1、常用链接

力扣列表 牛客网 标准输入输出 牛客网华为真题 代码随想录 github 代码随想录官网 lab 算法小抄 lab 算法小抄精简版 lab 小白刷题 lab 进阶刷题 lab 突击笔试--重要 lab 查漏补缺 lab 力扣高频

acwing 个人分享 acwing 网站分享

dfs 岛屿问题

ASCII 码

2、输入

输入

```
final static int sizeX=3,sizeY=4;
    static int arr[][] = new int[sizeX][sizeY];
    public static void main(String[] args) {
        for (int i = 0; i < sizeX; i++) {
            Arrays.fill(arr[i],-1);
        }
        Scanner sc = new Scanner(new BufferedInputStream(System.in));
        //数字
        System.out.println(sc.nextInt());
        System.out.println(sc.nextDouble());
        System.out.println(sc.nextLong());
        // 字符
        char c = sc.next().charAt(0);
        System.out.println(c);
        // 字符串
        System.out.println(sc.next());  // 遇到空格会换行
        System.out.println(sc.nextLine()); // 遇到空格不会换行
        // 多维数组
        while (sc.hasNext()){
            for (int i = 0; i < sizeX; i++) {
                for (int i1 = 0; i1 < sizeY; i1++) {
                    arr[i][i1]=sc.nextInt();
                }
            }
        }
        sc.close();
    }
```

3、基本数据结构

StringBuilder（字符串拼接）

```
StringBuilder sb = new StringBuilder();
sb.append("%20");
return sb.toString();
// 清零  不能设置为null
stringBuilder.setLength(0);
```

String

```
String.valueOf("4");// 数据类型转化
String[] split = "213123".split(""); //分割
String[] split1 = "a-b-c".split("-"); // 分割
String.join("",split);  // 合并
```

二叉树

```
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
            this.val = val;
            this.left = left;
            this.right = right;
    }
}
```

一维数组

```
String[] ss = new String[10];  //下标赋值、访问
int[] nums = {10,3,4,5,18,17,1,6};
```

多维数组

```
int[][] people1 = {{7,1},{4,4},{7,0},{5,3},{6,1},{5,2}}; //创建
 Arrays.sort(points,(a,b)->{  // 排序  要用三元不用相减  超时再换成减
      if(a[0]==b[0]){
          return a[1]>b[1] ? 1:-1;
      }
```

```
      return a[0]>b[0] ? 1:-1;
});
// int[][] 作为返回值时   创建 赋值 转化
LinkedList res = new LinkedList<>();
res.add(new int[]{start, intervals[intervals.length - 1][1]});
return res.toArray(new int[res.size()][]);
```

列表

```
List list = new ArrayList<>();
LinkedList linkedList = new LinkedList<>();
```

MAP

```
HashMap map = new HashMap<>(8);
Map map = new TreeMap<>();
        map.put("orange", 1);
        map.put("apple", 2);
        map.put("pear", 3);
        for (String key : map.keySet()) {
            System.out.println(key);
        }
        // apple, orange, pear
```

SET

```
HashSet set = new HashSet();
Set set = new TreeSet<>();
        set.add("apple");
        set.add("banana");
        set.add("pear");
        set.add("orange");
        for (String s : set) {
            System.out.println(s);
        }
```

栈

```
Stack stack = new Stack();
```

队列

```
 Queue queue = new LinkedList();
```

双端队列

```
Deque queue = new LinkedList();
                                Queue                    Deque
添加元素到队尾          add(E e) / offer(E e)       addLast(E e) / off
erLast(E e)
取队首元素并删除         E remove() / E poll()       E removeFirst() /
 E pollFirst()
取队首元素但不删除       E element() / E peek()      E getFirst() / E
 peekFirst()
添加元素到队首           无                      addFirst(E e) / offer
First(E e)
取队尾元素并删除           无                      E removeLast() /
 E pollLast()
取队尾元素但不删除         无                    E getLast() / E pee
kLast()
```

