1.大数阶乘

```
#include<stdio.h>
#define MAX 100000                  // 若为10000，则会因缓存区小而溢出

int main()
{
    int numN = 0;                   // 正整数N
    while (scanf("%d",&numN) != EOF)
    {
        if (numN < 1 || numN >= 10000)
        {
            break;
        }

        int array[MAX] = {0};
        array[1] = 1;               // 从array[1]开始
        int point = 1;              // point表示位数，刚开始只有一位array[1] 且 array[1] = 1，不能为0，0乘任何数为0
        int carry = 0;              // carry表示进位数，刚开始进位为0
        int j = 0;

        for (int i = 2; i <= numN; i++)     // N的阶乘
        {
            for (j = 1; j <= point; j++)    // 循环array[]，让每一位都与i乘
            {
                int temp = array[j] * i + carry;    // temp变量表示不考虑进位的值
                carry = temp / 10;                  // 计算进位大小
                array[j] = temp % 10;               // 计算本位值
            }
            // 处理最后一位的进位情况
            // 由于计算数组的最后一位也得考虑进位情况，所以用循环讨论
            // 因为可能最后一位可以进多位，比如 12 * 本位数8，可以进两位
            while(carry)                            // 当进位数存在时，循环的作用就是将一个数分割，分割的每一位放入数组中
            {
                array[j] = carry % 10;
                carry = carry / 10;
                j++;                                // 表示下一位
            }
            printf("%d\n",point);
            point = j - 1;                          // 由于上面while中循环有j++,所以位会多出一位，这里减去
        }

        for (int i = point; i >= 1; i--)        // 逆序打印结果
        {
            printf("%d", array[i]);
        }
    }
}
```

2.栈实现队列

//1. 声明两个栈s1和s2。s1作为主栈，s2作为辅助栈。
//2. 为了达到队列先进先出的功能， 每次执行push操作时，先把s1的元素都压栈到s2中，然后将当前元素入栈s1。
//3. 再将s2的元素依次弹出，压入栈s1中，即实现了先入的元素更接近栈顶。
//4. 每次执行pop操作时，直接弹出s1的栈顶元素。

```
import java.util.Stack;

public class Solution {
    Stack<Integer> stack1 = new Stack<Integer>();
    Stack<Integer> stack2 = new Stack<Integer>();

    public void push(int node) {
        while(!stack1.empty()){
            stack2.push(stack1.pop());
        }
        stack1.push(node);
        while(!stack2.empty()){
            stack1.push(stack2.pop());
        }
    }

    public int pop() {
        return stack1.pop();
    }
}
```

3.单向链表

```
class SingleLinkedList {
    //先初始化一个头节点, 头节点不要动, 不存放具体的数据
    private HeroNode head = new HeroNode(0, "", "");


    //返回头节点
    public HeroNode getHead() {
        return head;
    }

    //添加节点到单向链表
    //思路，当不考虑编号顺序时
    //1. 找到当前链表的最后节点
    //2. 将最后这个节点的next 指向 新的节点
    public void add(HeroNode heroNode) {

        //因为head节点不能动，因此我们需要一个辅助遍历 temp
        HeroNode temp = head;
        //遍历链表，找到最后
        while(true) {
            //找到链表的最后
            if(temp.next == null) {//
                break;
            }
```

```
            //如果没有找到最后，将将temp后移
            temp = temp.next;
        }
        //当退出while循环时，temp就指向了链表的最后
        //将最后这个节点的next 指向 新的节点
        temp.next = heroNode;
    }

    //第二种方式在添加英雄时，根据排名将英雄插入到指定位置
    //(如果有这个排名，则添加失败，并给出提示)
    public void addByOrder(HeroNode heroNode) {
        //因为头节点不能动，因此我们仍然通过一个辅助指针(变量)来帮助找到添加的位置
        //因为单链表，因为我们找的temp 是位于 添加位置的前一个节点，否则插入不了
        HeroNode temp = head;
        boolean flag = false; // flag标志添加的编号是否存在，默认为false
        while(true) {
            if(temp.next == null) {//说明temp已经在链表的最后
                break; //
            }
            if(temp.next.no > heroNode.no) { //位置找到，就在temp的后面插入
                break;
            } else if (temp.next.no == heroNode.no) {//说明希望添加的heroNode的编号
已然存在

                flag = true; //说明编号存在
                break;
            }
            temp = temp.next; //后移，遍历当前链表
        }
        //判断flag 的值
        if(flag) { //不能添加，说明编号存在
            System.out.printf("准备插入的英雄的编号 %d 已经存在了，不能加入\n",
heroNode.no);
        } else {
            //插入到链表中，temp的后面
            heroNode.next = temp.next;
            temp.next = heroNode;
        }
    }

    //修改节点的信息，根据no编号来修改，即no编号不能改.
    //说明
    //1. 根据 newHeroNode 的 no 来修改即可
    public void update(HeroNode newHeroNode) {
        //判断是否空
        if(head.next == null) {
            System.out.println("链表为空~");
            return;
        }
        //找到需要修改的节点，根据no编号
        //定义一个辅助变量
        HeroNode temp = head.next;
        boolean flag = false; //表示是否找到该节点
        while(true) {
            if (temp == null) {
                break; //已经遍历完链表
            }
            if(temp.no == newHeroNode.no) {
```

```
                //找到
                flag = true;
                break;
            }
            temp = temp.next;
        }
        //根据flag 判断是否找到要修改的节点
        if(flag) {
            temp.name = newHeroNode.name;
            temp.nickname = newHeroNode.nickname;
        } else { //没有找到
            System.out.printf("没有找到 编号 %d 的节点，不能修改\n", newHeroNode.no);
        }
    }

    //方法：获取到单链表的节点的个数(如果是带头结点的链表，需求不统计头节点)
    /**
     *
     * @param head 链表的头节点
     * @return 返回的就是有效节点的个数
     */
    public static int getLength(HeroNode head) {
        if(head.next == null) { //空链表
            return 0;
        }
        int length = 0;
        //定义一个辅助的变量, 这里我们没有统计头节点
        HeroNode cur = head.next;
        while(cur != null) {
            length++;
            cur = cur.next; //遍历
        }
        return length;
    }

    //删除节点
    //思路
    //1. head 不能动，因此我们需要一个temp辅助节点找到待删除节点的前一个节点
    //2. 说明我们在比较时，是temp.next.no 和  需要删除的节点的no比较
    public void del(int no) {
        HeroNode temp = head;
        boolean flag = false; // 标志是否找到待删除节点的
        while(true) {
            if(temp.next == null) { //已经到链表的最后
                break;
            }
            if(temp.next.no == no) {
                //找到的待删除节点的前一个节点temp
                flag = true;
                break;
            }
            temp = temp.next; //temp后移，遍历
        }
        //判断flag
        if(flag) { //找到
            //可以删除
            temp.next = temp.next.next;
        }else {
```

```
            System.out.printf("要删除的 %d 节点不存在\n", no);
        }
    }

    //显示链表[遍历]
    public void list() {
        //判断链表是否为空
        if(head.next == null) {
            System.out.println("链表为空");
            return;
        }
        //因为头节点，不能动，因此我们需要一个辅助变量来遍历
        HeroNode temp = head.next;
        while(true) {
            //判断是否到链表最后
            if(temp == null) {
                break;
            }
            //输出节点的信息
            System.out.println(temp);
            //将temp后移， 一定小心
            temp = temp.next;
        }
    }
}

//定义HeroNode ， 每个HeroNode 对象就是一个节点
class HeroNode {
    public int no;
    public String name;
    public String nickname;
    public HeroNode next; //指向下一个节点
    //构造器
    public HeroNode(int no, String name, String nickname) {
        this.no = no;
        this.name = name;
        this.nickname = nickname;
    }
    //为了显示方法，我们重新toString
    @Override
    public String toString() {
        return "HeroNode [no=" + no + ", name=" + name + ", nickname=" +
nickname + "]";
    }

}
```

