Leetcode 题解

原作者github: https://github.com/CyC2018/Interview-Notebook

PDF制作github: https://github.com/sjsdfg/Interview-Notebook-PDF

算法思想

贪心思想

贪心思想保证每次操作都是局部最优的，并且最后得到的结果是全局最优的。

分配饼干

455. Assign Cookies (Easy)

```
1.   Input: [1,2], [1,2,3]
2.   Output: 2
3.
4.   Explanation: You have 2 children and 3 cookies. The greed factors of 2
     children are 1, 2.
5.   You have 3 cookies and their sizes are big enough to gratify all of th
     e children,
6.   You need to output 2.
```

题目描述：每个孩子都有一个满足度，每个饼干都有一个大小，只有饼干的大小大于等于一个孩子的满足度，该孩子才会获得满足。求解最多可以获得满足的孩子数量。

因为最小的孩子最容易得到满足，因此先满足最小孩子。给一个孩子的饼干应当尽量小又能满足该孩子，这样大饼干就能拿来给满足度比较大的孩子。因此贪心策略

证明：假设在某次选择中，贪心策略选择给当前满足度最小的孩子分配第 m 个饼干，第 m 个饼干为可以满足该孩子的最小饼干。假设存在一种最优策略，给该孩子分配第 n 个饼干，并且 m < n。我们可以发现，经过这一轮分配，贪心策略分配后剩下的饼干一定有一个比最优策略来得大。因此在后续的分配中，贪心策略一定能满足更多的孩子。也就是说不存在比贪心策略

更优的策略，即贪心策略就是最优策略。

```java
public int findContentChildren(int[] g, int[] s) {
    Arrays.sort(g);
    Arrays.sort(s);
    int gi = 0, si = 0;
    while (gi < g.length && si < s.length) {
        if (g[gi] <= s[si]) {
            gi++;
        }
        si++;
    }
    return gi;
}
```

不重叠的区间个数

435. Non-overlapping Intervals (Medium)

```
Input: [ [1,2], [1,2], [1,2] ]

Output: 2

Explanation: You need to remove two [1,2] to make the rest of intervals non-overlapping.
```

```
Input: [ [1,2], [2,3] ]

Output: 0

Explanation: You don't need to remove any of the intervals since they're already non-overlapping.
```

题目描述：计算让一组区间不重叠所需要移除的区间个数。

计算最多能组成的不重叠区间个数，然后用区间总个数减去不重叠区间的个数。

在每次选择中，区间的结尾最为重要，选择的区间结尾越小，留给后面的区间的空间越大，那么后面能够选择的区间个数也就越大。

按区间的结尾进行排序，每次选择结尾最小，并且和前一个区间不重叠的区间。

```java
public int eraseOverlapIntervals(Interval[] intervals) {
    if (intervals.length == 0) {
        return 0;
    }
    Arrays.sort(intervals, Comparator.comparingInt(o -> o.end));
    int cnt = 1;
    int end = intervals[0].end;
    for (int i = 1; i < intervals.length; i++) {
        if (intervals[i].start < end) {
            continue;
        }
        end = intervals[i].end;
        cnt++;
    }
    return intervals.length - cnt;
}
```

使用 lambda 表示式创建 Comparator 会导致算法运行时间过长，如果注重运行时间，可以修改为普通创建 Comparator 语句：

```java
Arrays.sort(intervals, new Comparator<Interval>() {
    @Override
    public int compare(Interval o1, Interval o2) {
        return o1.end - o2.end;
    }
});
```

投飞镖刺破气球

452. Minimum Number of Arrows to Burst Balloons (Medium)

```
Input:
[[10,16], [2,8], [1,6], [7,12]]

Output:
2
```

题目描述：气球在一个水平数轴上摆放，可以重叠，飞镖垂直投向坐标轴，使得路径上的气球

都会刺破。求解最小的投飞镖次数使所有气球都被刺破。

也是计算不重叠的区间个数，不过和 Non-overlapping Intervals 的区别在于，[1, 2] 和 [2, 3] 在本题中算是重叠区间。

```java
public int findMinArrowShots(int[][] points) {
    if (points.length == 0) {
        return 0;
    }
    Arrays.sort(points, Comparator.comparingInt(o -> o[1]));
    int cnt = 1, end = points[0][1];
    for (int i = 1; i < points.length; i++) {
        if (points[i][0] <= end) {
            continue;
        }
        cnt++;
        end = points[i][1];
    }
    return cnt;
}
```

根据身高和序号重组队列

[406. Queue Reconstruction by Height(Medium)](#)

```
Input:
[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Output:
[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]
```

题目描述：一个学生用两个分量 (h, k) 描述，h 表示身高，k 表示排在前面的有 k 个学生的身高比他高或者和他一样高。

为了在每次插入操作时不影响后续的操作，身高较高的学生应该先做插入操作，否则身高较小的学生原先正确插入第 k 个位置可能会变成第 k+1 个位置。

身高降序、k 值升序，然后按排好序的顺序插入队列的第 k 个位置中。

```java
public int[][] reconstructQueue(int[][] people) {
```

```
 2.      if (people == null || people.length == 0 || people[0].length == 0)
     {
 3.          return new int[0][0];
 4.      }
 5.      Arrays.sort(people, (a, b) -> (a[0] == b[0] ? a[1] - b[1] : b[0] - a
     [0]));
 6.      List<int[]> queue = new ArrayList<>();
 7.      for (int[] p : people) {
 8.          queue.add(p[1], p);
 9.      }
10.      return queue.toArray(new int[queue.size()][]);
11.  }
```

分隔字符串使同种字符出现在一起

763. Partition Labels (Medium)

```
 1.  Input: S = "ababcbacadefegdehijhklij"
 2.  Output: [9,7,8]
 3.  Explanation:
 4.  The partition is "ababcbaca", "defegde", "hijhklij".
 5.  This is a partition so that each letter appears in at most one part.
 6.  A partition like "ababcbacadefegde", "hijhklij" is incorrect, because
     it splits S into less parts.
```

```
 1.  public List<Integer> partitionLabels(String S) {
 2.      int[] lastIndexsOfChar = new int[26];
 3.      for (int i = 0; i < S.length(); i++) {
 4.          lastIndexsOfChar[char2Index(S.charAt(i))] = i;
 5.      }
 6.      List<Integer> partitions = new ArrayList<>();
 7.      int firstIndex = 0;
 8.      while (firstIndex < S.length()) {
 9.          int lastIndex = firstIndex;
10.          for (int i = firstIndex; i < S.length() && i <= lastIndex; i++)
     {
11.              int index = lastIndexsOfChar[char2Index(S.charAt(i))];
12.              if (index > lastIndex) {
13.                  lastIndex = index;
14.              }
15.          }
16.          partitions.add(lastIndex - firstIndex + 1);
17.          firstIndex = lastIndex + 1;
```

```
18.          }
19.      return partitions;
20.  }
21.
22.  private int char2Index(char c) {
23.      return c - 'a';
24.  }
```

种植花朵

605. Can Place Flowers (Easy)

```
1.   Input: flowerbed = [1,0,0,0,1], n = 1
2.   Output: True
```

题目描述：花朵之间至少需要一个单位的间隔，求解是否能种下 n 朵花。

```
1.   public boolean canPlaceFlowers(int[] flowerbed, int n) {
2.       int len = flowerbed.length;
3.       int cnt = 0;
4.       for (int i = 0; i < len && cnt < n; i++) {
5.           if (flowerbed[i] == 1) {
6.               continue;
7.           }
8.           int pre = i == 0 ? 0 : flowerbed[i - 1];
9.           int next = i == len - 1 ? 0 : flowerbed[i + 1];
10.          if (pre == 0 && next == 0) {
11.              cnt++;
12.              flowerbed[i] = 1;
13.          }
14.      }
15.      return cnt >= n;
16.  }
```

判断是否为子序列

392. Is Subsequence (Medium)

```
1.   s = "abc", t = "ahbgdc"
2.   Return true.
```

```java
public boolean isSubsequence(String s, String t) {
    int index = -1;
    for (char c : s.toCharArray()) {
        index = t.indexOf(c, index + 1);
        if (index == -1) {
            return false;
        }
    }
    return true;
}
```

修改一个数成为非递减数组

665. Non-decreasing Array (Easy)

```
Input: [4,2,3]
Output: True
Explanation: You could modify the first 4 to 1 to get a non-decreasing
array.
```

题目描述：判断一个数组能不能只修改一个数就成为非递减数组。

在出现 nums[i] < nums[i - 1] 时，需要考虑的是应该修改数组的哪个数，使得本次修改能使 i 之前的数组成为非递减数组，并且 **不影响后续的操作** 。优先考虑令 nums[i - 1] = nums[i]，因为如果修改 nums[i] = nums[i - 1] 的话，那么 nums[i] 这个数会变大，就有可能比 nums[i + 1] 大，从而影响了后续操作。还有一个比较特别的情况就是 nums[i] < nums[i - 2]，只修改 nums[i - 1] = nums[i] 不能使数组成为非递减数组，只能修改 nums[i] = nums[i - 1]。

```java
public boolean checkPossibility(int[] nums) {
    int cnt = 0;
    for (int i = 1; i < nums.length && cnt < 2; i++) {
        if (nums[i] >= nums[i - 1]) {
            continue;
        }
        cnt++;
        if (i - 2 >= 0 && nums[i - 2] > nums[i]) {
            nums[i] = nums[i - 1];
        } else {
            nums[i - 1] = nums[i];
```

```
12.            }
13.        }
14.        return cnt <= 1;
15.    }
```

股票的最大收益

122. Best Time to Buy and Sell Stock II (Easy)

题目描述：一次股票交易包含买入和卖出，多个交易之间不能交叉进行。

对于 [a, b, c, d]，如果有 a <= b <= c <= d，那么最大收益为 d - a。而 d - a = (d - c) + (c - b) + (b - a)，因此当访问到一个 prices[i] 且 prices[i] - prices[i-1] > 0，那么就把 prices[i] - prices[i-1] 添加到收益中，从而在局部最优的情况下也保证全局最优。

```
1.    public int maxProfit(int[] prices) {
2.        int profit = 0;
3.        for (int i = 1; i < prices.length; i++) {
4.            if (prices[i] > prices[i - 1]) {
5.                profit += (prices[i] - prices[i - 1]);
6.            }
7.        }
8.        return profit;
9.    }
```

双指针

双指针主要用于遍历数组，两个指针指向不同的元素，从而协同完成任务。

有序数组的 Two Sum

Leetcode ：167. Two Sum II - Input array is sorted (Easy)

```
1.    Input: numbers={2, 7, 11, 15}, target=9
2.    Output: index1=1, index2=2
```

题目描述：在有序数组中找出两个数，使它们的和为 target。

使用双指针，一个指针指向值较小的元素，一个指针指向值较大的元素。指向较小元素的指针从头向尾遍历，指向较大元素的指针从尾向头遍历。

如果两个指针指向元素的和 sum == target，那么得到要求的结果；如果 sum > target，移动较大的元素，使 sum 变小一些；如果 sum < target，移动较小的元素，使 sum 变大一些。

```java
public int[] twoSum(int[] numbers, int target) {
    int i = 0, j = numbers.length - 1;
    while (i < j) {
        int sum = numbers[i] + numbers[j];
        if (sum == target) {
            return new int[]{i + 1, j + 1};
        } else if (sum < target) {
            i++;
        } else {
            j--;
        }
    }
    return null;
}
```

两数平方和

633. Sum of Square Numbers (Easy)

```
Input: 5
Output: True
Explanation: 1 * 1 + 2 * 2 = 5
```

题目描述：判断一个数是否为两个数的平方和，例如 $5 = 1^2 + 2^2$。

```java
public boolean judgeSquareSum(int c) {
    int i = 0, j = (int) Math.sqrt(c);
    while (i <= j) {
        int powSum = i * i + j * j;
        if (powSum == c) {
            return true;
        } else if (powSum > c) {
            j--;
```

```
9.        } else {
10.            i++;
11.        }
12.    }
13.    return false;
14. }
```

反转字符串中的元音字符

345. Reverse Vowels of a String (Easy)

```
1.    Given s = "leetcode", return "leotcede".
```

使用双指针，指向待反转的两个元音字符，一个指针从头向尾遍历，一个指针从尾到头遍历。

```
1.    private final static HashSet<Character> vowels = new HashSet<>(Arrays.a
sList('a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'));
2.
3.    public String reverseVowels(String s) {
4.        int i = 0, j = s.length() - 1;
5.        char[] result = new char[s.length()];
6.        while (i <= j) {
7.            char ci = s.charAt(i);
8.            char cj = s.charAt(j);
9.            if (!vowels.contains(ci)) {
10.               result[i++] = ci;
11.           } else if (!vowels.contains(cj)) {
12.               result[j--] = cj;
13.           } else {
14.               result[i++] = cj;
15.               result[j--] = ci;
16.           }
17.       }
18.       return new String(result);
19.   }
```

回文字符串

680. Valid Palindrome II (Easy)

```
1.    Input: "abca"
```

```
2.    Output: True
3.    Explanation: You could delete the character 'c'.
```

题目描述：可以删除一个字符，判断是否能构成回文字符串。

```java
1.    public boolean validPalindrome(String s) {
2.        int i = -1, j = s.length();
3.        while (++i < --j) {
4.            if (s.charAt(i) != s.charAt(j)) {
5.                return isPalindrome(s, i, j - 1) || isPalindrome(s, i + 1,
      j);
6.            }
7.        }
8.        return true;
9.    }
10.
11.   private boolean isPalindrome(String s, int i, int j) {
12.        while (i < j) {
13.            if (s.charAt(i++) != s.charAt(j--)) {
14.                return false;
15.            }
16.        }
17.        return true;
18.    }
```

归并两个有序数组

88. Merge Sorted Array (Easy)

```
1.    Input:
2.    nums1 = [1,2,3,0,0,0], m = 3
3.    nums2 = [2,5,6],        n = 3
4.
5.    Output: [1,2,2,3,5,6]
```

题目描述：把归并结果存到第一个数组上。

需要从尾开始遍历，否则在 nums1 上归并得到的值会覆盖还未进行归并比较的值

```java
1.    public void merge(int[] nums1, int m, int[] nums2, int n) {
2.        int index1 = m - 1, index2 = n - 1;
```

```
 3.         int indexMerge = m + n - 1;
 4.         while (index1 >= 0 || index2 >= 0) {
 5.             if (index1 < 0) {
 6.                 nums1[indexMerge--] = nums2[index2--];
 7.             } else if (index2 < 0) {
 8.                 nums1[indexMerge--] = nums1[index1--];
 9.             } else if (nums1[index1] > nums2[index2]) {
10.                 nums1[indexMerge--] = nums1[index1--];
11.             } else {
12.                 nums1[indexMerge--] = nums2[index2--];
13.             }
14.         }
15.     }
```

判断链表是否存在环

141. Linked List Cycle (Easy)

使用双指针，一个指针每次移动一个节点，一个指针每次移动两个节点，如果存在环，那么这两个指针一定会相遇。

```
 1.     public boolean hasCycle(ListNode head) {
 2.         if (head == null) {
 3.             return false;
 4.         }
 5.         ListNode l1 = head, l2 = head.next;
 6.         while (l1 != null && l2 != null && l2.next != null) {
 7.             if (l1 == l2) {
 8.                 return true;
 9.             }
10.             l1 = l1.next;
11.             l2 = l2.next.next;
12.         }
13.         return false;
14.     }
```

最长子序列

524. Longest Word in Dictionary through Deleting (Medium)

```
 1.     Input:
```

```
2.    s = "abpcplea", d = ["ale","apple","monkey","plea"]
3.
4.    Output:
5.    "apple"
```

题目描述：删除 s 中的一些字符，使得它构成字符串列表 d 中的一个字符串，找出能构成的最长字符串。如果有多个相同长度的结果，返回按字典序排序的最大字符串。

```
1.    public String findLongestWord(String s, List<String> d) {
2.        String longestWord = "";
3.        for (String target : d) {
4.            int l1 = longestWord.length(), l2 = target.length();
5.            if (l1 > l2 || (l1 == l2 && longestWord.compareTo(target) < 0)) {
6.                continue;
7.            }
8.            if (isValid(s, target)) {
9.                longestWord = target;
10.           }
11.       }
12.       return longestWord;
13.   }
14.
15.   private boolean isValid(String s, String target) {
16.       int i = 0, j = 0;
17.       while (i < s.length() && j < target.length()) {
18.           if (s.charAt(i) == target.charAt(j)) {
19.               j++;
20.           }
21.           i++;
22.       }
23.       return j == target.length();
24.   }
```

排序

快速选择

一般用于求解 **Kth Element** 问题，可以在 O(N) 时间复杂度，O(1) 空间复杂度完成求解工作。

与快速排序一样，快速选择一般需要先打乱数组，否则最坏情况下时间复杂度为 $O(N^2)$。

堆排序

堆排序用于求解 **TopK Elements** 问题，通过维护一个大小为 K 的堆，堆中的元素就是 TopK Elements。当然它也可以用于求解 Kth Element 问题，堆顶元素就是 Kth Element。快速选择也可以求解 TopK Elements 问题，因为找到 Kth Element 之后，再遍历一次数组，所有小于等于 Kth Element 的元素都是 TopK Elements。可以看到，快速选择和堆排序都可以求解 Kth Element 和 TopK Elements 问题。

Kth Element

215. Kth Largest Element in an Array (Medium)

排序 ：时间复杂度 O(NlogN)，空间复杂度 O(1)

```
1.  public int findKthLargest(int[] nums, int k) {
2.      Arrays.sort(nums);
3.      return nums[nums.length - k];
4.  }
```

堆排序 ：时间复杂度 O(NlogK)，空间复杂度 O(K)。

```
1.  public int findKthLargest(int[] nums, int k) {
2.      PriorityQueue<Integer> pq = new PriorityQueue<>(); // 小顶堆
3.      for (int val : nums) {
4.          pq.add(val);
5.          if (pq.size() > k) // 维护堆的大小为 K
6.              pq.poll();
7.      }
8.      return pq.peek();
9.  }
```

快速选择 ：时间复杂度 O(N)，空间复杂度 O(1)

```
1.  public int findKthLargest(int[] nums, int k) {
2.      k = nums.length - k;
3.      int l = 0, h = nums.length - 1;
```

```
4.        while (l < h) {
5.            int j = partition(nums, l, h);
6.            if (j == k) {
7.                break;
8.            } else if (j < k) {
9.                l = j + 1;
10.           } else {
11.               h = j - 1;
12.           }
13.       }
14.       return nums[k];
15.   }
16.
17.   private int partition(int[] a, int l, int h) {
18.       int i = l, j = h + 1;
19.       while (true) {
20.           while (a[++i] < a[l] && i < h) ;
21.           while (a[--j] > a[l] && j > l) ;
22.           if (i >= j) {
23.               break;
24.           }
25.           swap(a, i, j);
26.       }
27.       swap(a, l, j);
28.       return j;
29.   }
30.
31.   private void swap(int[] a, int i, int j) {
32.       int t = a[i];
33.       a[i] = a[j];
34.       a[j] = t;
35.   }
```

桶排序

出现频率最多的 k 个数

347. Top K Frequent Elements (Medium)

```
1.    Given [1,1,1,2,2,3] and k = 2, return [1,2].
```

设置若干个桶，每个桶存储出现频率相同的数，并且桶的下标代表桶中数出现的频率，即第 i

个桶中存储的数出现的频率为 i。把数都放到桶之后，从后向前遍历桶，最先得到的 k 个数就是出现频率最多的的 k 个数。

```java
public List<Integer> topKFrequent(int[] nums, int k) {
    Map<Integer, Integer> frequencyForNum = new HashMap<>();
    for (int num : nums) {
        frequencyForNum.put(num, frequencyForNum.getOrDefault(num, 0) + 1);
    }
    List<Integer>[] buckets = new ArrayList[nums.length + 1];
    for (int key : frequencyForNum.keySet()) {
        int frequency = frequencyForNum.get(key);
        if (buckets[frequency] == null) {
            buckets[frequency] = new ArrayList<>();
        }
        buckets[frequency].add(key);
    }
    List<Integer> topK = new ArrayList<>();
    for (int i = buckets.length - 1; i >= 0 && topK.size() < k; i--) {
        if (buckets[i] != null) {
            topK.addAll(buckets[i]);
        }
    }
    return topK;
}
```

按照字符出现次数对字符串排序

451. Sort Characters By Frequency (Medium)

```
Input:
"tree"

Output:
"eert"

Explanation:
'e' appears twice while 'r' and 't' both appear once.
So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a
valid answer.
```

```java
public String frequencySort(String s) {
```

```
 2.        Map<Character, Integer> frequencyForNum = new HashMap<>();
 3.        for (char c : s.toCharArray())
 4.            frequencyForNum.put(c, frequencyForNum.getOrDefault(c, 0) + 1);
 5.
 6.        List<Character>[] frequencyBucket = new ArrayList[s.length() + 1];
 7.        for (char c : frequencyForNum.keySet()) {
 8.            int f = frequencyForNum.get(c);
 9.            if (frequencyBucket[f] == null) {
10.                frequencyBucket[f] = new ArrayList<>();
11.            }
12.            frequencyBucket[f].add(c);
13.        }
14.        StringBuilder str = new StringBuilder();
15.        for (int i = frequencyBucket.length - 1; i >= 0; i--) {
16.            if (frequencyBucket[i] == null) {
17.                continue;
18.            }
19.            for (char c : frequencyBucket[i]) {
20.                for (int j = 0; j < i; j++) {
21.                    str.append(c);
22.                }
23.            }
24.        }
25.        return str.toString();
26.    }
```

荷兰国旗问题

荷兰国旗包含三种颜色：红、白、蓝。有这三种颜色的球，算法的目标是将这三种球按颜色顺序正确地排列。

它其实是三向切分快速排序的一种变种，在三向切分快速排序中，每次切分都将数组分成三个区间：小于切分元素、等于切分元素、大于切分元素，而该算法是将数组分成三个区间：等于红色、等于白色、等于蓝色。

```
1.    Input: [2,0,2,1,1,0]
2.    Output: [0,0,1,1,2,2]
```

题目描述：只有 0/1/2 三种颜色。

```java
1.    public void sortColors(int[] nums) {
2.        int zero = -1, one = 0, two = nums.length;
3.        while (one < two) {
4.            if (nums[one] == 0) {
5.                swap(nums, ++zero, one++);
6.            } else if (nums[one] == 2) {
7.                swap(nums, --two, one);
8.            } else {
9.                ++one;
10.           }
11.       }
12.   }
13.
14.   private void swap(int[] nums, int i, int j) {
15.       int t = nums[i];
16.       nums[i] = nums[j];
17.       nums[j] = t;
18.   }
```

二分查找

正常实现

```java
1.    public int binarySearch(int[] nums, int key) {
2.        int l = 0, h = nums.length - 1;
3.        while (l <= h) {
4.            int m = l + (h - l) / 2;
5.            if (nums[m] == key) {
6.                return m;
7.            } else if (nums[m] > key) {
8.                h = m - 1;
9.            } else {
10.               l = m + 1;
11.           }
```

```
12.        }
13.      return -1;
14.  }
```

时间复杂度

二分查找也称为折半查找，每次都能将查找区间减半，这种折半特性的算法时间复杂度都为 O(logN)。

m 计算

有两种计算中值 m 的方式：

- m = (l + h) / 2
- m = l + (h - l) / 2

l + h 可能出现加法溢出，最好使用第二种方式。

返回值

循环退出时如果仍然没有查找到 key，那么表示查找失败。可以有两种返回值：

- -1：以一个错误码表示没有查找到 key
- l：将 key 插入到 nums 中的正确位置

变种

二分查找可以有很多变种，变种实现要注意边界值的判断。例如在一个有重复元素的数组中查找 key 的最左位置的实现如下：

```
1.  public int binarySearch(int[] nums, int key) {
2.      int l = 0, h = nums.length - 1;
3.      while (l < h) {
4.          int m = l + (h - l) / 2;
5.          if (nums[m] >= key) {
6.              h = m;
7.          } else {
8.              l = m + 1;
9.          }
10.     }
```

```
11.        return l;
12.    }
```

该实现和正常实现有以下不同：

- 循环条件为 l < h
- h 的赋值表达式为 h = m
- 最后返回 l 而不是 -1

在 nums[m] >= key 的情况下，可以推导出最左 key 位于 [l, m] 区间中，这是一个闭区间。h 的赋值表达式为 h = m，因为 m 位置也可能是解。

在 h 的赋值表达式为 h = mid 的情况下，如果循环条件为 l <= h，那么会出现循环无法退出的情况，因此循环条件只能是 l < h。以下演示了循环条件为 l <= h 时循环无法退出的情况：

```
1.    nums = {0, 1, 2}, key = 1
2.    l   m   h
3.    0   1   2  nums[m] >= key
4.    0   0   1  nums[m] < key
5.    1   1   1  nums[m] >= key
6.    1   1   1  nums[m] >= key
7.    ...
```

当循环体退出时，不表示没有查找到 key，因此最后返回的结果不应该为 -1。为了验证有没有查找到，需要在调用端判断一下返回位置上的值和 key 是否相等。

求开方

[69. Sqrt(x) (Easy)](#)

```
1.    Input: 4
2.    Output: 2
3.
4.    Input: 8
5.    Output: 2
6.    Explanation: The square root of 8 is 2.82842..., and since we want to
      return an integer, the decimal part will be truncated.
```

一个数 x 的开方 sqrt 一定在 0 \~ x 之间，并且满足 sqrt == x / sqrt。可以利用二分查找在

0 \~ x 之间查找 sqrt。

对于 x = 8，它的开方是 2.82842...，最后应该返回 2 而不是 3。在循环条件为 l <= h 并且循环退出时，h 总是比 l 小 1，也就是说 h = 2，l = 3，因此最后的返回值应该为 h 而不是 l。

```java
public int mySqrt(int x) {
    if (x <= 1) {
        return x;
    }
    int l = 1, h = x;
    while (l <= h) {
        int mid = l + (h - l) / 2;
        int sqrt = x / mid;
        if (sqrt == mid) {
            return mid;
        } else if (mid > sqrt) {
            h = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return h;
}
```

