂度 O(n)

   这个问题有点不一样的地方是，输入很可能不合理，也就是括号很可能不匹配，这怎么做？我的做法是用一个数组来做标记，依然是左括号入栈（入栈的是他的位置）。右括号出栈，当一对括号成功匹配的时候，把标记数组中他们相对应的位置标记为 1。这样过一遍之后，扫描一下整个标记数组，有多少个连续的 1，就是有多少个连续的匹配，输出一下个数就行了。

```
1  public int longestValidParentheses(String s) {
2      Stack<Integer> k = new Stack<Integer>();
3      int [] res = new int[s.length()];
4      char j;
5      int tmp = 0;
6      for (int i = 0; i < s.length(); i++) {
7          j = s.charAt(i);
8          if (j == '(')
9              k.push(i);
10         else if (j == ')'){
11             if (!k.isEmpty()) {
12                 res[i] = 1;
13                 tmp = k.pop();
14                 res[tmp] = 1;
15             }
16         }
17     }
18     int max = 0;
19     int i = 0;
20     while (i < s.length()) {
21         tmp = 0;   // reset value
22         while (i < s.length() && res[i] == 0) i++;
23         while (i < s.length() && res[i] == 1) {
24             tmp++;
25             i++;
26         }
27         max = Math.max(max, tmp);
28     }
29     return max;
30 }
```

不用数组的思路：

在处理括号匹配问题上，我们一般使用栈来解决。这一题也可以。

顺序扫描字符串：

初始化：在栈中压入-1

一、若碰到'('，则把当前位置压入栈中

二、若碰到')'：

(1)、如果栈顶元素不是'('，则把当前位置压入栈中；

(2)、如果栈顶元素时'('：栈顶元素出栈，当前的合法子串长度 = 当前字符索引 - 新的栈顶元素；更新最大子串长度

```java
public int longestValidParentheses1(String s) {
    Stack<Integer> k = new Stack<Integer>();
    int res = 0;
    int tmp = 0;
    k.push(-1); //
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == '(')
            k.push(i);
        else { // ')'
            if (k.isEmpty())
                k.push(i);
            else {
                tmp = k.peek();
                if (tmp >= 0 && s.charAt(tmp) == '(') {
                    k.pop();
                    res = Math.max(res, i - k.peek());
                } else k.push(i);
            }
        }
    }
    return res;
}
```

2. 动态规划:

这种括号匹配的问题一般使用动态规划进行解决。使用 dp[i] 表示从位置 i 开始的最长合法括号序列的长度。

考虑第 i 个位置的括号:

- 如果是右括号, 则 dp[i] = 0, 因为第一个是右括号是没有合法匹配

- 如果是左括号, 则考虑下一个位置 i+1 的情况, 事实上, 这是 dp[i+1] 的问题, 可以以同样方法解决。之后可以知道第 i 位该匹配的位置为 i + dp[i+1] + 1 的位置, 直接判断其是否匹配即可。

注意做括号匹配到其对应的右括号时, 后面可能还有左括号以继续匹配。

```java
public int longestValidParentheses(String s) {
    int n = s.length();
    int res = 0;
    int [] dp = new int[n]; // dp[i] 表示从 i 开始的最长合法括号序列
    int j = 0;
    for (int i = n - 2; i >= 0; i--) {
        if (s.charAt(i) == '(') {  // 第一个括号必须为左括号
            j = dp[i + 1] + i + 1; // 匹配该左括号的位置
            if (j < n && s.charAt(j) == ')') { // 满足匹配
                dp[i] = dp[i + 1] + 2;
                if (j + 1 < n)
                    dp[i] += dp[j + 1];      // 后面可能还有
                res = Math.max(res, dp[i]);
```

```
14                    }
15                }
16        }
17        return res;
18  }
```

1.5.5   Trapping Rain Water

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example,

Given [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1], return 6.
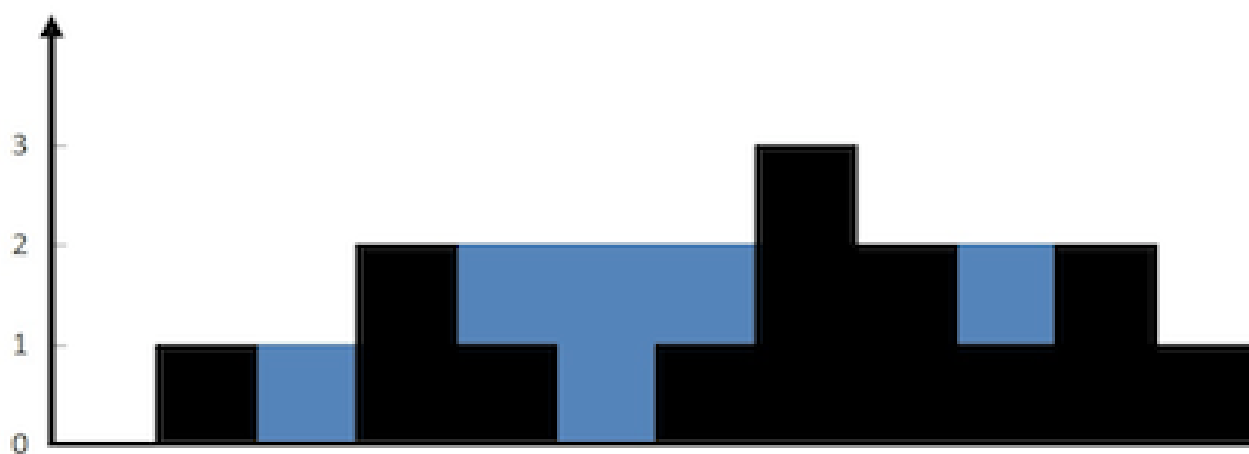
Figure 1.5: Trapping Rain Water

The above elevation map is represented by array [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]. In this case, 6 units of rain water (blue section) are being trapped. Thanks Marcos for contributing this image!

Tags: Array, Stack, Two Pointers

对于每个柱子, 找到其左右两边最高的柱子, 该柱子能容纳的面积就是 min(max_left, max_right) - height。所以,

1. 从左往右扫描一遍, 对于每个柱子, 求取左边最大值;

2. 从右往左扫描一遍, 对于每个柱子, 求最大右值;

3. 再扫描一遍, 把每个柱子的面积并累加。

也可以,

1. 扫描一遍, 找到最高的柱子, 这个柱子将数组分为两半;

2. 处理左边一半;

3. 处理右边一半。

1. 思路 1, 时间复杂度 O(n), 空间复杂度 O(n)

2. 思路 2, 时间复杂度 O(n), 空间复杂度 O(1)

3. 思路 3, 时间复杂度 O(n), 空间复杂度 O(n)

4.

1.5.6 Largest Rectangle in Histogram

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

Above is a histogram where width of each bar is 1, given height = [2, 1, 5, 6, 2, 3].

The largest rectangle is shown in the shaded area, which has area = 10 unit.

For example,

Given height = [2, 1, 5, 6, 2, 3],

return 10.

简单的, 类似于 Container With Most Water(), 对每个柱子, 左右扩展, 直到碰到比自己矮的, 计算这个矩形的面积, 用一个变量记录最大的面积, 复杂度 $O(n^2)$, 会超时。

如图所示, 从左到右处理直方, 当 i = 4 时, 小于当前栈顶 (即直方 3), 对于直方 3, 无论后面还是前面的直方, 都不可能得到比目前栈顶元素更高的高度了, 处理掉直方 3(计算从直方 3 到直方 4 之间的矩形的面积, 然后从栈里弹出); 对于直方 2 也是如此; 直到碰到比直方 4 更矮的直方 1。

这就意味着, 可以维护一个递增的栈, 每次比较栈顶与当前元素。如果当前元素小于栈顶元素, 则入栈, 否则合并现有栈, 直至栈顶元素小于当前元素。结尾时入栈元素 0, 重复合并一次。

1. $O(n^2)$

   最暴力的方法就是枚举所有区间的最大矩形值，然后选择最大的。可以通过分别枚举区间右边界和区间左边界，时间复杂度 $O(n^2)$ 这样做大数据会超时。

   枚举的过程中可以优化一下：如果 height[i+1] > height[i], 那么区间 [k...i] 内的最大矩形肯定不会超过区间 [k...i+1] 内的最大矩形，比如上例中的区间 [0...3] 内的矩形要大于 [0...2] 内的矩形，因为 height[3] > height[2]。因此我们在枚举区间右边界时，只选择那些 height 上升区间的最大值处作为右边界（比如例子中的 2 、6 、3）。优化后可以通过 leetcode 的大数据，虽然做了优化，但是时间复杂度还是 $O(n^2)$。

```java
public int largestRectangleArea(int[] height) {
    int n = height.length;
    int res = 0;
    int right = 0;
    int local = 0;
    int minVal = 0;
    while (right < n) {
        if (right + 1 < n && height[right + 1] >= height[right])
            right++;
        else { // found right border
            minVal = height[right];
            for (int left = right; left >= 0; left--) {
                minVal = Math.min(minVal, height[left]);
                local = minVal * (right - left + 1);
                res = Math.max(res, local);
            }
```

```
17                right++;
18                local = 0;
19                minVal = 0;
20            }
21        }
22        return res;
23    }
```

2. 栈: 时间复杂度 O(n), 空间复杂度 O(n)。

   主要思路:

   我们知道，每个木板将对应多个包含自身的直方图面积，我们从中选取最大的一个面积。

   以上题为例，height[1] 对应的包含自身直方图面积集合为 {1, 2, 3, 4, 5, 6 }，我们只需要知道最大的面积 6 即可，其它的面积就不用考虑。

   我们就比较所有 height[i] 对应最大直方图面积，把整体最大的直方图面积记为 maxArea。

   分析到这，我们很明显的可以知道时间复杂度为 O(n)。

   现在关键问题就变为如何得到 height[i] 对应的最大直方图面积。

   这里用一个栈 s 来保存之前的元素索引。

   s 为空时，直接让 i 栈: 以下取 i 从 0 到 n-1 。

   - 当 height[i] < height[s.top()] 时，说明 height[s.top()] 对应的直方图面积不会再增加了，此时达到最大，可以出栈了;
   - 当 height[i] >= height[s.top()] 时，说明 height[s.top()] 对应的直方图面积还是有潜力增加的哈，此时，i 入栈，++i 。

   当 i 达到 n 时，而栈不为空，说明还有部分 height 中元素对应的最大直方图面积没有求出来.

   于是，当栈不为空时，求 height[s.top()] 对应最大直方图面积。

```java
1  public int largestRectangleArea(int[] height) {
2      Stack<Integer> s = new Stack<Integer>();
3      int [] hnew = new int[height.length + 1];// 数组末尾插入 dummy 元素 0
4      hnew = Arrays.copyOf(height, height.length + 1);
5      int res = 0;
6      int tmp;
7      int left;
8      for (int i = 0; i <= height.length; i++) {
9          if (s.isEmpty() || hnew[i] > hnew[s.peek()])
10             s.push(i); // 栈内保存的是数组 height 的下标索引
11         else {
12             tmp = s.pop();
13             res = Math.max(res, hnew[tmp] *
14                         (s.isEmpty() ? i : i - s.peek() - 1));
15             i--;
16         }
17     }
18     return res;
19 }
```

这题应该还有很多很巧妙的解法，念天脑袋不太转，暂时留在这里吧，改天再回来修改 ~

http://www.cnblogs.com/felixfang/p/3676193.html

1.5.7 others Covered

- Binary Tree Preorder Traversal 1.4.4

- Binary Tree Inorder Traversal 1.4.5

- Binary Tree Postorder Traversal 1.4.6

- Binary Search Tree Iterator 1.4.28

- Binary Tree Zigzag Level Order Traversal 1.4.9

- Maximal Rectangle 2.8.14

- Min Stack 3.5.2

1.6 Hash Table

1.6.1 Valid Sudoku

1.6.2 Two Sum III

1.6.3 Two Sum

1.6.4 4Sum

1.6.5 Binary Tree Inorder Traversal

1.6.6 Fraction to Recurring Decimal

1.6.7 Single Number

1.6.8 Anagrams

1.6.9 Longest Substring Without Repeating Characters

1.6.10 Minimum Window Substring

1.6.11 Copy List with Random Pointer

1.6.12 Sudoku Solver

1.6.13 Max Points on a Line

1.6.14 Substring with Concatenation of All Words

1.6.15 Longest Substring with At Most Two Distinct Characters

other Covered

- Maximal Rectangle 2.8.14
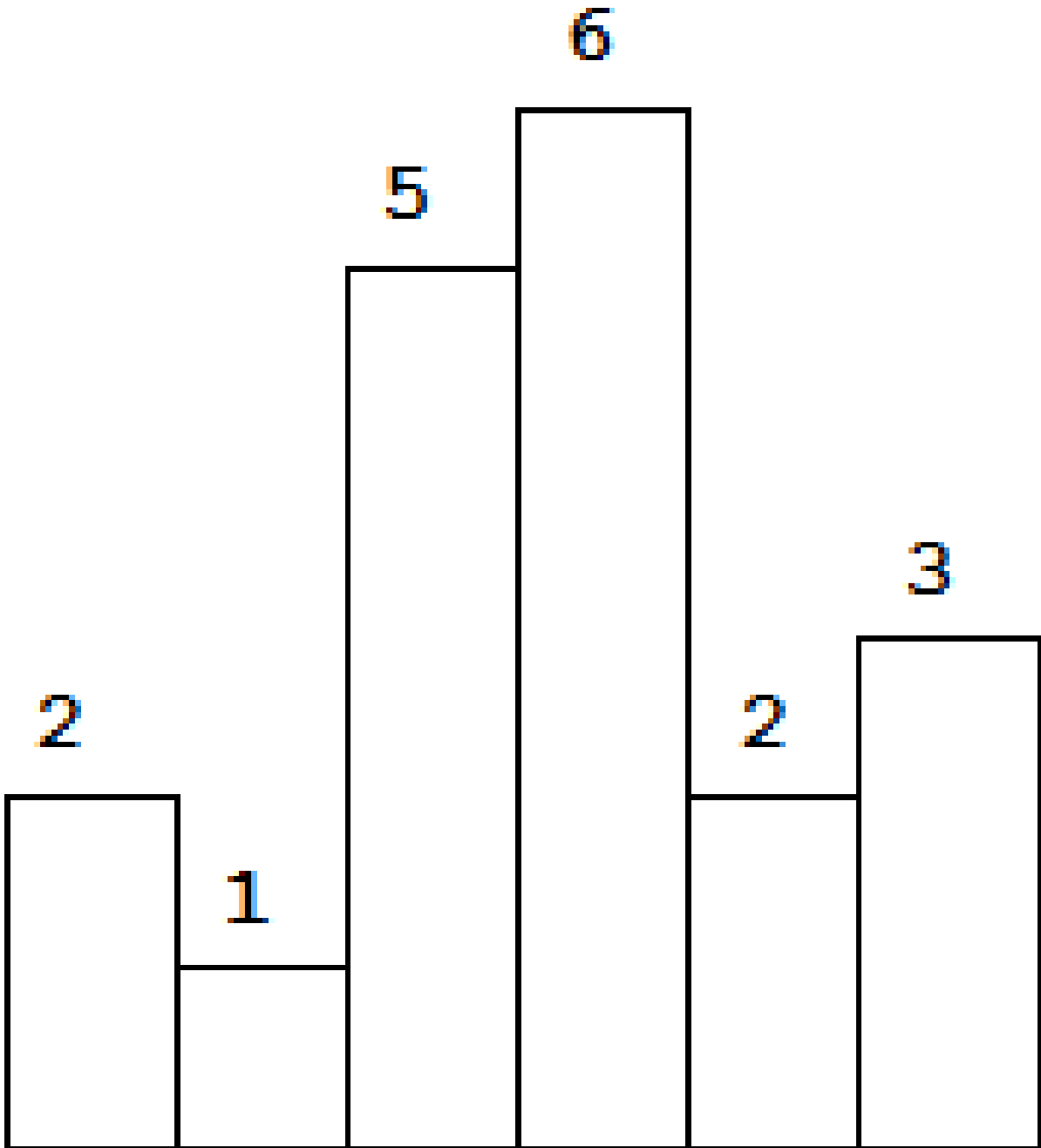
Figure 1.6: Largest Rectangle in Histogram 1
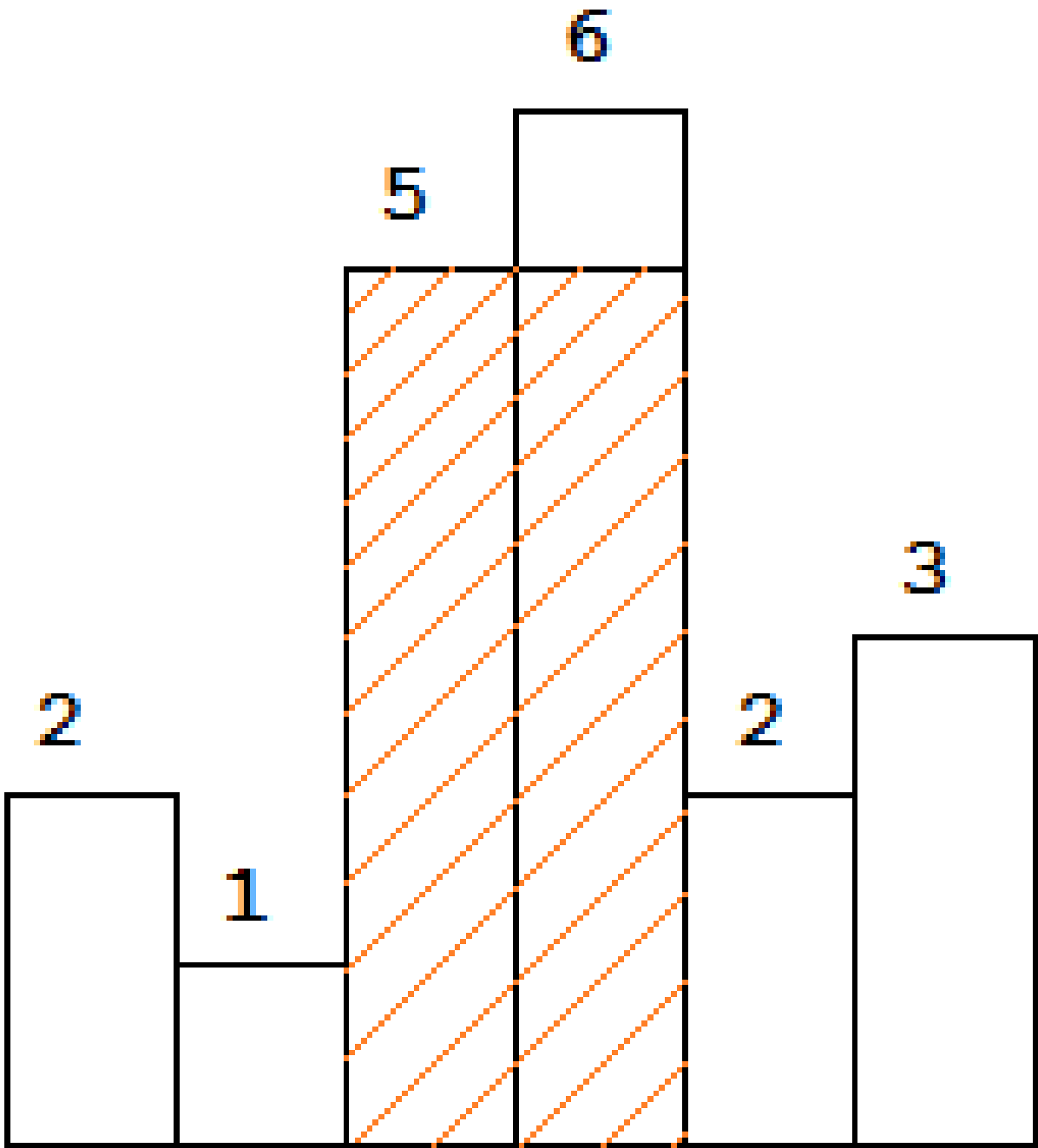
Figure 1.7: Largest Rectangle in Histogram 2

Chapter 2

LeetCode Summary - Part II

2.1 排序

2.1.1 Merge Sorted Array

Given two sorted integer arrays A and B, merge B into A as one sorted array.

Note:

You may assume that A has enough space (size that is greater or equal to m + n) to hold additional elements from B. The number of elements initialized in A and B are m and n respectively.

时间复杂度 O(m+n), 空间复杂度 O(1).

```java
public void merge(int a[], int m, int b[], int n) {
    int ia = m - 1;
    int ib = n - 1;
    int icur = m + n - 1;
    while (ia >= 0 && ib >= 0)
        a[icur--] = a[ia] > b[ib] ? a[ia--] : b[ib--];
    while (ib >= 0)
        a[icur--] = b[ib--];
}
```

2.1.2 Merge Two Sorted List

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

时间复杂度 O(m+n), 空间复杂度 O(1).

```java
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode head = new ListNode(-1);      // dummy List ~!
    for (ListNode p = head; l1 != null || l2 != null; p = p.next) {
        int val1 = l1 == null ? Integer.MAX_VALUE : l1.val;
        int val2 = l2 == null ? Integer.MAX_VALUE : l2.val;
        if (val1 < val2) {
            p.next = l1;
            l1 = l1.next;
        } else {
```

```
10              p.next = l2;
11              l2 = l2.next;
12          }
13      }
14      return head.next;
15  }
```

2.1.3 Sort Colors

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.
Note:

You are not suppose to use the library's sort function for this problem.

click to show follow up.

Follow up:

A rather straight forward solution is a two-pass algorithm using counting sort.

First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.

Could you come up with an one-pass algorithm using only constant space?

Tags: Array, Two Pointers, Sort

由于 0, 1, 2 非常紧凑, 首先想到计数排序 (counting sort), 但需要扫描两遍, 不符合题目要求。

由于只有三种颜色, 可以设置两个 index, 一个是 red 的 index, 一个是 blue 的 index, 两边往中间走。时间复杂度 O(n), 空间复杂度 O(1)。

第 3 种思路, 利用快速排序里 partition 的思想, 第一次将数组按 0 分割, 第二次按 1 分割, 排序完毕, 可以推广到 n 种颜色, 每种颜色有重复元素的情况。

时间复杂度 O(n), 空间复杂度 O(1).

1. Counting Sort: 时间复杂度 O(n), 空间复杂度 O(1)

```
1  public void sortColors(int[] a) {
2      int [] cnt = new int[3];  // 记录每个颜色出现的次数
3      for (int i = 0; i < a.length; i++)
4          cnt[a[i]]++;
5      for (int i = 0, index = 0; i < 3; i++) {
6          for (int j = 0; j < cnt[i]; j++) {
7              a[index++] = i;
8          }
9      }
10  }
```

2. 双指针: 时间复杂度 O(n), 空间复杂度 O(1)

```
1  public void swap(int [] a, int i, int j) {
2      int tmp = a[i];
3      a[i] = a[j];
4      a[j] = tmp;
5  }
6
```

```java
7   public void sortColors(int[] a) {
8       if (a == null || a.length < 2) return;
9       int n = a.length;
10      int i = 0;
11      // 一个是 red 的 index, 一个是 blue 的 index, 两边往中间走
12      int zEnd = -1;   // 0 end
13      int tEnd = n;    // 2 end
14      while (i < tEnd) {
15          if (a[i] == 0 && i != ++zEnd) // first 0
16              swap(a, zEnd, i);
17          else if (a[i] == 2 && i != --tEnd)
18              swap(a, i, tEnd);
19          else i++;
20      }
21      return;
22  }
```

3. use partition():

4. use partition():

2.1.4 Sort List

Sort a linked list in O(n log n) time using constant space complexity.

```java
1   public ListNode divideList(ListNode head) {
2       ListNode fast = head;
3       ListNode slow = head;
4       ListNode prev = head;
5       while (fast != null && fast.next != null) {
6           prev = slow;
7           slow = slow.next;
8           fast = fast.next.next;
9       }
10      prev.next = null;
11      return slow;
12  }
13
14  public ListNode mergeList(ListNode one, ListNode two) {
15      if (one == null) return two;
16      if (two == null) return one;
17      ListNode dummy = new ListNode(Integer.MIN_VALUE);
18      ListNode curr = dummy;
19      while (one != null && two != null) {
20          if (one.val < two.val) {
21              curr.next = one;
22              one = one.next;
23          } else {
24              curr.next = two;
```

```
25              two = two.next;
26          }
27          curr.next.next = null;
28          curr = curr.next;
29      }
30      if (one != null) curr.next = one;
31      else curr.next = two;
32      return dummy.next;
33  }
34
35  public ListNode sortList(ListNode head) {
36      if (head == null || head.next == null)
37          return head;
38      ListNode second = divideList(head);
39      ListNode firstSorted = sortList(head);
40      ListNode secondSorted = sortList(second);
41      return mergeList(firstSorted, secondSorted);
42  }
```

2.1.5   Merge k sorted Lists

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

Tags: Divide and Conquer, Linked List, Heap

可以复用 Merge Two Sorted Lists(见 Merge Two Sorted List 2.1.2) 的函数

时间复杂度 O(n1 + n2 + ...), 空间复杂度 O(1).

```
1   public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
2       ListNode head = new ListNode(-1);       // dummy List ~!
3       for (ListNode p = head; l1 != null || l2 != null; p = p.next) {
4           int val1 = l1 == null ? Integer.MAX_VALUE : l1.val;
5           int val2 = l2 == null ? Integer.MAX_VALUE : l2.val;
6           if (val1 < val2) {
7               p.next = l1;
8               l1 = l1.next;
9           } else {
10              p.next = l2;
11              l2 = l2.next;
12          }
13      }
14      return head.next;
15  }
16
17  public ListNode mergeKLists(List<ListNode> lists) {
18      if (lists.size() == 0) return null;
19      ListNode p = lists.get(0); // Time Limit Exceeded
20      for (int i = 1; i < lists.size(); i++) {
21          p = mergeTwoLists(p, lists.get(i));
22      }
```

```
23        return p;
24   }
25
26   public ListNode mergeKLists(List<ListNode> lists) {
27        if (lists.size() == 0) return null;
28        if (lists.size() == 1) return lists.get(0);
29        if (lists.size() == 2) return mergeTwoLists(lists.get(0), lists.get(1));
30        return mergeTwoLists((mergeKLists(lists.subList(0, lists.size() / 2))),
31                             (mergeKLists(lists.subList(lists.size() / 2, lists.size()))));
32   }
```

2.1.6 Insertion Sort List

Sort a linked list using insertion sort.

这个，代码过于混乱，需要优化。。。

```
1   public ListNode insertionSortList(ListNode head) {
2        if (head == null) return null;
3        ListNode prev = head;
4        ListNode curr = head.next;
5        ListNode next = head;
6        ListNode tnext = head;
7        ListNode tmp = head;
8        while (curr != null) {
9            next = curr.next;
10           if (curr.val < prev.val) {
11               prev.next = next;
12               curr.next = null;
13               if (curr.val < head.val) {
14                   curr.next = head;
15                   head = curr;
16               } else {
17                   tmp = head;
18                   while (tmp.next != prev && curr.val > tmp.next.val) {
19                       tmp = tmp.next;
20                   }
21                   tnext = tmp.next;
22                   tmp.next = curr;
23                   curr.next = tnext;
24               }
25               curr = next;
26           } else {
27               prev = curr;
28               curr = curr.next;
29           }
30       }
31       return head;
32   }
```

2.1.7   First Missing Positive

Given an unsorted integer array, find the first missing positive integer.

For example,

Given [1, 2, 0] return 3,

and [3, 4, -1, 1] return 2.

Your algorithm should run in O(n) time and uses constant space.

