算法总结

一）十大排序算法

1）冒泡排序Bubble Sort

- **算法步骤**：每一轮都把前后元素中较大的一位放到后面，这样子一趟下来，最大的元素就自然放在了后面。平均时间复杂度n^2，最佳时间复杂度n（正序），最差时间复杂度n^2（逆序）

```
public int[] bubbleSort(int[] nums){
    boolean flag = false;
    for (int i = 0;i < nums.length - 1;i++){
        if (flag) break;
        for (int j = 0;j < nums.length - i - 1;j++){
            if (nums[j] > nums[j + 1]){
                flag = true;
                swap(nums, j, j + 1);
            }
        }
    }
    return nums;
}
```

2）选择排序Select Sort

- **算法步骤**：每一轮选择当前的最大（最小）元素放在当前轮次的末尾。时间复杂度n^2

```
public int[] selectSort(int[] nums){
    for (int i = 1;i < nums.length;i++){
        int maxIndex = 0;
        for (int j = 1;j <= nums.length - i;j++){
            if (nums[maxIndex] < nums[j]) maxIndex = j;
        }
        swap(nums, maxIndex, nums.length - i);
    }
    return nums;
}
```

3）插入排序Insert Sort

- **算法步骤**：后面的元素插入到前方已排序的数组中的合适位置，第一轮次将第一个元素视为已排序。平均时间复杂度n^2，最佳n，最差n^2

```
public int[] insertSort(int[] nums){
    for (int i = 1;i < nums.length;i++){
        int j = i - 1;
        while (j >= 0){
            if (nums[j] < nums[i]) break;
```

```
            else{
                nums[j + 1] = nums[j];
                j--;
            }
        }
        nums[j + 1] = nums[i];
    }
    return nums;
}
```

4）希尔排序Shell Sort（插入排序升级版）

- **算法思想**：考虑到插入排序的两个性质：插入排序一般较为低效率、插入排序在遇到大致上排好序的数组时效率更高。将数组分为多个子数组，分别进行插入排序形成已排序子数组后，再排序数组。

```
public int[] shellSort(int[] nums){
    for (int step = nums.length/2;step >= 1;step /= 2){
        for (int i = step;i < nums.length;i++){
            int j = i - step;
            while (j >= 0){
                if (nums[j] < nums[i]) break;
                else{
                    nums[j + step] = nums[j];
                    j -= step;
                }
            }
            nums[j + step] = nums[i];
        }
    }
    return nums;
}
```

5）归并排序Merge Sort

- **算法步骤**：本质上就是合并两个已排好序的数组，对于一整个待排序数组，采用递归的方式来进行。平均时间复杂度nlogn。

```
public int[] sort(int[] nums){
    int[] array = Arrays.copyOf(nums, nums.length);
    if (nums.length == 1) return array;
    int mid = nums.length / 2;
    int[] left = Arrays.copyOfRange(array, 0, mid);
    int[] right = Arrays.copyOfRange(array, mid, array.length);
    return merge(sort(left), sort(right));
}

public int[] merge(int[] left, int[] right){
    int[] result = new int[left.length + right.length];
    int l = 0;
    int r = 0;
    int index = 0;
```

```
    while (l < left.length && r < right.length){
        if (left[l] < right[r]){
            result[index++] = left[l++];
        }else{
            result[index++] = right[r++];
        }
    }
    while (l < left.length){
        result[index++] = left[l++];
    }
    while (r < right.length){
        result[index++] = right[r++];
    }
    return result;
}
```

6）快速排序Quick Sort

- **算法步骤**：每一次从数组中找一个数做基准，将大于它的元素放后面，小于它的放前面，一轮下来它能找到自己的位置。并且，数组被分为两个待排序数组。平均时间复杂度nlogn。

```
public void quickSort(int[] nums, int start, int end){
    if (start >= end) return;
    int pivot = nums[start];
    int left = start;
    int right = end;
    while (left < right){
        while (nums[right] > pivot && right > left) right--;
        if (left == right) break;
        nums[left++] = nums[right];
        while (nums[left] < pivot && left < right) left++;
        if (left == right) break;
        nums[right--] = nums[left];
    }
    nums[left] = pivot;
    quickSort(nums, start, left);
    quickSort(nums, left + 1, end);
}
```

7）堆排序Heap Sort

- **算法步骤**：建立堆，然后逐步从里面取堆顶数据。平均时间复杂度nlogn

```
public void heapify(int[] nums, int i, int length){
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int min = i;
    if (left < length && nums[min] > nums[left]){
        min = left;
    }
    if (right < length && nums[min] > nums[right]){
        min = right;
```

```
    }
    if (min != i){
        swap(nums, i, min);
        heapify(nums, min, length);
    }
}

public void buildMinHeap(int[] nums){
    for (int i = nums.length / 2;i >= 0;i--){
        heapify(nums, i, nums.length);
    }
}

public int[] heapSort(int[] nums){
    buildMinHeap(nums);
    int[] result = new int[nums.length];
    for (int i = 0;i < nums.length;i++){
        result[i] = nums[0];
        swap(nums, 0, nums.length - i);
        heapify(nums, 0, nums.length - i);
    }
    return result;
}
```

8）计数排序Counting Sort

- **算法步骤**：不进行比较，取出待排序数组的最大值和最小值，建立一个差值+1大小的数组，存放对应下标元素出现的个数，然后再依次取出。

```
public int[] countingSort(int[] nums){
    int min = nums[0];
    int max = nums[0];
    for (int i = 1;i < nums.length;i++){
        min = nums[i] < min ? nums[i] : min;
        max = nums[i] > max ? nums[i] : max;
    }
    int[] bucket = new int[max - min + 1];
    for (int i = 0;i < nums.length;i++){
        bucket[nums[i] - min]++;
    }
    int index = 0;
    for (int i = 0;i < bucket.length;i++){
        while (bucket[i] != 0){
            nums[index++] = min + i;
            bucket[i]--;
        }
    }
    return nums;
}
```

9）桶排序Bucket Sort（计数排序升级版）

- **算法思想**：计数排序的升级版，即桶bucket的数量减少，通过一定的映射函数将元素均匀分布于几个桶中，在桶内的元素使用排序方法进行排序，再逐个取出

10）基数排序Radix Sort

- **算法步骤**：按基数进行非比较排序

```
// 假设已实例化
List<List<Integer>> bucket;

public int getNumLength(int max){
    int length = 0;
    for (int i = 1;i < max;i *= 10){
        length++;
    }
    return length;
}

public int[] radixSort(int[] nums){
    int max = nums[0];
    for (int i = 0;i < nums.length;i++){
        max = nums[i] > max ? nums[i] : max;
    }
    int numLength = getNumLength(max);
    int mod = 10;
    int div = 1;
    for (int i = 1;i <= numLength;i++){
        for (int j = 0;j < nums.length;j++){
            if (nums[j] < div || nums[j] == 0){
                bucket.get(0).add(nums[j]);
            }else{
                int index = nums[j] % mod / div;
                bucket.get(index).add(nums[j]);
            }
        }
        mod *= 10;
        div *= 10;
        int count = 0;
        for (int j = 0;j <= 9;j++){
            for (int k = 0;k < bucket.get(j).size();k++){
                nums[count++] = bucket.get(j).get(k);
            }
        }
    }
    return nums;
}
```

