

Algorithm Library

LiuJ

July 9, 2020

Contents

1 技巧	4
1.1 标准输入输出	4
1.2 链表调试	5
1.3 双向链表	6
1.4 自定义类	7
1.5 自定义排序	8
1.6 查找表	8
1.7 类型转换	9
1.8 高精度运算	9
2 基础知识	11
2.1 排序	11
2.1.1 冒泡排序	11
2.1.2 选择排序	12
2.1.3 插入排序	13
2.1.4 希尔排序	15
2.1.5 归并排序	15
2.1.6 快速排序	19
2.1.7 堆排序	21
2.1.8 计数排序	23
2.1.9 桶排序	24
2.1.10 基数排序	25
2.2 二分查找	27
2.2.1 基本	27
2.2.2 第一个等于	27
2.2.3 第一个大于等于	27
2.2.4 第一个大于	28
2.2.5 最后一个等于	28
2.2.6 最后一个小于等于	29
2.2.7 最后一个小于	29
2.3 二叉树遍历	30
2.3.1 前序	30
2.3.2 中序	31
2.3.3 后序	32
2.3.4 层次	35
2.4 搜索	36
2.4.1 回溯	36
2.4.2 深度优先搜索	36
2.4.3 宽度优先搜索	37

3	数据结构	39
3.1	并查集	39
3.2	树状数组	41
3.2.1	单点修改、区间查询	41
3.2.2	区间修改、单点查询	42
3.2.3	区间修改、区间查询	43
3.3	线段树	45
3.4	字典树	49
3.5	单调栈	52
3.6	单调队列	53
4	动态规划	54
4.1	区间	54
5	图论	55
5.1	拓扑排序	55
5.2	连通性	56
5.2.1	强连通分量	56
5.2.2	点双连通分量	57
5.2.3	边双连通分量	59
5.2.4	割点与桥	60
6	字符串	62
6.1	KMP	62
6.2	Manacher	63
7	数学	64
7.1	高精度运算	64
7.1.1	加法	64
7.1.2	乘法	64
7.1.3	阶乘	65
7.2	快速幂	67
7.2.1	乘法快速幂	67
7.2.2	矩阵快速幂	68
7.3	数论	69
7.3.1	最大公约数与最小公倍数	69
7.3.2	素数筛法	69

1 技巧

1.1 标准输入输出

```
1  import java.io.OutputStream;
2  import java.io.IOException;
3  import java.io.InputStream;
4  import java.io.PrintWriter;
5  import java.math.BigDecimal;
6  import java.math.BigInteger;
7  import java.util.StringTokenizer;
8  import java.io.BufferedReader;
9  import java.io.InputStreamReader;
10
11  /**
12   * Built using CHelper plug-in
13   * Actual solution is at the top
14   */
15  public class Main {
16
17      public static void main(String[] args) {
18          InputStream inputStream = System.in;
19          OutputStream outputStream = System.out;
20          InputReader in = new InputReader(inputStream);
21          PrintWriter out = new PrintWriter(outputStream);
22          Task solver = new Task();
23          solver.solve(1, in, out);
24          out.close();
25      }
26
27      static class Task {
28          static final long MOD = (long) (1e9 + 7);
29
30          public void solve(int testNumber, InputReader in, PrintWriter out) {
31              /**
32               * write your main code here
33               */
34          }
35
36          static class InputReader {
37              public BufferedReader reader;
38              public StringTokenizer tokenizer;
39
40              public InputReader(InputStream stream) {
```

```

41         reader = new BufferedReader(new InputStreamReader(stream), 32768);
42         tokenizer = null;
43     }
44
45     public String next() {
46         while (tokenizer == null || !tokenizer.hasMoreTokens()) {
47             try {
48                 tokenizer = new StringTokenizer(reader.readLine());
49             } catch (IOException e) {
50                 throw new RuntimeException(e);
51             }
52         }
53         return tokenizer.nextToken();
54     }
55
56     public int nextInt() {
57         return Integer.parseInt(next());
58     }
59
60     public double nextDouble()
61     {
62         return Double.parseDouble(next());
63     }
64
65     public long nextLong()
66     {
67         return Long.parseLong(next());
68     }
69
70     public BigInteger nextBigInteger() {
71         return new BigInteger(next());
72     }
73
74     public BigDecimal nextBigDecimal() {
75         return new BigDecimal(next());
76     }
77 }
78
79 }
```

1.2 链表调试

```

1 public class ListNode {
2
```

```

3     public int val;
4     public ListNode next = null;
5
6     ListNode(int val) {
7         this.val = val;
8     }
9
10    public ListNode(int[] arr){
11        if(arr == null || arr.length == 0){
12            throw new IllegalArgumentException("arr can not be empty");
13        }
14        this.val = arr[0];
15        ListNode cur = this;
16        for(int i = 1; i < arr.length; i++){
17            cur.next = new ListNode(arr[i]);
18            cur = cur.next;
19        }
20    }
21
22    @Override
23    public String toString(){
24        StringBuilder res = new StringBuilder();
25        ListNode cur = this;
26        while(cur != null){
27            res.append(cur.val + "->");
28            cur = cur.next;
29        }
30        res.append("NULL");
31        return res.toString();
32    }
33
34 }

```

1.3 双向链表

```

1     class DoubleLinkedListNode{
2         int key;
3         int value;
4         DoubleLinkedListNode pre;
5         DoubleLinkedListNode post;
6     }
7
8     private DoubleLinkedListNode head, tail;
9

```

```

10 private void addNode(DoubleLinkedListNode node) {
11     node.pre = head;
12     node.post = head.post;
13     head.post.pre = node;
14     head.post = node;
15 }
16
17 private DoubleLinkedListNode removeNode(DoubleLinkedListNode node) {
18     node.post.pre = node.pre;
19     node.pre.post = node.post;
20     return node;
21 }
22
23 private void moveNode(DoubleLinkedListNode node) {
24     removeNode(node);
25     addNode(node);
26 }
27
28 private DoubleLinkedListNode removeLastNode() {
29     DoubleLinkedListNode node = tail.pre;
30     return removeNode(node);
31 }

```

1.4 自定义类

```

1 class Pair {
2     int pos;
3     int val;
4     public Pair(int pos, int val) {
5         this.pos = pos;
6         this.val = val;
7     }
8 }
9
10 class Tuple {
11     int x;
12     int y;
13     int val;
14     public Tuple(int x, int y, int val) {
15         this.x = x;
16         this.y = y;
17         this.val = val;
18     }
19 }

```

```

20
21 class Pair {
22     int x;
23     int y;
24     public Pair(int x, int y) {
25         this.x = x;
26         this.y = y;
27     }
28
29     public boolean equals(Object o) {
30         if (this == o) {
31             return true;
32         }
33         if (o == null || getClass() != o.getClass()) {
34             return false;
35         }
36         Pair pair = (Pair) o;
37         return x == pair.x && y == pair.y;
38     }
39
40     public int hashCode(){
41         return this.x * 31 + this.y;
42     }
43 }

```

1.5 自定义排序

```

1 Collections.sort(vals, new Comparator<T>() {
2     public int compare(T o1, T o2) {
3         return o1.compareTo(o2);
4     }
5 });
6
7 Arrays.sort(vals, new Comparator<T>() {
8     public int compare(T o1, T o2) {
9         return o1.compareTo(o2);
10    }
11 });

```

1.6 查找表

```

1 // 按键排序
2 Map<Character, Integer> map = new TreeMap<>(new Comparator<Character>() {
3     public int compare(Character o1, Character o2) {

```



```

4         return o2 - o1;
5     }
6 });
7
8 // 按值排序
9 Map<Character, Integer> map = new HashMap<>();
10 List<Map.Entry<Character, Integer>> list = new ArrayList<>(map.entrySet());
11 Collections.sort(list, new Comparator<Map.Entry<Character, Integer>>() {
12     public int compare(Map.Entry<Character, Integer> o1, Map.Entry<Character, Integer> o2) {
13         return o2.getValue() - o1.getValue();
14     }
15 });
16
17 // 遍历
18 for (Map.Entry<K, V> entry: coursesMap.entrySet()) {
19     System.out.println(entry.getKey());
20     System.out.println(entry.getValue());
21 }

```

1.7 类型转换

```

1 List<Integer> list = Arrays.asList(new Integer[]{0, 1});
2 // 鸡肋
3 Integer[] array = list.toArray(new Integer[0]);
4
5
6 // 以 int 举例，其它基本数据类型类似
7 String str = null;
8 int num = 0;
9
10 str = String.valueOf(num);
11 str = Integer.toString(num);
12 str = num + "";
13
14 num = Integer.valueOf(str).intValue();

```

1.8 高精度运算

```

1 // BigDecimal 类似
2 // 构造对象最好使用 String，其它类型可转换为 String 进行构造，保证精度
3 BigInteger a = new BigInteger("123");
4 BigInteger b = BigInteger.valueOf(123);
5 BigInteger c = new BigInteger(String.valueOf(0));
6

```

```
7  // 四则运算
8  c = a.add(b);
9  c = a.subtract(b);
10 c = a.divide(b);
11 c = a.multiply(b);
12
13 // 比较大小
14 a.compareTo(b);
15 a.equals(b);
16
17 // 常用方法
18 c = a.mod(b);
19 c = a.gcd(b);
20 c = a.max(b);
21 c = a.min(b);
22 c = a.abs();
23 c = a.negate();
24 c = a.pow(b);
25
26 // 转换
27 int d = a.intValue();
```