大于给定元素的最小元素

[744. Find Smallest Letter Greater Than Target (Easy)](#)

```
Input:
letters = ["c", "f", "j"]
target = "d"
Output: "f"

Input:
letters = ["c", "f", "j"]
target = "k"
Output: "c"
```

题目描述：给定一个有序的字符数组 letters 和一个字符 target，要求找出 letters 中大于 target 的最小字符，如果找不到就返回第 1 个字符。

```java
public char nextGreatestLetter(char[] letters, char target) {
    int n = letters.length;
    int l = 0, h = n - 1;
    while (l <= h) {
        int m = l + (h - l) / 2;
        if (letters[m] <= target) {
            l = m + 1;
        } else {
            h = m - 1;
        }
    }
    return l < n ? letters[l] : letters[0];
}
```

有序数组的 Single Element

540. Single Element in a Sorted Array (Medium)

```
Input: [1,1,2,3,3,4,4,8,8]
Output: 2
```

题目描述：一个有序数组只有一个数不出现两次，找出这个数。要求以 O(logN) 时间复杂度进行求解。

令 index 为 Single Element 在数组中的位置。如果 m 为偶数，并且 m + 1 < index，那么 nums[m] == nums[m + 1]；m + 1 >= index，那么 nums[m] != nums[m + 1]。

从上面的规律可以知道，如果 nums[m] == nums[m + 1]，那么 index 所在的数组位置为 [m + 2, h]，此时令 l = m + 2；如果 nums[m] != nums[m + 1]，那么 index 所在的数组位置为 [l, m]，此时令 h = m。

因为 h 的赋值表达式为 h = m，那么循环条件也就只能使用 l < h 这种形式。

```java
public int singleNonDuplicate(int[] nums) {
    int l = 0, h = nums.length - 1;
    while (l < h) {
        int m = l + (h - l) / 2;
        if (m % 2 == 1) {
            m--;    // 保证 l/h/m 都在偶数位，使得查找区间大小一直都是奇数
        }
```

```
8.          if (nums[m] == nums[m + 1]) {
9.              l = m + 2;
10.         } else {
11.             h = m;
12.         }
13.     }
14.     return nums[l];
15. }
```

第一个错误的版本

278. First Bad Version (Easy)

题目描述：给定一个元素 n 代表有 [1, 2, ..., n] 版本，可以调用 isBadVersion(int x) 知道某个版本是否错误，要求找到第一个错误的版本。

如果第 m 个版本出错，则表示第一个错误的版本在 [l, m] 之间，令 h = m；否则第一个错误的版本在 [m + 1, h] 之间，令 l = m + 1。

因为 h 的赋值表达式为 h = m，因此循环条件为 l < h。

```
1.  public int firstBadVersion(int n) {
2.      int l = 1, h = n;
3.      while (l < h) {
4.          int mid = l + (h - l) / 2;
5.          if (isBadVersion(mid)) {
6.              h = mid;
7.          } else {
8.              l = mid + 1;
9.          }
10.     }
11.     return l;
12. }
```

旋转数组的最小数字

153. Find Minimum in Rotated Sorted Array (Medium)

```
1.  Input: [3,4,5,1,2],
2.  Output: 1
```

```
1.   public int findMin(int[] nums) {
2.       int l = 0, h = nums.length - 1;
3.       while (l < h) {
4.           int m = l + (h - l) / 2;
5.           if (nums[m] <= nums[h]) {
6.               h = m;
7.           } else {
8.               l = m + 1;
9.           }
10.      }
11.      return nums[l];
12.  }
```

查找区间

34. Search for a Range (Medium)

```
1.   Input: nums = [5,7,7,8,8,10], target = 8
2.   Output: [3,4]
3.
4.   Input: nums = [5,7,7,8,8,10], target = 6
5.   Output: [-1,-1]
```

```
1.   public int[] searchRange(int[] nums, int target) {
2.       int first = binarySearch(nums, target);
3.       int last = binarySearch(nums, target + 1) - 1;
4.       if (first == nums.length || nums[first] != target) {
5.           return new int[]{-1, -1};
6.       } else {
7.           return new int[]{first, Math.max(first, last)};
8.       }
9.   }
10.
11.  private int binarySearch(int[] nums, int target) {
12.      int l = 0, h = nums.length; // 注意 h 的初始值
13.      while (l < h) {
14.          int m = l + (h - l) / 2;
15.          if (nums[m] >= target) {
16.              h = m;
17.          } else {
18.              l = m + 1;
19.          }
```
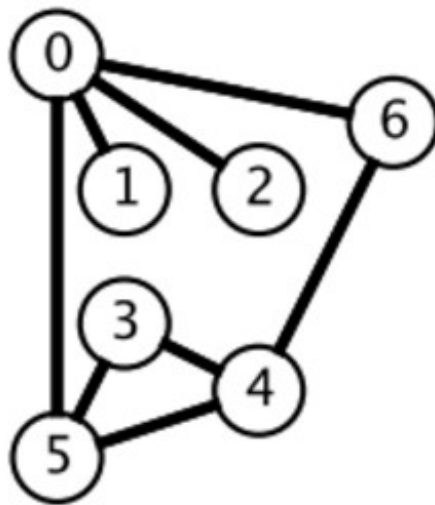
```
20.         }
21.     return l;
22.  }
```

搜索

深度优先搜索和广度优先搜索广泛运用于树和图中，但是它们的应用远远不止如此。

BFS

ii

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

1 表示可以经过某个位置，求解从 (0, 0) 位置到 (tr, tc) 位置的最短路径长度。

```
1.  public int minPathLength(int[][] grids, int tr, int tc) {
2.      final int[][] direction = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
3.      final int m = grids.length, n = grids[0].length;
4.      Queue<Pair<Integer, Integer>> queue = new LinkedList<>();
5.      queue.add(new Pair<>(0, 0));
6.      int pathLength = 0;
7.      while (!queue.isEmpty()) {
```

```
8.            int size = queue.size();
9.            pathLength++;
10.           while (size-- > 0) {
11.               Pair<Integer, Integer> cur = queue.poll();
12.               for (int[] d : direction) {
13.                   int nr = cur.getKey() + d[0], nc = cur.getValue() + d[1];
14.                   Pair<Integer, Integer> next = new Pair<>(nr, nc);
15.                   if (next.getKey() < 0 || next.getValue() >= m
16.                           || next.getKey() < 0 || next.getValue() >= n) {
17.
18.                       continue;
19.                   }
20.                   grids[next.getKey()][next.getValue()] = 0; // 标记
21.                   if (next.getKey() == tr && next.getValue() == tc) {
22.                       return pathLength;
23.                   }
24.                   queue.add(next);
25.               }
26.           }
27.       }
28.       return -1;
29.   }
```

组成整数的最小平方数数量

279. Perfect Squares (Medium)

```
1.    For example, given n = 12, return 3 because 12 = 4 + 4 + 4; given n =
      13, return 2 because 13 = 4 + 9.
```

可以将每个整数看成图中的一个节点，如果两个整数之差为一个平方数，那么这两个整数所在的节点就有一条边。

要求解最小的平方数数量，就是求解从节点 n 到节点 0 的最短路径。

本题也可以用动态规划求解，在之后动态规划部分中会再次出现。

```
1.    public int numSquares(int n) {
2.        List<Integer> squares = generateSquares(n);
3.        Queue<Integer> queue = new LinkedList<>();
```

```java
        boolean[] marked = new boolean[n + 1];
        queue.add(n);
        marked[n] = true;
        int level = 0;
        while (!queue.isEmpty()) {
            int size = queue.size();
            level++;
            while (size-- > 0) {
                int cur = queue.poll();
                for (int s : squares) {
                    int next = cur - s;
                    if (next < 0) {
                        break;
                    }
                    if (next == 0) {
                        return level;
                    }
                    if (marked[next]) {
                        continue;
                    }
                    marked[next] = true;
                    queue.add(cur - s);
                }
            }
        }
        return n;
    }

    /**
     * 生成小于 n 的平方数序列
     * @return 1,4,9,...
     */
    private List<Integer> generateSquares(int n) {
        List<Integer> squares = new ArrayList<>();
        int square = 1;
        int diff = 3;
        while (square <= n) {
            squares.add(square);
            square += diff;
            diff += 2;
        }
        return squares;
    }
```

最短单词路径

127. Word Ladder (Medium)

```
1.    Input:
2.    beginWord = "hit",
3.    endWord = "cog",
4.    wordList = ["hot","dot","dog","lot","log","cog"]
5.
6.    Output: 5
7.
8.    Explanation: As one shortest transformation is "hit" -> "hot" -> "dot"
      -> "dog" -> "cog",
9.    return its length 5.
```

```
1.    Input:
2.    beginWord = "hit"
3.    endWord = "cog"
4.    wordList = ["hot","dot","dog","lot","log"]
5.
6.    Output: 0
7.
8.    Explanation: The endWord "cog" is not in wordList, therefore no possib
      le transformation.
```

找出一条从 beginWord 到 endWord 的最短路径，每次移动规定为改变一个字符，并且改变之后的字符串必须在 wordList 中。

```java
1.    public int ladderLength(String beginWord, String endWord, List<String>
      wordList) {
2.        wordList.add(beginWord);
3.        int N = wordList.size();
4.        int start = N - 1;
5.        int end = 0;
6.        while (end < N && !wordList.get(end).equals(endWord)) {
7.            end++;
8.        }
9.        if (end == N) {
10.           return 0;
11.       }
12.       List<Integer>[] graphic = buildGraphic(wordList);
13.       return getShortestPath(graphic, start, end);
```

```java
14.    }
15.
16.    private List<Integer>[] buildGraphic(List<String> wordList) {
17.        int N = wordList.size();
18.        List<Integer>[] graphic = new List[N];
19.        for (int i = 0; i < N; i++) {
20.            graphic[i] = new ArrayList<>();
21.            for (int j = 0; j < N; j++) {
22.                if (isConnect(wordList.get(i), wordList.get(j))) {
23.                    graphic[i].add(j);
24.                }
25.            }
26.        }
27.        return graphic;
28.    }
29.
30.    private boolean isConnect(String s1, String s2) {
31.        int diffCnt = 0;
32.        for (int i = 0; i < s1.length() && diffCnt <= 1; i++) {
33.            if (s1.charAt(i) != s2.charAt(i)) {
34.                diffCnt++;
35.            }
36.        }
37.        return diffCnt == 1;
38.    }
39.
40.    private int getShortestPath(List<Integer>[] graphic, int start, int
       end) {
41.        Queue<Integer> queue = new LinkedList<>();
42.        boolean[] marked = new boolean[graphic.length];
43.        queue.add(start);
44.        marked[start] = true;
45.        int path = 1;
46.        while (!queue.isEmpty()) {
47.            int size = queue.size();
48.            path++;
49.            while (size-- > 0) {
50.                int cur = queue.poll();
51.                for (int next : graphic[cur]) {
52.                    if (next == end) {
53.                        return path;
54.                    }
55.                    if (marked[next]) {
56.                        continue;
57.                    }
```

```
58.                    marked[next] = true;
59.                    queue.add(next);
60.                }
61.            }
62.        }
63.        return 0;
64.    }
```
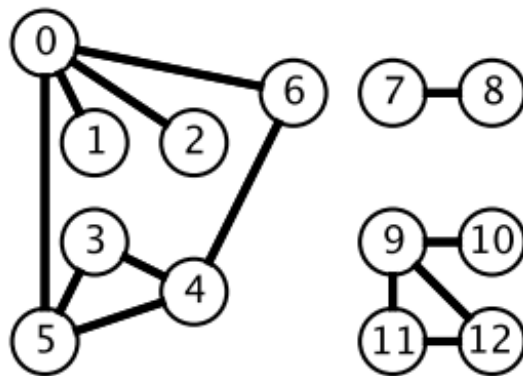
DFS


```
1.    [[0,0,1,0,0,0,0,1,0,0,0,0,0],
2.     [0,0,0,0,0,0,0,1,1,1,0,0,0],
3.     [0,1,1,0,1,0,0,0,0,0,0,0,0],
4.     [0,1,0,0,1,1,0,0,1,0,1,0,0],
5.     [0,1,0,0,1,1,0,0,1,1,1,0,0],
6.     [0,0,0,0,0,0,0,0,0,0,1,0,0],
7.     [0,0,0,0,0,0,0,1,1,1,0,0,0],
8.     [0,0,0,0,0,0,0,1,1,0,0,0,0]]
```

```
1.    private int m, n;
2.    private int[][] direction = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
3.
4.    public int maxAreaOfIsland(int[][] grid) {
5.        if (grid == null || grid.length == 0) {
6.            return 0;
7.        }
8.        m = grid.length;
9.        n = grid[0].length;
10.       int maxArea = 0;
11.       for (int i = 0; i < m; i++) {
12.           for (int j = 0; j < n; j++) {
```

```
13.              maxArea = Math.max(maxArea, dfs(grid, i, j));
14.          }
15.      }
16.      return maxArea;
17.  }
18.
19.  private int dfs(int[][] grid, int r, int c) {
20.      if (r < 0 || r >= m || c < 0 || c >= n || grid[r][c] == 0) {
21.          return 0;
22.      }
23.      grid[r][c] = 0;
24.      int area = 1;
25.      for (int[] d : direction) {
26.          area += dfs(grid, r + d[0], c + d[1]);
27.      }
28.      return area;
29.  }
```

矩阵中的连通分量数目

200. Number of Islands (Medium)

```
1.   Input:
2.   11000
3.   11000
4.   00100
5.   00011
6.
7.   Output: 3
```

可以将矩阵表示看成一张有向图。

```
1.   private int m, n;
2.   private int[][] direction = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
3.
4.   public int numIslands(char[][] grid) {
5.       if (grid == null || grid.length == 0) {
6.           return 0;
7.       }
8.       m = grid.length;
9.       n = grid[0].length;
10.      int islandsNum = 0;
```

```
11.        for (int i = 0; i < m; i++) {
12.            for (int j = 0; j < n; j++) {
13.                if (grid[i][j] != '0') {
14.                    dfs(grid, i, j);
15.                    islandsNum++;
16.                }
17.            }
18.        }
19.        return islandsNum;
20.    }
21.
22.    private void dfs(char[][] grid, int i, int j) {
23.        if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] == '0') {
24.            return;
25.        }
26.        grid[i][j] = '0';
27.        for (int[] d : direction) {
28.            dfs(grid, i + d[0], j + d[1]);
29.        }
30.    }
```

好友关系的连通分量数目

547. Friend Circles (Medium)

```
1.    Input:
2.    [[1,1,0],
3.     [1,1,0],
4.     [0,0,1]]
5.    Output: 2
6.    Explanation:The 0th and 1st students are direct friends, so they are i
      n a friend circle.
7.    The 2nd student himself is in a friend circle. So return 2.
```

好友关系可以看成是一个无向图，例如第 0 个人与第 1 个人是好友，那么 M[0][1] 和 M[1][0] 的值都为 1。

```
1.    private int n;
2.
3.    public int findCircleNum(int[][] M) {
4.        n = M.length;
5.        int circleNum = 0;
```

```
6.          boolean[] hasVisited = new boolean[n];
7.          for (int i = 0; i < n; i++) {
8.              if (!hasVisited[i]) {
9.                  dfs(M, i, hasVisited);
10.                 circleNum++;
11.             }
12.         }
13.         return circleNum;
14.     }
15.
16.     private void dfs(int[][] M, int i, boolean[] hasVisited) {
17.         hasVisited[i] = true;
18.         for (int k = 0; k < n; k++) {
19.             if (M[i][k] == 1 && !hasVisited[k]) {
20.                 dfs(M, k, hasVisited);
21.             }
22.         }
23.     }
```

填充封闭区域

130. Surrounded Regions (Medium)

```
1.    For example,
2.    X X X X
3.    X O O X
4.    X X O X
5.    X O X X
6.
7.    After running your function, the board should be:
8.    X X X X
9.    X X X X
10.   X X X X
11.   X O X X
```

使被 'X' 包围的 'O' 转换为 'X'。

先填充最外侧，剩下的就是里侧了。

```
1.    private int[][] direction = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
2.    private int m, n;
3.
```

```
 4.    public void solve(char[][] board) {
 5.        if (board == null || board.length == 0) {
 6.            return;
 7.        }
 8.
 9.        m = board.length;
10.        n = board[0].length;
11.
12.        for (int i = 0; i < m; i++) {
13.            dfs(board, i, 0);
14.            dfs(board, i, n - 1);
15.        }
16.        for (int i = 0; i < n; i++) {
17.            dfs(board, 0, i);
18.            dfs(board, m - 1, i);
19.        }
20.
21.        for (int i = 0; i < m; i++) {
22.            for (int j = 0; j < n; j++) {
23.                if (board[i][j] == 'T') {
24.                    board[i][j] = 'O';
25.                } else if (board[i][j] == 'O') {
26.                    board[i][j] = 'X';
27.                }
28.            }
29.        }
30.    }
31.
32.    private void dfs(char[][] board, int r, int c) {
33.        if (r < 0 || r >= m || c < 0 || c >= n || board[r][c] != 'O') {
34.            return;
35.        }
36.        board[r][c] = 'T';
37.        for (int[] d : direction) {
38.            dfs(board, r + d[0], c + d[1]);
39.        }
40.    }
```

能到达的太平洋和大西洋的区域

417. Pacific Atlantic Water Flow (Medium)

```
 1.    Given the following 5x5 matrix:
 2.
```

```
  3.      Pacific ~   ~   ~   ~   ~
  4.           ~  1   2   2   3  (5)  *
  5.           ~  3   2   3  (4) (4)  *
  6.           ~  2   4  (5)  3   1   *
  7.           ~ (6) (7)  1   4   5   *
  8.           ~ (5)  1   1   2   4   *
  9.              *   *   *   *   * Atlantic
 10.
 11.   Return:
 12.   [[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]] (positions
        with parentheses in above matrix).
```

左边和上边是太平洋，右边和下边是大西洋，内部的数字代表海拔，海拔高的地方的水能够流到低的地方，求解水能够流到太平洋和大西洋的所有位置。

```java
  1.
  2.   private int m, n;
  3.   private int[][] matrix;
  4.   private int[][] direction = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
  5.
  6.   public List<int[]> pacificAtlantic(int[][] matrix) {
  7.       List<int[]> ret = new ArrayList<>();
  8.       if (matrix == null || matrix.length == 0) {
  9.           return ret;
 10.       }
 11.
 12.       m = matrix.length;
 13.       n = matrix[0].length;
 14.       this.matrix = matrix;
 15.       boolean[][] canReachP = new boolean[m][n];
 16.       boolean[][] canReachA = new boolean[m][n];
 17.
 18.       for (int i = 0; i < m; i++) {
 19.           dfs(i, 0, canReachP);
 20.           dfs(i, n - 1, canReachA);
 21.       }
 22.       for (int i = 0; i < n; i++) {
 23.           dfs(0, i, canReachP);
 24.           dfs(m - 1, i, canReachA);
 25.       }
 26.
 27.       for (int i = 0; i < m; i++) {
 28.           for (int j = 0; j < n; j++) {
 29.               if (canReachP[i][j] && canReachA[i][j]) {
```

```
30.                ret.add(new int[]{i, j});
31.            }
32.        }
33.    }
34.
35.    return ret;
36. }
37.
38. private void dfs(int r, int c, boolean[][] canReach) {
39.     if (canReach[r][c]) {
40.         return;
41.     }
42.     canReach[r][c] = true;
43.     for (int[] d : direction) {
44.         int nextR = d[0] + r;
45.         int nextC = d[1] + c;
46.         if (nextR < 0 || nextR >= m || nextC < 0 || nextC >= n
47.                 || matrix[r][c] > matrix[nextR][nextC]) {
48.
49.             continue;
50.         }
51.         dfs(nextR, nextC, canReach);
52.     }
53. }
```

Backtracking

Backtracking（回溯）属于 DFS。

- 普通 DFS 主要用在 **可达性问题** ，这种问题只需要执行到特点的位置然后返回即可。
- 而 Backtracking 主要用于求解 **排列组合** 问题，例如有 { 'a','b','c' } 三个字符，求解所有由这三个字符排列得到的字符串，这种问题在执行到特定的位置返回之后还会继续执行求解过程。

因为 Backtracking 不是立即就返回，而要继续求解，因此在程序实现时，需要注意对元素的标记问题：

- 在访问一个新元素进入新的递归调用时，需要将新元素标记为已经访问，这样才能在继续递归调用时不用重复访问该元素；
- 但是在递归返回时，需要将元素标记为未访问，因为只需要保证在一个递归链中不同时访

问一个元素，可以访问已经访问过但是不在当前递归链中的元素。

数字键盘组合

17. Letter Combinations of a Phone Number (Medium)


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

```java
1.
2.  private static final String[] KEYS = {"", "", "abc", "def", "ghi", "jkl
    ", "mno", "pqrs", "tuv", "wxyz"};
3.
4.  public List<String> letterCombinations(String digits) {
5.      List<String> combinations = new ArrayList<>();
6.      if (digits == null || digits.length() == 0) {
7.          return combinations;
8.      }
9.      doCombination(new StringBuilder(), combinations, digits);
10.     return combinations;
11.  }
12.
13. private void doCombination(StringBuilder prefix, List<String>
    combinations, final String digits) {
14.     if (prefix.length() == digits.length()) {
15.         combinations.add(prefix.toString());
16.         return;
17.     }
18.     int curDigits = digits.charAt(prefix.length()) - '0';
19.     String letters = KEYS[curDigits];
20.     for (char c : letters.toCharArray()) {
21.         prefix.append(c);                              // 添加
```

```
22.          doCombination(prefix, combinations, digits);
23.          prefix.deleteCharAt(prefix.length() - 1); // 删除
24.      }
25.  }
```

IP 地址划分

93. Restore IP Addresses(Medium)

```
1.   Given "25525511135",
2.   return ["255.255.11.135", "255.255.111.35"].
```

```
1.   public List<String> restoreIpAddresses(String s) {
2.       List<String> addresses = new ArrayList<>();
3.       StringBuilder tempAddress = new StringBuilder();
4.       doRestore(0, tempAddress, addresses, s);
5.       return addresses;
6.   }
7.
8.   private void doRestore(int k, StringBuilder tempAddress, List<String> addresses, String s) {
9.       if (k == 4 || s.length() == 0) {
10.          if (k == 4 && s.length() == 0) {
11.              addresses.add(tempAddress.toString());
12.          }
13.          return;
14.      }
15.      for (int i = 0; i < s.length() && i <= 2; i++) {
16.          if (i != 0 && s.charAt(0) == '0') {
17.              break;
18.          }
19.          String part = s.substring(0, i + 1);
20.          if (Integer.valueOf(part) <= 255) {
21.              if (tempAddress.length() != 0) {
22.                  part = "." + part;
23.              }
24.              tempAddress.append(part);
25.              doRestore(k + 1, tempAddress, addresses, s.substring(i + 1));
26.              tempAddress.delete(tempAddress.length() - part.length(), tempAddress.length());
27.          }
28.      }
```

```
29.    }
```

在矩阵中寻找字符串

79. Word Search (Medium)

```
1.   For example,
2.   Given board =
3.   [
4.     ['A','B','C','E'],
5.     ['S','F','C','S'],
6.     ['A','D','E','E']
7.   ]
8.   word = "ABCCED", -> returns true,
9.   word = "SEE", -> returns true,
10.  word = "ABCB", -> returns false.
```

```
1.   private final static int[][] direction = {{1, 0}, {-1, 0}, {0, 1}, {0,
     -1}};
2.   private int m;
3.   private int n;
4.
5.   public boolean exist(char[][] board, String word) {
6.       if (word == null || word.length() == 0) {
7.           return true;
8.       }
9.       if (board == null || board.length == 0 || board[0].length == 0) {
10.          return false;
11.      }
12.
13.      m = board.length;
14.      n = board[0].length;
15.      boolean[][] hasVisited = new boolean[m][n];
16.
17.      for (int r = 0; r < m; r++) {
18.          for (int c = 0; c < n; c++) {
19.              if (backtracking(0, r, c, hasVisited, board, word)) {
20.                  return true;
21.              }
22.          }
23.      }
24.
25.      return false;
```

```
26.     }
27.
28.   private boolean backtracking(int curLen, int r, int c, boolean[][]
      visited, final char[][] board, final String word) {
29.       if (curLen == word.length()) {
30.           return true;
31.       }
32.       if (r < 0 || r >= m || c < 0 || c >= n
33.               || board[r][c] != word.charAt(curLen) || visited[r][c]) {
34.
35.           return false;
36.       }
37.
38.       visited[r][c] = true;
39.
40.       for (int[] d : direction) {
41.           if (backtracking(curLen + 1, r + d[0], c + d[1], visited, board
      , word)) {
42.               return true;
43.           }
44.       }
45.
46.       visited[r][c] = false;
47.
48.       return false;
49.   }
```

输出二叉树中所有从根到叶子的路径

257. Binary Tree Paths (Easy)

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

```
1.   ["1->2->5", "1->3"]
```

```
1.
2.   public List<String> binaryTreePaths(TreeNode root) {
3.       List<String> paths = new ArrayList<>();
```

```
4.      if (root == null) {
5.          return paths;
6.      }
7.      List<Integer> values = new ArrayList<>();
8.      backtracking(root, values, paths);
9.      return paths;
10.  }
11.
12.  private void backtracking(TreeNode node, List<Integer> values, List<String> paths) {
13.      if (node == null) {
14.          return;
15.      }
16.      values.add(node.val);
17.      if (isLeaf(node)) {
18.          paths.add(buildPath(values));
19.      } else {
20.          backtracking(node.left, values, paths);
21.          backtracking(node.right, values, paths);
22.      }
23.      values.remove(values.size() - 1);
24.  }
25.
26.  private boolean isLeaf(TreeNode node) {
27.      return node.left == null && node.right == null;
28.  }
29.
30.  private String buildPath(List<Integer> values) {
31.      StringBuilder str = new StringBuilder();
32.      for (int i = 0; i < values.size(); i++) {
33.          str.append(values.get(i));
34.          if (i != values.size() - 1) {
35.              str.append("->");
36.          }
37.      }
38.      return str.toString();
39.  }
```