11）十大排序算法总结

名称	数据对象	稳定性	时间复杂度		额外空间复杂度	描述
			平均	最坏		
冒泡排序	数组	✓	$O(n^2)$		$O(1)$	（无序区，有序区）。 从无序区透过交换找出最大元素放到有序区前端。
选择排序	数组	✗	$O(n^2)$		$O(1)$	（有序区，无序区）。 在无序区里找一个最小的元素跟在有序区的后面。对数组：比较得多，换得少。
	链表	✓				
插入排序	数组、链表	✓	$O(n^2)$		$O(1)$	（有序区，无序区）。 把无序区的第一个元素插入到有序区的合适的位置。对数组：比较得少，换得多。
堆排序	数组	✗	$O(n\log n)$		$O(1)$	（最大堆，有序区）。 从堆顶把根卸出来放在有序区之前，再恢复堆。
归并排序	数组	✓	$O(n\log^2 n)$		$O(1)$	把数据分为两段，从两段中逐个选最小的元素移入新数据段的末尾。 可从上到下或从下到上进行。
			$O(n\log n)$		$O(n)+O(\log n)$ 如果不是从下到上	
	链表				$O(1)$	
快速排序	数组	✗	$O(n\log n)$	$O(n^2)$	$O(\log n)$	（小数，基准元素，大数）。 在区间中随机挑选一个元素作基准，将小于基准的元素放在基准之前，大于基准的元素放在基准之后，再分别对小数区与大数区进行排序。
希尔排序	数组	✗	$O(n\log^2 n)$	$O(n^2)$	$O(1)$	每一轮按照事先决定的间隔进行插入排序，间隔会依次缩小，最后一次一定要是1。
计数排序	数组、链表	✓	$O(n+m)$		$O(n+m)$	统计小于等于该元素值的元素的个数i，于是该元素就放在目标数组的索引i位（i≥0）。
桶排序	数组、链表	✓	$O(n)$		$O(m)$	将值为i的元素放入i号桶，最后依次把桶里的元素倒出来。
基数排序	数组、链表	✓	$O(k\times n)$	$O(n^2)$		一种多关键字的排序算法，可用桶排序实现。

二）常用算法与对应情况

1）动态规划问题详解

- **常见形式**：用于求**最值**。求最值意味着要穷举所有状态，但一般情况下存在重叠子问题，因此需要找出最优子结构，写出状态转移方程来简化穷举中重复的内容。
- **解题步骤**：明确基础情况 -> 明确状态有哪些 -> 明确每次可做的选择有哪些 -> 定义状态转移方程
- **框架**：

```
// 数组维度取决于状态有多少种
dp[o][0][...] = base;
// 进行状态转移
for 状态1 in 状态1所有取值：
    for 状态2 in 状态2所有取值：
        for ...
            dp[状态1][状态2][...] = 求最值(选择1，选择2...)
```

2）回溯算法详解

- **常见形式**：解决一个回溯问题，实际上就是一个决策树的遍历问题
- **框架**：

```
result = [];
public void backtrack(路径, 选择列表){
    if (满足结束条件){
        result.add(路径);
        return
    }
    for 选择 in 选择列表:
        做选择;
        backtrack(路径，选择列表);
        撤销选择;
}
```

3）二分查找详解

- **常见形式**：查找一个数，查找左侧边界，查找右侧边界
- **框架**：

```
// 最基础，查找一个数
public int binarySearch(int[] nums, int target){
    int left = 0;
    int right = nums.length - 1;
    while (left <= right){
        int mid = left + (right - left) / 2;
        if (nums[mid] == target){
            return mid;
        }else if (nums[mid] < target){
            left = mid + 1;
        }else if (nums[mid] > target){
            right = mid - 1;
        }
    }
    return -1;
}

// 左边界
public int leftBound(int[] nums, int target){
    int left = 0;
    int right = nums.length - 1;
    while (left <= right){
        int mid = left + (right - left) / 2;
        if (nums[mid] < target){
            left = mid + 1;
        }else if (nums[mid] > target){
            right = mid - 1;
        }else if (nums[mid] == target){
            right = mid - 1;
        }
    }
    if (left == nums.length || nums[left] != target) return -1;
    return left;
}

// 右边界
public int rightBound(int[] nums, int target){
    int left = 0;
    int right = nums.length - 1;
    while (left <= right){
        int mid = left + (right - left) / 2;
        if (nums[mid] < target){
            left = mid + 1;
        }else if (nums[mid] > target){
            right = mid - 1;
        }else if (nums[mid] == target){
            left = mid + 1;
        }
    }
    if (right < 0 || nums[right] != target) return -1;
    return right;
}
```

4）滑动窗口详解

- **常见形式**：常用于字符串的子串查找等
- **框架**：

```
int left = 0;
int right = 0;
while (right < s.length()){
    window.add(s[right]);
    right++;
    while (valid){
        window.remove(s[left]);
        left++;
    }
}
```

5）双指针——快慢指针

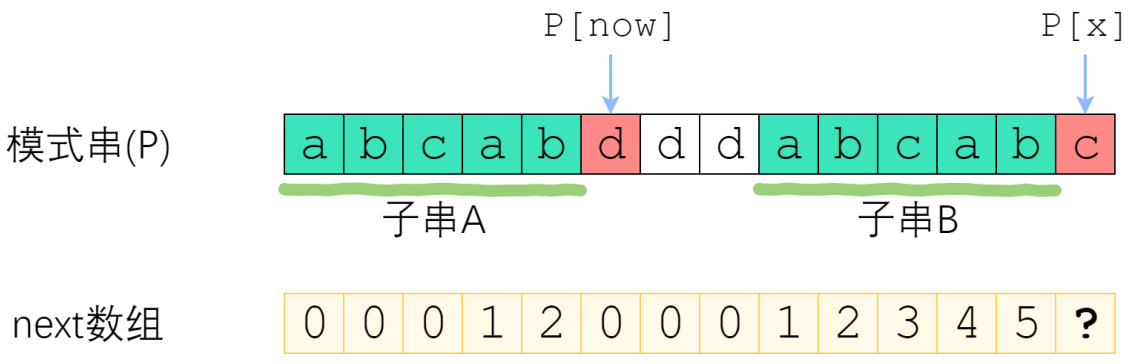
- **常见形式**：用于解决链表中的问题，如检查链表中是否包含环
- **框架**：
    - **判断链表中是否含有环**：经典解法就是用两个指针，一个跑得快，一个跑得慢。如果不含有环，跑得快的那个指针最终会遇到 null，说明链表不含环；如果含有环，快指针最终会超慢指针一圈，和慢指针相遇，说明链表含有环。
    - **已知链表中含有环，返回这个环的起始位置**：当快慢指针相遇时，让其中任一个指针指向头节点，然后让它俩以相同速度前进，再次相遇时所在的节点位置就是环开始的位置。
    - **寻找链表的中点**：让快指针一次前进两步，慢指针一次前进一步，当快指针到达链表尽头时，慢指针就处于链表的中间位置。
    - **寻找链表的倒数第 k 个元素**：让快指针先走 k 步，然后快慢指针开始同速前进。这样当快指针走到链表末尾 null 时，慢指针所在的位置就是倒数第 k 个链表节点。

6）双指针——左右指针

- **常见形式**：用于解决数组或字符串的问题，比如二分查找
- **框架**：常见的二分查找、两数之和、反转数组

7）KMP算法

- **常见形式**：用于字符串中查找子串
- **框架**：利用之前比较过的信息，减少不必要的移动




```
public int[] buildNext(String pat){
    int[] next = new int[pat.length()];
```

```
    int i = 1, now = 0;
    while (i < pat.length()){
        if (pat.charAt(i) == pat.charAt(now)){
            now++;
            next[i] = now;
            i++;
        } else if (now != 0){
            now = next[now - 1];
        } else {
            i++;
            next[i] = now;
        }
    }
}

public int search(String str){
    int tar = 0, pos = 0;
    while (tar < str.length()){
        if (s.charAt(tar) == pat.charAt(pos)){
            tar++;
            pos++;
        } else if (pos != 0){
            pos = next[pos - 1];
        } else{
            tar++;
        }
        if (pos == pat.length()){
            return tar - pos - 1;
        }
    }
}
```