2 基础知识

2.1 排序

2.1.1 冒泡排序

```
1  /***** 数组 *****/
2
3  int[] bubbleSort(int[] arr) {
4      if (arr == null || arr.length == 0) {
5          return arr;
6      }
7      int n = arr.length;
8      for (int j = n - 1; j > 0; --j) {
9          boolean isSort = true;
10         // 冒泡
11         for (int i = 0; i < j; ++i) {
12             if (arr[i] > arr[i + 1]) {
13                 swap(arr, i, i + 1);
14                 isSort = false;
15             }
16         }
17         // 提前结束
18         if (isSort) {
19             break;
20         }
21     }
22     return arr;
23 }
24
25 /***** 链表 *****/
26
27 ListNode bubbleSort(ListNode head) {
28     ListNode cur = head, tail = null;
29     // 双指针
30     while (cur != tail) {
31         while (cur.next != tail) {
32             boolean isSort = true;
33             if (cur.val > cur.next.val) {
34                 int tmp = cur.val;
35                 cur.val = cur.next.val;
36                 cur.next.val = tmp;
37                 isSort = false;
38             }
39             cur = cur.next;
```

```
40     }
41     if (isSort) {
42         break;
43     }
44     // 下次遍历的尾结点是当前结点，每次减少访问最后结点
45     tail = cur;
46     cur = head;
47 }
48 return head;
49 }
```

2.1.2 选择排序

```
1  /***** 数组 *****/
2
3  int[] selectSort(int[] arr) {
4      if (arr == null || arr.length == 0) {
5          return arr;
6      }
7      int n = arr.length;
8      for (int i = 0; i < n; ++i) {
9          int minIdx = i;
10         // 找到区间最小值索引
11         for (int j = i + 1; j < n; ++j) {
12             if (arr[j] < arr[minIdx]) {
13                 minIdx = j;
14             }
15         }
16         if (minIdx != i) {
17             swap(arr, minIdx, i);
18         }
19     }
20     return arr;
21 }
22
23 /***** 链表 *****/
24
25 ListNode selectSort(ListNode head) {
26     ListNode cur = head;
27     // 相当于双指针
28     while (cur != null) {
29         ListNode tmpNode = new ListNode(cur.val);
30         ListNode next = cur.next;
31         while (next != null) {
```

```
32         if (next.val < tmpNode.val) {
33             tmpNode = next;
34         }
35         next = next.next;
36     }
37     // 最小值交换
38     if (cur.val != tmpNode.val) {
39         int tmp = tmpNode.val;
40         tmpNode.val = cur.val;
41         cur.val = tmp;
42     }
43     cur = cur.next;
44 }
45 return head;
46 }
```

2.1.3 插入排序

```
1  /***** 数组 *****/
2
3  int[] insertionSort(int[] arr) {
4      if (arr == null || arr.length == 0) {
5          return arr;
6      }
7      int n = arr.length;
8      for (int i = 1; i < n; ++i) {
9          int val = arr[i], j = i - 1;
10         // 查找插入位置
11         for (; j >= 0 && arr[j] > val; --j) {
12             // 数据移动
13             arr[j + 1] = arr[j];
14         }
15         arr[j + 1] = val;
16     }
17     return arr;
18 }
19
20 /***** 二分插入排序 *****/
21
22 int[] insertionSort(int[] arr) {
23     if (arr == null || arr.length == 0) {
24         return arr;
25     }
26     int n = arr.length;
```

```
27     for (int i = 1; i < n; ++i) {
28         int l = 0, r = i - 1;
29         int val = arr[i];
30         // 找第一个大于 val 的位置
31         while (l <= r) {
32             int mid = l + (r - l) / 2;
33             if (arr[mid] > val) {
34                 r = mid - 1;
35             } else {
36                 l = mid + 1;
37             }
38         }
39         for (int j = i - 1; j >= l; --j) {
40             // 数据移动
41             arr[j + 1] = arr[j];
42         }
43         arr[l] = val;
44     }
45     return arr;
46 }
47
48 /***** 链表 *****/
49
50 ListNode insertionSort(ListNode head) {
51     if (head == null) {
52         return head;
53     }
54     ListNode helper = new ListNode(0);
55     // 当前要被插入的结点
56     ListNode cur = head;
57     // 插入位置在 pre 和 pre.next 之间
58     ListNode pre = helper;
59     // 下一个要插入的结点
60     ListNode next = null;
61     while (cur != null) {
62         next = cur.next;
63         // 找到正确插入的位置
64         while (pre.next != null && cur.val >= pre.next.val) {
65             pre = pre.next;
66         }
67         // 插入操作
68         cur.next = pre.next;
69         pre.next = cur;
70         pre = helper;
```

```
71         cur = next;
72     }
73     return helper.next;
74 }
```

2.1.4 希尔排序

```
1  int[] shellSort(int[] arr) {
2      if (arr == null || arr.length == 0) {
3          return arr;
4      }
5      int n = arr.length;
6      // 增量序列每次减半
7      for (int gap = n; gap > 0; gap /= 2) {
8          // 对每个序列做插入排序
9          for (int end = gap; end < n; ++end) {
10             int val = arr[end], i = end - gap;
11             for (; i >= 0 && arr[i] > val; i -= gap) {
12                 arr[i + gap] = arr[i];
13             }
14             arr[i + gap] = val;
15         }
16     }
17     return arr;
18 }
```

2.1.5 归并排序

```
1  /***** 数组 *****/
2
3  int[] mergeSort(int[] arr) {
4      if (arr == null || arr.length == 0) {
5          return arr;
6      }
7      int n = arr.length;
8      mergeSort(arr, 0, n - 1);
9      return arr;
10 }
11
12 void mergeSort(int[] arr, int l, int r) {
13     if (l >= r) {
14         return;
15     }
16     int mid = l + (r - l) / 2;
```

```

17     mergeSort(arr, l, mid);
18     mergeSort(arr, mid + 1, r);
19     merge(arr, l, mid, r);
20 }
21
22 void merge(int[] arr, int l, int mid, int r) {
23     int[] aux = new int[r - l + 1];
24     int p1 = l, p2 = mid + 1;
25     int k = 0;
26     while (p1 <= mid && p2 <= r) {
27         // 保证稳定性
28         aux[k++] = arr[p1] <= arr[p2] ? arr[p1++] : arr[p2++];
29     }
30     while (p1 <= mid) {
31         aux[k++] = arr[p1++];
32     }
33     while (p2 <= r) {
34         aux[k++] = arr[p2++];
35     }
36     for (int i = 0; i < k; ++i) {
37         arr[l + i] = aux[i];
38     }
39 }
40
41 /***** 数组自底向上 *****/
42
43 int[] mergeSortBU(int[] arr) {
44     if (arr == null || arr.length == 0) {
45         return arr;
46     }
47     int n = arr.length;
48     // 区间个数, 1..2..4..8
49     for (int sz = 1; sz <= n; sz += sz) {
50         // 对 [i, i + sz - 1] 和 [i + sz, i + 2 * sz - 1] 归并
51         for (int i = 0; sz + i < n; i += sz + sz) {
52             // min 防数组越界
53             merge(arr, i, i + sz - 1, Math.min(n - 1, i + 2 * sz - 1));
54         }
55     }
56     return arr;
57 }
58
59 /***** 链表 *****/
60

```



```
61 ListNode mergeSort(ListNode head) {
62     if(head == null || head.next == null){
63         return head;
64     }
65     ListNode pre = null, cur = head, next = head;
66     // 链表分为两半
67     while(next != null && next.next != null){
68         pre = cur;
69         cur = cur.next;
70         next = next.next.next;
71     }
72     pre.next = null;
73     // 对每一半分别排序
74     ListNode l1 = mergeSort(head);
75     ListNode l2 = mergeSort(cur);
76     // 合并
77     return merge(l1, l2);
78 }
79
80 ListNode merge(ListNode l1, ListNode l2){
81     ListNode dummy = new ListNode(0);
82     ListNode cur = dummy;
83     while(l1 != null && l2 != null){
84         if(l1.val < l2.val){
85             cur.next = l1;
86             l1 = l1.next;
87         }
88         else{
89             cur.next = l2;
90             l2 = l2.next;
91         }
92         cur = cur.next;
93     }
94     cur.next = l1 == null ? l2 : l1;
95     return dummy.next;
96 }
97
98 /***** 链表自底向上 *****/
99
100 ListNode mergeSort(ListNode head) {
101     ListNode dummy = new ListNode(0);
102     dummy.next = head;
103     int len = 0;
104     while (head != null) {
```

```

105         head = head.next;
106         ++len;
107     }
108     for (int step = 1; step < len; step <= 1) {
109         ListNode prev = dummy;
110         ListNode cur = dummy.next;
111         while (cur != null) {
112             ListNode left = cur;
113             ListNode right = split(left, step);
114             cur = split(right, step);
115             // 拼接分组排序链表
116             prev = merge(left, right, prev);
117         }
118     }
119     return dummy.next;
120 }

121
122 ListNode split(ListNode head, int step) {
123     if (head == null) {
124         return null;
125     }
126     for (int i = 1; head.next != null && i < step; ++i) {
127         head = head.next;
128     }
129     ListNode right = head.next;
130     head.next = null;
131     return right;
132 }

133
134 ListNode merge(ListNode left, ListNode right, ListNode prev) {
135     ListNode cur = prev;
136     while (left != null && right != null) {
137         if (left.val < right.val) {
138             cur.next = left;
139             left = left.next;
140         }
141         else {
142             cur.next = right;
143             right = right.next;
144         }
145         cur = cur.next;
146     }
147
148     cur.next = left == null ? right : left;

