面试常考算法模板 Cheat Sheet V4.3

面试常考算法模板 Cheat Sheet V4.3

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

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

5. 时间复杂度：O(logn)
6. 空间复杂度：O(1)

领扣例题

- LintCode 14. 二分查找(在排序的数据集上进行二分)
- LintCode 460. 在排序数组中找最接近的K个数 (在未排序的数据集上进行二分)
- LintCode 437. 书籍复印(在答案集上进行二分)

代码模版

Java

```
1 int binarySearch(int[] nums, int target) {
```

```
    // corner case 处理
    if (nums == null || nums.length == 0) {
        return -1;
    }

    int start = 0, end = nums.length - 1;

    // 要点1: start + 1 < end
    while (start + 1 < end) {
     // 要点2: start + (end - start) / 2
        int mid = start + (end - start) / 2;
        // 要点3: =, <, > 分开讨论，mid 不 +1 也不 -1
        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            start = mid;
        } else {
            end = mid;
        }
    }

    // 要点4: 循环结束后，单独处理start和end
    if (nums[start] == target) {
        return start;
    }
    if (nums[end] == target) {
        return end;
    }
    return -1;
}
```

Python

```
def binary_search(self, nums, target):
    # corner case 处理
    # 这里等价于 nums is None or len(nums) == 0
    if not nums:
        return -1

    start, end = 0, len(nums) - 1

    # 用 start + 1 < end 而不是 start < end 的目的是为了避免死循环
    # 在 first position of target 的情况下不会出现死循环
    # 但是在 last position of target 的情况下会出现死循环
```

```
    # 样例：nums=[1, 1] target = 1
    # 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
    while start + 1 < end:
        # python 没有 overflow 的问题，直接 // 2 就可以了
        # java和C++ 最好写成 mid = start + (end - start) / 2
        # 防止在 start = 2^31 - 1, end = 2^31 - 1 的情况下出现加法 overflow
        mid = (start + end) // 2
        # > , =, < 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
        if nums[mid] < target:
            start = mid
        elif nums[mid] == target:
            end = mid
        else:
            end = mid

    # 因为上面的循环退出条件是 start + 1 < end
    # 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）
    # 因此需要再单独判断 start 和 end 这两个数谁是我们要的答案
    # 如果是找 first position of target 就先看 start，否则就先看 end
    if nums[start] == target:
        return start
    if nums[end] == target:
        return end
    return -1
```

双指针 Two Pointers

使用条件

7. 滑动窗口 (90%)
8. 时间复杂度要求 O(n) (80%是双指针)
9. 要求原地操作，只可以使用交换，不能使用额外空间 (80%)
10. 有子数组 subarray /子字符串 substring 的关键词 (50%)
11. 有回文 Palindrome 关键词(50%)

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)
- LintCode1712.和相同的二元子数组(相向双指针)
- LintCode627. 最长回文串 (背向双指针)
- LintCode 64: 合并有序数组

代码模版

Java

```java
// 相向双指针(patition in quicksort)
public void patition(int[] A, int start, int end) {
    if (start >= end) {
            return;
        }
        int left = start, right = end;
        // key point 1: pivot is the value, not the index
        int pivot = A[(start + end) / 2];
        // key point 2: every time you compare left & right, it should be
        // left <= right not left < right
        while (left <= right) {
            while (left <= right && A[left] < pivot) {
                left++;
            }
            while (left <= right && A[right] > pivot) {
                right--;
            }
            if (left <= right) {
                int temp = A[left];
                A[left] = A[right];
                A[right] = temp;
                left++;
                right--;
            }
        }
}

// 背向双指针
left = position;
right = position + 1;
while (left >= 0 && right < length) {
```

```
    if (可以停下来了) {
        break;
    }
    left--;
    right++;
}

// 同向双指针
int j = 0;
for (int i = 0; i < n; i++) {
    // 不满足则循环到满足搭配为止
    while (j < n && i 到 j之间不满足条件) {
        j += 1;
    }
    if (i 到 j之间满足条件) {
        处理i，j这次搭配
    }
}

// 合并双指针
ArrayList<Integer> merge(ArrayList<Integer> list1, ArrayList<Integer> list2) {
    // 需要 new 一个新的 list，而不是在 list1 或者 list2 上直接改动
    ArrayList<Integer> newList = new ArrayList<Integer>();

    int i = 0, j = 0;
    while (i < list1.size() && j < list2.size()) {
        if (list1.get(i) < list2.get(j)) {
            newList.add(list1.get(i));
            i++;
        } else {
            newList.add(list2.get(j));
            j++;
        }
    }

    // 合并上下的数到 newList 里
    // 无需用 if (i < list1.size())，直接 while 即可
    while (i < list1.size()) {
        newList.add(list1.get(i));
        i++;
    }
    while (j < list2.size()) {
        newList.add(list2.get(j);
        j++;
    }

