**最长递增子序列（LIS）：**

给定一个长度为 n 的数组 A = [a1, a2, ..., an]，求其中最长递增子序列的长度。

例如：  
输入：[5, 6, 7, 1, 2, 8]  
输出：4，因为最长递增子序列是 [5,6,7,8]

1：最长公共子序列（LCS）

**原理：**

1. 复制数组 A 并升序排序为数组 X。
2. 计算 A 与 X 的 **LCS**，即为 LIS。

**示例：**

A = [5,6,7,1,2,8]  
X = [1,2,5,6,7,8]  
LCS(A, X) = [5,6,7,8]，长度为 4。

**2.动态规划：**

public int lengthOfLIS\_DP(int[] A) {

int n = A.length;

int[] dp = new int[n];

Arrays.fill(dp, 1); // 每个元素最少是自己

int maxLen = 1;

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

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

if (A[j] < A[i]) {

dp[i] = Math.max(dp[i], dp[j] + 1);

}

}

maxLen = Math.max(maxLen, dp[i]);

}

return maxLen;

}

**3.二分查找：**

维护一个数组 tail[]，含义为：

tail[i] 表示所有长度为 i+1 的递增子序列中，最小的结尾元素。

如果 num > tail[len-1]，则直接扩展新的递增序列；

否则使用二分查找，找到第一个 tail[i] >= num，替换之。

public int lengthOfLIS\_BinarySearch(int[] A) {

int[] tail = new int[A.length];

int len = 0;

for (int num : A) {

int left = 0, right = len;

while (left < right) {

int mid = (left + right) / 2;

if (tail[mid] < num) {

left = mid + 1;

} else {

right = mid;

}

}

tail[left] = num;

if (left == len) len++;

}

return len;

}

**2.环形石子问题：**

给定一圈石子，共 n 堆，每堆有若干个石子。每次只能合并**相邻的两堆**，花费为它们的石子数量之和。合并后成为一堆，继续合并，直到只剩一堆。求整个合并过程的**最小代价**或**最大代价**。

因为是环型，所以要变成线性，将数组复制一份接在后面，长度从 n 变为 2n。枚举所有长度为 n 的区间 [i, i+n-1]，求其中的最小/最大代价。

public class StoneMergeSimple {

public static int stoneMerge(int[] stones, boolean isMax) {

int n = stones.length;

int[] sum = new int[2 \* n + 1];

int[] arr = new int[2 \* n];

System.arraycopy(stones, 0, arr, 0, n);

System.arraycopy(stones, 0, arr, n, n);

for (int i = 1; i <= 2 \* n; i++) {

sum[i] = sum[i - 1] + arr[i - 1];

}

int[][] dp = new int[2 \* n][2 \* n];

for (int len = 2; len <= n; len++) {

for (int i = 0; i + len <= 2 \* n; i++) {

int j = i + len - 1;

dp[i][j] = isMax ? 0 : Integer.MAX\_VALUE;

for (int k = i; k < j; k++) {

int cost = dp[i][k] + dp[k + 1][j] + sum[j + 1] - sum[i];

if (isMax)

dp[i][j] = Math.max(dp[i][j], cost);

else

dp[i][j] = Math.min(dp[i][j], cost);

}

}

}

int res = isMax ? 0 : Integer.MAX\_VALUE;

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

int val = dp[i][i + n - 1];

res = isMax ? Math.max(res, val) : Math.min(res, val);

}

return res;

}

public static void main(String[] args) {

int[] stones = {4, 5, 9, 4};

System.out.println("最小得分: " + stoneMerge(stones, false));

System.out.println("最大得分: " + stoneMerge(stones, true));

}

}

01背包：

给定 n 个物品，每个物品体积为 weight[i]，价值为 value[i]，背包容量为 W，每个物品最多只能选一次。

public int knapsack01(int[] weights, int[] values, int W) {

int n = weights.length;

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

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

for (int j = W; j >= weights[i]; j--) {

dp[j] = Math.max(dp[j], dp[j - weights[i]] + values[i]);