```

```
149     while (cur.next != null) {
150         cur = cur.next;
151     }
152     return cur;
153 }
```

2.1.6 快速排序

```
1  /***** 数组 *****/
2
3  int[] quickSort(int[] arr) {
4      if (arr == null || arr.length == 0) {
5          return arr;
6      }
7      int n = arr.length;
8      quickSort(arr, 0, n - 1);
9      return arr;
10 }
11
12 void quickSort(int[] arr, int l, int r) {
13     if (l >= r) {
14         return;
15     }
16     // 随机选择 pivot, 防止退化为  $O(n^2)$ 
17     swap(arr, l, l + (int)(Math.random() * (r - l + 1)));
18     int[] p = partition(arr, l, r);
19     quickSort(arr, l, p[0]);
20     quickSort(arr, p[1], r);
21 }
22
23 int[] partition(int[] arr, int l, int r) {
24     // 用 arr[l] 作为划分点
25     int val = arr[l];
26     int start = l, end = r + 1;
27     int cur = l + 1;
28     // 交换导致的不稳定性
29     while (cur < end) {
30         if (arr[cur] < val) {
31             swap(arr, ++start, cur++);
32         } else if (arr[cur] > val) {
33             swap(arr, --end, cur);
34         } else {
35             cur++;
36         }
37     }
38     return new int[] { start, end };
39 }
```

```

37     }
38     swap(arr, l, start);
39     // 返回下次开始的位置, 一左一右
40     return new int[]{start - 1, end};
41 }
42
43 void shuffle(int arr[]) {
44     final Random random = new Random();
45     for (int idx = 1; idx < arr.length; ++idx) {
46         final int r = random.nextInt(idx + 1);
47         swap(arr, idx, r);
48     }
49 }
50
51 /***** 链表 *****/
52
53 ListNode quickSort(ListNode head){
54     if(head == null || head.next == null) {
55         return head;
56     }
57
58     // 划分为三个子序列
59     ListNode fakesmall = new ListNode(0), small = fakesmall;
60     ListNode fakelarge = new ListNode(0), large = fakelarge;
61     ListNode fakeequal = new ListNode(0), equal = fakeequal;
62     // pivot
63     ListNode cur = head;
64     while(cur != null){
65         if(cur.val < head.val){
66             small.next = cur;
67             small = small.next;
68         }
69         else if(cur.val == head.val){
70             equal.next = cur;
71             equal = equal.next;
72         }
73         else{
74             large.next = cur;
75             large = large.next;
76         }
77         cur = cur.next;
78     }
79
80     // put an end.

```

```

81     small.next = equal.next = large.next = null;
82     // merge them and return.
83     return merge(merge(quickSort(fakesmall.next), quickSort(fakelarge.next)), fakeequal.next);
84 }
85
86 ListNode merge(ListNode l1, ListNode l2){
87     ListNode dummy = new ListNode(0);
88     ListNode cur = dummy;
89     while(l1 != null && l2 != null){
90         if(l1.val < l2.val){
91             cur.next = l1;
92             l1 = l1.next;
93         }
94         else{
95             cur.next = l2;
96             l2 = l2.next;
97         }
98         cur = cur.next;
99     }
100     cur.next = l1 == null ? l2 : l1;
101     return dummy.next;
102 }

```

2.1.7 堆排序

```

1  /***** 大顶堆 *****/
2
3  int[] heapSort(int[] arr) {
4      // 每个结点的值都大于等于其左右孩子结点的值
5      if (arr == null || arr.length <= 1) {
6          return arr;
7      }
8      int n = arr.length;
9      //上浮方式建堆
10     for (int i = 0; i < arr.length; i++) {
11         siftUp(arr, i);
12     }
13     int size = n - 1;
14     swap(arr, 0, size);
15     while (size > 0) {
16         siftDown(arr, 0, size);
17         swap(arr, 0, --size);
18     }
19     return arr;

```

```
20 }
21
22 // 上浮
23 void siftUp(int[] arr, int i) {
24     // 当前结点为 i, 父亲结点为 (i-1)/2
25     while (arr[i] > arr[(i - 1) / 2]) {
26         swap(arr, i, (i - 1) / 2);
27         i = (i - 1) / 2;
28     }
29 }
30
31 // 下沉
32 private void siftDown(int[] arr, int i, int heapSize) {
33     // 父亲结点为 i, 左孩子结点为 2*i+1, 右孩子结点为 2*i+2
34     int l = 2 * i + 1;
35     // 每次保证堆的性质
36     while (l < heapSize) {
37         int maxIndex = l + 1 < heapSize && arr[l + 1] > arr[l] ? l + 1 : l;
38         maxIndex = arr[i] > arr[maxIndex] ? i : maxIndex;
39         if (maxIndex == i) {
40             break;
41         }
42         swap(arr, i, maxIndex);
43         i = maxIndex;
44         l = 2 * i + 1;
45     }
46 }
47
48 /*****heapfiy 优化 *****/
49
50 int[] heapSort(int[] arr) {
51     if (arr == null || arr.length <= 1) {
52         return arr;
53     }
54     int n = arr.length;
55     int size = n - 1;
56     for (int i = (size - 1) / 2; i >= 0; --i) {
57         // 注意这儿是 n, 因为还没有 swap
58         siftDown(arr, i, n);
59     }
60     swap(arr, 0, size);
61     while (size > 0) {
62         siftDown(arr, 0, size);
63         swap(arr, 0, --size);
64     }
```

```
64     }
65     return arr;
66 }
67
68 void siftDown(int[] arr, int i, int heapSize) {
69     //从 arr[i] 开始往下调整
70     int l = 2 * i + 1;
71     int r = 2 * i + 2;
72     int maxIdx = i;
73     if (l < heapSize && arr[l] > arr[maxIdx]) {
74         maxIdx = l;
75     }
76     if (r < heapSize && arr[r] > arr[maxIdx]) {
77         maxIdx = r;
78     }
79     if (maxIdx != i) {
80         swap(arr, i, maxIdx);
81         siftDown(arr, maxIdx, heapSize);
82     }
83 }
```

2.1.8 计数排序

```
1  /***** 常用 *****/
2
3  int[] countSort(int[] arr) {
4      if (arr == null || arr.length == 0) {
5          return arr;
6      }
7      int n = arr.length;
8      int min = arr[0], max = arr[0];
9      // 最大最小值
10     for (int i = 1; i < n; ++i) {
11         min = Math.min(min, arr[i]);
12         max = Math.max(max, arr[i]);
13     }
14     // 计数数组
15     int[] count = new int[max - min + 1];
16     // 辅助数组
17     int[] aux = new int[n];
18     for (int num : arr) {
19         count[num - min]++;
20     }
21     int index = 0;
```

```
22     // 累加, count[i] 存储小于等于 i 的元素个数
23     for (int i = 1; i < count.length; ++i) {
24         count[i] += count[i - 1];
25     }
26     // 关键步骤, 自己该放置在哪个位置
27     for (int i = arr.length - 1; i >= 0; --i) {
28         aux[--count[arr[i] - min]] = arr[i];
29     }
30     return aux;
31 }
32
33
34 int[] countSort(int[] arr) {
35     if (arr == null || arr.length == 0) {
36         return arr;
37     }
38     int n = arr.length;
39     int min = arr[0], max = arr[0];
40     // 最大最小值
41     for (int i = 1; i < n; ++i) {
42         min = Math.min(min, arr[i]);
43         max = Math.max(max, arr[i]);
44     }
45     // 计数数组
46     int[] count = new int[max - min + 1];
47     for (int num : arr) {
48         count[num - min]++;
49     }
50     int index = 0;
51     // 遍历输出
52     for (int i = 0; i < count.length; ++i) {
53         while (count[i] > 0) {
54             arr[index++] = i + min;
55             count[i]--;
56         }
57     }
58     return arr;
59 }
```

2.1.9 桶排序

```
1 int[] bucketSort(int[] arr, int bucketCount) {
2     if (arr == null || arr.length == 0) {
3         return arr;
```



```

4      }
5      int n = arr.length;
6      int[] res = new int[n];
7      int min = arr[0], max = arr[0];
8      // 最大最小值
9      for (int i = 1; i < n; ++i) {
10         min = Math.min(min, arr[i]);
11         max = Math.max(max, arr[i]);
12     }
13     // 桶容量
14     int gap = (int) Math.ceil((double)(max - min) / bucketCount);
15     List[] buckets = new ArrayList[bucketCount];
16     // 元素放入相应桶中
17     for (int i = 0; i < n; ++i) {
18         int idx = (arr[i] - min) / gap;
19         if (buckets[idx] == null) {
20             buckets[idx] = new ArrayList<>();
21         }
22         buckets[idx].add(arr[i]);
23     }
24     int k = 0;
25     for (int i = 0; i < bucketCount; ++i) {
26         if (buckets[i] == null) {
27             continue;
28         }
29         // 集合排序
30         Collections.sort(buckets[i]);
31         for (int j = 0; j < buckets[i].size(); ++j) {
32             res[k++] = (int) buckets[i].get(j);
33         }
34     }
35     return res;
36 }