```

```
    return newList;
}
```

Python

```python
# 相向双指针(patition in quicksort)
def patition(self, A, start, end):
        if start >= end:
            return
        left, right = start, end
        # key point 1: pivot is the value, not the index
        pivot = A[(start + end) // 2];
        # key point 2: every time you compare left & right, it should be
        # left <= right not left < right
        while left <= right:
            while left <= right and A[left] < pivot:
                left += 1
            while left <= right and A[right] > pivot:
                right -= 1
            if left <= right:
                A[left], A[right] = A[right], A[left]
                left += 1
                right -= 1

# 背向双指针
left = position
right = position + 1
while left >= 0 and right < len(s):
    if left 和 right 可以停下来了:
        break
    left -= 1
    right += 1

# 同向双指针
j = 0
for i in range(n):
    # 不满足则循环到满足搭配为止
    while j < n and i到j之间不满足条件:
        j += 1
    if i到j之间满足条件:
        处理i到j这段区间

# 合并双指针
def merge(list1, list2):
```

```
    new_list = []
    i, j = 0, 0

    # 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
    # 因为 pop(0) 是 O(n) 的时间复杂度
    while i < len(list1) and j < len(list2):
        if list1[i] < list2[j]:
            new_list.append(list1[i])
            i += 1
        else:
            new_list.append(list2[j])
            j += 1

    # 合并剩下的数到 new_list 里
    # 不要用 new_list.extend(list1[i:]) 之类的方法
    # 因为 list1[i:] 会产生额外空间耗费
    while i < len(list1):
        new_list.append(list1[i])
        i += 1
    while j < len(list2):
        new_list.append(list2[j])
        j += 1

    return new_list
```

排序算法 Sorting

使用条件

复杂度

- 时间复杂度：
  - 快速排序(期望复杂度)： O(nlogn)
  - 归并排序(最坏复杂度)： O(nlogn)
- 空间复杂度：
  - 快速排序： O(1)
  - 归并排序： O(n)

领扣例题

- LintCode 463. 整数排序
- LintCode 464. 整数排序 II

代码模板

Java

Java

```java
// quick sort
public class Solution {
    /**
     * @param A an integer array
     * @return void
     */
    public void sortIntegers(int[] A) {
        quickSort(A, 0, A.length - 1);
    }

    private void quickSort(int[] A, int start, int end) {
        if (start >= end) {
            return;
        }

        int left = start, right = end;
        // key point 1: pivot is the value, not the index
        int pivot = A[(start + end) / 2];

        // key point 2: every time you compare left & right, it should be
        // left <= right not left < right
        while (left <= right) {
            while (left <= right && A[left] < pivot) {
                left++;
            }
            while (left <= right && A[right] > pivot) {
                right--;
            }
            if (left <= right) {
                int temp = A[left];
                A[left] = A[right];
                A[right] = temp;

                left++;
                right--;
```

```
            }
        }

        quickSort(A, start, right);
        quickSort(A, left, end);
    }
}
// merge sort
public class Solution {
    public void sortIntegers(int[] A) {
        if (A == null || A.length == 0) {
            return;
        }
        int[] temp = new int[A.length];
        mergeSort(A, 0, A.length - 1, temp);
    }

    private void mergeSort(int[] A, int start, int end, int[] temp) {
        if (start >= end) {
            return;
        }
        // 处理左半区间
        mergeSort(A, start, (start + end) / 2, temp);
        // 处理右半区间
        mergeSort(A, (start + end) / 2 + 1, end, temp);
        // 合并排序数组
        merge(A, start, end, temp);
    }

    private void merge(int[] A, int start, int end, int[] temp) {
        int middle = (start + end) / 2;
        int leftIndex = start;
        int rightIndex = middle + 1;
        int index = start;
        while (leftIndex <= middle && rightIndex <= end) {
            if (A[leftIndex] < A[rightIndex]) {
                temp[index++] = A[leftIndex++];
            } else {
                temp[index++] = A[rightIndex++];
            }
        }
        while (leftIndex <= middle) {
            temp[index++] = A[leftIndex++];
        }
        while (rightIndex <= end) {
```

```
            temp[index++] = A[rightIndex++];
        }
        for (int i = start; i <= end; i++) {
            A[i] = temp[i];
        }
    }
}
```