}

}

return dp[W];

}

**数塔问题：**

我们需要同时记录路径，所以维护一个路径数组 path[i][j] 保存下一步的方向（0 = 左，1 = 右）。

int[] dp = new int[n];

System.arraycopy(triangle[n - 1], 0, dp, 0, n);

for (int i = n - 2; i >= 0; i--) {

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

dp[j] = Math.max(dp[j], dp[j + 1]) + triangle[i][j];

}

}

**反转方法：**

目标是把 a[0:k-1] 和 a[k:n-1] 对换为 a[k:n-1] + a[0:k-1]，要求 **时间 O(n)、空间 O(1)**。

第一步：反转前半段 reverse(a, 0, k - 1)

第二步：反转后半段 reverse(a, k, n - 1)

第三步：反转整体 reverse(a, 0, n - 1)

**递归：**

**台阶问题：记忆搜索**

**class Solution1143{**

**private char[] s,t;**

**private int[][] memo;**

**public int longestCommonSubsequence(String text1, String text2){**

**s =text1.toCharArray();**

**t = text2.toCharArray();**

**int n =s.length;**

**int m = t.length;**

**memo = new int[n][m];**

**for (int[] row:memo){**

**Arrays.fill(row,-1);**

**}**

**return dfs(n-1,m-1);**

**}**

**private int dfs(int i ,int j){**

**if (i<0||j<0) return 0;**

**if (memo[i][j]!=-1) return memo[i][j];**

**if (s[i]==t[j]) return memo[i][j] = dfs(i-1,j-1)+1;**

**return memo[i][j] = Math.max(dfs(i-1,j),dfs(i,j-1));**

**}**

**}**

**class Solution1143{**

**public int longestCommonSubsequenceB(String text1, String text2){**

**char[] t = text2.toCharArray();**

**int m = t.length;**

**int[] f = new int[m + 1];**

**for (char x:text1.toCharArray()){**

**int pre= 0;**

**for (int j=0;j<m;j++){**

**int tmp = f[j+1];**

**f[j+1] = x==t[j]?pre+1:Math.max(f[j+1],f[j]);**

**pre = tmp;**

**}**

**}**

**return f[m];**

**}**

**}**

**滑动窗口**

看到一串数组，要求数量的时候，可能就会用到滑动窗口

**定长**

只需要考虑进出即可

然后考虑窗口大小不足的时候,continue

模板：

 class Solution643{  
     public double findMaxAverage(int[] nums, int k){  
         int maxS = Integer.MIN\_VALUE;  
         int s=  0;  
         for (int i =0;i<nums.length;i++){  
             s +=nums[i];  
             if (i<k-1){  
                 continue;  
            }  
             maxS = Math.max(maxS,s);  
             s -=nums[i-k+1];  
        }  
         return (double) maxS/k;  
    }  
 }

头进尾部出，然后考虑变化

**不定长**

不定长的基本就是考虑窗口的缩小。

**求最值**

模板：

 class Solution2958{  
     public int maxSubarrayLength(int[] nums, int k){  
         int ans = 0, left= 0;  
         Map<Integer,Integer> cnt = new HashMap<>();  
         for (int right = 0;right<nums.length;right++){  
             cnt.merge(nums[right],1,Integer::sum);  
             while (cnt.get(nums[right])>k){  
                 cnt.merge(nums[left++],-1,Integer::sum);  
            }  
             ans = Math.max(ans,right-left+1);  
        }  
         return ans;  
    }  
 }

就是右边进入了，然后check一下，然后进行某些操作，然后左边出去

一般到最后需要的都是窗口的长度right-left+1

**求数目**

分为越长越合法，越短越合法和**恰好形**

**越长越合法**是指

[left,right] 这个子数组是不满足题目要求的，但在退出循环之前的最后一轮循环，**[left−1,right]** 是满足题目要求的。由于子数组越长，越能满足题目要求，所以除了 [left−1,right]，还有 [left−2,right],[left−3,right],…,[0,right] 都是满足要求的。也就是说，当右端点固定在 right 时，左端点在 0,1,2,…,left−1 的所有子数组都是满足要求的，这一共有 **left** 个。