本质上是桶排序 (bucket sort), 每当 A[i]!= i+1 的时候, 将 A[i] 与 A[A[i]-1] 交换, 直到无法交换为止, 终止条件是 A[i]== A[A[i]-1]。

```java
public void swap(int [] a, int i, int j) {
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}

public void bucket_sort(int [] a) {
    for (int i = 0; i < a.length; i++) {
        while (a[i] != i + 1) {
            if (a[i] <= 0 || a[i] > a.length || a[i] == a[i] - 1)
                break;
            swap(a, i, a[i] - 1);
        }
    }
}

// need to clean code
// 时间复杂度 O(n), 空间复杂度 O(1)
public int firstMissingPositive(int[] a) {
    // Time Limit Exceeded
    bucket_sort(a);
    for (int i = 0; i < a.length; i++)
        if (a[i] != (i + 1))
            return i + 1;
    return a.length + 1;
}

public int firstMissingPositive0(int[] a) {
    if (a == null || a.length == 0) return 1;
    if (a.length == 1) return a[0] == 1 ? 2 : 1;

    int n = a.length;
    boolean equl = false;
    int i;
    for ( i = 0; i < n; i++) {
        if (a[i] < 0) continue;
        if (a[i] == n) {
            equl = true;
            continue;
```

```
40          }
41          while ( (i == 0 || (i != 0 && a[i] != i))
42                  && a[i] < n && a[i] > 0 ) {
43              swap(a, i, a[i]);
44              if (a[i] == n) {
45                  equl = true;
46                  break;
47              }
48              if ((a[i] > n || a[i] < 0) || (a[i] >= 0 && a[i] == a[a[i]]))
49                  break; // to avoid infite loop
50          }
51      }
52      for ( i = 1; i < n; i++)
53          if (a[i] != i) return i;
54      if (equl || a[0] == n) return n + 1;
55      else return n;
56 }
```

2.1.8   other Covered

2.1.9   Largest Number

2.1.10   Maximum Gap

- Merge Intervals

- Insert Interval

2.2   查找

2.2.1   Search for a Range

Given a sorted array of integers, find the starting and ending position of a given target value.

Your algorithm's runtime complexity must be in the order of O(log n).

If the target is not found in the array, return [-1, -1].

For example,

Given [5, 7, 7, 8, 8, 10] and target value 8,

return [3, 4].

Tags: Array, Binary Search

已经排好了序, 用二分查找。

时间复杂度 O(logn), 空间复杂度 O(1).

分两步来做, 找一次第一个, 找一次最后一个, 返回。

```
1 public int searchFirsst(int [] a, int target, int start, int end) {
2     if (start == end) return (a[end] == target) ? start : -1;
3     if (start == end - 1) {
4         if (a[start] == target) return start;
5         else if (a[end] == target) return end;
6         else return -1;
```

```java
7          }
8
9      while (start < end - 1) {
10             int mid = start + (end - start) / 2;
11             if (a[mid] >= target)
12                 end = mid;
13             else
14                 start = mid + 1;
15         }
16         if (start == end) return (a[end] == target) ? start : -1;
17         if (start == end - 1) {
18             if (a[start] == target) return start;
19             else if (a[end] == target) return end;
20             else return -1;
21         }
22         return -1;
23     }
24
25     public int searchLast(int [] a, int target, int start, int end) {
26         if (start == end) return (a[end] == target) ? end : -1;
27         if (start == end - 1) {
28             if (a[end] == target) return end;
29             else if (a[start] == target) return start;
30             else return -1;
31         }
32
33         while (start < end - 1) {
34             int mid = start + (end - start) / 2;
35             if (a[mid] > target)
36                 end = mid - 1;
37             else
38                 start = mid;
39         }
40         if (start == end) return (a[end] == target) ? end : -1;
41         if (start == end - 1) {
42             if (a[end] == target) return end;
43             else if (a[start] == target) return start;
44             else return -1;
45         }
46         return -1;
47     }
48
49     public int[] searchRange(int[] A, int target) {
50         int [] res = new int[2];
51         if (A == null || A.length == 0 ||
52             (A.length == 1 && A[0] != target)) {
53             res[0] = -1;
54             res[1] = -1;
```

```
55        return res;
56    } else if (A.length == 1) {
57        res[0] = 0;
58        res[1] = 0;
59        return res;
60    }
61    res[0] = searchFirsst(A, target, 0, A.length-1);
62    res[1] = searchLast(A, target, 0, A.length-1);
63    return res;
64 }
```

2.2.2 Search Insert Position

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

Here are few examples.

```
1 [1, 3, 5, 6],  5 → 2
2 [1, 3, 5, 6],  2 → 1
3 [1, 3, 5, 6],  7 → 4
4 [1, 3, 5, 6],  0 → 0
```

直接的二分查找，查找第一个大于或等于 target 的位置。对于中间位置 mid：

如果 A[mid] >= target，那么答案至少应该为 mid，或者为 mid 的左边位置，则 end = mid
否则 A[mid] < target，那么答案肯定在 mid 的右边，而且不可能为 mid，则 bgn = mid + 1
时间复杂度 O(logn)，空间复杂度 O(1)。

```
1 public int searchInsert(int[] a, int target) {
2     int bgn = 0, end = a.length - 1;
3     while (bgn <= end) {
4         int mid = bgn + (end - bgn) / 2;
5         if (a[mid] ==  target) return mid;
6         if (mid > bgn && a[mid] > target && a[mid - 1] < target) return mid;
7         if (a[mid] > target)
8             end = mid - 1;
9         else
10            bgn = mid + 1;
11    }
12    return bgn;
13 }
```

2.2.3 Search a 2D Matrix

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

Integers in each row are sorted from left to right.

The first integer of each row is greater than the last integer of the previous row.

For example,

Consider the following matrix:

```
[
  [1,    3,  5,   7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
```

Given target = 3, return true.
二分查找。
时间复杂度 O(logn), 空间复杂度 O(1).

```java
public boolean binarySearch(int [] a, int target, int bgn, int end) {
    while (bgn <= end) {
        int mid = bgn + (end - bgn) / 2;
        if (a[mid] == target) return true;
        if (a[mid] > target) end = mid - 1;
        else bgn = mid + 1;
    }
    return false;
}

public boolean searchMatrix(int[][] matrix, int target) {
    int m = matrix.length;
    int n = matrix[0].length;
    int [] a = new int[m * n];
    int cnt = 0;
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            a[cnt++] = matrix[i][j];
    return binarySearch(a, target, 0, m * n - 1);
}
```

也可以把两个函数合并成一个,, 省去中间合成桥梁数组的步骤, 代码如下:

```java
public boolean searchMatrix(int[][] matrix, int target) {
    int m = matrix.length;
    int n = matrix[0].length;
    int bgn = 0;
    int end = m * n- 1;
    while (bgn <= end) {
        int mid = bgn + (end - bgn) / 2;
        int val = matrix[mid / n][mid % n];
        if (val == target) return true;
        if (val > target) end = mid - 1;
        else bgn = mid + 1;
    }
    return false;
}
```

2.3 暴力枚举法

2.3.1 Subsets

Given a set of distinct integers, S, 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 S = [1,2,3], a solution is:

```
1  [
2    [3],
3    [1],
4    [2],
5    [1,2,3],
6    [1,3],
7    [2,3],
8    [1,2],
9    []
10  ]
```

Tags: Array Backtracking, Bit Manipulation

1. 递归，增量构造法: 增量构造法, 深搜, 时间复杂度 $O(2^n)$, 空间复杂度 $O(n)$
   每个元素, 都有两种选择, 选或者不选。

```java
1  public void helper(int [] s, List<Integer> path, int step, List<List<Integer>> res) {
2      if (step == s.length) {
3          res.add(new ArrayList<Integer>(path));
4          return;
5      }
6      helper(s, path, step + 1, res);  // 不选 S[step]
7      path.add(s[step]);               // 选 S[step]
8      helper(s, path, step + 1, res);
9      path.remove(path.size() - 1);
10  }
11
12  public List<List<Integer>> subsets(int[] s) {
13      Arrays.sort(s);            // 输出要求有序
14      List<List<Integer>> res = new ArrayList<List<Integer>>();
15      List<Integer> path = new ArrayList<Integer>();
16      helper(s, path, 0, res);
17      return res;
18  }
```

2. 递归，位向量法：时间复杂度 $O(2^n)$, 空间复杂度 $O(n)$.

   开一个位向量 bool selected[n], 每个元素可以选或者不选。

   位向量法, 深搜, 时间复杂度 $O(2^n)$, 空间复杂度 $O(n)$.

```java
public void helper(int [] s, boolean [] selected, int step, List<List<Integer>> res) {
    if (step == s.length) {
        List<Integer> path = new ArrayList<Integer>();
        for (int i = 0; i < s.length; i++) {
            if (selected[i]) path.add(s[i]);
        }
        res.add(path);
        return;
    }
    selected[step] = false;
    helper(s, selected, step + 1, res);
    selected[step] = true;
    helper(s, selected, step + 1, res);
}

public List<List<Integer>> subsets(int[] s) {
    Arrays.sort(s);
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    boolean [] selected = new boolean[s.length];
    helper(s, selected, 0, res);
    return res;
}
```

3. 迭代，增量构造法，时间复杂度 $O(2^n)$, 空间复杂度 $O(1)$

```cpp
vector<vector<int> > subsets(vector<int> &S) {
    sort(S.begin(), S.end()); // 输出要求有序
    vector<vector<int> > result(1);
    for (auto elem : S) {
        result.reserve(result.size() * 2);
        auto half = result.begin() + result.size();
        copy(result.begin(), half, back_inserter(result));
        for_each(half, result.end(), [&elem](decltype(result[0]) &e){
            e.push_back(elem);
        });
    }
    return result;
}
```

   需要改写成 Java 版。

4. 二进制法

   本方法的前提是: 集合的元素不超过 int 位数。用一个 int 整数表示位向量, 第 i 位为 1, 则表示选择 S[i], 为 0 则不选择。例如 S={A, B, C, D}, 则 0110=6 表示子集 {B, C}。

这种方法最巧妙。因为它不仅能生成子集, 还能方便的表示集合的并、交、差等集合运算。设两个集合的位向量分别为 B1 和 B2 , 则 B1 | B2 , B1 & B2 , B1 B2 分别对应集合的并、交、对称差。

二进制法, 也可以看做是位向量法, 只不过更加优化。

二进制法, 时间复杂度 $O(2^n)$, 空间复杂度 $O(1)$

```java
public List<List<Integer>> subsets(int[] s) {
    Arrays.sort(s);
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    int n = s.length;
    for (int i = 0; i < (1 << n); i++) {
        for (int j = 0; j < n; j++) {
            // if ((i & (1 << j)) != 0) path.add(s[j]);
            if(((i >> j) & 1 ) != 0) path.add(s[j]);
        }
        res.add(path);
        path = new ArrayList<Integer>();
    }
    return res;
}
```

总体看来，不管是用向量数组来保存选与不选，还是位操作，都是选与不选的控制。通过有序控制来得到想要的结果。

2.3.2   Subsets II

Given a collection of integers that might contain duplicates, S, 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 S = [1,2,2], a solution is:

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

这题有重复元素, 但本质上, 跟上一题很类似, 上一题中元素没有重复, 相当于每个元素只能选 0 或 1 次, 这里扩充到了每个元素可以选 0 到若干次而已。

1. 递归，增量构造法: 增量构造法, 深搜, 时间复杂度 $O(2^n)$, 空间复杂度 $O(n)$

```java
public void helper(int [] s, List<Integer> path, List<List<Integer>> res) {
    res.add(new ArrayList<Integer>(path));
    for (int i = 0; i < s.length; i++) {
        if (i != 0 && s[i] == s[i - 1]) continue;
        path.add(s[i]);
        helper(s, path, res);
        path.remove(path.size() - 1);
    }
}

// Memory Limit Exceeded
public List<List<Integer>> subsetsWithDup(int[] s) {
    Arrays.sort(s);
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    helper(s, path, res);
    return res;
}
```

2. 增量构造法, 版本 2, 时间复杂度 $O(2^n)$, 空间复杂度 O(n)

3. 位向量法, 时间复杂度 $O(2^n)$, 空间复杂度 O(n)

   这个相对复杂，改天再写

4. 迭代，增量构造法，时间复杂度 $O(2^n)$, 空间复杂度 O(1)

5. 二进制法

```java
public List<List<Integer>> subsetsWithDup(int[] s) {
    Arrays.sort(s);
    HashSet<List<Integer>> res = new HashSet<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    int n = s.length;
    for (int i = 0; i < (1 << n); i++) {
        for (int j = 0; j < n; j++) {
            if ((i & (1 << j)) != 0) path.add(s[j]);
            //if(((i >> j) & 1 ) != 0) path.add(s[j]);
        }
        res.add(path);
        path = new ArrayList<Integer>();
    }
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    result.addAll(res);
    return result;
}
```

   与上一题类似，只是多一步控制对重复的处理，或不再执行重复的，或执行后去重处理。

2.3.3 Permutation

Given a collection of numbers, return all possible permutations.

For example,

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

偷懒的做法, 可以直接使用 c++ std::next_permutation()。如果是在 OJ 网站上, 可以用这个 API 偷个懒; 如果是在面试中, 面试官肯定会让你重新实现。

时间复杂度 O(n!), 空间复杂度 O(1)

1. 重新实现 $next_{permutation}$(): 时间复杂度 O(n!), 空间复杂度 O(1)

2. 递归: 时间复杂度 O(n!), 空间复杂度 O(n)

   本题是求路径本身, 求所有解, 函数参数需要标记当前走到了哪步, 还需要中间结果的引用, 最终结果的引用。

   扩展节点, 每次从左到右, 选一个没有出现过的元素。

   本题不需要判重, 因为状态装换图是一颗有层次的树。收敛条件是当前走到了最后一个元素。

```java
1  public void helper(int [] num, boolean [] used, List<Integer> one,
2                     List<List<Integer>> res) {
3      if (one.size() == num.length) {
4          res.add(new ArrayList<Integer>(one));
5          return;
6      }
7      for (int i = 0; i < num.length; i++) {
8          if (!used[i]) {
9              used[i] = true;
10             one.add(num[i]);
11             helper(num, used, one, res);
12             one.remove(one.size() - 1);
13             used[i] = false;
14         }
15     }
16     return;
17 }
18
19 public List<List<Integer>> permute(int[] num) {
20     List<List<Integer>> res = new ArrayList<List<Integer>>();
21     if (num == null || num.length == 0) return res;
22     boolean [] used = new boolean[num.length];
23     helper(num, used, new ArrayList<Integer>(), res);
24     return res;
25 }
```

2.3.4   Combinationas

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],
]
```

1. 递归: 深搜, 时间复杂度 O(n!), 空间复杂度 O(n)

   时间复杂度 O(n!), 空间复杂度 O(n).

```java
// bgn, 开始的数，cur, 已经选择的数目
public void dfs(List<List<Integer>> res, List<Integer> path, int n, int k, int bgn, int
    if (cur == k) {
        res.add(new ArrayList<Integer>(path));
    }
    for (int i = bgn; i <= n; i ++) {
        path.add(i);
        dfs(res, new ArrayList<Integer>(path), n, k, i + 1, cur + 1);
        path.remove(path.size() - 1);
    }
}

public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    dfs(res, path, n, k, 1, 0);
    return res;
}
```

2. 迭代: 时间复杂度 O((n-k)!), 空间复杂度 O(n)

   时间复杂度 O((n-k)!), 空间复杂度 O(n).

   some other day~ !!@

2.3.5   Letter Combinationas of 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"].
```

Figure 2.1: Letter Combinationas of Phone Number

Note:

Although the above answer is in lexicographical order, your answer could be in any order you want.

1. 递归: 深搜, 时间复杂度 $O(3^n)$, 空间复杂度 $O(n)$

```
public char [] getArr(char x) {
    char [] two = {'a', 'b', 'c'};
    char [] thr = {'d', 'e', 'f'};
    char [] fou = {'g', 'h', 'i'};
    char [] fiv = {'j', 'k', 'l'};
    char [] six = {'m', 'n', 'o'};
    char [] sev = {'p', 'q', 'r', 's'};
    char [] eig = {'t', 'u', 'v'};
    char [] nin = {'w', 'x', 'y', 'z'};
    switch (x) {
    case '2': return two;
    case '3': return thr;
    case '4': return fou;
```

```
14        case '5': return fiv;
15        case '6': return six;
16        case '7': return sev;
17        case '8': return eig;
18        case '9': return nin;
19        }
20        return null;
21    }
22
23    public void recursion(String digits, List<String> result) {
24        char [] base = getArr(digits.charAt(digits.length()-1));
25        int n = result.size();
26        //for(String x : result) // ConcurrentModificationException, result is changing
27        for (int j = 0; j < n; j++) {
28            for (int i = 0; i < base.length; i++)
29                result.add(new StringBuffer(base[i]+ result.get(j)).toString());
30        }
31        for (int i = n-1; i >= 0; i--)
32            result.remove(i);
33        if (digits.length() > 1)
34            recursion(digits.substring(0, digits.length()-1), result);
35    }
36
37    public List<String> letterCombinations(String digits) {
38        List<String> result = new ArrayList<String>();
39        if (digits.equals("")) {
40            result.add("");
41            return result;
42        }
43        char [] base = getArr(digits.charAt(digits.length()-1));
44        for (int i = 0; i < base.length; i++)
45            result.add("" + base[i]);
46        if (digits.length() > 1)
47            recursion(digits.substring(0, digits.length()-1), result);
48        return result;
49    }
```

2. 迭代: 时间复杂度 $O(3^n)$, 空间复杂度 $O(1)$

2.3.6 Permutation II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.
    For example,

```
1  [1,1,2] have the following unique permutations:
2  [1,1,2], [1,2,1], and [2,1,1].
```

1. $next_{permutation}()$

    直接使用 std::next_permutation(), 代码与上一题相同。

2. 重新实现 next_permutation()

   重新实现 std::next_permutation(), 代码与上一题相同。

3. 递归:

   递归函数 permute() 的参数 p, 是中间结果, 它的长度又能标记当前走到了哪一步, 用于判断收敛条件。

   扩展节点, 每次从小到大, 选一个没有被用光的元素, 直到所有元素被用光。

   本题不需要判重, 因为状态装换图是一颗有层次的树。

```java
public void helper(int [] num, boolean [] used, List<Integer> one,
                   List<List<Integer>> res, int cnt) {
    if (one.size() == num.length) {
        res.add(new ArrayList<Integer>(one));
        return;
    }
    for (int i = 0; i < num.length; i++) {
        // !used[i - 1] marks the completion of first 0 done, so ignore second 0
        if (i > 0 && !used[i - 1] && num[i] == num[i - 1]) continue;
        if (!used[i]) {
            used[i] = true;
            one.add(num[i]);
            helper(num, used, one, res, ++cnt);
            one.remove(one.size() - 1);
            used[i] = false;
        }
    }
    return;
}

public List<List<Integer>> permuteUnique(int[] num) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    if (num == null || num.length == 0) return res;
    boolean [] used = new boolean[num.length];
    Arrays.sort(num);
    helper(num, used, new ArrayList<Integer>(), res, 1);
    return res;
}
```

应该还有其它更好的控制原理，只是今天不在状态，不用多想。。。

2.4 广度优先搜索 Breadth First Search

当题目看不出任何规律, 既不能用分治, 贪心, 也不能用动规时, 这时候万能方法——搜索, 就派上用场了。搜索分为广搜和深搜, 广搜里面又有普通广搜, 双向广搜, A* 搜索等。深搜里面又有普通深搜, 回溯法等。

广搜和深搜非常类似 (除了在扩展节点这部分不一样), 二者有相同的框架, 如何表示状态? 如何扩展状态? 如何判重? 尤其是判重, 解决了这个问题, 基本上整个问题就解决了。

2.4.1   Surrounded Regions

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

For example,

```
1  X X X X
2  X O O X
3  X X O X
4  X O X X
```

After running your function, the board should be:

```
1  X X X X
2  X X X X
3  X X X X
4  X O X X
```

Tags: Breadth-first Search

广搜。从上下左右四个边界往里走, 凡是能碰到的'O', 都是跟边界接壤的, 应该删除。

这叫广搜啊？我还以为延着 0 深搜呢。。。仔细想了下，以树的 level order Traversal 作为类比，这个确实应该是广搜，而不是深搜。

四面的最外层搜索，有 O 通路的就为不被包围的区域，其他都可以置为 X；没把握这个特点，难度为五星级的。思路还是很新颖的。

注意栈溢出总题。

```java
public void solve(char[][] board) {
    if (board.length == 0) return;
    int row = board.length;
    int col = board[0].length;
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            if ((i == 0 || j == 0 || i == row-1 || j == col-1) && board[i][j] == 'O') {
                Stack<Integer> stk = new Stack<Integer>();
                stk.push(i * col + j);         // cannot *row, produce error, the larger
                while (!stk.isEmpty()) {
                    int tmp = stk.pop();
                    int x = tmp / col;
                    int y = tmp % col;
                    if (x < 0 || y < 0 || x >= row || y >= col ||
                        !(board[x][y] == 'O'))
                        continue;
                    board[x][y] = '#';
                    stk.push((x-1) * col + y);
                    stk.push((x+1) * col + y);
                    stk.push(x * col + (y - 1));
                    stk.push(x * col + (y + 1));
                }
            }
        }
    }
```

```
25          }
26
27      for (int i = 0; i < row; i++) {
28          for (int j = 0; j < col; j++) {
29              if (board[i][j] == 'O')
30                  board[i][j] = 'X';
31              if (board[i][j] == '#')
32                  board[i][j] = 'O';
33          }
34      }
35  }
```

2.4.2 Word Ladder

Given two words (start and end), and a dictionary, find the length of shortest transformation sequence 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:

```
1  start = "hit"
2  end = "cog"
3  dict = ["hot","dot","dog","lot","log"]
4  As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",
5  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.

时间复杂度 O(n), 空间复杂度 O(n).

分析：这种题，肯定是每次改变单词的一个字母，然后逐渐搜索，很多人一开始就想到用 dfs，其实像这种求最短路径、树最小深度问题 bfs 最适合，可以参考我的这篇博客 bfs（层序遍历）求二叉树的最小深度。本题 bfs 要注意的问题：

和当前单词相邻的单词是：对当前单词改变一个字母且在字典中存在的单词, 找到一个单词的相邻单词，加入 bfs 队列后，要从字典中删除，因为不删除的话会造成类似于 hog->hot->hog 的死循环。而删除对求最短路径没有影响，因为我们第一次找到该单词肯定是最短路径，即使后面其他单词也可能转化得到它，路径肯定不会比当前的路径短（如果要输出所有最短路径，则不能立即从字典中删除，具体见下一题）

bfs 队列中用 NULL 来标识层与层的间隔，每次碰到层的结尾，遍历深度 +1

我们利用和求二叉树最小深度层序遍历的方法来进行 bfs，代码如下：
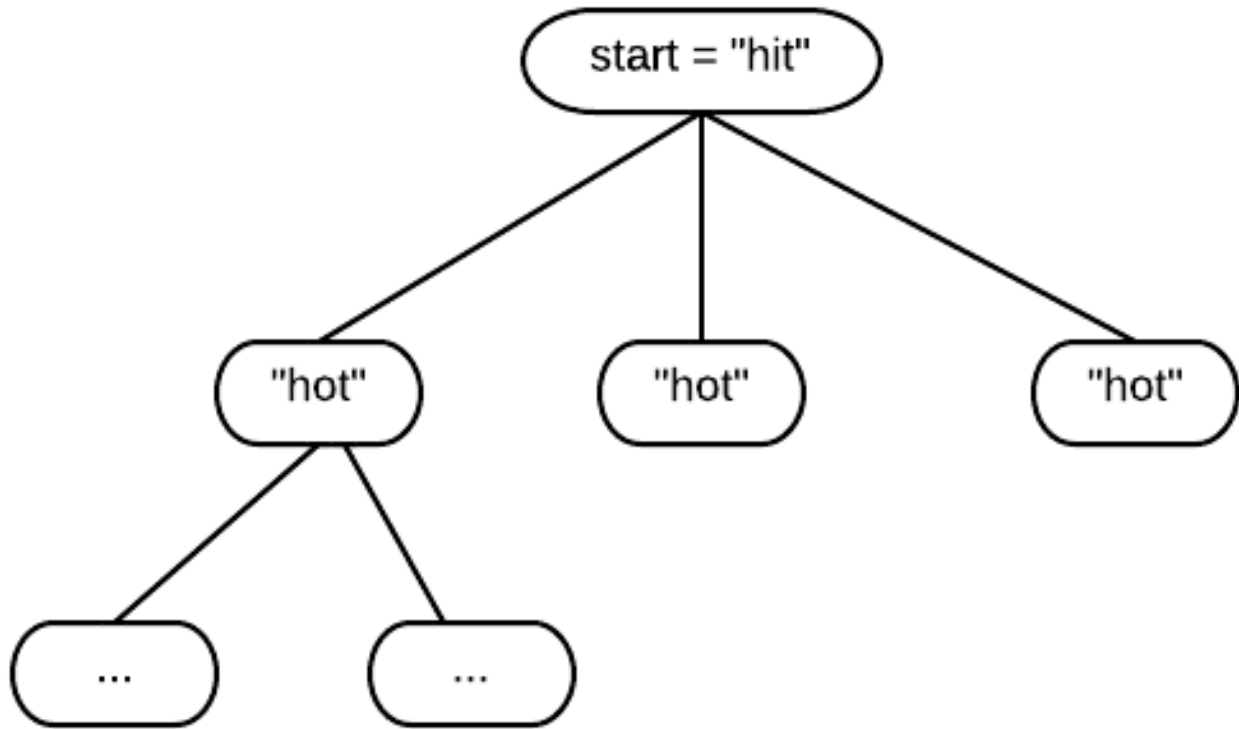
Figure 2.2: Word Ladder

```java
public int ladderLength(String start, String end, Set<String> dict) {
    if (start.length() != end.length()) return 0;
    if (start.compareTo("") == 0) return 1;
    List<String> newStart = new ArrayList<String>();
    List<String> nextStart = new ArrayList<String>();
    newStart.add(start);
    int distance = 1;
    for (int index = 0; index < newStart.size(); index++) {
        String curr = newStart.get(index);
        for (int i = 0; i < curr.length(); i++) {
            for (char j = 'a'; j <= 'z'; j++) {
                if (curr.charAt(i) == j) continue;
                String tmp = curr.substring(0, i) + j + curr.substring(i + 1);
                if (tmp.equals(end)) return distance + 1;
                if (dict.contains(tmp)) {
                    nextStart.add(tmp);
                    dict.remove(tmp);
                }
            }
        }
        if (index == newStart.size() - 1) {
            newStart.addAll(new ArrayList<String>(nextStart));
            nextStart = new ArrayList<String>();
```

```
24              distance++;
25          }
26      }
27      return 0;
28  }
```

应该还有更多更好的方法，改天再深入去写。

2.4.3 Word Ladder II

Given two words (start and end), and a dictionary, find all shortest transformation sequence(s) from start to end, such that:

1. Only one letter can be changed at a time

2. Each intermediate word must exist in the dictionary

For example,
    Given:
    start = "hit", end = "cog"
    dict = ["hot","dot","dog","lot","log"]
    Return

```
1   [
2     ["hit","hot","dot","dog","cog"],
3     ["hit","hot","lot","log","cog"]
4   ]
```

Note:

- All words have the same length.

- All words contain only lowercase alphabetic characters.

跟 Word Ladder 比, 这题是求路径本身, 不是路径长度, 也是 BFS, 略微麻烦点。

这题跟普通的广搜有很大的不同, 就是要输出所有路径, 因此在记录前驱和判重地方与普通广搜不同。

时间复杂度 O(n), 空间复杂度 O(n)。

这道题是 LeetCode 中 AC 率最低的题目，确实是比较难。一方面是因为对时间有比较严格的要求（容易超时），另一方面是它有很多细节需要实现。思路上和 Word Ladder 是比较类似的，但是因为是要求出所有路径，仅仅保存路径长度是不够的，而且这里还有更多的问题，那就是为了得到所有路径，不是每个结点访问一次就可以标记为 visited 了，因为有些访问过的结点也会是别的路径上的结点，所以访问的集合要进行回溯（也就是标记回未访问）。所以时间上不再是一次广度优先搜索的复杂度了，取决于结果路径的数量。同样空间上也是相当高的复杂度，因为我们要保存过程中满足的中间路径到某个数据结构中，以便最后可以获取路径，这里我们维护一个 HashMap，把一个结点前驱结点都进行保存。

在 LeetCode 中用 Java 实现上述算法非常容易超时。为了提高算法效率，需要注意一下两点：

1）在替换 String 的某一位的字符时，先转换成 char 数组再操作；

2）如果按照正常的方法从 start 找 end，然后根据这个来构造路径，代价会比较高，因为保存前驱结点容易，而保存后驱结点则比较困难。所以我们在广度优先搜索时反过来先从 end 找 start，最后再根据生成的前驱结点映射从 start 往 end 构造路径，这样算法效率会有明显提高。

```
1          /* Graph of example:
2           *                |--- dot --- dog ---|
3           * hit --- hot -- |      |         |       |--- cog
4           *                |--- lot --- log ---|
5           */
6          // set: visited   // current level used already
7          // set: unvisited // leftover for choosing for next level to avoid repeat, dict + e
8          // List<List<String>> res; // result paths
9          // List<String, Queue<String>> adjMap; // adjacent directed graph,
10         // List<string, List<String>> works 2 ~! clone graph
```