1.查找单链表中的倒数第k个结点（面试题）

```
//思路
    //1. 编写一个方法，接收head节点，同时接收一个index
    //2. index 表示是倒数第index个节点
    //3. 先把链表从头到尾遍历，得到链表的总的长度 getLength
    //4. 得到size 后，我们从链表的第一个开始遍历 (size-index)个，就可以得到
    //5. 如果找到了，则返回该节点，否则返回null
    public static HeroNode findLastIndexNode(HeroNode head, int index) {
```

```
        //判断如果链表为空，返回null
        if(head.next == null) {
            return null;//没有找到
        }
        //第一个遍历得到链表的长度(节点个数)
        int size = getLength(head);
        //第二次遍历  size-index 位置，就是我们倒数的第K个节点
        //先做一个index的校验
        if(index <=0 || index > size) {
            return null;
        }
        //定义给辅助变量， for 循环定位到倒数的index
        HeroNode cur = head.next; //3 // 3 - 1 = 2
        for(int i =0; i< size - index; i++) {
            cur = cur.next;
        }
        return cur;

    }
```

2.将单链表反转

```
//将单链表反转
    public static void reversetList(HeroNode head) {
        //如果当前链表为空，或者只有一个节点，无需反转，直接返回
        if(head.next == null || head.next.next == null) {
            return ;
        }

        //定义一个辅助的指针(变量)，帮助我们遍历原来的链表
        HeroNode cur = head.next;
        HeroNode next = null;// 指向当前节点[cur]的下一个节点
        HeroNode reverseHead = new HeroNode(0, "", "");
        //遍历原来的链表，每遍历一个节点，就将其取出，并放在新的链表reverseHead 的最前端
        //动脑筋
        while(cur != null) {
            next = cur.next;//先暂时保存当前节点的下一个节点，因为后面需要使用
            cur.next = reverseHead.next;//将cur的下一个节点指向新的链表的最前端
            reverseHead.next = cur; //将cur 连接到新的链表上
            cur = next;//让cur后移
        }
        //将head.next 指向 reverseHead.next , 实现单链表的反转
        head.next = reverseHead.next;
    }

//方式2：
    //可以利用栈这个数据结构，将各个节点压入到栈中，然后利用栈的先进后出的特点，就实现了逆序打印的效果
    public static void reversePrint(HeroNode head) {
        if(head.next == null) {
            return;//空链表，不能打印
        }
        //创建要给一个栈，将各个节点压入栈
        Stack<HeroNode> stack = new Stack<HeroNode>();
        HeroNode cur = head.next;
        //将链表的所有节点压入栈
        while(cur != null) {
```

```
            stack.push(cur);
            cur = cur.next; //cur后移，这样就可以压入下一个节点
        }
        //将栈中的节点进行打印,pop 出栈
        while (stack.size() > 0) {
            System.out.println(stack.pop()); //stack的特点是先进后出
        }
    }
```

4.树

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define OK 1


typedef struct Obj{
    const char *name;
    struct Obj *fchild, *schild, *tchild ;
}Obj;

/*
    一级材料
*/
int init_ironingot(Obj** p) {
    Obj* h = (Obj*)malloc(sizeof(Obj));



    if (h){
        (h->name) = "铁锭";
        h->fchild = NULL;
        h->schild = NULL;
        h->tchild = NULL;
        *p = h;
    }

    return OK;
}

int init_goldingot(Obj** p) {
    Obj* h = (Obj*)malloc(sizeof(Obj));

    if (h) {
        (h->name) = "金锭";
        h->fchild = NULL;
        h->schild = NULL;
        h->tchild = NULL;
    }

    *p = h;

    return OK;
}
```

```
int init_copperingot(Obj** p) {
    Obj* h = (Obj*)malloc(sizeof(Obj));

    if (h) {
        h->name = "铜锭";
        h->fchild = NULL;
        h->schild = NULL;
        h->tchild = NULL;
    }

    *p = h;

    return OK;
}

int init_log(Obj** p) {
    Obj* h = (Obj*)malloc(sizeof(Obj));

    if (h) {
        h->name = "原木";
        h->fchild = NULL;
        h->schild = NULL;
        h->tchild = NULL;
        *p = h;
    }

    return OK;
}

int init_gossamer(Obj** p) {
    Obj* h = (Obj*)malloc(sizeof(Obj));

    if (h) {
        (h->name) = "蜘蛛丝";
        h->fchild = NULL;
        h->schild = NULL;
        h->tchild = NULL;
    }

    *p = h;

    return OK;
}

/*
    二级材料
*/
int init_board(Obj** p) {
    Obj* h = (Obj*)malloc(sizeof(Obj));

    if (h) {
        (h->name) = "木板";
        init_log(&(h->fchild));
        h->schild = NULL;
        h->tchild = NULL;
    }

    *p = h;
```

```
    return OK;
}

/*
    三级材料
*/
int init_stick(Obj **p) {
    Obj *h = (Obj*)malloc(sizeof(Obj));

    if (h) {
        (h->name) = "木棒";
        init_board(&(h->fchild));
        h->schild = NULL;
        h->tchild = NULL;
    }

    *p = h;

    return OK;
}

/*
    四级材料
*/
int init_ironaxe(Obj **p) {
    Obj *h1  = (Obj*)malloc(sizeof(Obj));

    if (h1) {
        (h1->name) = "铁斧头";
        init_stick(&(h1->schild));
        init_ironingot(&(h1->fchild));
        h1->tchild = NULL;
    }

    *p = h1;
    printf("铁斧头初始化成功\n");

    return OK;
}

int init_ironshovel(Obj** p) {
    Obj* h1 = (Obj*)malloc(sizeof(Obj));

    if (h1) {
        (h1->name) = "铁锹";
        init_stick(&(h1->schild));
        init_ironingot(&(h1->fchild));
        h1->tchild = NULL;
    }

    *p = h1;

    printf("铁锹初始化成功\n");

    return OK;
}
```

```
int init_ironhoe(Obj** p) {
    Obj* h1 = (Obj*)malloc(sizeof(Obj));

    if (h1) {
        (h1->name) = "铁锄头";
        init_stick(&(h1->schild));
        init_ironingot(&(h1->fchild));
        h1->tchild = NULL;
    }

    *p = h1;

    printf("铁锄头初始化成功\n");

    return OK;
}

int init_fishingpole(Obj** p) {
    Obj* h1 = (Obj*)malloc(sizeof(Obj));

    if (h1) {
        h1->name = "钓鱼竿";
        init_stick(&(h1->schild));
        init_gossamer(&(h1->fchild));
        h1->tchild = NULL;
    }

