

# 面试常考算法模板 Cheat Sheet V4.0

## 前言

- 版权归属：九章算法（杭州）科技有限公司
- 可以原文转载和分享，转载时需保留此版权信息，不得对内容进行增删和修改
- 本文作者：九章算法令狐冲
- 官方网站：[www.jiuzhang.com/?utm\\_source=lhc-cheatsheet-v4.0](http://www.jiuzhang.com/?utm_source=lhc-cheatsheet-v4.0)

## 二分法 Binary Search

### 使用条件

1. 排序数组 (30-40%是二分)
2. 当面试官要求你找一个比  $O(n)$  更小的时间复杂度算法的时候(99%)
3. 找到数组中的一个分割位置，使得左半部分满足某个条件，右半部分不满足(100%)
4. 找到一个最大/最小的值使得某个条件被满足(90%)

### 复杂度

1. 时间复杂度： $O(\log n)$
2. 空间复杂度： $O(1)$

### 领扣例题

### 代码模版

Java

```
1 int binarySearch(int[] nums, int target) {
2     // corner case 处理
3     if (nums == null || nums.length == 0) {
4         return -1;
5     }
6
7     int start = 0, end = nums.length - 1;
8
9     // 要点1: start + 1 < end
10    while (start + 1 < end) {
```

```

11 // 要点2: start + (end - start) / 2
12     int mid = start + (end - start) / 2;
13     // 要点3: =, <, > 分开讨论, mid 不 +1 也不 -1
14     if (nums[mid] == target) {
15         return mid;
16     } else if (nums[mid] < target) {
17         start = mid;
18     } else {
19         end = mid;
20     }
21 }
22
23 // 要点4: 循环结束后, 单独处理start和end
24 if (nums[start] == target) {
25     return start;
26 }
27 if (nums[end] == target) {
28     return end;
29 }
30 return -1;
31 }

```

## Python

```

1 def binary_search(self, nums, target):
2     # corner case 处理
3     # 这里等价于 nums is None or len(nums) == 0
4     if not nums:
5         return -1
6
7     start, end = 0, len(nums) - 1
8
9     # 用 start + 1 < end 而不是 start < end 的目的是为了避免死循环
10    # 在 first position of target 的情况下不会出现死循环
11    # 但是在 last position of target 的情况下会出现死循环
12    # 样例: nums=[1, 1] target = 1
13    # 为了统一模板, 我们就都采用 start + 1 < end, 就保证不会出现死循环
14    while start + 1 < end:
15        # python 没有 overflow 的问题, 直接 // 2 就可以了
16        # java和C++ 最好写成 mid = start + (end - start) / 2
17        # 防止在 start = 2^31 - 1, end = 2^31 - 1 的情况下出现加法 overflow
18        mid = (start + end) // 2
19        # >, =, < 的逻辑先分开写, 然后在看看 = 的情况是否能合并到其他分支里
20        if nums[mid] < target:

```

```

21         start = mid
22     elif nums[mid] == target:
23         end = mid
24     else:
25         end = mid
26
27     # 因为上面的循环退出条件是 start + 1 < end
28     # 因此这里循环结束的时候, start 和 end 的关系是相邻关系 (1和2, 3和4这种)
29     # 因此需要再单独判断 start 和 end 这两个数谁是我们需要的答案
30     # 如果是找 first position of target 就先看 start, 否则就先看 end
31     if nums[start] == target:
32         return start
33     if nums[end] == target:
34         return end
35     return -1

```

## 双指针 Two Pointers

### 使用条件

### 复杂度

- 时间复杂度:  $O(n)$ 
  - 时间复杂度与最内层循环主体的执行次数有关
  - 与有多少重循环无关
- 空间复杂度:  $O(1)$ 
  - 只需要分配两个指针的额外内存

### 领扣例题

- LintCode 1879. 两数之和VII(同向双指针)
  - <https://www.lintcode.com/problem/two-sum-vii/description>
- LintCode1712.和相同的二元子数组(相向双指针)
  - <https://www.lintcode.com/problem/binary-subarrays-with-sum/description>
- LintCode627. 最长回文串 (背向双指针)
  - <https://www.lintcode.com/problem/longest-palindrome/description>
- LintCode 64: 合并有序数组
  - <https://www.lintcode.com/problem/merge-sorted-array>

