Algorithm Library

LiuJ

May 18, 2020

Contents

1 技巧

1.1 标准输入输出

```java
import java.io.OutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.PrintWriter;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.StringTokenizer;
import java.io.BufferedReader;
import java.io.InputStreamReader;

/**
 * Built using CHelper plug-in
 * Actual solution is at the top
 */
public class Main {

    public static void main(String[] args) {
        InputStream inputStream = System.in;
        OutputStream outputStream = System.out;
        InputReader in = new InputReader(inputStream);
        PrintWriter out = new PrintWriter(outputStream);
        Task solver = new Task();
        solver.solve(1, in, out);
        out.close();
    }

    static class Task {
        static final long MOD = (long) (1e9 + 7);

        public void solve(int testNumber, InputReader in, PrintWriter out) {
            /**
             * write your main code here
             */
        }

    static class InputReader {
        public BufferedReader reader;
        public StringTokenizer tokenizer;

        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 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
```

```java
class Tuple {
    int x;
    int y;
    int val;
    public Tuple(int x, int y, int val) {
        this.x = x;
        this.y = y;
        this.val = val;
    }
}
```

1.4 自定义排序

```java
Collections.sort(vals, new Comparator<T>() {
    public int compare(T o1, T o2) {
        return o1.compareTo(o2);
    }
});

Arrays.sort(vals, new Comparator<T>() {
    public int compare(T o1, T o2) {
        return o1.compareTo(o2);
    }
});
```

2 基础知识

2.1 排序

2.1.1 冒泡排序

```java
/***************** 数组 *****************/

int[] bubbleSort(int[] arr) {
    if (arr == null || arr.length == 0) {
        return arr;
    }
    int n = arr.length;
    for (int j = n - 1; j > 0; --j) {
        boolean isSort = true;
        // 冒泡
        for (int i = 0; i < j; ++i) {
            if (arr[i] > arr[i + 1]) {
                swap(arr, i, i + 1);
                isSort = false;
            }
        }
        // 提前结束
        if (isSort) {
            break;
        }
    }
    return arr;
}

/***************** 链表 *****************/

ListNode bubbleSort(ListNode head) {
    ListNode cur = head, tail = null;
    // 双指针
    while (cur != tail) {
        while (cur.next != tail) {
            boolean isSort = true;
            if (cur.val > cur.next.val) {
                int tmp = cur.val;
                cur.val = cur.next.val;
                cur.next.val = tmp;
                isSort = false;
            }
            cur = cur.next;
```

```
        }
        if (isSort) {
            break;
        }
        // 下次遍历的尾结点是当前结点，每次减少访问最后结点
        tail = cur;
        cur = head;
    }
    return head;
}
```

2.1.2 选择排序

```
/***************** 数组 *****************/

int[] selectSort(int[] arr) {
    if (arr == null || arr.length == 0) {
        return arr;
    }
    int n = arr.length;
    for (int i = 0; i < n; ++i) {
        int minIdx = i;
        // 找到区间最小值索引
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        if(minIdx != i) {
            swap(arr, minIdx, i);
        }
    }
    return arr;
}

/***************** 链表 *****************/

ListNode selectSort(ListNode head) {
    ListNode cur = head;
    // 相当于双指针
    while (cur != null) {
        ListNode tmpNode = new ListNode(cur.val);
        ListNode next = cur.next;
        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[] qucikSort(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(n²)
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      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)),fakeequ
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);
```

```
    }
    return arr;
}

void siftDown(int[] arr, int i, int heapSize) {
    //从 arr[i] 开始往下调整
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    int maxIdx = i;
    if (l < heapSize && arr[l] > arr[maxIdx]) {
        maxIdx = l;
    }
    if (r < heapSize && arr[r] > arr[maxIdx]) {
        maxIdx = r;
    }
    if (maxIdx != i) {
        swap(arr, i, maxIdx);
        siftDown(arr, maxIdx, heapSize);
    }
}
```