```

2.1.10 基数排序

```

1  int[] radixSort(int[] arr, int len) {
2      if (arr == null || arr.length == 0) {
3          return arr;
4      }
5      int n = arr.length;
6      int exp = 10, R = 10;
7      for (int i = 0; i < len; ++i) {
8          List[] digits = new ArrayList[R * 2];

```

```
9      for (int j = 0; j < n; ++j) {
10          // 特定位上的值
11          int bucket = (arr[j] / exp) % 10 + R;
12          if (digits[bucket] == null) {
13              digits[bucket] = new ArrayList();
14          }
15          digits[bucket].add(arr[j]);
16      }
17      int index = 0;
18      // 完成一次排序后拷贝
19      for (int k = 0; k < digits.length; ++k) {
20          if (digits[k] == null) {
21              continue;
22          }
23          for (int l = 0; l < digits[k].size(); ++l) {
24              arr[index++] = (int)digits[k].get(l);
25          }
26      }
27      exp *= 10;
28  }
29  return arr;
30 }
```

2.2 二分查找

2.2.1 基本

```
1 int binarySearch(int[] arr, int key) {
2     int l = 0, r = arr.length - 1;
3     while (l <= r) {
4         int mid = l + (r - l) / 2;
5         if (arr[mid] == key) {
6             return mid;
7         } else if (arr[mid] > key) {
8             r = mid - 1;
9         } else {
10            l = mid + 1;
11        }
12    }
13    return -1;
14 }
```

2.2.2 第一个等于

```
1 int firstEqual(int[] arr, int key) {
2     int l = 0, r = arr.length - 1;
3     while (l <= r) {
4         int mid = l + (r - l) / 2;
5         if (arr[mid] >= key) {
6             r = mid - 1;
7         } else {
8             l = mid + 1;
9         }
10    }
11    // 注意判断条件
12    if (l < arr.length && arr[l] == key) {
13        return l;
14    }
15    return -1;
16 }
```

2.2.3 第一个大于等于

```
1 int firstLargeEqual(int[] arr, int key) {
2     int l = 0, r = arr.length - 1;
3     while (l <= r) {
4         int mid = l + (r - l) / 2;
5         if (arr[mid] >= key) {
```

```
6         r = mid - 1;
7     } else {
8         l = mid + 1;
9     }
10 }
11 return l;
12 }
```

2.2.4 第一个大于

```
1 int firstLarge(int[] arr, int key) {
2     int l = 0, r = arr.length - 1;
3     while (l <= r) {
4         int mid = l + (r - l) / 2;
5         // 注意
6         if (arr[mid] > key) {
7             r = mid - 1;
8         } else {
9             l = mid + 1;
10        }
11    }
12    return l;
13 }
```

2.2.5 最后一个等于

```
1 int lastEqual(int[] arr, int key) {
2     int l = 0, r = arr.length - 1;
3     while (l <= r) {
4         int mid = l + (r - l) / 2;
5         if (arr[mid] <= key) {
6             l = mid + 1;
7         } else {
8             r = mid - 1;
9         }
10    }
11    // 注意判断条件
12    if (r >= 0 && arr[r] == key) {
13        return r;
14    }
15    return -1;
16 }
```

2.2.6 最后一个小于等于

```
1  int lastEqualSmall(int[] arr, int key) {
2      int l = 0, r = arr.length - 1;
3      while (l <= r) {
4          int mid = l + (r - l) / 2;
5          if (arr[mid] <= key) {
6              l = mid + 1;
7          } else {
8              r = mid - 1;
9          }
10     }
11     return r;
12 }
```

2.2.7 最后一个小于

```
1  int lastSmall(int[] arr, int key) {
2      int l = 0, r = arr.length - 1;
3      while (l <= r) {
4          int mid = l + (r - l) / 2;
5          // 注意
6          if (arr[mid] < key) {
7              l = mid + 1;
8          } else {
9              r = mid - 1;
10         }
11     }
12     return r;
13 }
```

2.3 二叉树遍历

2.3.1 前序

```
1  /***** 递归 *****/
2
3  public void preOrder(TreeNode root) {
4      if (root != null) {
5          // write your code here
6          preOrder(root.left);
7          preOrder(root.right);
8      }
9  }
10
11 /***** 非递归 *****/
12
13 public void preOrder(TreeNode root) {
14     if (root == null) {
15         return;
16     }
17     Deque<TreeNode> stack = new ArrayDeque<>();
18     TreeNode p = root;
19     while (p != null || !stack.isEmpty()) {
20         while (p != null) {
21             // write your code here
22             stack.push(p);
23             p = p.left;
24         }
25         p = stack.pop();
26         p = p.right;
27     }
28 }
29
30 /*****Morris*****/
31
32 public void preOrder(TreeNode root) {
33     TreeNode cur = root, pre = null;
34     for (; cur != null;) {
35         if (cur.left != null) {
36             pre = cur.left;
37             // 寻找前驱结点
38             while (pre.right != null && pre.right != cur) {
39                 pre = pre.right;
40             }
```

```
41         if (pre.right == null) {
42             // write your code here
43             pre.right = cur;
44             cur = cur.left;
45         } else {
46             // 删除线索
47             pre.right = null;
48             cur = cur.right;
49         }
50     } else {
51         // write your code here
52         cur = cur.right;
53     }
54 }
55 }
```

2.3.2 中序

```
1  /***** 递归 *****/
2
3  public void inOrder(TreeNode root) {
4      if (root != null) {
5          inOrder(root.left);
6          // write your code here
7          inOrder(root.right);
8      }
9  }
10
11 /***** 非递归 *****/
12
13 public void inOrder(TreeNode root) {
14     if (root == null) {
15         return;
16     }
17     Deque<TreeNode> stack = new ArrayDeque<>();
18     TreeNode p = root;
19     while (p != null || !stack.isEmpty()) {
20         while (p != null) {
21             stack.push(p);
22             p = p.left;
23         }
24         p = stack.pop();
25         // write your code here
26         p = p.right;
```

```

27     }
28 }
29
30 /******Morris******/
31
32 public void inOrder(TreeNode root) {
33     TreeNode cur = root, pre = null;
34     for (; cur != null;) {
35         if (cur.left != null) {
36             pre = cur.left;
37             while (pre.right != null && pre.right != cur) {
38                 pre = pre.right;
39             }
40             if (pre.right == null) {
41                 pre.right = cur;
42                 cur = cur.left;
43             } else {
44                 // write your code here
45                 pre.right = null;
46                 cur = cur.right;
47             }
48         } else {
49             // write your code here
50             cur = cur.right;
51         }
52     }
53 }

```

2.3.3 后序

```

1 /****** 递归 ******/
2
3 public void postOrder(TreeNode root) {
4     if (root != null) {
5         postOrder(root.left);
6         postOrder(root.right);
7         // write your code here
8     }
9 }
10
11 /****** 非递归 ******/
12
13 // 第一种双栈
14 public void postOrder(TreeNode root) {

```



```

15     if (root == null) {
16         return;
17     }
18     Deque<TreeNode> stack1 = new ArrayDeque<>();
19     Deque<TreeNode> stack2 = new ArrayDeque<>();
20     TreeNode p = root;
21     stack1.push(root);
22     while (!stack1.isEmpty()) {
23         TreeNode node = stack1.pop();
24         stack2.push(node);
25         if (node.left != null) {
26             stack1.push(node.left);
27         }
28         if (node.right != null) {
29             stack1.push(node.right);
30         }
31     }
32     while (!stack2.isEmpty()) {
33         // write your code here
34     }
35 }
36
37 // 第二种 pre
38 public void postOrder(TreeNode root) {
39     if (root == null) {
40         return;
41     }
42     Deque<TreeNode> stack = new ArrayDeque<>();
43     stack.push(root);
44     TreeNode pre = null;
45     while (!stack.isEmpty()) {
46         TreeNode node = stack.peek();
47         if ((node.left == null && node.right == null) || (pre != null && (pre == node || pre == node.left))) {
48             // write your code here
49             stack.pop();
50             pre = node;
51         } else {
52             if (node.right != null) {
53                 stack.push(node.right);
54             }
55             if (node.left != null) {
56                 stack.push(node.left);
57             }
58         }
59     }
60 }

```

```

59     }
60 }
61
62 /******Morris******/
63
64 public void postOrder(TreeNode root) {
65     TreeNode dummy = new TreeNode(-1);
66     dummy.left = root;
67     TreeNode cur = dummy, pre = null;
68     for (; cur != null;) {
69         if (cur.left != null) {
70             pre = cur.left;
71             while (pre.right != null && pre.right != cur) {
72                 pre = pre.right;
73             }
74             if (pre.right == null) {
75                 pre.right = cur;
76                 cur = cur.left;
77             } else {
78                 reverse(cur.left, pre);
79                 print(pre, cur.left);
80                 reverse(pre, cur.left);
81                 pre.right = null;
82                 cur = cur.right;
83             }
84         } else {
85             cur = cur.right;
86         }
87     }
88 }
89
90
91 private void print(TreeNode from, TreeNode to) {
92     for (; from = from.right) {
93         // write your code here
94         if (from == to) {
95             break;
96         }
97     }
98 }
99
100 private void reverse(TreeNode from, TreeNode to) {
101     if (from == to) {
102         return;