排列

46. Permutations (Medium)

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

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

```java
1.   public List<List<Integer>> permute(int[] nums) {
2.       List<List<Integer>> permutes = new ArrayList<>();
3.       List<Integer> permuteList = new ArrayList<>();
4.       boolean[] hasVisited = new boolean[nums.length];
5.       backtracking(permuteList, permutes, hasVisited, nums);
6.       return permutes;
7.   }
8.
9.   private void backtracking(List<Integer> permuteList, List<List<Integer
     >> permutes, boolean[] visited, final int[] nums) {
10.      if (permuteList.size() == nums.length) {
11.          permutes.add(new ArrayList<>(permuteList)); // 重新构造一个 List
12.          return;
13.      }
14.      for (int i = 0; i < visited.length; i++) {
15.          if (visited[i]) {
16.              continue;
17.          }
18.          visited[i] = true;
19.          permuteList.add(nums[i]);
20.          backtracking(permuteList, permutes, visited, nums);
21.          permuteList.remove(permuteList.size() - 1);
22.          visited[i] = false;
23.      }
24.  }
```

含有相同元素求排列

47. Permutations II (Medium)

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

数组元素可能含有相同的元素，进行排列时就有可能出现重复的排列，要求重复的排列只返回

一个。

在实现上，和 Permutations 不同的是要先排序，然后在添加一个元素时，判断这个元素是否等于前一个元素，如果等于，并且前一个元素还未访问，那么就跳过这个元素。

```java
public List<List<Integer>> permuteUnique(int[] nums) {
    List<List<Integer>> permutes = new ArrayList<>();
    List<Integer> permuteList = new ArrayList<>();
    Arrays.sort(nums);  // 排序
    boolean[] hasVisited = new boolean[nums.length];
    backtracking(permuteList, permutes, hasVisited, nums);
    return permutes;
}

private void backtracking(List<Integer> permuteList, List<List<Integer>> permutes, boolean[] visited, final int[] nums) {
    if (permuteList.size() == nums.length) {
        permutes.add(new ArrayList<>(permuteList));
        return;
    }

    for (int i = 0; i < visited.length; i++) {
        if (i != 0 && nums[i] == nums[i - 1] && !visited[i - 1]) {
            continue;  // 防止重复
        }
        if (visited[i]){
            continue;
        }
        visited[i] = true;
        permuteList.add(nums[i]);
        backtracking(permuteList, permutes, visited, nums);
        permuteList.remove(permuteList.size() - 1);
        visited[i] = false;
    }
}
```

组合

77. Combinations (Medium)

```
If n = 4 and k = 2, a solution is:
[
```

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

```java
1.    public List<List<Integer>> combine(int n, int k) {
2.        List<List<Integer>> combinations = new ArrayList<>();
3.        List<Integer> combineList = new ArrayList<>();
4.        backtracking(combineList, combinations, 1, k, n);
5.        return combinations;
6.    }
7.
8.    private void backtracking(List<Integer> combineList, List<List<Integer>> combinations, int start, int k, final int n) {
9.        if (k == 0) {
10.            combinations.add(new ArrayList<>(combineList));
11.            return;
12.        }
13.        for (int i = start; i <= n - k + 1; i++) {  // 剪枝
14.            combineList.add(i);
15.            backtracking(combineList, combinations, i + 1, k - 1, n);
16.            combineList.remove(combineList.size() - 1);
17.        }
18.    }
```

组合求和

39. Combination Sum (Medium)

```
1.    given candidate set [2, 3, 6, 7] and target 7,
2.    A solution set is:
3.    [[7],[2, 2, 3]]
```

```java
1.    public List<List<Integer>> combinationSum(int[] candidates, int target) {
2.        List<List<Integer>> combinations = new ArrayList<>();
3.        backtracking(new ArrayList<>(), combinations, 0, target, candidates);
4.        return combinations;
```

```
5.     }
6.
7.     private void backtracking(List<Integer> tempCombination, List<List<Int
eger>> combinations,
8.                               int start, int target, final int[] candidates)
{
9.
10.        if (target == 0) {
11.            combinations.add(new ArrayList<>(tempCombination));
12.            return;
13.        }
14.        for (int i = start; i < candidates.length; i++) {
15.            if (candidates[i] <= target) {
16.                tempCombination.add(candidates[i]);
17.                backtracking(tempCombination, combinations, i, target - can
didates[i], candidates);
18.                tempCombination.remove(tempCombination.size() - 1);
19.            }
20.        }
21.    }
```

含有相同元素的求组合求和

40. Combination Sum II (Medium)

```
1.  For example, given candidate set [10, 1, 2, 7, 6, 1, 5] and target 8,
2.  A solution set is:
3.  [
4.    [1, 7],
5.    [1, 2, 5],
6.    [2, 6],
7.    [1, 1, 6]
8.  ]
```

```
1.  public List<List<Integer>> combinationSum2(int[] candidates, int
target) {
2.      List<List<Integer>> combinations = new ArrayList<>();
3.      Arrays.sort(candidates);
4.      backtracking(new ArrayList<>(), combinations, new
boolean[candidates.length], 0, target, candidates);
5.      return combinations;
6.  }
7.
```

```
 8.    private void backtracking(List<Integer> tempCombination, List<List<Int
       eger>> combinations,
 9.                             boolean[] hasVisited, int start, int target, f
       inal int[] candidates) {
10.
11.        if (target == 0) {
12.            combinations.add(new ArrayList<>(tempCombination));
13.            return;
14.        }
15.        for (int i = start; i < candidates.length; i++) {
16.            if (i != 0 && candidates[i] == candidates[i - 1] && !hasVisited
       [i - 1]) {
17.                continue;
18.            }
19.            if (candidates[i] <= target) {
20.                tempCombination.add(candidates[i]);
21.                hasVisited[i] = true;
22.                backtracking(tempCombination, combinations, hasVisited, i +
       1, target - candidates[i], candidates);
23.                hasVisited[i] = false;
24.                tempCombination.remove(tempCombination.size() - 1);
25.            }
26.        }
27.    }
```

1-9 数字的组合求和

216. Combination Sum III (Medium)

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

从 1-9 数字中选出 k 个数不重复的数，使得它们的和为 n。

```
1.    public List<List<Integer>> combinationSum3(int k, int n) {
2.        List<List<Integer>> combinations = new ArrayList<>();
3.        List<Integer> path = new ArrayList<>();
4.        backtracking(k, n, 1, path, combinations);
5.        return combinations;
```

```
 6.        }
 7.
 8.    private void backtracking(int k, int n, int start,
 9.                              List<Integer> tempCombination,
       List<List<Integer>> combinations) {
10.
11.        if (k == 0 && n == 0) {
12.            combinations.add(new ArrayList<>(tempCombination));
13.            return;
14.        }
15.        if (k == 0 || n == 0) {
16.            return;
17.        }
18.        for (int i = start; i <= 9; i++) {
19.            tempCombination.add(i);
20.            backtracking(k - 1, n - i, i + 1, tempCombination, combinations
       );
21.            tempCombination.remove(tempCombination.size() - 1);
22.        }
23.    }
```

子集

78. Subsets (Medium)

找出集合的所有子集，子集不能重复，[1, 2] 和 [2, 1] 这种子集算重复

```
 1.    public List<List<Integer>> subsets(int[] nums) {
 2.        List<List<Integer>> subsets = new ArrayList<>();
 3.        List<Integer> tempSubset = new ArrayList<>();
 4.        for (int size = 0; size <= nums.length; size++) {
 5.            backtracking(0, tempSubset, subsets, size, nums); // 不同的子集大
       小
 6.        }
 7.        return subsets;
 8.    }
 9.
10.    private void backtracking(int start, List<Integer> tempSubset,
       List<List<Integer>> subsets,
11.                              final int size, final int[] nums) {
12.
13.        if (tempSubset.size() == size) {
14.            subsets.add(new ArrayList<>(tempSubset));
```

```
15.          return;
16.       }
17.       for (int i = start; i < nums.length; i++) {
18.           tempSubset.add(nums[i]);
19.           backtracking(i + 1, tempSubset, subsets, size, nums);
20.           tempSubset.remove(tempSubset.size() - 1);
21.       }
22.    }
```

含有相同元素求子集

90. Subsets II (Medium)

```
1.    For example,
2.    If nums = [1,2,2], a solution is:
3.
4.    [
5.      [2],
6.      [1],
7.      [1,2,2],
8.      [2,2],
9.      [1,2],
10.      []
11.    ]
```

```
1.    public List<List<Integer>> subsetsWithDup(int[] nums) {
2.        Arrays.sort(nums);
3.        List<List<Integer>> subsets = new ArrayList<>();
4.        List<Integer> tempSubset = new ArrayList<>();
5.        boolean[] hasVisited = new boolean[nums.length];
6.        for (int size = 0; size <= nums.length; size++) {
7.            backtracking(0, tempSubset, subsets, hasVisited, size, nums); /
      / 不同的子集大小
8.        }
9.        return subsets;
10.    }
11.
12.    private void backtracking(int start, List<Integer> tempSubset,
      List<List<Integer>> subsets, boolean[] hasVisited,
13.                              final int size, final int[] nums) {
14.
15.        if (tempSubset.size() == size) {
16.            subsets.add(new ArrayList<>(tempSubset));
```

```
17.            return;
18.        }
19.        for (int i = start; i < nums.length; i++) {
20.            if (i != 0 && nums[i] == nums[i - 1] && !hasVisited[i - 1]) {
21.                continue;
22.            }
23.            tempSubset.add(nums[i]);
24.            hasVisited[i] = true;
25.            backtracking(i + 1, tempSubset, subsets, hasVisited, size, nums
    );
26.            hasVisited[i] = false;
27.            tempSubset.remove(tempSubset.size() - 1);
28.        }
29.    }
```

分割字符串使得每个部分都是回文数

131. Palindrome Partitioning (Medium)

```
1.    For example, given s = "aab",
2.    Return
3.
4.    [
5.      ["aa","b"],
6.      ["a","a","b"]
7.    ]
```

```
1.    public List<List<String>> partition(String s) {
2.        List<List<String>> partitions = new ArrayList<>();
3.        List<String> tempPartition = new ArrayList<>();
4.        doPartition(s, partitions, tempPartition);
5.        return partitions;
6.    }
7.
8.    private void doPartition(String s, List<List<String>> partitions, List
    <String> tempPartition) {
9.        if (s.length() == 0) {
10.            partitions.add(new ArrayList<>(tempPartition));
11.            return;
12.        }
13.        for (int i = 0; i < s.length(); i++) {
14.            if (isPalindrome(s, 0, i)) {
15.                tempPartition.add(s.substring(0, i + 1));
```

```
16.              doPartition(s.substring(i + 1), partitions, tempPartition);
17.              tempPartition.remove(tempPartition.size() - 1);
18.          }
19.      }
20.  }
21.
22.  private boolean isPalindrome(String s, int begin, int end) {
23.      while (begin < end) {
24.          if (s.charAt(begin++) != s.charAt(end--)) {
25.              return false;
26.          }
27.      }
28.      return true;
29.  }
```

数独

37. Sudoku Solver (Hard)


```
1.  private boolean[][] rowsUsed = new boolean[9][10];
2.  private boolean[][] colsUsed = new boolean[9][10];
3.  private boolean[][] cubesUsed = new boolean[9][10];
4.  private char[][] board;
5.
6.  public void solveSudoku(char[][] board) {
7.      this.board = board;
```

```java
8.        for (int i = 0; i < 9; i++)
9.            for (int j = 0; j < 9; j++) {
10.                if (board[i][j] == '.') {
11.                    continue;
12.                }
13.                int num = board[i][j] - '0';
14.                rowsUsed[i][num] = true;
15.                colsUsed[j][num] = true;
16.                cubesUsed[cubeNum(i, j)][num] = true;
17.            }
18.
19.        for (int i = 0; i < 9; i++) {
20.            for (int j = 0; j < 9; j++) {
21.                backtracking(i, j);
22.            }
23.        }
24.    }
25.
26.    private boolean backtracking(int row, int col) {
27.        while (row < 9 && board[row][col] != '.') {
28.            row = col == 8 ? row + 1 : row;
29.            col = col == 8 ? 0 : col + 1;
30.        }
31.        if (row == 9) {
32.            return true;
33.        }
34.        for (int num = 1; num <= 9; num++) {
35.            if (rowsUsed[row][num] || colsUsed[col][num] || cubesUsed[cubeN
    um(row, col)][num]) {
36.                continue;
37.            }
38.            rowsUsed[row][num] = colsUsed[col][num] = cubesUsed[cubeNum(row
    , col)][num] = true;
39.            board[row][col] = (char) (num + '0');
40.            if (backtracking(row, col)) {
41.                return true;
42.            }
43.            board[row][col] = '.';
44.            rowsUsed[row][num] = colsUsed[col][num] = cubesUsed[cubeNum(row
    , col)][num] = false;
45.        }
46.        return false;
47.    }
48.
49.    private int cubeNum(int i, int j) {
```
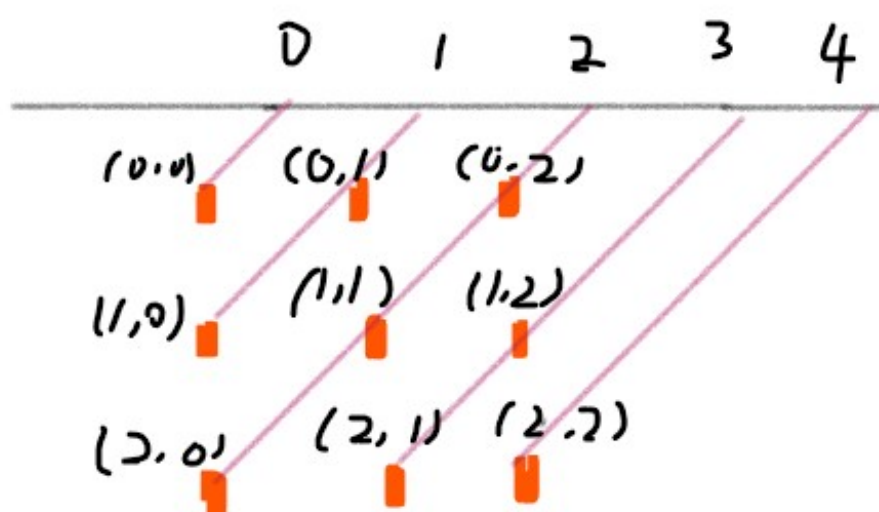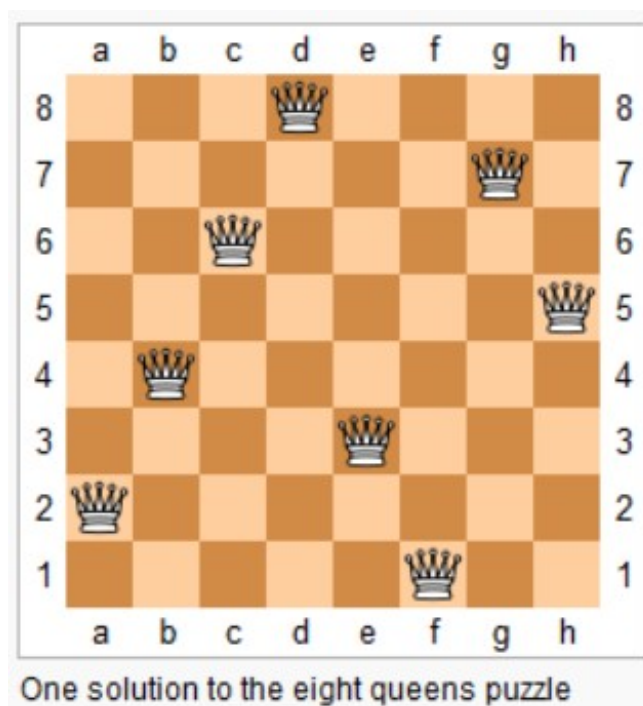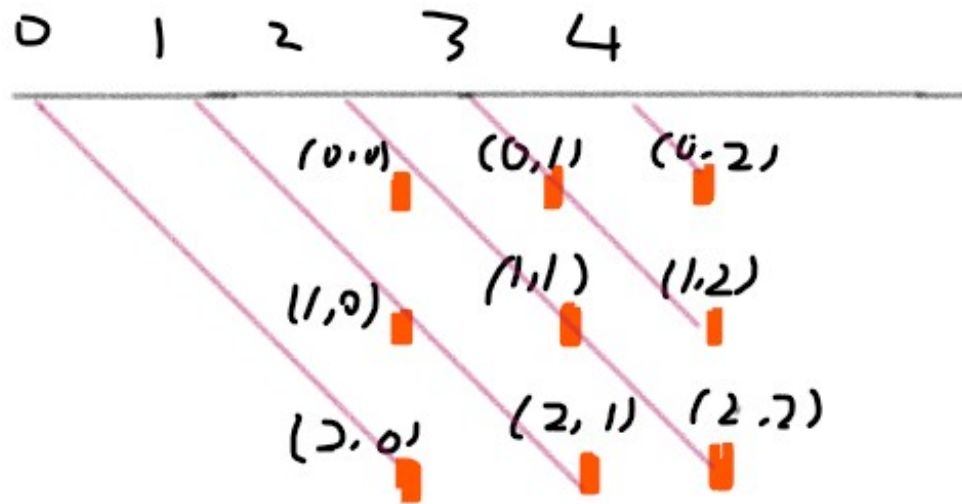
```
50.        int r = i / 3;
51.        int c = j / 3;
52.        return r * 3 + c;
53.    }
```

N 皇后

51. N-Queens (Hard)

One solution to the eight queens puzzle

```java
1.   private List<List<String>> solutions;
2.   private char[][] nQueens;
3.   private boolean[] colUsed;
4.   private boolean[] diagonals45Used;
5.   private boolean[] diagonals135Used;
6.   private int n;
7.
8.   public List<List<String>> solveNQueens(int n) {
9.       solutions = new ArrayList<>();
10.      nQueens = new char[n][n];
11.      for (int i = 0; i < n; i++) {
12.          Arrays.fill(nQueens[i], '.');
13.      }
14.      colUsed = new boolean[n];
15.      diagonals45Used = new boolean[2 * n - 1];
16.      diagonals135Used = new boolean[2 * n - 1];
17.      this.n = n;
18.      backtracking(0);
19.      return solutions;
20.  }
21.
22.  private void backtracking(int row) {
23.      if (row == n) {
24.          List<String> list = new ArrayList<>();
25.          for (char[] chars : nQueens) {
26.              list.add(new String(chars));
27.          }
28.          solutions.add(list);
```

```
29.          return;
30.      }
31.
32.      for (int col = 0; col < n; col++) {
33.          int diagonals45Idx = row + col;
34.          int diagonals135Idx = n - 1 - (row - col);
35.          if (colUsed[col] || diagonals45Used[diagonals45Idx] || diagonal
     s135Used[diagonals135Idx]) {
36.              continue;
37.          }
38.          nQueens[row][col] = 'Q';
39.          colUsed[col] = diagonals45Used[diagonals45Idx] =
     diagonals135Used[diagonals135Idx] = true;
40.          backtracking(row + 1);
41.          colUsed[col] = diagonals45Used[diagonals45Idx] =
     diagonals135Used[diagonals135Idx] = false;
42.          nQueens[row][col] = '.';
43.      }
44.  }
```

分治

给表达式加括号

241. Different Ways to Add Parentheses (Medium)

```
1.  Input: "2-1-1".
2.
3.  ((2-1)-1) = 0
4.  (2-(1-1)) = 2
5.
6.  Output : [0, 2]
```

```
1.  public List<Integer> diffWaysToCompute(String input) {
2.      List<Integer> ways = new ArrayList<>();
3.      for (int i = 0; i < input.length(); i++) {
4.          char c = input.charAt(i);
5.          if (c == '+' || c == '-' || c == '*') {
6.              List<Integer> left = diffWaysToCompute(input.substring(0, i
     ));
7.              List<Integer> right = diffWaysToCompute(input.substring(i +
```

```
    1));
 8.                 for (int l : left) {
 9.                     for (int r : right) {
10.                         switch (c) {
11.                             case '+':
12.                                 ways.add(l + r);
13.                                 break;
14.                             case '-':
15.                                 ways.add(l - r);
16.                                 break;
17.                             case '*':
18.                                 ways.add(l * r);
19.                                 break;
20.                         }
21.                     }
22.                 }
23.             }
24.         }
25.         if (ways.size() == 0) {
26.             ways.add(Integer.valueOf(input));
27.         }
28.         return ways;
29.     }
```

动态规划

递归和动态规划都是将原问题拆成多个子问题然后求解，他们之间最本质的区别是，动态规划保存了子问题的解，避免重复计算。

斐波那契数列

爬楼梯

70. Climbing Stairs (Easy)

题目描述：有 N 阶楼梯，每次可以上一阶或者两阶，求有多少种上楼梯的方法。

定义一个数组 dp 存储上楼梯的方法数（为了方便讨论，数组下标从 1 开始），dp[i] 表示走到第 i 个楼梯的方法数目。第 i 个楼梯可以从第 i-1 和 i-2 个楼梯再走一步到达，走到第 i 个

楼梯的方法数为走到第 i-1 和第 i-2 个楼梯的方法数之和。

$$dp[i] = dp[i-1] + dp[i-2]$$

```java
1.   public int climbStairs(int n) {
2.       if (n <= 2) {
3.           return n;
4.       }
5.       int pre2 = 1, pre1 = 2;
6.       for (int i = 2; i < n; i++) {
7.           int cur = pre1 + pre2;
8.           pre2 = pre1;
9.           pre1 = cur;
10.      }
11.      return pre1;
12.  }
```

强盗抢劫

198. House Robber (Easy)

题目描述：抢劫一排住户，但是不能抢邻近的住户，求最大抢劫量。

定义 dp 数组用来存储最大的抢劫量，其中 dp[i] 表示抢到第 i 个住户时的最大抢劫量。由于不能抢劫邻近住户，因此如果抢劫了第 i 个住户那么只能抢劫 i - 2 或者 i - 3 的住户，所以

$$dp[i] = max(dp[i-2], dp[i-3]) + nums[i]$$

```java
1.   public int rob(int[] nums) {
2.       int n = nums.length;
3.       if (n == 0) {
4.           return 0;
5.       }
6.       if (n == 1) {
7.           return nums[0];
8.       }
9.       int pre3 = 0, pre2 = 0, pre1 = 0;
10.      for (int i = 0; i < n; i++) {
11.          int cur = Math.max(pre2, pre3) + nums[i];
12.          pre3 = pre2;
```

```
13.            pre2 = pre1;
14.            pre1 = cur;
15.        }
16.        return Math.max(pre1, pre2);
17.    }
```

强盗在环形街区抢劫

213. House Robber II (Medium)

```
1.   public int rob(int[] nums) {
2.       if (nums == null || nums.length == 0) {
3.           return 0;
4.       }
5.       int n = nums.length;
6.       if (n == 1) {
7.           return nums[0];
8.       }
9.       return Math.max(rob(nums, 0, n - 2), rob(nums, 1, n - 1));
10.   }
11.
12.   private int rob(int[] nums, int first, int last) {
13.       int pre3 = 0, pre2 = 0, pre1 = 0;
14.       for (int i = first; i <= last; i++) {
15.           int cur = Math.max(pre3, pre2) + nums[i];
16.           pre3 = pre2;
17.           pre2 = pre1;
18.           pre1 = cur;
19.       }
20.       return Math.max(pre2, pre1);
21.   }
```

母牛生产

程序员代码面试指南-P181

题目描述：假设农场中成熟的母牛每年都会生 1 头小母牛，并且永远不会死。第一年有 1 只小母牛，从第二年开始，母牛开始生小母牛。每只小母牛 3 年之后成熟又可以生小母牛。给定整数 N，求 N 年后牛的数量。

第 i 年成熟的牛的数量为：

$$dp[i] = dp[i-1] + dp[i-3]$$

$$dp[i] = (i-1) * dp[i-2] + (i-1) * dp[i-1]$$

矩阵路径

矩阵的最小路径和

64. Minimum Path Sum (Medium)

```
[[1,3,1],
 [1,5,1],
 [4,2,1]]
Given the above grid map, return 7. Because the path 1→3→1→1→1 minimizes the sum.
```

题目描述：求从矩阵的左上角到右下角的最小路径和，每次只能向右和向下移动。

```java
public int minPathSum(int[][] grid) {
    if (grid.length == 0 || grid[0].length == 0) {
        return 0;
    }
    int m = grid.length, n = grid[0].length;
    int[] dp = new int[n];
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (j == 0) {
                dp[j] = dp[j];          // 只能从上侧走到该位置
            } else if (i == 0) {
                dp[j] = dp[j - 1];      // 只能从左侧走到该位置
            } else {
                dp[j] = Math.min(dp[j - 1], dp[j]);
            }
            dp[j] += grid[i][j];
        }
    }
    return dp[n - 1];
}
```

矩阵的总路径数

62. Unique Paths (Medium)

题目描述：统计从矩阵左上角到右下角的路径总数，每次只能向右或者向下移动。


```java
public int uniquePaths(int m, int n) {
    int[] dp = new int[n];
    Arrays.fill(dp, 1);
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[j] = dp[j] + dp[j - 1];
        }
    }
    return dp[n - 1];
}
```

也可以直接用数学公式求解，这是一个组合问题。机器人总共移动的次数 S=m+n-2，向下移动的次数 D=m-1，那么问题可以看成从 S 从取出 D 个位置的组合数量，这个问题的解为 C(S, D)。

```java
public int uniquePaths(int m, int n) {
    int S = m + n - 2;  // 总共的移动次数
    int D = m - 1;      // 向下的移动次数
    long ret = 1;
    for (int i = 1; i <= D; i++) {
        ret = ret * (S - D + i) / i;
    }
    return (int) ret;
}
```

```
9.    }
```

数组区间

数组区间和

303. Range Sum Query - Immutable (Easy)

```
1.    Given nums = [-2, 0, 3, -5, 2, -1]
2.
3.    sumRange(0, 2) -> 1
4.    sumRange(2, 5) -> -1
5.    sumRange(0, 5) -> -3
```

求区间 i \~ j 的和，可以转换为 sum[j] - sum[i-1]，其中 sum[i] 为 0 \~ i 的和。

```
1.    class NumArray {
2.        private int[] sums;
3.
4.        public NumArray(int[] nums) {
5.            sums = new int[nums.length];
6.            for (int i = 0; i < nums.length; i++) {
7.                sums[i] = i == 0 ? nums[0] : sums[i - 1] + nums[i];
8.            }
9.        }
10.
11.       public int sumRange(int i, int j) {
12.           return i == 0 ? sums[j] : sums[j] - sums[i - 1];
13.       }
14.   }
```