```
1  private List<String> getNextLadder(String currLadder, Set<String> unVisited) {
2      ArrayList<String> nextLadders = new ArrayList<String>();
3      StringBuffer replace = new StringBuffer(currLadder); // mutate from here
4      for(int i = 0; i < currLadder.length(); i++){
5          char old = replace.charAt(i);
6          for(char ch = 'a'; ch <= 'z'; ch++){
7              if (currLadder.charAt(i) != ch)
8                  replace.setCharAt(i, ch);
9              else continue;
10             String replaced = replace.toString();
11             if (unVisited.contains(replaced))
12                 nextLadders.add(replaced);
13         }
14         replace.setCharAt(i, old);
15     }
16     return nextLadders;
17 }
18
19 private void getLadders(String start, String end, LinkedList<String> path,
20                        List<List<String>> res,
21                        HashMap<String, Queue<String>> adjMap) {
22     if (end.equals(start))      // end condition, len ?
23         res.add(new ArrayList<String> (path));
24     else {                      // else if (len > 0) {
25         Queue<String> adjs = adjMap.get(end);
26         for (String lad : adjs) {
27             path.addFirst(lad); // path.add(0, lad); // equivallent
28             getLadders(start, lad, path, res, adjMap);
29             path.removeFirst(); // path.remove(0);   // remove ele at idx 0
30         }
31     }
32 }
33
34 public List<List<String>> findLadders(String start, String end, HashSet<String> dict) {
35     HashMap<String, Queue<String>> adjMap = new HashMap<String, Queue<String>>();
36     int currLen = 0;
37     boolean found = false;
```

```
38      List<List<String>> res = new ArrayList<List<String>>(); // results
39      Queue<String> queue = new LinkedList<String>();          // queue for BFS
40      Set<String> unv = new HashSet<String>(dict);             // unvstd words, delete f here
41      Set<String> vst = new HashSet<String>();                 // check vstd during same level
42      unv.add(end);           // dict may not contain end, so we add it~!@
43      queue.offer(start); // search & build from start
44      int currLev = 1;
45      int nextLev = 0;
46      for(String word : unv)     // all strings in dict + start
47          adjMap.put(word, new LinkedList<String>());  // just like clone graph
48      unv.remove(start);
49      while (!queue.isEmpty()) {
50          String currLadder = queue.poll();  // curr word, mutate, check dicts, add or not
51          for(String nextLadder : getNextLadder(currLadder, unv)){
52              if (vst.add(nextLadder)) {  // add to vst set, return true, succeed,
53                  nextLev++;              // vst set didn't contain it before
54                  queue.offer(nextLadder);
55              }
56              adjMap.get(nextLadder).offer(currLadder);
57              if(nextLadder.equals(end) && !found) { // control if start-->end, No here
58                  found = true;
59                  currLen += 2;
60              }
61          }
62          if(--currLev == 0){
63              if(found) break;
64              unv.removeAll(vst);  // remove all used string from cur level
65              currLev = nextLev;   // to avoid next level repeat & infinite loop
66              nextLev = 0;
67              currLen++;
68          }
69      }
70      if (found) {
71          LinkedList<String> path = new LinkedList<String>();
72          path.addFirst(end);     // add at beginning of list
73          getLadders(start, end, path, res, adjMap);
74      }
75      return res;
76  }
```

此题改天再回来优化。。。。

2.4.4 other Covered

- Binary Tree Zigzag Level Order Traversal 1.4.9

- Minimum Depth of Binary Tree 1.4.21

2.4.5 小结

2.4.6 适用场景

输入数据: 没什么特征, 不像深搜, 需要有"递归"的性质。如果是树或者图, 概率更大。
    状态转换图: 树或者图。
    求解目标: 求最短。

2.4.7 思考的步骤

1. 求路径长度, 还是路径本身 (或动作序列)?

    (a) 如果是求路径长度, 则状态里面要存路径长度

    (b) 如果是求路径本身或动作序列

    i. 要用一棵树存储宽搜过程中的路径

    ii. 是否可以预估状态个数的上限? 能够预估状态总数, 则开一个大数组, 用树的双亲表示法; 如果不能预估状态总数, 则要使用一棵通用的树。这一步也是第 4 步的必要不充分条件。

2. 如何表示状态? 即一个状态需要存储哪些些必要的数据, 才能够完整提供如何扩展到下一步状态的所有信息。一般记录当前位置或整体局面。

3. 如何扩展状态? 这一步跟第 2 步相关。状态里记录的数据不同, 扩展方法就不同。对于固定不变的数据结构 (一般题目直接给出, 作为输入数据), 如二叉树, 图等, 扩展方法很简单, 直接往下一层走, 对于隐式图, 要先在第 1 步里想清楚状态所带的数据, 想清楚了这点, 那如何扩展就很简单了。

4. 关于判重, 状态是否存在完美哈希方案? 即将状态一一映射到整数, 互相之间不会冲突。

    (a) 如果不存在, 则需要使用通用的哈希表 (自己实现或用标准库, 例如 unordered_set) 来判重; 自己实现哈希表的话, 如果能够预估状态个数的上限, 则可以开两个数组, head 和 next, 表示哈希表, 参考第 §?? 节方案 2。

    (b) 如果存在, 则可以开一个大布尔数组, 作为哈希表来判重, 且此时可以精确计算出状态总数, 而不仅仅是预估上限。

5. 目标状态是否已知? 如果题目已经给出了目标状态, 可以带来很大便利, 这时候可以从起始状态出发, 正向广搜; 也可以从目标状态出发, 逆向广搜; 也可以同时出发, 双向广搜。

2.5 深度优先搜索 Depth First Search

2.5.1 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,

```
1  given s = "aab",
2  Return
3    [
4      ["aa","b"],
5      ["a","a","b"]
6    ]
```

在每一步都可以判断中间结果是否为合法结果, 用回溯法。

一个长度为 n 的字符串, 有 n+1 个地方可以砍断, 每个地方可断可不断, 前后两个隔板默认已经使用, 因此复杂度为 $O(2^{n-1})$.

1. 深搜: 时间复杂度 $O(2^n)$, 空间复杂度 O(n)

```java
1  public void dfs(String s, List<String> path, List<List<String>> res) {
2      if (s.length() == 0) {
3          res.add(new ArrayList<String>(path));
4          return;
5      }
6      for (int i = 0; i < s.length(); i++) {
7          int bgn = 0;
8          int end = i;
9          while (bgn < end) {
10             if (s.charAt(bgn) == s.charAt(end)) {
11                 bgn ++;
12                 end --;
13             } else break;
14         }
15         if (bgn >= end) {
16             path.add(s.substring(0, i + 1));
17             dfs(s.substring(i + 1), path, res);
18             path.remove(path.size() - 1);
19         }
20     }
21     return;
22 }
23
24 public List<List<String>> partition(String s) {
25     List<List<String>> res = new ArrayList<List<String>>();
26     List<String> path = new ArrayList<String>();
27     dfs(s, path, res);
28     return res;
29 }
```

2. 深搜 2: 时间复杂度 $O(2^n)$, 空间复杂度 O(n)

   另一种写法, 更加简洁。这种写法也在 Combination Sum, Combination Sum II 中出现过。

3. 动态规划: 时间复杂度 $O(n^2)$, 空间复杂度 O(1)

   需要先消化一下别人的解法。。。。

2.5.2   Restore IP Addresses

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:

```
1  Given "25525511135",
2  return ["255.255.11.135", "255.255.111.35"]. (Order does not matter)
```

必须要走到底部才能判断解是否合法, 深搜。

需要优化自己的代码，写得太丑。。。

2.5.3   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:

```
1  "((()))", "(()())", "(())()", "()(())", "()()()"
```

小括号串是一个递归结构, 跟单链表、二叉树等递归结构一样, 首先想到用递归。

一步步构造字符串。当左括号出现次数 < n 时, 就可以放置新的左括号。当右括号出现次数小于左括号出现次数时, 就可以放置新的右括号。

1. 时间复杂度 O(TODO), 空间复杂度 O(n)

```
1  public void generate(List<String> res, int n, StringBuffer one, int left, int right) {
2      StringBuffer tmp = new StringBuffer(one);
3      if (left == n) {
4          for (int i = 0; i < n - right; i++)
5              tmp.append(")");
6          res.add(tmp.toString());
7          return;
8      }
9      generate(res, n, new StringBuffer(one).append("("), left + 1, right);
10     if (left > right)
11         generate(res, n, new StringBuffer(one).append(")"), left, right + 1);
12 }
13
14 public List<String> generateParenthesis(int n) {
15     List<String> res = new ArrayList<String>();
16     if (n > 0)
17         generate(res, n, new StringBuffer(), 0, 0);
18     return res;
19 }
```

反过来写。

```
1  public void generate(List<String> res, int n, StringBuffer tmp, int left, int right) {
2      if (left < 0 || right < left) return;
3      if (left == 0 && right == 0) {
4          res.add(tmp.toString());
5          return;
```

```
6          } else {
7              if (left > 0)
8                  generate(res, n, new StringBuffer(tmp).append("("), left - 1, right);
9              if (right > left)
10                 generate(res, n, new StringBuffer(tmp).append(")"), left, right - 1);
11         }
12  }
13
14  public List<String> generateParenthesis(int n) {
15      List<String> res = new ArrayList<String>();
16      if (n > 0)
17          generate(res, n, new StringBuffer(), n, n);
18      return res;
19  }
```

不太确定这是真正正确的操作 StringBuffer 的途径还。

2. 另一种写法:

2.5.4 Combination Sum

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

```
1  [7]
2  [2, 2, 3]
```

时间复杂度 O(n!), 空间复杂度 O(n).

```
1  public void dfs(int [] candidates, int bgn, int target,
2                  List<Integer> path, List<List<Integer>> res) {
3      if (target == 0) {                          // 找到一个合法解
4          res.add(new ArrayList<Integer> (path));
5          return;
6      }
7      for (int i = bgn; i < candidates.length; i++) {    // 扩展状态
8          if (candidates[i] > target) return;
9          path.add(candidates[i]);        // 执行扩展动作
10         dfs(candidates, i, target - candidates[i], path, res); // unlimited
11         path.remove(path.size() - 1); // 撤销动作
```

```
12          }
13      return;
14  }
15
16  public List<List<Integer>> combinationSum(int[] candidates, int target) {
17      Arrays.sort(candidates);
18      List<List<Integer>> res = new ArrayList<List<Integer>>();
19      List<Integer> path = new ArrayList<Integer>();
20      dfs(candidates, 0, target, path, res);
21      return res;
22  }
```

2.5.5   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 added 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, $\cdots$, ak) must be in non-descending order. (ie, a1   a2 $\cdots$ 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  [1, 7]
2  [1, 2, 5]
3  [2, 6]
4  [1, 1, 6]
```

时间复杂度 O(n!), 空间复杂度 O(n).

```
1  public void dfs(int [] candidates, int bgn, int target,
2                  List<Integer> path, List<List<Integer>> res) {
3      if (target == 0) {
4          res.add(new ArrayList<Integer> (path));
5          return;
6      }
7      int previous = -1;
8      for (int i = bgn; i < candidates.length; i++) {
9          //if (i > 0 && candidates[i] == candidates[i - 1]) cnt++;
10         //else cnt = 1;
11         //if (cnt == 1 || (path.size() >= cnt - 1
12         //                  && path.get(path.size() - cnt + 1) == candidates[i])) {
13         // 如果上一轮循环没有选 candidates[i]，则本次循环就不能再选 candidates[i]，
14         // 确保 candidates[i] 最多只用一次
15         if (previous == candidates[i]) continue;
```

```
16        if (candidates[i] > target) return;
17        previous = candidates[i];
18        path.add(candidates[i]);                        // , cnt
19        dfs(candidates, i + 1, target - candidates[i], path, res); // use once
20        path.remove(path.size() - 1);
21        // }
22      }
23      return;
24  }
25
26  public List<List<Integer>> combinationSum2(int[] candidates, int target) {
27      Arrays.sort(candidates);
28      List<List<Integer>> res = new ArrayList<List<Integer>>();
29      List<Integer> path = new ArrayList<Integer>();
30      dfs(candidates, 0, target, path, res);
31      return res;
32  }
```

2.5.6 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:

```
1  [
2   [".Q..",  // Solution 1
3    "...Q",
4    "Q...",
5    "..Q."],
6
7   ["..Q.",  // Solution 2
8    "Q...",
9    "...Q",
10   ".Q.."]
11 ]
```

2.5.7 N-Queens II

Follow up for N-Queens problem.

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

2.5.8 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.

A sudoku puzzle...

...and its solution numbers marked in red.

Tags: Backtracking, Hash Table

时间复杂度 $O(9^4)$, 空间复杂度 $O(1)$.

```java
public boolean isValidSudoku(char[][] bd, int x, int y) {
    for (int i = 0; i < 9; i++)
        if (i != x && bd[x][y] == bd[i][y]) return false;
    for (int j = 0; j < 9; j++)
        if (j != y && bd[x][y] == bd[x][j]) return false;
    for (int i = (x / 3) * 3; i < (x / 3 + 1) * 3; i++)
        for (int j = (y / 3) * 3; j < (y / 3 + 1) * 3; j++)
            if (i != x && j != y && bd[x][y] == bd[i][j])
                return false;
    return true;
}

public boolean solveSudoku(char[][] board) {
    for (int i = 0; i < 9; i++)
        for (int j = 0; j < 9; j++)
            if (board[i][j] == '.') {
                for (int k = 0; k < 9; k++) {
                    board[i][j] = (char)('1' + k);
                    if (isValidSudoku(board, i, j) && solveSudoku(board))
                        return true;
                    board[i][j] = '.';
                }
                return false;
            }
    return true;
}
```

相当于是暴力法，基本上测试了所有的可能性，然后就每种可能性作有效性判断。(277 ms)

我希望执行的速度能够稍微快一点儿，所以每一步都执行了最优的一步，这样每一步都不用有效性判断，因为选的就是最好的，可速度还是更慢了。。。(589 ms)

```java
public boolean helper(char[][] bd, List<Integer> list, int fst, List<Character> avb) {
    avb = new ArrayList<Character>();
    fst = getFirst(bd, list, avb);
    int val = list.get(fst);
    int x = list.get(fst) / 9;
    int y = list.get(fst) % 9;
    if (avb.size() == 0) return false;
    list.remove(fst);
    for (int k = 0; k < avb.size(); k++) {
        char c = avb.get(k);
```

```java
11            bd[x][y] = c;
12            if (list.size() == 0) return true;
13            if (helper(bd, new ArrayList<Integer>(list), fst, avb)) return true;
14            bd[x][y] = '.';
15        }
16        list.add(new Integer(val));
17        return false;
18    }
19
20    public int getFirst(char[][] bd, List<Integer> list, List<Character> res) {
21        Map<Integer, Integer> map = new HashMap<Integer, Integer>();
22        Set<Character> tmp = new HashSet<Character>();  // for each pos
23        Set<Character> bst = new HashSet<Character>();  // for each pos
24        Set<Character> avb = new HashSet<Character>();
25        int x, y; // row, col
26        int result = 0;
27        for (int i = 0; i < list.size(); i++) {
28            tmp.clear();
29            x = list.get(i) / 9;
30            y = list.get(i) % 9;
31            for (int j = 0; j < 9; j++) {
32                if (bd[x][j] != '.') tmp.add(bd[x][j]);
33                if (bd[j][y] != '.') tmp.add(bd[j][y]);
34            }
35            for (int j = 0; j < 3; j++)
36                for (int k = 0; k < 3; k++)
37                    if (bd[j + (x / 3) * 3][k + (y / 3) * 3] != '.')
38                        tmp.add(bd[j + (x / 3) * 3][k + (y / 3) * 3]);
39            if (tmp.size() == 8) {
40                for (char a = '1'; a <= '9'; a++)
41                    if (!tmp.contains(a)) res.add(a);
42                return i;
43            }
44            if (tmp.size() > result) {
45                result = i;
46                bst = new HashSet<Character>(tmp);
47            }
48        }
49        for (char i = '1'; i <= '9'; i++) {
50            if (!bst.contains(i)) res.add(i);
51        }
52        return result;
53    }
54
55    public void solveSudoku(char[][] board) {
56        List<Integer> list = new ArrayList<Integer>();
57        List<Character> avb = new ArrayList<Character>();
58        for (int i = 0; i < 9; i++)
```

```
59          for (int j = 0; j < 9; j++)
60              if (board[i][j] == '.') list.add(i * 9 + j);
61      int first = getFirst(board, list, avb);
62      helper(board, list, first, avb);
63  }
```

看来解题还是简明地满足最基本的要求会比较好。

2.5.9   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,

```
1  Given board =
2  [
3    ["ABCE"],
4    ["SFCS"],
5    ["ADEE"]
6  ]
7  word = "ABCCED", -> returns true,
8  word = "SEE", -> returns true,
9  word = "ABCB", -> returns false.
```

深搜, 递归: 时间复杂度 O(n²*m²), 空间复杂度 O(n²).

```
1  public boolean dfs(char[][] board, String word,
2                  int i, int j, int idx, boolean [][] visited) {
3      int m = board.length;
4      int n = board[0].length;
5      if (idx == word.length()) return true;    // 收敛条件
6
7      if (i < 0 || i >= m || j < 0 || j >= n)
8          return false;                         // 越界，终止条件
9      if (visited[i][j]) return false;    // 已经访问过，剪枝
10     if (word.charAt(idx) != board[i][j])
11         return false;                         // 不相等，剪枝
12
13     visited[i][j] = true;
14     if (dfs(board, word, i - 1, j, idx + 1, visited)    // 上
15         || dfs(board, word, i + 1, j, idx + 1, visited) // 下
16         || dfs(board, word, i, j + 1, idx + 1, visited) // 左
17         || dfs(board, word, i, j - 1, idx + 1, visited))// 右
18         return true;
19     visited[i][j] = false;
20     return false;
21 }
22
```

```java
23  public boolean exist(char[][] board, String word) {
24      int m = board.length;
25      int n = board[0].length;
26      boolean [][] visited = new boolean[m][n];
27      for (int i = 0; i < m; i++)
28          for (int j = 0; j < n; j++)
29              if (dfs(board, word, i, j, 0, visited) == true) return true;
30      return false;
31  }
```

2.5.10   other Covered

- Same Tree 1.4.11

- Symmetric Tree 1.4.12

- Balanced Binary Tree 1.4.13

- Validate Binary Search Tree 1.4.18

- Convert Sorted List to Binary Search Tree 1.3.13

- Convert Sorted Array to Binary Search Tree 1.4.19

- Construct Binary Tree from Preorder and Inorder Traversal 1.4.16

- Construct Binary Tree from Inorder and Postorder Traversal 1.4.17

- Flatten Binary Tree to Linked List 1.4.14

- Populating Next Right Pointers in Each Node 1.4.25

- Populating Next Right Pointers in Each Node II 1.4.15

- Sum Root to Leaf Numbers 1.4.26

- Path Sum 1.4.23

- Path Sum II 1.4.24

- Maximum Depth of Binary Tree 1.4.22

- Recover Binary Search Tree 1.4.10

- Clone Graph 3.4.1

- Binary Tree Maximum Path Sum 1.4.29

2.5.11　小结

2.5.12　适用场景

输入数据: 如果是递归数据结构, 如单链表, 二叉树, 集合, 则百分之百可以用深搜; 如果是非递归数据结构, 如一维数组, 二维数组, 字符串, 图, 则概率小一些。

　　状态转换图: 树或者图。

　　求解目标: 必须要走到最深 (例如对于树, 必须要走到叶子节点) 才能得到一个解, 这种情况适合用深搜。

2.5.13　思考的步骤

1. 是求路径条数, 还是路径本身 (或动作序列)? 深搜最常见的三个问题, 求可行解的总数, 求一个可行解, 求所有可行解。

   (a) 如果是求路径本身, 则要用一个数组 path[] 存储路径。跟宽搜不同, 宽搜虽然最终求的也是一条路径, 但是需要存储扩展过程中的所有路径, 在没找到答案之前所有路径都不能放弃; 而深搜, 在搜索过程中始终只有一条路径, 因此用一个数组就足够了。

   (b) 如果是路径条数, 则不需要存储路径。

2. 只要求一个解, 还是要求所有解? 如果只要求一个解, 那找到一个就可以返回; 如果要求所有解, 找到了一个后, 还要继续扩展, 直到遍历完。广搜一般只要求一个解, 因而不需要考虑这个问题 (广搜当然也可以求所有解, 这时需要扩展到所有叶子节点, 相当于在内存中存储整个状态转换图, 非常占内存, 因此广搜不适合解这类问题)。

3. 如何表示状态? 即一个状态需要存储哪些些必要的数据, 才能够完整提供如何扩展到下一步状态的所有信息。跟广搜不同, 深搜的惯用写法, 不是把数据记录在状态 struct 里, 而是添加函数参数 (有时为了节省递归堆栈, 用全局变量), struct 里的字段与函数参数一一对应。

4. 如何扩展状态? 这一步跟上一步相关。状态里记录的数据不同, 扩展方法就不同。对于固定不变的数据结构 (一般题目直接给出, 作为输入数据), 如二叉树, 图等, 扩展方法很简单, 直接往下一层走, 对于隐式图, 要先在第 1 步里想清楚状态所带的数据, 想清楚了这点, 那如何扩展就很简单了。

5. 关于判重 (a) 如果状态转换图是一棵树, 则不需要判重, 因为在遍历过程中不可能重复。(b) 如果状态转换图是一个图, 则需要判重, 方法跟广搜相同, 见第 §9.4 节。这里跟第 8 步中的加缓存是相同的, 如果有重叠子问题, 则需要判重, 此时加缓存自然也是有效果的。

6. 终止条件是什么? 终止条件是指到了不能扩展的末端节点。对于树, 是叶子节点, 对于图或隐式图, 是出度为 0 的节点。

7. 收敛条件是什么? 收敛条件是指找到了一个合法解的时刻。如果是正向深搜 (父状态处理完了才进行递归, 即父状态不依赖子状态, 递归语句一定是在最后, 尾递归), 则是指是否达到目标状态; 如果是逆向深搜 (处理父状态时需要先知道子状态的结果, 此时递归语句不在最后), 则是指是否到达初始状态。

   由于很多时候终止条件和收敛条件是是合二为一的, 因此很多人不区分这两种条件。仔细区分这两种条件, 还是很有必要的。

   为了判断是否到了收敛条件, 要在函数接口里用一个参数记录当前的位置 (或距离目标还有多远)。如果是求一个解, 直接返回这个解; 如果是求所有解, 要在这里收集解, 即把第一步中表示路径的数组 path[] 复制到解集合里。

8. 如何加速?

    (a) 剪枝。深搜一定要好好考虑怎么剪枝, 成本小收益大, 加几行代码, 就能大大加速。这里没有通用方法, 只能具体问题具体分析, 要充分观察, 充分利用各种信息来剪枝, 在中间节点提前返回。

    (b) 缓存。如果子问题的解会被重复利用, 可以考虑使用缓存。

    i. 前提条件: 子问题的解会被重复利用, 即子问题之间的依赖关系是有向无环图 (DAG)。如果依赖关系是树状的 (例如树, 单链表) , 没必要加缓存, 因为子问题只会一层层往下, 用一次就再也不会用到, 加了缓存也没什么加速效果。

    ii. 具体实现: 可以用数组或 HashMap。维度简单的, 用数组; 维度复杂的, 用 HashMap, C++ 有 map, C++ 11 以后有 unordered$_{map}$, 比 map 快。

拿到一个题目, 当感觉它适合用深搜解决时, 在心里面把上面 8 个问题默默回答一遍, 代码基本上就能写出来了。对于树, 不需要回答第 5 和第 8 个问题。如果读者对上面的经验总结看不懂或感觉"不实用", 很正常, 因为这些经验总结是笔者做了很多深搜题后总结出来的, 从思维的发展过程看, "经验总结"要晚于感性认识, 所以这时候建议读者先做做后面的题目, 积累一定的感性认识后, 在回过头来看这一节的总结, 相信会和笔者有共鸣。

2.5.14　代码模板

2.6　分治法 Divide and Conqure

2.6.1　Pow(x, n)

Implement pow(x, n).
    Tags: Math, Binary Search
    二分法, $x^n = x^{n/2} \times x^{n/2} \times x^{n\%2}$
    时间复杂度 O(logn), 空间复杂度 O(1).

```java
public double myPower(double x, int n) {
    if (n == 0) return 1;
    double val = myPower(x, n / 2);
    if (n % 2 == 0) return val * val;
    else return val * val * x;
}


public double pow(double x, int n) {
    if (n < 0) return 1.0 / myPower(x, -n);
    else return myPower(x, n);
}
```

2.6.2　Sqrt(x)

Implement int sqrt(int x).
    Compute and return the square root of x.
    Tags: Math, Binary Search
    二分查找
    时间复杂度 O(logn), 空间复杂度 O(1).

```java
public int sqrt(int x) {
    int bgn = 0, end = 0;
    int mid = 0;
    int lastMid = 0;
    if (x < 2) return x;
    // end = x / 2;              // which one is better ?
    end = x / 2 < (int)Math.sqrt(Integer.MAX_VALUE) ?
        x / 2 + 1 : (int)Math.sqrt(Integer.MAX_VALUE);
    while (bgn <= end) {
        mid = bgn + (end - bgn) / 2;
        if (x / mid == mid)  // 不要用 x > mid * mid, 会溢出
            return mid;
        else if (x / mid > mid) {
            bgn = mid + 1;
            lastMid = mid;
        } else  // x / mid < mid
            end = mid - 1;
    }
    return lastMid;
}
```

2.6.3   other Covered

- Majority Element 1.1.17

- Maximum Subarray `Maximum Subarray`

- Merge K sorted List `Merge K sorted List`

2.7   贪心法 Greedy Search

2.7.1   Gas Station

There are N gas stations along a circular route, where the amount of gas at station i is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.
Note:
The solution is guaranteed to be unique.

2.7.2   Best Time to Buy and Sell Stock

Say you have an array for which the ith element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

贪心法, 分别找到价格最低和最高的一天, 低进高出, 注意最低的一天要在最高的一天之前。把原始价格序列变成差分序列, 本题也可以做是最大 m 子段和, m = 1。

时间复杂度 O(n), 空间复杂度 O(1).

```
1  public int maxProfit(int[] prices) {
2      int lowest = Integer.MAX_VALUE;
3      int maxProfit = 0;
4      for (int i = 0; i < prices.length; i++) {
5          if (prices[i] < lowest) lowest = prices[i];
6          else maxProfit = Math.max(maxProfit, prices[i] - lowest);
7      }
8      return maxProfit;
9  }
```

2.7.3 Best Time to Buy and Sell Stock II

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

贪心法, 低进高出, 把所有正的价格差价相加起来。把原始价格序列变成差分序列, 本题也可以做是最大 m 子段和, m = 数组长度。

```
1  public int maxProfit(int[] prices) {
2      int result = 0;
3      for (int i = 0; i < prices.length-1; i++)
4          result += (prices[i] < prices[i+1]) ? prices[i + 1] - prices[i] : 0;
5      return result;
6  }
```

2.7.4 Jump Game

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

For example:

A = [2, 3, 1, 1, 4], return true.

A = [3, 2, 1, 0, 4], return false.

由于每层最多可以跳 A[i] 步, 也可以跳 0 或 1 步, 因此如果能到达最高层, 则说明每一层都可以到达。有了这个条件, 说明可以用贪心法。

思路一: 正向, 从 0 出发, 一层一层网上跳, 看最后能不能超过最高层, 能超过, 说明能到达, 否则不能到达。

思路二: 逆向, 从最高层下楼梯, 一层一层下降, 看最后能不能下降到第 0 层。

思路三: 如果不敢用贪心, 可以用动规, 设状态为 f[i], 表示从第 0 层出发, 走到 A[i] 时剩余的最大步数, 则状态转移方程为:

```
1  f[i] = max(f[i - 1], A[i - 1]) - 1, i > 0
```

1. 思路 1, 时间复杂度 O(n), 空间复杂度 O(1)

```
1  public boolean canJump(int[] a) {
2      int reach = 1;         // 最右能跳到哪里
3      for (int i = 0; i < reach && reach < a.length; i++)
```

```
4        reach = Math.max(reach, i + 1 + a[i]);
5    return reach >= a.length;
6  }
```

2. 思路 2, 时间复杂度 O(n), 空间复杂度 O(1)

```
1  public boolean canJump(int[] a) {
2      if (a.length == 0) return true;
3      // 逆向下楼梯，最左能下降到第几层
4      int leftMost = a.length - 1;
5      for (int i = a.length - 2; i >= 0; i--)
6          if (i + a[i] >= leftMost)
7              leftMost = i;
8      return leftMost == 0;
9  }
```

3. 思路三, 动规, 时间复杂度 O(n), 空间复杂度 O(n)

```
1  public boolean canJump(int[] a) {
2      int f[] = new int[a.length];
3      for (int i = 1; i < a.length; i++) {
4          f[i] = Math.max(f[i - 1], a[i - 1]) - 1;
5          if (f[i] < 0) return false;
6      }
7      return f[a.length - 1] >= 0;
8  }
```

2.7.5 Jump Game II

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

For example:

Given array A = [2, 3, 1, 1, 4]

The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

1. 思路 1: 时间复杂度 O(n), 空间复杂度 O(1)

2. 思路 2: 时间复杂度 O(n), 空间复杂度 O(1)

2.7.6 Candy

There are N children standing in a line. Each child is assigned a rating value.