### 代码模版

## Java

```
1 // 相向双指针(partition in quicksort)
2 public void partition(int[] A, int start, int end) {
3     if (start >= end) {
4         return;
5     }
6     int left = start, right = end;
7     // key point 1: pivot is the value, not the index
8     int pivot = A[(start + end) / 2];
9     // key point 2: every time you compare left & right, it should be
10    // left <= right not left < right
11    while (left <= right) {
12        while (left <= right && A[left] < pivot) {
13            left++;
14        }
15        while (left <= right && A[right] > pivot) {
16            right--;
17        }
18        if (left <= right) {
19            int temp = A[left];
20            A[left] = A[right];
21            A[right] = temp;
22            left++;
23            right--;
24        }
25    }
26 }
27
28 // 背向双指针
29 left = position;
30 right = position + 1;
31 while (left >= 0 && right < length) {
32     if (可以停下来了) {
33         break;
34     }
35     left--;
36     right++;
37 }
38
39 // 同向双指针
40 int j = 0;
41 for (int i = 0; i < n; i++) {
42     // 不满足则循环到满足搭配为止
43     while (j < n && i 到 j之间不满足条件) {
```

```

44     j += 1;
45 }
46 if (i 到 j之间满足条件) {
47     处理i, j这次搭配
48 }
49 }
50
51 // 合并双指针
52 ArrayList<Integer> merge(ArrayList<Integer> list1, ArrayList<Integer> list2) {
53     // 需要 new 一个新的 list, 而不是在 list1 或者 list2 上直接改动
54     ArrayList<Integer> newList = new ArrayList<Integer>();
55
56     int i = 0, j = 0;
57     while (i < list1.size() && j < list2.size()) {
58         if (list1.get(i) < list2.get(j)) {
59             newList.add(list1.get(i));
60             i++;
61         } else {
62             newList.add(list2.get(j));
63             j++;
64         }
65     }
66
67     // 合并上下的数到 newList 里
68     // 无需用 if (i < list1.size()), 直接 while 即可
69     while (i < list1.size()) {
70         newList.add(list1.get(i));
71         i++;
72     }
73     while (j < list2.size()) {
74         newList.add(list2.get(j));
75         j++;
76     }
77
78     return newList;
79 }

```

## Python

```

1 # 相向双指针(partition in quicksort)
2 def partition(self, A, start, end):
3     if start >= end:
4         return
5     left, right = start, end

```

```

6         # key point 1: pivot is the value, not the index
7         pivot = A[(start + end) // 2];
8         # key point 2: every time you compare left & right, it should be
9         # left <= right not left < right
10        while left <= right:
11            while left <= right and A[left] < pivot:
12                left += 1
13            while left <= right and A[right] > pivot:
14                right -= 1
15            if left <= right:
16                A[left], A[right] = A[right], A[left]
17                left += 1
18                right -= 1
19
20    # 背向双指针
21    left = position
22    right = position + 1
23    while left >= 0 and right < len(s):
24        if left 和 right 可以停下来了:
25            break
26        left -= 1
27        right += 1
28
29    # 同向双指针
30    j = 0
31    for i in range(n):
32        # 不满足则循环到满足搭配为止
33        while j < n and i到j之间不满足条件:
34            j += 1
35        if i到j之间满足条件:
36            处理i到j这段区间
37
38    # 合并双指针
39    def merge(list1, list2):
40        new_list = []
41        i, j = 0, 0
42
43        # 合并的过程只能操作 i, j 的移动, 不要去用 list1.pop(0) 之类的操作
44        # 因为 pop(0) 是 O(n) 的时间复杂度
45        while i < len(list1) and j < len(list2):
46            if list1[i] < list2[j]:
47                new_list.append(list1[i])
48                i += 1
49            else:

```

```

50         new_list.append(list2[j])
51         j += 1
52
53     # 合并剩下的数到 new_list 里
54     # 不要用 new_list.extend(list1[i:]) 之类的方法
55     # 因为 list1[i:] 会产生额外空间耗费
56     while i < len(list1):
57         new_list.append(list1[i])
58         i += 1
59     while j < len(list2):
60         new_list.append(list2[j])
61         j += 1
62
63     return new_list

```

## 二叉树分治 Binary Tree Divide & Conquer

### 使用条件

- 二叉树相关的问题 (99%)
- 可以一分为二去分别处理之后再合并结果 (100%)
- 数组相关的问题 (10%)

### 复杂度

时间复杂度  $O(n)$

空间复杂度  $O(n)$  (含递归调用的栈空间最大耗费)

### 领扣例题

- LintCode 1534. 将二叉搜索树转换为已排序的双向链接列表
  - <https://www.lintcode.com/problem/convert-binary-search-tree-to-sorted-doubly-linked-list/description>
- LintCode 94. 二叉树中的最大路径和
  - <https://www.lintcode.com/problem/binary-tree-maximum-path-sum/description>
- LintCode 95. 验证二叉查找树
  - <https://www.lintcode.com/problem/validate-binary-search-tree/description>