    *p = h1;

    printf("钓鱼竿初始化成功\n");

    return OK;
}

int init_copperaxe(Obj** p) {
    Obj* h1 = (Obj*)malloc(sizeof(Obj));

    if (h1) {
        h1->name = "铜斧头";
        init_stick(&(h1->schild));
        init_copperingot(&(h1->fchild));
        h1->tchild = NULL;
    }

    *p = h1;

    printf("铜斧头初始化成功\n");

    return OK;
}

int init_coppershovel(Obj** p) {
    Obj* h1 = (Obj*)malloc(sizeof(Obj));

    if (h1) {
        h1->name = "铜锹";
        init_stick(&(h1->schild));
        init_copperingot(&(h1->fchild));
```

```
        h1->tchild = NULL;
    }

    *p = h1;

    printf("铜锹初始化成功\n");

    return OK;
}

int init_copperhoe(Obj** p) {
    Obj* h1 = (Obj*)malloc(sizeof(Obj));

    if (h1) {
        h1->name = "铜锄头";
        init_stick(&(h1->schild));
        init_copperingot(&(h1->fchild));
        h1->tchild = NULL;
    }

    *p = h1;

    printf("铜锄头初始化成功\n");

    return OK;
}

int init_goldaxe(Obj** p) {
    Obj* h1 = (Obj*)malloc(sizeof(Obj));

    if (h1) {
        h1->name = "金斧头";
        init_stick(&(h1->schild));
        init_copperingot(&(h1->fchild));
        h1->tchild = NULL;
    }

    *p = h1;

    printf("金斧头初始化成功\n");

    return OK;
}

int init_goldshovel(Obj** p) {
    Obj* h1 = (Obj*)malloc(sizeof(Obj));

    if (h1) {
        h1->name = "金锹";
        init_stick(&(h1->schild));
        init_copperingot(&(h1->fchild));
        h1->tchild = NULL;
    }

    *p = h1;

    printf("金锹初始化成功\n");
```

```
    return OK;
}

int init_goldhoe(Obj** p) {
    Obj* h1 = (Obj*)malloc(sizeof(Obj));

    if (h1) {
        h1->name = "金锄头";
        init_stick(&(h1->schild));
        init_copperingot(&(h1->fchild));
        h1->tchild = NULL;
    }

    *p = h1;

    printf("金锄头初始化成功\n");

    return OK;
}

int init_total(Obj **p) {
    init_ironaxe(p);
    init_ironshovel(p + 1);
    init_ironhoe(p + 2);
    init_fishingpole(p + 3);
    init_copperaxe(p + 4);
    init_coppershovel(p + 5);
    init_copperhoe(p + 6);
    init_goldaxe(p + 7);
    init_goldshovel(p + 8);
    init_goldhoe(p + 9);

    return 0;

}

int PostOrderTrvaverse(Obj* T, char *p) {
    int flag = 0;//用flag表示是否找到用户输入的材料，找到为1，  没找到为零
    if (T == NULL || flag == 1) {
        return flag;
    }

    if (strcmp(p,T->name) == 0){
        flag = 1;
    }
    if(flag != 1){
        flag = PostOrderTrvaverse(T->fchild, p);
    }
    if (flag != 1) {
        flag = PostOrderTrvaverse(T->schild, p);
    }
    if (flag != 1) {
        flag =PostOrderTrvaverse(T->tchild, p);
    }

    return flag;

}
```

```
int searchtop(char* p, Obj** list) {
    int flag;
    int flag2 =0;
    if (!list) {
        exit(-1);
    }
    printf("\n可以最终生成。。。\n");
    for (int i = 0; i < 10; i++) {
        flag = PostOrderTrvaverse(*(list + i), p);
        //printf("%d", flag);
        if (flag == 1) {
            printf("%s\t", (*(list + i))->name);
            flag = 0;
            flag2 = 1;
        }

    }
    if (flag2 == 0) {
        printf("\n没有找到");
    }
    printf("\n");
    return 0;
}

int menu() {
    printf("\n一级原材料有：铁锭 金锭 铜锭 原木 蜘蛛丝\n");
    printf("二级原材料有：木板\n");
    printf("三级原材料有：木棒\n");
    printf("四级原材料有：铁斧头 铁锹 铁锄头 钓鱼竿 铜斧头 铜锹 铜锄头 金斧头 金锹 金锄头
\n");


    return 0;
}

int main() {
    Obj* totallist[10];
    init_total(totallist);
    char c[10];
    menu();
    printf("输入需要查询的材料：");
    scanf("%s", c);

    searchtop(c, totallist);