You are giving candies to these children subjected to the following requirements:

Each child must have at least one candy.

Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

2.7.7 other Covered

- Wildcard Matching 2.8.21

2.8 动态归划 Dynamic Programming

2.8.1 Climbing Stairs

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

1. O(n)

```java
public int climbStairs(int n) {
    int [] res = new int[n + 1];
    res[0] = 1; // 1 stair
    res[1] = 2; // 2 stair
    for (int i = 2; i < n; i++)
        res[i] = res[i - 2] + res[i - 1];
    return res[n - 1];
}
```

Considering odd and even, 实现滚动数组

```java
public int climbStairs(int n) {
    if(n <= 0) return 0;
    int [] stairs = {1, 2};
    for(int i = 2;i < n; i++)
        stairs[i % 2] = stairs[0] + stairs[1];
    return n % 2 == 0 ? stairs[1] : stairs[0];
}
```

2. O(log(n))

   https://oj.leetcode.com/discuss/11211/o-log-n-solution-with-matrix-multiplication

   Most solutions are DP with runtime O(n) and O(1) space, the only O(log(n)) solution so far is lucastan's using Binet's formula.

   There actually is a matrix multiplication solution which also runs in O(log(n)). It basically calculates fibonacci numbers by power of matrix ((0, 1), (1, 1)) ^ (n-1).

```java
private int[][] pow(int[][] a, int n) {
    int[][] ret = {{1, 0}, {0, 1}};
    while (n > 0) {
        if ((n & 1) == 1)
            ret = multiply(ret, a);
        n >>= 1;
        a = multiply(a, a);
    }
    return ret;
}
```

```
11
12   private int[][] multiply(int[][] a, int[][] b) {
13       int[][] c = new int[2][2];
14       for (int i = 0; i < 2; i++)
15           for (int j = 0; j < 2; j++)
16               c[i][j] = a[i][0] * b[0][j] + a[i][1] * b[1][j];
17       return c;
18   }
19
20   public int climbStairs(int n) {
21       int[][] a = {{0, 1}, {1, 1}};
22       int[][] m = pow(a, n - 1);
23       return m[0][1] + m[1][1];
24   }
```

2.8.2   Unique Path

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

Above is a 3 x 7 grid. How many possible unique paths are there?

Note: m and n will be at most 100.

这里 Note 的含义是: 如果数值过大会溢出的。–《靖空间》

方法论总结: 遇上新颖没见过的题目:

- 查看其特征;

- 搜索大脑有什么熟悉的题目, 可以套的, 或者相似的可以退到出来的;

- 罗列出来这些题目: 全排列, 水槽, TwoSum, 等等;

- 搜索大脑里面有什么熟悉的算法;

- 罗列出来: 动态规划法, 递归, 回溯, 二叉树遍历, 贪心法, 递归树, 分治法, 观察特征计算法, 等等, 肯定有可以套的算法的!

- 最后选定算法解题;

- 没有优化的算法, 可以使用暴力法, 先解决再说!

最后实在没招了, 想办法让人提示, 主要是提示用什么算法, 能给个思路更好了, O( )O~

1. 深搜, 递归法

   深搜, 小集合可以过, 大集合会超时; 时间复杂度 $O(n^4)$, 空间复杂度 $O(n)$;

```
1   public int uniquePaths(int m, int n) {
2       if (m < 1 || n < 1) return 0;   // 终止条件
3       if (m == 1 && n == 1) return 1; // 收敛条件
4       return uniquePaths(m - 1, n) + uniquePaths(m, n - 1);
5   }
```

2. 备忘录法: 时间复杂度 $O(n^2)$, 空间复杂度 $O(n^2)$

   给前面的深搜, 加个缓存, 就可以过大集合了。即备忘录法。

```java
// 深搜 + 缓存，即备忘录法
// 时间复杂度 O(n^2)，空间复杂度 O(n^2)
class Solution {
    private int [][] f;     // 缓存

    public int dfs(int x, int y) {
        if (x < 1 || y < 1) return 0;    // 数据非法，终止条件
        if (x == 1 && y == 1) return 1; // 回到起点，收敛条件
        return getOrUpdate(x - 1, y) + getOrUpdate(x, y - 1);
    }

    public int getOrUpdate(int x, int y) {
        if (f[x][y] > 0) return f[x][y];
        else return f[x][y] = dfs(x, y);
    }

    public int uniquePaths(int m, int n) {
        // 0 行和 0 列未使用
        f = new int[m + 1][n + 1]; // 缓存
        return dfs(m, n);
    }
}
```

3. 动态规划

   算法 1 的递归解法中，其实我们计算了很多重复的子问题，比如计算 uniquePaths(4, 5) 和 uniquePaths(5, 3) 时都要计算子问题 uniquePaths(3,2)，再者由于 uniquePaths(m, n) = uniquePaths(n, m)，这也使得许多子问题被重复计算了。

   要保存子问题的状态，这样很自然的就想到了动态规划方法，设 dp[i][j] = uniquePaths(i, j)，那么动态规划方程为:

   动态方程: dp[i][j] = dp[i-1][j] + dp[i][j-1]

   边界条件: dp[i][1] = 1, dp[1][j] = 1

```java
public int uniquePaths(int m, int n) {
    if (m == 0 || n == 0) return 1;
    else if (m == 1 && n == 1) return 1;

    int [][] dp = new int[m][n];
    for (int i = 0; i < n; i++)
        dp[0][i] = 1;
    for (int i = 0; i < m; i++)
        dp[i][0] = 1;
    dp[0][0] = 0;
    for (int i = 1; i < m; i++)
        for (int j = 1; j < n; j++)
```

```
13                dp[i][j] = dp[i][j - 1] + dp[i - 1][j];
14        return dp[m-1][n-1];
15    }
```

滚动数组

```
1   // 动规，滚动数组
2   // 时间复杂度 O(n^2)，空间复杂度 O(n)
3   int uniquePaths(int m, int n) {
4       vector<int> f(n, 0);
5       f[0] = 1;
6       for (int i = 0; i < m; i++) {
7           for (int j = 1; j < n; j++) {
8               // 左边的 f[j]，表示更新后的 f[j]，与公式中的 f[i[[j] 对应
9               // 右边的 f[j]，表示老的 f[j]，与公式中的 f[i-1][j] 对应
10              f[j] = f[j - 1] + f[j];
11          }
12      }
13      return f[n - 1];
14  }
```

4. 组合数（C_{m+n-2}^{m-1} ）

一个 m 行, n 列的矩阵, 机器人从左上走到右下总共需要的步数是 m + n - 2, 其中向下走的步数是 m - 1, 因此问题变成了在 m + n - 2 个操作中, 选择 m-1 个时间点向下走, 选择方式有多少种。即 C_{m+n-2}^{m-1}。这里需要注意的是求组合数时防止乘法溢出。

```
1   public int combination(int a, int b) {
2       if (b > (a >>> 1)) b = a - b;
3       long res = 1;
4       for (int i = 1; i <= b ; i++)
5           res = res * (a - i + 1) / i;
6       return (int)res;
7   }
8
9   public int uniquePaths(int m, int n) {
10      return combination(m + n - 2, m - 1);
11  }
```

2.8.3   Unique Path with Obstacles

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

For example, There is one obstacle in the middle of a 3x3 grid as illustrated below.

```
1   [
2     [0,0,0],
3     [0,1,0],
4     [0,0,0]
5   ]
```

The total number of unique paths is 2.
Note: m and n will be at most 100.

1. 备忘录法

```cpp
// LeetCode, Unique Paths II
// 深搜 + 缓存，即备忘录法
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {
        const int m = obstacleGrid.size();
        const int n = obstacleGrid[0].size();
        // 0 行和 0 列未使用
        this->f = vector<vector<int> >(m + 1, vector<int>(n + 1, 0));
        return dfs(obstacleGrid, m, n);
    }

private:
    vector<vector<int> > f;                      // 缓存
    int dfs(const vector<vector<int> > &obstacleGrid,
            int x, int y) {
        if (x < 1 || y < 1) return 0;            // 数据非法，终止条件

        if (obstacleGrid[x-1][y-1]) return 0; // (x,y) 是障碍
        if (x == 1 and y == 1) return 1;         // 回到起点，收敛条件
        return getOrUpdate(obstacleGrid, x - 1, y) +
                getOrUpdate(obstacleGrid, x, y - 1);
    }
    int getOrUpdate(const vector<vector<int> > &obstacleGrid,
                    int x, int y) {
        if (f[x][y] > 0) return f[x][y];
        else return f[x][y] = dfs(obstacleGrid, x, y);
    }
};
```

2. 动态规划

与上一题类似, 但要特别注意第一列的障碍。在上一题中, 第一列全部是 1, 但是在这一题中不同, 第一列如果某一行有障碍物, 那么后面的行应该为 0。

```java
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
    int m = obstacleGrid.length;
    int n = obstacleGrid[0].length;
    if (m == 0 || n == 0) return 0;
    else if (obstacleGrid[0][0] == 1 || obstacleGrid[m - 1][n - 1] == 1) return 0;
    else if (m == 1 || n == 1) return obstacleGrid[m-1][n-1] == 1 ? 0 : 1;

    int [][] arr = new int[m][n];
    for (int i = 0; i < n; i++)
        if (obstacleGrid[0][i] == 1) {
```

```
11              for (int j = i; j < n; j++)
12                  arr[0][j] = 0;
13              i = n;
14          } else arr[0][i] = 1;
15
16      for (int i = 0; i < m; i++)
17          if (obstacleGrid[i][0] != 1)
18              arr[i][0] = 1;
19          else
20              for (int j = i; j < m; j++) {
21                  arr[j][0] = 0;
22                  i = m;
23              }
24
25      arr[0][0] = 0;
26      for (int i = 1; i < m; i++) {
27          for (int j = 1; j < n; j++) {
28              if (obstacleGrid[i][j] == 1)
29                  arr[i][j] = 0;
30              else
31                  arr[i][j] = arr[i][j - 1] + arr[i - 1][j];
32          }
33      }
34      return arr[m-1][n-1];
35  }
```

2.8.4 Minimum Path Sum

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

跟 Unique Paths (见 §10.2) 很类似。设状态为 f[i][j], 表示从起点 (0, 0) 到达 (i, j) 的最小路径和, 则状态转移方程为:

转移方程: f[i][j] = min(f[i-1][j], f[i][j-1]) + grid[i][j]

1. 备忘录法

   代码应该可以再优化一下。

```
1  private int [][] f;
2
3  private int getOrUpdate(int [][] grid, int x, int y) {
4      if (x < 0 || y < 0) return Integer.MAX_VALUE;
5      if (f[x][y] >= 0) return f[x][y];
6      else {
7          f[x][y] = dfs(grid, x, y);
8          return f[x][y];
9      }
10  }
```

```
11
12  private int dfs(int [][] grid, int x, int y) {
13      if (x < 0 || y < 0) return Integer.MAX_VALUE;
14      if (x == 0 && y == 0) return grid[0][0];
15      return Math.min(getOrUpdate(grid, x - 1, y),
16                      getOrUpdate(grid, x, y - 1)) + grid[x][y];
17  }
18
19  public int minPathSum(int[][] grid) {
20      int m = grid.length;
21      int n = grid[0].length;
22      f = new int[m][n];
23      for (int i = 0; i < m; i++)
24          for (int j = 0; j < n; j++)
25              f[i][j] = -1;
26      return dfs(grid, m - 1, n - 1);
27  }
```

2. 动态规划

```
1   public int minPathSum(int[][] grid) {
2       int m = grid.length;
3       int n = grid[0].length;
4       if (m == 0 || n == 0) return 0;
5
6       int [][] f = new int[m][n];
7       f[0][0] = grid[0][0];
8       for (int i = 1; i < n; i++)
9           f[0][i] = f[0][i - 1] + grid[0][i];
10      for (int i = 1; i < m; i++)
11          f[i][0] = f[i - 1][0] + grid[i][0];
12
13      for (int i = 1; i < m; i++)
14          for (int j = 1; j < n; j++)
15              f[i][j] = Math.min(f[i][j - 1], f[i - 1][j]) + grid[i][j];
16      return f[m-1][n-1];
17  }
```

3. 动态规划 + 滚动数组

2.8.5 Unique Binary Search Tree

Given n, how many structurally unique BST's (binary search trees) that store values 1...n?

For example, Given n = 3, there are a total of 5 unique BST's.

```
1       1           3       3       2           1
2        \         /       /       / \           \
3         3       2       1       1   3           2
4        /       /         \                       \
5       2       1           2                       3
```

如果把上例的顺序改一下，就可以看出规律了。《水中的鱼》

```
1  1              1                  2                      3          3
2   \             \                / \                    /          /
3     3             2             1   3                  2          1
4   /                 \                                  /           \
5  2                    3                              1             2
```

比如，以 1 为根的树有几个，完全取决于有二个元素的子树有几种。同理，2 为根的子树取决于一个元素的子树有几个。以 3 为根的情况，则与 1 相同。

定义 Count[i] 为以 [0,i] 能产生的 Unique Binary Tree 的数目，

如果数组为空，毫无疑问，只有一种 BST，即空树, Count[0] =1

如果数组仅有一个元素 {1}，只有一种 BST，单个节点, Count[1] = 1

如果数组有两个元素 {1,2}，那么有如下两种可能

```
1  1                2
2   \              /
3     2          1
4  Count[2] = Count[0] * Count[1]    （ 1 为根的情况 ）
5           + Count[1] * Count[0]    （ 2 为根的情况 ）
```
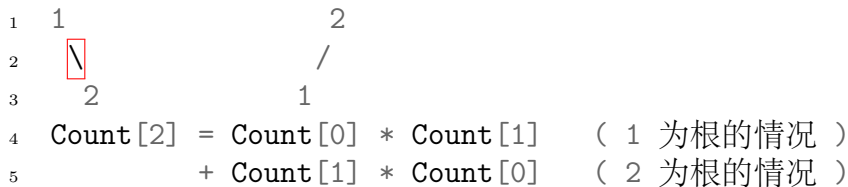
再看一遍三个元素的数组，可以发现 BST 的取值方式如下:

```
1  Count[3] = Count[0]*Count[2]    （ 1 为根的情况 ）
2           + Count[1]*Count[1]    （ 2 为根的情况 ）
3           + Count[2]*Count[0]    （ 3 为根的情况 ）
```
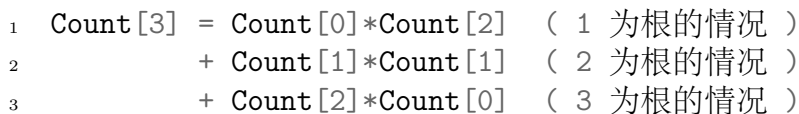
所以，由此观察，当数组为 1, 2, 3, …, n 时, 基于以下原则的构建的 BST 树具有唯一性: 以 i 为根节点的树, 其左子树由 [1, i-1] 构成, 其右子树由 [i+1, n] 构成。

可以得出 Count 的递推公式为

Count(i) = $\sum_{k=1}^{i}$ Count(k - 1) * Count(i - k)

问题至此划归为一维动态规划。

```java
public int numTrees(int n) {
    int [] cnt = new int[n + 1];
    cnt[0] = 1; // empty element
    cnt[1] = 1; // one element
    for (int i = 2; i <= n ; i++)
        for (int j = 0; j < i; j++)
            cnt[i] += cnt[j]*cnt[i - 1 - j];
    return cnt[n];
}
```
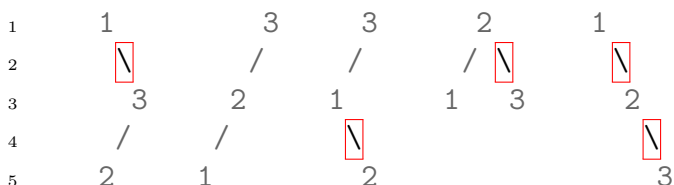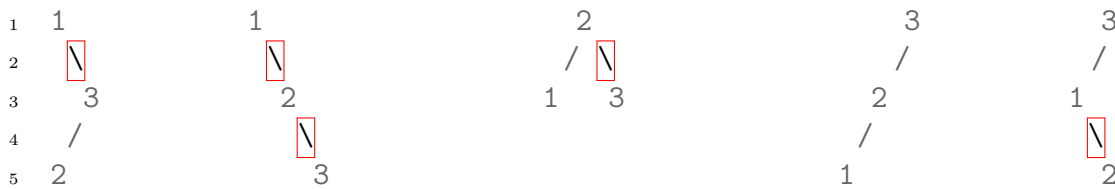
2.8.6 Unique Binary Search Tree II

Given n, generate all structurally unique BST's (binary search trees) that store values 1...n.

For example,

Given n = 3, your program should return all 5 unique BST's shown below.

```
1     1          3      3       2       1
2       \          /       /        / \        \
3        3       2       1        1   3        2
4       /       /          \                   \
5      2      1             2                   3
```
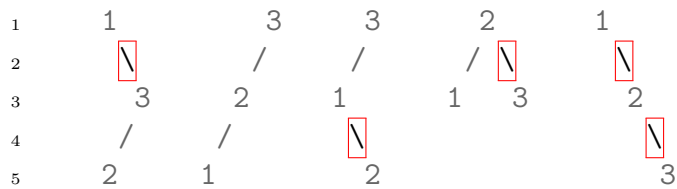
```java
public List<TreeNode> helper(int bgn, int end) {
    List<TreeNode> res = new ArrayList<TreeNode>();
    if (bgn > end) {
        res.add(null);  // important
        return res;
    }

    List<TreeNode> left = null;
    List<TreeNode> right = null;
    TreeNode root = null;
    for (int i = bgn; i <= end ; i++) {
        left = helper(bgn, i - 1);
        right = helper(i + 1, end);
        for (int j = 0; j < left.size(); j++) {
            for (int k = 0; k < right.size(); k++) {
                root = new TreeNode(i);
                root.left = left.get(j);
                root.right = right.get(k);
                res.add(root);
                root = null;
            }
        }
        left = null;
        right = null;
    }
    return res;
}

public List<TreeNode> generateTrees(int n) {
    return helper(1, n);
}
```

2.8.7   Maximum Sum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [-2, 1, -3, 4, -1, 2, 1, -5, 4],
the contiguous subarray [4, -1, 2, 1] has the largest sum = 6.
click to show more practice.
More practice:
If you have figured out the O(n) solution, try coding another solution using the divide and conquer approach, which is more subtle.

Tags: Divide and Conquer, Array, Dynamic Programming

分析:

最大连续子序列和, 非常经典的题。

当我们从头到尾遍历这个数组的时候, 对于数组里的一个整数, 它有几种选择呢? 它只有两种选择:
1、加入之前的 SubArray; 2. 自己另起一个 SubArray。那什么时候会出现这两种情况呢?

- 如果之前 SubArray 的总体和大于 0 的话, 我们认为其对后续结果是有贡献的。这种情况下我们选择加入之前的 SubArray

- 如果之前 SubArray 的总体和为 0 或者小于 0 的话, 我们认为其对后续结果是没有贡献, 甚至是有害的 (小于 0 时)。这种情况下我们选择以这个数字开始, 另起一个 SubArray。

设状态为 f[j], 表示以 S[j] 结尾的最大连续子序列和, 则状态转移方程如下:

```
f[j] = max (f[j-1] + S[j], S[j]) ，其中 1 ⫇ j ⫇ n
target = max (f[j]) ，其中 1 ⫇ j ⫇ n
```

解释如下:

情况一, S[j] 不独立, 与前面的某些数组成一个连续子序列, 则最大连续子序列和为 f[j-1] + S[j]。

情况二, S[j] 独立划分成为一段, 即连续子序列仅包含一个数 S[j], 则最大连续子序列和为 S[j]。

其他思路:

- 思路 2: 直接在 i 到 j 之间暴力枚举, 复杂度是 $O(n^3)$

- 思路 3: 处理后枚举, 连续子序列的和等于两个前缀和之差, 复杂度 $O(n^2)$。

- 思路 4: 分治法, 把序列分为两段, 分别求最大连续子序列和, 然后归并, 复杂度 $O(n*(\log(n)))$

- 思路 5: 把思路 2 $O(n^2)$ 的代码稍作处理, 得到 $O(n)$ 的算法

- 思路 6: 当成 M = 1 的最大 M 子段和

1. 动态规划: 时间复杂度 O(n), 空间复杂度 O(1)

```java
public int maxSubArray(int[] a) {
    int result = Integer.MIN_VALUE, f = 0;
    for (int i = 0; i < a.length; i++) {
        f = Math.max(f + a[i], a[i]);
        result = Math.max(result, f);
    }
    return result;
}
```

2. 思路 5: 时间复杂度 O(n), 空间复杂度 O(n)

```java
public int maxSubArray(int[] a) {
    int result, curMin;
    int [] sum = new int[a.length + 1];
    sum[0] = 0;
    result = Integer.MIN_VALUE;
    curMin = sum[0];
    for (int i = 1; i <= a.length; i++)
```

```
8          sum[i] = sum[i - 1] + a[i - 1];
9      for (int i = 1; i <= a.length; i++) {
10         result = Math.max(result, sum[i] - curMin);
11         curMin = Math.min(curMin, sum[i]);
12     }
13     return result;
14 }
```

2.8.8 Maximum Product Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array [2, 3, -2, 4],

the contiguous subarray [2, 3] has the largest product = 6.

这道题跟2.8.7模型上和思路上都比较类似，还是用一维动态规划中的"局部最优和全局最优法"。

这里的区别是维护一个局部最优不足以求得后面的全局最优，这是由于乘法的性质不像加法那样，累加结果只要是正的一定是递增，乘法中有可能现在看起来小的一个负数，后面跟另一个负数相乘就会得到最大的乘积。

不过事实上也没有麻烦很多，我们只需要在维护一个局部最大的同时，在维护一个局部最小，这样如果下一个元素遇到负数时，就有可能与这个最小相乘得到当前最大的乘积和，这也是利用乘法的性质得到的。

来自《Code Ganker 征服代码》

```
1  public int maxProduct(int[] a) {
2      int n = a.length;
3      int maxLocal = a[0];
4      int minLocal = a[0];
5      int res = a[0];
6      for (int i = 1; i < n; i++) {
7          int tmp = maxLocal;
8          maxLocal = Math.max(Math.max(maxLocal * a[i], a[i]),
9                              Math.max(minLocal * a[i], a[i]));
10         minLocal = Math.min(Math.min(a[i], a[i] * tmp),
11                             Math.min(a[i], a[i] * minLocal));
12         res = Math.max(res, maxLocal);
13     }
14     return res;
15 }
```

2.8.9 Decode Ways, bug~ !!

A message containing letters from A-Z is being encoded to numbers using the following mapping:

```
1  'A' -> 1
2  'B' -> 2
3  ...
4  'Z' -> 26
```

Given an encoded message containing digits, determine the total number of ways to decode it.

For example,

Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).

The number of ways decoding "12" is 2.

参考 Climb stairs, 2.8.1

最原始型的 follow、控制 logic：

```java
public int numDecodings(String s) {
    if (s == null || s.length() == 0) return 0;
    else if (s.charAt(0) == '0') return 0;
    else if (s.length() == 1) return 1;
    else if (s.length() >= 2 && s.charAt(0) - '0' > 2 && s.charAt(1) == '0') return 0;

    int j = 0; // idx for s
    char c;    // char at pos j
    int [] res = new int[s.length()];
    res[0] = 1;
    res[1] = 1;
    if (s.charAt(1) != '0'
        && ( (s.charAt(0) == '1'
            || (s.charAt(0) == '2') && s.charAt(1) - '0' < 7)))
        res[1] = 2;

    for (int i = 2; i < s.length(); i++) {
        c = s.charAt(i);
        if (c != '0')
            res[i] += res[i - 1]; // separate dijit // except '0', NO, 10, 20 ok

        switch (c) {
        case '0':
            if (s.charAt(i - 1) == '1' || s.charAt(i - 1) == '2')
                if (s.charAt(i - 2) == '1' || s.charAt(i - 2) == '2')
                    res[i] = res[i - 2];      // 110 = 1 10, ... 11 0, NO~!
                else
                    res[i] = res[i - 1];
            else return 0;
            break;
        case '7':      // 17, 18, 19
        case '8':
        case '9':
            if (s.charAt(i - 1) == '1')
                res[i] += res[i - 2];
            break;
        default:       // 1, 2, 3, 4, 5, 6: Individual, 1-pre, 2-pre, 11, 21, 12, 22 ...
            if (s.charAt(i - 1) == '1' || s.charAt(i - 1) == '2')
                res[i] += res[i - 2];
            break;
        }
    }
```

```
42        }
43        return res[s.length() - 1];
44    }
```

1. 动态规划, 时间复杂度 O(n), 空间复杂度 O(1), java v m bug ~~!!!!

   模拟 pdf c++ 版大牛精华版代码:

```java
1   // my Java version code has bug here, fix it later ~~~
2   public int numDecodings(String s) {
3       if (s == null || s.length() == 0 || s.charAt(0) == '0') return 0;
4       int prev = 0;
5       int curr = 1;
6       int tmp = 0;
7       for (int i = 1; i <= s.length() ; i++) {
8           if (s.charAt(i - 1) == '0') curr = 0;
9           if (i < 2 || s.charAt(i - 2) != '1' ||
10              (s.charAt(i - 2) == '2' && s.charAt(i - 1) <= '6'))
11              prev = 0;
12          tmp = curr;
13          curr = prev + curr;
14          prev = tmp;
15      }
16      return curr;
17  }
```

2.8.10   Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

```
1   [
2        [2],
3       [3,4],
4      [6,5,7],
5     [4,1,8,3]
6   ]
```

The minimum path sum from top to bottom is 11 (i.e., $2 + 3 + 5 + 1 = 11$).

Note: Bonus point if you are able to do this using only O(n) extra space, where n is the total number of rows in the triangle.

1. 自顶向下: 时间复杂度是 $O(n^2)$, 空间复杂度是 $O(n)$

   这是一道动态规划的题目, 求一个三角形二维数组从顶到低端的最小路径和。思路是维护到某一个元素的最小路径和, 那么在某一个元素 i, j 的最小路径和就是它上层对应的相邻两个元素的最小路径和加上自己的值, 递推式是

   sum[i][j] = min(sum[i-1][j-1],sum[i-1][j]) + triangle[i][j]。

最后扫描一遍最后一层的路径和，取出最小的即可。每个元素需要维护一次，总共有 1+2+…+n = n*(n+1)/2 个元素，所以时间复杂度是 O(n^2)。而空间上每次只需维护一层即可（因为当前层只用到上一层的元素），所以空间复杂度是 O(n)。

来自《Code Ganker 征服代码》

```java
public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {
    if(triangle == null || triangle.size() == 0)
        return 0;
    if(triangle.size() == 1)
        return triangle.get(0).get(0);

    int[] dp = new int[triangle.size()];
    dp[0] = triangle.get(0).get(0);
    for(int i = 1; i < triangle.size(); i++) {
        dp[i] = dp[i - 1] + triangle.get(i).get(i);
        for(int j = i - 1 ; j >= 1; j--)
            dp[j] = (dp[j] < dp[j-1] ? dp[j] : dp[j - 1]) + triangle.get(i).get(j);
        dp[0] = dp[0] + triangle.get(i).get(0);
    }

    int minimum = dp[0];
    for(int i = 1; i < dp.length; i++)
        if (dp[i] < minimum)
            minimum = dp[i];
    return minimum;
}
```

上述代码实现时要注意每层第一个和最后一个元素特殊处理一下。

2. 自底向上：

换个角度考虑一下，如果这道题不自顶向下进行动态规划，而是放过来自底向上来规划，递归式只是变成下一层对应的相邻两个元素的最小路径和加上自己的值，原理和上面的方法是相同的，这样考虑到优势在于不需要最后对于所有路径找最小的，而且第一个元素和最后元素也不需要单独处理了，所以代码简洁了很多。

```java
public int minimumTotal(List<List<Integer>> triangle) {
    List<Integer> res = new ArrayList<Integer>(triangle.get(triangle.size() - 1));
    for (int i = triangle.size() - 2; i >= 0 ; i--)
        for (int j = 0; j <= i ; j++)
            res.set(j, Math.min(res.get(j), res.get(j + 1))
                    + triangle.get(i).get(j));
    return res.get(0);
}
```

2.8.11   Word Break

Given a string s and a dictionary of words dict, determine if s can be segmented into a space-separated sequence of one or more dictionary words.