### 代码模板

## Java

```
1 public ResultType divideConquer(TreeNode node) {
2     // 递归出口
3     // 一般处理 node == null 就够了
4     // 大部分情况不需要处理 node == leaf
5     if (node == null) {
6         return ...;
7     }
8     // 处理左子树
9     ResultType leftResult = divideConquer(node.left);
10    // 处理右子树
11    ResultType rightResult = divideConquer(node.right);
12    // 合并答案
13    ResultType result = merge leftResult and rightResult
14    return result;
15 }
```

## Python

```
1 def divide_conquer(root):
2     # 递归出口
3     # 一般处理 node == null 就够了
4     # 大部分情况不需要处理 node == leaf
5     if root is None:
6         return ...
7     # 处理左子树
8     left_result = divide_conquer(node.left)
9     # 处理右子树
10    right_result = divide_conquer(node.right)
11    # 合并答案
12    result = merge left_result and right_result to get merged result
13    return result
```

# 二叉搜索树非递归 BST Iterator

## 使用条件

- 用非递归的方式（Non-recursion / Iteration）实现二叉树的中序遍历
- 常用于 BST 但不仅仅可以用于 BST

## 复杂度



时间复杂度  $O(n)$

空间复杂度  $O(n)$

## 领扣例题

- LintCode 67. 二叉树的中序遍历
  - <https://www.lintcode.com/problem/binary-tree-inorder-traversal/description>
- LintCode 902. 二叉搜索树的第 k 大元素
  - <https://www.lintcode.com/problem/kth-smallest-element-in-a-bst/description>

## 代码模板

Java

```
1 List<TreeNode> inorderTraversal(TreeNode root) {
2     List<TreeNode> inorder = new ArrayList<>();
3     if (root == null) {
4         return inorder;
5     }
6     // 创建一个 dummy node, 右指针指向 root
7     // 放到 stack 里, 此时栈顶 dummy 就是 iterator 的当前位置
8     TreeNode dummy = new TreeNode(0);
9     dummy.right = root;
10    Stack<TreeNode> stack = new Stack<>();
11    stack.push(dummy);
12
13    // 每次将 iterator 挪到下一个点
14    // 就是调整 stack 使得栈顶是下一个点
15    while (!stack.isEmpty()) {
16        TreeNode node = stack.pop();
17        inorder.add(node);
18        if (node.right != null) {
19            node = node.right;
20            while (node != null) {
21                stack.push(node);
22                node = node.left;
23            }
24        }
25    }
26    return inorder;
27 }
```

Python

```

1 def inorder_traversal(root):
2     if root is None:
3         return []
4
5     # 创建一个 dummy node, 右指针指向 root
6     # 并放到 stack 里, 此时 stack 的栈顶 dummy
7     # 是 iterator 的当前位置
8     dummy = TreeNode(0)
9     dummy.right = root
10    stack = [dummy]
11
12    inorder = []
13    # 每次将 iterator 挪到下一个点
14    # 也就是调整 stack 使得栈顶到下一个点
15    while stack:
16        node = stack.pop()
17        inorder.append(node.val)
18        if node.right:
19            node = node.right
20            while node:
21                stack.append(node)
22                node = node.left
23    return inorder

```

## 宽度优先搜索 BFS

### 使用条件

1. 拓扑排序(100%)
2. 出现连通块的关键词(100%)
3. 分层遍历(100%)
4. 简单图最短路径(100%)
5. 给定一个变换规则, 从初始状态变到终止状态最少几步(100%)

### 复杂度

- 时间复杂度:  $O(n + m)$ 
  - $n$  是点数,  $m$  是边数
- 空间复杂度:  $O(n)$

### 领扣例题

- LintCode 974. 01 矩阵(分层遍历)
  - <https://www.lintcode.com/problem/01-matrix/description>
- LintCode 431. 找无向图的连通块
  - <https://www.lintcode.com/problem/connected-component-in-undirected-graph/description>
- LintCode 127. 拓扑排序
  - <https://www.lintcode.com/problem/topological-sorting/description>