    return 0;
}
```

5.排序

2)冒泡排序

原始数组：3, 9, -1, 10, 20

第一趟排序
(1) 3, 9, -1, 10, 20 // 如果相邻的元素逆序就交换
(2) 3, -1, 9, 10, 20
(3) 3, -1, 9, 10, 20
(4) 3, -1, 9, 10, **20**

第二趟排序
(1) -1, 3, 9, 10, **20 //交换**
(2) -1, 3, 9, 10, **20**
(3) -1, 3, 9, 10, **20**

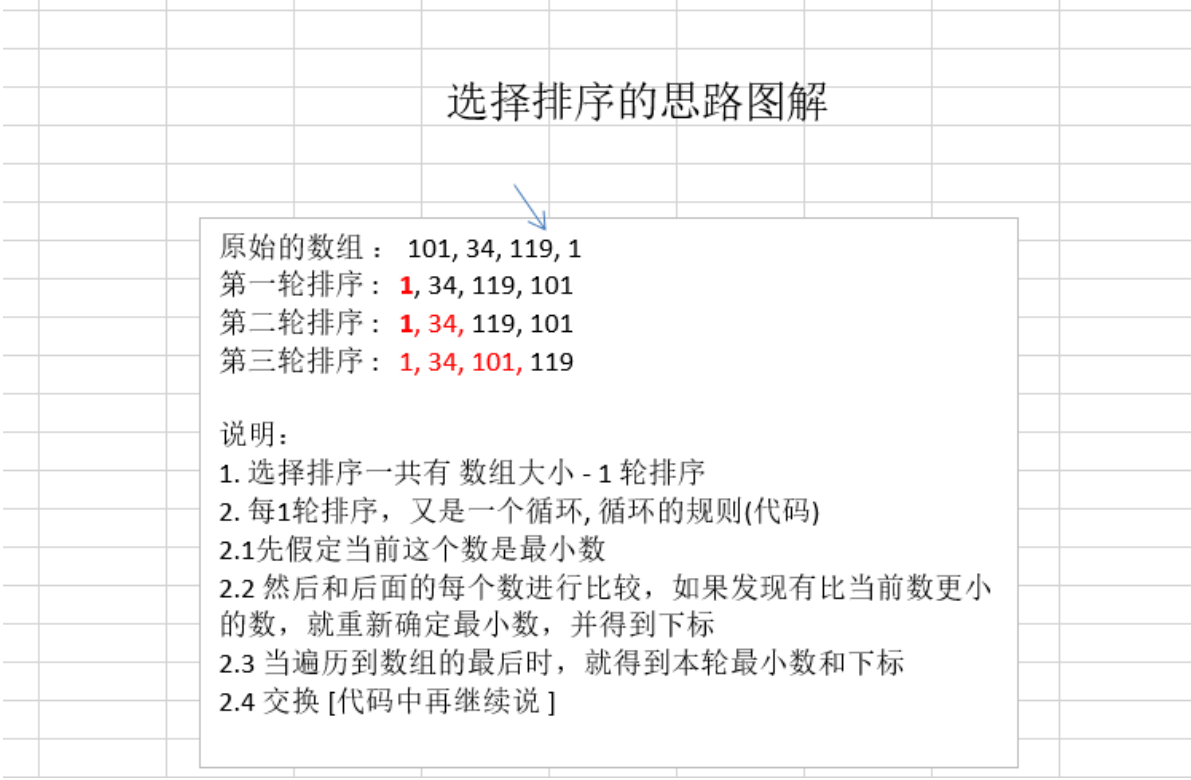
第三趟排序
(1) -1, 3, 9, 10, **20**
(2) -1, 3, 9, 10, **20**

第四趟排序
(1) -1, 3, 9, 10, **20**

小结冒泡排序规则
(1) 一共进行 数组的大小-1 次 大的循环
(2)每一趟排序的次数在逐渐的减少
(3) 如果我们发现在某趟排序中，没有发生一次交换， 可以提前结束冒泡排序。这个就是优化

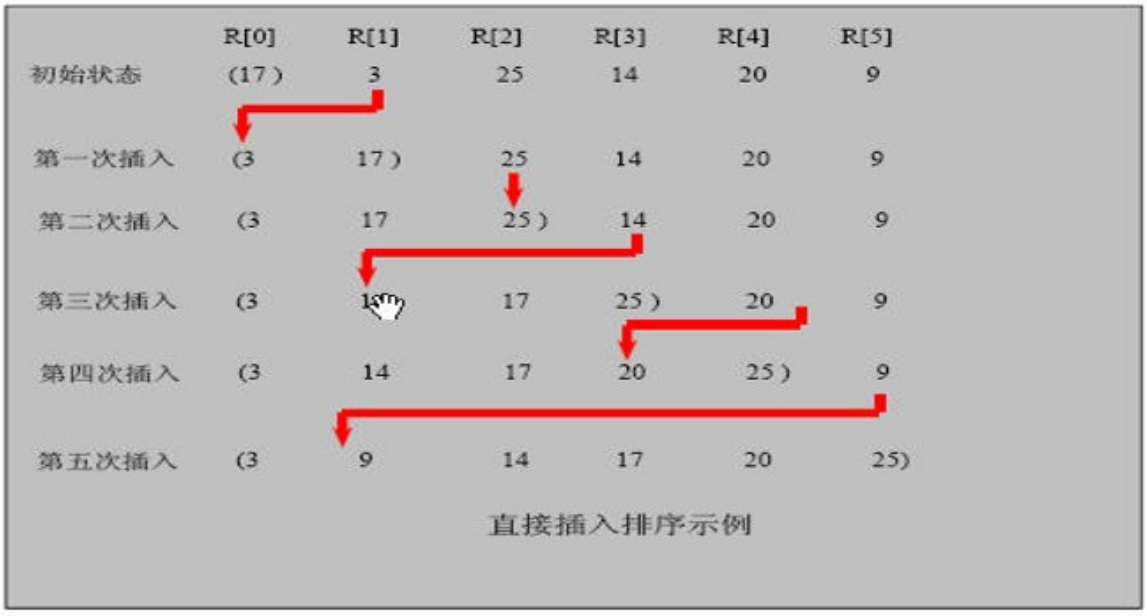
```
public static void bubbleSort(int[] arr) {

        int temp = 0;
        boolean flag = false;
        for (int i = 0; i < arr.length - 1; i++) {

            for (int j = 0; j < arr.length - 1 - i; j++) {

                if (arr[j] > arr[j + 1]) {
                    flag = true;
                    temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }

            if (!flag) {
                break;
            } else {
                flag = false;
            }
        }
    }//时间复杂度为n^2
```

3)选择排序




```
public static void selectSort(int[] arr) {
        //在推导的过程，我们发现了规律，因此，可以使用for来解决
        //选择排序时间复杂度是 O(n^2)
        for (int i = 0; i < arr.length - 1; i++) {
            int minIndex = i;
            int min = arr[i];
            for (int j = i + 1; j < arr.length; j++) {
                if (min > arr[j]) { // 说明假定的最小值，并不是最小
                    min = arr[j]; // 重置min
                    minIndex = j; // 重置minIndex
                }
            }

            // 将最小值，放在arr[0], 即交换
            if (minIndex != i) {
                arr[minIndex] = arr[i];
                arr[i] = min;
            }

            //System.out.println("第"+(i+1)+"轮后~~");
            //System.out.println(Arrays.toString(arr));// 1, 34, 119, 101
            //时间复杂度为n^2
        }
}
```

4)插入排序



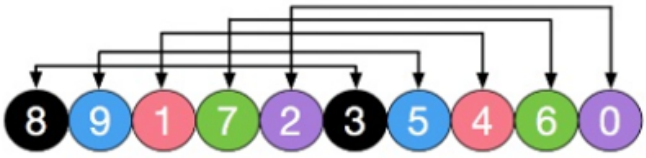

```
public static void insertSort(int[] arr) {
        int insertVal = 0;
        int insertIndex = 0;
        //使用for循环来把代码简化
        for(int i = 1; i < arr.length; i++) {
            //定义待插入的数
            insertVal = arr[i];
            insertIndex = i - 1; // 即arr[1]的前面这个数的下标

            // 给insertVal 找到插入的位置
            // 说明
            // 1. insertIndex >= 0 保证在给insertVal 找插入位置，不越界
            // 2. insertVal < arr[insertIndex] 待插入的数，还没有找到插入位置
            // 3. 就需要将 arr[insertIndex] 后移
            while (insertIndex >= 0 && insertVal < arr[insertIndex]) {
                arr[insertIndex + 1] = arr[insertIndex];// arr[insertIndex]
                insertIndex--;
            }
            // 当退出while循环时，说明插入的位置找到, insertIndex + 1
            // 举例：理解不了，我们一会 debug
            //这里我们判断是否需要赋值
            if(insertIndex + 1 != i) {
                arr[insertIndex + 1] = insertVal;
            }