8）位操作

- **字符相关位操作**:

```
'b' | ' ' = 'b';// 或 ' ' 变小写
'b' & '_' = 'B';// 与 '_' 变大写
'b' ^ ' ' = 'B';// 亦或 ' ' 大写变小写，小写变大写
```

- **数字相关位操作**:

```
n & (n - 1); // 消除数字的二进制的最后一个1，可用于计算数的二进制有多少个1
n ^ n = 0; // 可用于查找数组中只出现一次的数
```

三）经典题目解法——动态规划系列

1）最长递增子序列

- **题目描述**：给定一个无序的整数数组，找到其中最长上升子序列的长度
- **题目解法**：确定状态为数组中的每个数，选择为是否将当前数加入序列中。定义dp[i]表示以nums[i]为结尾的上升子序列长度，每次选择的时候进行循环，分析nums[0...i - 1]为结尾的子序列是否可以加入nums[i]，以此来比较取最大值。

```
public int lengthOfLIS(int[] nums) {
    int[] dp = new int[nums.length];
    // base case：dp 数组全都初始化为 1
    Arrays.fill(dp, 1);
    for (int i = 0; i < nums.length; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] > nums[j])
                dp[i] = Math.max(dp[i], dp[j] + 1);
        }
    }

    int res = 0;
    for (int i = 0; i < dp.length; i++) {
        res = Math.max(res, dp[i]);
    }
    return res;
}
```

2）编辑距离

- **题目描述**：给定两个字符串，计算出将其中一个字符串变成另一个所使用的最少操作数，可进行删除、插入、替换一个字符。
- **题目解法**：两个字符串解题步骤中出现穷举的情况时考虑动态规划。定义dp[i][j]表示s1[0..i]到s2[0..j]的最小编辑距离，用i和j分别表示走到s1和s2的哪个字符，基础操作有4个。绘制dp表格，根据操作决定遍历方向。

```
if s1[i] == s2[j]:
    啥都别做（skip）
    i, j 同时向前移动
else:
    三选一：
        插入（insert） s1[i] -> s2[j - 1] + 1
        删除（delete） s1[i - 1] -> s2[j] + 1
        替换（replace）s1[i - 1] -> s2[j - 1] + 1
```

```
int minDistance(String s1, String s2) {
    int m = s1.length(), n = s2.length();
    int[][] dp = new int[m + 1][n + 1];
    // base case
    for (int i = 1; i <= m; i++)
        dp[i][0] = i;
    for (int j = 1; j <= n; j++)
        dp[0][j] = j;
    // 自底向上求解
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            if (s1.charAt(i-1) == s2.charAt(j-1))
                dp[i][j] = dp[i - 1][j - 1];
```

```
            else
                dp[i][j] = min(
                    dp[i - 1][j] + 1,
                    dp[i][j - 1] + 1,
                    dp[i-1][j-1] + 1
                );
    // 储存着整个 s1 和 s2 的最小编辑距离
    return dp[m][n];
}

int min(int a, int b, int c) {
    return Math.min(a, Math.min(b, c));
}
```

3）0-1背包问题

- **题目描述**：给你一个可装载重量为 `W` 的背包和 `N` 个物品，每个物品有重量和价值两个属性。其中第 `i` 个物品的重量为 `wt[i]`，价值为 `val[i]`，现在让你用这个背包装物品，最多能装的价值是多少?
- **题目解法**：定义dp[i][w]表示对于前i个物品，容量为w时所装的最大价值。主要思路为逐步引入物品。

```
int knapsack(int W, int N, vector<int>& wt, vector<int>& val) {
    // base case 已初始化
    vector<vector<int>> dp(N + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= N; i++) {
        for (int w = 1; w <= W; w++) {
            if (w - wt[i-1] < 0) {
                // 这种情况下只能选择不装入背包
                dp[i][w] = dp[i - 1][w];
            } else {
                // 装入或者不装入背包，择优
                dp[i][w] = max(dp[i - 1][w - wt[i-1]] + val[i-1],
                               dp[i - 1][w]);
            }
        }
    }

    return dp[N][W];
}
```

4）高楼鸡蛋问题

- **题目描述**：你面前有一栋从 1 到 `N` 共 `N` 层的楼，然后给你 `K` 个鸡蛋（`K` 至少为 1）。现在确定这栋楼存在楼层 `0 <= F <= N`，在这层楼将鸡蛋扔下去，鸡蛋**恰好没摔碎**（高于 `F` 的楼层都会碎，低于 `F` 的楼层都不会碎）。现在问你，**最坏**情况下，你**至少**要扔几次鸡蛋，才能**确定**这个楼层 `F` 呢?
- **题目解法**：该问题不存在规律，因此采用动态规划来进行解决，定义dp[k][n]表示有K个鸡蛋N层楼时的最小损耗。
    - 我们选择在第 `i` 层楼扔了鸡蛋之后，可能出现两种情况：鸡蛋碎了，鸡蛋没碎。**注意，这时候状态转移就来了**：
    - **如果鸡蛋碎了**，那么鸡蛋的个数 `K` 应该减一，搜索的楼层区间应该从 `[1..N]` 变为 `[1..i-1]` 共 `i-1` 层楼；

- **如果鸡蛋没碎**，那么鸡蛋的个数 `K` 不变，搜索的楼层区间应该从 `[1..N]` 变为 `[i+1..N]` 共 `N-i` 层楼。

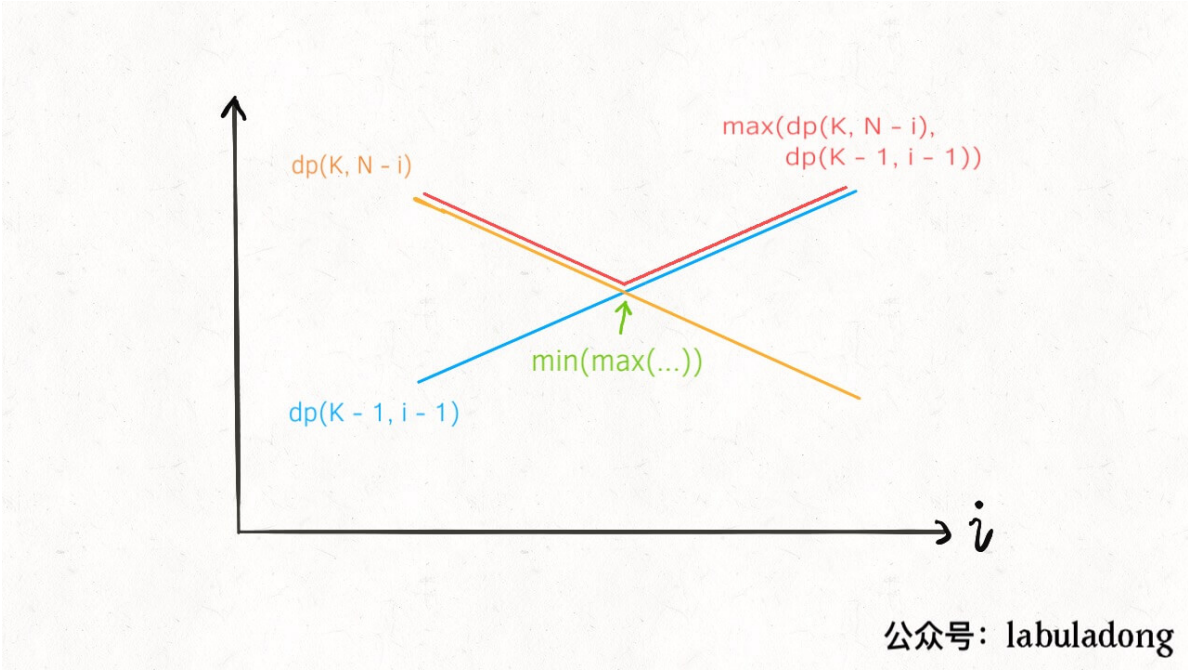
```
def superEggDrop(K: int, N: int):

    memo = dict()
    def dp(K, N) -> int:
        # base case
        if K == 1: return N
        if N == 0: return 0
        # 避免重复计算
        if (K, N) in memo:
            return memo[(K, N)]

        res = float('INF')
        # 穷举所有可能的选择
        for i in range(1, N + 1):
            res = min(res,
                      max(
                            dp(K, N - i),
                            dp(K - 1, i - 1)
                        ) + 1
                  )
        # 记入备忘录
        memo[(K, N)] = res
        return res

    return dp(K, N)
```