2.1.8 计数排序

```
/***************** 常用 *****************/

int[] countSort(int[] arr) {
    if (arr == null || arr.length == 0) {
        return arr;
    }
    int n = arr.length;
    int min = arr[0], max = arr[0];
    // 最大最小值
        for (int i = 1; i < n; ++i) {
        min = Math.min(min, arr[i]);
        max = Math.max(max, arr[i]);
    }
    // 计数数组
    int[] count = new int[max - min + 1];
    // 辅助数组
    int[] aux = new int[n];
    for (int num : arr) {
        count[num - min]++;
    }
    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;
```

```
    }
    int n = arr.length;
    int[] res = new int[n];
    int min = arr[0], max = arr[0];
    // 最大最小值
    for (int i = 1; i < n; ++i) {
        min = Math.min(min, arr[i]);
        max = Math.max(max, arr[i]);
    }
    // 桶容量
    int gap = (int)Math.ceil((double)(max - min) / bucketCount);
    List[] buckets = new ArrayList[bucketCount];
    // 元素放入相应桶中
    for (int i = 0; i < n; ++i) {
        int idx = (arr[i] - min) / gap;
        if (buckets[idx] == null) {
            buckets[idx] = new ArrayList<>();
        }
        buckets[idx].add(arr[i]);
    }
    int k = 0;
    for (int i = 0; i < bucketCount; ++i) {
        if (buckets[i] == null) {
                continue;
            }
        // 集合排序
        Collections.sort(buckets[i]);
        for (int j = 0; j < buckets[i].size(); ++j) {
            res[k++] = (int)buckets[i].get(j);
        }
    }
    return res;
}
```

2.1.10 基数排序

```
int[] redixSort(int[] arr, int len) {
    if (arr == null || arr.length == 0) {
        return arr;
    }
    int n = arr.length;
    int exp = 10, R = 10;
    for (int i = 0; i < len; ++i) {
        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

```java
int binarySearch(int[] arr, int key) {
    int l = 0, r = arr.length - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == key) {
            return mid;
        } else if (arr[mid] > key) {
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return -1;
}
```

2.2.2 第一个等于

```java
int firstEqual(int[] arr, int key) {
    int l = 0, r = arr.length - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] >= key) {
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    // 注意判断条件
    if (l < arr.length && arr[l] == key) {
        return l;
    }
    return -1;
}
```

2.2.3 第一个大于等于

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

```
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return l;
}
```

2.2.4 第一个大于

```
int firstLarge(int[] arr, int key) {
    int l = 0, r = arr.length - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        // 注意
        if (arr[mid] > key) {
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return l;
}
```

2.2.5 最后一个等于

```
int lastEqual(int[] arr, int key) {
    int l = 0, r = arr.length - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] <= key) {
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    // 注意判断条件
    if (r >= 0 && arr[r] == key) {
        return r;
    }
    return -1;
}
```

2.2.6 最后一个小于等于

```
int lastEqualSmall(int[] arr, int key) {
    int l = 0, r = arr.length - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] <= key) {
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    return r;
}
```

2.2.7 最后一个小于

```
int lastSmall(int[] arr, int key) {
    int l = 0, r = arr.length - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        // 注意
        if (arr[mid] < key) {
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    return r;
}
```

2.3 二叉树遍历

2.3.1 前序

```java
/***************** 递归 *****************/

public void preOrder(TreeNode root) {
    if (root != null) {
        // write your code here
        preOrder(root.left);
        preOrder(root.right);
    }
}

/*************** 非递归 ***************/

public void preOrder(TreeNode root) {
    if (root == null) {
        return;
    }
    Deque<TreeNode> stack = new ArrayDeque<>();
    TreeNode p = root;
    while (p != null || !stack.isEmpty()) {
        while (p != null) {
            // write your code here
            stack.push(p);
            p = p.left;
        }
        p = stack.pop();
        p = p.right;
    }
}

/***************Morris***************/

public void preOrder(TreeNode root) {
    TreeNode cur = root, pre = null;
    for (; cur != null;) {
        if (cur.left != null) {
            pre = cur.left;
                // 寻找前驱结点
            while (pre.right != null && pre.right != cur) {
                pre = pre.right;
            }
```

```
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;
```

```
    }
}

/****************Morris****************/

public void inOrder(TreeNode root) {
    TreeNode cur = root, pre = null;
    for (; cur != null;) {
        if (cur.left != null) {
            pre = cur.left;
            while (pre.right != null && pre.right != cur) {
                pre = pre.right;
            }
            if (pre.right == null) {
                pre.right = cur;
                cur = cur.left;
            } else {
                // write your code here
                pre.right = null;
                cur = cur.right;
            }
        } else {
            // write your code here
            cur = cur.right;
        }
    }
}
```

2.3.3 后序

```
/***************** 递归 *****************/

public void postOrder(TreeNode root) {
    if (root != null) {
        postOrder(root.left);
        postOrder(root.right);
        // write your code here
    }
}