            //System.out.println("第"+i+"轮插入");
            //System.out.println(Arrays.toString(arr));
        }
}
```

5)希尔排序

对插入排序的优化版本

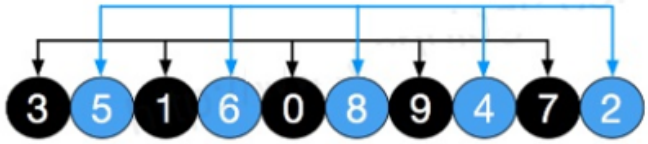
原始数组 以下数据元素颜色相同为一组

初始增量 gap=length/2=5，意味着整个数组被分为5组，[8,3] [9,5] [1,4] [7,6] [2,0]



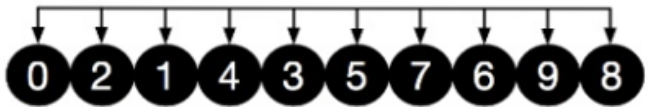
对这5组分别进行直接插入排序，结果如下，可以看到，像3，5，6这些小元素都被调到前面了，

然后缩小增量 gap=5/2=2,数组被分为2组 [3,1,0,9,7] [5,6,8,4,2]



对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。

再缩小增量gap=2/2=1，此时，整个数组为1组[0,2,1,4,3,5,7,6,9,8]，如下

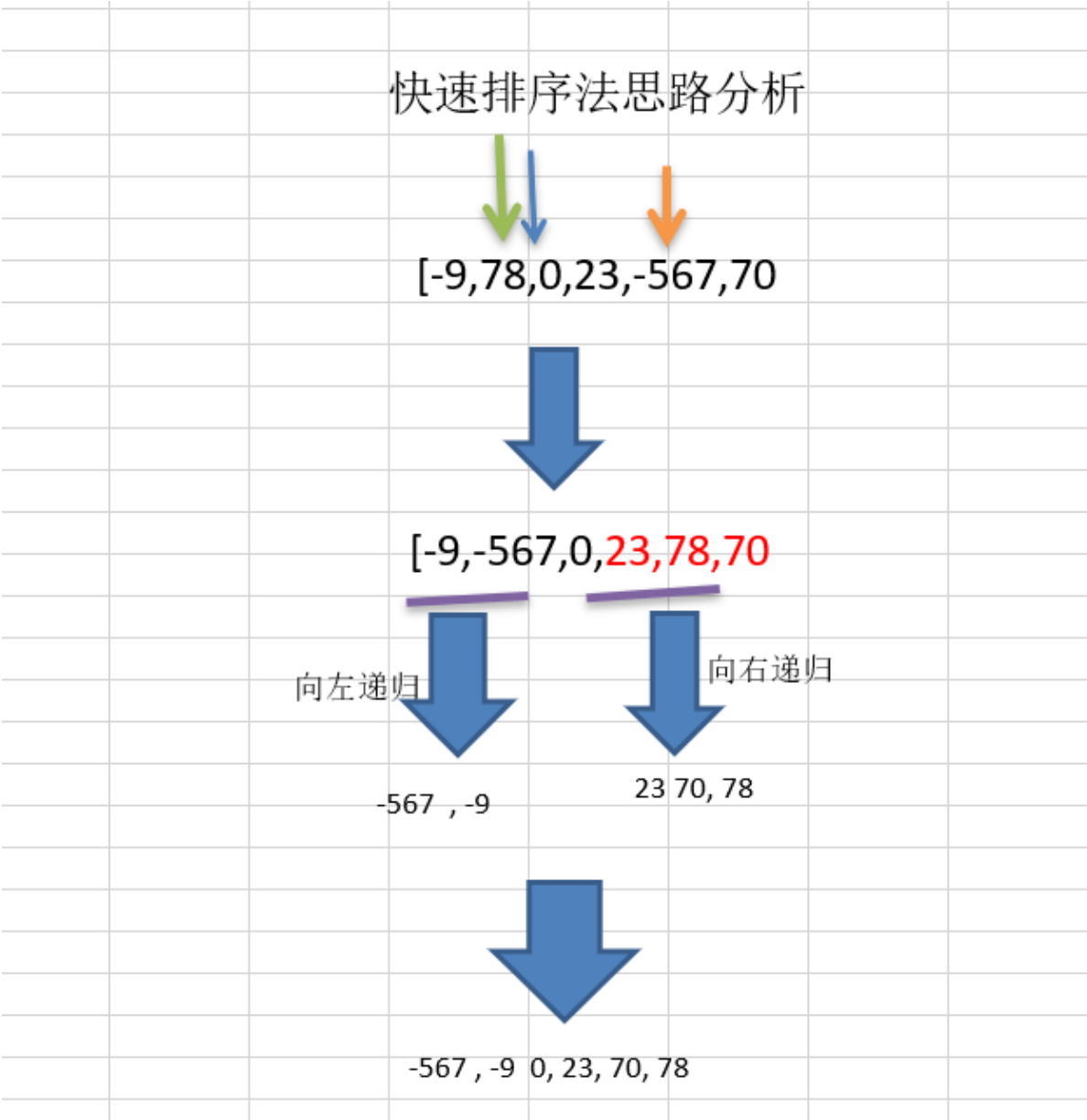



```
    // 希尔排序时， 对有序序列在插入时采用交换法，
    // 思路(算法) ===> 代码
    public static void shellSort(int[] arr) {

        int temp = 0;
        int count = 0;
        // 根据前面的逐步分析，使用循环处理
        for (int gap = arr.length / 2; gap > 0; gap /= 2) {
            for (int i = gap; i < arr.length; i++) {
                // 遍历各组中所有的元素(共gap组，每组有个元素)，步长gap
                for (int j = i - gap; j >= 0; j -= gap) {
                    // 如果当前元素大于加上步长后的那个元素，说明交换
                    if (arr[j] > arr[j + gap]) {
                        temp = arr[j];
                        arr[j] = arr[j + gap];
                        arr[j + gap] = temp;
                    }
                }
            }
            //System.out.println("希尔排序第" + (++count) + "轮 =" +
Arrays.toString(arr));
        }
    }
```

```
//移位法
public static void shellSort2(int[] arr) {

        // 增量gap，并逐步的缩小增量
        for (int gap = arr.length / 2; gap > 0; gap /= 2) {
```

```
            // 从第gap个元素，逐个对其所在的组进行直接插入排序
            for (int i = gap; i < arr.length; i++) {
                int j = i;
                int temp = arr[j];
                if (arr[j] < arr[j - gap]) {
                    while (j - gap >= 0 && temp < arr[j - gap]) {
                        //移动
                        arr[j] = arr[j-gap];
                        j -= gap;
                    }
                    //当退出while后，就给temp找到插入的位置
                    arr[j] = temp;
                }

            }
        }
    }
```

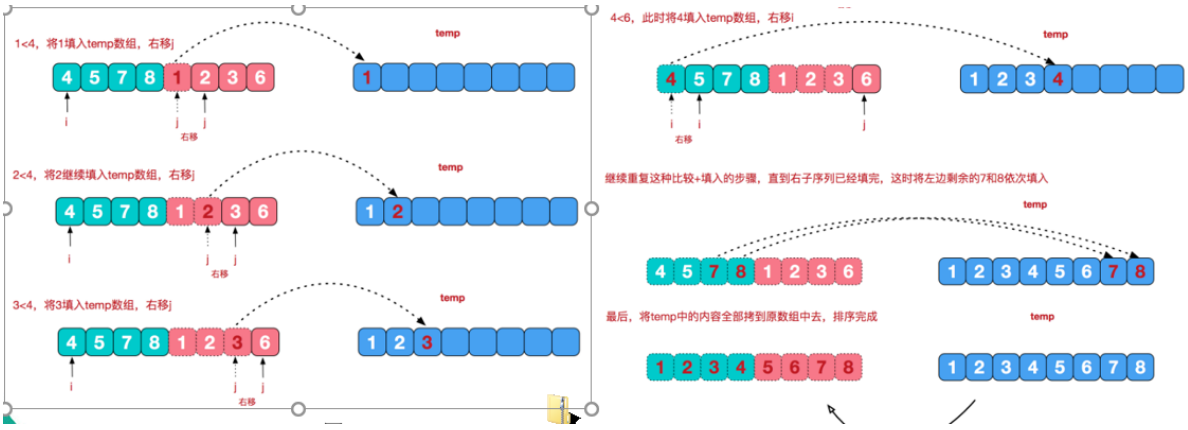
6)快速排序




```
public static void quickSort(int[] arr,int left, int right) {
        int l = left; //左下标
        int r = right; //右下标
        //pivot 中轴值
        int pivot = arr[(left + right) / 2];
        int temp = 0; //临时变量，作为交换时使用
        //while循环的目的是让比pivot 值小放到左边
        //比pivot 值大放到右边
        while( l < r) {
            //在pivot的左边一直找,找到大于等于pivot值,才退出
            while( arr[l] < pivot) {
                l += 1;
            }
            //在pivot的右边一直找,找到小于等于pivot值,才退出
            while(arr[r] > pivot) {
                r -= 1;
            }
            //如果l >= r说明pivot 的左右两的值，已经按照左边全部是
            //小于等于pivot值，右边全部是大于等于pivot值
            if( l >= r) {
                break;
            }
```