优先队列--堆

```
    小顶堆：根节点总是小于左右子节点  默认
    大顶堆：根节点总是大于左右子节点
        PriorityQueue small = new PriorityQueue<>();
        PriorityQueue big = new PriorityQueue<>((a, b) -> {
            return b - a;
        });

        for (int i = 0; i < 5; i++) {
            small.offer(i);
        }
        while (small.peek()!=null){
            Integer poll = small.poll();
            System.out.println(poll);
        }
```

4、kmp next

返回示例

```
//求 next 数组
    public static int[] next(String s){
        // -1 不存在  0 存在一个 1 存在两个
        //字符串装数组
        char[] chars = s.toCharArray();
        //创建 next 数组
        int[] next =new int[chars.length];
        // 0 设置为-1  表示不存在公共前缀
```

```
        next[0] = -1;
        //当前位置的公共前缀数量
        int len = -1;
        for (int i = 1; i < chars.length; i++) {
            //当前字符不等于前缀+1 去匹配前缀字符串的最长前缀
            while (len > -1 && chars[i] != chars[len+1]){
                len = next[len];
            }
            //当前字符等于前缀+1  前缀+1
            if(chars[i] == chars[len+1]){
                len = len + 1;
            }
            next[i] = len;
        }
        return next;
    }

//kmp
    public static int kmp(String haystack, String needle){
        //特殊情况直接返回
        if(needle.length()==0){
            return 0;
        }
        if(haystack.length()==0){
            return -1;
        }
        if(haystack.length() < needle.length()){
            return -1;
        }
        //创建 next 数组并赋值
        int[] next = next(needle);
        char[] haveChars = haystack.toCharArray();
        char[] needChars = needle.toCharArray();
        //创建模式串数组的指针  need = 已匹配字符下标
        int need = -1;
        for (int i = 0; i < haveChars.length; i++) {
            //不匹配 只需要移动 need 不断寻找 不一定 i 所以用 while
            while (need >=0 && haveChars[i] != needChars[need + 1]) {
                need = next[need];
            }
            //匹配 同时移动所以用 if
            if (haveChars[i] == needChars[need + 1]) {
                need ++;
            }
```

```
            //need 指向模式串最后一位 代表匹配上  返回  否则指针会不断往前面跳
            if (need == needChars.length -1) {
                return i - need;
            }
        }
        return -1;
    }
```

5、排序

选择排序

```
public static void xvanze(int[] arr){
        // 23 784
        for (int i = 0; i < arr.length-1; i++) {
            int minIndex = i;
            for (int j = i; j < arr.length-1; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }
            if (minIndex !=i) {
                swap(arr,minIndex,i);
            }
        }
    }
```

冒泡排序

```
public static void maopao(int[] arr){
        // 3 1 2 1 3 5 8 7 8 9
        for (int i = 0; i < arr.length-1; i++) {
            for (int j = 0; j < arr.length-1-i; j++) {
                if (arr[j] > arr[j+1]) {
                    swap(arr,j,j+1);
                }
            }
        }
    }
```

插入排序

```
public static void charu(int[] arr){
```

```
        //-14589 012376
        for (int i = 1; i < arr.length; i++) {
            int j = i;
            while (j > 0 && arr[j] < arr[j-1]){
                swap(arr,j,j-1);
                j--;
            }
        }
    }
```

希尔排序（3 个循环 + 1 个判断）

```
public static void xier(int[] arr){
    // -1,3,4,5,45,34,27,35,36,18,9,0,7,8,100
        for (int i = arr.length / 2; i > 0; i/=2) {
            for (int j = i; j < arr.length; j++) {
                while (j> i && arr[j] < arr[j-i]){
                    swap(arr,j,j-i);
                    j -= i;
                }
            }
        }
    }
```

归并排序

```
public static void guibing(int[] arr){
        int[] temp = new int[arr.length];
        sort(arr,temp,0,arr.length-1);
}
public static void sort(int[] arr,int[] temp,int left,int right){
        if(left<=mid && j<=right){
            if(arr[i]<=mid){
            temp[t++] = arr[i++];
        }

        while (j<=right){
            temp[t++] = arr[j++];
        }

        t = 0;
        while (left<=right){
            arr[left++]=temp[t++];
        }
    }
```