For example, given

```
1  s = "leetcode",
2  dict = ["leet", "code"].
3  Return true because "leetcode" can be segmented as "leet code".
```

设状态为 f(i), 表示 s[0,i] 是否可以分词, 则状态转移方程为

转移方程: f (i) = any$_{of}$ (f(j) && s[j+1, i] ∈ dict), 0 ≤ j < i

1. 深搜: 时间复杂度 $O(2^n)$, 空间复杂度 $O(n)$

   OJ 超时。

```
1  public boolean helper(String s, int i, Set<String> dict) {
2      int n = s.length();
3      if (i >= n) return false;
4      int j = i;
5      while (j < n) {
6          while (j < n && !dict.contains(s.substring(i, j + 1))) j++;
7          if (j == n) return false;
8          if (helper(s, j + 1, dict) == true) return true;
9          j++;
10     }
11     return false;
12 }
13
14 public boolean wordBreak(String s, Set<String> dict) {
15     if (dict.contains(s)) return true;
16     if (s == null || s.length() == 0) return false;
17     return helper(s, 0, dict);
18 }
```

2. 动态规划:

```
1  public boolean wordBreak(String s, Set<String> dict) {
2      if (dict.contains(s)) return true;
3      boolean [] dp = new boolean[s.length()];
4      for (int i = 0; i < s.length(); i++)
5          dp[i] = false;
6
7      for (int i = 0; i < s.length(); i++)
8          for (int j = i; j >= 0; j--) {
9              String tmp = new String(s.substring(j, i+1));
10             if (dict.contains(tmp))
11                 if ((j == 0) || (j >= 1 && dp[j-1])) {
12                     dp[i] = true;
13                     break;
14                 }
15         }
16     return dp[s.length()-1];
17 }
```

2.8.12   Best Time to Buy and Sell Stock III

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

设状态 f(i), 表示区间 [0, i](0 ≤ i ≤ n−1) 的最大利润, 状态 g(i), 表示区间 [i, n−1](0 ≤ i ≤ n−1) 的最大利润, 则最终答案为 max{ f(i) + g(i) }, 0 ≤ i ≤ n−1。

允许在一天内买进又卖出, 相当于不交易, 因为题目的规定是最多两次, 而不是一定要两次。

将原数组变成差分数组, 本题也可以看做是最大 m 子段和, m = 2.

```java
public int maxProfit(int[] prices) {
    if (prices.length < 2) return 0;
    int [] past = new int[prices.length];
    int [] futu = new int[prices.length];
    int result = 0;
    for (int i = 1, valley = prices[0]; i < prices.length; i++) {
        valley = Math.min(valley, prices[i]);
        past[i] = Math.max(past[i - 1], prices[i]-valley);
    }
    for (int i = prices.length-2, peak = prices[prices.length-1]; i >= 0 ; i--) {
        peak = Math.max(peak, prices[i]);
        futu[i] = Math.max(futu[i+1], peak-prices[i]);
        result = Math.max(result, past[i] + futu[i]);
    }
    return result;
}
```

2.8.13   Coins in a Line

2.8.14   Maximal Rectangle

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

Tags: Array Hash Table Stack Dynamic Programming

1. 栈: O(n) 这是一道非常综合的题目，要求在 0-1 矩阵中找出面积最大的全 1 矩阵。刚看到这道题会比较无从下手，brute force 就是对于每个矩阵都看一下，总共有 m(m+1)/2*n(n+1)/2 个子矩阵（原理跟字符串子串类似，字符串的子串数有 n(n+1)/2，只是这里是二维情形，所以是两个相乘），复杂度相当高，肯定不是面试官想要的答案，就不继续想下去了。

   这道题的解法灵感来自于 Largest Rectangle in Histogram 这道题，假设我们把矩阵沿着某一行切下来，然后把切的行作为底面，将自底面往上的矩阵看成一个直方图（histogram）。直方图的中每个项的高度就是从底面行开始往上 1 的数量。根据 Largest Rectangle in Histogram 我们就可以求出当前行作为矩阵下边缘的一个最大矩阵。接下来如果对每一行都做一次 Largest Rectangle in Histogram，从其中选出最大的矩阵，那么它就是整个矩阵中面积最大的子矩阵。算法的基本思路已经出来了，剩下的就是一些节省时间空间的问题了。

   我们如何计算某一行为底面时直方图的高度呢？如果重新计算，那么每次需要的计算数量就是当前行数乘以列数。然而在这里我们会发现一些动态规划的踪迹，如果我们知道上一行直方图

的高度，我们只需要看新加进来的行（底面）上对应的列元素是不是 0，如果是，则高度是 0，否则则是上一行直方图的高度加 1。利用历史信息，我们就可以在线行时间内完成对高度的更新。我们知道，Largest Rectangle in Histogram 的算法复杂度是 O(n)。所以完成对一行为底边的矩阵求解复杂度是 O(n+n)=O(n)。接下来对每一行都做一次，那么算法总时间复杂度是 O(m*n)。

空间上，我们只需要保存上一行直方图的高度 O(n)，加上 Largest Rectangle in Histogram 中所使用的空间 O(n)，所以总空间复杂度还是 O(n)。

```java
private int largestRectangleArea(int[] height) {
    Stack<Integer> s = new Stack<Integer>();
    int [] hnew = new int[height.length + 1];
    hnew = Arrays.copyOf(height, height.length + 1);
    int res = 0;
    int tmp;
    int left;
    for (int i = 0; i <= height.length; i++) {
        if (s.isEmpty() || hnew[i] > hnew[s.peek()]) s.push(i);
        else {
            tmp = s.pop();
            res = Math.max(res, hnew[tmp] *
                            (s.isEmpty() ? i : i - s.peek() - 1));
            i--;
        }
    }
    return res;
}

public int maximalRectangle(char[][] matrix) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;
    int [] h = new int[matrix[0].length];
    int res = 0;      // record max area
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[0].length; j++)
            h[j] = matrix[i][j] == '0' ? 0 : h[j] + 1;
        res = Math.max(res, largestRectangleArea(h));
    }
    return res;
}
```

2. 动态规划:

一般一个题目我首先会想想怎么暴力解决，比如这一题，可以枚举出所有的矩形，求出其中的面积最大者，那么怎么枚举呢，如果分别枚举矩形的宽度和高度，这样还得枚举矩形的位置，复杂度至少为 O(n^4) (计算复杂度是我们把 matrix 的行、列长度都泛化为 n，下同)，我们可以枚举矩形左上角的位置，那么知道了矩形左上角的位置，怎么计算以某一点为左上角的矩形的最大面积呢? 举例如下，下面的矩阵我们以 (0,0) 为矩形的左上角:

```
1 1 1 1 0 0
1 1 1 0 1 1
1 0 1 0 1 1
```

```
4  0 1 1 1 1 1
5  1 1 1 1 1 1
```

矩形高度是 1 时, 宽度为第一行中从第一个位置起连续的 1 的个数, 为 4, 面积为 4 * 1=4

矩形高度是 2 时, 第二行从第一个位置起连续 1 的个数是 3, 宽度为 min(3,4)=3, 面积为 3*2=6

矩形高度为 3 时, 第三行从第一个位置起连续 1 的个数是 1, 宽度为 min(1,3)=1, 面积为 1*3=3

矩形高度为 4 时, 第四行从第一个位置起连续 1 的个数是 0, 宽度为 min(0,1)=0, 面积为 0*4=0

后面的行就不用计算了, 因为上一行计算的宽度是 0, 下面所有宽度都是 0

因此以 (0,0) 为左上角的矩形的最大面积是 6, 计算以某一点为左上角的矩形的最大面积复杂度是 O(n)。

注意到上面我们用到了信息"从某一行某个位置开始连续的 1 的个数", 这个我们可以通过动态规划求得: 设 dp[i][j] 是从点 (i,j) 开始, 这一行连续 1 的个数, 动态规划方程如下:

初始条件:

- dp[i][n-1] = (matrix[i][n-1] == '1') （n 是 matrix 的列数）
- dp[i][j] = (matrix[i][j] == '1') ? 1 + dp[i][j + 1] : 0 (从方程看出我们应该从每一行的后往前计算)

计算 dp 复杂度 O(n^2), 枚举左上角位置以及计算以该位置为左上角的矩形最大面积复杂度是 O(n^2*n)=O(n^3), 总的复杂度是 O(n^3)

这个算法还可以优化, 枚举到某个点时我们可以假设该点右下方全是 1, 得到一个假设最大面积, 如果这个面积比当前计算好的面积还要小, 该点就可以直接跳过; 在上面计算以某点为左上角的矩形面积时, 也可以剪枝, 剪枝方法同上。《JustDoIT》

```java
public int maximalRectangle(char[][] matrix) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;
    int m = matrix.length;
    int n = matrix[0].length;
    int [][] dp = new int[m][n];
    int res = 0;
    int width = 0;
    for (int i = 0; i < m; i++)
        dp[i][n - 1] = matrix[i][n - 1] == '1' ? 1 : 0;
    for (int i = 0; i < m; i++)
        for (int j = n - 2; j >= 0; j--)
            if (matrix[i][j] == '1')
                dp[i][j] = dp[i][j + 1] + 1;
    for (int i = 0; i < m; i++) // 以每个点作为矩形的左上角计算所得的最大矩形面积
        for (int j = 0; j < n; j++) {
            if ((n - j) * (m - i) <= res)
                break;              // 剪枝，column-j 是最大宽度，row-i 是最大高度
            width = dp[i][j];
            for (int k = i; k < m && width > 0; k++) {
                if (width * (m - i) <= res)
                    break;     // 剪枝,row-i 是以点 (i,j) 为左上角的矩形的最大高度
                if (width > dp[k][j])
```

```
23                        width = dp[k][j]; //矩形宽度要取从第 i 行到第 k 行宽度的最小值
24                    res = Math.max(res, width * (k - i + 1));
25                }
26            }
27        return res;
28    }
```

2.8.15 Edit Distance

Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

a) Insert a character

b) Delete a character

c) Replace a character

设状态为 f[i][j], 表示 A[0, i] 和 B[0, j] 之间的最小编辑距离。设 A[0, i] 的形式是 str1c, B[0, j] 的形式是 str2d,

一. 如果 c == d, 则 f[i][j] = f[i-1][j-1];

二. 如果 c != d,

(a) 如果将 c 替换成 d, 则　　　　　f[i][j] = f[i-1][j-1] + 1;

(b) 如果在 c 后面添加一个 d, 则　f[i][j] = f[i][j-1] + 1;

(c) 如果将 c 删除, 则　　　　　　f[i][j] = f[i-1][j] + 1;

1. 动态规划: 时间复杂度 O(n*m), 空间复杂度 O(n*m)。

```
1  public int minDistance(String s, String t) {
2      int m = s.length();
3      int n = t.length();
4      int [][] res = new int[m + 1][n + 1];
5      for (int i = 0; i <= m; i++)
6          res[i][0] = i;
7      for (int i = 0; i <= n; i++)
8          res[0][i] = i;
9      for (int i = 1; i <= m; i++)
10         for (int j = 1; j <= n; j++)
11             if (s.charAt(i - 1) == t.charAt(j - 1))
12                 res[i][j] = res[i - 1][j - 1];
13             else
14                 res[i][j] = Math.min(Math.min(res[i - 1][j], res[i][j - 1]),
15                                     Math.min(res[i - 1][j],
16                                             res[i - 1][j - 1])) + 1;
17     return res[m][n];
18 }
```

2. 动态规划 + 滚动数组:

2.8.16 Distinct Subsequence

Given a string S and a string T, count the number of distinct subsequences of T in S.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

Here is an example:

S = "rabbbit", T = "rabbit"

Return 3.

设状态为 f(i, j), 表示 T[0, j] 在 S[0, i] 里出现的次数。首先, 无论 S[i] 和 T[j] 是否相等,

若不使用 S[i], 则   f(i, j) = f(i-1, j);

若 S[i] == T[j], 则可以使用 S[i], 则 f(i, j) = f(i-1, j) + f(i-1, j-1)。

```java
public int numDistinct(String s, String t) {
    if (s == null || s.length() == 0) return 0;
    int m = s.length();
    int n = t.length();
    int [][] res = new int[m + 1][n + 1];
    for (int i = 0; i <= m; i++)
        res[i][0] = 1;
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (s.charAt(i - 1) == t.charAt(j - 1))
                res[i][j] = res[i - 1][j] + res[i - 1][j - 1];
            else
                res[i][j] = res[i - 1][j];
        }
    }
    return res[m][n];
}
```

滚动数组: 时间复杂度 O(m*n), 空间复杂度 O(n)

```java
public int numDistinct(String s, String t) {
    if (s == null || s.length() == 0) return 0;
    int m = s.length();
    int n = t.length();
    int [] res = new int[n + 1];
    res[0] = 1;
    for (int i = 0; i < m; i++)
        for (int j = n - 1; j >= 0; j--)
            res[j + 1] += s.charAt(i) == t.charAt(j) ? res[j] : 0;
    return res[n];
}
```

2.8.17   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.
```

1. 思路 1: leetcode online

   忘记从线上哪个贴子看的了，根据回文可能有的两种特性，奇数长度回文和偶数长度回文，进行一维 dp 的判断。

```java
public int minCut(String s) {
    int n = s.length();
    int [] dp = new int[n + 1];
    for (int i = 0; i <= n ; i++) // initiate
        dp[i] = i - 1;
    for (int i = 0; i < n; i++) {
        // s[i - j] == s[i + j], aba
        for (int j = 0; i - j >= 0 && i + j <= n - 1 && s.charAt(i - j) == s.charAt(i +
            dp[i + j + 1] = Math.min(dp[i - j] + 1, dp[i + j + 1]);
        // s[i - j] == s[i + 1 + j], abba
        for (int j = 0; i - j >= 0 && i + 1 + j <= n - 1 && s.charAt(i - j) == s.charAt(
            dp[i + j + 2] = Math.min(dp[i - j] + 1, dp[i + j + 2]);
    }
    return dp[n];
}
```

2. 动态规划: 时间复杂度 $O(n^2)$, 空间复杂度 $O(n^2)$

   定义状态 f(i,j) 表示区间 [i,j] 之间最小的 cut 数, 则状态转移方程为

   $f(i, j) = \min \{f(i, k) + f(k + 1, j)\}$, i k j, 0 i j < n

   这是一个二维函数, 实际写代码比较麻烦。

   所以要转换成一维 DP。如果每次, 从 i 往右扫描, 每找到一个回文就算一次 DP 的话, 就可以转换为 f(i)= 区间 [i, n-1] 之间最小的 cut 数, n 为字符串长度, 则状态转移方程为

   $f(i) = \min \{f(j + 1) + 1\}$, i j < n

   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。

   定义状态 P[i][j] = true if [i, j] 为回文, 那么

   P[i][j] = str[i] == str[j] && P[i+1][j-1]

```java
public int minCut(String s) {
    int n = s.length();
    int [] f = new int[n + 1];
    boolean [][] p = new boolean[n][n];
    for (int i = 0; i <= n; i++)
        f[i] = n - 1 - i;  // 最后一个 f[n] = -1
    for (int i = n - 1; i >= 0; i--) {
        for (int j = i; j < n; j++) {
            if (s.charAt(i) == s.charAt(j) &&
                (j - i < 2 || p[i + 1][j - 1] == true)) {
                p[i][j] = true;
                f[i] = Math.min(f[i], f[j + 1] + 1);
            }
        }
    }
```

```
15        }
16        return f[0];
17    }
```

2.8.18   Interleaving String

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.

  For example, Given:

```
1  s1 = "aabcc",
2  s2 = "dbbca",
3  When s3 = "aadbbcbcac", return true.
4  When s3 = "aadbbbaccc", return false.
```

1. 递归: 会超时, 仅用来帮助理解

   今天写不出来代码，明天再写～

2. 动态规划: 二维动规, 时间复杂度 $O(n^2)$, 空间复杂度 $O(n^2)$

   设状态 f[i][j], 表示 s1[0, i] 和 s2[0, j], 匹配 s3[0, i+j]。如果 s1 的最后一个字符等于 s3 的最后一个字符, 则 f[i][j]=f[i-1][j]; 如果 s2 的最后一个字符等于 s3 的最后一个字符, 则 f[i][j]=f[i][j-1]。

   因此状态转移方程如下:

```
1  f[i][j] = (s1[i - 1] == s3 [i + j - 1] && f[i - 1][j])
2          || (s2[j - 1] == s3 [i + j - 1] && f[i][j - 1]);
```

```
1  public boolean isInterleave(String s1, String s2, String s3) {
2      int m = s1.length();
3      int n = s2.length();
4      if (s3.length() != m + n) return false;
5      boolean [][] f = new boolean[m + 1][n + 1];
6      f[0][0] = true;
7      for (int j = 1; j <= m ; j++)
8          f[j][0] = f[j - 1][0] && s1.charAt(j - 1) == s3.charAt(j - 1);
9      for (int j = 1; j <= n ; j++)
10         f[0][j] = f[0][j - 1] && s2.charAt(j - 1) == s3.charAt(j - 1);
11     for (int i = 1; i <= m; i++) {
12         for (int j = 1; j <= n ; j++) {
13             f[i][j] = ((f[i - 1][j] && s1.charAt(i - 1) == s3.charAt(i + j - 1)) ||
14                     (f[i][j - 1] && (s2.charAt(j - 1) == s3.charAt(i + j - 1))));
15         }
16     }
17     return f[m][n];
18 }
```

3. 二维动规 + 滚动数组: 时间复杂度 $O(n^2)$, 空间复杂度 $O(n)$

```
1  public boolean isInterleave(String s1, String s2, String s3) {
2      int m = s1.length();
3      int n = s2.length();
```

```
4        if (s3.length() != m + n) return false;
5        if (m < n) return isInterleave(s2, s1, s3);
6        boolean [] f = new boolean[n + 1];
7        for (int i = 0; i < n + 1; i++)
8            f[i] = true;
9        for (int i = 1; i < n + 1 ; i++)
10           f[i] = s2.charAt(i - 1) == s3.charAt(i - 1) && f[i - 1];
11       for (int i = 1; i < m + 1; i++) {
12           f[0] = s1.charAt(i - 1) == s3.charAt(i - 1) && f[0];
13           for (int j = 1; j < n + 1; j++)
14               f[j] = (s1.charAt(i - 1) == s3.charAt(i + j - 1) && f[j])
15                   || (s2.charAt(j - 1) == s3.charAt(i + j - 1) && f[j - 1]);
16       }
17       return f[n];
18   }
```

2.8.19 Scramble String

Given a string s1, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of s1 = "great":

```
1        great
2        /      \
3     gr      eat
4    /  \    /    \
5  g    r   e     at
6                 /  \
7                a    t
```
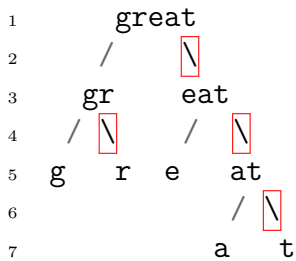
To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".

```
1        rgeat
2        /      \
3     rg      eat
4    /  \    /    \
5  r    g   e     at
6                 /  \
7                a    t
```
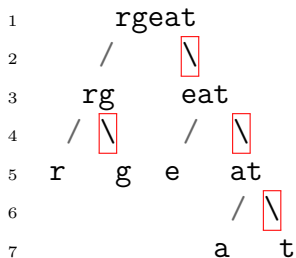
We say that "rgeat" is a scrambled string of "great".

Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string "rgtae".

```
1        rgtae
2        /      \
3     rg      tae
```
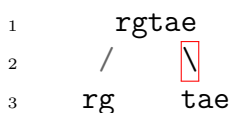
```
4      / \      / \
5    r   g   ta   e
6            / \
7          t   a
```

We say that "rgtae" is a scrambled string of "great".

Given two strings s1 and s2 of the same length, determine if s2 is a scrambled string of s1.

首先想到的是递归 (即深搜), 对两个 string 进行分割, 然后比较四对字符串。代码虽然简单, 但是复杂度比较高。有两种加速策略, 一种是剪枝, 提前返回; 一种是加缓存, 缓存中间结果, 即 memorization(翻译为记忆化搜索)。

这两种思路应该都非常容易理解, 有大量工作要做, 遇到已经干过的 quit 不干了, 剪枝; 或者保存了前面工作的结果, 遇见干过的, 直接调结果, 缓存?

剪枝可以五花八门, 要充分观察, 充分利用信息, 找到能让节点提前返回的条件。例如, 判断两个字符串是否互为 scamble, 至少要求每个字符在两个字符串中出现的次数要相等, 如果不相等则返回 false。

加缓存, 可以用数组或 HashMap。本题维数较高, 用 HashMap, map 和 unordered$_{map}$ 均可。既然可以用记忆化搜索, 这题也一定可以用动规。设状态为 f[n][i][j], 表示长度为 n, 起点为 s1[i] 和起点为 s2[j] 两个字符串是否互为 scramble, 则状态转移方程为

```
1   f[n][i][j] = (f[k][i][j] && f[n-k][i+k][j+k])
2              || (f[k][i][j+n-k] && f[n-k][i+k][j])
```

1. 递归: 时间复杂度 $O(n^6)$, 空间复杂度 $O(1)$

2. 动态规划: 时间复杂度 $O(n^3)$, 空间复杂度 $O(n^3)$

   代码的变量名称循环方式还是看起来很奇怪, 改天再优化一下 ~

```java
1   public boolean isScramble(String s1, String s2) {
2       int n = s1.length();
3       // f[k][i][j]，表示长度各为 k, 起点为 s1[i] 和
4       // 起点为 s2[j] 两个字符串是否互为 scramble
5       boolean [][][] f = new boolean[n + 1][n][n];
6       for (int i = 0; i < n; i++)
7           for (int j = 0; j < n; j++)
8               f[1][i][j] = s1.charAt(i) == s2.charAt(j);
9       for (int k = 2; k <= n; k++) {
10          for (int i = 0; i <= n - k; i++) {
11              for (int j = 0; j <= n - k; j++) {
12                  f[k][i][j] = false;
13                  for (int x = 1; x < k && !f[k][i][j]; x++) {
14                      f[k][i][j] = ( (f[x][i][j] && f[k - x][x + i][x + j])
15                                  || (f[k - x] [x + i][j] && f[x][i][k - x + j]));
16                  }
17              }
18          }
19      }
20      return f[n][0][0];
21  }
```

3. 递归 + 剪枝: 时间复杂度 $O(n^6)$, 空间复杂度 $O(1)$

4. 备忘录法: 递归 +map 做 cache, 时间复杂度 $O(n^3)$, 空间复杂度 $O(n^3)$

5. 备忘录法:

   定制一个哈希函数, 递归 + (c++) unordered$_{map}$ 做 cache, 比 map 快, 时间复杂度 $O(n^3)$, 空间复杂度 $O(n^3)$.

   Java 是一样的吗，改天试一下。。。这三个题 need every effort to work on them....

   http://www.cnblogs.com/TenosDoIt/p/3452004.html

   http://blog.unieagle.net/2012/10/23/leetcode%E9%A2%98%E7%9B%AE%EF%BC%9Ascramble-string%EF%BC%8C%E4%B8%89%E7%BB%B4%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92/

2.8.20 Regular Expression Matching

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

```
1  '.' Matches any single character.
2  '*' Matches zero or more of the preceding element.
```

The matching should cover the entire input string (not partial).
The function prototype should be:
bool isMatch(const char *s, const char *p)
Some examples:

```
1  isMatch("aa","a")   → false
2  isMatch("aa","aa")  → true
3  isMatch("aaa","aa") → false
4  isMatch("aa", "a*") → true
5  isMatch("aa", ".*") → true
6  isMatch("ab", ".*") → true
7  isMatch("aab", "c*a*b") → true
```

Tags: Dynamic Programming Backtracking String

这是一道很有挑战的题（三个最难的之一，本楼层居中，还有其上其下 neighbor 楼。）。这是 LeetCode 里仅有的两个我觉得面试时应该用 c++ 写的题目，尤其是递归法，假如面试时实在想不出 DP 思路的话。所以这里代码直接贴 c++ 版的。。。

1. 递归: 时间复杂度 $O(n)$, 空间复杂度 $O(1)$

   来自 http://leetcode.com/2011/09/regular-expression-matching.html

```
1  bool isMatch(const char *s, const char *p) {
2      if (*p == '\0') return *s == '\0';
3      if (*(p + 1) != '*') {
4          if (*p == *s || (*p == '.' && *s != '\0'))
5              return isMatch(s + 1, p + 1);
6          else return false;
7      } else { // next char is '*'
8          while (*p == *s || (*p == '.' && *s != '\0')) {
```

```
9              if (isMatch(s, p + 2)) // can optimize
10                 return true;
11             s++;                    // but easier to understand
12         }
13     }
14     return isMatch(s, p + 2);
15 }
```

搜到两种 Java 的解法，改天自己得再好好研究下。。。来自《压栈底的弱渣》

暂时只想出来了递归解法。其实就是自动机的实现原理。

对于两个字符串：string，pattern 有两种情况：

当 pattern[1] 不为"*"的时候（或 pattern 长度为 1），直接比较 patter[0] 和 string[0] 是否相等。若不等，则直接返回 false；若相等，则接着比较他们后面的字符，即跳到自动机的下一个 state。

当 pattern[1] 为"*"的时候，情况就稍微复杂一点了。

若 pattern[0] != string[0]，或 string 长度为 0 时，应跳过 pattern[0] 和 pattern[1] 继续比较后面的字符，因为"*"为 optional symbol。

若 pattern[0] == string[0]，我们既可以跳过 string[0]，也可以跳过 pattern[0] 和 pattern[1]。

```java
1  public boolean isMatch(String s, String p) {
2      if (p.length() == 0) return s.length() == 0;
3      if (p.length() == 1 || p.charAt(1)!='*') {
4          if (s.length() > 0 &&
5              (p.charAt(0) == '.' || s.charAt(0) == p.charAt(0)))
6              return isMatch(s.substring(1), p.substring(1));
7          else return false;
8      } else {
9          if (s.length() == 0) return isMatch(s, p.substring(2));
10         if (p.charAt(0) == '.' || s.charAt(0) == p.charAt(0))
11             return isMatch(s.substring(1), p) || isMatch(s, p.substring(2));
12         else return isMatch(s, p.substring(2));
13     }
14 }
```

在 Java 1.7 以后，String.substring() 这个方法会重新拷贝一个新的字符串，这样在递归过程中内存效率很低，而且会消耗大量时间用来对象初始化。因而可以考虑在递归的时候传 index 进去。

改进代码如下：

```java
1  public boolean isMatchHelper(String s, String p, int is, int ip){
2      if (ip == p.length()) return is  ==  s.length();
3      if (ip == p.length()-1 || p.charAt(ip + 1)!='*'){ // no *
4          if ( is < s.length() &&
5              (p.charAt(ip) == '.' || s.charAt(is) == p.charAt(ip)) )
6              return isMatchHelper(s, p, is + 1, ip + 1);
7          else return false;
```

```
8           } else {                                          // with *
9               if (is == s.length())
10                  return isMatchHelper(s,p,is,ip + 2);       // empty
11              if (p.charAt(ip) == '.' || s.charAt(is) == p.charAt(ip)) //match
12                  return isMatchHelper(s, p, is + 1, ip)
13                      || isMatchHelper(s, p, is, ip + 2);  // two choices for continue
14              else return isMatchHelper(s, p, is, ip + 2); // not match
15          }
16      }
17
18      public boolean isMatch(String s, String p) {
19          return isMatchHelper(s, p, 0, 0);
20      }
```

2. 动态规划:

仅次于 Scramble String 的最复杂的动规。。。

思路改天再写，...