/***************** 非递归 *****************/

// 第一种双栈
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 == noc
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  }
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
85          } else {
86              cur = cur.right;
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

```java
private List<List<Integer>> res;
private boolean[] visited;

public List<List<Integer>> permuteUnique(int[] nums) {
    res = new ArrayList<>();
    visited = new boolean[nums.length];
    Arrays.sort(nums);
    backtracking(nums, new ArrayDeque<>());
    return res;
}

private void backtracking(int[] nums, Deque<Integer> stack){
    // 满足条件
    if(stack.size() == nums.length){
        res.add(new ArrayList<>(stack));
        return;
    }
    for(int i = 0; i < nums.length; i++){
        // 不满足条件
        if (visited[i] || (i > 0 && nums[i] == nums[i - 1] && !visited[i - 1])) {
            continue;
        }
        // 做选择
        visited[i] = true;
        stack.push(nums[i]);

        backtracking(nums, stack);

        // 撤销选择
        stack.pop();
        visited[i] = false;
    }
}
```

2.4.2 深度优先搜索

```java
private int[][] dir = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};
private int m, n;

public int maxAreaOfIsland(int[][] grid) {
    if (grid == null || grid.length == 0) {
```

```
        return 0;
    }
    int ans = 0;
    m = grid.length;
    n = grid[0].length;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] != 0) {
                ans = Math.max(ans, dfs(grid, i, j, 1));
            }
        }
    }
    return ans;
}

private int dfs(int[][] grid, int i, int j, int area) {
    // 不满足条件
    if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] == 0) {
        return 0;
    }
    // 原地标记
    grid[i][j] = 0;
    // 搜索
    for (int[] d : dir) {
        area += dfs(grid, i + d[0], j + d[1], 1);
    }
    return area;
}
```

2.4.3 宽度优先搜索

```
private int[][] dir = {{-1, -1}, {-1, 0}, {-1, 1}, {0, -1}, {0, 1}, {1, -1}, {1, 0}
private int m, n;
private boolean[][] visited;

class Node {
    int x;
    int y;
    public Node(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

```

```java
public int bfs(int[][] grid) {
    m = grid.length;
    n = grid[0].length;
    visited = new boolean[m][n];
    Queue<Node> queue = new LinkedList<>();
    queue.offer(new Node(0, 0));
    visited[0][0] = true;
    int level = 0;
    while (!queue.isEmpty()) {
        level++;
        int sz = queue.size();
        for (int i = 0; i < sz; i++) {
            Node node = queue.poll();
            int x = node.x;
            int y = node.y;
            // 不满足条件
            if (grid[x][y] == 1) {
                continue;
            }
            // 到达终点
            if (x == m - 1 && y == n - 1) {
                return level;
            }
            // 扩散
            for (int[] d : dir) {
                int xx = x + d[0];
                int yy = y + d[1];
                if (xx < 0 || xx >= m || yy < 0 || yy >= n) {
                    continue;
                }
                if (!visited[xx][yy]) {
                    // 标记
                    visited[xx][yy] = true;
                    queue.offer(new Node(xx, yy));
                }
            }
        }
    }
    return -1;
}
```

3 数据结构

3.1 并查集

```java
class UnionFind {

    private int[] parent;
    private int[] rank;

    public UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];

        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    public int size() {
        return parent.length;
    }

    public boolean union(int x, int y) {
        x = find(x);
        y = find(y);
        if (x == y) {
            return false;
        }
        if (rank[x] > rank[y]) {
            parent[y] = x;
        } else {
            parent[x] = y;
            if (rank[x] == rank[y]) {
                rank[y]++;
            }
        }
        return true;
    }

    public boolean same(int x, int y) {
        return find(x) == find(y);
    }

```

```
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

```java
class BinaryIndexedTree {

    private int[] sums;
    private int n;
    // 数组 array 下标从 1 开始
    public BinaryIndexedTree(int[] array) {
        n = array.length - 1;
        sums = new int[n + 1];
        for (int i = 1; i <= n; i++) {
            sums[i] += array[i];
            int j = i + lowbit(i);
            if (j <= n) {
                sums[j] += sums[i];
            }
        }
    }

    private int lowbit(int x) {
        return x & (-x);
    }

    public void update(int x, int add) {
        while (x <= n) {
            sums[x] += add;
            x += lowbit(x);
        }
    }

    public int query(int x) {
        int ret = 0;
        while (x != 0) {
            ret += sums[x];
            x -= lowbit(x);
        }
        return ret;
    }

    public int queryRange(int x, int y) {
        return query(y) - query(x - 1);
    }
```