子数组最大的和

53. Maximum Subarray (Easy)

```
1.    For example, given the array [-2,1,-3,4,-1,2,1,-5,4],
2.    the contiguous subarray [4,-1,2,1] has the largest sum = 6.
```

```
1.    public int maxSubArray(int[] nums) {
```

```
2.        if (nums == null || nums.length == 0) {
3.            return 0;
4.        }
5.        int preSum = nums[0];
6.        int maxSum = preSum;
7.        for (int i = 1; i < nums.length; i++) {
8.            preSum = preSum > 0 ? preSum + nums[i] : nums[i];
9.            maxSum = Math.max(maxSum, preSum);
10.        }
11.        return maxSum;
12.    }
```

数组中等差递增子区间的个数

413. Arithmetic Slices (Medium)

```
1.    A = [1, 2, 3, 4]
2.    return: 3, for 3 arithmetic slices in A: [1, 2, 3], [2, 3, 4] and [1,
      2, 3, 4] itself.
```

dp[i] 表示以 A[i] 为结尾的等差递增子区间的个数。

如果 A[i] - A[i - 1] == A[i - 1] - A[i - 2] , 表示 [A[i - 2], A[i - 1], A[i]] 是一个等差递增子区间。如果 [A[i - 3], A[i - 2], A[i - 1]] 是一个等差递增子区间 , 那么 [A[i - 3], A[i - 2], A[i - 1], A[i]] 也是。因此在这个条件下 , dp[i] = dp[i-1] + 1。

```
1.    public int numberOfArithmeticSlices(int[] A) {
2.        if (A == null || A.length == 0) {
3.            return 0;
4.        }
5.        int n = A.length;
6.        int[] dp = new int[n];
7.        for (int i = 2; i < n; i++) {
8.            if (A[i] - A[i - 1] == A[i - 1] - A[i - 2]) {
9.                dp[i] = dp[i - 1] + 1;
10.            }
11.        }
12.        int total = 0;
13.        for (int cnt : dp) {
14.            total += cnt;
15.        }
```

```
16.        return total;
17.    }
```

分割整数

分割整数的最大乘积

343. Integer Break (Medim)

题目描述：For example, given n = 2, return 1 (2 = 1 + 1); given n = 10, return 36 (10 =
3 + 3 + 4).

```
1.   public int integerBreak(int n) {
2.       int[] dp = new int[n + 1];
3.       dp[1] = 1;
4.       for (int i = 2; i <= n; i++) {
5.           for (int j = 1; j <= i - 1; j++) {
6.               dp[i] = Math.max(dp[i], Math.max(j * dp[i - j], j * (i - j))
     );
7.           }
8.       }
9.       return dp[n];
10.  }
```

按平方数来分割整数

279. Perfect Squares(Medium)

题目描述：For example, given n = 12, return 3 because 12 = 4 + 4 + 4; given n = 13,
return 2 because 13 = 4 + 9.

```
1.   public int numSquares(int n) {
2.       List<Integer> squareList = generateSquareList(n);
3.       int[] dp = new int[n + 1];
4.       for (int i = 1; i <= n; i++) {
5.           int min = Integer.MAX_VALUE;
6.           for (int square : squareList) {
7.               if (square > i) {
8.                   break;
9.               }
```

```
10.            min = Math.min(min, dp[i - square] + 1);
11.        }
12.        dp[i] = min;
13.    }
14.    return dp[n];
15. }
16.
17. private List<Integer> generateSquareList(int n) {
18.     List<Integer> squareList = new ArrayList<>();
19.     int diff = 3;
20.     int square = 1;
21.     while (square <= n) {
22.         squareList.add(square);
23.         square += diff;
24.         diff += 2;
25.     }
26.     return squareList;
27. }
```

分割整数构成字母字符串

91. Decode Ways (Medium)

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

```
1.  public int numDecodings(String s) {
2.      if (s == null || s.length() == 0) {
3.          return 0;
4.      }
5.      int n = s.length();
6.      int[] dp = new int[n + 1];
7.      dp[0] = 1;
8.      dp[1] = s.charAt(0) == '0' ? 0 : 1;
9.      for (int i = 2; i <= n; i++) {
10.         int one = Integer.valueOf(s.substring(i - 1, i));
11.         if (one != 0) {
12.             dp[i] += dp[i - 1];
13.         }
14.         if (s.charAt(i - 2) == '0') {
15.             continue;
16.         }
17.         int two = Integer.valueOf(s.substring(i - 2, i));
18.         if (two <= 26) {
```

```
19.            dp[i] += dp[i - 2];
20.        }
21.    }
22.    return dp[n];
23. }
```

最长递增子序列

已知一个序列 {S$_1$, S$_2$,...,S$_n$}，取出若干数组成新的序列 {S$_{i1}$, S$_{i2}$,..., S$_{im}$}，其中 i1、i2 ... im 保持递增，即新序列中各个数仍然保持原数列中的先后顺序，称新序列为原序列的一个 **子序列** 。

如果在子序列中，当下标 ix > iy 时，S$_{ix}$ > S$_{iy}$，称子序列为原序列的一个 **递增子序列** 。

定义一个数组 dp 存储最长递增子序列的长度，dp[n] 表示以 S$_n$ 结尾的序列的最长递增子序列长度。对于一个递增子序列 {S$_{i1}$, S$_{i2}$,...,S$_{im}$}，如果 im < n 并且 S$_{im}$ < S$_n$ ，此时 {S$_{i1}$, S$_{i2}$,..., S$_{im}$, S$_n$} 为一个递增子序列，递增子序列的长度增加 1。满足上述条件的递增子序列中，长度最长的那个递增子序列就是要找的，在长度最长的递增子序列上加上 S$_n$ 就构成了以 S$_n$ 为结尾的最长递增子序列。因此 dp[n] = max{ dp[i]+1 | S$_i$ < S$_n$ && i < n} 。

因为在求 dp[n] 时可能无法找到一个满足条件的递增子序列，此时 {S$_n$} 就构成了递增子序列，需要对前面的求解方程做修改，令 dp[n] 最小为 1，即：

$$dp[n] = max\{1, dp[i] + 1 | S_i < S_n \&\& i < n\}$$

N
```
1.  public int lengthOfLIS(int[] nums) {
2.      int n = nums.length;
3.      int[] dp = new int[n];
4.      for (int i = 0; i < n; i++) {
5.          int max = 1;
6.          for (int j = 0; j < i; j++) {
7.              if (nums[i] > nums[j]) {
8.                  max = Math.max(max, dp[j] + 1);
9.              }
10.         }
11.         dp[i] = max;
12.     }
```

```
13.         return Arrays.stream(dp).max().orElse(0);
14.     }
```

使用 Stream 求最大值会导致运行时间过长，可以改成以下形式：

```
1.     int ret = 0;
2.     for (int i = 0; i < n; i++) {
3.         ret = Math.max(ret, dp[i]);
4.     }
5.     return ret;
```

以上解法的时间复杂度为 O(N²)，可以使用二分查找将时间复杂度降低为 O(NlogN)。

定义一个 tails 数组，其中 tails[i] 存储长度为 i + 1 的最长递增子序列的最后一个元素。对于一个元素 x，

- 如果它大于 tails 数组所有的值，那么把它添加到 tails 后面，表示最长递增子序列长度加 1；
- 如果 tails[i-1] < x <= tails[i]，那么更新 tails[i-1] = x。

例如对于数组 [4,3,6,5]，有：

```
1.     tails       len       num
2.     []          0         4
3.     [4]         1         3
4.     [3]         1         6
5.     [3,6]       2         5
6.     [3,5]       2         null
```

可以看出 tails 数组保持有序，因此在查找 S_i 位于 tails 数组的位置时就可以使用二分查找。

```
1.     public int lengthOfLIS(int[] nums) {
2.         int n = nums.length;
3.         int[] tails = new int[n];
4.         int len = 0;
5.         for (int num : nums) {
6.             int index = binarySearch(tails, len, num);
7.             tails[index] = num;
8.             if (index == len) {
9.                 len++;
```

```
10.              }
11.         }
12.         return len;
13.     }
14.
15.     private int binarySearch(int[] tails, int len, int key) {
16.         int l = 0, h = len;
17.         while (l < h) {
18.             int mid = l + (h - l) / 2;
19.             if (tails[mid] == key) {
20.                 return mid;
21.             } else if (tails[mid] > key) {
22.                 h = mid;
23.             } else {
24.                 l = mid + 1;
25.             }
26.         }
27.         return l;
28.     }
```

一组整数对能够构成的最长链

646. Maximum Length of Pair Chain (Medium)

```
1.   Input: [[1,2], [2,3], [3,4]]
2.   Output: 2
3.   Explanation: The longest chain is [1,2] -> [3,4]
```

题目描述：对于 (a, b) 和 (c, d)，如果 b < c，则它们可以构成一条链。

```
1.   public int findLongestChain(int[][] pairs) {
2.       if (pairs == null || pairs.length == 0) {
3.           return 0;
4.       }
5.       Arrays.sort(pairs, (a, b) -> (a[0] - b[0]));
6.       int n = pairs.length;
7.       int[] dp = new int[n];
8.       Arrays.fill(dp, 1);
9.       for (int i = 1; i < n; i++) {
10.          for (int j = 0; j < i; j++) {
11.              if (pairs[j][1] < pairs[i][0]) {
12.                  dp[i] = Math.max(dp[i], dp[j] + 1);
```

```
13.                 }
14.             }
15.         }
16.         return Arrays.stream(dp).max().orElse(0);
17.     }
```

最长摆动子序列

376. Wiggle Subsequence (Medium)

```
1.    Input: [1,7,4,9,2,5]
2.    Output: 6
3.    The entire sequence is a wiggle sequence.
4.
5.    Input: [1,17,5,10,13,15,10,5,16,8]
6.    Output: 7
7.    There are several subsequences that achieve this length. One is [1,17,
      10,13,10,16,8].
8.
9.    Input: [1,2,3,4,5,6,7,8,9]
10.   Output: 2
```

要求：使用 O(N) 时间复杂度求解。

```
1.    public int wiggleMaxLength(int[] nums) {
2.        if (nums == null || nums.length == 0) {
3.            return 0;
4.        }
5.        int up = 1, down = 1;
6.        for (int i = 1; i < nums.length; i++) {
7.            if (nums[i] > nums[i - 1]) {
8.                up = down + 1;
9.            } else if (nums[i] < nums[i - 1]) {
10.               down = up + 1;
11.           }
12.       }
13.       return Math.max(up, down);
14.   }
```

最长公共子序列

对于两个子序列 S1 和 S2，找出它们最长的公共子序列。

定义一个二维数组 dp 用来存储最长公共子序列的长度，其中 dp[i][j] 表示 S1 的前 i 个字符与 S2 的前 j 个字符最长公共子序列的长度。考虑 $S1_i$ 与 $S2_j$ 值是否相等，分为两种情况：

- 当 $S1_i == S2_j$ 时，那么就能在 S1 的前 i-1 个字符与 S2 的前 j-1 个字符最长公共子序列的基础上再加上 $S1_i$ 这个值，最长公共子序列长度加 1 ，即 dp[i][j] = dp[i-1][j-1] + 1。
- 当 $S1_i\ != S2_j$ 时，此时最长公共子序列为 S1 的前 i-1 个字符和 S2 的前 j 个字符最长公共子序列，与 S1 的前 i 个字符和 S2 的前 j-1 个字符最长公共子序列，它们的最大者，即 dp[i][j] = max{ dp[i-1][j], dp[i][j-1] }。

综上，最长公共子序列的状态转移方程为：

$$
dp[i][j] = \begin{cases} dp[i-1][j-1] & S1_i == S2_j \\ max(dp[i-1][j], dp[i][j-1]) & S1_i <> S2_j \end{cases}
$$

1212iiijN

```java
public int lengthOfLCS(int[] nums1, int[] nums2) {
    int n1 = nums1.length, n2 = nums2.length;
    int[][] dp = new int[n1 + 1][n2 + 1];
    for (int i = 1; i <= n1; i++) {
        for (int j = 1; j <= n2; j++) {
            if (nums1[i - 1] == nums2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[n1][n2];
}
```

0-1 背包

有一个容量为 N 的背包，要用这个背包装下物品的价值最大，这些物品有两个属性：体积 w 和价值 v。

定义一个二维数组 dp 存储最大价值，其中 dp[i][j] 表示前 i 件物品体积不超过 j 的情况下能

达到的最大价值。设第 i 件物品体积为 w，价值为 v，根据第 i 件物品是否添加到背包中，可以分两种情况讨论：

- 第 i 件物品没添加到背包，总体积不超过 j 的前 i 件物品的最大价值就是总体积不超过 j 的前 i-1 件物品的最大价值，dp[i][j] = dp[i-1][j]。
- 第 i 件物品添加到背包中，dp[i][j] = dp[i-1][j-w] + v。

第 i 件物品可添加也可以不添加，取决于哪种情况下最大价值更大。

综上，0-1 背包的状态转移方程为：

$$dp[i][j] = max(dp[i-1][j], dp[i-1][j-w] + v)$$

```java
public int knapsack(int W, int N, int[] weights, int[] values) {
    int[][] dp = new int[N + 1][W + 1];
    for (int i = 1; i <= N; i++) {
        int w = weights[i - 1], v = values[i - 1];
        for (int j = 1; j <= W; j++) {
            if (j >= w) {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - w] + v);
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
    return dp[N][W];
}
```

空间优化

在程序实现时可以对 0-1 背包做优化。观察状态转移方程可以知道，前 i 件物品的状态仅由前 i-1 件物品的状态有关，因此可以将 dp 定义为一维数组，其中 dp[j] 既可以表示 dp[i-1][j] 也可以表示 dp[i][j]。此时，

$$dp[j] = max(dp[j], dp[j-w] + v)$$

```java
public int knapsack(int W, int N, int[] weights, int[] values) {
    int[] dp = new int[W + 1];
    for (int i = 1; i <= N; i++) {
```

```
4.            int w = weights[i - 1], v = values[i - 1];
5.            for (int j = W; j >= 1; j--) {
6.                if (j >= w) {
7.                    dp[j] = Math.max(dp[j], dp[j - w] + v);
8.                }
9.            }
10.        }
11.        return dp[W];
12.    }
```

无法使用贪心算法的解释

0-1 背包问题无法使用贪心算法来求解，也就是说不能按照先添加性价比最高的物品来达到最优，这是因为这种方式可能造成背包空间的浪费，从而无法达到最优。考虑下面的物品和一个容量为 5 的背包，如果先添加物品 0 再添加物品 1，那么只能存放的价值为 16，浪费了大小为 2 的空间。最优的方式是存放物品 1 和物品 2，价值为 22.

id	w	v	v/w
0	1	6	6
1	2	10	5
2	3	12	4

变种

- 完全背包：物品数量为无限个

- 多重背包：物品数量有限制

- 多维费用背包：物品不仅有重量，还有体积，同时考虑这两种限制

- 其它：物品之间相互约束或者依赖

划分数组为和相等的两部分

416. Partition Equal Subset Sum (Medium)

```
1.    Input: [1, 5, 11, 5]
2.
3.    Output: true
```

```
4.
5.     Explanation: The array can be partitioned as [1, 5, 5] and [11].
```

可以看成一个背包大小为 sum/2 的 0-1 背包问题。

```java
1.   public boolean canPartition(int[] nums) {
2.       int sum = computeArraySum(nums);
3.       if (sum % 2 != 0) {
4.           return false;
5.       }
6.       int W = sum / 2;
7.       boolean[] dp = new boolean[W + 1];
8.       dp[0] = true;
9.       Arrays.sort(nums);
10.      for (int num : nums) {                    // 0-1 背包一个物品只能用一次
11.          for (int i = W; i >= num; i--) {      // 从后往前，先计算 dp[i] 再计算 dp[i-num]
12.              dp[i] = dp[i] || dp[i - num];
13.          }
14.      }
15.      return dp[W];
16.  }
17.
18.  private int computeArraySum(int[] nums) {
19.      int sum = 0;
20.      for (int num : nums) {
21.          sum += num;
22.      }
23.      return sum;
24.  }
```

改变一组数的正负号使得它们的和为一给定数

494. Target Sum (Medium)

```
1.   Input: nums is [1, 1, 1, 1, 1], S is 3.
2.   Output: 5
3.   Explanation:
4.
5.   -1+1+1+1+1 = 3
6.   +1-1+1+1+1 = 3
7.   +1+1-1+1+1 = 3
```

```
8.     +1+1+1-1+1 = 3
9.     +1+1+1+1-1 = 3
10.
11.    There are 5 ways to assign symbols to make the sum of nums be target 3
       .
```

该问题可以转换为 Subset Sum 问题，从而使用 0-1 背包的方法来求解。

可以将这组数看成两部分，P 和 N，其中 P 使用正号，N 使用负号，有以下推导：

```
1.                      sum(P) - sum(N) = target
2.    sum(P) + sum(N) + sum(P) - sum(N) = target + sum(P) + sum(N)
3.                         2 * sum(P) = target + sum(nums)
```

因此只要找到一个子集，令它们都取正号，并且和等于 (target + sum(nums))/2，就证明存在解。

```java
1.    public int findTargetSumWays(int[] nums, int S) {
2.        int sum = computeArraySum(nums);
3.        if (sum < S || (sum + S) % 2 == 1) {
4.            return 0;
5.        }
6.        int W = (sum + S) / 2;
7.        int[] dp = new int[W + 1];
8.        dp[0] = 1;
9.        Arrays.sort(nums);
10.       for (int num : nums) {
11.           for (int i = W; i >= num; i--) {
12.               dp[i] = dp[i] + dp[i - num];
13.           }
14.       }
15.       return dp[W];
16.   }
17.
18.   private int computeArraySum(int[] nums) {
19.       int sum = 0;
20.       for (int num : nums) {
21.           sum += num;
22.       }
23.       return sum;
24.   }
```

DFS 解法：

```
1.   public int findTargetSumWays(int[] nums, int S) {
2.       return findTargetSumWays(nums, 0, S);
3.   }
4.
5.   private int findTargetSumWays(int[] nums, int start, int S) {
6.       if (start == nums.length) {
7.           return S == 0 ? 1 : 0;
8.       }
9.       return findTargetSumWays(nums, start + 1, S + nums[start])
10.              + findTargetSumWays(nums, start + 1, S - nums[start]);
11.  }
```

字符串按单词列表分割

139. Word Break (Medium)

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

dict 中的单词没有使用次数的限制，因此这是一个完全背包问题。

0-1 背包和完全背包在实现上的不同之处是，0-1 背包对物品的迭代是在最外层，而完全背包对物品的迭代是在最里层。

```
1.   public boolean wordBreak(String s, List<String> wordDict) {
2.       int n = s.length();
3.       boolean[] dp = new boolean[n + 1];
4.       dp[0] = true;
5.       for (int i = 1; i <= n; i++) {
6.           for (String word : wordDict) {    // 完全一个物品可以使用多次
7.               int len = word.length();
8.               if (len <= i && word.equals(s.substring(i - len, i))) {
9.                   dp[i] = dp[i] || dp[i - len];
10.              }
11.          }
12.      }
13.      return dp[n];
14.  }
```

01 字符构成最多的字符串

474. Ones and Zeroes (Medium)

```
1.   Input: Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3
2.   Output: 4
3.
4.   Explanation: There are totally 4 strings can be formed by the using of
     5 0s and 3 1s, which are "10","0001","1","0"
```

这是一个多维费用的 0-1 背包问题，有两个背包大小，0 的数量和 1 的数量。

```java
1.   public int findMaxForm(String[] strs, int m, int n) {
2.       if (strs == null || strs.length == 0) {
3.           return 0;
4.       }
5.       int[][] dp = new int[m + 1][n + 1];
6.       for (String s : strs) {    // 每个字符串只能用一次
7.           int ones = 0, zeros = 0;
8.           for (char c : s.toCharArray()) {
9.               if (c == '0') {
10.                  zeros++;
11.              } else {
12.                  ones++;
13.              }
14.          }
15.          for (int i = m; i >= zeros; i--) {
16.              for (int j = n; j >= ones; j--) {
17.                  dp[i][j] = Math.max(dp[i][j], dp[i - zeros][j - ones] +
     1);
18.              }
19.          }
20.      }
21.      return dp[m][n];
22.  }
```

找零钱的方法数

322. Coin Change (Medium)

```
1.    Example 1:
2.    coins = [1, 2, 5], amount = 11
3.    return 3 (11 = 5 + 5 + 1)
4.
5.    Example 2:
6.    coins = [2], amount = 3
7.    return -1.
```

题目描述：给一些面额的硬币，要求用这些硬币来组成给定面额的钱数，并且使得硬币数量最少。硬币可以重复使用。

- 物品：硬币
- 物品大小：面额
- 物品价值：数量

因为硬币可以重复使用，因此这是一个完全背包问题。

```
1.    public int coinChange(int[] coins, int amount) {
2.        if (coins == null || coins.length == 0) {
3.            return 0;
4.        }
5.        int[] minimum = new int[amount + 1];
6.        Arrays.fill(minimum, amount + 1);
7.        minimum[0] = 0;
8.        Arrays.sort(coins);
9.        for (int i = 1; i <= amount; i++) {
10.           for (int j = 0; j < coins.length && coins[j] <= i; j++) {
11.               minimum[i] = Math.min(minimum[i], minimum[i - coins[j]] + 1);
12.           }
13.       }
14.       return minimum[amount] > amount ? -1 : minimum[amount];
15.   }
```

组合总和

377. Combination Sum IV (Medium)

```
1.    nums = [1, 2, 3]
2.    target = 4
3.
```

```
  4.     The possible combination ways are:
  5.     (1, 1, 1, 1)
  6.     (1, 1, 2)
  7.     (1, 2, 1)
  8.     (1, 3)
  9.     (2, 1, 1)
 10.     (2, 2)
 11.     (3, 1)
 12.
 13.     Note that different sequences are counted as different combinations.
 14.
 15.     Therefore the output is 7.
```

完全背包。

```
  1.   public int combinationSum4(int[] nums, int target) {
  2.       if (nums == null || nums.length == 0) {
  3.           return 0;
  4.       }
  5.       int[] maximum = new int[target + 1];
  6.       maximum[0] = 1;
  7.       Arrays.sort(nums);
  8.       for (int i = 1; i <= target; i++) {
  9.           for (int j = 0; j < nums.length && nums[j] <= i; j++) {
 10.               maximum[i] += maximum[i - nums[j]];
 11.           }
 12.       }
 13.       return maximum[target];
 14.   }
```

只能进行 k 次的股票交易

188. Best Time to Buy and Sell Stock IV (Hard)

```
  1.   public int maxProfit(int k, int[] prices) {
  2.       int n = prices.length;
  3.       if (k >= n / 2) {   // 这种情况下该问题退化为普通的股票交易问题
  4.           int maxProfit = 0;
  5.           for (int i = 1; i < n; i++) {
  6.               if (prices[i] > prices[i - 1]) {
  7.                   maxProfit += prices[i] - prices[i - 1];
  8.               }
```

```
 9.            }
10.            return maxProfit;
11.        }
12.        int[][] maxProfit = new int[k + 1][n];
13.        for (int i = 1; i <= k; i++) {
14.            int localMax = maxProfit[i - 1][0] - prices[0];
15.            for (int j = 1; j < n; j++) {
16.                maxProfit[i][j] = Math.max(maxProfit[i][j - 1], prices[j] +
    localMax);
17.                localMax = Math.max(localMax, maxProfit[i - 1][j] - prices[
    j]);
18.            }
19.        }
20.        return maxProfit[k][n - 1];
21.    }
```

只能进行两次的股票交易

123. Best Time to Buy and Sell Stock III (Hard)

```
 1.    public int maxProfit(int[] prices) {
 2.        int firstBuy = Integer.MIN_VALUE, firstSell = 0;
 3.        int secondBuy = Integer.MIN_VALUE, secondSell = 0;
 4.        for (int curPrice : prices) {
 5.            if (firstBuy < -curPrice) {
 6.                firstBuy = -curPrice;
 7.            }
 8.            if (firstSell < firstBuy + curPrice) {
 9.                firstSell = firstBuy + curPrice;
10.            }
11.            if (secondBuy < firstSell - curPrice) {
12.                secondBuy = firstSell - curPrice;
13.            }
14.            if (secondSell < secondBuy + curPrice) {
15.                secondSell = secondBuy + curPrice;
16.            }
17.        }
18.        return secondSell;
19.    }
```
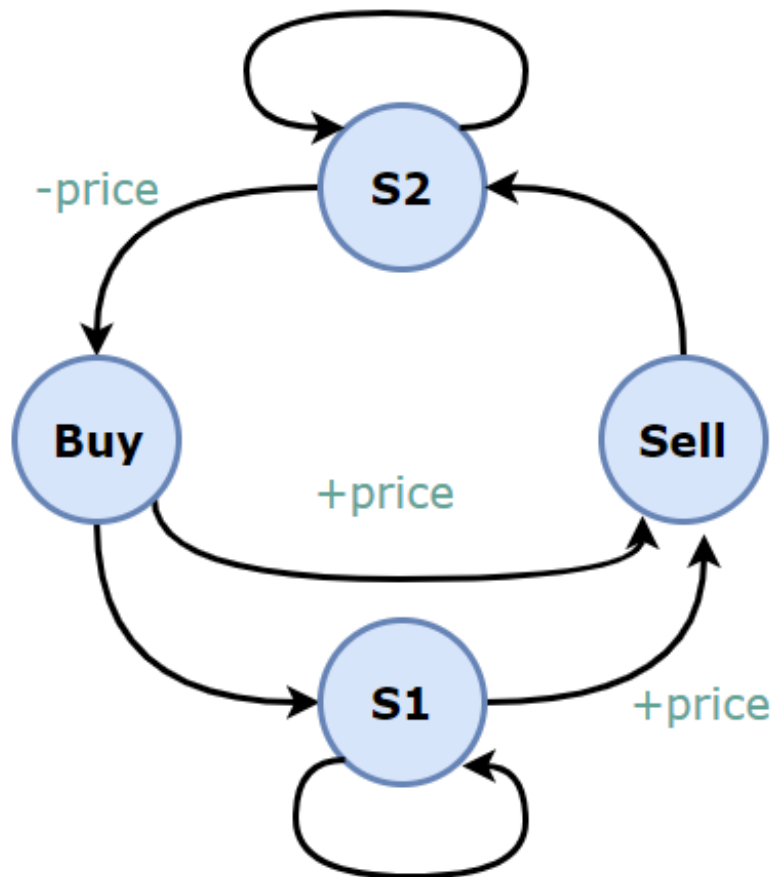
股票交易