```

```

103     }
104     TreeNode x = from, y = from.right, z= null;
105     x.right = null;
106     for (;;) {
107         z = y.right;
108         y.right = x;
109         x = y;
110         if (y == to) {
111             break;
112         }
113         y = z;
114     }
115 }

```

2.3.4 层次

```

1  public void levelOrder(TreeNode root) {
2      if (root == null) {
3          return;
4      }
5      Queue<TreeNode> queue = new LinkedList<>();
6      queue.offer(root);
7      while (!queue.isEmpty()) {
8          int sz = queue.size();
9          for (int i = 0; i < sz; i++) {
10             TreeNode node = queue.poll();
11             // write your code here
12             if (node.left != null) {
13                 queue.offer(node.left);
14             }
15             if (node.right != null) {
16                 queue.offer(node.right);
17             }
18         }
19     }
20 }

```

2.4 搜索

2.4.1 回溯

```
1 private List<List<Integer>> res;
2 private boolean[] visited;
3
4 public List<List<Integer>> permuteUnique(int[] nums) {
5     res = new ArrayList<>();
6     visited = new boolean[nums.length];
7     Arrays.sort(nums);
8     backtracking(nums, new ArrayDeque<>());
9     return res;
10 }
11
12 private void backtracking(int[] nums, Deque<Integer> stack){
13     // 满足条件
14     if(stack.size() == nums.length){
15         res.add(new ArrayList<>(stack));
16         return;
17     }
18     for(int i = 0; i < nums.length; i++){
19         // 不满足条件
20         if (visited[i] || (i > 0 && nums[i] == nums[i - 1] && !visited[i - 1])) {
21             continue;
22         }
23         // 做选择
24         visited[i] = true;
25         stack.push(nums[i]);
26
27         backtracking(nums, stack);
28
29         // 撤销选择
30         stack.pop();
31         visited[i] = false;
32     }
33 }
```

2.4.2 深度优先搜索

```
1 private int[][] dir = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};
2 private int m, n;
3
4 public int maxAreaOfIsland(int[][] grid) {
5     if (grid == null || grid.length == 0) {
```

```
6         return 0;
7     }
8     int ans = 0;
9     m = grid.length;
10    n = grid[0].length;
11    for (int i = 0; i < m; i++) {
12        for (int j = 0; j < n; j++) {
13            if (grid[i][j] != 0) {
14                ans = Math.max(ans, dfs(grid, i, j, 1));
15            }
16        }
17    }
18    return ans;
19 }
20
21 private int dfs(int[][] grid, int i, int j, int area) {
22     // 不满足条件
23     if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] == 0) {
24         return 0;
25     }
26     // 原地标记
27     grid[i][j] = 0;
28     // 搜索
29     for (int[] d : dir) {
30         area += dfs(grid, i + d[0], j + d[1], 1);
31     }
32     return area;
33 }
```

2.4.3 宽度优先搜索

```
1 private int[][] dir = {{-1, -1}, {-1, 0}, {-1, 1}, {0, -1}, {0, 1}, {1, -1}, {1, 0}};
2 private int m, n;
3 private boolean[][] visited;
4
5 class Node {
6     int x;
7     int y;
8     public Node(int x, int y) {
9         this.x = x;
10        this.y = y;
11    }
12 }
13
```

```
14 public int bfs(int[] [] grid) {
15     m = grid.length;
16     n = grid[0].length;
17     visited = new boolean[m][n];
18     Queue<Node> queue = new LinkedList<>();
19     queue.offer(new Node(0, 0));
20     visited[0][0] = true;
21     int level = 0;
22     while (!queue.isEmpty()) {
23         level++;
24         int sz = queue.size();
25         for (int i = 0; i < sz; i++) {
26             Node node = queue.poll();
27             int x = node.x;
28             int y = node.y;
29             // 不满足条件
30             if (grid[x][y] == 1) {
31                 continue;
32             }
33             // 到达终点
34             if (x == m - 1 && y == n - 1) {
35                 return level;
36             }
37             // 扩散
38             for (int[] d : dir) {
39                 int xx = x + d[0];
40                 int yy = y + d[1];
41                 if (xx < 0 || xx >= m || yy < 0 || yy >= n) {
42                     continue;
43                 }
44                 if (!visited[xx][yy]) {
45                     // 标记
46                     visited[xx][yy] = true;
47                     queue.offer(new Node(xx, yy));
48                 }
49             }
50         }
51     }
52     return -1;
53 }
```

3 数据结构

3.1 并查集

```
1  class UnionFind {
2
3      private int[] parent;
4      private int[] rank;
5
6      public UnionFind(int n) {
7          parent = new int[n];
8          rank = new int[n];
9
10         for (int i = 0; i < n; i++) {
11             parent[i] = i;
12             rank[i] = 0;
13         }
14     }
15
16     public int size() {
17         return parent.length;
18     }
19
20     public boolean union(int x, int y) {
21         x = find(x);
22         y = find(y);
23         if (x == y) {
24             return false;
25         }
26         if (rank[x] > rank[y]) {
27             parent[y] = x;
28         } else {
29             parent[x] = y;
30             if (rank[x] == rank[y]) {
31                 rank[y]++;
32             }
33         }
34         return true;
35     }
36
37     public boolean same(int x, int y) {
38         return find(x) == find(y);
39     }
40 }
```

```
41     public int find(int x) {  
42         if (parent[x] != x) {  
43             parent[x] = find(parent[x]);  
44         }  
45         return parent[x];  
46     }  
47  
48 }
```


3.2 树状数组

3.2.1 单点修改、区间查询

```
1  class BinaryIndexedTree {
2
3      private int[] sums;
4      private int n;
5      // 数组 array 下标从 1 开始
6      public BinaryIndexedTree(int[] array) {
7          n = array.length - 1;
8          sums = new int[n + 1];
9          for (int i = 1; i <= n; i++) {
10             sums[i] += array[i];
11             int j = i + lowbit(i);
12             if (j <= n) {
13                 sums[j] += sums[i];
14             }
15         }
16     }
17
18     private int lowbit(int x) {
19         return x & (-x);
20     }
21
22     public void update(int x, int add) {
23         while (x <= n) {
24             sums[x] += add;
25             x += lowbit(x);
26         }
27     }
28
29     public int query(int x) {
30         int ret = 0;
31         while (x != 0) {
32             ret += sums[x];
33             x -= lowbit(x);
34         }
35         return ret;
36     }
37
38     public int queryRange(int x, int y) {
39         return query(y) - query(x - 1);
40     }
```

```
41  
42 }
```

3.2.2 区间修改、单点查询

```
1  class BinaryIndexedTree {  
2  
3      private int[] sums;  
4      private int n;  
5      // 传入的是差分数组  
6      public BinaryIndexedTree(int[] array) {  
7          n = array.length - 1;  
8          sums = new int[n + 1];  
9          for (int i = 1; i <= n; i++) {  
10             sums[i] += array[i];  
11             int j = i + lowbit(i);  
12             if (j <= n) {  
13                 sums[j] += sums[i];  
14             }  
15         }  
16     }  
17  
18     private int lowbit(int x) {  
19         return x & (-x);  
20     }  
21  
22     public void update(int x, int add) {  
23         while (x <= n) {  
24             sums[x] += add;  
25             x += lowbit(x);  
26         }  
27     }  
28  
29     public void updateRange(int x, int y, int add) {  
30         update(x, add);  
31         update(y + 1, -add);  
32     }  
33  
34     public int query(int x) {  
35         int ret = 0;  
36         while (x != 0) {  
37             ret += sums[x];  
38             x -= lowbit(x);  
39         }  
40     }  
41 }  
42 }
```

```
40         return ret;
41     }
42
43 }
```

3.2.3 区间修改、区间查询

```
1  class BinaryIndexedTree {
2
3      private long[] sums1;
4      private long[] sums2;
5      private int n;
6
7      public BinaryIndexedTree(int[] array) {
8          n = array.length - 1;
9          sums1 = new long[n + 1];
10         sums2 = new long[n + 1];
11         for (int i = 1; i <= n; i++) {
12             sums1[i] += array[i];
13             sums2[i] += i * array[i];
14             int j = i + lowbit(i);
15             if (j <= n) {
16                 sums1[j] += sums1[i];
17                 sums2[j] += sums2[i];
18             }
19         }
20     }
21
22     private int lowbit(int x) {
23         return x & (-x);
24     }
25
26     public void update(int x, long add) {
27         long add1 = x * add;
28         while (x <= n) {
29             sums1[x] += add;
30             sums2[x] += add1;
31             x += lowbit(x);
32         }
33     }
34
35     public long query(int x) {
36         long ret = 0;
37         long x1 = x;
```

```

38         while (x != 0) {
39             ret += sums1[x] * (x1 + 1);
40             ret -= sums2[x];
41             x -= lowbit(x);
42         }
43         return ret;
44     }
45
46     public void updateRange(int x, int y, long add) {
47         update(x, add);
48         update(y + 1, -add);
49     }
50
51     public long queryRange(int x, int y) {
52         return query(y) - query(x - 1);
53     }
54
55 }
```