- **优化**：由于核心在于求**dp(K, N - i)和dp(K - 1, i - 1)中的最大值的最小值**，且K和N为固定常数，因此两个dp可看作是i的一元一次方程，前者是单调递减，后者是单调递增的，所求相交点必定在中间，因此可用二分查找优化




```
res = float('INF')
# 用二分搜索代替线性搜索
lo, hi = 1, N
while lo <= hi:
```

```
    mid = (lo + hi) // 2
    broken = dp(K - 1, mid - 1) # 碎
    not_broken = dp(K, N - mid) # 没碎
    # res = min(max(碎，没碎) + 1)
    if broken > not_broken:
        hi = mid - 1
        res = min(res, broken + 1)
    else:
        lo = mid + 1
        res = min(res, not_broken + 1)

memo[(K, N)] = res
return res
```

5）最长回文子序列

- **题目描述**：给你一个字符串 `s` ，找出其中最长的回文子序列，并返回该序列的长度。
- **题目解法**：字符串的序列问题，存在穷举过程，因此首先考虑动态规划。在子串 `s[i..j]` 中，最长回文子序列的长度为 `dp[i][j]`。若已知了dp[i + 1][j - 1]的结果，则可求dp[i][j]。由操作决定遍历方向。

```
if (s[i] == s[j])
    // 它俩一定在最长回文子序列中
    dp[i][j] = dp[i + 1][j - 1] + 2;
else
    // s[i+1..j] 和 s[i..j-1] 谁的回文子序列更长？
    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
```

```
int longestPalindromeSubseq(string s) {
    int n = s.size();
    // dp 数组全部初始化为 0
    vector<vector<int>> dp(n, vector<int>(n, 0));
    // base case
    for (int i = 0; i < n; i++)
        dp[i][i] = 1;
    // 反着遍历保证正确的状态转移
    for (int i = n - 1; i >= 0; i--) {
        for (int j = i + 1; j < n; j++) {
            // 状态转移方程
            if (s[i] == s[j])
                dp[i][j] = dp[i + 1][j - 1] + 2;
            else
                dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
        }
    }
    // 整个 s 的最长回文子串长度
    return dp[0][n - 1];
}
```

6）博弈问题

- **题目描述**：你和你的朋友面前有一排石头堆，用一个数组 piles 表示，piles[i] 表示第 i 堆石子有多少个。你们轮流拿石头，一次拿一堆，但是只能拿走最左边或者最右边的石头堆。所有石头被拿完后，谁拥有的石头多，谁获胜。
- **题目解法**：假定两个人都是理性人，都会选择当前情况下的最优解。则后手选择的就相当于堆数减少情况下的先手选择。

```
dp[i][j].fir 表示，对于 piles[i...j] 这部分石头堆，先手能获得的最高分数。
dp[i][j].sec 表示，对于 piles[i...j] 这部分石头堆，后手能获得的最高分数。

举例理解一下，假设 piles = [3, 9, 1, 2]，索引从 0 开始
dp[0][1].fir = 9 意味着：面对石头堆 [3, 9]，先手最终能够获得 9 分。
dp[1][3].sec = 2 意味着：面对石头堆 [9, 1, 2]，后手最终能够获得 2 分。
```

```
/* 返回游戏最后先手和后手的得分之差 */
int stoneGame(int[] piles) {
    int n = piles.length;
    // 初始化 dp 数组
    Pair[][] dp = new Pair[n][n];
    for (int i = 0; i < n; i++)
        for (int j = i; j < n; j++)
            dp[i][j] = new Pair(0, 0);
    // 填入 base case
    for (int i = 0; i < n; i++) {
        dp[i][i].fir = piles[i];
        dp[i][i].sec = 0;
    }
    // 斜着遍历数组
    for (int l = 2; l <= n; l++) {
        for (int i = 0; i <= n - l; i++) {
            int j = l + i - 1;
            // 先手选择最左边或最右边的分数
            int left = piles[i] + dp[i+1][j].sec;
            int right = piles[j] + dp[i][j-1].sec;
            // 套用状态转移方程
            if (left > right) {
                dp[i][j].fir = left;
                dp[i][j].sec = dp[i+1][j].fir;
            } else {
                dp[i][j].fir = right;
                dp[i][j].sec = dp[i][j-1].fir;
            }
        }
    }
    Pair res = dp[0][n-1];
    return res.fir - res.sec;
}
```

7）区间调度

- **题目描述**：你今天有好几个活动，每个活动都可以用区间 `[start, end]` 表示开始和结束的时间，请问你今天最多能参加几个活动呢？
- **题目解法**：使用贪心算法，每次选取最早结束的

```java
public int intervalSchedule(int[][] intvs) {
    if (intvs.length == 0) return 0;
    // 按 end 升序排序
    Arrays.sort(intvs, new Comparator<int[]>() {
        @Override
        public int compare(int[] a, int[] b) {
            // 这里不能使用 a[1] - b[1]，要注意溢出问题
            if (a[1] < b[1])
                return -1;
            else if (a[1] > b[1])
                return 1;
            else return 0;
        }
    });
    // 至少有一个区间不相交
    int count = 1;
    // 排序后，第一个区间就是 x
    int x_end = intvs[0][1];
    for (int[] interval : intvs) {
        int start = interval[0];
        if (start >= x_end) {
            // 找到下一个选择的区间了
            count++;
            x_end = interval[1];
        }
    }
    return count;
}
```

8）股票问题

- **题目描述**：给定一个整数数组 prices ， 它的第 i 个元素 prices[i] 是一支给定的股票在第 i 天的价格。设计一个算法来计算你所能获取的最大利润。你最多可以完成 k 笔交易。
- **题目解法**：首先定义状态有天数、交易次数、持有状态。选择有买，卖，持有三种。

```
dp[i][k][0 or 1]
0 <= i <= n - 1, 1 <= k <= K
n 为天数，大 K 为交易数的上限，0 和 1 代表是否持有股票。
此问题共 n × K × 2 种状态，全部穷举就能搞定。

for 0 <= i < n:
    for 1 <= k <= K:
        for s in {0, 1}:
            dp[i][k][s] = max(buy, sell, rest)
```

```java
// k = 2
int maxProfit_k_2(int[] prices) {
    int max_k = 2, n = prices.length;
    int[][][] dp = new int[n][max_k + 1][2];
    for (int i = 0; i < n; i++) {
        for (int k = max_k; k >= 1; k--) {
            if (i - 1 == -1) {
                // 处理 base case
                dp[i][k][0] = 0;
                dp[i][k][1] = -prices[i];
                continue;
```

```
            }
            dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
            dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
        }
    }
    // 穷举了 n × max_k × 2 个状态，正确。
    return dp[n - 1][max_k][0];
}
```

9）打家劫舍问题

- **题目描述**：你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。给定一个代表每个房屋存放金额的非负整数数组，计算你不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。
- **题目解法**：存在穷举过程，使用动态规划。首先确定状态有房间以及是否被抢，选择有抢或者不抢两种。

```
 int rob(int[] nums) {
    int n = nums.length;
    // dp[i] = x 表示：
    // 从第 i 间房子开始抢劫，最多能抢到的钱为 x
    // base case: dp[n] = 0
    int[] dp = new int[n + 2];
    for (int i = n - 1; i >= 0; i--) {
        dp[i] = Math.max(dp[i + 1], nums[i] + dp[i + 2]);
    }
    return dp[0];
}
```