## 代码模版

### Java

```
1  ReturnType bfs(Node startNode) {
2      // BFS 必须要用队列 queue, 别用栈 stack!
3      Queue<Node> queue = new ArrayDeque<>();
4      // hashmap 有两个作用, 一个是记录一个点是否被丢进过队列了, 避免重复访问
5      // 另外一个作用是记录 startNode 到其他所有节点的最短距离
6      // 如果只求连通性的话, 可以换成 HashSet 就行
7      // node 做 key 的时候比较的是内存地址
8      Map<Node, Integer> distance = new HashMap<>();
9
10     // 把起点放进队列和哈希表里, 如果有多个起点, 都放进去
11     queue.offer(startNode);
12     distance.put(startNode, 0); // or 1 if necessary
13
14     // while 队列不空, 不停的从队列里拿出一个点, 拓展邻居节点放到队列中
15     while (!queue.isEmpty()) {
16         Node node = queue.poll();
17         // 如果有明确的终点可以在这里加终点的判断
18         if (node 是终点) {
19             break or return something;
20         }
21         for (Node neighbor : node.getNeighbors()) {
22             if (distance.containsKey(neighbor)) {
23                 continue;
24             }
25             queue.offer(neighbor);
26             distance.put(neighbor, distance.get(node) + 1);
27         }
28     }
29     // 如果需要返回所有点离起点的距离, 就 return hashmap
30     return distance;
31     // 如果需要返回所有连通的节点, 就 return HashMap 里的所有点
32     return distance.keySet();
```

```

33 // 如果需要返回离终点的最短距离
34 return distance.get(endNode);
35 }

```

## Python

```

1 def bfs(start_node):
2     # BFS 必须要用队列 queue, 别用栈 stack!
3     # distance(dict) 有两个作用, 一个是记录一个点是否被丢进过队列了, 避免重复访问
4     # 另外一个作用是记录 start_node 到其他所有节点的最短距离
5     # 如果只求连通性的话, 可以换成 set 就行
6     # node 做 key 的时候比较的是内存地址
7     queue = collections.deque([start_node])
8     distance = {start_node: 0}
9
10    # while 队列不空, 不停的从队列里拿出一个点, 拓展邻居节点放到队列中
11    while queue:
12        node = queue.popleft()
13        # 如果有明确的终点可以在这里加终点的判断
14        if node 是终点:
15            break or return something
16        for neighbor in node.get_neighbors():
17            if neighbor in distance:
18                continue
19            queue.append(neighbor)
20            distance[neighbor] = distance[node] + 1
21
22    # 如果需要返回所有点离起点的距离, 就 return hashmap
23    return distance
24    # 如果需要返回所有连通的节点, 就 return HashMap 里的所有点
25    return distance.keys()
26    # 如果需要返回离终点的最短距离
27    return distance[end_node]

```

## Java 拓扑排序 BFS 模板

```

1 List<Node> topologicalSort(List<Node> nodes) {
2     // 统计所有点的入度信息, 放入 hashmap 里
3     Map<Node, Integer> indegrees = getIndegrees(nodes);
4
5     // 将所有入度为 0 的点放到队列中
6     Queue<Node> queue = new ArrayDeque<>();
7     for (Node node : nodes) {
8         if (indegrees.get(node) == 0) {

```

```

9         queue.offer(node);
10    }
11 }
12
13 List<Node> topoOrder = new ArrayList<>();
14 while (!queue.isEmpty()) {
15     Node node = queue.poll();
16     topoOrder.add(node);
17     for (Node neighbor : node.getNeighbors()) {
18         // 入度减一
19         indegrees.put(neighbor, indegrees.get(neighbor) - 1);
20         // 入度减到0说明不再依赖任何点，可以被放到队列（拓扑序）里了
21         if (indegrees.get(neighbor) == 0) {
22             queue.offer(neighbor);
23         }
24     }
25 }
26
27 // 如果 queue 是空的时候，图中还有点没有被挖出来，说明存在环
28 // 有环就没有拓扑序
29 if (topoOrder.size() != nodes.size()) {
30     return 没有拓扑序;
31 }
32 return topoOrder;
33 }
34
35 Map<Node, Integer> getIndegrees(List<Node> nodes) {
36     Map<Node, Integer> counter = new HashMap<>();
37     for (Node node : nodes) {
38         counter.put(node, 0);
39     }
40     for (Node node : nodes) {
41         for (Node neighbor : node.getNeighbors()) {
42             counter.put(neighbor, counter.get(neighbor) + 1);
43         }
44     }
45     return counter;
46 }