需要冷却期的股票交易

309. Best Time to Buy and Sell Stock with Cooldown(Medium)

题目描述：交易之后需要有一天的冷却时间。


```java
public int maxProfit(int[] prices) {
    if (prices == null || prices.length == 0) {
        return 0;
    }
    int N = prices.length;
    int[] buy = new int[N];
    int[] s1 = new int[N];
    int[] sell = new int[N];
    int[] s2 = new int[N];
    s1[0] = buy[0] = -prices[0];
    sell[0] = s2[0] = 0;
    for (int i = 1; i < N; i++) {
```
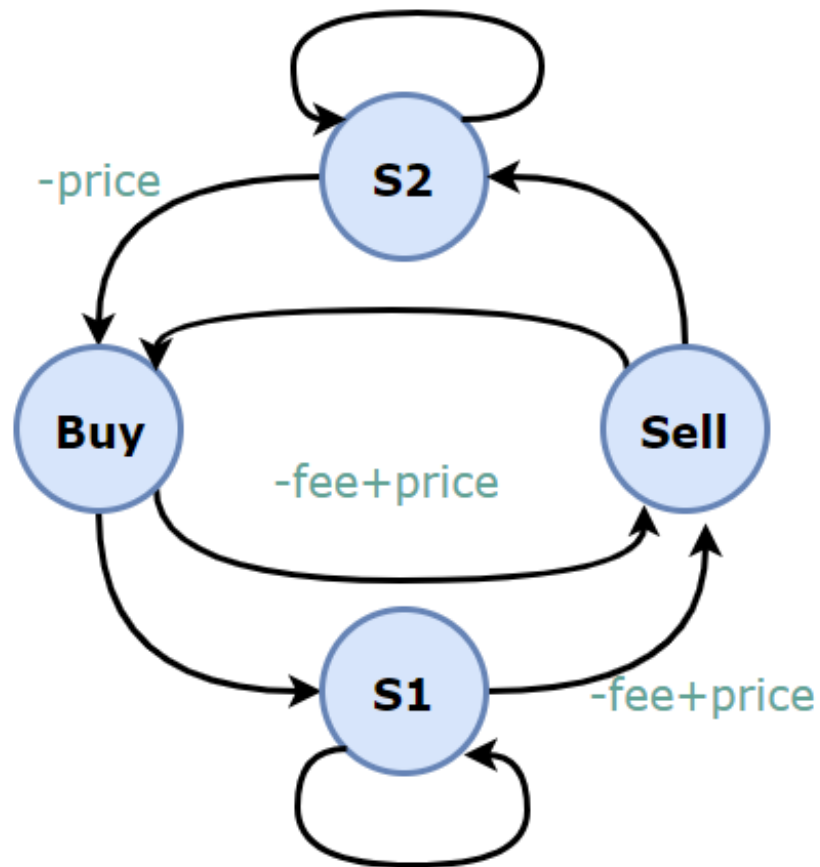
```
13.            buy[i] = s2[i - 1] - prices[i];
14.            s1[i] = Math.max(buy[i - 1], s1[i - 1]);
15.            sell[i] = Math.max(buy[i - 1], s1[i - 1]) + prices[i];
16.            s2[i] = Math.max(s2[i - 1], sell[i - 1]);
17.        }
18.        return Math.max(sell[N - 1], s2[N - 1]);
19.    }
```

需要交易费用的股票交易

714. Best Time to Buy and Sell Stock with Transaction Fee (Medium)

```
1.    Input: prices = [1, 3, 2, 8, 4, 9], fee = 2
2.    Output: 8
3.    Explanation: The maximum profit can be achieved by:
4.    Buying at prices[0] = 1
5.    Selling at prices[3] = 8
6.    Buying at prices[4] = 4
7.    Selling at prices[5] = 9
8.    The total profit is ((8 - 1) - 2) + ((9 - 4) - 2) = 8.
```

题目描述：每交易一次，都要支付一定的费用。

```
1.    public int maxProfit(int[] prices, int fee) {
2.        int N = prices.length;
3.        int[] buy = new int[N];
4.        int[] s1 = new int[N];
5.        int[] sell = new int[N];
6.        int[] s2 = new int[N];
7.        s1[0] = buy[0] = -prices[0];
8.        sell[0] = s2[0] = 0;
9.        for (int i = 1; i < N; i++) {
10.           buy[i] = Math.max(sell[i - 1], s2[i - 1]) - prices[i];
11.           s1[i] = Math.max(buy[i - 1], s1[i - 1]);
12.           sell[i] = Math.max(buy[i - 1], s1[i - 1]) - fee + prices[i];
13.           s2[i] = Math.max(s2[i - 1], sell[i - 1]);
14.       }
15.       return Math.max(sell[N - 1], s2[N - 1]);
16.   }
```

买入和售出股票最大的收益

121. Best Time to Buy and Sell Stock (Easy)

题目描述：只进行一次交易。

只要记录前面的最小价格，将这个最小价格作为买入价格，然后将当前的价格作为售出价格，查看当前收益是不是最大收益。

```java
public int maxProfit(int[] prices) {
    int n = prices.length;
    if (n == 0) return 0;
    int soFarMin = prices[0];
    int max = 0;
    for (int i = 1; i < n; i++) {
        if (soFarMin > prices[i]) soFarMin = prices[i];
        else max = Math.max(max, prices[i] - soFarMin);
    }
    return max;
}
```

字符串编辑

删除两个字符串的字符使它们相等

583. Delete Operation for Two Strings (Medium)

```
Input: "sea", "eat"
Output: 2
Explanation: You need one step to make "sea" to "ea" and another step
to make "eat" to "ea".
```

可以转换为求两个字符串的最长公共子序列问题。

```java
public int minDistance(String word1, String word2) {
    int m = word1.length(), n = word2.length();
    int[][] dp = new int[m + 1][n + 1];
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                continue;
```

```
8.                  }
9.              if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
10.                 dp[i][j] = dp[i - 1][j - 1] + 1;
11.             } else {
12.                 dp[i][j] = Math.max(dp[i][j - 1], dp[i - 1][j]);
13.             }
14.         }
15.     }
16.     return m + n - 2 * dp[m][n];
17. }
```

编辑距离

72. Edit Distance (Hard)

```
1.  Example 1:
2.
3.  Input: word1 = "horse", word2 = "ros"
4.  Output: 3
5.  Explanation:
6.  horse -> rorse (replace 'h' with 'r')
7.  rorse -> rose (remove 'r')
8.  rose -> ros (remove 'e')
9.  Example 2:
10.
11. Input: word1 = "intention", word2 = "execution"
12. Output: 5
13. Explanation:
14. intention -> inention (remove 't')
15. inention -> enention (replace 'i' with 'e')
16. enention -> exention (replace 'n' with 'x')
17. exention -> exection (replace 'n' with 'c')
18. exection -> execution (insert 'u')
```

题目描述：修改一个字符串成为另一个字符串，使得修改次数最少。一次修改操作包括：插入一个字符、删除一个字符、替换一个字符。

```
1.  public int minDistance(String word1, String word2) {
2.      if (word1 == null || word2 == null) {
3.          return 0;
4.      }
5.      int m = word1.length(), n = word2.length();
```

```
6.        int[][] dp = new int[m + 1][n + 1];
7.        for (int i = 1; i <= m; i++) {
8.            dp[i][0] = i;
9.        }
10.       for (int i = 1; i <= n; i++) {
11.           dp[0][i] = i;
12.       }
13.       for (int i = 1; i <= m; i++) {
14.           for (int j = 1; j <= n; j++) {
15.               if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
16.                   dp[i][j] = dp[i - 1][j - 1];
17.               } else {
18.                   dp[i][j] = Math.min(dp[i - 1][j - 1], Math.min(dp[i][j -
    1], dp[i - 1][j])) + 1;
19.               }
20.           }
21.       }
22.       return dp[m][n];
23.   }
```

复制粘贴字符

650. 2 Keys Keyboard (Medium)

题目描述：最开始只有一个字符 A，问需要多少次操作能够得到 n 个字符 A，每次操作可以复制当前所有的字符，或者粘贴。

```
1.    Input: 3
2.    Output: 3
3.    Explanation:
4.    Intitally, we have one character 'A'.
5.    In step 1, we use Copy All operation.
6.    In step 2, we use Paste operation to get 'AA'.
7.    In step 3, we use Paste operation to get 'AAA'.
```

```
1.    public int minSteps(int n) {
2.        if (n == 1) return 0;
3.        for (int i = 2; i <= Math.sqrt(n); i++) {
4.            if (n % i == 0) return i + minSteps(n / i);
5.        }
6.        return n;
7.    }
```

```java
public int minSteps(int n) {
    int[] dp = new int[n + 1];
    int h = (int) Math.sqrt(n);
    for (int i = 2; i <= n; i++) {
        dp[i] = i;
        for (int j = 2; j <= h; j++) {
            if (i % j == 0) {
                dp[i] = dp[j] + dp[i / j];
                break;
            }
        }
    }
    return dp[n];
}
```

数学

素数

素数分解

每一个数都可以分解成素数的乘积，例如 $84 = 2^2 * 3^1 * 5^0 * 7^1 * 11^0 * 13^0 * 17^0 * ...$

整除

令 $x = 2^{m0} * 3^{m1} * 5^{m2} * 7^{m3} * 11^{m4} * ...$

令 $y = 2^{n0} * 3^{n1} * 5^{n2} * 7^{n3} * 11^{n4} * ...$

如果 x 整除 y（y mod x == 0），则对于所有 i，mi <= ni。

最大公约数最小公倍数

x 和 y 的最大公约数为：$gcd(x,y) = 2^{min(m0,n0)} * 3^{min(m1,n1)} * 5^{min(m2,n2)} * ...$

x 和 y 的最小公倍数为：$lcm(x,y) = 2^{max(m0,n0)} * 3^{max(m1,n1)} * 5^{max(m2,n2)} * ...$

生成素数序列

204. Count Primes (Easy)

埃拉托斯特尼筛法在每次找到一个素数时，将能被素数整除的数排除掉。

```java
public int countPrimes(int n) {
    boolean[] notPrimes = new boolean[n + 1];
    int count = 0;
    for (int i = 2; i < n; i++) {
        if (notPrimes[i]) {
            continue;
        }
        count++;
        // 从 i * i 开始，因为如果 k < i，那么 k * i 在之前就已经被去除过了
        for (long j = (long) (i) * i; j < n; j += i) {
            notPrimes[(int) j] = true;
        }
    }
    return count;
}
```

最大公约数

```java
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a% b);
}
```

最小公倍数为两数的乘积除以最大公约数。

```java
int lcm(int a, int b) {
    return a * b / gcd(a, b);
}
```

使用位操作和减法求解最大公约数

编程之美：2.7

对于 a 和 b 的最大公约数 f(a, b)，有：

- 如果 a 和 b 均为偶数，f(a, b) = 2*f(a/2, b/2);
- 如果 a 是偶数 b 是奇数，f(a, b) = f(a/2, b);
- 如果 b 是偶数 a 是奇数，f(a, b) = f(a, b/2);
- 如果 a 和 b 均为奇数，f(a, b) = f(b, a-b);

乘 2 和除 2 都可以转换为移位操作。

```java
public int gcd(int a, int b) {
    if (a < b) {
        return gcd(b, a);
    }
    if (b == 0) {
        return a;
    }
    boolean isAEven = isEven(a), isBEven = isEven(b);
    if (isAEven && isBEven) {
        return 2 * gcd(a >> 1, b >> 1);
    } else if (isAEven && !isBEven) {
        return gcd(a >> 1, b);
    } else if (!isAEven && isBEven) {
        return gcd(a, b >> 1);
    } else {
        return gcd(b, a - b);
    }
}
```

进制转换

7 进制

504. Base 7 (Easy)

```java
public String convertToBase7(int num) {
    if (num == 0) {
        return "0";
    }
    StringBuilder sb = new StringBuilder();
    boolean isNegative = num < 0;
    if (isNegative) {
        num = -num;
    }
```

```
10.        while (num > 0) {
11.            sb.append(num % 7);
12.            num /= 7;
13.        }
14.        String ret = sb.reverse().toString();
15.        return isNegative ? "-" + ret : ret;
16.    }
```

Java 中 static String toString(int num, int radix) 可以将一个整数转换为 redix 进制表示的字符串。

```
1.    public String convertToBase7(int num) {
2.        return Integer.toString(num, 7);
3.    }
```

16 进制

405. Convert a Number to Hexadecimal (Easy)

```
1.    Input:
2.    26
3.
4.    Output:
5.    "1a"
6.
7.    Input:
8.    -1
9.
10.   Output:
11.   "ffffffff"
```

负数要用它的补码形式。

```
1.    public String toHex(int num) {
2.        char[] map = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a',
      'b', 'c', 'd', 'e', 'f'};
3.        if (num == 0) return "0";
4.        StringBuilder sb = new StringBuilder();
5.        while (num != 0) {
6.            sb.append(map[num & 0b1111]);
7.            num >>>= 4; // 因为考虑的是补码形式，因此符号位就不能有特殊的意义，需要
```

```
8.        }
9.        return sb.reverse().toString();
10.   }
```

26 进制

168. Excel Sheet Column Title (Easy)

```
1.    1 -> A
2.    2 -> B
3.    3 -> C
4.    ...
5.    26 -> Z
6.    27 -> AA
7.    28 -> AB
```

因为是从 1 开始计算的，而不是从 0 开始，因此需要对 n 执行 -1 操作。

```
1.    public String convertToTitle(int n) {
2.        if (n == 0) {
3.            return "";
4.        }
5.        n--;
6.        return convertToTitle(n / 26) + (char) (n % 26 + 'A');
7.    }
```

阶乘

统计阶乘尾部有多少个 0

172. Factorial Trailing Zeroes (Easy)

尾部的 0 由 2 * 5 得来，2 的数量明显多于 5 的数量，因此只要统计有多少个 5 即可。

对于一个数 N，它所包含 5 的个数为：$N/5 + N/5^2 + N/5^3 + ...$，其中 N/5 表示不大于 N 的数中 5 的倍数贡献一个 5，$N/5^2$ 表示不大于 N 的数中 5^2 的倍数再贡献一个 5 ....。

```
1.    public int trailingZeroes(int n) {
```

```
2.        return n == 0 ? 0 : n / 5 + trailingZeroes(n / 5);
3.    }
```

如果统计的是 N! 的二进制表示中最低位 1 的位置，只要统计有多少个 2 即可，该题目出自
编程之美：2.2 。和求解有多少个 5 一样，2 的个数为 N/2 + N/2^2 + N/2^3 + ...

字符串加法减法

二进制加法

67. Add Binary (Easy)

```
1.    a = "11"
2.    b = "1"
3.    Return "100".
```

```
1.    public String addBinary(String a, String b) {
2.        int i = a.length() - 1, j = b.length() - 1, carry = 0;
3.        StringBuilder str = new StringBuilder();
4.        while (carry == 1 || i >= 0 || j >= 0) {
5.            if (i >= 0 && a.charAt(i--) == '1') {
6.                carry++;
7.            }
8.            if (j >= 0 && b.charAt(j--) == '1') {
9.                carry++;
10.           }
11.           str.append(carry % 2);
12.           carry /= 2;
13.       }
14.       return str.reverse().toString();
15.   }
```

字符串加法

415. Add Strings (Easy)

字符串的值为非负整数。

```
1.    public String addStrings(String num1, String num2) {
```

```
  2.        StringBuilder str = new StringBuilder();
  3.        int carry = 0, i = num1.length() - 1, j = num2.length() - 1;
  4.        while (carry == 1 || i >= 0 || j >= 0) {
  5.            int x = i < 0 ? 0 : num1.charAt(i--) - '0';
  6.            int y = j < 0 ? 0 : num2.charAt(j--) - '0';
  7.            str.append((x + y + carry) % 10);
  8.            carry = (x + y + carry) / 10;
  9.        }
 10.        return str.reverse().toString();
 11.    }
```

相遇问题

改变数组元素使所有的数组元素都相等

[462. Minimum Moves to Equal Array Elements II (Medium)](#)

```
  1.    Input:
  2.    [1,2,3]
  3.
  4.    Output:
  5.    2
  6.
  7.    Explanation:
  8.    Only two moves are needed (remember each move increments or decrements
       one element):
  9.
 10.    [1,2,3]  =>  [2,2,3]  =>  [2,2,2]
```

每次可以对一个数组元素加一或者减一，求最小的改变次数。

这是个典型的相遇问题，移动距离最小的方式是所有元素都移动到中位数。理由如下：

设 m 为中位数。a 和 b 是 m 两边的两个元素，且 b > a。要使 a 和 b 相等，它们总共移动的次数为 b - a，这个值等于 (b - m) + (m - a)，也就是把这两个数移动到中位数的移动次数。

设数组长度为 N，则可以找到 N/2 对 a 和 b 的组合，使它们都移动到 m 的位置。

解法 1

先排序，时间复杂度：O(NlogN)

```java
public int minMoves2(int[] nums) {
    Arrays.sort(nums);
    int move = 0;
    int l = 0, h = nums.length - 1;
    while (l <= h) {
        move += nums[h] - nums[l];
        l++;
        h--;
    }
    return move;
}
```

解法 2

使用快速选择找到中位数，时间复杂度 O(N)

```java
public int minMoves2(int[] nums) {
    int move = 0;
    int median = findKthSmallest(nums, nums.length / 2);
    for (int num : nums) {
        move += Math.abs(num - median);
    }
    return move;
}

private int findKthSmallest(int[] nums, int k) {
    int l = 0, h = nums.length - 1;
    while (l < h) {
        int j = partition(nums, l, h);
        if (j == k) {
            break;
        }
        if (j < k) {
            l = j + 1;
        } else {
            h = j - 1;
        }
    }
    return nums[k];
}
```

```
26.    private int partition(int[] nums, int l, int h) {
27.        int i = l, j = h + 1;
28.        while (true) {
29.            while (nums[++i] < nums[l] && i < h) ;
30.            while (nums[--j] > nums[l] && j > l) ;
31.            if (i >= j) {
32.                break;
33.            }
34.            swap(nums, i, j);
35.        }
36.        swap(nums, l, j);
37.        return j;
38.    }
39.
40.    private void swap(int[] nums, int i, int j) {
41.        int tmp = nums[i];
42.        nums[i] = nums[j];
43.        nums[j] = tmp;
44.    }
```

多数投票问题

数组中出现次数多于 n / 2 的元素

169. Majority Element (Easy)

先对数组排序，最中间那个数出现次数一定多于 n / 2。

```
1.    public int majorityElement(int[] nums) {
2.        Arrays.sort(nums);
3.        return nums[nums.length / 2];
4.    }
```

可以利用 Boyer-Moore Majority Vote Algorithm 来解决这个问题，使得时间复杂度为 O(N)。可以这么理解该算法：使用 cnt 来统计一个元素出现的次数，当遍历到的元素和统计元素不相等时，令 cnt--。如果前面查找了 i 个元素，且 cnt == 0，说明前 i 个元素没有 majority，或者有 majority，但是出现的次数少于 i / 2，因为如果多于 i / 2 的话 cnt 就一定不会为 0。此时剩下的 n - i 个元素中，majority 的数目依然多于 (n - i) / 2，因此继续查找就能找出 majority。

```java
public int majorityElement(int[] nums) {
    int cnt = 0, majority = nums[0];
    for (int num : nums) {
        majority = (cnt == 0) ? num : majority;
        cnt = (majority == num) ? cnt + 1 : cnt - 1;
    }
    return majority;
}
```

其它

平方数

367. Valid Perfect Square (Easy)

```
Input: 16
Returns: True
```

平方序列：1,4,9,16,..

间隔：3,5,7,...

间隔为等差数列，使用这个特性可以得到从 1 开始的平方序列。

```java
public boolean isPerfectSquare(int num) {
    int subNum = 1;
    while (num > 0) {
        num -= subNum;
        subNum += 2;
    }
    return num == 0;
}
```

3 的 n 次方

326. Power of Three (Easy)

```java
public boolean isPowerOfThree(int n) {
    return n > 0 && (1162261467 % n == 0);
```

```
3.   }
```

乘积数组

238. Product of Array Except Self (Medium)

```
1.   For example, given [1,2,3,4], return [24,12,8,6].
```

给定一个数组，创建一个新数组，新数组的每个元素为原始数组中除了该位置上的元素之外所有元素的乘积。

要求时间复杂度为 O(N)，并且不能使用除法。

```java
public int[] productExceptSelf(int[] nums) {
    int n = nums.length;
    int[] products = new int[n];
    Arrays.fill(products, 1);
    int left = 1;
    for (int i = 1; i < n; i++) {
        left *= nums[i - 1];
        products[i] *= left;
    }
    int right = 1;
    for (int i = n - 2; i >= 0; i--) {
        right *= nums[i + 1];
        products[i] *= right;
    }
    return products;
}
```

找出数组中的乘积最大的三个数

628. Maximum Product of Three Numbers (Easy)

```
1.   Input: [1,2,3,4]
2.   Output: 24
```

```java
public int maximumProduct(int[] nums) {
    int max1 = Integer.MIN_VALUE, max2 = Integer.MIN_VALUE, max3 = Inte
```

```java
ger.MIN_VALUE, min1 = Integer.MAX_VALUE, min2 = Integer.MAX_VALUE;
    for (int n : nums) {
        if (n > max1) {
            max3 = max2;
            max2 = max1;
            max1 = n;
        } else if (n > max2) {
            max3 = max2;
            max2 = n;
        } else if (n > max3) {
            max3 = n;
        }

        if (n < min1) {
            min2 = min1;
            min1 = n;
        } else if (n < min2) {
            min2 = n;
        }
    }
    return Math.max(max1*max2*max3, max1*min1*min2);
}
```

数据结构相关

栈和队列

用栈实现队列

232. Implement Queue using Stacks (Easy)

```java
class MyQueue {

    private Stack<Integer> in = new Stack<>();
    private Stack<Integer> out = new Stack<>();

    public void push(int x) {
        in.push(x);
    }
```

```
 9.
10.        public int pop() {
11.            in2out();
12.            return out.pop();
13.        }
14.
15.        public int peek() {
16.            in2out();
17.            return out.peek();
18.        }
19.
20.        private void in2out() {
21.            if (out.isEmpty()) {
22.                while (!in.isEmpty()) {
23.                    out.push(in.pop());
24.                }
25.            }
26.        }
27.
28.        public boolean empty() {
29.            return in.isEmpty() && out.isEmpty();
30.        }
31.    }
```

用队列实现栈

225. Implement Stack using Queues (Easy)

在将一个元素 x 插入队列时，需要让除了 x 之外的所有元素出队列，再入队列。此时 x 在队首，第一个出队列，实现了后进先出顺序。

```
 1.    class MyStack {
 2.
 3.        private Queue<Integer> queue;
 4.
 5.        public MyStack() {
 6.            queue = new LinkedList<>();
 7.        }
 8.
 9.        public void push(int x) {
10.            queue.add(x);
11.            int cnt = queue.size();
12.            while (cnt-- > 1) {
```

```
13.            queue.add(queue.poll());
14.        }
15.    }
16.
17.    public int pop() {
18.        return queue.remove();
19.    }
20.
21.    public int top() {
22.        return queue.peek();
23.    }
24.
25.    public boolean empty() {
26.        return queue.isEmpty();
27.    }
28. }
```

最小值栈

155. Min Stack (Easy)

```
1.  class MinStack {
2.
3.      private Stack<Integer> dataStack;
4.      private Stack<Integer> minStack;
5.      private int min;
6.
7.      public MinStack() {
8.          dataStack = new Stack<>();
9.          minStack = new Stack<>();
10.         min = Integer.MAX_VALUE;
11.     }
12.
13.     public void push(int x) {
14.         dataStack.add(x);
15.         min = Math.min(min, x);
16.         minStack.add(min);
17.     }
18.
19.     public void pop() {
20.         dataStack.pop();
21.         minStack.pop();
22.         min = minStack.isEmpty() ? Integer.MAX_VALUE : minStack.peek();
23.     }
```

```
24.
25.      public int top() {
26.          return dataStack.peek();
27.      }
28.
29.      public int getMin() {
30.          return minStack.peek();
31.      }
32.  }
```

对于实现最小值队列问题，可以先将队列使用栈来实现，然后就将问题转换为最小值栈，这个问题出现在 编程之美：3.7。

用栈实现括号匹配

20. Valid Parentheses (Easy)

```
1.   "()[]{}"
2.
3.   Output : true
```

```
1.   public boolean isValid(String s) {
2.       Stack<Character> stack = new Stack<>();
3.       for (char c : s.toCharArray()) {
4.           if (c == '(' || c == '{' || c == '[') {
5.               stack.push(c);
6.           } else {
7.               if (stack.isEmpty()) {
8.                   return false;
9.               }
10.              char cStack = stack.pop();
11.              boolean b1 = c == ')' && cStack != '(';
12.              boolean b2 = c == ']' && cStack != '[';
13.              boolean b3 = c == '}' && cStack != '{';
14.              if (b1 || b2 || b3) {
15.                  return false;
16.              }
17.          }
18.      }
19.      return stack.isEmpty();
20.  }
```

数组中元素与下一个比它大的元素之间的距离

739. Daily Temperatures (Medium)

```
1.    Input: [73, 74, 75, 71, 69, 72, 76, 73]
2.    Output: [1, 1, 4, 2, 1, 1, 0, 0]
```

在遍历数组时用 Stack 把数组中的数存起来，如果当前遍历的数比栈顶元素来的大，说明栈顶元素的下一个比它大的数就是当前元素。

```java
1.    public int[] dailyTemperatures(int[] temperatures) {
2.        int n = temperatures.length;
3.        int[] dist = new int[n];
4.        Stack<Integer> indexs = new Stack<>();
5.        for (int curIndex = 0; curIndex < n; curIndex++) {
6.            while (!indexs.isEmpty() && temperatures[curIndex] > temperatures[indexs.peek()]) {
7.                int preIndex = indexs.pop();
8.                dist[preIndex] = curIndex - preIndex;
9.            }
10.           indexs.add(curIndex);
11.       }
12.       return dist;
13.   }
```