3.3 线段树

```
1 public class SegmentTreeRMQPos {
2     public int M, H, N;
3     public int[] st;
4     public int[] pos;
5
6     public SegmentTreeRMQPos(int n) {
7         N = n;
8         M = Integer.highestOneBit(Math.max(N - 1, 1)) << 2;
9         H = M >>> 1;
10        st = new int[M];
11        pos = new int[M];
12        for (int i = 0; i < N; i++) {
13            pos[H + i] = i;
14        }
15        Arrays.fill(st, 0, M, Integer.MAX_VALUE);
16        for (int i = H - 1; i >= 1; i--) {
17            propagate(i);
18        }
19    }
20
21    public SegmentTreeRMQPos(int[] a) {
22        N = a.length;
23        M = Integer.highestOneBit(Math.max(N - 1, 1)) << 2;
24        H = M >>> 1;
25        st = new int[M];
26        pos = new int[M];
27        for (int i = 0; i < N; i++) {
28            st[H + i] = a[i];
29            pos[H + i] = i;
30        }
31        Arrays.fill(st, H + N, M, Integer.MAX_VALUE);
32        for (int i = H - 1; i >= 1; i--) {
33            propagate(i);
34        }
35    }
36
37    public void update(int pos, int x) {
38        st[H + pos] = x;
39        for (int i = (H + pos) >>> 1; i >= 1; i >>= 1) {
40            propagate(i);
41        }
42    }
```

```
43
44     private void propagate(int i) {
45         if (st[2 * i] <= st[2 * i + 1]) {
46             st[i] = st[2 * i];
47             pos[i] = pos[2 * i];
48         } else {
49             st[i] = st[2 * i + 1];
50             pos[i] = pos[2 * i + 1];
51         }
52     }
53
54     public int minpos;
55     public int minval;
56
57     public int minx(int l, int r) {
58         minval = Integer.MAX_VALUE;
59         minpos = -1;
60         if (l >= r) {
61             return minval;
62         }
63         while (l != 0) {
64             int f = l & -l;
65             if (l + f > r) {
66                 break;
67             }
68             int v = st[(H + l) / f];
69             if (v < minval || v == minval && pos[(H + l) / f] < minpos) {
70                 minval = v;
71                 minpos = pos[(H + l) / f];
72             }
73             l += f;
74         }
75
76         while (l < r) {
77             int f = r & -r;
78             int v = st[(H + r) / f - 1];
79             if (v < minval || v == minval && pos[(H + r) / f - 1] < minpos) {
80                 minval = v;
81                 minpos = pos[(H + r) / f - 1];
82             }
83             r -= f;
84         }
85         return minval;
86     }
```

```

87
88     public int min(int l, int r) {
89         minpos = -1;
90         minval = Integer.MAX_VALUE;
91         min(l, r, 0, H, 1);
92         return minval;
93     }
94
95     private void min(int l, int r, int cl, int cr, int cur) {
96         if (l <= cl && cr <= r) {
97             if (st[cur] < minval) {
98                 minval = st[cur];
99                 minpos = pos[cur];
100             }
101         } else {
102             int mid = cl + cr >>> 1;
103             if (cl < r && l < mid) {
104                 min(l, r, cl, mid, 2 * cur);
105             }
106             if (mid < r && l < cr) {
107                 min(l, r, mid, cr, 2 * cur + 1);
108             }
109         }
110     }
111
112     public int sumFenwick(int[] ft, int i) {
113         int sum = 0;
114         for (i++; i > 0; i -= i & -i) {
115             sum += ft[i];
116         }
117         return sum;
118     }
119
120     public void addFenwick(int[] ft, int l, int r, int v) {
121         addFenwick(ft, l, v);
122         addFenwick(ft, r, -v);
123     }
124
125     public void addFenwick(int[] ft, int i, int v) {
126         if (v == 0 || i < 0) {
127             return;
128         }
129         int n = ft.length;
130         for (i++; i < n; i += i & -i) {

```

```
131         ft[i] += v;
132     }
133 }
134 }
```


3.4 字典树

```
1  class Trie {
2
3      static final int MAX = 26;
4
5      private class TrieNode {
6          int path;
7          int end;
8          TrieNode[] next;
9
10         public TrieNode() {
11             path = 0;
12             end = 0;
13             next = new TrieNode[MAX];
14         }
15     }
16
17     private TrieNode root;
18
19     public Trie() {
20         root = new TrieNode();
21     }
22
23     public void insert(String word) {
24         if (word == null) {
25             return;
26         }
27         TrieNode cur = root;
28         int index = 0;
29         for (int i = 0; i < word.length(); i++) {
30             index = word.charAt(i) - 'a';
31             if (cur.next[index] == null) {
32                 cur.next[index] = new TrieNode();
33             }
34             cur = cur.next[index];
35             cur.path++;
36         }
37         cur.end++;
38     }
39
40     public int count(String word) {
41         if (word == null) {
42             return 0;
```

```
43     }
44     TrieNode cur = root;
45     int index = 0;
46     for (int i = 0; i < word.length(); i++) {
47         index = word.charAt(i) - 'a';
48         if (cur.next[index] == null) {
49             return 0;
50         }
51         cur = cur.next[index];
52     }
53     return cur.end;
54 }
55
56 public boolean search(String word) {
57     return count(word) > 0;
58 }
59
60 public int prefixNum(String prefix) {
61     if (prefix == null) {
62         return 0;
63     }
64     TrieNode cur = root;
65     int index = 0;
66     for (int i = 0; i < prefix.length(); i++) {
67         index = prefix.charAt(i) - 'a';
68         if (cur.next[index] == null) {
69             return 0;
70         }
71         cur = cur.next[index];
72     }
73     return cur.path;
74 }
75
76 public boolean startsWith(String prefix) {
77     return prefixNum(prefix) > 0;
78 }
79
80 public void remove(String word) {
81     if (word == null || !search(word)) {
82         return;
83     }
84     TrieNode cur = root;
85     int index = 0;
86     for (int i = 0; i < word.length(); i++) {
```

```
87         index = word.charAt(i) - 'a';
88         if (--cur.next[index].path == 0) {
89             cur.next[index] = null;
90             return;
91         }
92         cur = cur.next[index];
93     }
94     cur.end--;
95 }
96
97 }
```

3.5 单调栈

```
1  /**
2   * 寻找数组中的每一个元素  左边离它最近的比它大的数
3   * 栈底到栈顶：由大到小（也可以自定义从小到大）
4   */
5  void monotoneStack(int[] arr) {
6      int n = arr.length;
7      int[] L = new int[n];
8      Deque<Integer> stack = new ArrayDeque<>();
9      for (int i = 0; i < n; i++) {
10         while (!stack.isEmpty() && arr[i] > arr[stack.peek()]) {
11             int top = stack.pop();
12             if (stack.isEmpty()) {
13                 L[top] = -1;
14             } else {
15                 L[top] = stack.peek();
16             }
17         }
18         stack.push(i);
19     }
20     while (!stack.isEmpty()) {
21         int top = stack.pop();
22         if (stack.isEmpty()) {
23             L[top] = -1;
24         } else {
25             L[top] = stack.peek();
26         }
27     }
28 }
```

3.6 单调队列

```
1  /**
2   * 单调队列：用来求出在数组的某个区间范围内的最值
3   */
4  void monotoneQueue(int[] arr, int k) {
5      int n = arr.length;
6      List<Integer> res = new ArrayList<>();
7      Deque<Integer> deque = new LinkedList<>();
8      for (int i = 0; i < n; i++) {
9          while (!deque.isEmpty() && arr[deque.peekLast()] <= arr[i]) {
10              deque.pollLast();
11          }
12          deque.offerLast(i);
13          // k: 窗口大小
14          if (i - k == deque.peekFirst()) {
15              deque.pollFirst();
16          }
17          // 窗口内最大值
18          if (i >= k - 1) {
19              res.add(arr[deque.peekFirst()]);
20          }
21      }
22  }
```

4 动态规划

4.1 区间

```
1  int[] [] dp = new int[n] [n];
2
3  for (int i = 0; i <= n; i++) {
4      // 初始化 dp 状态
5  }
6  // 枚举区间长度
7  for (int len = 2; len <= n; len++) {
8      // 枚举区间起点
9      for (int i = 0; i + len <= n; i++) {
10         // 区间终点
11         int j = i + len - 1;
12         // 枚举分割点
13         for (int k = i + 1; k < j; k++) {
14             // dp 状态转移
15         }
16     }
17 }
```

5 图论

5.1 拓扑排序

```
1  class TopologySort{
2
3      private ArrayList<Integer>[] G;
4      // 入度数组
5      private int[] deg;
6      // 顶点数、边数
7      private int n, m;
8
9      public void topoSort() {
10         Queue<Integer> queue = new LinkedList<>();
11         for (int i = 1; i < n + 1; i++) {
12             if (deg[i] == 0) {
13                 queue.add(i);
14             }
15         }
16         while (!queue.isEmpty()) {
17             int u = queue.poll();
18             for (int i = 0; i < G[u].size(); i++) {
19                 int v = G[u].get(i);
20                 if (--deg[v] == 0) {
21                     queue.add(v);
22                 }
23             }
24         }
25     }
26
27 }
```