**越短越合法**是指：

[left,right] 这个子数组是满足题目要求的。由于子数组越短，越能满足题目要求，所以除了 [left,right]，还有 **[left+1,right]**,[left+2,right],…,[right,right] 都是满足要求的。也就是说，当右端点固定在 right 时，左端点在 left,left+1,left+2,…,right 的所有子数组都是满足要求的，这一共有 **right−left+1** 个。

 class Solution713{  
     public int numSubarrayProductLessThanK(int[] nums, int k){  
         if (k<=1){  
             return 0;  
        }  
         int ans= 0;  
         int x = 1;  
         int left  = 0;  
         for (int right = 0;right<nums.length;right++){  
             x \*=nums[right];  
             while (x>=k){  
                 x /=nums[left++];  
            }  
             ans +=right-left+1;  
        }  
         return ans;  
    }  
 }

基本差不多，都是右边进去了，，然后经过check，然后左边出去，窗口缩小。‘

只不过返回的不同罢了

**恰好**就是正好为这个的数目

例如，要计算有多少个元素和恰好等于 k 的子数组，可以把问题变成：

计算有多少个元素和 ≥k 的子数组。 计算有多少个元素和 >k，也就是 ≥k+1 的子数组。 答案就是元素和 ≥k 的子数组个数，减去元素和 ≥k+1 的子数组个数。这里把 > 转换成 ≥，从而可以把滑窗逻辑封装成一个函数 f，然后用 **f(k) - f(k + 1)** 计算，无需编写两份滑窗代码。

总结：**「恰好」可以拆分成两个「至少」，也就是两个「越长越合法」的滑窗问题。**

注：也可以把问题变成 **≤k 减去 ≤k−1（两个至多）。可根据题目选择合适的变形方式。**

也可以把两个滑动窗口合并起来，维护同一个右端点 *right* 和两个左端点 *left*1 和 *left*2，我把这种写法叫做**三指针滑动窗口**。

一般来说一个函数写f(k) - f(k + 1)

另一个函数实现滑动窗口，就足够了。

 class Solution930{  
     public int numSubarraysWithSumA(int[] nums, int goal){  
         int ans1 = 0, left1=0,left2=0,ans2=0;  
         int sum1=0,sum2=0;  
         for (int right = 0;right<nums.length;right++){  
             sum1 +=nums[right];  
             while (sum1>=goal&&left1<=right){  
                 sum1 -=nums[left1++];  
            }  
             ans1 +=left1;  
             sum2 +=nums[right];  
             while (sum2>=goal+1&&left2<=right){  
                 sum2 -=nums[left2++];  
            }  
             ans2 +=left2;  
        }  
         return ans1-ans2;  
    }  
     public int numSubarraysWithSum(int[] nums, int goal){  
         return atMost(nums,goal)-atMost(nums,goal+1);  
    }  
     private int atMost(int[] nums, int goal){  
         int ans = 0,left = 0,sum = 0;  
         for (int right = 0;right<nums.length;right++){  
             sum +=nums[right];  
             while (sum>=goal&&left<=right){  
                 sum -=nums[left++];  
            }  
             ans +=left;  
        }  
         return ans;  
    }  
 }

**二分查找**

二分查找的原理就是取一个中间值，然后那中间值和目标值进行比较。

如果比目标值大的话，说明目标值在左边，中间值mid就变为右边right

相对应的，小于目标值的话，说明目标值在右边，中间值mid就变为left

二分查找的总结：

必须数组/序列是**有序的**，二分前必须先进行排序。

要确定搜索区间常见形式：[lo, hi]、[lo, hi)、(lo, hi]、(lo, hi)