```
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          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

3.4 字典树

```java
class Trie {

    static final int MAx = 26;

    private class TrieNode {
        int path;
        int end;
        TrieNode[] next;

        public TrieNode() {
            path = 0;
            end = 0;
            next = new TrieNode[MAX];
        }
    }

    private TrieNode root;

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

    public void insert(String word) {
        if (word == null) {
            return;
        }
        TrieNode cur = root;
        int index = 0;
        for (int i = 0; i < word.length(); i++) {
            index = word.charAt(i) - 'a';
            if (cur.next[index] == null) {
                cur.next[index] = new TrieNode();
            }
            cur = cur.next[index];
            cur.path++;
        }
        cur.end++;
    }

    public int count(String word) {
        if (word == null) {
            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 = word.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

```java
/**
 * 寻找数组中的每一个元素  左边离它最近的比它大的数
 * 栈底到栈顶：由大到小 (也可以自定义从小到大)
 */
void monotoneStack(int[] arr) {
    int n = arr.length;
    int[] L = new int[n];
    Deque<Integer> stack = new ArrayDeque<>();
    for (int i = 0; i < n; i++) {
        while (!stack.isEmpty() && arr[i] > arr[stack.peek()]) {
            int top = stack.pop();
            if (stack.isEmpty()) {
                L[top] = -1;
            } else {
                L[top] = stack.peek();
            }
        }
        stack.push(i);
    }
    while (!stack.isEmpty()) {
        int top = stack.pop();
        if (stack.isEmpty()) {
            L[top] = -1;
        } else {
            L[top] = stack.peek();
        }
    }
}
```

3.6 单调队列

```java
/**
 * 单调队列：用来求出在数组的某个区间范围内的最值
 */
void monotoneQueue(int[] arr, int k) {
    int n = arr.length;
    List<Integer> res = new ArrayLise<>();
    Deque<Integer> deque = new LinkedList<>();
    for (int i = 0; i < n; i++) {
        while (!deque.isEmpty() && arr[deque.peekLast()] <= nums[i]) {
            deque.pollLast();
        }
        deque.offerLast(i);
        // k：窗口大小
        if (i - k == deque.peekFirst()) {
            deque.pollFirst();
        }
        // 窗口内最大值
        if (i >= k - 1) {
            res.add(arr[deque.peekFirst()]);
        }
    }
}
```

4 动态规划

5 图论

5.1 拓扑排序

```java
class TopologySort{

    private ArrayList<Integer>[] G;
    // 入度数组
    private int[] deg;
    // 顶点数、边数
    private int n, m;

    public void topoSort() {
        Queue<Integer> queue = new LinkedList<>();
        for (int i = 1; i < n + 1; i++) {
            if (deg[i] == 0) {
                queue.add(i);
            }
        }
        while (!queue.isEmpty()) {
            int u = queue.poll();
            for (int i = 0; i < G[u].size(); i++) {
                int v = G[u].get(i);
                if (--deg[v] == 0) {
                    queue.add(v);
                }
            }
        }
    }

}
```

5.2 连通性

5.2.1 强连通分量

```java
class Tarjan {

    private int V;
    private List<Integer>[] G;
    private List<List<Integer>> res;
    private boolean[] inStack;
    private Deque<Integer> stack;
    // dfn[u]：深度优先搜索遍历时结点 u 被搜索的次序
    // low[u]：以 u 为根的子树中的结点的 dfn 的最小值
    private int[] dfn;
    private int[] low;
    private int[] sccBelong;
    private int[] sccSz;
    private int dfsCnt;
    private int sccCnt;

    public Tarjan(List<Integer>[] G, int V) {
        this.G = G;
        this.V = V;
        stack = new ArrayDeque<>();
        inStack = new boolean[V];
        dfn = new int[V];
        low = new int[V];
        sccBelong = new int[V];
        sccSz = new int[V];
        dfsCnt = sccCnt = 0;
        res = new ArrayList<>();
    }

    public List<List<Integer>> findScc() {
        // 从下标索引 1 开始
        for (int i = 1; i < V; i++) {
            if (dfn[i] == 0) {
                dfs(i);
            }
        }
        return res;
    }