循环数组中比当前元素大的下一个元素

503. Next Greater Element II (Medium)

```
1.    Input: [1,2,1]
2.    Output: [2,-1,2]
3.    Explanation: The first 1's next greater number is 2;
4.    The number 2 can't find next greater number;
5.    The second 1's next greater number needs to search circularly, which is also 2.
```

与 739. Daily Temperatures (Medium) 不同的是，数组是循环数组，并且最后要求的不是距离而是下一个元素。

```java
public int[] nextGreaterElements(int[] nums) {
    int n = nums.length;
    int[] next = new int[n];
    Arrays.fill(next, -1);
    Stack<Integer> pre = new Stack<>();
    for (int i = 0; i < n * 2; i++) {
        int num = nums[i % n];
        while (!pre.isEmpty() && nums[pre.peek()] < num) {
            next[pre.pop()] = num;
        }
        if (i < n){
            pre.push(i);
        }
    }
    return next;
}
```

哈希表

哈希表使用 O(N) 空间复杂度存储数据，从而能够以 O(1) 时间复杂度求解问题。

Java 中的 **HashSet** 用于存储一个集合，可以查找元素是否在集合中。

如果元素有穷，并且范围不大，那么可以用一个布尔数组来存储一个元素是否存在。例如对于只有小写字符的元素，就可以用一个长度为 26 的布尔数组来存储一个字符集合，使得空间复杂度降低为 O(1)。

Java 中的 **HashMap** 主要用于映射关系，从而把两个元素联系起来。

在对一个内容进行压缩或者其它转换时，利用 HashMap 可以把原始内容和转换后的内容联系起来。例如在一个简化 url 的系统中Leetcdoe : 535. Encode and Decode TinyURL (Medium)，利用 HashMap 就可以存储精简后的 url 到原始 url 的映射，使得不仅可以显示简化的 url，也可以根据简化的 url 得到原始 url 从而定位到正确的资源。

HashMap 也可以用来对元素进行计数统计，此时键为元素，值为计数。和 HashSet 类似，如果元素有穷并且范围不大，可以用整型数组来进行统计。

数组中的两个数和为给定值

1. Two Sum (Easy)

可以先对数组进行排序，然后使用双指针方法或者二分查找方法。这样做的时间复杂度为 O(NlogN)，空间复杂度为 O(1)。

用 HashMap 存储数组元素和索引的映射，在访问到 nums[i] 时，判断 HashMap 中是否存在 target - nums[i]，如果存在说明 target - nums[i] 所在的索引和 i 就是要找的两个数。该方法的时间复杂度为 O(N)，空间复杂度为 O(N)，使用空间来换取时间。

```java
public int[] twoSum(int[] nums, int target) {
    HashMap<Integer, Integer> indexForNum = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        if (indexForNum.containsKey(target - nums[i])) {
            return new int[]{indexForNum.get(target - nums[i]), i};
        } else {
            indexForNum.put(nums[i], i);
        }
    }
    return null;
}
```

判断数组是否含有重复元素

217. Contains Duplicate (Easy)

```java
public boolean containsDuplicate(int[] nums) {
    Set<Integer> set = new HashSet<>();
    for (int num : nums) {
        set.add(num);
    }
    return set.size() < nums.length;
}
```

最长和谐序列

594. Longest Harmonious Subsequence (Easy)

```
Input: [1,3,2,2,5,2,3,7]
Output: 5
```

```
3.    Explanation: The longest harmonious subsequence is [3,2,2,2,3].
```

和谐序列中最大数和最小数只差正好为 1，应该注意的是序列的元素不一定是数组的连续元素。

```java
public int findLHS(int[] nums) {
    Map<Integer, Integer> countForNum = new HashMap<>();
    for (int num : nums) {
        countForNum.put(num, countForNum.getOrDefault(num, 0) + 1);
    }
    int longest = 0;
    for (int num : countForNum.keySet()) {
        if (countForNum.containsKey(num + 1)) {
            longest = Math.max(longest, countForNum.get(num + 1) +
countForNum.get(num));
        }
    }
    return longest;
}
```

最长连续序列

128. Longest Consecutive Sequence (Hard)

```
Given [100, 4, 200, 1, 3, 2],
The longest consecutive elements sequence is [1, 2, 3, 4]. Return its
length: 4.
```

要求以 O(N) 的时间复杂度求解。

```java
public int longestConsecutive(int[] nums) {
    Map<Integer, Integer> countForNum = new HashMap<>();
    for (int num : nums) {
        countForNum.put(num, 1);
    }
    for (int num : nums) {
        forward(countForNum, num);
    }
    return maxCount(countForNum);
}
```

```
12.    private int forward(Map<Integer, Integer> countForNum, int num) {
13.        if (!countForNum.containsKey(num)) {
14.            return 0;
15.        }
16.        int cnt = countForNum.get(num);
17.        if (cnt > 1) {
18.            return cnt;
19.        }
20.        cnt = forward(countForNum, num + 1) + 1;
21.        countForNum.put(num, cnt);
22.        return cnt;
23.    }
24.
25.    private int maxCount(Map<Integer, Integer> countForNum) {
26.        int max = 0;
27.        for (int num : countForNum.keySet()) {
28.            max = Math.max(max, countForNum.get(num));
29.        }
30.        return max;
31.    }
```

字符串

两个字符串包含的字符是否完全相同

242. Valid Anagram (Easy)

```
1.    s = "anagram", t = "nagaram", return true.
2.    s = "rat", t = "car", return false.
```

字符串只包含小写字符，总共有 26 个小写字符。可以用 HashMap 来映射字符与出现次数。因为键的范围很小，因此可以使用长度为 26 的整型数组对字符串出现的字符进行统计，然后比较两个字符串出现的字符数量是否相同。

```
1.    public boolean isAnagram(String s, String t) {
2.        int[] cnts = new int[26];
3.        for (char c : s.toCharArray()) {
4.            cnts[c - 'a']++;
5.        }
6.        for (char c : t.toCharArray()) {
```

```
7.            cnts[c - 'a']--;
8.        }
9.        for (int cnt : cnts) {
10.            if (cnt != 0) {
11.                return false;
12.            }
13.        }
14.        return true;
15.    }
```

计算一组字符集合可以组成的回文字符串的最大长度

409. Longest Palindrome (Easy)

```
1.    Input : "abccccdd"
2.    Output : 7
3.    Explanation : One longest palindrome that can be built is "dccaccd", w
      hose length is 7.
```

使用长度为 256 的整型数组来统计每个字符出现的个数，每个字符有偶数个可以用来构成回文字符串。

因为回文字符串最中间的那个字符可以单独出现，所以如果有单独的字符就把它放到最中间。

```
1.    public int longestPalindrome(String s) {
2.        int[] cnts = new int[256];
3.        for (char c : s.toCharArray()) {
4.            cnts[c]++;
5.        }
6.        int palindrome = 0;
7.        for (int cnt : cnts) {
8.            palindrome += (cnt / 2) * 2;
9.        }
10.        if (palindrome < s.length()) {
11.            palindrome++;    // 这个条件下 s 中一定有单个未使用的字符存在，可以把这
               个字符放到回文的最中间
12.        }
13.        return palindrome;
14.    }
```

字符串同构

205. Isomorphic Strings (Easy)

```
1.  Given "egg", "add", return true.
2.  Given "foo", "bar", return false.
3.  Given "paper", "title", return true.
```

记录一个字符上次出现的位置，如果两个字符串中的字符上次出现的位置一样，那么就属于同构。

```java
1.  public boolean isIsomorphic(String s, String t) {
2.      int[] preIndexOfS = new int[256];
3.      int[] preIndexOfT = new int[256];
4.      for (int i = 0; i < s.length(); i++) {
5.          char sc = s.charAt(i), tc = t.charAt(i);
6.          if (preIndexOfS[sc] != preIndexOfT[tc]) {
7.              return false;
8.          }
9.          preIndexOfS[sc] = i + 1;
10.         preIndexOfT[tc] = i + 1;
11.     }
12.     return true;
13. }
```

回文子字符串

647. Palindromic Substrings (Medium)

```
1.  Input: "aaa"
2.  Output: 6
3.  Explanation: Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa"
    .
```

从字符串的某一位开始，尝试着去扩展子字符串。

```java
1.  private int cnt = 0;
2.
3.  public int countSubstrings(String s) {
4.      for (int i = 0; i < s.length(); i++) {
5.          extendSubstrings(s, i, i);     // 奇数长度
6.          extendSubstrings(s, i, i + 1); // 偶数长度
```

```
  7.           }
  8.       return cnt;
  9.   }
 10.
 11.   private void extendSubstrings(String s, int start, int end) {
 12.       while (start >= 0 && end < s.length() && s.charAt(start) == s.charA
       t(end)) {
 13.           start--;
 14.           end++;
 15.           cnt++;
 16.       }
 17.   }
```

判断一个整数是否是回文数

9. Palindrome Number (Easy)

要求不能使用额外空间，也就不能将整数转换为字符串进行判断。

将整数分成左右两部分，右边那部分需要转置，然后判断这两部分是否相等。

```
  1.   public boolean isPalindrome(int x) {
  2.       if (x == 0) {
  3.           return true;
  4.       }
  5.       if (x < 0 || x % 10 == 0) {
  6.           return false;
  7.       }
  8.       int right = 0;
  9.       while (x > right) {
 10.           right = right * 10 + x % 10;
 11.           x /= 10;
 12.       }
 13.       return x == right || x == right / 10;
 14.   }
```

统计二进制字符串中连续 1 和连续 0 数量相同的子字符串个数

696. Count Binary Substrings (Easy)

```
  1.   Input: "00110011"
```

```
2.    Output: 6
3.    Explanation: There are 6 substrings that have equal number of
      consecutive 1's and 0's: "0011", "01", "1100", "10", "0011", and "01".
```

```java
1.  public int countBinarySubstrings(String s) {
2.      int preLen = 0, curLen = 1, count = 0;
3.      for (int i = 1; i < s.length(); i++) {
4.          if (s.charAt(i) == s.charAt(i - 1)) {
5.              curLen++;
6.          } else {
7.              preLen = curLen;
8.              curLen = 1;
9.          }
10.
11.         if (preLen >= curLen) {
12.             count++;
13.         }
14.     }
15.     return count;
16. }
```

字符串循环移位包含

编程之美：3.1

```
1.  s1 = AABCD, s2 = CDAA
2.  Return : true
```

给定两个字符串 s1 和 s2，要求判定 s2 是否能够被 s1 做循环移位得到的字符串包含。

s1 进行循环移位的结果是 s1s1 的子字符串，因此只要判断 s2 是否是 s1s1 的子字符串即可。

字符串循环移位

编程之美：2.17

```
1.  s = "abcd123" k = 3
2.  Return "123abcd"
```

将字符串向右循环移动 k 位。

将 abcd123 中的 abcd 和 123 单独逆序，得到 dcba321，然后对整个字符串进行逆序，得到 123abcd。

字符串中单词的翻转

程序员代码面试指南

```
1.  s = "I am a student"
2.  return "student a am I"
```

将每个单词逆序，然后将整个字符串逆序。

数组与矩阵

把数组中的 0 移到末尾

283. Move Zeroes (Easy)

```
1.  For example, given nums = [0, 1, 0, 3, 12], after calling your
    function, nums should be [1, 3, 12, 0, 0].
```

```java
1.  public void moveZeroes(int[] nums) {
2.      int idx = 0;
3.      for (int num : nums) {
4.          if (num != 0) {
5.              nums[idx++] = num;
6.          }
7.      }
8.      while (idx < nums.length) {
9.          nums[idx++] = 0;
10.     }
11. }
```

改变矩阵维度

566. Reshape the Matrix (Easy)

```
1.    Input:
2.    nums =
3.    [[1,2],
4.     [3,4]]
5.    r = 1, c = 4
6.
7.    Output:
8.    [[1,2,3,4]]
9.
10.   Explanation:
11.   The row-traversing of nums is [1,2,3,4]. The new reshaped matrix is a
      1 * 4 matrix, fill it row by row by using the previous list.
```

```java
1.    public int[][] matrixReshape(int[][] nums, int r, int c) {
2.        int m = nums.length, n = nums[0].length;
3.        if (m * n != r * c) {
4.            return nums;
5.        }
6.        int[][] reshapedNums = new int[r][c];
7.        int index = 0;
8.        for (int i = 0; i < r; i++) {
9.            for (int j = 0; j < c; j++) {
10.               reshapedNums[i][j] = nums[index / n][index % n];
11.               index++;
12.           }
13.       }
14.       return reshapedNums;
15.   }
```

找出数组中最长的连续 1

485. Max Consecutive Ones (Easy)

```java
1.    public int findMaxConsecutiveOnes(int[] nums) {
2.        int max = 0, cur = 0;
3.        for (int x : nums) {
4.            cur = x == 0 ? 0 : cur + 1;
5.            max = Math.max(max, cur);
6.        }
7.        return max;
```

```
8.    }
```

一个数组元素在 [1, n] 之间，其中一个数被替换为另一个数，找出重复的数和丢失的数

645. Set Mismatch (Easy)

```
1.    Input: nums = [1,2,2,4]
2.    Output: [2,3]
```

```
1.    Input: nums = [1,2,2,4]
2.    Output: [2,3]
```

最直接的方法是先对数组进行排序，这种方法时间复杂度为 O(NlogN)。本题可以以 O(N) 的时间复杂度、O(1) 空间复杂度来求解。

主要思想是通过交换数组元素，使得数组上的元素在正确的位置上。遍历数组，如果第 i 位上的元素不是 i + 1，那么一直交换第 i 位和 nums[i] - 1 位置上的元素。

```java
1.    public int[] findErrorNums(int[] nums) {
2.        for (int i = 0; i < nums.length; i++) {
3.            while (nums[i] != i + 1 && nums[nums[i] - 1] != nums[i]) {
4.                swap(nums, i, nums[i] - 1);
5.            }
6.        }
7.        for (int i = 0; i < nums.length; i++) {
8.            if (nums[i] != i + 1) {
9.                return new int[]{nums[i], i + 1};
10.           }
11.       }
12.       return null;
13.   }
14.
15.   private void swap(int[] nums, int i, int j) {
16.       int tmp = nums[i];
17.       nums[i] = nums[j];
18.       nums[j] = tmp;
19.   }
```

类似题目：

- 448. Find All Numbers Disappeared in an Array (Easy)，寻找所有丢失的元素
- 442. Find All Duplicates in an Array (Medium)，寻找所有重复的元素。

找出数组中重复的数，数组值在 [1, n] 之间

287. Find the Duplicate Number (Medium)

要求不能修改数组，也不能使用额外的空间。

二分查找解法：

```java
public int findDuplicate(int[] nums) {
    int l = 1, h = nums.length - 1;
    while (l <= h) {
        int mid = l + (h - l) / 2;
        int cnt = 0;
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] <= mid) cnt++;
        }
        if (cnt > mid) h = mid - 1;
        else l = mid + 1;
    }
    return l;
}
```

双指针解法，类似于有环链表中找出环的入口：

```java
public int findDuplicate(int[] nums) {
    int slow = nums[0], fast = nums[nums[0]];
    while (slow != fast) {
        slow = nums[slow];
        fast = nums[nums[fast]];
    }
    fast = 0;
    while (slow != fast) {
        slow = nums[slow];
        fast = nums[fast];
    }
    return slow;
}
```

有序矩阵查找

240. Search a 2D Matrix II (Medium)

```
1.    [
2.      [ 1,  5,  9],
3.      [10, 11, 13],
4.      [12, 13, 15]
5.    ]
```

```java
1.   public boolean searchMatrix(int[][] matrix, int target) {
2.       if (matrix == null || matrix.length == 0 || matrix[0].length == 0)
     return false;
3.       int m = matrix.length, n = matrix[0].length;
4.       int row = 0, col = n - 1;
5.       while (row < m && col >= 0) {
6.           if (target == matrix[row][col]) return true;
7.           else if (target < matrix[row][col]) col--;
8.           else row++;
9.       }
10.      return false;
11.  }
```

有序矩阵的 Kth Element

378. Kth Smallest Element in a Sorted Matrix ((Medium))

```
1.   matrix = [
2.     [ 1,  5,  9],
3.     [10, 11, 13],
4.     [12, 13, 15]
5.   ],
6.   k = 8,
7.
8.   return 13.
```

解题参考：Share my thoughts and Clean Java Code

二分查找解法：

```
1.   public int kthSmallest(int[][] matrix, int k) {
2.       int m = matrix.length, n = matrix[0].length;
3.       int lo = matrix[0][0], hi = matrix[m - 1][n - 1];
4.       while(lo <= hi) {
5.           int mid = lo + (hi - lo) / 2;
6.           int cnt = 0;
7.           for(int i = 0; i < m; i++) {
8.               for(int j = 0; j < n && matrix[i][j] <= mid; j++) {
9.                   cnt++;
10.              }
11.          }
12.          if(cnt < k) lo = mid + 1;
13.          else hi = mid - 1;
14.      }
15.      return lo;
16.  }
```

堆解法：

```
1.   public int kthSmallest(int[][] matrix, int k) {
2.       int m = matrix.length, n = matrix[0].length;
3.       PriorityQueue<Tuple> pq = new PriorityQueue<Tuple>();
4.       for(int j = 0; j < n; j++) pq.offer(new Tuple(0, j, matrix[0][j]));
5.       for(int i = 0; i < k - 1; i++) { // 小根堆，去掉 k - 1 个堆顶元素，此时
         堆顶元素就是第 k 的数
6.           Tuple t = pq.poll();
7.           if(t.x == m - 1) continue;
8.           pq.offer(new Tuple(t.x + 1, t.y, matrix[t.x + 1][t.y]));
9.       }
10.      return pq.poll().val;
11.  }
12.
13.  class Tuple implements Comparable<Tuple> {
14.      int x, y, val;
15.      public Tuple(int x, int y, int val) {
16.          this.x = x; this.y = y; this.val = val;
17.      }
18.
19.      @Override
20.      public int compareTo(Tuple that) {
21.          return this.val - that.val;
22.      }
23.  }
```

数组相邻差值的个数

667. Beautiful Arrangement II (Medium)

```
1.    Input: n = 3, k = 2
2.    Output: [1, 3, 2]
3.    Explanation: The [1, 3, 2] has three different positive integers rangi
      ng from 1 to 3, and the [2, 1] has exactly 2 distinct integers: 1 and
      2.
```

题目描述：数组元素为 1\~n 的整数，要求构建数组，使得相邻元素的差值不相同的个数为 k。

让前 k+1 个元素构建出 k 个不相同的差值，序列为：1 k+1 2 k 3 k-1 ... k/2 k/2+1.

```
1.    public int[] constructArray(int n, int k) {
2.        int[] ret = new int[n];
3.        ret[0] = 1;
4.        for (int i = 1, interval = k; i <= k; i++, interval--) {
5.            ret[i] = i % 2 == 1 ? ret[i - 1] + interval : ret[i - 1] - inter
      val;
6.        }
7.        for (int i = k + 1; i < n; i++) {
8.            ret[i] = i + 1;
9.        }
10.       return ret;
11.   }
```

数组的度

697. Degree of an Array (Easy)

```
1.    Input: [1,2,2,3,1,4,2]
2.    Output: 6
```

题目描述：数组的度定义为元素出现的最高频率，例如上面的数组度为 3。要求找到一个最小的子数组，这个子数组的度和原数组一样。

```
1.    public int findShortestSubArray(int[] nums) {
```

```
2.          Map<Integer, Integer> numsCnt = new HashMap<>();
3.          Map<Integer, Integer> numsLastIndex = new HashMap<>();
4.          Map<Integer, Integer> numsFirstIndex = new HashMap<>();
5.          for (int i = 0; i < nums.length; i++) {
6.              int num = nums[i];
7.              numsCnt.put(num, numsCnt.getOrDefault(num, 0) + 1);
8.              numsLastIndex.put(num, i);
9.              if (!numsFirstIndex.containsKey(num)) {
10.                 numsFirstIndex.put(num, i);
11.             }
12.         }
13.         int maxCnt = 0;
14.         for (int num : nums) {
15.             maxCnt = Math.max(maxCnt, numsCnt.get(num));
16.         }
17.         int ret = nums.length;
18.         for (int i = 0; i < nums.length; i++) {
19.             int num = nums[i];
20.             int cnt = numsCnt.get(num);
21.             if (cnt != maxCnt) continue;
22.             ret = Math.min(ret, numsLastIndex.get(num) - numsFirstIndex.get
    (num) + 1);
23.         }
24.         return ret;
25.     }
```

对角元素相等的矩阵

766. Toeplitz Matrix (Easy)

```
1.   1234
2.   5123
3.   9512
4.
5.   In the above grid, the diagonals are "[9]", "[5, 5]", "[1, 1, 1]", "[2
     , 2, 2]", "[3, 3]", "[4]", and in each diagonal all elements are the s
     ame, so the answer is True.
```

```
1.   public boolean isToeplitzMatrix(int[][] matrix) {
2.       for (int i = 0; i < matrix[0].length; i++) {
3.           if (!check(matrix, matrix[0][i], 0, i)) {
4.               return false;
5.           }
```

```
6.            }
7.        for (int i = 0; i < matrix.length; i++) {
8.            if (!check(matrix, matrix[i][0], i, 0)) {
9.                return false;
10.           }
11.       }
12.       return true;
13.   }
14.
15.   private boolean check(int[][] matrix, int expectValue, int row, int col) {
16.       if (row >= matrix.length || col >= matrix[0].length) {
17.           return true;
18.       }
19.       if (matrix[row][col] != expectValue) {
20.           return false;
21.       }
22.       return check(matrix, expectValue, row + 1, col + 1);
23.   }
```

嵌套数组

565. Array Nesting (Medium)

```
1.    Input: A = [5,4,0,3,1,6,2]
2.    Output: 4
3.    Explanation:
4.    A[0] = 5, A[1] = 4, A[2] = 0, A[3] = 3, A[4] = 1, A[5] = 6, A[6] = 2.
5.
6.    One of the longest S[K]:
7.    S[0] = {A[0], A[5], A[6], A[2]} = {5, 6, 2, 0}
```

题目描述：S[i] 表示一个集合，集合的第一个元素是 A[i]，第二个元素是 A[A[i]]，如此嵌套下去。求最大的 S[i]。

```
1.    public int arrayNesting(int[] nums) {
2.        int max = 0;
3.        for (int i = 0; i < nums.length; i++) {
4.            int cnt = 0;
5.            for (int j = i; nums[j] != -1; ) {
6.                cnt++;
7.                int t = nums[j];
```

```
  8.              nums[j] = -1; // 标记该位置已经被访问
  9.              j = t;
 10.
 11.          }
 12.          max = Math.max(max, cnt);
 13.      }
 14.      return max;
 15.  }
```

分隔数组

769. Max Chunks To Make Sorted (Medium)

```
  1.  Input: arr = [1,0,2,3,4]
  2.  Output: 4
  3.  Explanation:
  4.  We can split into two chunks, such as [1, 0], [2, 3, 4].
  5.  However, splitting into [1, 0], [2], [3], [4] is the highest number of
      chunks possible.
```

题目描述：分隔数组，使得对每部分排序后数组就为有序。

```
  1.  public int maxChunksToSorted(int[] arr) {
  2.      if (arr == null) return 0;
  3.      int ret = 0;
  4.      int right = arr[0];
  5.      for (int i = 0; i < arr.length; i++) {
  6.          right = Math.max(right, arr[i]);
  7.          if (right == i) ret++;
  8.      }
  9.      return ret;
 10.  }
```

链表

链表是空节点，或者有一个值和一个指向下一个链表的指针，因此很多链表问题可以用递归来处理。

找出两个链表的交点

160. Intersection of Two Linked Lists (Easy)

```
A:          a1 → a2
                    ↘
                      c1 → c2 → c3
                    ↗
B:    b1 → b2 → b3
```

要求：时间复杂度为 O(N)，空间复杂度为 O(1)

设 A 的长度为 a + c，B 的长度为 b + c，其中 c 为尾部公共部分长度，可知 a + c + b = b + c + a。

当访问 A 链表的指针访问到链表尾部时，令它从链表 B 的头部开始访问链表 B；同样地，当访问 B 链表的指针访问到链表尾部时，令它从链表 A 的头部开始访问链表 A。这样就能控制访问 A 和 B 两个链表的指针能同时访问到交点。

```java
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    ListNode l1 = headA, l2 = headB;
    while (l1 != l2) {
        l1 = (l1 == null) ? headB : l1.next;
        l2 = (l2 == null) ? headA : l2.next;
    }
    return l1;
}
```

如果只是判断是否存在交点，那么就是另一个问题，即 编程之美：3.6 的问题。有两种解法：把第一个链表的结尾连接到第二个链表的开头，看第二个链表是否存在环；或者直接比较两个链表的最后一个节点是否相同。

链表反转

206. Reverse Linked List (Easy)

递归

```java
public ListNode reverseList(ListNode head) {
    if (head == null || head.next == null) {
```

```
 3.            return head;
 4.        }
 5.        ListNode next = head.next;
 6.        ListNode newHead = reverseList(next);
 7.        next.next = head;
 8.        head.next = null;
 9.        return newHead;
10.    }
```

头插法

```
 1.    public ListNode reverseList(ListNode head) {
 2.        ListNode newHead = new ListNode(-1);
 3.        while (head != null) {
 4.            ListNode next = head.next;
 5.            head.next = newHead.next;
 6.            newHead.next = head;
 7.            head = next;
 8.        }
 9.        return newHead.next;
10.    }
```

归并两个有序的链表

21. Merge Two Sorted Lists (Easy)

```
 1.    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
 2.        if (l1 == null) return l2;
 3.        if (l2 == null) return l1;
 4.        if (l1.val < l2.val) {
 5.            l1.next = mergeTwoLists(l1.next, l2);
 6.            return l1;
 7.        } else {
 8.            l2.next = mergeTwoLists(l1, l2.next);
 9.            return l2;
10.        }
11.    }
```