```

## Python

```

1 def get_indegrees(nodes):
2     counter = {node: 0 for node in nodes}
3     for node in nodes:

```

```

4         for neighbor in node.get_neighbors():
5             counter[neighbor] += 1
6         return counter
7
8 def topological_sort(nodes):
9     # 统计入度
10    indegrees = get_indegrees(nodes)
11    # 所有入度为 0 的点都放到队列里
12    queue = collections.deque([
13        node
14        for node in nodes
15        if indegrees[node] == 0
16    ])
17    # 用 BFS 算法一个个把点从图里挖出来
18    topo_order = []
19    while queue:
20        node = queue.popleft()
21        topo_order.append(node)
22        for neighbor in node.get_neighbors():
23            indegrees[neighbor] -= 1
24            if indegrees[neighbor] == 0:
25                queue.append(neighbor)
26    # 判断是否有循环依赖
27    if len(topo_order) != len(nodes):
28        return 有循环依赖(环),没有拓扑序
29    return topo_order

```

## 深度优先搜索 DFS

### 使用条件

- 找满足某个条件的所有方案 (99%)
- 二叉树 Binary Tree 的问题 (90%)
- 组合问题(95%)
  - 问题模型：求出所有满足条件的“组合”
  - 判断条件：组合中的元素是顺序无关的
- 排列问题 (95%)
  - 问题模型：求出所有满足条件的“排列”
  - 判断条件：组合中的元素是顺序“相关”的。

### 不要用 DFS 的场景

1. 连通块问题（一定要用 BFS，否则 StackOverflow）
2. 拓扑排序（一定要用 BFS，否则 StackOverflow）
3. 一切 BFS 可以解决的问题

## 复杂度

- 时间复杂度：O(方案个数 \* 构造每个方案的时间)
  - 树的遍历：O(n)
  - 排列问题：O(n! \* n)
  - 组合问题：O(2^n \* n)

## 领扣例题

- LintCode 67. 二叉树的中序遍历(遍历树)
  - <https://www.lintcode.com/problem/binary-tree-inorder-traversal/description>
- LintCode 652. 因式分解(枚举所有情况)
  - <https://www.lintcode.com/problem/factorization/description>

## 代码模版

### Java

```
1 public ReturnType dfs(参数列表) {
2     if (递归出口) {
3         记录答案;
4         return;
5     }
6     for (所有的拆解可能性) {
7         修改所有的参数
8         dfs(参数列表);
9         还原所有被修改过的参数
10    }
11    return something 如果需要的话，很多时候不需要 return 值除了分治的写法
12 }
```

### Python

```
1 def dfs(参数列表):
2     if 递归出口:
3         记录答案
4         return
5     for 所有的拆解可能性:
6         修改所有的参数
7         dfs(参数列表)
```

## 动态规划 Dynamic Programming

### 使用条件

- 使用场景:
  - 求方案总数(90%)
  - 求最值(80%)
  - 求可行性(80%)
- 不适用的场景:
  - 找所有具体的方案 (准确率99%)
  - 输入数据无序(除了背包问题外, 准确率60%~70%)
  - 暴力算法已经是多项式时间复杂度 (准确率80%)
- 动态规划四要素(对比递归的四要素):
  - 状态 (State) -- 递归的定义
  - 方程 (Function) -- 递归的拆解
  - 初始化 (Initialization) -- 递归的出口
  - 答案 (Answer) -- 递归的调用
- 几种常见的动态规划:
- 背包型
  - 给出  $n$  个物品及其大小,问是否能挑选出一些物品装满大小为 $m$ 的背包
  - 题目中通常有“和”与“差”的概念,数值会被放到状态中
  - 通常是二维的状态数组,前  $i$  个组成和为  $j$  状态数组的大小需要开  $(n + 1) * (m + 1)$
  - 几种背包类型:

#### ■ 01背包

- 状态 state

`dp[i][j]` 表示前  $i$  个数里挑若干个数是否能组成和为  $j$

方程 function

`dp[i][j] = dp[i - 1][j] or dp[i - 1][j - A[i - 1]]` 如果  $j \geq A[i - 1]$

`dp[i][j] = dp[i - 1][j]` 如果  $j < A[i - 1]$

第  $i$  个数的下标是  $i - 1$ , 所以用的是 `A[i - 1]` 而不是 `A[i]`

初始化 initialization

`dp[0][0] = true`

`dp[0][1...m] = false`

答案 answer



使得  $dp[n][v]$ ,  $0 \leq v \leq m$  为 true 的最大  $v$

- 多重背包

- 状态 state

$dp[i][j]$  表示前  $i$  个物品挑出一些放到  $j$  的背包里的最大价值和

方程 function

$dp[i][j] = \max(dp[i-1][j - \text{count} * A[i-1]] + \text{count} * V[i-1])$

其中  $0 \leq \text{count} \leq j / A[i-1]$

初始化 initialization

$dp[0][0..m] = 0$

答案 answer

$dp[n][m]$

- 区间型

- 题目中有 subarray / substring 的信息

- 大区间依赖小区间

- 用  $dp[i][j]$  表示数组/字符串中  $i, j$  这一段区间的最优值/可行性/方案总数

- 状态 state

$dp[i][j]$  表示数组/字符串中  $i, j$  这一段区间的最优值/可行性/方案总数

方程 function

$dp[i][j] = \max/\min/\text{sum}/\text{or}(dp[i, j \text{ 之内更小的若干区间}])$

- 匹配型

- 通常给出两个字符串

- 两个字符串的匹配值依赖于两个字符串前缀的匹配值

- 字符串长度为  $n, m$  则需要开  $(n+1) \times (m+1)$  的状态数组

- 要初始化  $dp[i][0]$  与  $dp[0][i]$

- 通常都可以用滚动数组进行空间优化

- 状态 state

$dp[i][j]$  表示第一个字符串的前  $i$  个字符与第二个字符串的前  $j$  个字符怎么样怎么样  
( $\max/\min/\text{sum}/\text{or}$ )

- 划分型

- 是前缀型动态规划的一种, 有前缀的思想

- 如果指定了要划分为几个部分:

- $dp[i][j]$  表示前  $i$  个数/字符划分为  $j$  个部分的最优值/方案数/可行性

- 如果没有指定划分为几个部分:

- $dp[i]$  表示前  $i$  个数/字符划分为若干个部分的最优值/方案数/可行性

- 状态 state

指定了要划分为几个部分:  $dp[i][j]$  表示前  $i$  个数/字符划分为  $j$  个部分的最优值/方案数/可行性

没有指定划分为几个部分:  $dp[i]$  表示前  $i$  个数/字符划分为若干个部分的最优值/方案数/可行性

#### · 接龙型

- 通常会给一个接龙规则，问你最长的龙有多长
- 状态表示通常为:  $dp[i]$  表示以坐标为  $i$  的元素结尾的最长龙的长度
- 方程通常是:  $dp[i] = \max\{dp[j] + 1, j \text{ 的后面可以接上 } i\}$
- LIS 的二分做法选择性的掌握，但并不是所有的接龙型DP都可以用二分来优化
- 状态 state

状态表示通常为:  $dp[i]$  表示以坐标为  $i$  的元素结尾的最长龙的长度

方程 function

$dp[i] = \max\{dp[j] + 1, j \text{ 的后面可以接上 } i\}$

## 复杂度

#### · 时间复杂度:

- $O(\text{状态总数} * \text{每个状态的处理耗费})$
- 等于  $O(\text{状态总数} * \text{决策数})$

#### · 空间复杂度:

- $O(\text{状态总数})$  (不使用滚动数组优化)
- $O(\text{状态总数} / n)$  (使用滚动数组优化,  $n$  是被滚动掉的那一个维度)

## 领扣例题

#### · LintCode563. 背包问题V(背包型):

- <https://www.lintcode.com/problem/backpack-v/description>

#### · LintCode76. 最长上升子序列(接龙型):

- <https://www.lintcode.com/problem/longest-increasing-subsequence/description>

#### · LintCode 476. 石子归并V(区间型):

- <https://www.lintcode.com/problem/stone-game/description>

#### · LintCode 192. 通配符匹配 (匹配型):

- <https://www.lintcode.com/problem/wildcard-matching/description>

#### · LintCode107. 单词拆分(划分型):

- <https://www.lintcode.com/problem/word-break/description>

## 堆 Heap

## 使用条件

1. 找最大值或者最小值(60%)
2. 找第 k 大(pop k 次 复杂度 $O(n\log k)$ )(50%)
3. 要求  $\log n$  时间对数据进行操作(40%)

## 堆不能解决的问题

1. 查询比某个数大的最小值/最接近的值 (平衡排序二叉树 Balanced BST 才可以解决)
2. 找某段区间的最大值最小值 (线段树 SegmentTree 可以解决)
3.  $O(n)$ 找第k大 (使用快排中的partition操作)

## 领扣例题

- LintCode 1274. 查找和最小的K对数字
  - <https://www.lintcode.com/problem/find-k-pairs-with-smallest-sums/description>
- LintCode 919. 会议室 II
  - <https://www.lintcode.com/problem/meeting-rooms-ii/description>
- LintCode 1512. 雇佣K个人的最低费用
  - <https://www.lintcode.com/problem/minimum-cost-to-hire-k-workers/description>