Python

Python

```python
# quick sort
class Solution:
    # @param {int[]} A an integer array
    # @return nothing
    def sortIntegers(self, A):
        # Write your code here
        self.quickSort(A, 0, len(A) - 1)

    def quickSort(self, A, start, end):
        if start >= end:
            return

        left, right = start, end
        # key point 1: pivot is the value, not the index
        pivot = A[(start + end) // 2];

        # key point 2: every time you compare left & right, it should be
        # left <= right not left < right
        while left <= right:
            while left <= right and A[left] < pivot:
                left += 1

            while left <= right and A[right] > pivot:
                right -= 1

            if left <= right:
                A[left], A[right] = A[right], A[left]

                left += 1
                right -= 1

```

```
        self.quickSort(A, start, right)
        self.quickSort(A, left, end)
# merge sort
class Solution:
    def sortIntegers(self, A):
        if not A:
            return A

        temp = [0] * len(A)
        self.merge_sort(A, 0, len(A) - 1, temp)

    def merge_sort(self, A, start, end, temp):
        if start >= end:
            return

        # 处理左半区间
        self.merge_sort(A, start, (start + end) // 2, temp)
        # 处理右半区间
        self.merge_sort(A, (start + end) // 2 + 1, end, temp)
        # 合并排序数组
        self.merge(A, start, end, temp)

    def merge(self, A, start, end, temp):
        middle = (start + end) // 2
        left_index = start
        right_index = middle + 1
        index = start

        while left_index <= middle and right_index <= end:
            if A[left_index] < A[right_index]:
                temp[index] = A[left_index]
                index += 1
                left_index += 1
            else:
                temp[index] = A[right_index]
                index += 1
                right_index += 1

        while left_index <= middle:
            temp[index] = A[left_index]
            index += 1
            left_index += 1

        while right_index <= end:
```

```
            temp[index] = A[right_index]
            index += 1
            right_index += 1

        for i in range(start, end + 1):
            A[i] = temp[i]
```

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
- LintCode 94. 二叉树中的最大路径和
- LintCode 95.验证二叉查找树

代码模板

Java

```java
public ResultType divideConquer(TreeNode node) {
    // 递归出口
    // 一般处理 node == null 就够了
    // 大部分情况不需要处理 node == leaf
   if (node == null) {
        return ...;
    }
    // 处理左子树
    ResultType leftResult = divideConquer(node.left);
    // 处理右子树
    ResultType rightResult = divideConquer(node.right);
```

```
    //合并答案
    ResultType result = merge leftResult and rightResult
    return result;
}
```

Python

```python
def divide_conquer(root):
    # 递归出口
    # 一般处理 node == null 就够了
    # 大部分情况不需要处理 node == leaf
    if root is None:
        return ...
    # 处理左子树
    left_result = divide_conquer(node.left)
    # 处理右子树
    right_result = divide_conquer(node.right)
    # 合并答案
    result = merge left_result and right_result to get merged result
    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
- LintCode 902. 二叉搜索树的第 k 大元素

代码模板

Java

```java
List<TreeNode> inorderTraversal(TreeNode root) {
    List<TreeNode> inorder = new ArrayList<>();
    if (root == null) {
        return inorder;
    }
    // 创建一个 dummy node，右指针指向 root
    // 放到 stack 里，此时栈顶 dummy 就是 iterator 的当前位置
    TreeNode dummy = new TreeNode(0);
    dummy.right = root;
    Stack<TreeNode> stack = new Stack<>();
    stack.push(dummy);

    // 每次将 iterator 挪到下一个点
    // 就是调整 stack 使得栈顶是下一个点
    while (!stack.isEmpty()) {
        TreeNode node = stack.pop();
        if (node.right != null) {
            node = node.right;
            while (node != null) {
                stack.push(node);
                node = node.left;
            }
        }
        if (!stack.isEmpty()) {
            inorder.add(stack.peek());
        }
    }
    return inorder;
}
```

Python

```python
def inorder_traversal(root):
    if root is None:
        return []