快速排序

```
public static void quickSort(int[] a, int l, int r) {
    if (l < r) {
        int i,j,x;

        i = l;
        j = r;
        x = a[i];
        while (i < j) {
            while(i < j && a[j] > x)
                j--; // 从右向左找第一个小于 x 的数
            if(i < j)
                a[i++] = a[j];
            while(i < j && a[i] < x)
                i++; // 从左向右找第一个大于 x 的数
            if(i < j)
                a[j--] = a[i];
        }
        a[i] = x;
        quickSort(a, l, i-1); /* 递归调用 */
        quickSort(a, i+1, r); /* 递归调用 */
    }
}
```

6、并查集

欢迎使用 ShowDoc！

7、其他

字符串相关问题处理

```
char[] chars = text1.toCharArray();
int[] a = new int[26];
for (int i = 0; i < chars.length; i++) {
     a[chars[i]-'a']++;
}
HashSet set = new HashSet();
for (int i = 0; i < 26; i++) {
    if(a[i]>0){
        set.add((char)(i+'a'));
    }
```

```
}
```

数组互相交换位置

```
public static void swap(int[] arr,int i,int j){
        //需要保证两者不相等
        if (i == j) {
         return;
        }
        // 1 2  -> 2 1
        // 3 2
        arr[i] = arr[i] + arr[j];
        // 3 1
        arr[j] = arr[i] - arr[j];
        // 2 1
        arr[i] = arr[i] - arr[j];
}
```

反转字符串

```
char[] s = chars.toCharArray();
        int left = 0;
        int right = s.length -1;
        while (left < right){
            char temp = s[right];
            s[right] = s[left];
            s[left] = temp;
            left++;
            right--;
        }
```

返回示例

```
int res = 0;
while (n > 0) {
   int temp = n % 10;
   // 对 temp 进行逻辑处理  such as  temp * temp
   res += temp * temp;
   n = n / 10;
}
return res;
```

8、Tire

欢迎使用 ShowDoc！

9、DFS（回溯算法）

模板

```
result = []

used = new boolean[nums.length];
Arrays.fill(used, false);
Arrays.sort(nums);

def backtrack(路径, 选择列表):
    if 路径 满足结束条件:
        result.add(路径)
        return
    if 路径一定不满足条件:
        //剪枝
        return
    for 选择 in 选择列表:
        if valid:
            做选择
        else
            continue; //直接剪枝
        if (used[i]){
            continue;
        }
        used[i] = true;
        backtrack(路径, 选择列表+1)
        撤销选择
        used[i] = false;

def valid(路径, 选择):
    if (条件 1 不满足) {
        return false;
    }
    if (条件 2 不满足) {
        return false;
    }
    return ture
```

返回示例

```
给定两个整数 n 和 k，返回 1 ... n 中所有可能的 k 个数的组合
```

```
class Solution {
    List<List> result = new ArrayList<>();
    LinkedList path = new LinkedList<>();
    public List<List> combine(int n, int k) {
        combineHelper(n, k, 1);
        return result;
    }
    private void combineHelper(int n, int k, int startIndex){
        //终止条件
        if (path.size() == k){
            result.add(new ArrayList<>(path));
            return;
        }
        for (int i = startIndex; i <= n - (k - path.size()) + 1; i++)
{
            path.add(i);
            combineHelper(n, k, i + 1);
            path.removeLast();
        }
    }
}
```

和为 n 的 k 个数的组合

```
class Solution {
    List<List> result = new ArrayList<>();
    LinkedList path = new LinkedList<>();
    public List<List> combinationSum3(int k, int n) {
        backTracking(n, k, 1, 0);
        return result;
    }
    private void backTracking(int targetSum, int k, int startIndex, i
nt sum) {
        // 减枝
        if (sum > targetSum) {
            return;
        }
        if (path.size() == k) {
            if (sum == targetSum) result.add(new ArrayList<>(path));
            return;
        }
        // 减枝 9 - (k - path.size()) + 1
        for (int i = startIndex; i <= 9 - (k - path.size()) + 1; i++)
 {
            path.add(i);
            sum += i;
```

```
            backTracking(targetSum, k, i + 1, sum);
            //回溯
            path.removeLast();
            //回溯
            sum -= i;
        }
    }
}
```