```
            //交换
            temp = arr[l];
            arr[l] = arr[r];
            arr[r] = temp;

            //如果交换完后，发现这个arr[l] == pivot值 相等 r--， 前移
            if(arr[l] == pivot) {
                r -= 1;
            }
            //如果交换完后，发现这个arr[r] == pivot值 相等 l++， 后移
            if(arr[r] == pivot) {
                l += 1;
            }
        }

        // 如果 l == r, 必须l++, r--, 否则为出现栈溢出
        if (l == r) {
            l += 1;
            r -= 1;
        }
        //向左递归
        if(left < r) {
            quickSort(arr, left, r);
        }
        //向右递归
        if(right > l) {
            quickSort(arr, l, right);
        }


    }
```

7)归并排序




```
public static void mergeSort(int[] arr, int left, int right, int[] temp) {
        if(left < right) {
            int mid = (left + right) / 2; //中间索引
            //向左递归进行分解
            mergeSort(arr, left, mid, temp);
            //向右递归进行分解
            mergeSort(arr, mid + 1, right, temp);
            //合并
            merge(arr, left, mid, right, temp);
```

```
        }
    }

    //合并的方法
    /**
     *
     * @param arr 排序的原始数组
     * @param left 左边有序序列的初始索引
     * @param mid 中间索引
     * @param right 右边索引
     * @param temp 做中转的数组
     */
    public static void merge(int[] arr, int left, int mid, int right, int[]
temp) {

        int i = left; // 初始化i, 左边有序序列的初始索引
        int j = mid + 1; //初始化j, 右边有序序列的初始索引
        int t = 0; // 指向temp数组的当前索引

        //(一)
        //先把左右两边(有序)的数据按照规则填充到temp数组
        //直到左右两边的有序序列，有一边处理完毕为止
        while (i <= mid && j <= right) {//继续
            //如果左边的有序序列的当前元素，小于等于右边有序序列的当前元素
            //即将左边的当前元素，填充到 temp数组
            //然后 t++, i++
            if(arr[i] <= arr[j]) {
                temp[t] = arr[i];
                t += 1;
                i += 1;
            } else { //反之,将右边有序序列的当前元素，填充到temp数组
                temp[t] = arr[j];
                t += 1;
                j += 1;
            }
        }

        //(二)
        //把有剩余数据的一边的数据依次全部填充到temp
        while( i <= mid) { //左边的有序序列还有剩余的元素，就全部填充到temp
            temp[t] = arr[i];
            t += 1;
            i += 1;
        }

        while( j <= right) { //右边的有序序列还有剩余的元素，就全部填充到temp
            temp[t] = arr[j];
            t += 1;
            j += 1;
        }


        //(三)
        //将temp数组的元素拷贝到arr
        //注意，并不是每次都拷贝所有
        t = 0;
        int tempLeft = left; //
```

```
        //第一次合并 tempLeft = 0 , right = 1 //  tempLeft = 2  right = 3 // tL=0
ri=3
        //最后一次 tempLeft = 0  right = 7
        while(tempLeft <= right) {
            arr[tempLeft] = temp[t];
            t += 1;
            tempLeft += 1;
        }

    }

}
```

6.查找

1）线性查找

2）二分查找

```
public static int binarySearch(int[] arr, int left, int right, int findVal) {

        if (left > right) {
            return -1;
        }
        int mid = (left + right) / 2;
        int midVal = arr[mid];

        if (findVal > midVal) {
            return binarySearch(arr, mid + 1, right, findVal);
        } else if (findVal < midVal) {
            return binarySearch(arr, left, mid - 1, findVal);
        } else {

            return mid;
        }

    }
```

7.图

1）图的创建

```
public class Graph {

    private ArrayList<String> vertexList; //存储顶点集合
    private int[][] edges; //存储图对应的邻结矩阵
    private int numOfEdges; //表示边的数目
    //定义给数组boolean[], 记录某个结点是否被访问
    private boolean[] isVisited;

    //构造器
    public Graph(int n) {
        //初始化矩阵和vertexList
        edges = new int[n][n];
        vertexList = new ArrayList<String>(n);
        numOfEdges = 0;
```

```
    }

    //图中常用的方法
    //返回结点的个数
    public int getNumOfVertex() {
        return vertexList.size();
    }
    //显示图对应的矩阵
    public void showGraph() {
        for(int[] link : edges) {
            System.err.println(Arrays.toString(link));
        }
    }
    //得到边的数目
    public int getNumOfEdges() {
        return numOfEdges;
    }
    //返回结点i(下标)对应的数据 0->"A" 1->"B" 2->"C"
    public String getValueByIndex(int i) {
        return vertexList.get(i);
    }
    //返回v1和v2的权值
    public int getWeight(int v1, int v2) {
        return edges[v1][v2];
    }
    //插入结点
    public void insertVertex(String vertex) {
        vertexList.add(vertex);
    }
    //添加边
    /**
     *
     * @param v1 表示点的下标即使第几个顶点  "A"-"B" "A"->0 "B"->1
     * @param v2 第二个顶点对应的下标
     * @param weight 表示
     */
    public void insertEdge(int v1, int v2, int weight) {
        edges[v1][v2] = weight;
        edges[v2][v1] = weight;
        numOfEdges++;
    }
}
```

2）深度优先算法

```
//得到第一个邻接结点的下标 w
    /**
     *
     * @param index
     * @return 如果存在就返回对应的下标，否则返回-1
     */
    public int getFirstNeighbor(int index) {
        for(int j = 0; j < vertexList.size(); j++) {
            if(edges[index][j] > 0) {
                return j;
            }
```

```
        }
        return -1;
    }
    //根据前一个邻接结点的下标来获取下一个邻接结点
    public int getNextNeighbor(int v1, int v2) {
        for(int j = v2 + 1; j < vertexList.size(); j++) {
            if(edges[v1][j] > 0) {
                return j;
            }
        }
        return -1;
    }
//深度优先遍历算法
    //i 第一次就是 0
    private void dfs(boolean[] isVisited, int i) {
        //首先我们访问该结点,输出
        System.out.print(getValueByIndex(i) + "->");
        //将结点设置为已经访问
        isVisited[i] = true;
        //查找结点i的第一个邻接结点w
        int w = getFirstNeighbor(i);
        while(w != -1) {//说明有
            if(!isVisited[w]) {
                dfs(isVisited, w);
            }
            //如果w结点已经被访问过
            w = getNextNeighbor(i, w);
        }