    # 创建一个 dummy node，右指针指向 root
    # 并放到 stack 里，此时 stack 的栈顶 dummy
    # 是 iterator 的当前位置
    dummy = TreeNode(0)
    dummy.right = root
    stack = [dummy]
```

```python

    inorder = []
    # 每次将 iterator 挪到下一个点
    # 也就是调整 stack 使得栈顶到下一个点
    while stack:
        node = stack.pop()
        if node.right:
            node = node.right
            while node:
                stack.append(node)
                node = node.left
        if stack:
            inorder.append(stack[-1])
    return inorder
```

宽度优先搜索 BFS

使用条件

12. 拓扑排序(100%)
13. 出现连通块的关键词(100%)
14. 分层遍历(100%)
15. 简单图最短路径(100%)
16. 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)
- LintCode 431. 找无向图的连通块
- LintCode 127. 拓扑排序

代码模版

Java

```java
ReturnType bfs(Node startNode) {
    // BFS 必须要用队列 queue，别用栈 stack!
    Queue<Node> queue = new ArrayDeque<>();
    // 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
    // 另外一个是记录 startNode 到其他所有节点的最短距离
    // 如果只求连通性的话，可以换成 HashSet 就行
    // node 做 key 的时候比较的是内存地址
    Map<Node, Integer> distance = new HashMap<>();

    // 把起点放进队列和哈希表里，如果有多个起点，都放进去
    queue.offer(startNode);
    distance.put(startNode, 0); // or 1 if necessary

    // 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
    while (!queue.isEmpty()) {
        Node node = queue.poll();
        // 如果有明确的终点可以在这里加终点的判断
        if (node 是终点) {
            break or return something;
        }
        for (Node neighbor : node.getNeighbors()) {
            if (distance.containsKey(neighbor)) {
                continue;
            }
            queue.offer(neighbor);
            distance.put(neighbor, distance.get(node) + 1);
        }
    }
    // 如果需要返回所有点离起点的距离，就 return hashmap
    return distance;
    // 如果需要返回所有连通的节点，就 return HashMap 里的所有点
    return distance.keySet();
   // 如果需要返回离终点的最短距离
    return distance.get(endNode);
}
```

Python

```python
def bfs(start_node):
    # BFS 必须要用队列 queue，别用栈 stack!
    # 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
    # 另外一个是记录 start_node 到其他所有节点的最短距离
    # 如果只求连通性的话，可以换成 set 就行
    # node 做 key 的时候比较的是内存地址
```

```
    queue = collections.deque([start_node])
    distance = {start_node: 0}

    # 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
    while queue:
        node = queue.popleft()
        # 如果有明确的终点可以在这里加终点的判断
        if  node 是终点:
            break or return something
        for neighbor in node.get_neighbors():
            if neighor in distnace:
                continue
            queue.append(neighbor)
            distance[neighbor] = distance[node] + 1

    # 如果需要返回所有点离起点的距离，就 return hashmap
    return distance
    # 如果需要返回所有连通的节点，就 return HashMap 里的所有点
    return distance.keys()
    # 如果需要返回离终点的最短距离
    return distance[end_node]
```

Java 拓扑排序 BFS 模板

```
List<Node> topologicalSort(List<Node> nodes) {
    // 统计所有点的入度信息，放入 hashmap 里
    Map<Node, Integer> indegrees = getIndegrees(nodes);

    // 将所有入度为 0 的点放到队列中
    Queue<Node> queue = new ArrayDeque<>();
    for (Node node : nodes) {
        if (indegrees.get(node) == 0) {
            queue.offer(node);
        }
    }

    List<Node> topoOrder = new ArrayList<>();
    while (!queue.isEmpty()) {
        Node node = queue.poll();
        topoOrder.add(node);
        for (Node neighbor : node.getNeighbors()) {
            // 入度减一
            indegrees.put(neighbor, indegrees.get(neighbor) - 1);
            // 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
```

```
            if (indegrees.get(neighbor) == 0) {
                queue.offer(neighbor);
            }
        }
    }

    // 如果 queue 是空的时候，图中还有点没有被挖出来，说明存在环
    // 有环就没有拓扑序
    if (topoOrder.size() != nodes.size()) {
        return 没有拓扑序;
    }
    return topoOrder;
}