10）四键键盘问题

- **题目描述**：可进行四种操作，敲出A，Ctrl+A，Ctrl+C，Ctrl+V。给定操作次数，求最多可输出多少个A
- **题目解法**：存在穷举过程，使用动态规划。首先确定状态有操作次数，缓冲区A个数，当前屏幕A个数，选择有四种。直接穷举发现时间达到n^3，对此进行优化。由于最优只存在2种情况，一种是直接按A，另一种是使用粘贴达到最多数量，因此最后一次操作一定是按A或者Ctrl+V

```
public int maxA(int N) {
    int[] dp = new int[N + 1];
    dp[0] = 0;
    for (int i = 1; i <= N; i++) {
        // 按 A 键
        dp[i] = dp[i - 1] + 1;
        // 由于不确定是从哪一步开始直接粘贴，因此进行穷举
        for (int j = 2; j < i; j++) {
            // 全选 & 复制 dp[j-2]，连续粘贴 i - j 次
            // 屏幕上共 dp[j - 2] * (i - j + 1) 个 A
            dp[i] = Math.max(dp[i], dp[j - 2] * (i - j + 1));
        }
    }
    // N 次按键之后最多有几个 A?
    return dp[N];
}
```

11）正则表达式匹配

- **题目描述**：给你一个字符串 `s` 和一个字符规律 `p`，请你来实现一个支持 `'.'` 和 `'*'` 的正则表达式匹配。
- **题目解法**：由于存在"."可以匹配任意多个字符，因此出现穷举（即"."是否匹配），使用动态规划。

```
/* 计算 p[j..] 是否匹配 s[i..] */
bool dp(string& s, int i, string& p, int j) {
    int m = s.size(), n = p.size();
    // base case
    if (j == n) {
        return i == m;
    }
    if (i == m) {
        if ((n - j) % 2 == 1) {
            return false;
        }
        for (; j + 1 < n; j += 2) {
            if (p[j + 1] != '*') {
                return false;
            }
        }
        return true;
    }

    // 记录状态 (i, j)，消除重叠子问题
    string key = to_string(i) + "," + to_string(j);
    if (memo.count(key)) return memo[key];

    bool res = false;

    if (s[i] == p[j] || p[j] == '.') { // 字符相等或当前字符为.
        if (j < n - 1 && p[j + 1] == '*') { // 后面字符为*，存在多次匹配或不匹配可能
            res = dp(s, i, p, j + 2)
               || dp(s, i + 1, p, j);
        } else {    // 后面字符不为*
            res = dp(s, i + 1, p, j + 1);
        }
    } else { // 字符不相等，查看后面字符是否为*，否则输出false
        if (j < n - 1 && p[j + 1] == '*') {
            res = dp(s, i, p, j + 2);
        } else {
            res = false;
        }
    }
    // 将当前结果记入备忘录
    memo[key] = res;

    return res;
}
```

12）最长公共子序列LCS - Longest Common Subsequent

- **题目描述**：给定两个字符串 `text1` 和 `text2`，返回这两个字符串的最长 **公共子序列** 的长度。如果不存在 **公共子序列** ，返回 `0` 。
- **题目解法**：两个字符串且存在子序列相关问题，考虑使用动态规划。一般跟字符串有关的动态规划问题，状态都跟字符索引相关。当s1[i]==s2[j]时，说明该字符是LCS中的一员，长度+1；若s1[i] != s2[j]，则说明有一个字符不在LCS中，需要舍弃。

```
public int longestCommonSubsequence(String text1, String text2) {
    // 字符串转为char数组以加快访问速度
    char[] str1 = text1.toCharArray();
    char[] str2 = text2.toCharArray();
    int m = str1.length, n = str2.length;
    // 构建dp table，初始值默认为0
    int[][] dp = new int[m + 1][n + 1];
    // 状态转移
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            if (str1[i - 1] == str2[j - 1])
                // 找到LCS中的字符
                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[m][n];
}
```

四）经典题目解法——数据结构

1）二叉堆相关算法

- **数据结构介绍**：二叉堆，即以数组的形式表示一个二叉树，常见的有大顶堆和小顶堆，优先级队列等，用于求最大/小的K个数很有效

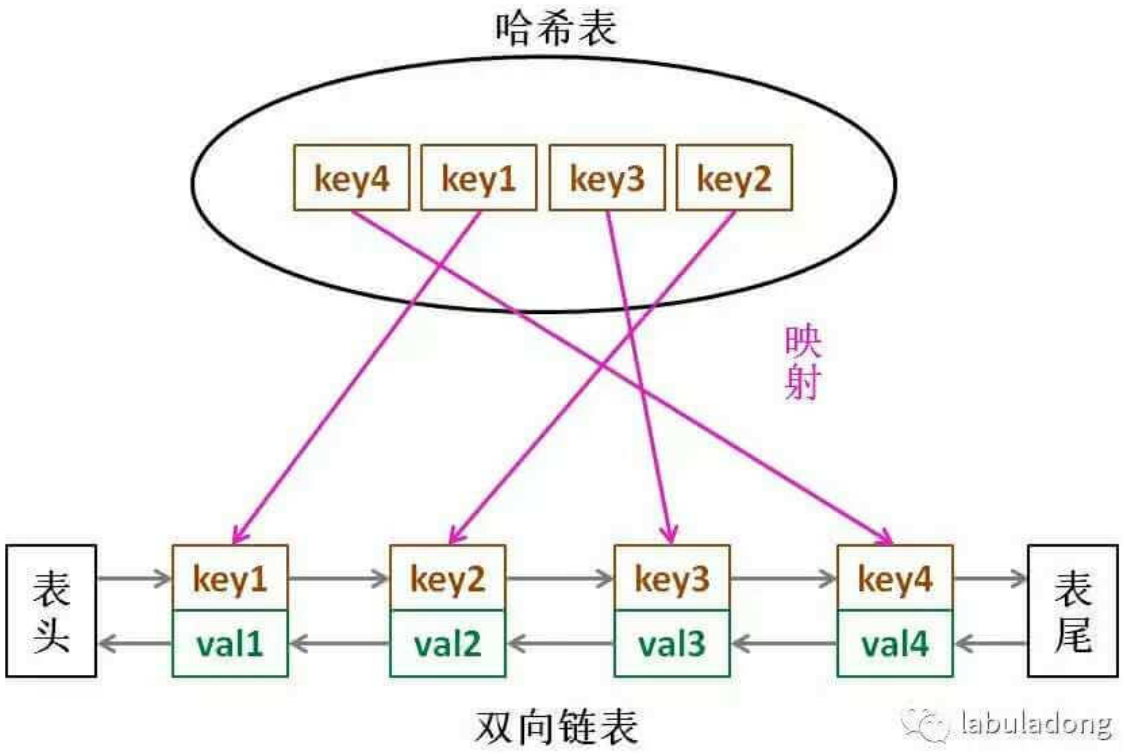
```
// 主要算法，上升和下降
public void swim(int index){
    int parent = index / 2;
    while (index != 0 && heap[index] > heap[parent]){
        swap(heap, index, parent);
        swim(parent);
    }
}

public void sink(int index){
    int left = index * 2 + 1;
    int right = index * 2 + 2;
    int max = heap[index];
    if (left < heap.size && max < heap[left]){
        max = heap[left];
    }
    if (right < heap.size && max < heap[right]){
        max = heap[right];
    }
```

```
    if (max != heap[index]){
        if (max == heap[left]){
            swap(heap, left, index);
            sink(left);
        }else{
            swap(heap, right, index);
            sink(right);
        }
    }
}
```