    }

    //对dfs 进行一个重载, 遍历我们所有的结点，并进行 dfs
    public void dfs() {
        isVisited = new boolean[vertexList.size()];
        //遍历所有的结点，进行dfs[回溯]
        for(int i = 0; i < getNumOfVertex(); i++) {
            if(!isVisited[i]) {
                dfs(isVisited, i);
            }
        }
    }
```

3）广度优先算法

```
//对一个结点进行广度优先遍历的方法
    private void bfs(boolean[] isVisited, int i) {
        int u ; // 表示队列的头结点对应下标
        int w ; // 邻接结点w
        //队列，记录结点访问的顺序
        LinkedList queue = new LinkedList();
        //访问结点，输出结点信息
        System.out.print(getValueByIndex(i) + "=>");
        //标记为已访问
        isVisited[i] = true;
        //将结点加入队列
        queue.addLast(i);
```

```
        while( !queue.isEmpty()) {
            //取出队列的头结点下标
            u = (Integer)queue.removeFirst();
            //得到第一个邻接结点的下标 w
            w = getFirstNeighbor(u);
            while(w != -1) {//找到
                //是否访问过
                if(!isVisited[w]) {
                    System.out.print(getValueByIndex(w) + "=>");
                    //标记已经访问
                    isVisited[w] = true;
                    //入队
                    queue.addLast(w);
                }
                //以u为前驱点，找w后面的下一个邻结点
                w = getNextNeighbor(u, w); //体现出我们的广度优先
            }
        }

    }

    //遍历所有的结点，都进行广度优先搜索
    public void bfs() {
        isVisited = new boolean[vertexList.size()];
        for(int i = 0; i < getNumOfVertex(); i++) {
            if(!isVisited[i]) {
                bfs(isVisited, i);
            }
        }
    }
```

8.分治算法

分治算法的基本步骤

分治法在每一层递归上都有三个步骤：

1)分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题

2)解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题

3)合并：将各个子问题的解合并为原问题的解。

```
public static void hanoiTower(int num, char a, char b, char c) {
        //如果只有一个盘
        if(num == 1) {
            System.out.println("第1个盘从 " + a + "->" + c);
        } else {
            //如果我们有 n >= 2 情况，我们总是可以看做是两个盘 1.最下边的一个盘 2. 上面的所有盘
            //1. 先把 最上面的所有盘 A->B， 移动过程会使用到 c
            hanoiTower(num - 1, a, c, b);
            //2. 把最下边的盘 A->C
            System.out.println("第" + num + "个盘从 " + a + "->" + c);
            //3. 把B塔的所有盘 从 B->C , 移动过程使用到 a塔
            hanoiTower(num - 1, b, a, c);

        }
```

```
    }
```

9.动态规划算法

```
//创建二维数组,
        //v[i][j] 表示在前i个物品中能够装入容量为j的背包中的最大价值
        int[][] v = new int[n+1][m+1];
        //为了记录放入商品的情况，我们定一个二维数组
        int[][] path = new int[n+1][m+1];

        //初始化第一行和第一列, 这里在本程序中，可以不去处理，因为默认就是0
        for(int i = 0; i < v.length; i++) {
            v[i][0] = 0; //将第一列设置为0
        }
        for(int i=0; i < v[0].length; i++) {
            v[0][i] = 0; //将第一行设置0
        }


        //根据前面得到公式来动态规划处理
        for(int i = 1; i < v.length; i++) { //不处理第一行 i是从1开始的
            for(int j=1; j < v[0].length; j++) {//不处理第一列, j是从1开始的
                //公式
                if(w[i-1]> j) { // 因为我们程序i 是从1开始的，因此原来公式中的 w[i] 修改
成 w[i-1]
                    v[i][j]=v[i-1][j];
                } else {
                    //说明:
                    //因为我们的i 从1开始的， 因此公式需要调整成
                    //v[i][j]=Math.max(v[i-1][j], val[i-1]+v[i-1][j-w[i-1]]);
                    //v[i][j] = Math.max(v[i - 1][j], val[i - 1] + v[i - 1][j -
w[i - 1]]);
                    //为了记录商品存放到背包的情况，我们不能直接的使用上面的公式，需要使用
if-else来体现公式
                    if(v[i - 1][j] < val[i - 1] + v[i - 1][j - w[i - 1]]) {
                        v[i][j] = val[i - 1] + v[i - 1][j - w[i - 1]];
                        //把当前的情况记录到path
                        path[i][j] = 1;
                    } else {
                        v[i][j] = v[i - 1][j];
                    }

                }
            }
        }

        //输出一下v 看看目前的情况
        for(int i =0; i < v.length;i++) {
            for(int j = 0; j < v[i].length;j++) {
                System.out.print(v[i][j] + " ");
            }
            System.out.println();
        }

        int i = path.length - 1; //行的最大下标
        int j = path[0].length - 1;  //列的最大下标
        while(i > 0 && j > 0 ) { //从path的最后开始找
```

```
            if(path[i][j] == 1) {
                System.out.printf("第%d个商品放入到背包\n", i);
                j -= w[i-1]; //w[i-1]
            }
            i--;
        }

    }
```

```
#include <stdio.h>

int house[] = {2, 3, 5, 6,7, 8,9, 3};
int len = sizeof(house)/sizeof(int);


int total(int i, int *house){
    int s;
    if(i<2){
        s = house[i];
    }
    else{

        if(total(i-1) > total(i-2) + house[i]){
            s = total(i-1);
        }
        else{
            s = total(i-2) + house[i];
        }
    }

    return s;
}
```

10.寻路算法

```
public class Node implements Comparable<Node>
{

    public Coord coord; // 坐标
    public Node parent; // 父结点
    public double G; // G：是个准确的值，是起点到当前结点的代价
    public double H; // H：是个估值，当前结点到目的结点的估计代价

    public Node(int x, int y)
    {
        this.coord = new Coord(x, y);
    }

    public Node(Coord coord, Node parent, double g, double h)
    {
        this.coord = coord;
        this.parent = parent;
        G = g;
        H = h;
    }
```

```
    @Override
    public int compareTo(Node o)
    {
        if (o == null) return -1;
        if (G + H > o.G + o.H)
            return 1;
        else if (G + H < o.G + o.H) return -1;
        return 0;
    }
}

public class Coord {

    public int x;
    public int y;

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

    @Override
    public boolean equals(Object obj)
    {
        if (obj == null) return false;
        if (obj instanceof Coord)
        {
            Coord c = (Coord) obj;
            return x == c.x && y == c.y;
        }
        return false;
    }
}

public class MapInfo {
    public int[][] maps; // 二维数组的地图
    public int width; // 地图的宽
    public int hight; // 地图的高
    public Node start; // 起始结点
    public Node end; // 最终结点

    public MapInfo(int[][] maps, int width, int hight, Node start, Node end)
    {
        this.maps = maps;
        this.width = width;
        this.hight = hight;
        this.start = start;
        this.end = end;
    }
}
```

AStar

```
import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Queue;

/**
 *
 * ClassName: AStar
 * @Description: A星算法
 * @author
 */
public class AStar {
    public final static int BAR = 1; // 障碍值
    public final static int PATH = 2; // 路径
    public final static double DIRECT_VALUE = 1; // 横竖移动代价
    public final static double OBLIQUE_VALUE = 1.4; // 斜移动代价

    Queue<Node> openList = new PriorityQueue<Node>(); // 优先队列(升序)
    List<Node> closeList = new ArrayList<Node>();