5.2 连通性

5.2.1 强连通分量

```
1  class Tarjan {
2
3      private int V;
4      private List<Integer>[] G;
5      private List<List<Integer>> res;
6      private boolean[] inStack;
7      private Deque<Integer> stack;
8      // dfn[u]: 深度优先搜索遍历时结点 u 被搜索的次序
9      // low[u]: 以 u 为根的子树中的结点的 dfn 的最小值
10     private int[] dfn;
11     private int[] low;
12     private int[] sccBelong;
13     private int[] sccSz;
14     private int dfsCnt;
15     private int sccCnt;
16
17     public Tarjan(List<Integer>[] G, int V) {
18         this.G = G;
19         this.V = V;
20         stack = new ArrayDeque<>();
21         inStack = new boolean[V];
22         dfn = new int[V];
23         low = new int[V];
24         sccBelong = new int[V];
25         sccSz = new int[V];
26         dfsCnt = sccCnt = 0;
27         res = new ArrayList<>();
28     }
29
30     public List<List<Integer>> findScc() {
31         // 从下标索引 1 开始
32         for (int i = 1; i < V; i++) {
33             if (dfn[i] == 0) {
34                 dfs(i);
35             }
36         }
37         return res;
38     }
39
40     private void dfs(int u) {
```



```

41     dfn[u] = low[u] = ++dfsCnt;
42     inStack[u] = true;
43     stack.push(u);
44     for (int v : G[u]) {
45         if (dfn[v] == 0) {
46             dfs(v);
47             if (low[v] < low[u]) {
48                 low[u] = low[v];
49             }
50         } else if (inStack[v] && dfn[v] < dfn[u]) {
51             low[u] = dfn[v];
52         }
53     }
54     if (dfn[u] == low[u]) {
55         List<Integer> tmp = new ArrayList<>();
56         sccCnt++;
57         for (; ; ) {
58             int v = stack.pop();
59             inStack[v] = false;
60             sccBelong[v] = sccCnt;
61             sccSz[sccCnt]++;
62             tmp.add(v);
63             if (v == u) {
64                 break;
65             }
66         }
67         res.add(tmp);
68     }
69 }
70
71 }

```

5.2.2 点双连通分量

```

1  class Tarjan {
2
3      private int V;
4      private List<Integer>[] G;
5      private Deque<Integer> stack;
6      private List<List<Integer>> bcc;
7      // dfn[u]: 深度优先搜索遍历时结点 u 被搜索的次序
8      // low[u]: 以 u 为根的子树中的结点的 dfn 的最小值
9      private int[] dfn;
10     private int[] low;;

```

```

11     private int dfsCnt;
12     private int bccCnt;
13
14     public Tarjan(List<Integer>[] G, int V) {
15         this.G = G;
16         this.V = V;
17         stack = new ArrayDeque<>();
18         bcc = new ArrayList<>();
19         dfn = new int[V];
20         low = new int[V];
21         dfsCnt = bccCnt = 0;
22     }
23
24     public List<List<Integer>> findBccVE() {
25         // 从下标索引 1 开始
26         for (int i = 1; i < V; i++) {
27             if (dfn[i] == 0) {
28                 dfs(i, -1);
29             }
30         }
31     }
32
33     private void dfs(int u, int fa) {
34         dfn[u] = low[u] = ++dfsCnt;
35         stack.push(u);
36         for (int v : G[u]) {
37             if (dfn[v] == 0) {
38                 dfs(v, u);
39                 if (low[v] < low[u]) {
40                     low[u] = low[v];
41                 }
42                 if (fa != -1 && low[v] >= dfn[u]) {
43                     bccCnt++;
44                     List<Integer> tmp = new ArrayList<>();
45                     while (!stack.isEmpty() && stack.peek() != v) {
46                         tmp.add(stack.pop());
47                     }
48                     tmp.add(stack.pop());
49                     tmp.add(u);
50                     bcc.add(tmp);
51                 }
52             } else if (fa != v && dfn[v] < dfn[u]) {
53                 low[u] = Math.min(low[u], dfn[v]);
54             }
55         }
56     }
57 }

```

```
55     }
56 }
57
58 }
```

5.2.3 边双连通分量

```
1  class Tarjan {
2
3      private int V;
4      private List<Integer>[] G;
5      private Deque<Integer> stack;
6      // dfn[u]: 深度优先搜索遍历时结点 u 被搜索的次序
7      // low[u]: 以 u 为根的子树中的结点的 dfn 的最小值
8      private int[] dfn;
9      private int[] low;
10     private int dfsCnt;
11
12     public Tarjan(List<Integer>[] G, int V) {
13         this.G = G;
14         this.V = V;
15         stack = new ArrayDeque<>();
16         dfn = new int[V];
17         low = new int[V];
18         dfsCnt = 0;
19     }
20
21     public List<List<Integer>> findBccE() {
22         // 从下标索引 1 开始
23         for (int i = 1; i < V; i++) {
24             if (dfn[i] == 0) {
25                 dfs(i, -1);
26             }
27         }
28     }
29
30     private void dfs(int u, int fa) {
31         dfn[u] = low[u] = ++dfsCnt;
32         stack.push(u);
33         for (int v : G[u]) {
34             if (dfn[v] == 0) {
35                 dfs(v, u);
36                 if (low[v] < low[u]) {
37                     low[u] = low[v];
```

```

38         }
39         } else if (fa != v && dfn[v] < dfn[u]) {
40             low[u] = Math.min(low[u], dfn[v]);
41         }
42     }
43     if (dfn[u] == low[u]) {
44         while (!stack.isEmpty() && stack.peek() != u) {
45             low[stack.pop()] = low[u];
46         }
47     }
48 }
49
50 }
```

5.2.4 割点与桥

```

1  /**
2   * 当 isPoint[x] 为真时, x 为割点。
3   * 当 isBridge[x] 为真时, (father[x], x) 为桥。
4   */
5  class Tarjan {
6
7      private int V;
8      private List<Integer>[] G;
9      private boolean[] isPoint;
10     private boolean[] isBridge;
11     // dfn[u]: 深度优先搜索遍历结点 u 被搜索的次序
12     // low[u]: 以 u 为根的子树中的结点的 dfn 的最小值
13     private int[] dfn;
14     private int[] low;
15     private int[] father;
16     private int dfsCnt;
17     private int pointCnt;
18     private int bridgeCnt;
19
20     public Tarjan(List<Integer>[] G, int V) {
21         this.G = G;
22         this.V = V;
23         isPoint = new boolean[V];
24         isBridge = new boolean[V];
25         dfn = new int[V];
26         low = new int[V];
27         father = new int[V];
28         dfsCnt = pointCnt = bridgeCnt = 0;
```

```
29     }
30
31     public void findVE() {
32         // 从下标索引 1 开始
33         for (int i = 1; i < V; i++) {
34             if (dfn[i] == 0) {
35                 dfs(i, -1);
36             }
37         }
38     }
39
40     private void dfs(int u, int fa) {
41         father[u] = fa;
42         dfn[u] = low[u] = ++dfsCnt;
43         int child = 0;
44         for (int v : G[u]) {
45             if (dfn[v] == 0) {
46                 child++;
47                 dfs(v, u);
48                 if (low[v] < low[u]) {
49                     low[u] = low[v];
50                 }
51                 if (fa == -1 && child >= 2) {
52                     isPoint[u] = true;
53                     pointCnt++;
54                 }
55                 if (fa != -1 && low[v] >= dfn[u]) {
56                     isPoint[u] = true;
57                     pointCnt++;
58                 }
59                 if (low[v] > dfn[u]) {
60                     isBridge[v] = true;
61                     bridgeCnt++;
62                 }
63             } else if (fa != v && dfn[v] < dfn[u]) {
64                 low[u] = Math.min(low[u], dfn[v]);
65             }
66         }
67     }
68
69 }
```

6 字符串

6.1 KMP

```
1  // next[j] 表示字符串前 j+1 位最长公共前后缀长度
2  private int[] getNext(String p) {
3      int m = p.length();
4      int[] next = new int[m];
5      next[0] = 0;
6      int i, j;
7      for (j = 1, i = 0; j < m; j++) {
8          while (i > 0 && p.charAt(j) != p.charAt(i)) {
9              i = next[i - 1];
10         }
11         if (p.charAt(j) == p.charAt(i)) {
12             i++;
13         }
14         next[j] = i;
15     }
16 }
17
18 // 返回匹配串出现次数
19 public int kmp(String s, String p) {
20     int ans = 0;
21     int n = s.length();
22     int m = p.length();
23     int[] next = getNext(p);
24     for (int i = 0, j = 0; i < n; i++) {
25         while (j > 0 && p.charAt(j) != s.charAt(i)) {
26             j = next[j - 1];
27         }
28         if (p.charAt(j) == s.charAt(i)) {
29             j++;
30         }
31         if (j == m) {
32             ans++;
33             // 此处 j=0 处理不可重叠情形，可重叠直接注释掉即可
34             j = 0;
35         }
36     }
37     return ans;
38 }
```