从有序链表中删除重复节点

83. Remove Duplicates from Sorted List (Easy)

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

```
1.    public ListNode deleteDuplicates(ListNode head) {
2.        if(head == null || head.next == null) return head;
3.        head.next = deleteDuplicates(head.next);
4.        return head.val == head.next.val ? head.next : head;
5.    }
```

删除链表的倒数第 n 个节点

19. Remove Nth Node From End of List (Medium)

```
1.    Given linked list: 1->2->3->4->5, and n = 2.
2.    After removing the second node from the end, the linked list becomes 1
      ->2->3->5.
```

```
1.    public ListNode removeNthFromEnd(ListNode head, int n) {
2.        ListNode fast = head;
3.        while (n-- > 0) {
4.            fast = fast.next;
5.        }
6.        if (fast == null) return head.next;
7.        ListNode slow = head;
8.        while (fast.next != null) {
9.            fast = fast.next;
10.           slow = slow.next;
11.       }
12.       slow.next = slow.next.next;
13.       return head;
14.   }
```

交换链表中的相邻结点

24. Swap Nodes in Pairs (Medium)

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

题目要求：不能修改结点的 val 值，O(1) 空间复杂度。

```
1.    public ListNode swapPairs(ListNode head) {
2.        ListNode node = new ListNode(-1);
3.        node.next = head;
4.        ListNode pre = node;
5.        while (pre.next != null && pre.next.next != null) {
6.            ListNode l1 = pre.next, l2 = pre.next.next;
7.            ListNode next = l2.next;
8.            l1.next = next;
9.            l2.next = l1;
10.           pre.next = l2;
11.
12.           pre = l1;
13.       }
14.       return node.next;
15.   }
```

链表求和

445. Add Two Numbers II (Medium)

```
1.    Input: (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)
2.    Output: 7 -> 8 -> 0 -> 7
```

题目要求：不能修改原始链表。

```
1.    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
2.        Stack<Integer> l1Stack = buildStack(l1);
3.        Stack<Integer> l2Stack = buildStack(l2);
4.        ListNode head = new ListNode(-1);
5.        int carry = 0;
6.        while (!l1Stack.isEmpty() || !l2Stack.isEmpty() || carry != 0) {
7.            int x = l1Stack.isEmpty() ? 0 : l1Stack.pop();
8.            int y = l2Stack.isEmpty() ? 0 : l2Stack.pop();
9.            int sum = x + y + carry;
10.           ListNode node = new ListNode(sum % 10);
11.           node.next = head.next;
12.           head.next = node;
13.           carry = sum / 10;
14.       }
15.       return head.next;
16.   }
17.
```

```
18.    private Stack<Integer> buildStack(ListNode l) {
19.        Stack<Integer> stack = new Stack<>();
20.        while (l != null) {
21.            stack.push(l.val);
22.            l = l.next;
23.        }
24.        return stack;
25.    }
```

回文链表

234. Palindrome Linked List (Easy)

题目要求：以 O(1) 的空间复杂度来求解。

切成两半，把后半段反转，然后比较两半是否相等。

```
1.    public boolean isPalindrome(ListNode head) {
2.        if (head == null || head.next == null) return true;
3.        ListNode slow = head, fast = head.next;
4.        while (fast != null && fast.next != null) {
5.            slow = slow.next;
6.            fast = fast.next.next;
7.        }
8.        if (fast != null) slow = slow.next;    // 偶数节点，让 slow 指向下一个节点
9.        cut(head, slow);                        // 切成两个链表
10.        return isEqual(head, reverse(slow));
11.    }
12.
13.    private void cut(ListNode head, ListNode cutNode) {
14.        while (head.next != cutNode) {
15.            head = head.next;
16.        }
17.        head.next = null;
18.    }
19.
20.    private ListNode reverse(ListNode head) {
21.        ListNode newHead = null;
22.        while (head != null) {
23.            ListNode nextNode = head.next;
24.            head.next = newHead;
25.            newHead = head;
```

```
26.            head = nextNode;
27.        }
28.        return newHead;
29.    }
30.
31.    private boolean isEqual(ListNode l1, ListNode l2) {
32.        while (l1 != null && l2 != null) {
33.            if (l1.val != l2.val) return false;
34.            l1 = l1.next;
35.            l2 = l2.next;
36.        }
37.        return true;
38.    }
```

分隔链表

725. Split Linked List in Parts(Medium)

```
1.   Input:
2.   root = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], k = 3
3.   Output: [[1, 2, 3, 4], [5, 6, 7], [8, 9, 10]]
4.   Explanation:
5.   The input has been split into consecutive parts with size difference a
     t most 1, and earlier parts are a larger size than the later parts.
```

题目描述：把链表分隔成 k 部分，每部分的长度都应该尽可能相同，排在前面的长度应该大于等于后面的。

```
1.   public ListNode[] splitListToParts(ListNode root, int k) {
2.       int N = 0;
3.       ListNode cur = root;
4.       while (cur != null) {
5.           N++;
6.           cur = cur.next;
7.       }
8.       int mod = N % k;
9.       int size = N / k;
10.      ListNode[] ret = new ListNode[k];
11.      cur = root;
12.      for (int i = 0; cur != null && i < k; i++) {
13.          ret[i] = cur;
14.          int curSize = size + (mod-- > 0 ? 1 : 0);
```

```
15.            for (int j = 0; j < curSize - 1; j++) {
16.                cur = cur.next;
17.            }
18.            ListNode next = cur.next;
19.            cur.next = null;
20.            cur = next;
21.        }
22.        return ret;
23.    }
```

链表元素按奇偶聚集

328. Odd Even Linked List (Medium)

```
1.    Example:
2.    Given 1->2->3->4->5->NULL,
3.    return 1->3->5->2->4->NULL.
```

```
1.    public ListNode oddEvenList(ListNode head) {
2.        if (head == null) {
3.            return head;
4.        }
5.        ListNode odd = head, even = head.next, evenHead = even;
6.        while (even != null && even.next != null) {
7.            odd.next = odd.next.next;
8.            odd = odd.next;
9.            even.next = even.next.next;
10.           even = even.next;
11.       }
12.       odd.next = evenHead;
13.       return head;
14.   }
```

树

递归

一棵树要么是空树，要么有两个指针，每个指针指向一棵树。树是一种递归结构，很多树的问题可以使用递归来处理。

树的高度

104. Maximum Depth of Binary Tree (Easy)

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

平衡树

110. Balanced Binary Tree (Easy)

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

平衡树左右子树高度差都小于等于 1

```
1.   private boolean result = true;
2.
3.   public boolean isBalanced(TreeNode root) {
4.       maxDepth(root);
5.       return result;
6.   }
7.
8.   public int maxDepth(TreeNode root) {
9.       if (root == null) return 0;
10.      int l = maxDepth(root.left);
11.      int r = maxDepth(root.right);
12.      if (Math.abs(l - r) > 1) result = false;
13.      return 1 + Math.max(l, r);
14.  }
```

两节点的最长路径

543. Diameter of Binary Tree (Easy)

```
1.    Input:
2.
3.             1
4.            / \
5.           2   3
6.          / \
7.         4   5
8.
9.    Return 3, which is the length of the path [4,2,1,3] or [5,2,1,3].
```

```
1.    private int max = 0;
2.
3.    public int diameterOfBinaryTree(TreeNode root) {
4.        depth(root);
5.        return max;
6.    }
7.
8.    private int depth(TreeNode root) {
9.        if (root == null) return 0;
10.       int leftDepth = depth(root.left);
11.       int rightDepth = depth(root.right);
12.       max = Math.max(max, leftDepth + rightDepth);
13.       return Math.max(leftDepth, rightDepth) + 1;
14.   }
```

翻转树

226. Invert Binary Tree (Easy)

```
1.    public TreeNode invertTree(TreeNode root) {
2.        if (root == null) return null;
3.        TreeNode left = root.left;  // 后面的操作会改变 left 指针, 因此先保存下来
4.        root.left = invertTree(root.right);
5.        root.right = invertTree(left);
6.        return root;
7.    }
```

归并两棵树

617. Merge Two Binary Trees (Easy)

```
1.  Input:
2.        Tree 1                     Tree 2
3.            1                           2
4.          / \                         / \
5.         3   2                       1   3
6.        /                               \   \
7.       5                                 4   7
8.  Output:
9.  Merged tree:
10.            3
11.          / \
12.         4   5
13.        / \   \
14.       5   4   7
```

```java
1.  public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {
2.      if (t1 == null && t2 == null) return null;
3.      if (t1 == null) return t2;
4.      if (t2 == null) return t1;
5.      TreeNode root = new TreeNode(t1.val + t2.val);
6.      root.left = mergeTrees(t1.left, t2.left);
7.      root.right = mergeTrees(t1.right, t2.right);
8.      return root;
9.  }
```

判断路径和是否等于一个数

Leetcdoe : 112. Path Sum (Easy)

```
1.  Given the below binary tree and sum = 22,
2.                5
3.               / \
4.              4   8
5.             /   / \
6.            11  13  4
7.           / \       \
8.          7   2       1
9.  return true, as there exist a root-to-leaf path 5->4->11->2 which sum
    is 22.
```

路径和定义为从 root 到 leaf 的所有节点的和

```
1.    public boolean hasPathSum(TreeNode root, int sum) {
2.        if (root == null) return false;
3.        if (root.left == null && root.right == null && root.val == sum) ret
    urn true;
4.        return hasPathSum(root.left, sum - root.val) || hasPathSum(root.rig
    ht, sum - root.val);
5.    }
```

统计路径和等于一个数的路径数量

437. Path Sum III (Easy)

```
1.    root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8
2.
3.         10
4.        /  \
5.       5   -3
6.      / \    \
7.     3   2   11
8.    / \    \
9.   3  -2   1
10.
11.   Return 3. The paths that sum to 8 are:
12.
13.   1.  5 -> 3
14.   2.  5 -> 2 -> 1
15.   3. -3 -> 11
```

路径不一定以 root 开头，也不一定以 leaf 结尾，但是必须连续。

```
1.    public int pathSum(TreeNode root, int sum) {
2.        if (root == null) return 0;
3.        int ret = pathSumStartWithRoot(root, sum) + pathSum(root.left, sum)
    + pathSum(root.right, sum);
4.        return ret;
5.    }
6.
7.    private int pathSumStartWithRoot(TreeNode root, int sum) {
8.        if (root == null) return 0;
9.        int ret = 0;
10.       if (root.val == sum) ret++;
11.       ret += pathSumStartWithRoot(root.left, sum - root.val) + pathSumSta
```

```
rtWithRoot(root.right, sum - root.val);
12.        return ret;
13.    }
```

子树

572. Subtree of Another Tree (Easy)

```
1.    Given tree s:
2.         3
3.        / \
4.       4   5
5.      / \
6.     1   2
7.
8.    Given tree t:
9.        4
10.      / \
11.     1   2
12.
13.    Return true, because t has the same structure and node values with a s
       ubtree of s.
14.
15.    Given tree s:
16.
17.         3
18.        / \
19.       4   5
20.      / \
21.     1   2
22.        /
23.       0
24.
25.    Given tree t:
26.        4
27.      / \
28.     1   2
29.
30.    Return false.
```

```
1.    public boolean isSubtree(TreeNode s, TreeNode t) {
2.        if (s == null) return false;
3.        return isSubtreeWithRoot(s, t) || isSubtree(s.left, t) || isSubtree
```

```
 4.        (s.right, t);
 5.    }
 6.
 7.    private boolean isSubtreeWithRoot(TreeNode s, TreeNode t) {
 8.        if (t == null && s == null) return true;
 9.        if (t == null || s == null) return false;
10.        if (t.val != s.val) return false;
11.        return isSubtreeWithRoot(s.left, t.left) && isSubtreeWithRoot(s.rig
           ht, t.right);
       }
```

树的对称

101. Symmetric Tree (Easy)

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

```
 1.    public boolean isSymmetric(TreeNode root) {
 2.        if (root == null) return true;
 3.        return isSymmetric(root.left, root.right);
 4.    }
 5.
 6.    private boolean isSymmetric(TreeNode t1, TreeNode t2) {
 7.        if (t1 == null && t2 == null) return true;
 8.        if (t1 == null || t2 == null) return false;
 9.        if (t1.val != t2.val) return false;
10.        return isSymmetric(t1.left, t2.right) && isSymmetric(t1.right, t2.l
           eft);
11.    }
```

最小路径

111. Minimum Depth of Binary Tree (Easy)

树的根节点到叶子节点的最小路径长度

```
1.    public int minDepth(TreeNode root) {
```

```
2.        if (root == null) return 0;
3.        int left = minDepth(root.left);
4.        int right = minDepth(root.right);
5.        if (left == 0 || right == 0) return left + right + 1;
6.        return Math.min(left, right) + 1;
7.    }
```

统计左叶子节点的和

404. Sum of Left Leaves (Easy)

```
1.        3
2.       / \
3.      9  20
4.        /  \
5.      15    7
6.
7.    There are two left leaves in the binary tree, with values 9 and 15 res
   pectively. Return 24.
```

```
1.  public int sumOfLeftLeaves(TreeNode root) {
2.      if (root == null) return 0;
3.      if (isLeaf(root.left)) return root.left.val + sumOfLeftLeaves(root.
   right);
4.      return sumOfLeftLeaves(root.left) + sumOfLeftLeaves(root.right);
5.  }
6.
7.  private boolean isLeaf(TreeNode node){
8.      if (node == null) return false;
9.      return node.left == null && node.right == null;
10. }
```

相同节点值的最大路径长度

687. Longest Univalue Path (Easy)

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

```
6.
7.   Output : 2
```

```
1.   private int path = 0;
2.
3.   public int longestUnivaluePath(TreeNode root) {
4.       dfs(root);
5.       return path;
6.   }
7.
8.   private int dfs(TreeNode root){
9.       if (root == null) return 0;
10.      int left = dfs(root.left);
11.      int right = dfs(root.right);
12.      int leftPath = root.left != null && root.left.val == root.val ? lef
   t + 1 : 0;
13.      int rightPath = root.right != null && root.right.val == root.val ?
   right + 1 : 0;
14.      path = Math.max(path, leftPath + rightPath);
15.      return Math.max(leftPath, rightPath);
16.  }
```

间隔遍历

337. House Robber III (Medium)

```
1.       3
2.      / \
3.     2   3
4.      \   \
5.       3   1
6.   Maximum amount of money the thief can rob = 3 + 3 + 1 = 7.
```

```
1.   public int rob(TreeNode root) {
2.       if (root == null) return 0;
3.       int val1 = root.val;
4.       if (root.left != null) val1 += rob(root.left.left) + rob(root.left
   .right);
5.       if (root.right != null) val1 += rob(root.right.left) + rob(root.rig
   ht.right);
6.       int val2 = rob(root.left) + rob(root.right);
7.       return Math.max(val1, val2);
```

```
8.    }
```

找出二叉树中第二小的节点

671. Second Minimum Node In a Binary Tree (Easy)

```
1.    Input:
2.       2
3.      / \
4.     2   5
5.        / \
6.       5   7
7.
8.    Output: 5
```

一个节点要么具有 0 个或 2 个子节点，如果有子节点，那么根节点是最小的节点。

```java
1.    public int findSecondMinimumValue(TreeNode root) {
2.        if (root == null) return -1;
3.        if (root.left == null && root.right == null) return -1;
4.        int leftVal = root.left.val;
5.        int rightVal = root.right.val;
6.        if (leftVal == root.val) leftVal = findSecondMinimumValue(root.left
);
7.        if (rightVal == root.val) rightVal = findSecondMinimumValue(root.ri
ght);
8.        if (leftVal != -1 && rightVal != -1) return Math.min(leftVal, right
Val);
9.        if (leftVal != -1) return leftVal;
10.       return rightVal;
11.   }
```

层次遍历

使用 BFS 进行层次遍历。不需要使用两个队列来分别存储当前层的节点和下一层的节点，因为在开始遍历一层的节点时，当前队列中的节点数就是当前层的节点数，只要控制遍历这么多节点数，就能保证这次遍历的都是当前层的节点。

一棵树每层节点的平均数

637. Average of Levels in Binary Tree (Easy)

```java
public List<Double> averageOfLevels(TreeNode root) {
    List<Double> ret = new ArrayList<>();
    if (root == null) return ret;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        int cnt = queue.size();
        double sum = 0;
        for (int i = 0; i < cnt; i++) {
            TreeNode node = queue.poll();
            sum += node.val;
            if (node.left != null) queue.add(node.left);
            if (node.right != null) queue.add(node.right);
        }
        ret.add(sum / cnt);
    }
    return ret;
}
```

得到左下角的节点

513. Find Bottom Left Tree Value (Easy)

```
Input:

        1
       / \
      2   3
     /   / \
    4   5   6
       /
      7

Output:
7
```

```java
public int findBottomLeftValue(TreeNode root) {
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    while (!queue.isEmpty()) {
```

```
 5.          root = queue.poll();
 6.          if (root.right != null) queue.add(root.right);
 7.          if (root.left != null) queue.add(root.left);
 8.      }
 9.      return root.val;
10.  }
```

前中后序遍历

```
1.       1
2.      / \
3.     2   3
4.    / \   \
5.   4   5   6
```

- 层次遍历顺序：[1 2 3 4 5 6]
- 前序遍历顺序：[1 2 4 5 3 6]
- 中序遍历顺序：[4 2 5 1 3 6]
- 后序遍历顺序：[4 5 2 6 3 1]

层次遍历使用 BFS 实现，利用的就是 BFS 一层一层遍历的特性；而前序、中序、后序遍历利用了 DFS 实现。

前序、中序、后序遍只是在对节点访问的顺序有一点不同，其它都相同。

① 前序

```
1.  void dfs(TreeNode root) {
2.      visit(root);
3.      dfs(root.left);
4.      dfs(root.right);
5.  }
```

② 中序

```
1.  void dfs(TreeNode root) {
2.      dfs(root.left);
3.      visit(root);
4.      dfs(root.right);
```

```
5.    }
```

③ 后序

```
1.    void dfs(TreeNode root) {
2.        dfs(root.left);
3.        dfs(root.right);
4.        visit(root);
5.    }
```

非递归实现二叉树的前序遍历

144. Binary Tree Preorder Traversal (Medium)

```
1.    public List<Integer> preorderTraversal(TreeNode root) {
2.        List<Integer> ret = new ArrayList<>();
3.        Stack<TreeNode> stack = new Stack<>();
4.        stack.push(root);
5.        while (!stack.isEmpty()) {
6.            TreeNode node = stack.pop();
7.            if (node == null) continue;
8.            ret.add(node.val);
9.            stack.push(node.right);   // 先右后左，保证左子树先遍历
10.           stack.push(node.left);
11.       }
12.       return ret;
13.   }
```

非递归实现二叉树的后序遍历

145. Binary Tree Postorder Traversal (Medium)

前序遍历为 root -> left -> right，后序遍历为 left -> right -> root，可以修改前序遍历成为 root -> right -> left，那么这个顺序就和后序遍历正好相反。

```
1.    public List<Integer> postorderTraversal(TreeNode root) {
2.        List<Integer> ret = new ArrayList<>();
3.        Stack<TreeNode> stack = new Stack<>();
4.        stack.push(root);
5.        while (!stack.isEmpty()) {
```

```
 6.            TreeNode node = stack.pop();
 7.            if (node == null) continue;
 8.            ret.add(node.val);
 9.            stack.push(node.left);
10.            stack.push(node.right);
11.        }
12.        Collections.reverse(ret);
13.        return ret;
14.    }
```

非递归实现二叉树的中序遍历

94. Binary Tree Inorder Traversal (Medium)

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

BST

二叉查找树（BST）：根节点大于等于左子树所有节点，小于等于右子树所有节点。

二叉查找树中序遍历有序。

修剪二叉查找树

669. Trim a Binary Search Tree (Easy)

```
1.   Input:
2.
3.        3
4.       / \
5.      0   4
6.       \
7.        2
8.       /
9.      1
10.
11.     L = 1
12.     R = 3
13.
14.   Output:
15.
16.          3
17.         /
18.        2
19.       /
20.      1
```

题目描述：只保留值在 L \~ R 之间的节点

```java
1.   public TreeNode trimBST(TreeNode root, int L, int R) {
2.       if (root == null) return null;
3.       if (root.val > R) return trimBST(root.left, L, R);
4.       if (root.val < L) return trimBST(root.right, L, R);
5.       root.left = trimBST(root.left, L, R);
6.       root.right = trimBST(root.right, L, R);
7.       return root;
8.   }
```

寻找二叉查找树的第 k 个元素

230. Kth Smallest Element in a BST (Medium)

中序遍历解法：

```java
1.   private int cnt = 0;
2.   private int val;
3.
4.   public int kthSmallest(TreeNode root, int k) {
```

```
5.        inOrder(root, k);
6.        return val;
7.    }
8.
9.    private void inOrder(TreeNode node, int k) {
10.       if (node == null) return;
11.       inOrder(node.left, k);
12.       cnt++;
13.       if (cnt == k) {
14.           val = node.val;
15.           return;
16.       }
17.       inOrder(node.right, k);
18.   }
```

递归解法：

```
1.    public int kthSmallest(TreeNode root, int k) {
2.        int leftCnt = count(root.left);
3.        if (leftCnt == k - 1) return root.val;
4.        if (leftCnt > k - 1) return kthSmallest(root.left, k);
5.        return kthSmallest(root.right, k - leftCnt - 1);
6.    }
7.
8.    private int count(TreeNode node) {
9.        if (node == null) return 0;
10.       return 1 + count(node.left) + count(node.right);
11.   }
```

把二叉查找树每个节点的值都加上比它大的节点的值

Convert BST to Greater Tree (Easy)

```
1.    Input: The root of a Binary Search Tree like this:
2.
3.              5
4.            /   \
5.           2     13
6.
7.    Output: The root of a Greater Tree like this:
8.
9.               18
```

```
10.                / \
11.               20   13
```

先遍历右子树。

```java
1.   private int sum = 0;
2.
3.   public TreeNode convertBST(TreeNode root) {
4.       traver(root);
5.       return root;
6.   }
7.
8.   private void traver(TreeNode node) {
9.       if (node == null) return;
10.      traver(node.right);
11.      sum += node.val;
12.      node.val = sum;
13.      traver(node.left);
14.  }
```
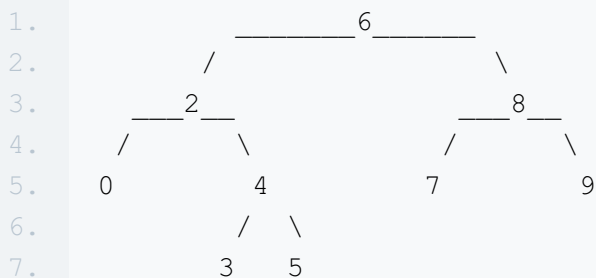
二叉查找树的最近公共祖先

235. Lowest Common Ancestor of a Binary Search Tree (Easy)

```
1.             _____6_____
2.            /              \
3.         ___2__          ___8__
4.        /      \        /      \
5.       0        4      7        9
6.               / \
7.              3   5
8.
9.   For example, the lowest common ancestor (LCA) of nodes 2 and 8 is 6. A
     nother example is LCA of nodes 2 and 4 is 2, since a node can be a des
     cendant of itself according to the LCA definition.
```

```java
1.   public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
     TreeNode q) {
2.       if (root.val > p.val && root.val > q.val) return
     lowestCommonAncestor(root.left, p, q);
3.       if (root.val < p.val && root.val < q.val) return
```
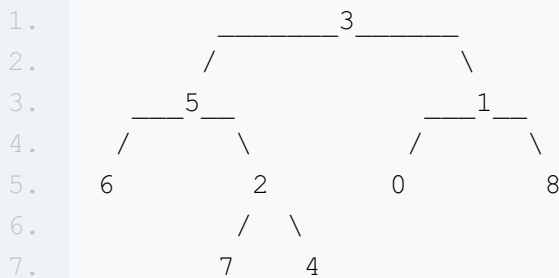
```
       lowestCommonAncestor(root.right, p, q);
4.         return root;
5.     }
```

二叉树的最近公共祖先

236. Lowest Common Ancestor of a Binary Tree (Medium)

```
1.             _____3_____
2.            /              \
3.         ___5__          ___1__
4.        /      \        /      \
5.       6       2       0       8
6.              / \
7.             7   4
8.
9.     For example, the lowest common ancestor (LCA) of nodes 5 and 1 is 3. A
       nother example is LCA of nodes 5 and 4 is 5, since a node can be a des
       cendant of itself according to the LCA definition.
```

```java
1.  public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
    TreeNode q) {
2.      if (root == null || root == p || root == q) return root;
3.      TreeNode left = lowestCommonAncestor(root.left, p, q);
4.      TreeNode right = lowestCommonAncestor(root.right, p, q);
5.      return left == null ? right : right == null ? left : root;
6.  }
```

从有序数组中构造二叉查找树

108. Convert Sorted Array to Binary Search Tree (Easy)

```java
1.  public TreeNode sortedArrayToBST(int[] nums) {
2.      return toBST(nums, 0, nums.length - 1);
3.  }
4.
5.  private TreeNode toBST(int[] nums, int sIdx, int eIdx){
6.      if (sIdx > eIdx) return null;
7.      int mIdx = (sIdx + eIdx) / 2;
8.      TreeNode root = new TreeNode(nums[mIdx]);
9.      root.left =  toBST(nums, sIdx, mIdx - 1);
```

```
10.        root.right = toBST(nums, mIdx + 1, eIdx);
11.        return root;
12.    }
```

根据有序链表构造平衡的二叉查找树

109. Convert Sorted List to Binary Search Tree (Medium)

```
1.    Given the sorted linked list: [-10,-3,0,5,9],
2.
3.    One possible answer is: [0,-3,9,-10,null,5], which represents the foll
      owing height balanced BST:
4.
5.          0
6.         / \
7.       -3   9
8.       /   /
9.     -10  5
```

```
1.    public TreeNode sortedListToBST(ListNode head) {
2.        if (head == null) return null;
3.        if (head.next == null) return new TreeNode(head.val);
4.        ListNode preMid = preMid(head);
5.        ListNode mid = preMid.next;
6.        preMid.next = null;  // 断开链表
7.        TreeNode t = new TreeNode(mid.val);
8.        t.left = sortedListToBST(head);
9.        t.right = sortedListToBST(mid.next);
10.       return t;
11.   }
12.
13.   private ListNode preMid(ListNode head) {
14.       ListNode slow = head, fast = head.next;
15.       ListNode pre = head;
16.       while (fast != null && fast.next != null) {
17.           pre = slow;
18.           slow = slow.next;
19.           fast = fast.next.next;
20.       }
21.       return pre;
22.   }
```