2）Least Recently Used——LRU最近最少使用

- **数据结构介绍**：一种缓存淘汰策略，当某种缓存需要清除部分空间时，使用该策略来将最近最少使用部分进行清除。要求put和get的时间复杂度为1，要想获取快，使用哈希表。要想插入删除快，使用双向链表。结合之后，该情况应该使用哈希链表。



3）二叉搜索树

- **数据结构介绍**：即BST，该类题型有标准框架，解题只需要考虑找到符合条件的节点时如何操作即可。

```
public void BST(TreeNode root, int target) {
    if (root.val == target)
        // 找到目标，做点什么
    if (root.val < target)
        BST(root.right, target);
    if (root.val > target)
        BST(root.left, target);
}
```

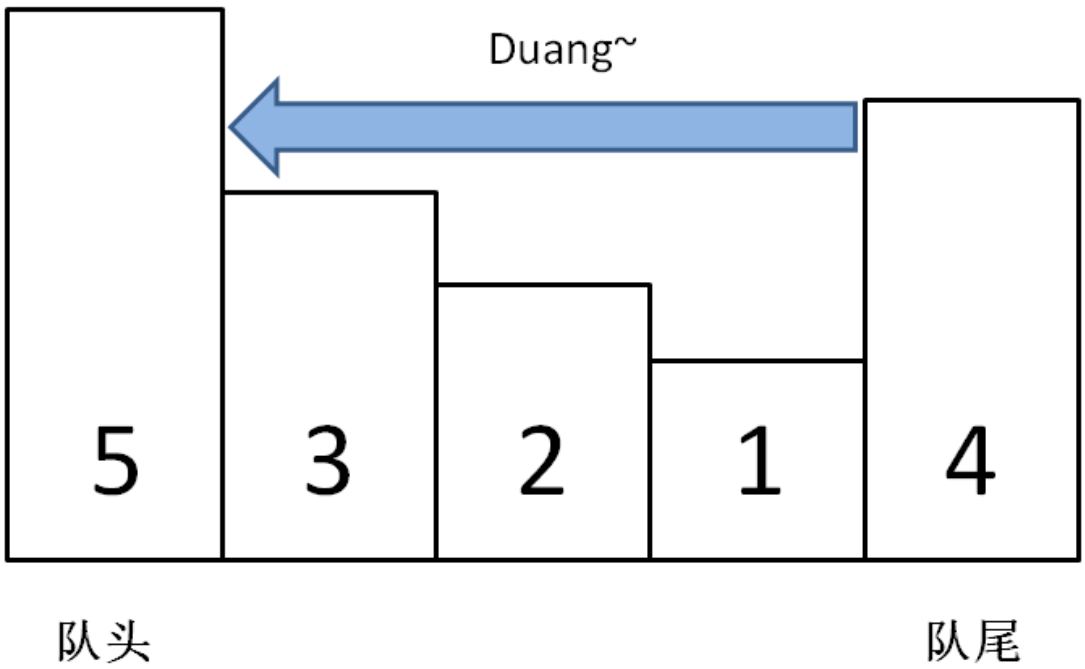
4）单调栈

- **数据结构介绍**：本质是栈，利用了一些巧妙的逻辑，使得每次新元素入栈后，栈内的元素都保持有序（单调递增或单调递减）。常用于解决类似“下一个更大元素”和“下一个更暖一天”。

```
vector<int> nextGreaterElement(vector<int>& nums) {
    vector<int> res(nums.size()); // 存放答案的数组
    stack<int> s;
    // 倒着往栈里放
    for (int i = nums.size() - 1; i >= 0; i--) {
        // 判定个子高矮
        while (!s.empty() && s.top() <= nums[i]) {
            // 矮个起开，反正也被挡着了。。。
            s.pop();
        }
        // nums[i] 身后的 next great number
        res[i] = s.empty() ? -1 : s.top();
        //
        s.push(nums[i]);
    }
    return res;
}
```

5）单调队列

- **数据结构介绍**：本质是队列，利用了一些巧妙的逻辑，使得每次新元素入队列后，队列内的元素都保持有序（单调递增或单调递减）。常用于解决类似“滑动窗口最大值”。




```
class MonotonicQueue {
private:
    deque<int> data;
public:
    void push(int n) {
        while (!data.empty() && data.back() < n)
```

```
            data.pop_back();
        data.push_back(n);
    }

    int max() { return data.front(); }

    void pop(int n) {
        if (!data.empty() && data.front() == n)
            data.pop_front();
    }
};

vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    MonotonicQueue window;
    vector<int> res;
    for (int i = 0; i < nums.size(); i++) {
        if (i < k - 1) { //先填满窗口的前 k - 1
            window.push(nums[i]);
        } else { // 窗口向前滑动
            window.push(nums[i]);
            res.push_back(window.max());
            window.pop(nums[i - k + 1]);
        }
    }
    return res;
}
```

6）递归反转链表

- **数据结构介绍**：通过递归的方式，在只扫描一次链表的要求下对链表进行反转

```
ListNode successor = null; // 后驱节点

// 反转以 head 为起点的 n 个节点，返回新的头结点
ListNode reverseN(ListNode head, int n) {
    if (n == 1) {
        // 记录第 n + 1 个节点
        successor = head.next;
        return head;
    }
    // 以 head.next 为起点，需要反转前 n - 1 个节点
    ListNode last = reverseN(head.next, n - 1);

    head.next.next = head;
    // 让反转之后的 head 节点和后面的节点连起来
    head.next = successor;
    return last;
}

ListNode reverseBetween(ListNode head, int m, int n) {
    // base case
    if (m == 1) {
        return reverseN(head, n);
    }
    // 前进到反转的起点触发 base case
    head.next = reverseBetween(head.next, m - 1, n - 1);
    return head;
```

```
}
```

7）队列实现栈，栈实现队列

- **思路**：2个栈可实现队列，1个队列即可实现栈

五）经典题目解法——算法思维

1）回溯法解决子集、组合和排列问题

- **框架**：给出子集框架，其余2个问题差不多

```
vector<vector<int>> res;

vector<vector<int>> subsets(vector<int>& nums) {
    // 记录走过的路径
    vector<int> track;
    backtrack(nums, 0, track);
    return res;
}

void backtrack(vector<int>& nums, int start, vector<int>& track) {
    res.push_back(track);
    for (int i = start; i < nums.size(); i++) {
        // 做选择
        track.push_back(nums[i]);
        // 回溯
        backtrack(nums, i + 1, track);
        // 撤销选择
        track.pop_back();
    }
}
```

2）滑动窗口技巧

- **常用形式**：类似于字符串的子串或者数组等相关问题，考虑滑动窗口有时比直接循环要效率更高
- **框架**：主要修改部分为valid的判断部分

```
int left = 0, right = 0;

while (right < s.size()) {
    window.add(s[right]);
    right++;

    while (valid) {
        window.remove(s[left]);
        left++;
    }
}
```

3）two sum问题另类思维

- **问题思路**：在给定的无序数组中查找两个和为目标的数，直接循环将达到n^2时间复杂度。由于确定了一个数之后，能达到和为目标的另一个数是确定的，因此改良的关键在于如何在数组中找到该数。因此考虑使用哈希表来存放数据，使得获取另一个书的效率变高。

```
int[] twoSum(int[] nums, int target) {
    int n = nums.length;
    index<Integer, Integer> index = new HashMap<>();
    // 构造一个哈希表：元素映射到相应的索引
    for (int i = 0; i < n; i++)
        index.put(nums[i], i);

    for (int i = 0; i < n; i++) {
        int other = target - nums[i];
        // 如果 other 存在且不是 nums[i] 本身
        if (index.containsKey(other) && index.get(other) != i)
            return new int[] {i, index.get(other)};
    }

    return new int[] {-1, -1};
}
```