6.2 Manacher

```
1 public String manacher(String s) {
2     int n = s.length();
3     StringBuilder sb = new StringBuilder();
4     sb.append("$");
5     sb.append("#");
6     for (int i = 0; i < n; i++) {
7         sb.append(s.substring(i, i + 1));
8         sb.append("#");
9     }
10    sb.append("0");
11    int mx = 0, id = 0;
12    int resLen = 0, resCenter = 0;
13    int[] p = new int[sb.length()];
14    for (int i = 1; i < sb.length(); i++) {
15        p[i] = mx > i ? Math.min(p[2 * id - i], mx - i) : 1;
16        while (i + p[i] < sb.length() && i - p[i] >= 0) {
17            if (sb.charAt(i + p[i]) == sb.charAt(i - p[i])) {
18                p[i]++;
19            } else {
20                break;
21            }
22        }
23        if (mx < i + p[i]) {
24            mx = i + p[i];
25            id = i;
26        }
27        if (resLen < p[i]) {
28            resLen = p[i];
29            resCenter = i;
30        }
31    }
32    return s.substring((resCenter - resLen) / 2, (resCenter + resLen) / 2 - 1);
33 }
```

7 数学

7.1 高精度运算

7.1.1 加法

```
1 // 大数加法
2 public String add(String str1, String str2) {
3     char[] s1 = str1.toCharArray();
4     char[] s2 = str2.toCharArray();
5     int n1 = s1.length, n2 = s2.length;
6     int maxL = Math.max(n1, n2);
7     int[] a = new int[maxL + 1];
8     int[] b = new int[maxL + 1];
9     for (int i = 0; i < n1; i++) {
10         a[i] = s1[n1 - i - 1] - '0';
11     }
12     for (int i = 0; i < n2; i++) {
13         b[i] = s2[n2 - i - 1] - '0';
14     }
15     for (int i = 0; i < maxL; i++) {
16         if (a[i] + b[i] >= 10) {
17             int tmp = a[i] + b[i];
18             a[i] = tmp % 10;
19             a[i + 1] += tmp / 10;
20         } else {
21             a[i] += b[i];
22         }
23     }
24     StringBuilder sb = new StringBuilder();
25     if (a[maxL] != 0) {
26         sb.append((char)(a[maxL] + '0'));
27     }
28     for (int i = maxL - 1; i >= 0; i--) {
29         sb.append((char)(a[i] + '0'));
30     }
31     return sb.toString();
32 }
```

7.1.2 乘法

```
1 // 大数乘法
2 public String mul(String str1, String str2){
3     char[] s1 = str1.toCharArray();
4     char[] s2 = str2.toCharArray();
```



```
5     int n1 = s1.length, n2 = s2.length;
6     int[] a = new int[n1];
7     int[] b = new int[n2];
8     int[] c = new int[n1 + n2];
9     for (int i = 0; i < n1; i++) {
10         a[i] = s1[n1 - i - 1] - '0';
11     }
12     for (int i = 0; i < n2; i++) {
13         b[i] = s2[n2 - i - 1] - '0';
14     }
15     for (int i = 0; i < n1; i++) {
16         for (int j = 0; j < n2; j++) {
17             c[i + j] += a[i] * b[j];
18         }
19     }
20     for (int i = 0; i < n1 + n2 - 1; i++) {
21         if (c[i] >= 10) {
22             c[i + 1] += c[i] / 10;
23             c[i] %= 10;
24         }
25     }
26     int i;
27     for (i = n1 + n2 - 1; i >= 0; i--) {
28         if (c[i] != 0) {
29             break;
30         }
31     }
32     StringBuilder sb = new StringBuilder();
33     for (; i >= 0; i--) {
34         sb.append((char)(c[i] + '0'));
35     }
36     return sb.toString();
37 }
```

7.1.3 阶乘

```
1 // 大数阶乘
2 public String bigFactorial(int n) {
3     int[] res = new int[100001];
4     int digit = 1;
5     res[0] = 1;
6     for (int i = 2; i <= n; i++) {
7         int carry = 0;
8         for (int j = 0; j < digit; j++) {
```

```
9         int temp = res[j] * i + carry; //每一位的运算结果
10        res[j] = temp % 10; //将最低位保留在原位置
11        carry = temp / 10; //计算进位，等下这个进位会累加到 j+1
12    }
13    while (carry != 0) {
14        res[digit] = carry % 10;
15        carry /= 10;
16        digit++;
17    }
18 }
19 StringBuilder sb = new StringBuilder();
20 for (int i = digit - 1; i >= 0; i--) {
21     sb.append((char)(res[i] + '0'));
22 }
23 return sb.toString();
24 }
25 // 大数计算阶乘位数
26 private int factorialDigit(int n) {
27     double sum = 0;
28     for (int i = 1; i <= n; i++) {
29         sum += Math.log10(i);
30     }
31     return (int) sum + 1;
32 }
```

7.2 快速幂

7.2.1 乘法快速幂

```
1  // 计算  $(a^n) \% mod$ 
2  public long powMod(long a, long n, long mod) {
3      long res = 0;
4      while (n > 0) {
5          if ((n & 1) != 0) {
6              res = res * a % mod;
7          }
8          a = a * a % mod;
9          n >>= 1;
10     }
11     return res;
12 }
13
14 // 计算  $(a * b) \% mod$ 
15 public long mulMod(long a, long n, long mod) {
16     long res = 0;
17     while (b > 0) {
18         if ((b & 1) != 0) {
19             res = (res + a) % mod;
20         }
21         a = (a << 1) % mod;
22         b >>= 1;
23     }
24     return res;
25 }
26
27 // 计算  $(a^n) \% mod$  配合 mul
28 public long powMod(long a, long n, long mod) {
29     long res = 1;
30     while (n > 0) {
31         if ((n & 1) != 0) {
32             res = mulMod(res, a, mod) % mod;
33         }
34         a = mulMod(a, a, mod) % mod;
35         n >>= 1;
36     }
37     return res;
38 }
```

7.2.2 矩阵快速幂

```
1  class Matrix {
2      int row;
3      int col;
4      int[] [] m;
5
6      public Matrix(int row, int col) {
7          this.row = row;
8          this.col = col;
9          m = new int[row][col];
10     }
11 }
12
13 static final int MOD = 10000;
14
15 public Matrix mul(Matrix a, Matrix b) {
16     Matrix c = new Matrix(a.row, b.col); // 注意这里
17
18     for (int i = 0; i < a.row; i++) {
19         for (int j = 0; j < b.col; j++) {
20             for (int k = 0; k < a.col; k++) {
21                 c.m[i][j] = (c.m[i][j] + a.m[i][k] * b.m[k][j]) % MOD;
22             }
23         }
24     }
25     return c;
26 }
27
28 public Matrix pow(Matrix a, int k) {
29     Matrix res = new Matrix(a.row, a.col); // 方阵
30     for (int i = 0; i < a.row; i++) {
31         res.m[i][i] = 1;
32     }
33     while (k > 0) {
34         if ((k & 1) != 0) {
35             res = mul(res, a);
36         }
37         a = mul(a, a);
38         k >>= 1;
39     }
40     return res;
41 }
```

7.3 数论

7.3.1 最大公约数与最小公倍数

```
1 // 非递归
2 public int gcd(int a, int b) {
3     int r;
4     while (b != 0) {
5         r = a % b;
6         a = b;
7         b = r;
8     }
9     return a;
10 }
11
12 public int gcd(int a, int b) {
13     return b == 0 ? a : gcd(b, a % b);
14 }
15
16 public int lcm(int a, int b) {
17     return a / gcd(a, b) * b;
18 }
19
20 public int ngcd(int arr[], int n) {
21     if (n == 1) {
22         return arr[0];
23     }
24     return gcd(arr[n - 1], ngcd(arr, n - 1));
25 }
26
27 public int nlcm(int arr[], int n) {
28     if (n == 1) {
29         return arr[0];
30     }
31     return lcm(arr[n - 1], nlcm(arr, n - 1));
32 }
```

7.3.2 素数筛法

```
1 // 经典筛法
2 public List<Integer> primary(boolean[] isPrime, int n) {
3     List<Integer> prime = new ArrayList<>();
4     isPrime[0] = isPrime[1] = false;
5     boolean flag;
6     for (int i = 2; i <= n; i++) {
```

```
7         flag = true;
8         // 根号 i 的时间复杂度
9         for (int j = 2; j * j <= i; j++) {
10             if (i % j == 0) {
11                 isPrime[i] = false;
12                 flag = false;
13                 break;
14             }
15         }
16         if (flag) {
17             prime.add(i);
18             isPrime[i] = true;
19         }
20     }
21     return prime;
22 }
23
24 // 经典的埃式筛法
25 public List<Integer> sieve(boolean[] isPrime, int n) {
26     List<Integer> prime = new ArrayList<>();
27     Arrays.fill(isPrime, true);
28     isPrime[0] = isPrime[1] = false;
29     for (int i = 2; i <= n; i++) {
30         if (isPrime[i]) {
31             prime.add(i);
32             for (int j = 2 * i; j <= n; j += i) {
33                 isPrime[j] = false;
34             }
35         }
36     }
37     return prime;
38 }
39
40 // 优化筛法
41 public List<Integer> sieve2(boolean[] isPrime, int n) {
42     ArrayList<Integer> prime = new ArrayList<>();
43     Arrays.fill(isPrime, true);
44     isPrime[0] = isPrime[1] = false;
45     for (int i = 2; i <= n; i++) {
46         if (isPrime[i]) {
47             prime.add(i);
48         }
49         for (int j = 0; j < prime.size() && prime.get(j) <= n / i; j++) {
50             isPrime[prime.get(j) * i] = false;
```

```
51         if (i % prime.get(j) == 0) {
52             break;
53         }
54     }
55 }
56 return prime;
57 }
```