Map<Node, Integer> getIndegrees(List<Node> nodes) {
    Map<Node, Integer> counter = new HashMap<>();
    for (Node node : nodes) {
        counter.put(node, 0);
    }
    for (Node node : nodes) {
        for (Node neighbor : node.getNeighbors()) {
            counter.put(neighbor, counter.get(neighbor) + 1);
        }
    }
    return counter;
}
```

Python

```
def get_indegrees(nodes):
    counter = {node: 0 for node in nodes}
    for node in nodes:
        for neighbor in node.get_neighbors():
            counter[neighbor] += 1
    return counter

def topological_sort(nodes):
    # 统计入度
    indegrees = get_indegrees(nodes)
    # 所有入度为 0 的点都放到队列里
    queue = collections.deque([
        node
        for node in nodes
        if indegrees[node] == 0
```

```
    ])
    # 用 BFS 算法一个个把点从图里挖出来
    topo_order = []
    while queue:
        node = queue.popleft()
        topo_order.append(node)
        for neighbor in node.get_neighbors():
            indegrees[neighbor] -= 1
            if indegrees[neighbor] == 0:
                queue.append(neighbor)
    # 判断是否有循环依赖
    if len(topo_order) != len(nodes):
        return 有循环依赖(环),没有拓扑序
    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

17. 连通块问题（一定要用 BFS，否则 StackOverflow）
18. 拓扑排序（一定要用 BFS，否则 StackOverflow）
19. 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)
- LintCode 652.因式分解(枚举所有情况)

代码模版

Java

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

Python

```python
def dfs(参数列表):
    if 递归出口:
        记录答案
        return
    for 所有的拆解可能性:
        修改所有的参数
        dfs(参数列表)
        还原所有被修改过的参数
    return something 如果需要的话，很多时候不需要 return 值除了分治的写法
```

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 >= 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 s <= v <= 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 - count * A[i - 1]] + count * V[i - 1])`

  `其中 0 <= count <= 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/sum/or(dp[i,j 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) x (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/sum/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 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 的后面可以接上 i`

复杂度

- 时间复杂度:
  - O(状态总数 * 每个状态的处理耗费)
  - 等于O(状态总数 * 决策数)
- 空间复杂度:
  - O(状态总数) (不使用滚动数组优化)
  - O(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)
- LintCode76.最长上升子序列(接龙型)
- LintCode 476.石子归并V(区间型)
- LintCode 192. 通配符匹配 (匹配型)
- LintCode107.单词拆分(划分型)

堆 Heap

使用条件

20. 找最大值或者最小值(60%)
21. 找第 k 大(pop k 次 复杂度O(nlogk))(50%)
22. 要求 logn 时间对数据进行操作(40%)

堆不能解决的问题

23. 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）
24. 找某段区间的最大值最小值（线段树 SegmentTree 可以解决）
25. 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
- LintCode 919. 会议室 II
- LintCode 1512.雇佣K个人的最低费用

代码模板

Java 带删除特定元素功能的堆

```
class ValueIndexPair {
    int val, index;
    public ValueIndexPair(int val, int index) {
        this.val = val;
        this.index = index;
    }
}
class Heap {
    private Queue<ValueIndexPair> minheap;
    private Set<Integer> deleteSet;
    public Heap() {
        minheap = new PriorityQueue<>((p1, p2) -> (p1.val - p2.val));
        deleteSet = new HashSet<>();
    }

    public void push(int index, int val) {
        minheap.add(new ValueIndexPair(val, index));
    }

    private void lazyDeletion() {
        while (minheap.size() != 0 && deleteSet.contains(minheap.peek().index)) {
            ValueIndexPair pair = minheap.poll();
            deleteSet.remove(pair.index);
```

```
        }
    }

    public ValueIndexPair top() {
        lazyDeletion();
        return minheap.peek();
    }

    public void pop() {
        lazyDeletion();
        minheap.poll();
    }

    public void delete(int index) {
        deleteSet.add(index);
    }

    public boolean isEmpty() {
        return minheap.size() == 0;
    }
}
```

Python 带删除特定元素功能的堆

```
from heapq import heappush, heappop

class Heap:

    def __init__(self):
        self.minheap = []
        self.deleted_set = set()

    def push(self, index, val):
        heappush(self.minheap, (val, index))

    def _lazy_deletion(self):
        while self.minheap and self.minheap[0][1] in self.deleted_set:
            heappop(self.minheap)

    def top(self):
        self._lazy_deletion()
        return self.minheap[0]

    def pop(self):
```

```
        self._lazy_deletion()
        heappop(self.minheap)

    def delete(self, index):
        self.deleted_set.add(index)

    def is_empty(self):
        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
- LintCode 1014. 打砖块
- LintCode 1813. 构造二叉树

代码模板

java