4）烧饼排序

- **题目描述**：给你一个整数数组 `arr` ，请使用煎饼翻转完成对数组的排序。一次煎饼翻转的执行过程如下：选择一个整数 `k` ， `1 <= k <= arr.length` ，反转子数组 `arr[0...k-1]` （**下标从 0 开始**）。
- **问题思路**：采用递归的思路，每次递归过程为找出未归位数组中的最大数位置，将它翻转放置最后，然后再往前递归

```
// 记录反转操作序列
LinkedList<Integer> res = new LinkedList<>();

List<Integer> pancakeSort(int[] cakes) {
    sort(cakes, cakes.length);
    return res;
}

void sort(int[] cakes, int n) {
    // base case
    if (n == 1) return;
    // 寻找最大饼的索引
    int maxCake = 0;
    int maxCakeIndex = 0;
    for (int i = 0; i < n; i++)
        if (cakes[i] > maxCake) {
            maxCakeIndex = i;
            maxCake = cakes[i];
        }
    // 第一次翻转，将最大饼翻到最上面
    reverse(cakes, 0, maxCakeIndex);
    res.add(maxCakeIndex + 1);
    // 第二次翻转，将最大饼翻到最下面
    reverse(cakes, 0, n - 1);
```

```
    res.add(n);
    // 递归调用
    sort(cakes, n - 1);
}

void reverse(int[] arr, int i, int j) {
    while (i < j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
        i++; j--;
    }
}
```

5）前缀和技巧

- **题目描述**：给定一个整数数组和一个整数K，需要找出和为K的连续子数组的个数。
- **算法分析**：对于数组的连续子数组，为了避免多次循环，某些要求下可进行预处理，例如前缀和。同时，由于目标K是确定的，前缀和找到后另一个数也是确定的，因此可以使用哈希表减少搜索时间。

```
int subarraySum(int[] nums, int k) {
    int n = nums.length;
    // map：前缀和 -> 该前缀和出现的次数
    HashMap<Integer, Integer> preSum = new HashMap<>();
    // base case
    preSum.put(0, 1);

    int ans = 0, sum0_i = 0;
    for (int i = 0; i < n; i++) {
        sum0_i += nums[i];
        // 这是我们想找的前缀和 nums[0..j]
        int sum0_j = k - sum0_i;
        // 如果前面有这个前缀和，则直接更新答案
        if (preSum.containsKey(sum0_j))
            ans += preSum.get(sum0_j);
        // 把前缀和 nums[0..i] 加入并记录出现次数
        preSum.put(sum0_i,
            preSum.getOrDefault(sum0_i, 0) + 1);
    }
    return ans;
}
```

6）套娃信封问题

- **题目描述**：给你一个二维整数数组 envelopes ，其中 envelopes[i] = [wi, hi] ，表示第 i 个信封的宽度和高度。当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。请计算 最多能有多少个 信封能组成一组“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。
- **算法分析**：其实相当于最长递增子序列问题，区别在于这个是二维数组。为了使问题回到最长递增子序列问题，需要先对其中某一维进行排序。在这个题目中，可以选择先按宽度进行升序排序。同时，由于宽度相同时，即使高度不同也不能放入，即相同宽度只能选择一个，因此对相同宽度的信封进行高度的逆序排序。

```
// envelopes = [[w, h], [w, h]...]
```

```
public int maxEnvelopes(int[][] envelopes) {
    int n = envelopes.length;
    // 按宽度升序排列，如果宽度一样，则按高度降序排列
    Arrays.sort(envelopes, new Comparator<int[]>()
    {
        public int compare(int[] a, int[] b) {
            return a[0] == b[0] ?
                b[1] - a[1] : a[0] - b[0];
        }
    });
    // 对高度数组寻找 LIS
    int[] height = new int[n];
    for (int i = 0; i < n; i++)
        height[i] = envelopes[i][1];

    return lengthOfLIS(height);
}
```

六）高频面试算法

1）计数素数

- **算法分析**：由于非素数可以由素数乘以素数得到，因此可以先确定非素数有多少，剩下的都是素数，时间复杂度O(N * loglogN)

```
int countPrimes(int n) {
    boolean[] isPrim = new boolean[n];
    Arrays.fill(isPrim, true);
    for (int i = 2; i * i < n; i++)
        if (isPrim[i])
            for (int j = i * i; j < n; j += i)
                isPrim[j] = false;

    int count = 0;
    for (int i = 2; i < n; i++)
        if (isPrim[i]) count++;

    return count;
}
```

2）接雨水

- **问题描述**：给定 `n` 个非负整数表示每个宽度为 `1` 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。


```
int trap(vector<int>& height) {
    int n = height.size();
    int res = 0;
    for (int i = 1; i < n - 1; i++) {
        int l_max = 0, r_max = 0;
        // 找右边最高的柱子
        for (int j = i; j < n; j++)
            r_max = max(r_max, height[j]);
        // 找左边最高的柱子
        for (int j = i; j >= 0; j--)
            l_max = max(l_max, height[j]);
        // 如果自己就是最高的话，
        // l_max == r_max == height[i]
        res += min(l_max, r_max) - height[i];
    }
    return res;
}
```

3）去除有序数组的重复元素

- **算法分析**：对于数组来说，在尾部插入、删除元素是比较高效的，时间复杂度是 O(1)，但是如果在中间或者开头插入、删除元素，就会涉及数据的搬移，时间复杂度为 O(N)，效率较低。

4）最长回文子串

- **算法分析**：找回文子串应该以当前字符为中心（奇数子串）/ 中心的一部分（偶数子串），通过左右指针来确定包含当前字符的回文子串长度。

```
string palindrome(string& s, int l, int r) {
    // 防止索引越界
    while (l >= 0 && r < s.size()
            && s[l] == s[r]) {
        // 向两边展开
        l--; r++;
    }
    // 返回以 s[l] 和 s[r] 为中心的最长回文串
    return s.substr(l + 1, r - l - 1);
}

string longestPalindrome(string s) {
    string res;
    for (int i = 0; i < s.size(); i++) {
```

```
        // 以 s[i] 为中心的最长回文子串
        string s1 = palindrome(s, i, i);
        // 以 s[i] 和 s[i+1] 为中心的最长回文子串
        string s2 = palindrome(s, i, i + 1);
        // res = longest(res, s1, s2)
        res = res.size() > s1.size() ? res : s1;
        res = res.size() > s2.size() ? res : s2;
    }
    return res;
}
```

5）k个一组反转链表

- **算法分析**：链表是一种兼具递归和迭代性质的数据结构，因此涉及链表的问题应该考虑递归实现的可能性。

```
ListNode reverseKGroup(ListNode head, int k) {
    if (head == null) return null;
    // 区间 [a, b) 包含 k 个待反转元素
    ListNode a, b;
    a = b = head;
    for (int i = 0; i < k; i++) {
        // 不足 k 个，不需要反转，base case
        if (b == null) return head;
        b = b.next;
    }
    // 反转前 k 个元素
    ListNode newHead = reverse(a, b);
    // 递归反转后续链表并连接起来
    a.next = reverseKGroup(b, k);
    return newHead;
}
```

6）消失的元素

- **算法分析**：像这样元素和下标有对应关系的，可以考虑使用下标来优化代码，或避免整型溢出

```
public int missingNumber(int[] nums) {
    int n = nums.length;
    int res = 0;
    // 新补的索引
    res += n - 0;
    // 剩下索引和元素的差加起来
    for (int i = 0; i < n; i++)
        res += i - nums[i];
    return res;
}
```