## 代码模板

Java 带删除特定元素功能的堆

```
1 class ValueIndexPair {
2     int val, index;
3     public ValueIndexPair(int val, int index) {
4         this.val = val;
5         this.index = index;
6     }
7 }
8 class Heap {
9     private Queue<ValueIndexPair> minheap;
10    private Set<Integer> deleteSet;
11    public Heap() {
12        minheap = new PriorityQueue<>((p1, p2) -> (p1.val - p2.val));
13        deleteSet = new HashSet<>();
14    }
15
16    public void push(int index, int val) {
17        minheap.add(new ValueIndexPair(val, index));
18    }
19 }
```

```

20     private void lazyDeletion() {
21         while (minheap.size() != 0 && deleteSet.contains(minheap.peek().index)) {
22             ValueIndexPair pair = minheap.poll();
23             deleteSet.remove(pair.index);
24         }
25     }
26
27     public ValueIndexPair top() {
28         lazyDeletion();
29         return minheap.peek();
30     }
31
32     public void pop() {
33         lazyDeletion();
34         minheap.poll();
35     }
36
37     public void delete(int index) {
38         deleteSet.add(index);
39     }
40
41     public boolean isEmpty() {
42         return minheap.size() == 0;
43     }
44 }

```

## Python 带删除特定元素功能的堆

```

1  from heapq import heappush, heappop
2
3  class Heap:
4
5      def __init__(self):
6          self.minheap = []
7          self.deleted_set = set()
8
9      def push(self, index, val):
10         heappush(self.minheap, (val, index))
11
12     def _lazy_deletion(self):
13         while self.minheap and self.minheap[0][1] in self.deleted_set:
14             heappop(self.minheap)
15
16     def top(self):

```

```
17         self._lazy_deletion()
18         return self.minheap[0]
19
20     def pop(self):
21         self._lazy_deletion()
22         heappop(self.minheap)
23
24     def delete(self, index):
25         self.deleted_set.add(index)
26
27     def is_empty(self):
28         return not bool(self.minheap)
```

## 并查集 Union Find

### 使用条件

- 需要查询图的连通状况的问题
- 需要支持快速合并两个集合的问题

### 复杂度

- 时间复杂度 union  $O(1)$ , find  $O(1)$
- 空间复杂度  $O(n)$

### 领扣例题

- LintCode 1070. 账号合并
  - <https://www.lintcode.com/problem/accounts-merge/>
- LintCode 1014. 打砖块
  - <https://www.lintcode.com/problem/bricks-falling-when-hit/>
- LintCode 1813. 构造二叉树
  - <https://www.lintcode.com/problem/construct-binary-tree/>

### 代码模板

java

```
1 class UnionFind {
2     private Map<Integer, Integer> father;
3     private Map<Integer, Integer> sizeOfSet;
4     private int numOfSet = 0;
5     public UnionFind() {
```

```

6      // 初始化父指针, 集合大小, 集合数量
7      father = new HashMap<Integer, Integer>();
8      sizeOfSet = new HashMap<Integer, Integer>();
9      numOfSet = 0;
10 }
11
12 public void add(int x) {
13     // 点如果已经出现, 操作无效
14     if (father.containsKey(x)) {
15         return;
16     }
17     // 初始化点的父亲为 空对象null
18     // 初始化该点所在集合大小为 1
19     // 集合数量增加 1
20     father.put(x, null);
21     sizeOfSet.put(x, 1);
22     numOfSet++;
23 }
24
25 public void merge(int x, int y) {
26     // 找到两个节点的根
27     int rootX = find(x);
28     int rootY = find(y);
29     // 如果根不是同一个则连接
30     if (rootX != rootY) {
31         // 将一个点的根变成新的根
32         // 集合数量减少 1
33         // 计算新的根所在集合大小
34         father.put(rootX, rootY);
35         numOfSet--;
36         sizeOfSet.put(rootY, sizeOfSet.get(rootX) + sizeOfSet.get(rootY));
37     }
38 }
39
40 public int find(int x) {
41     // 指针 root 指向被查找的点 x
42     // 不断找到 root 的父亲
43     // 直到 root 指向 x 的根节点
44     int root = x;
45     while (father.get(root) != null) {
46         root = father.get(root);
47     }
48     // 将路径上所有点指向根节点 root
49     while (x != root) {

```

```

50         // 暂存 x 原本的父亲
51         // 将 x 指向根节点
52         // x 指针上移至 x 的父节点
53         int originalFather = father.get(x);
54         father.put(x, root);
55         x = originalFather;
56     }
57
58     return root;
59 }
60
61 public boolean isConnected(int x, int y) {
62     // 两个节点连通 等价于 两个节点的根相同
63     return find(x) == find(y);
64 }
65
66 public int getNumOfSet() {
67     // 获得集合数量
68     return numOfSet;
69 }
70
71 public int getSizeOfSet(int x) {
72     // 获得某个点所在集合大小
73     return sizeOfSet.get(find(x));
74 }
75 }