```
class UnionFind {
    private Map<Integer, Integer> father;
    private Map<Integer, Integer> sizeOfSet;
    private int numOfSet = 0;
    public UnionFind() {
        // 初始化父指针，集合大小，集合数量
        father = new HashMap<Integer, Integer>();
        sizeOfSet = new HashMap<Integer, Integer>();
        numOfSet = 0;
    }

```

```
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
```

```
        return root;
    }

    public boolean isConnected(int x, int y) {
        // 两个节点连通 等价于 两个节点的根相同
        return find(x) == find(y);
    }

    public int getNumOfSet() {
        // 获得集合数量
        return numOfSet;
    }

    public int getSizeOfSet(int x) {
        // 获得某个点所在集合大小
        return sizeOfSet.get(find(x));
    }
}
```

Python

```
class UnionFind:
    def __init__(self):
        # 初始化父指针，集合大小，集合数量
        self.father = {}
        self.size_of_set = {}
        self.num_of_set = 0

    def add(self, x):
        # 点如果已经出现，操作无效
        if x in self.father:
            return
        # 初始化点的父亲为 空对象None
        # 初始化该点所在集合大小为 1
        # 集合数量增加 1
        self.father[x] = None
        self.num_of_set += 1
        self.size_of_set[x] = 1

    def merge(self, x, y):
        # 找到两个节点的根
        root_x, root_y = self.find(x), self.find(y)
        # 如果根不是同一个则连接
        if root_x != root_y:
```

```
            # 将一个点的根变成新的根
            # 集合数量减少 1
            # 计算新的根所在集合大小
            self.father[root_x] = root_y
            self.num_of_set -= 1
            self.size_of_set[root_y] += self.size_of_set[root_x]

    def find(self, x):
        # 指针 root 指向被查找的点 x
        # 不断找到 root 的父亲
        # 直到 root 指向 x 的根节点
        root = x
        while self.father[root] != None:
            root = self.father[root]
        # 将路径上所有点指向根节点 root
        while x != root:
            # 暂存 x 原本的父亲
            # 将 x 指向根节点
            # x 指针上移至 x 的父节点
            original_father = self.father[x]
            self.father[x] = root
            x = original_father
        return root

    def is_connected(self, x, y):
        # 两个节点连通 等价于 两个节点的根相同
        return self.find(x) == self.find(y)

    def get_num_of_set(self):
        # 获得集合数量
        return self.num_of_set

    def get_size_of_set(self, x):
        # 获得某个点所在集合大小
        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
- LintCode 1624. 最大距离
- LintCode 1090. 映射配对之和

代码模板

java

```java
class TrieNode {
    // 儿子节点
    public Map<Character, TrieNode> children;
    // 根节点到该节点是否是一个单词
    public boolean isWord;
    // 根节点到该节点的单词是什么
    public String word;
    public TrieNode() {
        sons = new HashMap<Character, TrieNode>();
        isWord = false;
        word = null;
    }
}

public class Trie {
    private TrieNode root;
    public Trie() {
        root = new TrieNode();
    }

    public TrieNode getRoot() {
        return root;
    }

    // 插入单词
    public void insert(String word) {
        TrieNode node = root;
        for (int i = 0; i < word.length(); i++) {
            char letter = word.charAt(i);
            if (!node.sons.containsKey(letter)) {
```

```
                node.sons.put(letter, new TrieNode());
            }
            node = node.sons.get(letter);
        }
        node.isWord = true;
        node.word = word;
    }

    // 判断单词 word 是不是在字典树中
    public boolean hasWord(String word) {
        int L = word.length();
        TrieNode node = root;
        for (int i = 0; i < L; i++) {
            char letter = word.charAt(i);
            if (!node.sons.containsKey(letter)) {
                return false;
            }
            node = node.sons.get(letter);
        }

        return node.isWord;
    }

    // 判断前缀 prefix 是不是在字典树中
    public boolean hasPrefix(String prefix) {
        int L = prefix.length();
        TrieNode node = root;
        for (int i = 0; i < L; i++) {
            char letter = prefix.charAt(i);
            if (!node.sons.containsKey(letter)) {
                return false;
            }
            node = node.sons.get(letter);
        }
        return true;
    }
}
```