确定开区间闭区间

开区间：

 private int lowerBound(int[] nums,int right,int target){  
         int left=-1;  
         while (left+1<right){  
             int mid = (left+right)>>>1;  
             if (nums[mid]>=target){  
                 right=mid;  
            }else {  
                 left=mid;  
            }  
        }  
         return right;  
    }

闭区间：

 class Solution{  
     public int searchInsert(int[] nums,int target){  
         int left = 0,right = nums.length-1;  
         while (left<=right){  
             int mid = (left+right)/2;  
             if (nums[mid] == target){  
                 return mid;  
            }  
             else if (nums[mid]<target){  
                 left =mid+1;  
            }  
             else {  
                 right = mid-1;  
            }  
        }  
         return left;  
    }  
 }

mid的取值：通常用 mid = lo + (hi - lo) / 2（或无符号右移 >>> 1）这样来防止溢出

还要设计check条件：

将问题转化为一个布尔函数 check(mid)，能准确告诉你“mid 是否满足某侧条件”。

根据 check(mid) 结果，把 lo 或 hi 缩到 mid 及其左／右一侧。

 int lo = L, hi = R;  
 while (lo < hi) {  
     int mid = lo + (hi - lo) / 2;  
     if (check(mid)) {  
         hi = mid;      // 保留 mid  
    } else {  
         lo = mid + 1;  // 丢弃 mid  
    }  
 }  
 return lo;  
 ​

lowerBound: 找到**第一个 ≥ target** 的索引

upperBound: 找到**第一个 > target** 的索引

| **场景** | **区间形式** | **备注** |
| --- | --- | --- |
| 查找某值 / 插入位置 | [0, n-1] | 经典闭区间；找不到时返回 lo 作为插入点 |
| lowerBound / upperBound | (-1, n] | 开区间；left = -1, right = n |
| 最接近元素（差值比较） | [0, n-k] | 窗口长度为 k，比较左右边界距离取决于差值大小 |
| 双指针对撞 | lo < hi | 例如找最大满足条件的下标 |

**可以使用查找某些值的问题，省去了遍历**

**双指针**

**单序列**

相向双指针：两个指针 *left*=0, *right*=*n*−1，从数组的两端开始，向中间移动，这叫**相向双指针**。上面的滑动窗口相当于**同向双指针**。

然后到中间去汇聚

一般来说条件是left<right

然后进行运行

**同向双指针**

两个指针的移动方向相同（都向右，或者都向左）。

从同一个方向开始

外层循环控制右指针，内层条件满足时移动左指针收缩窗口

就是滑动窗口哈哈

**背向双指针**

从中间开始往两边

**原地修改：**

主要是运用到了栈的思想

常用于数组原地修改，慢指针标记“结果区域”，快指针用于扫描

**原地修改+下标**

就是下标的问题

如果是顺序排序，不缺少元素的话，

|nums[i]| = index+1 防止负数哈

然后如果是乱序的话，也是可以对应起来的的

如果if (nums[index]>0){ nums[index] \*=-1; }

的话，缺少的那个正好是正数

然后要返回那个数的话，i+1即可

上面那个题，出现两次跟这个缺少是一样的

会把index变为负的

 1. 值域是 [1, n]，考虑 nums[i] - 1 做下标  
 2. 标记：将 nums[nums[i] - 1] \*= -1  
 3. 查询：  
    - 找缺失 ➜ 哪些 index 上还为正，对应值就是缺失的 i+1  
    - 找重复 ➜ 哪些 index 第一次访问时就已经是负数

**双序列**

**子序列问题**

还是标准的子序列问题，两个指针移动，当满足条件的时候子序列的指针移动

然后字典的一直移动

看最后子序列的指针能不能到末尾，到了就说明是，反则不是

 private boolean isSubseq(String s, String t){  
         int i =0;  
         for (char c:t.toCharArray()){  
             if (s.charAt(i)==c&&++i==s.length()){  
                 return true;  
            }  
        }  
      return false;  
    }