7）判断回文链表

- **算法分析**：最简单的办法就是，把原始链表反转存入一条新的链表，然后比较这两条链表是否相同。同时，链表兼具递归结构，树结构不过是链表的衍生。那么，链表其实也可以有前序遍历和后序遍历，**借助二叉树后序遍历的思路，不需要显式反转原始链表也可以倒序遍历链表。**

```
// 左侧指针
```

```
ListNode left;

boolean isPalindrome(ListNode head) {
    left = head;
    return traverse(head);
}

boolean traverse(ListNode right) {
    if (right == null) return true;
    boolean res = traverse(right.next);
    // 后序遍历代码
    res = res && (right.val == left.val);
    left = left.next;
    return res;
}
```

8）水塘抽样算法

- **算法描述**：从未知长度的链表中，随机抽出K个数，保证链表中每个数被抽中的概率相等
- **水塘抽样算法详解**：证明略。对于从未知数组中抽取一个数时，操作如下：对于第i个数，有1/i的概率保留它，经过这样拓展，后续每个数出现的概率相等，且无需完整遍历整个数组，也不在乎数组长度是否已知。**同理，如果要随机选择 `k` 个数，对于前k个数，我们全部保留，对于第i（i>k）个数，我们以 $\frac{k}{i}$ 的概率保留第i个数，并以 $\frac{1}{k}$ 的概率与前面已选择的k个数中的任意一个替换。**

```
/* 返回链表中 k 个随机节点的值 */
int[] getRandom(ListNode head, int k) {
    Random r = new Random();
    int[] res = new int[k];
    ListNode p = head;

    // 前 k 个元素先默认选上
    for (int j = 0; j < k && p != null; j++) {
        res[j] = p.val;
        p = p.next;
    }

    int i = k;
    // while 循环遍历链表
    while (p != null) {
        // 生成一个 [0, i) 之间的整数
        int j = r.nextInt(++i);
        // 这个整数小于 k 的概率就是 k/i
        if (j < k) {
            res[j] = p.val;
        }
        p = p.next;
    }
    return res;
}
```

9）座位调度

- **题目描述**：假设有一个考场，考场有一排共 `N` 个座位，索引分别是 `[0..N-1]`，考生会**陆续**进入考场考试，并且可能在**任何时候**离开考场。你作为考官，要安排考生们的座位，满足：**每当一个学生进入时，你需要最大化他和最近其他人的距离；如果有多个这样的座位，安排到他到索引最小的那个座位。**
- **算法分析**：如果将每两个相邻的考生看做线段的两端点，新安排考生就是找最长的线段，然后让该考生在中间把这个线段「二分」，中点就是给他分配的座位。`leave(p)` 其实就是去除端点 `p`，使得相邻两个线段合并为一个。

```
// 将端点 p 映射到以 p 为左端点的线段
private Map<Integer, int[]> startMap;
// 将端点 p 映射到以 p 为右端点的线段
private Map<Integer, int[]> endMap;
// 根据线段长度从小到大存放所有线段
private TreeSet<int[]> pq;
private int N;

public ExamRoom(int N) {
    this.N = N;
    startMap = new HashMap<>();
    endMap = new HashMap<>();
    pq = new TreeSet<>((a, b) -> {
        int distA = distance(a);
        int distB = distance(b);
        // 如果长度相同，就比较索引
        if (distA == distB)
            return b[0] - a[0];
        return distA - distB;
    });
    // 在有序集合中先放一个虚拟线段
    addInterval(new int[] {-1, N});
}

/* 去除一个线段 */
private void removeInterval(int[] intv) {
    pq.remove(intv);
    startMap.remove(intv[0]);
    endMap.remove(intv[1]);
}

/* 增加一个线段 */
private void addInterval(int[] intv) {
    pq.add(intv);
    startMap.put(intv[0], intv);
    endMap.put(intv[1], intv);
}

/* 计算一个线段的长度 */
private int distance(int[] intv) {
    int x = intv[0];
    int y = intv[1];
    if (x == -1) return y;
    if (y == N) return N - 1 - x;
    // 中点和端点之间的长度
    return (y - x) / 2;
```

```
}

public int seat() {
    // 从有序集合拿出最长的线段
    int[] longest = pq.last();
    int x = longest[0];
    int y = longest[1];
    int seat;
    if (x == -1) { // 情况一
        seat = 0;
    } else if (y == N) { // 情况二
        seat = N - 1;
    } else { // 情况三
        seat = (y - x) / 2 + x;
    }
    // 将最长的线段分成两段
    int[] left = new int[] {x, seat};
    int[] right = new int[] {seat, y};
    removeInterval(longest);
    addInterval(left);
    addInterval(right);
    return seat;
}

public void leave(int p) {
    // 将 p 左右的线段找出来
    int[] right = startMap.get(p);
    int[] left = endMap.get(p);
    // 合并两个线段成为一个线段
    int[] merged = new int[] {left[0], right[1]};
    removeInterval(left);
    removeInterval(right);
    addInterval(merged);
}
```

10）Union-Find算法

- **算法介绍**：常说的并查集算法，主要是解决图论中「动态连通性」问题的
- **算法实现**：为了简单判断两个结点是否互相连通，可将结点表示为二叉树的形式，通过判断两个结点是否有共同的根结点来判断。如果某两个节点被连通，则让其中的（任意）一个节点的根节点接到另一个节点的根节点上。
- **算法优化**：通过转化为二叉树的形式，则实现将相关API接口时间复杂度降为logn，但仍然有极端情况出现。因此需要对形成的二叉树进行平衡性优化，小一些的树接到大一些的树下面，这样就能避免头重脚轻。最后，将路径进行压缩，使得树的高度达到常量级别。

```
class UF {
    // 连通分量个数
    private int count;
    // 存储一棵树
    private int[] parent;
    // 记录树的“重量”
    private int[] size;

    public UF(int n) {
        this.count = n;
        parent = new int[n];
```

```
        size = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i; // 自连接
            size[i] = 1;
        }
    }

    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ)
            return;

        // 小树接到大树下面，较平衡
        if (size[rootP] > size[rootQ]) {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        } else {
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        }
        count--;
    }

    public boolean connected(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        return rootP == rootQ;
    }

    private int find(int x) {
        while (parent[x] != x) {
            // 进行路径压缩，使得树高最高为3
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }

    public int count() {
        return count;
    }
}
```

11）二分查找高效判定子序列

- **算法分析**：判定字符串集合 `s` 中有多少个是字符串 `t` 的子序列，通过二分查找，可以优化效率。对 `t` 进行预处理，用一个字典 `index` 将每个字符出现的索引位置按顺序存储下来，借助 `index` 中记录的信息进行二分搜索

```
boolean isSubsequence(String s, String t) {
    int m = s.length(), n = t.length();
    // 对 t 进行预处理
    ArrayList<Integer>[] index = new ArrayList[256];
    for (int i = 0; i < n; i++) {
        char c = t.charAt(i);
```

```
            if (index[c] == null)
                index[c] = new ArrayList<>();
            index[c].add(i);
        }

        // 串 t 上的指针
        int j = 0;
        // 借助 index 查找 s[i]
        for (int i = 0; i < m; i++) {
            char c = s.charAt(i);
            // 整个 t 压根儿没有字符 c
            if (index[c] == null) return false;
            int pos = left_bound(index[c], j);
            // 二分搜索区间中没有找到字符 c
            if (pos == index[c].size()) return false;
            // 向前移动指针 j
            j = index[c].get(pos) + 1;
        }
        return true;
    }

    // 查找左侧边界的二分查找，由于要寻找恰好比tar大的索引，所以必须是左侧边界
    int left_bound(ArrayList<Integer> arr, int tar) {
        int lo = 0, hi = arr.size();
        while (lo < hi) {
            int mid = lo + (hi - lo) / 2;
            if (tar > arr.get(mid)) {
                lo = mid + 1;
            } else {
                hi = mid;
            }
        }
        return lo;
    }
```