```
1   public boolean isMatch(String s, String p) {
2       if (p.length() == 0 && s.length() == 0) return true;
3       if (p.length() == 0 && s.length() > 0) return false;
4       int m = s.length();
5       int n = p.length();
6       boolean [][] dp = new boolean[m + 1][n + 1];
7       dp[0][0] = true;
8       for (int i = 1; i <= m ; i++)
9           dp[i][0] = false;
10      for (int i = 1; i <= n ; i++)
11          if (i >= 2)
12              dp[0][i] = (p.charAt(i - 1) == '*') ? dp[0][i - 2] : false;
13      for (int i = 1; i <= m ; i++) {
14          for (int j = 1; j <= n ; j++) {
15              if (p.charAt(j - 1) == '.')
16                  dp[i][j] = dp[i - 1][j - 1];
17              else if (p.charAt(j - 1) == '*')
18                  dp[i][j] = ( dp[i][j - 1] || dp[i][j - 2] ||
19                              (dp[i - 1][j] &&
20                              ((s.charAt(i - 1) == p.charAt(j - 2)  || p.charAt(j - 2) =
21              else if (s.charAt(i - 1) == p.charAt(j - 1))
22                  dp[i][j] = dp[i - 1][j - 1];
23          }
24      }
25      return dp[m][n];
26  }
```

2.8.21 Wildcard Matching

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

```
1  '?' Matches any single character.
2  '*' Matches any sequence of characters (including the empty sequence).
```

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

The function prototype should be:

bool isMatch(const char *s, const char *p)

Some examples:

```
1  isMatch("aa","a")  →  false
2  isMatch("aa","aa")  →  true
3  isMatch("aaa","aa")  →  false
4  isMatch("aa", "*")  →  true
5  isMatch("aa", "a*")  →  true
6  isMatch("ab", "?*")  →  true
7  isMatch("aab", "c*a*b")  →  false
```

Tags: Dynamic Programming, Backtracking, Greedy, String

跟上一题很类似。

主要是'*' 的匹配问题。p 每遇到一个'*', 就保留住当前'*' 的坐标和 s 的坐标, 然后 s 从前往后扫描, 如果不成功, 则 s++, 重新扫描。

除 Scramble String 外最复杂的 DP, 因为'*' 可以什么也不用配。

1. 递归版: 时间复杂度 O(n!*m!), 空间复杂度 O(n)

   递归版, 会超时, 用于帮助理解题意.

```
1  bool isMatch(const char *s, const char *p) {
2      if (*p == '*') {
3          while (*p == '*') ++p;
4          if (*p == '\0') return true;
5          while (*s != '\0' && !isMatch(s, p)) ++s;
6          return *s != '\0';
7      } else if (*p == '\0' || *s == '\0') return *p == *s;
8      else if (*p == *s || *p == '?') return isMatch(++s, ++p);
9      else return false;
10 }
```

2. 迭代版: 时间复杂度 O(n*m), 空间复杂度 O(1)

3. 动态规划:

   直接使用递归会超出内存或者超时，因为递归会重复计算很多相同的内容，可以使用动态规划利用中间结果来避免重复计算。

   用 dp[i][j] 表示 s 串的前 i 个字符和 p 串的前 j 个字符是否匹配，则计算

   p[i][j] 时，可以根据 p 的第 j 个字符分别讨论

   p[j-1] 为'*', s 可以选择使用这个 *, 转化为 dp[i-1][j]; 或者不使用这个'*', 变为 dp[i][j-1]。

   p[j-1] 为'?', 此时 p[j-1] 必定要匹配 s[i-1]。

   p[j-1] 为普通字符, 如果 p[j-1]==s[i-1], 匹配成功, 问题转化为 dp[i-1][j-1], 否则匹配失败, dp[i][j] = false。

```
1  if (p[j-1] == '*') then dp[i][j] = dp[i-1][j] || dp[i][j-1]
2  if (p[j-1] == '?') then dp[i][j] = dp[i-1][j-1]
3  if (p[j-1] == s[i-1])   dp[i][j] = dp[i-1][j-1] else dp[i][j]=false//普通字符
```

当 i 或 j 为 0 时作为初始化部分，根据定义，应有:

dp[0][0] = true。

dp[i][0] = false，因为有长度的 s 肯定无法和空串 p 匹配。

dp[0][i]，只有 p[i-1] 为 * 且 dp[0][i-1] 为真是才为真。

```
1  public boolean isMatch(String s, String p) {
2      int minLen = 0;
3      for (int i = 0; i < p.length(); i++)
4          if (p.charAt(i) != '*') ++minLen;
5      if (s.length() < minLen) return false;
6      int m = s.length();
7      int n = p.length();
8      boolean [][] dp = new boolean[m + 1][n + 1];
9      dp[0][0] = true;
10     for (int i = 1; i <= m ; i++)
11         dp[i][0] = false;
12     for (int i = 1; i <= n ; i++)
13         dp[0][i] = (p.charAt(i - 1) == '*' && dp[0][i - 1]) ? true : false;
14     for (int i = 1; i <= m; i++)
15         for (int j = 1; j <= n ; j++) {
16             if (p.charAt(j - 1) == '*')  // user * or not
17                 dp[i][j] = (dp[i - 1][j] || dp[i][j - 1]);
18             else if (s.charAt(i - 1) == p.charAt(j - 1) || p.charAt(j - 1) == '?')
19                 dp[i][j] = dp[i - 1][j - 1];
20         }
21     return dp[m][n];
22  }
```

2.8.22   others Covered

- Best Time to Buy and Sell Stock 2.7.2

- Best Time to Buy and Sell Stock II 2.7.3

- Longest Valid Parentheses 1.5.4

- Word Break II 2.4.3

- Maximum Rectangle `Maximum Rectangle`
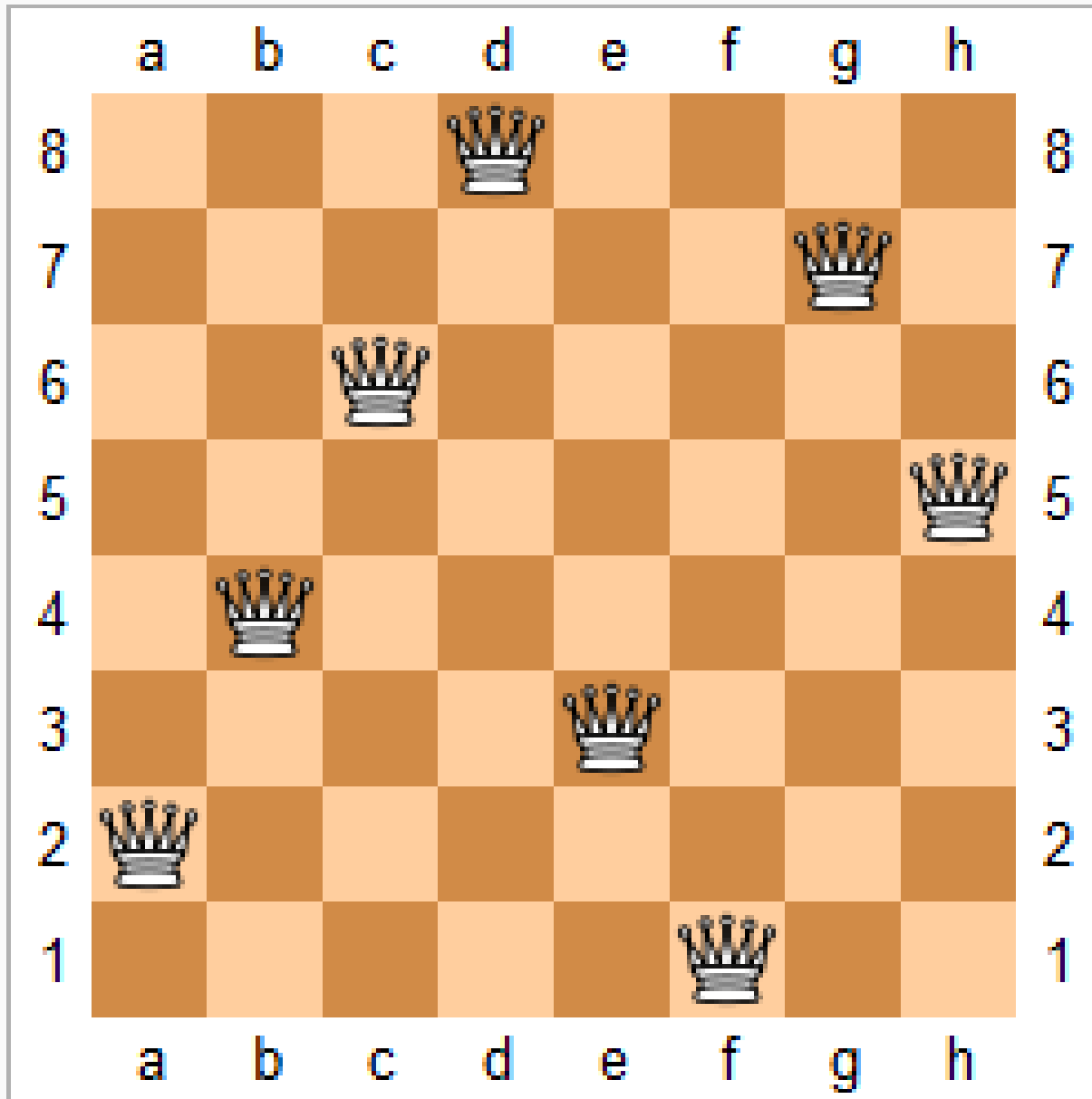
One solution to the eight queens puzzle

Figure 2.3: N-Queens

Figure 2.4: Sudoku Solver 1

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 2.5: Sudoku Solver 2
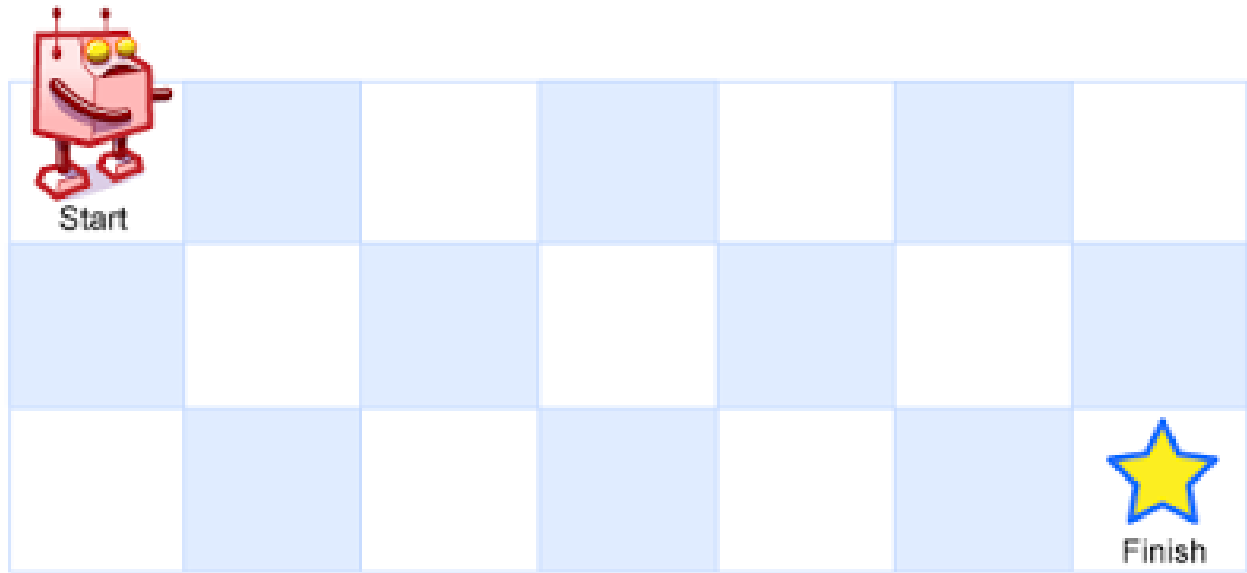
Figure 2.6: Unique Path
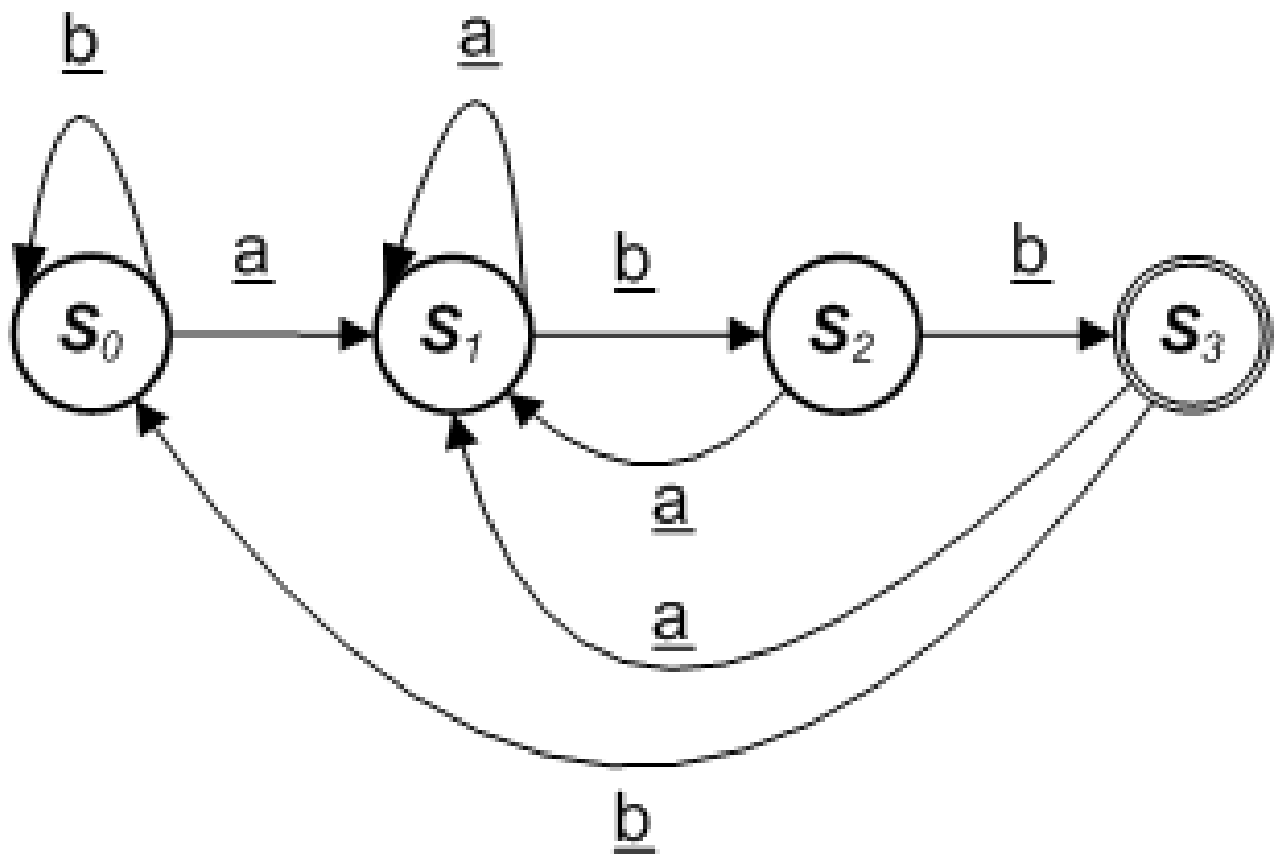
Figure 2.7: Regular Expression Matching

Chapter 3

LeetCode Summary - Part III

3.1 Two Pointers and Sliding Window

3.1.1 Valid Palindrome

3.1.2 Remove Nth Node From End of List

3.1.3 Remove Element

3.1.4 Remove Duplicates from Sorted Array

3.1.5 Merge Sorted Array

3.1.6 Implement strStr()

3.1.7 3Sum

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.

Note:

Elements in a triplet (a,b,c) must be in non-descending order. (ie, a   b   c)

The solution set must not contain duplicate triplets.

For example, given array S = {-1 0 1 2 -1 -4},

A solution set is:

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

先排序, 然后二分查找, 复杂度 O(n 2 log n)。

```
1  public List<List<Integer>> threeSum0(int[] num) {
2      ArrayList<List<Integer>> result = new ArrayList<List<Integer>>();
3      Arrays.sort(num);
4      List<Integer> one = new ArrayList<Integer>();
5      Map<List<Integer>, Integer> hash = new HashMap<List<Integer>, Integer>();
6      int sum, j, k;
7      for (int i = 0; i < num.length-2; i++) {
8          if (i > 0 && num[i] == num[i-1]) continue;
```

```
9           j = i + 1;
10          k = num.length - 1;
11          while (j < k) {
12              while (k > num.length - 1 && num[k] == num[k+1]) k--;
13              sum = num[i] + num[j] + num[k];
14              if (sum > 0) k--;
15              else if (sum < 0) j++;
16              else {
17                  one.add(num[i]);
18                  one.add(num[j]);
19                  one.add(num[k]);
20                  if (!hash.containsKey(one))
21                      hash.put(one, 1);
22                  one = new ArrayList<Integer>();
23                  j++;
24                  k--;
25              }
26          }
27      }
28      for(List<Integer> key : hash.keySet())
29          result.add(key);
30      return result;
31 }
```

3.1.8   3Sum Closest

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

For example, given array S = {-1 2 1 -4}, and target = 1.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

先排序, 然后左右夹逼, 复杂度 O(n 2)。

1. 代码 1:

```
1  // LeetCode, 3Sum Closest
2  // 先排序，然后左右夹逼，时间复杂度 O(n^2)，空间复杂度 O(1)
3  class Solution {
4  public:
5      int threeSumClosest(vector<int>& num, int target) {
6          int result = 0;
7          int min_gap = INT_MAX;
8          sort(num.begin(), num.end());
9          for (auto a = num.begin(); a != prev(num.end(), 2);
10              a = upper_bound(a, prev(num.end(), 2), *a)) {
11              auto b = next(a);
12              auto c = prev(num.end());
13              while (b < c) {
```

```
14              const int sum = *a + *b + *c;
15              const int gap = abs(sum - target);
16              if (gap < min_gap) {
17                  result = sum;
18                  min_gap = gap;
19              }
20              if (sum < target) ++b;
21              else
22                  --c;
23          }
24      }
25      return result;
26  }
27 };
```

2. 代码 2:

```
1  // LeetCode, 3Sum Closest
2  // 先排序，然后左右夹逼
3  // 时间复杂度 O(n^2)，空间复杂度 O(1)，与代码 1 比，有常数级的优化
4  class Solution {
5  public:
6      int threeSumClosest(vector<int>& num, int target) {
7          int result = 0;
8          int min_gap = INT_MAX;
9          sort(num.begin(), num.end());
10         for (auto a = num.begin(); a != prev(num.end(), 2);
11              a = upper_bound(a, prev(num.end(), 2), *a)) {
12             auto b = next(a);
13             auto c = prev(num.end());
14             while (b < c) {
15                 const int sum = *a + *b + *c;
16                 const int gap = abs(sum - target);
17                 if (gap < min_gap) {
18                     result = sum;
19                     min_gap = gap;
20                 }
21                 if (sum < target)
22                     b = upper_bound(b, c, *b);
23                 else
24                     c = prev(lower_bound(b, c, *c));
25             }
26         }
27         return result;
28     }
29 };
```

from original......

3.1.9 4Sum

Given an array S of n integers, are there elements a, b, c, and d in S such that a + b + c + d = target?
Find all unique quadruplets in the array which gives the sum of target.

Note: Elements in a quadruplet (a,b,c,d) must be in non-descending order. (ie, a   b   c   d) The
solution set must not contain duplicate quadruplets. For example, given array S = {1 0 -1 0 -2 2}, and
target = 0.

A solution set is:

```
1    (-1,  0, 0, 1)
2    (-2, -1, 1, 2)
3    (-2,  0, 0, 2)
```

Tags: Array, Hash Table, Two Pointers
跟 3Sum 很类似, 先排序, 然后二分查找, 复杂度 O(n 3 log n), 会超时。
然后想到, 可以先缓存两个数的和, 最终复杂度 O(n 3)。这个策略也适用于 3Sum 。

1. 二分查找: 先排序, 然后二分查找, 复杂度 O(n^3*logn), 会超时

```cpp
1  vector<vector<int>> fourSum(vector<int>& num, int target) {
2      vector<vector<int>> result;
3      if (num.size() < 4) return result;
4      sort(num.begin(), num.end());
5      auto last = num.end();
6      for (auto a = num.begin(); a < prev(last, 3);
7           a = upper_bound(a, prev(last, 3), *a)) {
8          for (auto b = next(a); b < prev(last, 2);
9               b = upper_bound(b, prev(last, 2), *b)) {
10             for (auto c = next(b); c < prev(last);
11                  c = upper_bound(c, prev(last), *c)) {
12                 const int d = target - *a - *b - *c;
13                 if (binary_search(next(c), last, d))
14                     result.push_back(vector<int> { *a, *b, *c, d });
15             }
16         }
17     }
18     return result;
19 }
```

2. map 做缓存: 先缓存两个数的和, 时间复杂度 O(n^3), 空间复杂度 O(n^2)

```cpp
1  vector<vector<int> > fourSum(vector<int> &num, int target) {
2      if (num.size() < 4) return vector<vector<int> >();
3      sort(num.begin(), num.end());
4      map<int, vector<pair<int, int> > > cache;
5      for (size_t a = 0; a < num.size(); ++a) {
6          for (size_t b = a + 1; b < num.size(); ++b) {
7              cache[num[a] + num[b]].push_back(pair<int, int>(a, b));
8          }
9      }
```

```
10        set<vector<int>> result; // 去重，因为 num 里有重复元素
11        for (size_t c = 2; c < num.size(); ++c) {
12            for (size_t d = c + 1; d < num.size(); ++d) {
13                const int key = target - num[c] - num[d];
14                if (cache.find(key) != cache.end()) {
15                    for (size_t k = 0; k < cache[key].size(); ++k) {
16                        if (c <= cache[key][k].second) continue; // 有重叠
17                        result.insert(vector<int> { num[cache[key][k].first],
18                                num[cache[key][k].second], num[c], num[d] });
19                    }
20                }
21            }
22        }
23        return vector<vector<int> >(result.begin(), result.end());
24    }
```

3.1.10 Container With Most Water

3.1.11 Remove Duplicates from Sorted Array II

3.1.12 Partition List

3.1.13 Two Sum II - Input array is sorted

3.1.14 Linked List Cycle II

3.1.15 Longest Substring Without Repeating Characters

3.1.16 Linked List Cycle

3.1.17 Sort Colors

3.1.18 Rotate List

3.1.19 Trapping Rain Water

3.1.20 Longest Substring with At Most Two Distinct Characters

3.1.21 Substring with Concatenation of All Words

3.1.22 Minimum Window Substring

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

For example,

```
1  S = "ADOBECODEBANC"
2  T = "ABC"
3  Minimum window is "BANC".
```

Note:

If there is no such window in S that covers all characters in T, return the emtpy string "".

If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

Tags: Hash Table Two Pointers String

双指针, 动态维护一个区间。尾指针不断往后扫, 当扫到有一个窗口包含了所有 T 的字符后, 然后再收缩头指针, 直到不能再收缩为止。最后记录所有可能的情况中窗口最小的

可以利用两个指针扫描（两个指针分别为 start，i），以

```
1  S = "e b a d b a c c b"(忽略空格),  T = "abc" 为例:
2      0 1 2 3 4 5 6 7 8
```

1. 初始化 start = i = 0

2. i 逐渐往后扫描 S 直到窗口 S[start···i] 包含所有 T 的字符，此时 i = 6（字符 c 的位置）

3. 缩减窗口：此时我们注意到窗口并不是最小的，需要调整 start 来缩减窗口。缩减规则为：如果 S[start] 不在 T 中或者 S[start] 在 T 中但是删除后窗口依然可以包含 T 中的所有字符，那么 start = start+1，直到不满足上述两个缩减规则。缩减后 i 即为窗口的起始位置，此例中从 e 开始窗口中要依次删掉 e、b、a、d, start 最后等于 4 ，那么得到一个窗口大小为 6-4+1 = 3

4. start = start+1(此时窗口肯定不会包含所有的 T 中的字符)，跳转到步骤 2 继续寻找下一个窗口。本例中还以找到一个窗口 start = 5，i = 8，比上个窗口大，因此最终的最小窗口是 S[4···6]

具体实现时，要用哈希表来映射 T 中字符以便在 O(1) 时间内判断一个字符是否在 T 中，由于是字符缘故，可以用数组简单的来实现；还需要一个哈希表来记录扫描时 T 中的某个字符在 S 中出现的次数，也可以用数组实现

1. 时间复杂度 O(n), 空间复杂度 O(1)

c++ 版:

```cpp
1  class Solution {
2  public:
3      string minWindow(string S, string T) {
4          if (S.empty()) return "";
5          if (S.size() < T.size()) return "";
6          const int ASCII_MAX = 256;
7          int appeared_count[ASCII_MAX];
8          int expected_count[ASCII_MAX];
9          fill(appeared_count, appeared_count + ASCII_MAX, 0);
10         fill(expected_count, expected_count + ASCII_MAX, 0);
11         for (size_t i = 0; i < T.size(); i++) expected_count[T[i]]++;
12         int minWidth = INT_MAX, min_start = 0; // 窗口大小，起点
13         int wnd_start = 0;
14         int appeared = 0; // 完整包含了一个 T
15         //尾指针不断往后扫
16         for (size_t wnd_end = 0; wnd_end < S.size(); wnd_end++) {
17             if (expected_count[S[wnd_end]] > 0) { // this char is a part of T
18                 appeared_count[S[wnd_end]]++;
19                 if (appeared_count[S[wnd_end]] <= expected_count[S[wnd_end]])
20                     appeared++;
```

```cpp
21                  }
22                  if (appeared == T.size()) { // 完整包含了一个 T
23                      // 收缩头指针
24                      while (appeared_count[S[wnd_start]] > expected_count[S[wnd_start]]
25                              || expected_count[S[wnd_start]] == 0) {
26                          appeared_count[S[wnd_start]]--;
27                          wnd_start++;
28                      }
29                      if (minWidth > (wnd_end - wnd_start + 1)) {
30                          minWidth = wnd_end - wnd_start + 1;
31                          min_start = wnd_start;
32                      }
33                  }
34              }
35          if (minWidth == INT_MAX) return "";
36          else return S.substr(min_start, minWidth);
37      }
38  };
```

Java 版: need to optimize my code....