**递归与分治**

**原理**

* **递归**：一个问题可以被分解成结构相同的子问题，最终用子问题的解构造原问题的解。
* **分治**：将原问题分成若干规模较小的子问题，分别求解后合并子问题的解。

**核心三步**

1. 分解：将问题分解为子问题。
2. 求解：递归地求解子问题。
3. 合并：将子问题解合并为原问题解。

**模板**

java

复制编辑

ReturnType divideAndConquer(Problem problem) {

// 递归终止条件

if (problem is small) {

return directly solve;

}

// 分解子问题

SubProblem sub1 = divide(problem);

SubProblem sub2 = divide(problem);

// 求解子问题

ResultType result1 = divideAndConquer(sub1);

ResultType result2 = divideAndConquer(sub2);

// 合并子问题结果

return merge(result1, result2);

}

**典型例子**

* 归并排序
* 快速排序
* 斐波那契数列（递归）
* 二分查找

**二、动态规划（DP）**

**原理**

将一个复杂问题拆解成子问题，通过 **记录子问题的最优解** 来避免重复计算（**最优子结构 + 重叠子问题**）。

**核心步骤**

1. 状态定义：dp[i][j] 表示子问题的解
2. 状态转移方程：dp[i][j] = f(dp[i-1][j], dp[i][j-1]...)
3. 初始化边界
4. 遍历顺序（通常是递推）

**模板（二维）**

java

复制编辑

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

// 初始化 dp

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

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

dp[i][j] = Math.min(dp[i-1][j], dp[i][j-1]) + cost(i, j);

}

}

return dp[n][m];

**典型例子**

* 背包问题
* 最长公共子序列 / 最长上升子序列
* 矩阵路径
* 斐波那契（优化版）

**三、贪心算法**

**原理**

每一步都做 **当前最优选择**，期望通过局部最优达到全局最优（并不总是可行，需证明贪心正确性）。

**核心特征**

* 贪心选择性质
* 无后效性（当前选择不影响后续）

**模板**

java

复制编辑

List<Item> items = ...;

items.sort((a, b) -> a.xxx - b.xxx); // 按策略排序

for (Item item : items) {

if (能选择(item)) {

选择(item);

}

}

**典型例子**

* 活动选择问题
* 最小生成树（Prim/Kruskal）
* Huffman 编码
* Dijkstra 最短路
* 区间调度

**四、分支限界法（Branch & Bound）**

**原理**

在解决问题的过程中构造问题的解空间树，**使用上界/下界剪枝**，避免无意义的搜索，提高效率。

**适用场景**

* 解空间为树或排列组合类（通常用于**NP问题的优化**）
* 可设计合理的剪枝策略（界限函数）

**常见操作**

* 使用队列/优先队列按某种估值扩展结点
* 用当前最优解去剪枝（限界）
* 构造可行解，更新最优值

**模板**

java

复制编辑

class Node {

状态信息;

int bound; // 界限函数估值

}

PriorityQueue<Node> queue = new PriorityQueue<>((a, b) -> a.bound - b.bound);

queue.add(rootNode);

while (!queue.isEmpty()) {

Node node = queue.poll();

if (node.bound > 当前最优解) continue; // 剪枝

for (Node child : expand(node)) {

if (是可行解(child)) {

更新最优解;

} else {

queue.add(child);