    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              }
```

```
        }
    }

}
```

5.2.3 边双连通分量

```
class Tarjan {

    private int V;
    private List<Integer>[] G;
    private Deque<Integer> stack;
    // dfn[u]：深度优先搜索遍历时结点 u 被搜索的次序
    // low[u]：以 u 为根的子树中的结点的 dfn 的最小值
    private int[] dfn;
    private int[] low;
    private int dfsCnt;

    public Tarjan(List<Integer>[] G, int V) {
        this.G = G;
        this.V = V;
        stack = new ArrayDeque<>();
        dfn = new int[V];
        low = new int[V];
        dfsCnt = 0;
    }

    public List<List<Integer>> findBccE() {
        // 从下标索引 1 开始
        for (int i = 1; i < V; i++) {
            if (dfn[i] == 0) {
                dfs(i, -1);
            }
        }
    }

    private void dfs(int u, int fa) {
        dfn[u] = low[u] = ++dfsCnt;
        stack.push(u);
        for (int v : G[u]) {
            if (dfn[v] == 0) {
                dfs(v, u);
                if (low[v] < low[u]) {
                    low[u] = low[v];
```

```
                }
            } else if (fa != v && dfn[v] < dfn[u]) {
                low[u] = Math.min(low[u], dfn[v]);
            }
        }
        if (dfn[u] == low[u]) {
            while (!stack.isEmpty() && stack.peek() != u) {
                low[stack.pop()] = low[u];
            }
        }
    }

}
```

5.2.4 割点与桥

```
/**
 * 当 isPoint[x] 为真时，x 为割点。
 * 当 isBridge[x] 为真时，(father[x], x) 为桥。
 */
class Tarjan {

    private int V;
    private List<Integer>[] G;
    private boolean[] isPoint;
    private boolean[] isBridge;
    // dfn[u]：深度优先搜索遍历时结点 u 被搜索的次序
    // low[u]：以 u 为根的子树中的结点的 dfn 的最小值
    private int[] dfn;
    private int[] low;
    private int[] father;
    private int dfsCnt;
    private int pointCnt;
    private int bridgeCnt;

    public Tarjan(List<Integer>[] G, int V) {
        this.G = G;
        this.V = V;
        isPoint = new boolean[V];
        isBridge = new boolean[V];
        dfn = new int[V];
        low = new int[V];
        father = new int[V];
        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

```java
// next[j] 表示字符串前 j+1 位最长公共前后缀长度
private int[] getNext(String p) {
    int m = p.length();
    int[] next = new int[m];
    next[0] = 0;
    int i, j;
    for (j = 1, i = 0; j < m; j++) {
        while (i > 0 && p.charAt(j) != p.charAt(i)) {
            i = next[i - 1];
        }
        if (p.charAt(j) == p.charAt(i)) {
            i++;
        }
        next[j] = i;
    }
}

// 返回匹配串出现次数
public int kmp(String s, String p) {
    int ans = 0;
    int n = s.length();
    int m = p.length();
    int[] next = getNext(p);
    for (int i = 0, j = 0; i < n; i++) {
        while (j > 0 && p.charAt(j) != s.charAt(i)) {
            j = next[j - 1];
        }
        if (p.charAt(j) == s.charAt(i)) {
            j++;
        }
        if (j == m) {
            ans++;
            // 此处 j=0 处理不可重叠情形，可重叠直接注释掉即可
            j = 0;
        }
    }
    return ans;
}
```

6.2 Manacher

```java
public String manacher(String s) {
    int n = s.length();
    StringBuilder sb = new StringBuilder();
    sb.append("$");
    sb.append("#");
    for (int i = 0; i < n; i++) {
        sb.append(s.substring(i, i + 1));
        sb.append("#");
    }
    sb.append("0");
    int mx = 0, id = 0;
    int resLen = 0, resCenter = 0;
    int[] p = new int[sb.length()];
    for (int i = 1; i < sb.length(); i++) {
        p[i] = mx > i ? Math.min(p[2 * id - i], mx - i) : 1;
        while (sb.charAt(i + p[i]) == sb.charAt(i - p[i])) {
            p[i]++;
        }
        if (mx < i + p[i]) {
            mx = i + p[i];
            id = i;
        }
        if (resLen < p[i]) {
            resLen = p[i];
            resCenter = i;
        }
    }
    return s.substring((resCenter - resLen) / 2, (resCenter + resLen) / 2 - 1);
}
```

7　数学