电话号码的字母组合

```
class Solution {
    //设置全局列表存储最后的结果
    List list = new ArrayList<>();
    public List letterCombinations(String digits) {
        if (digits == null || digits.length() == 0) {
            return list;
        }
        //初始对应所有的数字，为了直接对应 2-9，新增了两个无效的字符串""
        String[] numString = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs","tuv", "wxyz"};
        //迭代处理
        backTracking(digits, numString, 0);
        return list;
    }
    //每次迭代获取一个字符串，所以会设计大量的字符串拼接，所以这里选择更为高效的 StringBuild
    StringBuilder temp = new StringBuilder();
    //比如 digits 如果为"23",num 为 0，则 str 表示 2 对应的 abc
    public void backTracking(String digits, String[] numString, int num) {
        //遍历全部一次记录一次得到的字符串
        if (num == digits.length()) {
            list.add(temp.toString());
            return;
        }
        //str 表示当前 num 对应的字符串
        String str = numString[digits.charAt(num) - '0'];
        for (int i = 0; i < str.length(); i++) {
            temp.append(str.charAt(i));
            //c
            backTracking(digits, numString, num + 1);
            //剔除末尾的继续尝试
            temp.deleteCharAt(temp.length() - 1);
        }
```

```
    }
}
```

无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

```
class Solution {
    public List<List> combinationSum(int[] candidates, int target) {
        List<List> res = new ArrayList<>();
        Arrays.sort(candidates); // 先进行排序
        backtracking(res, new ArrayList<>(), candidates, target, 0, 0);
        return res;
    }
    public void backtracking(List<List> res, List path, int[] candidates, int target, int sum, int idx) {
        // 找到了数字和为 target 的组合
        if (sum == target) {
            res.add(new ArrayList<>(path));
            return;
        }
        for (int i = idx; i < candidates.length; i++) {
            // 如果 sum + candidates[i] > target 就终止遍历
            if (sum + candidates[i] > target) break;
            path.add(candidates[i]);
            backtracking(res, path, candidates, target, sum + candidates[i], i);
            path.remove(path.size() - 1); // 回溯，移除路径 path 最后一个元素
        }
    }
}
```

10、BFS

模板代码

```
// 计算从起点 start 到终点 target 的最近距离
int BFS(Node start, Node target) {
    Queue q; // 核心数据结构
    Set visited; // 避免走回头路

    q.offer(start); // 将起点加入队列
    visited.add(start);
    int step = 0; // 记录扩散的步数
```

```
    while (q not empty) {
        int sz = q.size();
        /* 将当前队列中的所有节点向四周扩散 */
        for (int i = 0; i < sz; i++) {
            Node cur = q.poll();
            /* 划重点：这里判断是否到达终点 */
            if (cur is target)
                return step;
            /* 将 cur 的相邻节点加入队列 */
            for (Node x : cur.adj())
                if (x not in visited) {
                    q.offer(x);
                    visited.add(x);
                }
        }
        /* 划重点：更新步数在这里 */
        step++;
    }
}
```

11、动态规划

模板代码

```
明确「状态」 -> 定义 dp 数组/函数的含义 -> 明确「选择」-> 明确 base case。
一般 dp 的第一维是可选择列表的长度，第二维是限制条件，第三维是状态。
class Solution {
    public int maxProfit(int k, int[] prices) {
        if (prices.length == 0) return 0;

        // [天数][交易次数][是否持有股票]
        int len = prices.length;
        int[][][] dp = new int[len][k + 1][2];

        // dp 数组初始化
        // 初始化所有的交易次数是为确保 最后结果是最多 k 次买卖的最大利润
        for (int i = 0; i <= k; i++) {
            dp[0][i][1] = -prices[0];
        }
```

```
        for (int i = 1; i < len; i++) {
            for (int j = 1; j <= k; j++) {
                // dp 方程, 0 表示不持有/卖出, 1 表示持有/买入
                dp[i][j][0] = Math.max(dp[i - 1][j][0], dp[i - 1][j]
[1] + prices[i]);
                dp[i][j][1] = Math.max(dp[i - 1][j][1], dp[i - 1][j -
 1][0] - prices[i]);
            }
        }
        return dp[len - 1][k][0];
    }
}
```