}

}

}

**典型例子**

* 0-1 背包问题（使用最优价值上界剪枝）
* TSP（旅行商问题）
* 八皇后问题（结合剪枝）

**一、递归与分治**

**🧠 原理复述**

将问题分为多个子问题，递归地解决子问题并合并结果，常用于结构清晰的二叉树、排序、数学递推等问题。

**🔹题型一：排序问题**

* **归并排序**：分成左右两部分，各自排序再合并。
* **快速排序**：选一个基准值，划分后递归排序两边。

**🔹题型二：树形结构问题（后序、先序遍历，计算树高等）**

* **求二叉树最大深度**：左右子树分别计算高度，取最大+1。
* **后序遍历：左->右->根，天然递归结构**

**🔹题型三：数学递推（斐波那契）**

* 递归实现，但要小心栈溢出
* 可以与动态规划对比

**🔹题型四：组合生成（回溯 + 分治）**

* 子集、排列、括号生成等

**✅ 二、动态规划（DP）**

**🧠 原理复述**

通过状态转移表达子问题的依赖，记录子问题最优解避免重复计算。

**🔹题型一：背包问题（01背包，完全背包，多重背包）**

* 状态：dp[i][j] 表示前 i 个物品，总容量 j 的最大价值
* 一维滚动优化：dp[j] 逆序遍历

**🔹题型二：最长公共子序列/子串**

* 状态：dp[i][j] = max(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]+1)
* 滚动数组优化空间

**🔹题型三：区间DP**

* 括号匹配、戳气球、石子合并等
* 状态：dp[i][j] = min/max(dp[i][k] + dp[k+1][j] + cost)

**🔹题型四：数字划分、划分类**

* 状态如：dp[i][j] = dp[i-1][j] + dp[i][j-i]

**🔹题型五：路径统计（网格走法）**

* 向右和向下走：dp[i][j] = dp[i-1][j] + dp[i][j-1]
* 加上障碍判断

**✅ 三、贪心算法**

**🧠 原理复述**

在每一步选择中都采取当前最优解，从而达到全局最优（需证明贪心正确性）

**🔹题型一：区间调度类**

* 按结束时间升序排序
* 贪心选择不重叠的最多活动

**🔹题型二：最小/最大代价类**

* 按照性价比排序，局部最小/最大
* 加油问题、跳跃问题、K站中转

**🔹题型三：Huffman 编码**

* 每次取最小两个合并
* 使用优先队列维护

**🔹题型四：钱币找零类**

* 贪心策略：从最大币值开始
* 注意并非所有面值都适合贪心（要证明）

**✅ 四、分支限界法**

**🧠 原理复述**

在搜索树中，通过估值函数设定上下界，对不可行分支提前剪枝，提高效率。

**🔹题型一：0-1背包（带剪枝）**

* 与完全DP相比，控制空间节省
* 加上价值上界（剩余物品最大价值）

**🔹题型二：旅行商问题（TSP）**

* 搜索排列路径，设定最小路径界限
* 可结合优先队列优化搜索顺序

**✅ 模板题**

* TSP 分支限界法优化（ACM/洛谷区域赛模拟题）

**🔹题型三：八皇后 / 数独等约束类问题**

* 用排列、约束状态（如列、对角线）剪枝

回溯：

void dfs(int start, List<Integer> path) {

res.add(new ArrayList<>(path)); // 每个节点都是一个子集

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

if (i > start && nums[i] == nums[i - 1]) continue; // 去重

path.add(nums[i]);

dfs(i + 1, path);

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

}

}

**题型二：组合型问题（选k个数）**

* 特点：无顺序、不能重复选同一个元素、只要组合结果
* 常见 k 数之和问题、组合数、组合总和等

**🔹题型三：排列型问题（关注顺序）**

* 特点：选k个元素，顺序不同算不同结果，元素只能用一次
* 通常使用 visited[] 标记数组

java

复制编辑

void dfs(List<Integer> path, boolean[] visited) {

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

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

return;

}

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

if (visited[i]) continue;

if (i > 0 && nums[i] == nums[i - 1] && !visited[i - 1]) continue;

visited[i] = true;

path.add(nums[i]);

dfs(path, visited);

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

visited[i] = false;

}

}

**🔹题型四：棋盘类约束问题（N皇后、数独）**

* 特点：二维网格 + 约束
* 通常维护额外的列、对角线状态数组

**🔹题型五：图的路径问题（找所有路径、判断能否到达）**

* 图的 DFS + visited
* 可能与 backtracking 融合使用
* ）

**🔹题型六：分割问题**

* 字符串分割为若干部分，每部分满足某种规则