在二叉查找树中寻找两个节点，使它们的和为一个给定值

653. Two Sum IV - Input is a BST (Easy)

```
Input:

    5
   / \
  3   6
 / \   \
2   4   7

Target = 9

Output: True
```

使用中序遍历得到有序数组之后，再利用双指针对数组进行查找。

应该注意到，这一题不能用分别在左右子树两部分来处理这种思想，因为两个待求的节点可能分别在左右子树中。

```java
public boolean findTarget(TreeNode root, int k) {
    List<Integer> nums = new ArrayList<>();
    inOrder(root, nums);
    int i = 0, j = nums.size() - 1;
    while (i < j) {
        int sum = nums.get(i) + nums.get(j);
        if (sum == k) return true;
        if (sum < k) i++;
        else j--;
    }
    return false;
}

private void inOrder(TreeNode root, List<Integer> nums) {
    if (root == null) return;
    inOrder(root.left, nums);
    nums.add(root.val);
    inOrder(root.right, nums);
}
```

在二叉查找树中查找两个节点之差的最小绝对值

530. Minimum Absolute Difference in BST (Easy)

```
Input:

   1
    \
     3
    /
   2

Output:

   1
```

利用二叉查找树的中序遍历为有序的性质，计算中序遍历中临近的两个节点之差的绝对值，取最小值。

```java
private int minDiff = Integer.MAX_VALUE;
private TreeNode preNode = null;

public int getMinimumDifference(TreeNode root) {
    inOrder(root);
    return minDiff;
}

private void inOrder(TreeNode node) {
    if (node == null) return;
    inOrder(node.left);
    if (preNode != null) minDiff = Math.min(minDiff, node.val - preNode.val);
    preNode = node;
    inOrder(node.right);
}
```

寻找二叉查找树中出现次数最多的值

501. Find Mode in Binary Search Tree (Easy)

```
   1
```

```
2.          \
3.           2
4.          /
5.        2
6.
7.    return [2].
```

答案可能不止一个，也就是有多个值出现的次数一样多，并且是最大的。

```java
 1.    private int curCnt = 1;
 2.    private int maxCnt = 1;
 3.    private TreeNode preNode = null;
 4.
 5.    public int[] findMode(TreeNode root) {
 6.        List<Integer> maxCntNums = new ArrayList<>();
 7.        inOrder(root, maxCntNums);
 8.        int[] ret = new int[maxCntNums.size()];
 9.        int idx = 0;
10.        for (int num : maxCntNums) {
11.            ret[idx++] = num;
12.        }
13.        return ret;
14.    }
15.
16.    private void inOrder(TreeNode node, List<Integer> nums) {
17.        if (node == null) return;
18.        inOrder(node.left, nums);
19.        if (preNode != null) {
20.            if (preNode.val == node.val) curCnt++;
21.            else curCnt = 1;
22.        }
23.        if (curCnt > maxCnt) {
24.            maxCnt = curCnt;
25.            nums.clear();
26.            nums.add(node.val);
27.        } else if (curCnt == maxCnt) {
28.            nums.add(node.val);
29.        }
30.        preNode = node;
31.        inOrder(node.right, nums);
32.    }
```

Trie
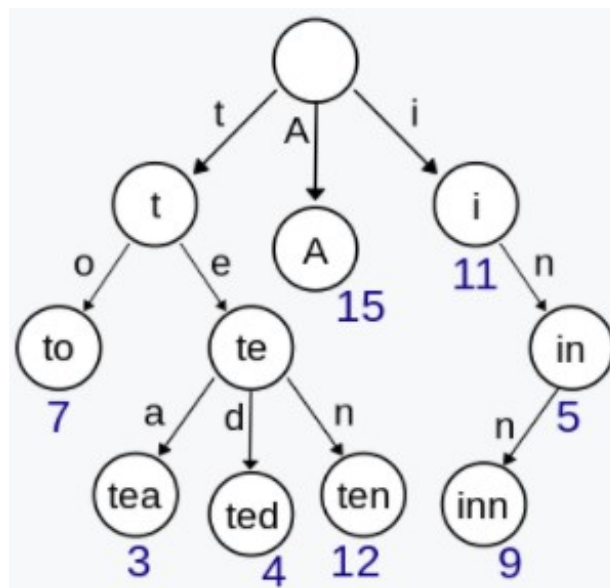
```
1.    class Trie {
2.
3.        private class Node {
4.            Node[] childs = new Node[26];
5.            boolean isLeaf;
6.        }
7.
8.        private Node root = new Node();
9.
10.       public Trie() {
11.       }
12.
13.       public void insert(String word) {
14.           insert(word, root);
15.       }
16.
17.       private void insert(String word, Node node) {
18.           if (node == null) return;
19.           if (word.length() == 0) {
20.               node.isLeaf = true;
21.               return;
22.           }
23.           int index = indexForChar(word.charAt(0));
24.           if (node.childs[index] == null) {
25.               node.childs[index] = new Node();
26.           }
27.           insert(word.substring(1), node.childs[index]);
28.       }
```

```
29.
30.        public boolean search(String word) {
31.            return search(word, root);
32.        }
33.
34.        private boolean search(String word, Node node) {
35.            if (node == null) return false;
36.            if (word.length() == 0) return node.isLeaf;
37.            int index = indexForChar(word.charAt(0));
38.            return search(word.substring(1), node.childs[index]);
39.        }
40.
41.        public boolean startsWith(String prefix) {
42.            return startWith(prefix, root);
43.        }
44.
45.        private boolean startWith(String prefix, Node node) {
46.            if (node == null) return false;
47.            if (prefix.length() == 0) return true;
48.            int index = indexForChar(prefix.charAt(0));
49.            return startWith(prefix.substring(1), node.childs[index]);
50.        }
51.
52.        private int indexForChar(char c) {
53.            return c - 'a';
54.        }
55.    }
```

实现一个 Trie，用来求前缀和

677. Map Sum Pairs (Medium)

```
1.    Input: insert("apple", 3), Output: Null
2.    Input: sum("ap"), Output: 3
3.    Input: insert("app", 2), Output: Null
4.    Input: sum("ap"), Output: 5
```

```
1.    class MapSum {
2.
3.        private class Node {
4.            Node[] child = new Node[26];
5.            int value;
6.        }
```

```java
    private Node root = new Node();

    public MapSum() {

    }

    public void insert(String key, int val) {
        insert(key, root, val);
    }

    private void insert(String key, Node node, int val) {
        if (node == null) return;
        if (key.length() == 0) {
            node.value = val;
            return;
        }
        int index = indexForChar(key.charAt(0));
        if (node.child[index] == null) {
            node.child[index] = new Node();
        }
        insert(key.substring(1), node.child[index], val);
    }

    public int sum(String prefix) {
        return sum(prefix, root);
    }

    private int sum(String prefix, Node node) {
        if (node == null) return 0;
        if (prefix.length() != 0) {
            int index = indexForChar(prefix.charAt(0));
            return sum(prefix.substring(1), node.child[index]);
        }
        int sum = node.value;
        for (Node child : node.child) {
            sum += sum(prefix, child);
        }
        return sum;
    }

    private int indexForChar(char c) {
        return c - 'a';
    }
}
```

图

二分图

如果可以用两种颜色对图中的节点进行着色，并且保证相邻的节点颜色不同，那么这个图就是二分图。

判断是否为二分图

785. Is Graph Bipartite? (Medium)

```
Input: [[1,3], [0,2], [1,3], [0,2]]
Output: true
Explanation:
The graph looks like this:
0----1
|    |
|    |
3----2
We can divide the vertices into two groups: {0, 2} and {1, 3}.
```

```
Example 2:
Input: [[1,2,3], [0,2], [0,1,3], [0,2]]
Output: false
Explanation:
The graph looks like this:
0----1
| \  |
|  \ |
3----2
We cannot find a way to divide the set of nodes into two independent s
ubsets.
```

```java
public boolean isBipartite(int[][] graph) {
    int[] colors = new int[graph.length];
    Arrays.fill(colors, -1);
    for (int i = 0; i < graph.length; i++) {  // 处理图不是连通的情况
        if (colors[i] == -1 && !isBipartite(i, 0, colors, graph)) {
            return false;
        }
    }
```

```
8.        }
9.        return true;
10.   }
11.
12.   private boolean isBipartite(int curNode, int curColor, int[] colors, int[][] graph) {
13.       if (colors[curNode] != -1) {
14.           return colors[curNode] == curColor;
15.       }
16.       colors[curNode] = curColor;
17.       for (int nextNode : graph[curNode]) {
18.           if (!isBipartite(nextNode, 1 - curColor, colors, graph)) {
19.               return false;
20.           }
21.       }
22.       return true;
23.   }
```

拓扑排序

常用于在具有先序关系的任务规划中。

课程安排的合法性

207. Course Schedule (Medium)

```
1.   2, [[1,0]]
2.   return true
```

```
1.   2, [[1,0],[0,1]]
2.   return false
```

题目描述：一个课程可能会先修课程，判断给定的先修课程规定是否合法。

本题不需要使用拓扑排序，只需要检测有向图是否存在环即可。

```
1.   public boolean canFinish(int numCourses, int[][] prerequisites) {
2.       List<Integer>[] graphic = new List[numCourses];
3.       for (int i = 0; i < numCourses; i++) {
4.           graphic[i] = new ArrayList<>();
```

```
5.            }
6.        for (int[] pre : prerequisites) {
7.            graphic[pre[0]].add(pre[1]);
8.        }
9.        boolean[] globalMarked = new boolean[numCourses];
10.        boolean[] localMarked = new boolean[numCourses];
11.        for (int i = 0; i < numCourses; i++) {
12.            if (hasCycle(globalMarked, localMarked, graphic, i)) {
13.                return false;
14.            }
15.        }
16.        return true;
17.    }
18.
19.    private boolean hasCycle(boolean[] globalMarked, boolean[] localMarked,
20.                             List<Integer>[] graphic, int curNode) {
21.
22.        if (localMarked[curNode]) {
23.            return true;
24.        }
25.        if (globalMarked[curNode]) {
26.            return false;
27.        }
28.        globalMarked[curNode] = true;
29.        localMarked[curNode] = true;
30.        for (int nextNode : graphic[curNode]) {
31.            if (hasCycle(globalMarked, localMarked, graphic, nextNode)) {
32.                return true;
33.            }
34.        }
35.        localMarked[curNode] = false;
36.        return false;
37.    }
```

课程安排的顺序

210. Course Schedule II (Medium)

```
1.    4, [[1,0],[2,0],[3,1],[3,2]]
2.    There are a total of 4 courses to take. To take course 3 you should ha
      ve finished both courses 1 and 2. Both courses 1 and 2 should be taken
      after you finished course 0. So one correct course order is [0,1,2,3].
      Another correct ordering is[0,2,1,3].
```

使用 DFS 来实现拓扑排序，使用一个栈存储后序遍历结果，这个栈的逆序结果就是拓扑排序结果。

证明：对于任何先序关系：v->w，后序遍历结果可以保证 w 先进入栈中，因此栈的逆序结果中 v 会在 w 之前。

```
1.   public int[] findOrder(int numCourses, int[][] prerequisites) {
2.       List<Integer>[] graphic = new List[numCourses];
3.       for (int i = 0; i < numCourses; i++) {
4.           graphic[i] = new ArrayList<>();
5.       }
6.       for (int[] pre : prerequisites) {
7.           graphic[pre[0]].add(pre[1]);
8.       }
9.       Stack<Integer> postOrder = new Stack<>();
10.      boolean[] globalMarked = new boolean[numCourses];
11.      boolean[] localMarked = new boolean[numCourses];
12.      for (int i = 0; i < numCourses; i++) {
13.          if (hasCycle(globalMarked, localMarked, graphic, i, postOrder))
     {
14.              return new int[0];
15.          }
16.      }
17.      int[] orders = new int[numCourses];
18.      for (int i = numCourses - 1; i >= 0; i--) {
19.          orders[i] = postOrder.pop();
20.      }
21.      return orders;
22.  }
23.
24.  private boolean hasCycle(boolean[] globalMarked, boolean[] localMarked
     , List<Integer>[] graphic,
25.                           int curNode, Stack<Integer> postOrder) {
26.
27.      if (localMarked[curNode]) {
28.          return true;
29.      }
30.      if (globalMarked[curNode]) {
31.          return false;
32.      }
33.      globalMarked[curNode] = true;
34.      localMarked[curNode] = true;
```

```
35.        for (int nextNode : graphic[curNode]) {
36.            if (hasCycle(globalMarked, localMarked, graphic, nextNode, post
    Order)) {
37.                return true;
38.            }
39.        }
40.        localMarked[curNode] = false;
41.        postOrder.push(curNode);
42.        return false;
43.    }
```

并查集

并查集可以动态地连通两个点，并且可以非常快速地判断两个点是否连通。

冗余连接

684. Redundant Connection (Medium)

```
1.    Input: [[1,2], [1,3], [2,3]]
2.    Output: [2,3]
3.    Explanation: The given undirected graph will be like this:
4.      1
5.     / \
6.    2 - 3
```

题目描述：有一系列的边连成的图，找出一条边，移除它之后该图能够成为一棵树。

```
1.    public int[] findRedundantConnection(int[][] edges) {
2.        int N = edges.length;
3.        UF uf = new UF(N);
4.        for (int[] e : edges) {
5.            int u = e[0], v = e[1];
6.            if (uf.connect(u, v)) {
7.                return e;
8.            }
9.            uf.union(u, v);
10.        }
11.        return new int[]{-1, -1};
12.    }
13.
```

```
14.    private class UF {
15.        private int[] id;
16.
17.        UF(int N) {
18.            id = new int[N + 1];
19.            for (int i = 0; i < id.length; i++) {
20.                id[i] = i;
21.            }
22.        }
23.
24.        void union(int u, int v) {
25.            int uID = find(u);
26.            int vID = find(v);
27.            if (uID == vID) {
28.                return;
29.            }
30.            for (int i = 0; i < id.length; i++) {
31.                if (id[i] == uID) {
32.                    id[i] = vID;
33.                }
34.            }
35.        }
36.
37.        int find(int p) {
38.            return id[p];
39.        }
40.
41.        boolean connect(int u, int v) {
42.            return find(u) == find(v);
43.        }
44.    }
```

位运算

1. 基本原理

0s 表示一串 0 , 1s 表示一串 1。

```
1.    x ^ 0s = x      x & 0s = 0      x | 0s = x
2.    x ^ 1s = ~x     x & 1s = x      x | 1s = 1s
3.    x ^ x = 0       x & x = x       x | x = x
```

- 利用 x ^ 1s = \~x 的特点，可以将位级表示翻转；利用 x ^ x = 0 的特点，可以将三个数中重复的两个数去除，只留下另一个数。
- 利用 x & 0s = 0 和 x & 1s = x 的特点，可以实现掩码操作。一个数 num 与 mask ：00111100 进行位与操作，只保留 num 中与 mask 的 1 部分相对应的位。
- 利用 x | 0s = x 和 x | 1s = 1s 的特点，可以实现设值操作。一个数 num 与 mask ：00111100 进行位或操作，将 num 中与 mask 的 1 部分相对应的位都设置为 1。

位与运算技巧：

- n&(n-1) 去除 n 的位级表示中最低的那一位。例如对于二进制表示 10110 **100** ，减去 1 得到 10110**011** ，这两个数相与得到 10110**000** 。
- n&(-n) 得到 n 的位级表示中最低的那一位。-n 得到 n 的反码加 1，对于二进制表示 10110 **100** ，-n 得到 01001**100** ，相与得到 00000**100** 。
- n-n&(\~n+1) 去除 n 的位级表示中最高的那一位。

移位运算：

- >> n 为算术右移，相当于除以 2^n ；
- >>> n 为无符号右移，左边会补上 0。
- << n 为算术左移，相当于乘以 2^n 。

2. mask 计算

要获取 111111111，将 0 取反即可，\~0。

要得到只有第 i 位为 1 的 mask，将 1 向左移动 i-1 位即可，1<<(i-1)。例如 1<<4 得到只有第 5 位为 1 的 mask ：00010000。

要得到 1 到 i 位为 1 的 mask，1<<(i+1)-1 即可，例如将 1<<(4+1)-1 = 00010000-1 = 00001111。

要得到 1 到 i 位为 0 的 mask，只需将 1 到 i 位为 1 的 mask 取反，即 \~(1<<(i+1)-1)。

3. Java 中的位操作

```
1.    static int Integer.bitCount();              // 统计 1 的数量
2.    static int Integer.highestOneBit();          // 获得最高位
3.    static String toBinaryString(int i);         // 转换为二进制表示的字符串
```

统计两个数的二进制表示有多少位不同

461. Hamming Distance (Easy)

```
Input: x = 1, y = 4

Output: 2

Explanation:
1   (0 0 0 1)
4   (0 1 0 0)
       ↑   ↑

The above arrows point to positions where the corresponding bits are different.
```

对两个数进行异或操作，位级表示不同的那一位为 1，统计有多少个 1 即可。

```java
public int hammingDistance(int x, int y) {
    int z = x ^ y;
    int cnt = 0;
    while(z != 0) {
        if ((z & 1) == 1) cnt++;
        z = z >> 1;
    }
    return cnt;
}
```

使用 z&(z-1) 去除 z 位级表示最低的那一位。

```java
public int hammingDistance(int x, int y) {
    int z = x ^ y;
    int cnt = 0;
    while (z != 0) {
        z &= (z - 1);
        cnt++;
    }
    return cnt;
}
```

可以使用 Integer.bitcount() 来统计 1 个的个数。

```
1.    public int hammingDistance(int x, int y) {
2.        return Integer.bitCount(x ^ y);
3.    }
```

数组中唯一一个不重复的元素

136. Single Number (Easy)

```
1.    Input: [4,1,2,1,2]
2.    Output: 4
```

两个相同的数异或的结果为 0，对所有数进行异或操作，最后的结果就是单独出现的那个数。

```
1.    public int singleNumber(int[] nums) {
2.        int ret = 0;
3.        for (int n : nums) ret = ret ^ n;
4.        return ret;
5.    }
```

找出数组中缺失的那个数

268. Missing Number (Easy)

```
1.    Input: [3,0,1]
2.    Output: 2
```

题目描述：数组元素在 0-n 之间，但是有一个数是缺失的，要求找到这个缺失的数。

```
1.    public int missingNumber(int[] nums) {
2.        int ret = 0;
3.        for (int i = 0; i < nums.length; i++) {
4.            ret = ret ^ i ^ nums[i];
5.        }
6.        return ret ^ nums.length;
7.    }
```

数组中不重复的两个元素

260. Single Number III (Medium)

两个不相等的元素在位级表示上必定会有一位存在不同。

将数组的所有元素异或得到的结果为不存在重复的两个元素异或的结果。

diff &= -diff 得到出 diff 最右侧不为 0 的位，也就是不存在重复的两个元素在位级表示上最右侧不同的那一位，利用这一位就可以将两个元素区分开来。

```java
public int[] singleNumber(int[] nums) {
    int diff = 0;
    for (int num : nums) diff ^= num;
    diff &= -diff;  // 得到最右一位
    int[] ret = new int[2];
    for (int num : nums) {
        if ((num & diff) == 0) ret[0] ^= num;
        else ret[1] ^= num;
    }
    return ret;
}
```

翻转一个数的比特位

190. Reverse Bits (Easy)

```java
public int reverseBits(int n) {
    int ret = 0;
    for (int i = 0; i < 32; i++) {
        ret <<= 1;
        ret |= (n & 1);
        n >>>= 1;
    }
    return ret;
}
```

如果该函数需要被调用很多次，可以将 int 拆成 4 个 byte，然后缓存 byte 对应的比特位翻转，最后再拼接起来。

```
1.    private static Map<Byte, Integer> cache = new HashMap<>();
2.
3.    public int reverseBits(int n) {
4.        int ret = 0;
5.        for (int i = 0; i < 4; i++) {
6.            ret <<= 8;
7.            ret |= reverseByte((byte) (n & 0b11111111));
8.            n >>= 8;
9.        }
10.       return ret;
11.   }
12.
13.   private int reverseByte(byte b) {
14.       if (cache.containsKey(b)) return cache.get(b);
15.       int ret = 0;
16.       byte t = b;
17.       for (int i = 0; i < 8; i++) {
18.           ret <<= 1;
19.           ret |= t & 1;
20.           t >>= 1;
21.       }
22.       cache.put(b, ret);
23.       return ret;
24.   }
```

不用额外变量交换两个整数

程序员代码面试指南：P317

```
1.    a = a ^ b;
2.    b = a ^ b;
3.    a = a ^ b;
```

判断一个数是不是 2 的 n 次方

231. Power of Two (Easy)

二进制表示只有一个 1 存在。

```
1.    public boolean isPowerOfTwo(int n) {
2.        return n > 0 && Integer.bitCount(n) == 1;
```

```
3.    }
```

利用 1000 & 0111 == 0 这种性质，得到以下解法：

```
1.    public boolean isPowerOfTwo(int n) {
2.        return n > 0 && (n & (n - 1)) == 0;
3.    }
```

判断一个数是不是 4 的 n 次方

342. Power of Four (Easy)

这种数在二进制表示中有且只有一个奇数位为 1，例如 16（10000）。

```
1.    public boolean isPowerOfFour(int num) {
2.        return num > 0 && (num & (num - 1)) == 0 && (num &
      0b01010101010101010101010101010101) != 0;
3.    }
```

也可以使用正则表达式进行匹配。

```
1.    public boolean isPowerOfFour(int num) {
2.        return Integer.toString(num, 4).matches("10*");
3.    }
```

判断一个数的位级表示是否不会出现连续的 0 和 1

693. Binary Number with Alternating Bits (Easy)

```
1.    Input: 10
2.    Output: True
3.    Explanation:
4.    The binary representation of 10 is: 1010.
5.
6.    Input: 11
7.    Output: False
8.    Explanation:
9.    The binary representation of 11 is: 1011.
```

对于 1010 这种位级表示的数，把它向右移动 1 位得到 101，这两个数每个位都不同，因此异或得到的结果为 1111。

```java
public boolean hasAlternatingBits(int n) {
    int a = (n ^ (n >> 1));
    return (a & (a + 1)) == 0;
}
```

求一个数的补码

476. Number Complement (Easy)

```
Input: 5
Output: 2
Explanation: The binary representation of 5 is 101 (no leading zero bits), and its complement is 010. So you need to output 2.
```

题目描述：不考虑二进制表示中的首 0 部分。

对于 00000101，要求补码可以将它与 00000111 进行异或操作。那么问题就转换为求掩码 00000111。

```java
public int findComplement(int num) {
    if (num == 0) return 1;
    int mask = 1 << 30;
    while ((num & mask) == 0) mask >>= 1;
    mask = (mask << 1) - 1;
    return num ^ mask;
}
```

可以利用 Java 的 Integer.highestOneBit() 方法来获得含有首 1 的数。

```java
public int findComplement(int num) {
    if (num == 0) return 1;
    int mask = Integer.highestOneBit(num);
    mask = (mask << 1) - 1;
    return num ^ mask;
}
```

对于 10000000 这样的数要扩展成 11111111，可以利用以下方法：

```
1.    mask |= mask >> 1    11000000
2.    mask |= mask >> 2    11110000
3.    mask |= mask >> 4    11111111
```

```
1.    public int findComplement(int num) {
2.        int mask = num;
3.        mask |= mask >> 1;
4.        mask |= mask >> 2;
5.        mask |= mask >> 4;
6.        mask |= mask >> 8;
7.        mask |= mask >> 16;
8.        return (mask ^ num);
9.    }
```

实现整数的加法

371. Sum of Two Integers (Easy)

a ^ b 表示没有考虑进位的情况下两数的和，(a & b) << 1 就是进位。

递归会终止的原因是 (a & b) << 1 最右边会多一个 0，那么继续递归，进位最右边的 0 会慢慢增多，最后进位会变为 0，递归终止。

```
1.    public int getSum(int a, int b) {
2.        return b == 0 ? a : getSum((a ^ b), (a & b) << 1);
3.    }
```

字符串数组最大乘积

318. Maximum Product of Word Lengths (Medium)

```
1.    Given ["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]
2.    Return 16
3.    The two words can be "abcw", "xtfn".
```

题目描述：字符串数组的字符串只含有小写字符。求解字符串数组中两个字符串长度的最大乘

积，要求这两个字符串不能含有相同字符。

本题主要问题是判断两个字符串是否含相同字符，由于字符串只含有小写字符，总共 26 位，因此可以用一个 32 位的整数来存储每个字符是否出现过。

```java
public int maxProduct(String[] words) {
    int n = words.length;
    int[] val = new int[n];
    for (int i = 0; i < n; i++) {
        for (char c : words[i].toCharArray()) {
            val[i] |= 1 << (c - 'a');
        }
    }
    int ret = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if ((val[i] & val[j]) == 0) {
                ret = Math.max(ret, words[i].length() * words[j].length());
            }
        }
    }
    return ret;
}
```

统计从 0 \~ n 每个数的二进制表示中 1 的个数

338. Counting Bits (Medium)

对于数字 6(110)，它可以看成是 4(100) 再加一个 2(10)，因此 dp[i] = dp[i&(i-1)] + 1;

```java
public int[] countBits(int num) {
    int[] ret = new int[num + 1];
    for(int i = 1; i <= num; i++){
        ret[i] = ret[i&(i-1)] + 1;
    }
    return ret;
}
```

参考资料

- Leetcode
- Weiss M A, 冯舜玺. 数据结构与算法分析——C 语言描述[J]. 2004.
- Sedgewick R. Algorithms[M]. Pearson Education India, 1988.
- 何海涛, 软件工程师. 剑指 Offer: 名企面试官精讲典型编程题[M]. 电子工业出版社, 2014.
- 《编程之美》小组. 编程之美[M]. 电子工业出版社, 2008.
- 左程云. 程序员代码面试指南[M]. 电子工业出版社, 2015.

github: https://github.com/sjsdfg/Interview-Notebook-PDF