```java
1   public String minWindow(String s, String t) {
2       Map<Character, Integer> hash = new HashMap<Character, Integer>();
3       for (int i = 0; i < t.length(); i++)
4           if (!hash.containsKey(t.charAt(i)))
5               hash.put(t.charAt(i), 1);
6           else hash.put(t.charAt(i), hash.get(t.charAt(i)) + 1);
7       Map<Character, Integer> fnd = new HashMap<Character, Integer>();
8       int fndCnt = 0;
9       int slow = 0, fast = slow, size, minl = Integer.MAX_VALUE;
10      StringBuffer res = new StringBuffer("");
11      char j;
12      while (fast < s.length()) {
13          j = s.charAt(fast);
14          if (hash.containsKey(j)) {
15              if (!fnd.containsKey(j)) {
16                  fnd.put(j, 1);
17                  fndCnt++;
18              } else {
19                  if (fnd.get(j) < hash.get(j))
20                      fndCnt++;
21                  fnd.put(j, fnd.get(j) + 1);
22              }
23          }
24          if (fndCnt == t.length()) {
25              char sc = s.charAt(slow);
26              while (!fnd.containsKey(sc) || fnd.get(sc) > hash.get(sc)) {
27                  if (fnd.containsKey(sc) && fnd.get(sc) > hash.get(sc))
28                      fnd.put(sc, fnd.get(sc) - 1);
```

```
29                    slow++;
30                    sc = s.charAt(slow);
31                }
32                if (fast - slow + 1 < minl) {
33                    minl = fast - slow + 1;
34                    res = new StringBuffer(s.substring(slow, fast + 1));
35                }
36            }
37            fast++;
38        }
39        return res.toString();
40    }
```

3.2 Backtracing and Recursion

3.2.1 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):

```
1   "123"
2   "132"
3   "213"
4   "231"
5   "312"
6   "321"
```

Given n and k, return the kth permutation sequence.

Note: Given n will be between 1 and 9 inclusive.

3.2.2 Gray Code

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

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

For example, given n = 2, return [0,1,3,2]. Its gray code sequence is:

```
1   00 - 0
2   01 - 1
3   11 - 3
4   10 - 2
```

Note:

- For a given n, a gray code sequence is not uniquely defined.

- For example, [0,2,3,1] is also a valid gray code sequence according to the above definition.

- For now, the judge is able to judge based on one instance of gray code sequence. Sorry about that.

格雷码 (Gray Code) 的定义请参考 wikipedia http://en.wikipedia.org/wiki/Gray_code。

自然二进制码转换为格雷码:$g_0 = b_0, g_i = b_i \oplus b_{i-1}$

保留自然二进制码的最高位作为格雷码的最高位, 格雷码次高位为二进制码的高位与次高位异或, 其余各位与次高位的求法类似。例如, 将自然二进制码 1001, 转换为格雷码的过程是: 保留最高位; 然后将第 1 位的 1 和第 2 位的 0 异或, 得到 1, 作为格雷码的第 2 位; 将第 2 位的 0 和第 3 位的 0 异或, 得到 0, 作为格雷码的第 3 位; 将第 3 位的 0 和第 4 位的 1 异或, 得到 1, 作为格雷码的第 4 位, 最终, 格雷码为 1101。

格雷码转换为自然二进制码:$b_0 = g_0, b_i = g_i \oplus b_{i-1}$

保留格雷码的最高位作为自然二进制码的最高位, 次高位为自然二进制高位与格雷码次高位异或, 其余各位与次高位的求法类似。例如, 将格雷码 1000 转换为自然二进制码的过程是: 保留最高位 1, 作为自然二进制码的最高位; 然后将自然二进制码的第 1 位 1 和格雷码的第 2 位 0 异或, 得到 1, 作为自然二进制码的第 2 位; 将自然二进制码的第 2 位 1 和格雷码的第 3 位 0 异或, 得到 1, 作为自然二进制码的第 3 位; 将自然二进制码的第 3 位 1 和格雷码的第 4 位 0 异或, 得到 1, 作为自然二进制码的第 4 位, 最终, 自然二进制码为 1111。

格雷码有数学公式, 整数 n 的格雷码是 $n \oplus (n/2)$。

这题要求生成 n 比特的所有格雷码。

最简单的方法, 利用数学公式, 对从 0 $2^n - 1$ 的所有整数, 转化为格雷码。

n 比特的格雷码, 可以递归地从 $n - 1$ 比特的格雷码生成。如图 §2-5 所示。
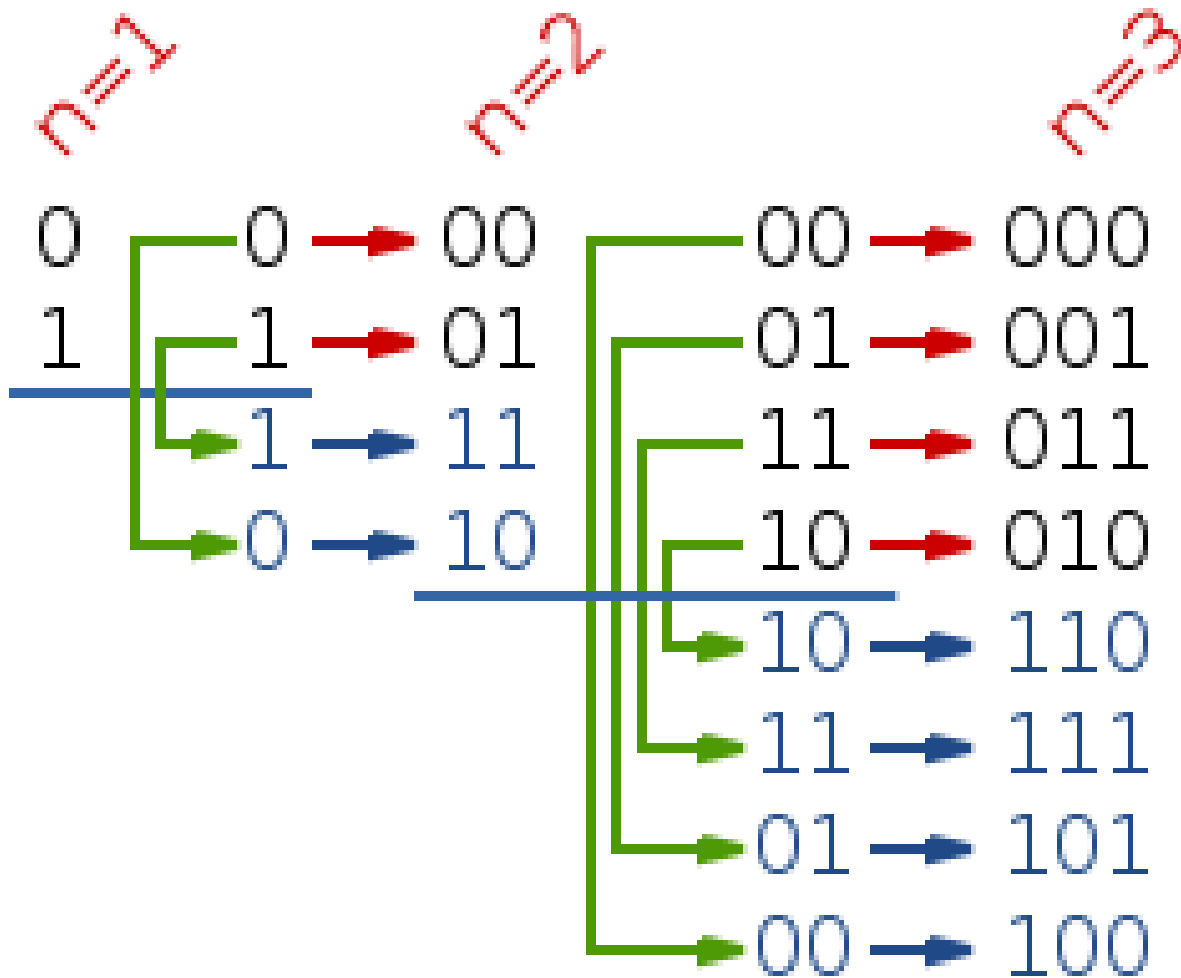
Figure 3.1: Gray Code
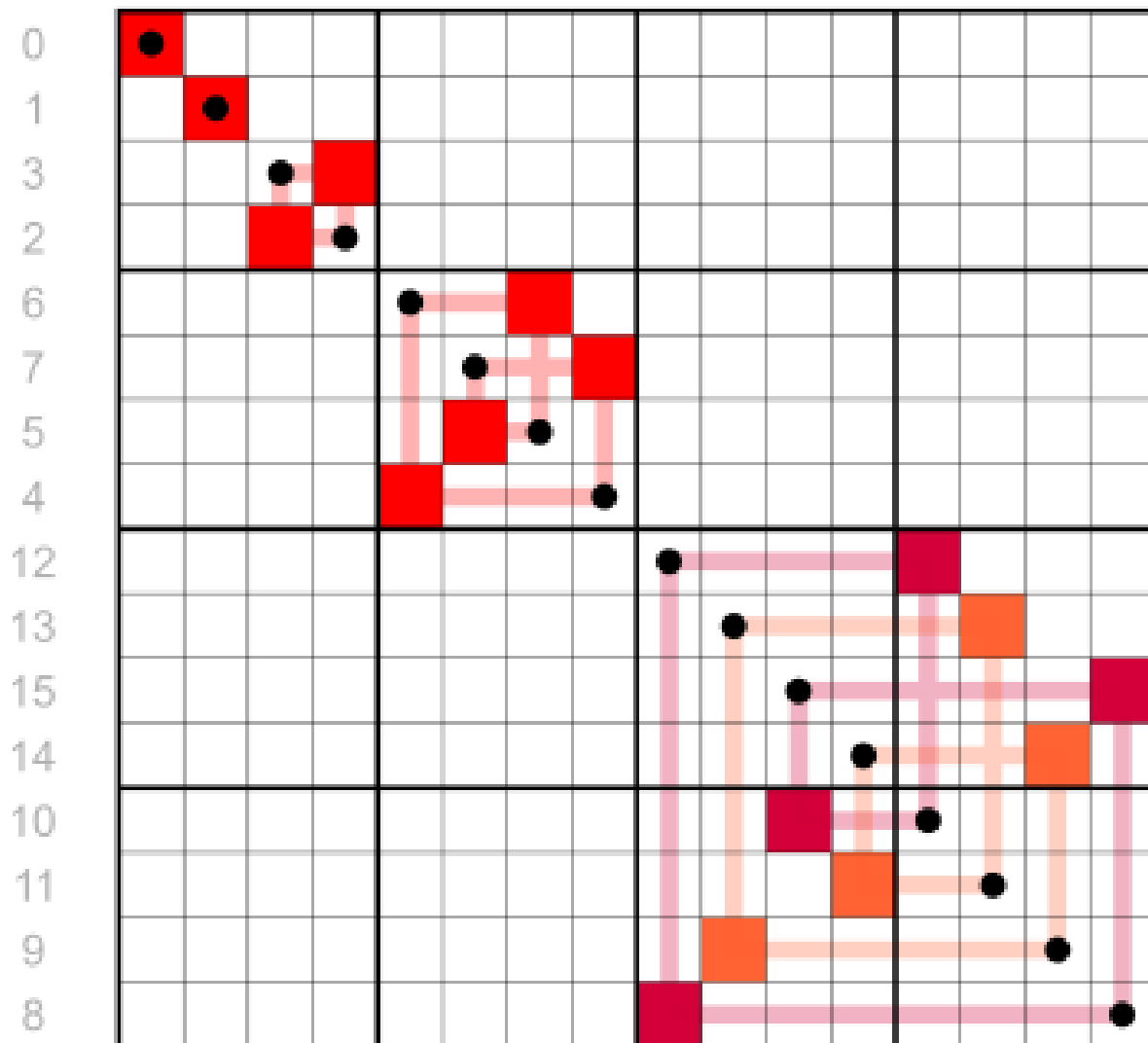
Figure 3.2: Gray Code 2

3.2.3 Word Break II

Given a string s and a dictionary of words dict, add spaces in s to construct a sentence where each word is a valid dictionary word.

Return all such possible sentences.

For example, given

```
1  s = "catsanddog",
2  dict = ["cat", "cats", "and", "sand", "dog"].
3
4  A solution is ["cats and dog", "cat sand dog"].
```

Tags: Dynamic Programming Backtracking

3.2.4 other Covered

- Subsets 2.3.1

- Subsets II 2.3.2

- Combinationas 2.3.4

- Letter Combinationas of Phone Number 2.3.5

- Permutation 2.3.3

- Permutation II 2.3.6

- Palindrome Partitioning 2.5.1

- Restore IP Addresses 2.5.2

- N-Queens 2.5.6

- N-Queens II 2.5.7

- Combination Sum 2.5.4

- Combination Sum II 2.5.5

- Generate Parentheses 2.5.3

- Sudoku Solver 1.6.12

- Word Search 2.5.9

- Word Ladder II 2.4.3

- Regular Expression Matching 2.8.20

- Wild Card Matching `Wild Card Matching`

3.3 Bit Manipulation

3.3.1 Majority Element

Given an array of size n, find the majority element. The majority element is the element that appears more than   n/2   times.

You may assume that the array is non-empty and the majority element always exist in the array.

Credits:

Special thanks to @ts for adding this problem and creating all test cases.

Tags: Divide and Conquer, Array, Bit Manipulation

3.3.2 Subsets: Bit Manipulation

Given a set of distinct integers, S, 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 S = [1,2,3], a solution is:

```
1  [
2    [3],
3    [1],
4    [2],
5    [1,2,3],
6    [1,3],
7    [2,3],
8    [1,2],
9    []
10 ]
```

Tags: Array Backtracking, Bit Manipulation

3.3.3 Single Number

Given an array of integers, every element appears twice except for one. Find that single one.
   Note:
   Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?
   Tags: Hash Table, Bit Manipulation

3.3.4 Single Number II

Given an array of integers, every element appears three times except for one. Find that single one.
   Note:
   Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

3.4   图 Graphics

3.4.1 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 #.

1. First node is labeled as 0. Connect node 0 to both nodes 1 and 2.

2. Second node is labeled as 1. Connect node 1 to node 2.

3. Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle.
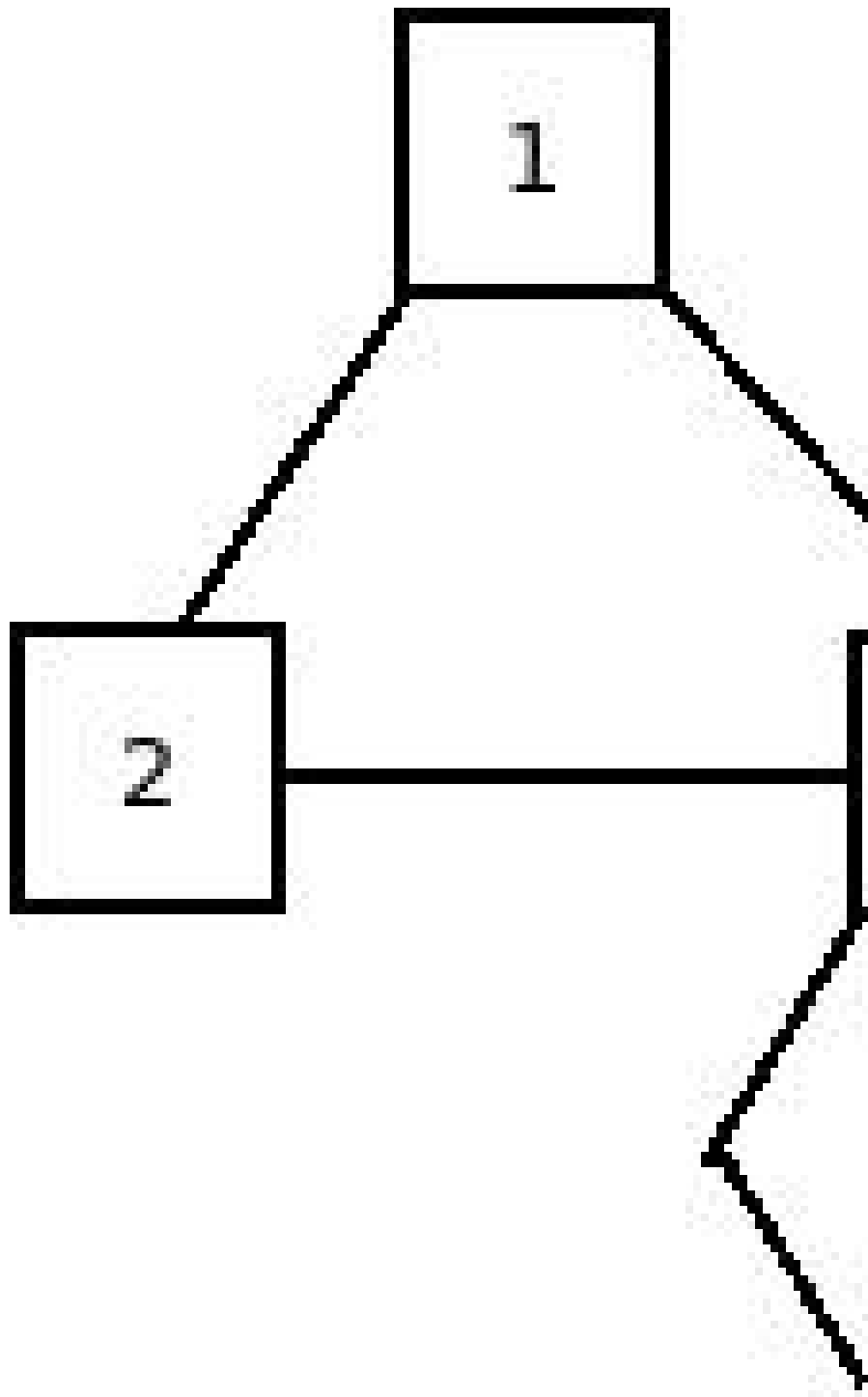
Visually, the graph looks like the following:

1. 分析：广度优先遍历或深度优先遍历都可以

2. DFS：时间复杂度 O(n), 空间复杂度 O(n)

```java
1   /**
2    * Definition for undirected graph.
3    * class UndirectedGraphNode {
4    *     int label;
5    *     List<UndirectedGraphNode> neighbors;
6    *     UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGra
7    * };
8    */
9   public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
10      if (node == null) return null;
11      UndirectedGraphNode res = new UndirectedGraphNode(node.label);  // result head
12      if (node.neighbors == null || node.neighbors.size() == 0) return res;
13
14      Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<UndirectedGraphNode,
15      Queue<UndirectedGraphNode> q = new LinkedList<UndirectedGraphNode>();
16      q.add(node);                           // added first node, need add its all Neighbors a
17      map.put(node, res);
18
19      List<UndirectedGraphNode> curNbr = new ArrayList<UndirectedGraphNode>();
20      UndirectedGraphNode curr = null;
21      while (!q.isEmpty()) {
22          curr = q.poll();
23          curNbr = curr.neighbors;                    // ori
24          for (UndirectedGraphNode aNbr : curNbr) {  // for build connection among copies
25              if (!map.containsKey(aNbr)) {
26                  UndirectedGraphNode acpNbr = new UndirectedGraphNode(aNbr.label);
27                  map.put(aNbr, acpNbr);
28                  map.get(curr).neighbors.add(acpNbr);
29                  q.add(aNbr);
30              } else
31                  map.get(curr).neighbors.add(map.get(aNbr));
32          }
33      }
34      return res;
35  }
```

3. BFS:

3.4.2   Check whether the graph is bigraph

1. Topological Sort Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is "5 4 2 3 1 0 . There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is "4 5 2 3 1 0 . The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).
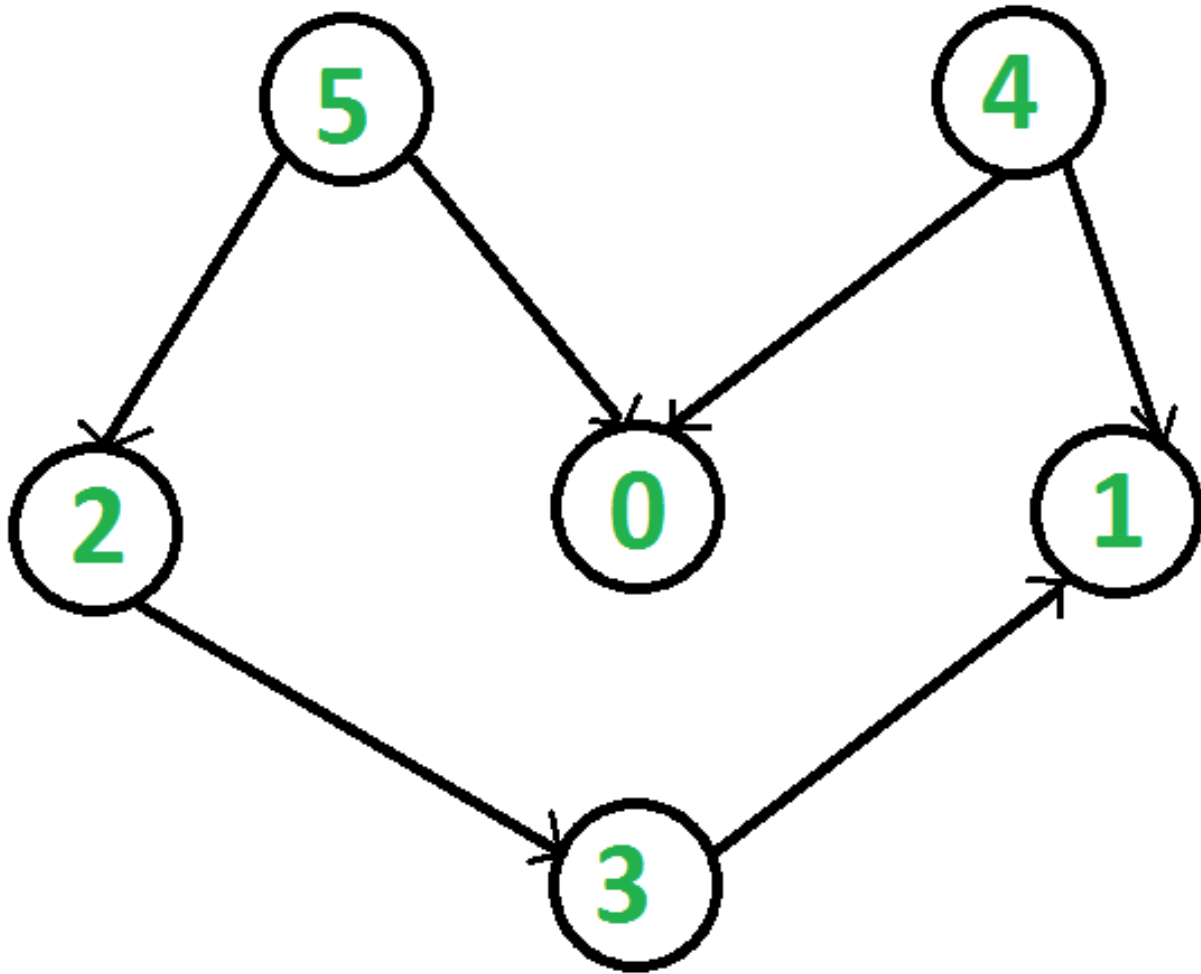
Figure 3.3: Topological Sorting

2. Topological Sorting vs Depth First Traversal (DFS): In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex '5'should be printed before vertex '0', but unlike DFS, the vertex '4'should also be printed before vertex '0'. So Topological sorting is different from DFS. For example, a DFS of the above graph is "5 2 3 1 0 4 , but it is not a topological sorting.

3. Algorithm to find Topological Sorting: We recommend to first see implementation of DFS here. We can modify DFS to find Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

3.4.3 other Covered

- Word Ladder 2.4.2

- Word Ladder II

3.5   Data Structure

3.5.1   Two Sum III

Design and implement a TwoSum class. It should support the following operations: add and find.

- add - Add the number to an internal data structure.

- find - Find if there exists any pair of numbers which sum is equal to the value.

For example,

```
1  add(1); add(3); add(5);
2  find(4) -> true
3  find(7) -> false
```

Tags: Hash Table, Data Structure

3.5.2   Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

- push(x) – Push element x onto stack.

- pop() – Removes the element on top of the stack.

- top() – Get the top element.

- getMin() – Retrieve the minimum element in the stack. **Tags:** Stack Data Structure

```java
1  public static class MinStack {
2      Stack<Integer> stack = new Stack<Integer>();
3      Stack<Integer> minStack = new Stack<Integer>();
4      public void push(int x) {
5          stack.push(x);
6          if (minStack.isEmpty() || x <= minStack.peek()) {
7              minStack.push(x);
8          }
9      }
10
11     public void pop() { // java boxing & unboxing, container, object specific methods
12         if (stack.peek().intValue() == minStack.peek().intValue())
13             minStack.pop();
14         stack.pop();
15     }
16
17     public int top() {
```

```
18          return stack.peek();
19      }
20
21      public int getMin() {
22          if (!minStack.isEmpty()) return minStack.peek();
23          else return -1;
24      }
25  }
```

3.5.3 LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

- get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

- set(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

```
1  public static class LRUCache {
2      public class Node {
3          int key;
4          int value;
5          Node prev;
6          Node next;
7          public Node(int x, int y){
8              key = x;
9              value = y;
10         }
11     }
12
13     private HashMap<Integer, Node> hash;
14     private int cap;
15     private int number;
16     Node head;
17     Node tail;
18     public LRUCache(int capacity) {
19         cap = capacity;
20         number = 0;
21         head = new Node(-1, -1);
22         head.prev = null;
23         head.next = null;
24         tail = head;
25         hash = new HashMap<Integer, Node>(capacity); // so I can restrict a size !!
26     }
27
28     public int get(int key) {
29         Node res = hash.get(new Integer(key)); // don't understand here
```

```java
30          if (res != null) {
31              refresh(res);    // update usage frequence
32              return res.value;
33          } else return -1;
34          /*
35            if (hash.containsKey(key)) {
36            //Node res = hash.get(new Integer(key)); // don't understand here
37            Node res = hash.get(key);
38            refresh(res);    // update usage frequence
39            return res.value;
40            } else {
41            return -1;
42            }
43          */
44      }

45
46      // so still, must maintain a doubly-linked list to order usage frequency
47      public  void refresh(Node tmp) {
48          if (tmp == head.next) return; // it's head already
49
50          Node temp = head.next;   // head node in the hash
51          Node prevNode = tmp.prev;
52          Node nextNode = tmp.next;
53          // set to be most recently used~~ move the tmp node to be head.next, connections
54          head.next = tmp;
55          tmp.prev = head;
56          tmp.next = temp;
57          temp.prev = tmp;
58          prevNode.next = nextNode;
59          if (nextNode != null)
60              nextNode.prev = prevNode;
61          else tail = prevNode;    // remember tail as well
62      }

63
64      public void set(int key, int value) {
65          Node res = hash.get(new Integer(key));
66          if (res != null) {
67              refresh(res);
68              res.value = value;
69          } else {
70              //if (!hash.containsKey(key)) { // another way of detecting existing
71              Node prevNode = new Node(key, value);
72              Node temp;
73              if (number == cap) { // remove tail;
74                  temp = tail.prev;
75                  hash.remove(tail.key);        // remember to remove from hash too !!!
76                  if (temp != null) {
77                      temp.next = null;
```

```
78              }
79              tail.prev = null;
80              tail = temp;
81              number--;
82          }
83          // add to tail first
84          tail.next = prevNode;
85          prevNode.prev = tail;
86          tail = prevNode;
87          refresh(prevNode);
88          hash.put(key, prevNode);
89          number++;  // count node numbers
90      }
91    }
92  }
```

3.6 细节实现题

3.6.1 Pascal's Triangle

Given numRows, generate the first numRows of Pascal's triangle.

For example, given numRows = 5, Return

```
1  [
2       [1],
3      [1,1],
4     [1,2,1],
5    [1,3,3,1],
6   [1,4,6,4,1]
7  ]
```

本题可以用队列, 计算下一行时, 给上一行左右各加一个 0, 然后下一行的每个元素, 就等于左上角和右上角之和。

另一种思路, 下一行第一个元素和最后一个元素赋值为 1, 中间的每个元素, 等于上一行的左上角和右上角元素之和。

1. 从左到右: 时间复杂度 $O(n^2)$, 空间复杂度 $O(n)$

```
1  public List<List<Integer>> generate(int numRows) {
2      List<List<Integer>> res = new ArrayList<List<Integer>>();
3      if (numRows == 0) return res;
4      res.add(new ArrayList<Integer>(Arrays.asList(1)));
5      for (int i = 2; i <= numRows; i++) {
6          List<Integer> curr = new ArrayList<Integer>(i); // 本行
7          for (int j = 0; j < i; j++)
8              curr.add(j, 1);
9          List<Integer> prev = res.get(i - 2);          // 上一行
10         for (int j = 1; j < i - 1; j++)
11             curr.set(j, prev.get(j - 1) + prev.get(j));//左上角和右上角之和
```

```
12            res.add(curr);
13        }
14        return res;
15    }
```

2. 从右到左: 时间复杂度 O(n²), 空间复杂度 O(n)

```
1  public List<List<Integer>> generate(int numRows) {
2      List<List<Integer>> res = new ArrayList<List<Integer>>();
3      List<Integer> arr = new ArrayList<Integer>();
4      for (int i = 1; i <= numRows; i++) {
5          for (int j = i - 2; j > 0; j--) {
6              arr.set(j, arr.get(j - 1) + arr.get(j));
7          }
8          arr.add(1);
9          res.add(new ArrayList<Integer>(arr));
10     }
11     return res;
12 }
```

3.6.2 Pascal's Triangle II

Given an index k, return the kth row of the Pascal's triangle.

For example, given k = 3,

Return [1,3,3,1].

Note: Could you optimize your algorithm to use only O(k) extra space?

滚动数组。

```
1  public List<Integer> getRow(int rowIndex) {
2      List<Integer> arr = new ArrayList<Integer>();
3      for (int i = 0; i <= rowIndex ; i++)
4          for (int j = i - 1; j > 0; j--)
5              arr.set(j, arr.get(j - 1) + arr.get(j));
6      arr.add(1);
7      return arr;
8  }
```

3.6.3 Two Sum

Given an array of integers, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have exactly one solution.