```

## Python

```

1 class UnionFind:
2     def __init__(self):
3         # 初始化父指针, 集合大小, 集合数量
4         self.father = {}
5         self.size_of_set = {}
6         self.num_of_set = 0
7
8     def add(self, x):
9         # 点如果已经出现, 操作无效
10        if x in self.father:
11            return
12        # 初始化点的父亲为 空对象None
13        # 初始化该点所在集合大小为 1
14        # 集合数量增加 1
15        self.father[x] = None

```

```
16         self.num_of_set += 1
17         self.size_of_set[x] = 1
18
19     def merge(self, x, y):
20         # 找到两个节点的根
21         root_x, root_y = self.find(x), self.find(y)
22         # 如果根不是同一个则连接
23         if root_x != root_y:
24             # 将一个点的根变成新的根
25             # 集合数量减少 1
26             # 计算新的根所在集合大小
27             self.father[root_x] = root_y
28             self.num_of_set -= 1
29             self.size_of_set[root_y] += self.size_of_set[root_x]
30
31     def find(self, x):
32         # 指针 root 指向被查找的点 x
33         # 不断找到 root 的父亲
34         # 直到 root 指向 x 的根节点
35         root = x
36         while self.father[root] != None:
37             root = self.father[root]
38         # 将路径上所有点指向根节点 root
39         while x != root:
40             # 暂存 x 原本的父亲
41             # 将 x 指向根节点
42             # x 指针上移至 x 的父节点
43             original_father = self.father[x]
44             self.father[x] = root
45             x = original_father
46         return root
47
48     def is_connected(self, x, y):
49         # 两个节点连通 等价于 两个节点的根相同
50         return self.find(x) == self.find(y)
51
52     def get_num_of_set(self):
53         # 获得集合数量
54         return self.num_of_set
55
56     def get_size_of_set(self, x):
57         # 获得某个点所在集合大小
58         return self.size_of_set[self.find(x)]
```



# 字典树 Trie

## 使用条件

- 需要查询包含某个前缀的单词/字符串是否存在
- 字符矩阵中找单词的问题

## 复杂度

- 时间复杂度  $O(L)$  增删查改
- 空间复杂度  $O(N * L)$   $N$  是单词数， $L$  是单词长度

## 领扣例题

- LintCode 1221. 连接词
  - <https://www.lintcode.com/problem/concatenated-words/>
- LintCode 1624. 最大距离
  - <https://www.lintcode.com/problem/max-distance/>
- LintCode 1090. 映射配对之和
  - <https://www.lintcode.com/problem/map-sum-pairs/>

## 代码模板

java

```
1 class TrieNode {
2     // 儿子节点
3     public Map<Character, TrieNode> children;
4     // 根节点到该节点是否是一个单词
5     public boolean isWord;
6     // 根节点到该节点的单词是什么
7     public String word;
8     public TrieNode() {
9         sons = new HashMap<Character, TrieNode>();
10        isWord = false;
11        word = null;
12    }
13 }
14
15 public class Trie {
16     private TrieNode root;
17     public Trie() {
18         root = new TrieNode();
```

```
19     }
20
21     public TrieNode getRoot() {
22         return root;
23     }
24
25     // 插入单词
26     public void insert(String word) {
27         TrieNode node = root;
28         for (int i = 0; i < word.length(); i++) {
29             char letter = word.charAt(i);
30             if (!node.sons.containsKey(letter)) {
31                 node.sons.put(letter, new TrieNode());
32             }
33             node = node.sons.get(letter);
34         }
35         node.isWord = true;
36         node.word = word;
37     }
```

```
38
39     // 判断单词 word 是不是在字典树中
40     public boolean hasWord(String word) {
41         int L = word.length();
42         TrieNode node = root;
43         for (int i = 0; i < L; i++) {
44             char letter = word.charAt(i);
45             if (!node.sons.containsKey(letter)) {
46                 return false;
47             }
48             node = node.sons.get(letter);
49         }
50
51         return node.isWord;
52     }
```

```
53
54     // 判断前缀 prefix 是不是在字典树中
55     public boolean hasPrefix(String prefix) {
56         int L = prefix.length();
57         TrieNode node = root;
58         for (int i = 0; i < L; i++) {
59             char letter = prefix.charAt(i);
60             if (!node.sons.containsKey(letter)) {
61                 return false;
62             }
```

```
63         node = node.sons.get(letter);
64     }
65     return true;
66 }
67 }
```