```
1  Input: numbers = {2, 7, 11, 15}, target = 9
2  Output: index1 = 1, index2 = 2
```

Tags: Array Hash Table
方法 1: 暴力, 复杂度 O(n 2), 会超时
方法 2:hash。用一个哈希表, 存储每个数对应的下标, 复杂度 O(n)

1. hash。用一个哈希表, 存储每个数对应的下标, 复杂度 O(n)

```java
public int[] twoSum(int[] numbers, int target) {
    int [] result = new int[2];
    Map<Integer, Integer> hash = new HashMap<Integer, Integer>();
    for (int i = 0; i < numbers.length; i++) {
        if (!hash.containsKey(numbers[i]))
            hash.put(numbers[i], i+1);
        else if (numbers[i] * 2 == target && hash.containsKey(numbers[i])) {
            result[0] = hash.get(numbers[i]) <= i + 1 ? hash.get(numbers[i]) : i+1;
            result[1] = hash.get(numbers[i]) <= i + 1 ? i+1 : hash.get(numbers[i]);
            return result;
        }
    }
    int two;
    for(int i : numbers) {
        two = target - i;
        if (hash.containsKey(two)) {
            if (two != i) {
                result[0] = hash.get(i) <= hash.get(two) ? hash.get(i) : hash.get(two);
                result[1] = hash.get(i) <= hash.get(two) ? hash.get(two) : hash.get(i);
                return result;
            }
        }
    }
    return result;
}
```

自已写的繁琐版的代码，改天再看有没有优化吧。。。

3.6.4 Two Sum II - Input array is sorted

3.6.5 Insert Interval

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).
You may assume that the intervals were initially sorted according to their start times.
Example 1:
Given intervals [1, 3], [6, 9], insert and merge [2, 5] in as [1, 5], [6, 9].
Example 2:
Given [1, 2], [3, 5], [6, 7], [8, 10], [12, 16], insert and merge [4, 9] in as [1, 2], [3, 10], [12, 16].
This is because the new interval [4, 9] overlaps with [3, 5], [6, 7], [8, 10].

```java
public class Interval {
    int start;
    int end;
```

```
4     Interval() {
5         start = 0;
6         end = 0;
7     }
8     Interval(int s, int e) {
9         start = s;
10        end = e;
11    }
12 }
```

1. 时间复杂度 O(n), 空间复杂度 O(1)

   c++ 版:

```cpp
1  class Solution {
2  public:
3      vector<Interval> insert(vector<Interval> &intervals, Interval newInterval) {
4          vector<Interval>::iterator it = intervals.begin();
5          while (it != intervals.end()) {
6              if (newInterval.end < it->start) {
7                  intervals.insert(it, newInterval);
8                  return intervals;
9              } else if (newInterval.start > it->end) {
10                 it++;
11                 continue;
12             } else {
13                 newInterval.start = min(newInterval.start, it->start);
14                 newInterval.end = max(newInterval.end, it->end);
15                 it = intervals.erase(it);
16             }
17         }
18         intervals.insert(intervals.end(), newInterval);
19         return intervals;
20     }
21 };
```

   Java 版:

```java
1  public List<Interval> insert(List<Interval> intervals, Interval newInterval) {
2      intervals.add(newInterval);
3      Collections.sort(intervals, new Comparator<Interval>(){
4              @Override
5                  public int compare(Interval a, Interval b) {
6                  return Integer.compare(a.start, b.start);
7              }
8          });
9      for (int i = 0; i < intervals.size() - 1; i++) {
10         int j = i + 1;
11         if (intervals.get(i).end < intervals.get(j).start)
12             continue;
```

```
13            else if (intervals.get(i).end >= intervals.get(j).start
14                    && intervals.get(i).end < intervals.get(j).end) {
15                intervals.get(i).end = intervals.get(j).end;
16                intervals.remove(intervals.get(j));
17                --i;
18            } else {
19                intervals.remove(intervals.get(j));
20                --i;
21            }
22        }
23        return intervals;
24    }
```

3.6.6 Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

For example,

Given [1, 3], [2, 6], [8, 10], [15, 18],

return [1, 6], [8, 10], [15, 18].

复用一下 Insert Intervals 的解法即可, 创建一个新的 interval 集合, 然后每次从旧的里面取一个 interval 出来, 然后插入到新的集合中。

c++ 版:

```
1  // LeetCode, Merge Interval
2  // 复用一下 Insert Intervals 的解法即可
3  // 时间复杂度 O(n1+n2+...), 空间复杂度 O(1)
4  class Solution {
5  public:
6      vector<Interval> merge(vector<Interval> &intervals) {
7          vector<Interval> result;
8          for (int i = 0; i < intervals.size(); i++) {
9              insert(result, intervals[i]);
10         }
11         return result;
12     }
13 private:
14     vector<Interval> insert(vector<Interval> &intervals, Interval newInterval) {
15         vector<Interval>::iterator it = intervals.begin();
16         while (it != intervals.end()) {
17             if (newInterval.end < it->start) {
18                 intervals.insert(it, newInterval);
19                 return intervals;
20             } else if (newInterval.start > it->end) {
21                 it++;
22                 continue;
23             } else {
24                 newInterval.start = min(newInterval.start, it->start);
25                 newInterval.end = max(newInterval.end, it->end);
```

```java
26                it = intervals.erase(it);
27            }
28        }
29        intervals.insert(intervals.end(), newInterval);
30        return intervals;
31    }
32 };
```

```java
1  public List<Interval> merge(List<Interval> intervals) {
2      Collections.sort(intervals, new Comparator<Interval>(){
3              @Override
4                  public int compare(Interval a, Interval b) {
5                  return Integer.compare(a.start, b.start);
6              }
7          });
8      for (int i = 0; i < intervals.size() - 1; i++) {
9          int j = i + 1;
10         if (intervals.get(i).end < intervals.get(j).start)
11             continue;
12         else if (intervals.get(i).end >= intervals.get(j).start
13                 && intervals.get(i).end < intervals.get(j).end) {
14             intervals.get(i).end = intervals.get(j).end;
15             intervals.remove(intervals.get(j));
16             --i;
17         } else {
18             intervals.remove(intervals.get(j));
19             --i;
20         }
21     }
22     return intervals;
23 }
```

3.6.7   Spiral Matrix

Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order.
   For example, Given the following matrix:

```
1  [
2   [ 1, 2, 3 ],
3   [ 4, 5, 6 ],
4   [ 7, 8, 9 ]
5  ]
```

You should return [1, 2, 3, 6, 9, 8, 7, 4, 5].
模拟。时间复杂度 $O(n^2)$, 空间复杂度 $O(1)$
c++ 版:

```cpp
1  vector<int> spiralOrder(vector<vector<int> >& matrix) {
2      vector<int> result;
```

```
3        if (matrix.empty()) return result;
4        int beginX = 0, endX = matrix[0].size() - 1;
5        int beginY = 0, endY = matrix.size() - 1;
6        while (true) {
7            // From left to right
8            for (int j = beginX; j <= endX; ++j) result.push_back(matrix[beginY][j]);
9            if (++beginY > endY) break;
10
11            // From top to bottom
12            for (int i = beginY; i <= endY; ++i) result.push_back(matrix[i][endX]);
13            if (beginX > --endX) break;
14
15            // From right to left
16            for (int j = endX; j >= beginX; --j) result.push_back(matrix[endY][j]);
17            if (beginY > --endY) break;
18
19            // From bottom to top
20            for (int i = endY; i >= beginY; --i) result.push_back(matrix[i][beginX]);
21            if (++beginX > endX) break;
22        }
23        return result;
24    }
```

```
1    public List<Integer> spiralOrder(int[][] matrix) {
2        List<Integer> res = new ArrayList<Integer>();
3        if (matrix == null || matrix.length == 0) return res;
4        int size = matrix.length * matrix[0].length;
5        int ibgn = 0, iend = matrix.length - 1;
6        int jbgn = 0, jend = matrix[0].length - 1;
7        boolean toup = false, toleft = false; // 向上，向左
8        while (res.size() < size && ibgn < matrix.length && jbgn < matrix[0].length) {
9            for (int j = jbgn; j <= jend; j++) {
10                if (res.size() == size) break;
11                res.add(matrix[ibgn][j]);
12            }
13            ++ibgn;
14            toleft = true;
15            for (int j = ibgn; j <= iend; j++) {
16                if (res.size() == matrix.length * matrix[0].length) break;
17                res.add(matrix[j][jend]);
18            }
19            --jend;
20            toup = true;
21            if (toleft && res.size() < matrix.length * matrix[0].length) {
22                for (int j = jend; j >= jbgn ; j --) {
23                    if (res.size() == matrix.length * matrix[0].length) break;
24                    res.add(matrix[iend][j]);
25                }
```

```
26            --iend;
27            toleft = false;
28        }
29        if (toup && res.size() < matrix.length * matrix[0].length) {
30            for (int i = iend; i >= ibgn ; i --) {
31                if (res.size() == matrix.length * matrix[0].length) break;
32                res.add(matrix[i][jbgn]);
33            }
34            ++jbgn;
35            toup = false;
36        }
37    }
38    return res;
39 }
```

3.6.8   Spiral Matrix II

Given an integer n, generate a square matrix filled with elements from 1 to n^2 in spiral order.

For example, Given n = 3,

You should return the following matrix:

```
1  [
2   [ 1, 2, 3 ],
3   [ 8, 9, 4 ],
4   [ 7, 6, 5 ]
5  ]
```

这题比上一题要简单。

1. 时间复杂度 $O(n^2)$, 空间复杂度 $O(n^2)$

```
1  public int[][] generateMatrix(int n) {
2      int [][] mat = new int[n][n];
3      int bgn = 0, end = n - 1;
4      int num = 1;
5      while (bgn < end) {
6          for (int j = bgn; j < end; j++) mat[bgn][j] = num++;
7          for (int i = bgn; i < end; i++) mat[i][end] = num++;
8          for (int j = end; j > bgn; j--) mat[end][j] = num++;
9          for (int i = end; i > bgn; i--) mat[i][bgn] = num++;
10         ++bgn;
11         --end;
12     }
13     if (bgn == end) mat[bgn][bgn] = num;
14     return mat;
15 }
```

2. 时间复杂度 $O(n^2)$, 空间复杂度 $O(n^2)$

```
1  public int[][] generateMatrix(int n) {
2      int[][] mat = new int[n][n];
3      int ibgn = 0, iend = n - 1;
4      int jbgn = 0, jend = n - 1;
5      int cnt = 1;
6      while (true) {
7          for (int j = jbgn; j <= jend; j++) mat[ibgn][j] = cnt++;
8          if (++ibgn > iend) break;
9
10         for (int j = ibgn; j <= iend; j++) mat[j][jend] = cnt++;
11         if (jbgn > --jend) break;
12
13         for (int j = jend; j >= jbgn ; j--) mat[iend][j] = cnt++;
14         if (ibgn > --iend) break;
15
16         for (int i = iend; i >= ibgn ; i--) mat[i][jbgn] = cnt++;
17         if (++jbgn > jend) break;
18     }
19     return mat;
20 }
```

同自己的思路很像，但是要精练很多。。。

3.6.9 Multiply Strings

Given two numbers represented as strings, return multiplication of the numbers as a string.

Note: The numbers can be arbitrarily large and are non-negative.

高精度乘法。

常见的做法是将字符转化为一个 int, 一一对应, 形成一个 int 数组。但是这样很浪费空间, 一个 int32 的最大值是 $2^{31} - 1 = 2147483647$, 可以与 9 个字符对应, 由于有乘法, 减半, 则至少可以与 4 个字符一一对应。一个 int64 可以与 9 个字符对应。

1. 一个字符对应一个 int: 时间复杂度 O(n*m), 空间复杂度 O(n+m)

2. 9 个字符对应一个 int64_t: 时间复杂度 O(n*m/81), 空间复杂度 O((n+m)/9)

   看得头大，自己的土方法:

```
1  public int getVal(char x) {
2      switch (x) {
3      case '0': return 0;
4      case '1': return 1;
5      case '2': return 2;
6      case '3': return 3;
7      case '4': return 4;
8      case '5': return 5;
9      case '6': return 6;
10     case '7': return 7;
11     case '8': return 8;
12     case '9': return 9;
```

```java
13          }
14          return 0;
15      }
16
17      public String myMul(String a, char s) {
18          StringBuffer res = new StringBuffer();
19          int b = getVal(s);
20          int leftover = 0;
21          int curr = 0;
22          for (int i = a.length() - 1; i >= 0 ; i--) {
23              curr = (getVal(a.charAt(i)) * b + leftover) % 10 ;
24              leftover = (getVal(a.charAt(i)) * b + leftover) / 10;
25              res.insert(0, curr);
26          }
27          if (leftover > 0)
28              res.insert(0, leftover);
29          return res.toString();
30      }
31
32      public String myAdd(StringBuffer a, StringBuffer b) {
33          StringBuffer res;
34          res = new StringBuffer(a.length() < b.length() ? a : b);
35          a = a.length() < b.length() ? b : a; // long
36          b = res;    // short
37          int gap = a.length() - b.length();
38          int leftover = 0;
39          int curr = 0;
40          res = new StringBuffer("");
41          for (int i = b.length() - 1; i >= 0; i--) {
42              curr = (getVal(a.charAt(i+gap)) + getVal(b.charAt(i)) + leftover) % 10;
43              leftover = (getVal(a.charAt(i+gap)) + getVal(b.charAt(i)) + leftover) / 10;
44              res.insert(0, curr);
45          }
46          for (int i = gap - 1; i >= 0; i--) {
47              curr = (getVal(a.charAt(i)) + leftover) % 10;
48              leftover = (getVal(a.charAt(i)) + leftover) / 10;
49              res.insert(0, curr);
50          }
51          if (leftover > 0)
52              res.insert(0, leftover);
53          return res.toString();
54      }
55
56      public String multiply(String num1, String num2) {
57          if (num1 == null || num2 == null || num1.length() == 0 || num2.length() == 0) return
58          if (num1.equals("0") || num2.equals("0")) return "0";  // forgot "0" case
59          StringBuffer res, tmp = new StringBuffer();
60          String temp;
```

```
61        res = new StringBuffer(num1.length() < num2.length() ? num1 : num2);
62        num1 = num1.length() < num2.length() ? num2 : num1; // long
63        num2 = res.toString();    // short
64        for (int i = num2.length() - 1; i >= 0 ; i--) {
65            temp = myMul(num1, num2.charAt(i));
66            if (i == num2.length() - 1)
67                res = new StringBuffer(temp);
68            else {
69                tmp = new StringBuffer(temp);
70                for (int j = 0; j < num2.length() - 1 - i; j++)
71                    tmp.append('0');
72                res = new StringBuffer(myAdd(res, tmp));
73            }
74        }
75        return res.toString();
76    }
```

3.6.10   Substring with Concatenation of All Words

You are given a string, S, and a list of words, L, that are all of the same length. Find all starting indices of substring(s) in S that is a concatenation of each word in L exactly once and without any intervening characters.

For example, given:

```
1  S: "barfoothefoobarman"
2  L: ["foo", "bar"]
```

You should return the indices: [0,9]. (order does not matter).

Tags: Hash Table, Two Pointers, String

```
1  public List<Integer> findSubstring(String s, String[] l) {
2      List<Integer> res = new ArrayList<Integer>();
3      int n = l.length;
4      int m = l[0].length();
5      if (s.length() < n * m) return res;
6      Map<String, Integer> hash = new HashMap<String, Integer>();
7      for(String i : l)
8          if (hash.containsKey(i)) hash.put(i, hash.get(i) + 1);
9          else hash.put(i, 1);
10     Map<String, Integer> fnd = new HashMap<String, Integer>();
11     String tmp;
12     int cnt = 0;
13     int j;
14     for (int i = 0; i <= s.length() - m * n; i++) {
15         tmp = s.substring(i, i + m);
16         cnt = 0;
17         j = i;
18         fnd.clear();
19         while (hash.containsKey(tmp) && (!fnd.containsKey(tmp) || fnd.get(tmp) < hash.get(tmp
```

```
20              if (!fnd.containsKey(tmp)) fnd.put(tmp, 1);
21              else fnd.put(tmp, fnd.get(tmp) + 1);
22              ++cnt;
23              j += m;
24              if (j + m <= s.length())
25                  tmp = s.substring(j, j + m);
26          }
27          if (cnt == n) res.add(i);
28      }
29      return res;
30  }
```

c++ 版:

```
1  vector<int> findSubstring(string s, vector<string>& dict) {
2      size_t wordLength = dict.front().length();
3      size_t catLength = wordLength * dict.size();
4      vector<int> result;
5      if (s.length() < catLength) return result;
6      unordered_map<string, int> wordCount;
7      for (auto const& word : dict) ++wordCount[word];
8      for (auto i = begin(s); i <= prev(end(s), catLength); ++i) {
9          unordered_map<string, int> unused(wordCount);
10         for (auto j = i; j != next(i, catLength); j += wordLength) {
11             auto pos = unused.find(string(j, next(j, wordLength)));
12             if (pos == unused.end() || pos->second == 0) break;
13             if (--pos->second == 0) unused.erase(pos);
14         }
15         if (unused.size() == 0) result.push_back(distance(begin(s), i));
16     }
17     return result;
18  }
```

3.6.11 ZigZag Conversion

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

```
1  P   A   H   N
2  A P L S I I G
3  Y   I   R
```

And then read line by line: "PAHNAPLSIIGYIR"
    Write the code that will take a string and make this conversion given a number of rows:
    string convert(string text, int nRows);
    convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".
    要找到数学规律。真正面试中, 不大可能出这种问题。

```
1  n = 4:
2  P        I        N
```

```
 3   A       L   S       I   G
 4   Y   A       H   R
 5   P           I
 6   n = 5:
 7   P                   H
 8   A               S   I
 9   Y           I       R
10   P   L               I   G
11   A                   N
```

0				8				16		
1			7	9			15	17		
2		6		10		14		18		22
3	5			11	13			19	21	
4				12				20		

Figure 3.4: ZigZag Conversion

所以, 对于每一层垂直元素的坐标 $(i, j) = (j + 1) * n + i$; 对于每两层垂直元素之间的插入元素 (斜对角元素),$(i, j) = (j + 1) * n - i$

```java
public String convert(String s, int nRows) {
    if (nRows <= 1 || s.length() <= nRows) return s;
    StringBuffer result = new StringBuffer();
    for (int i = 0; i < nRows; i++) {
        for (int j = 0, idx = i; idx < s.length();
                j++, idx = (2 * nRows - 2) * j + i) {
            result.append(s.charAt(idx));        // 垂直元素
            if (i == 0 || i == nRows - 1) continue; // 斜对角元素
            if (idx + (nRows - i - 1) * 2 < s.length())
                result.append(s.charAt(idx + (nRows - i - 1) * 2));
        }
    }
    return result.toString();
}
```

```cpp
string convert(string s, int nRows) {
    if (nRows <= 1 || s.size() <= 1) return s;
    string result;
    for (int i = 0; i < nRows; i++) {
        for (int j = 0, index = i; index < s.size();
                j++, index = (2 * nRows - 2) * j + i) {
            result.append(1, s[index]);          // 垂直元素
            if (i == 0 || i == nRows - 1) continue; // 斜对角元素
            if (index + (nRows - i - 1) * 2 < s.size())
                result.append(1, s[index + (nRows - i - 1) * 2]);
        }
    }
    return result;
}
```

3.6.12 Text Justification, bug...

Given an array of words and a length L, format the text such that each line has exactly L characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly L characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

For example,

```
words: ["This", "is", "an", "example", "of", "text", "justification."]
L: 16.
```

Return the formatted lines as:

```
[
    "This    is    an",
    "example  of text",
    "justification.  "
]
```

Note: Each word is guaranteed not to exceed L in length.

click to show corner cases.

Corner Cases:

A line other than the last line might contain only one word. What should you do in this case?

In this case, that line should be left-justified.

时间复杂度 O(n), 空间复杂度 O(1).

c++ 版:

```cpp
// LeetCode, Text Justification
// 时间复杂度 O(n), 空间复杂度 O(1)
class Solution {
public:
    vector<string> fullJustify(vector<string> &words, int L) {
        vector<string> result;
        const int n = words.size();
        int begin = 0, len = 0; // 当前行的起点，当前长度
        for (int i = 0; i < n; ++i) {
            if (len + words[i].size() + (i - begin) > L) {
                result.push_back(connect(words, begin, i - 1, len, L, false));
                begin = i;
                len = 0;
            }
            len += words[i].size();
        }
        // 最后一行不足 L
        result.push_back(connect(words, begin, n - 1, len, L, true));
        return result;
```

```cpp
20        }
21        /**
22         * @brief 将 words[begin, end] 连成一行
23         * @param[in] words 单词列表
24         * @param[in] begin 开始
25         * @param[in] end 结束
26         * @param[in] len words[begin, end] 所有单词加起来的长度
27         * @param[in] L 题目规定的一行长度
28         * @param[in] is_last 是否是最后一行
29         * @return 对齐后的当前行
30         */
31        string connect(vector<string> &words, int begin, int end,
32                       int len, int L, bool is_last) {
33            string s;
34            int n = end - begin + 1;
35            for (int i = 0; i < n; ++i) {
36                s += words[begin + i];
37                addSpaces(s, i, n - 1, L - len, is_last);
38            }
39            if (s.size() < L) s.append(L - s.size(), ' ');
40            return s;
41        }
42        /**
43         * @brief 添加空格.
44         * @param[inout]s 一行
45         * @param[in] i 当前空隙的序号
46         * @param[in] n 空隙总数
47         * @param[in] L 总共需要添加的空额数
48         * @param[in] is_last 是否是最后一行
49         * @return 无
50         */
51        void addSpaces(string &s, int i, int n, int L, bool is_last) {
52            if (n < 1 || i > n - 1) return;
53            int spaces = is_last ? 1 : (L / n + (i < (L % n) ? 1 : 0));
54            s.append(spaces, ' ');
55        }
56   };
```

Java has a bug, will fix it later…..

```java
1  public StringBuffer addSpaces(StringBuffer s, int i, int n, int L, boolean last) {
2      if (n < 1 || i > n - 1) return s;
3      int spaces = last ? 1 : (L / n + (i < (L % n) ? 1 : 0));
4      for (int j = 0; j < spaces; j++)
5          s.append(" ");
6      return s;
7  }
8
9  public StringBuffer connect(String [] words, int bgn, int end, int len, int L, boolean last)
```

```java
10        StringBuffer s = new StringBuffer();
11        int n = end - bgn + 1;
12        for (int i = 0; i < n; i++) {
13            s.append(words[bgn + i]);
14            s = addSpaces(s, i, n - 1, L - len, last);
15        }
16        if (s.length() < L)
17            for (int j = 0; j < L - s.length(); j++)
18                s.append(' ');
19        return s;
20    }
21
22    public List<String> fullJustify(String[] words, int L) {
23        List<String> res = new ArrayList<String>();
24        int n = words.length;
25        int bgn = 0, len = 0;
26        for (int i = 0; i < n; i++) {
27            if (len + words[i].length() + (i - bgn) > L) {
28                res.add(connect(words, bgn, i - 1, len, L, false).toString());
29                bgn = i;
30                len = 0;
31            }
32            len += words[i].length();
33        }
34        res.add(connect(words, bgn, n - 1, len, L, true).toString());
35        return res;
36    }
```

3.6.13   other Covered

- Majority Element

- Container With Most Water

- Minimum Path Sum

- Missing Ranges

- Find Minimum in Rotated Sorted Array

- Triangle

- Find Peak Element

- Find Minimum in Rotated Sorted Array II

- Jump Game II

- Word Ladder II

3.7 Math

3.7.1 Add Binary

3.7.2 String to Integer (atoi)

3.7.3 Palindrome Number

3.7.4 Factorial Trailing Zeroes

3.7.5 Excel Sheet Column Title

3.7.6 Excel Sheet Column Number

3.7.7 Reverse Integer

3.7.8 Multiply Strings

3.7.9 Divide Two Integers

3.7.10 Fraction to Recurring Decimal

3.7.11 Permutation Sequence

3.7.12 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  1,2,3 → 1,3,2
2  3,2,1 → 1,2,3
3  1,1,5 → 1,5,1
```
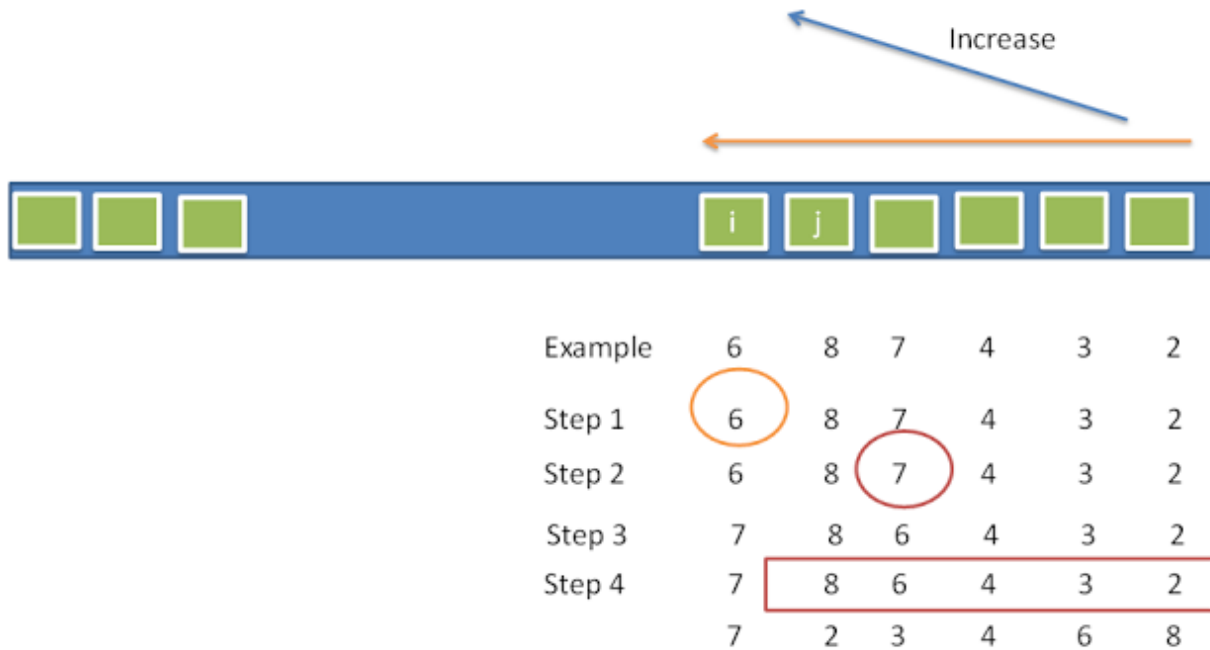
算法过程如图所示

```cpp
// LeetCode, Next Permutation
// 时间复杂度 O(n)，空间复杂度 O(1)
class Solution {
public:
    void nextPermutation(vector<int> &num) {
        next_permutation(num.begin(), num.end());
    }
    template<typename BidiIt>
    bool next_permutation(BidiIt first, BidiIt last) {
        // Get a reversed range to simplify reversed traversal.
        const auto rfirst = reverse_iterator<BidiIt>(last);
```

Figure 3.5: Next Permutation

```
12          const auto rlast = reverse_iterator<BidiIt>(first);
13          // Begin from the second last element to the first element.
14          auto pivot = next(rfirst);
15          // Find 'pivot', which is the first element that is no less than its
16          // successor. 'Prev' is used since 'pivort' is a 'reversed_iterator'.
17          while (pivot != rlast and !(*pivot < *prev(pivot)))
18              ++pivot;
19          // No such elemenet found, current sequence is already the largest
20          // permutation, then rearrange to the first permutation and return false.
21          if (pivot == rlast) {
22              reverse(rfirst, rlast);
23              return false;
24          }
25          // Scan from right to left, find the first element that is greater than
26          // 'pivot'.
27          auto change = find_if(rfirst, pivot, bind1st(less<int>(), *pivot));
28          swap(*change, *pivot);
29          reverse(rfirst, pivot);
30          return true;
```

```
31        }
32   };
```

```
1   public void swap(int [] a, int i, int j) {
2        int tmp = a[i];
3        a[i] = a[j];
4        a[j] = tmp;
5   }
6
7   public void nextPermutation(int[] a) {
8        if (a == null || a.length < 2) return;
9        int n = a.length - 1;
10       int i = n;
11       int j = n - 1;
12       while (i >= 1) {
13           while (j >= 0 && a[j] > a[i]) {
14               j--;
15               i--;
16           }
17           if (j < 0 && i >= 1 ||
18               (j >= 0 && i >= 1 && a[j] == a[i] && i - j >= 1)) {
19               i--;
20               j = i - 1;
21           } else if (j >= 0 && a[j] < a[i]) {
22               while (i <= n && a[i] > a[j]) i++;
23               swap(a, j, i - 1);
24               Arrays.sort(a, j + 1, n + 1);
25               return;
26           }
27           if (i < 1) {
28               Arrays.sort(a);
29               return;
30           }
31       }
32   }
```

3.7.13   Valid Number

3.7.14   Max Points on a Line

3.7.15   other Covered

- Plus One 1.1.3

- Roman to Integers `Roman to Integers`

- Integer to Roman 1.2.10

- Pow(x, n) 2.6.1

- Sqrt(x) 2.6.2