    /**
     * 开始算法
     */
    public void start(MapInfo mapInfo)
    {
        if(mapInfo==null) return;
        // clean
        openList.clear();
        closeList.clear();
        // 开始搜索
        openList.add(mapInfo.start);//在openList添加起点
        moveNodes(mapInfo);
    }

    /**
     * 移动当前结点
     */
    private void moveNodes(MapInfo mapInfo)
    {
        while (!openList.isEmpty())
        {
            Node current = openList.poll();//删除第一个元素,并返回这个元素
            closeList.add(current);
            addNeighborNodeInOpen(mapInfo,current);
            if (isCoordInClose(mapInfo.end.coord))//判断是否找到重点
            {
                drawPath(mapInfo.maps, mapInfo.end);
                break;
            }
        }
    }

    /**
     * 在二维数组中绘制路径
     */
```

```
    private void drawPath(int[][] maps, Node end)
    {
        if(end==null||maps==null) return;
        System.out.println("总代价：" + end.G);
        while (end != null)
        {
            Coord c = end.coord;
            maps[c.y][c.x] = PATH;//地图对应的位置改为2
            end = end.parent;//?
        }
    }

    /**
     * 添加所有邻结点到open表
     */
    private void addNeighborNodeInOpen(MapInfo mapInfo,Node current)
    {
        int x = current.coord.x;
        int y = current.coord.y;
        // 左
        addNeighborNodeInOpen(mapInfo,current, x - 1, y, DIRECT_VALUE);
        // 上
        addNeighborNodeInOpen(mapInfo,current, x, y - 1, DIRECT_VALUE);
        // 右
        addNeighborNodeInOpen(mapInfo,current, x + 1, y, DIRECT_VALUE);
        // 下
        addNeighborNodeInOpen(mapInfo,current, x, y + 1, DIRECT_VALUE);
        // 左上
        addNeighborNodeInOpen(mapInfo,current, x - 1, y - 1, OBLIQUE_VALUE );
        // 右上
        addNeighborNodeInOpen(mapInfo,current, x + 1, y - 1, OBLIQUE_VALUE );
        // 右下
        addNeighborNodeInOpen(mapInfo,current, x + 1, y + 1, OBLIQUE_VALUE );
        // 左下
        addNeighborNodeInOpen(mapInfo,current, x - 1, y + 1, OBLIQUE_VALUE );
    }

    /**
     * 添加一个邻结点到open表
     */
    private void addNeighborNodeInOpen(MapInfo mapInfo,Node current, int x, int
y, double value)
    {
        if (canAddNodeToOpen(mapInfo,x, y))//判断点是否可以走
        {
            Node end=mapInfo.end;
            Node start=mapInfo.start;
            Coord coord = new Coord(x, y);
            double G = calcH(start.coord,coord); // 计算邻结点的G值（起点到当前点
            Node child = findNodeInOpen(coord);//cood是否在open列表里
            if (child == null)
            {
                double H=calcH(end.coord,coord); // 计算H值（当前点到重点
                if(isEndNode(end.coord,coord))
                {
                    child=end;
                    child.parent=current;
                    child.G=calcH(start.coord,end.coord);
```

```
                    child.H=0;
                }
                else
                {
                    child = new Node(coord, current, G, H);
                }
                openList.add(child);
            }
            else if (child.G > G)
            {
                child.G = G;
                child.parent = current;
                openList.add(child);
            }
        }
    }

    /**
     * 从Open列表中查找结点
     */
    private Node findNodeInOpen(Coord coord)
    {
        if (coord == null || openList.isEmpty()) return null;
        for (Node node : openList)
        {
            if (node.coord.equals(coord))
            {
                return node;
            }
        }
        return null;
    }


    /**
     * 计算H的估值：“欧几里得”法，坐标分别取差值相加
     */
    private double calcH(Coord end,Coord coord)
    {
        return (Math.sqrt(Math.pow((end.x - coord.x),2) + Math.pow((end.y -
coord.y),2) ))* DIRECT_VALUE;
    }

    /**
     * 判断结点是否是最终结点
     */
    private boolean isEndNode(Coord end,Coord coord)
    {
        return coord != null && end.equals(coord);
    }

    /**
     * 判断结点能否放入Open列表
     */
    private boolean canAddNodeToOpen(MapInfo mapInfo,int x, int y)
    {
        // 是否在地图中
```

```
        if (x < 0 || x >= mapInfo.width || y < 0 || y >= mapInfo.hight) return
false;
        // 判断是否是不可通过的结点
        if (mapInfo.maps[y][x] == BAR) return false;
        // 判断结点是否存在close表
        if (isCoordInClose(x, y)) return false;

        return true;
    }

    /**
     * 判断坐标是否在close表中
     */
    private boolean isCoordInClose(Coord coord)
    {
        return coord!=null&&isCoordInClose(coord.x, coord.y);
    }

    /**
     * 判断坐标是否在close表中
     */
    private boolean isCoordInClose(int x, int y)
    {
        if (closeList.isEmpty()) return false;
        for (Node node : closeList)
        {
            if (node.coord.x == x && node.coord.y == y)
            {
                return true;
            }
        }
        return false;
    }
}
```

BFS

```
import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Queue;

/**
 *
 * ClassName:
 * @Description:
 * @author
 */
public class Bfs
{
    public final static int BAR = 1; // 障碍值
    public final static int PATH = 2; // 路径
    public final static double DIRECT_VALUE = 1.0; // 横竖移动代价
    public final static double OBLIQUE_VALUE = 1.4; // 斜移动代价

    List<Node> openList = new ArrayList<Node>(); // 优先队列(升序)
```

```
    List<Node> closeList = new ArrayList<Node>();

    /**
     * 开始算法
     */
    public void start(MapInfo mapInfo)
    {
        if(mapInfo==null) return;
        // clean
        openList.clear();
        closeList.clear();
        // 开始搜索
        openList.add(mapInfo.start);//在openList添加起点
        moveNodes(mapInfo);
    }

    /**
     * 移动当前结点
     */
    private void moveNodes(MapInfo mapInfo)
    {
        while (!openList.isEmpty())
        {
            Node current = openList.remove(0);//删除第一个元素
            closeList.add(current);
            addNeighborNodeInOpen(mapInfo,current);
            if (isCoordInClose(mapInfo.end.coord))
            {
                drawPath(mapInfo.maps, mapInfo.end);
                break;
            }
        }
    }

    /**
     * 在二维数组中绘制路径
     */
    private void drawPath(int[][] maps, Node end)
    {
        if(end==null||maps==null) return;
        System.out.println("总代价: " + end.G);
        while (end != null)
        {
            Coord c = end.coord;
            maps[c.y][c.x] = PATH;//地图对应的位置改为2
            end = end.parent;//?
        }
    }

    /**
     * 添加所有邻结点到open表
     */
    private void addNeighborNodeInOpen(MapInfo mapInfo,Node current)
    {
        int x = current.coord.x;
        int y = current.coord.y;
        // 左
        addNeighborNodeInOpen(mapInfo,current, x - 1, y, DIRECT_VALUE);
```

```
        // 上
        addNeighborNodeInOpen(mapInfo,current, x, y - 1, DIRECT_VALUE);
        // 右
        addNeighborNodeInOpen(mapInfo,current, x + 1, y, DIRECT_VALUE);
        // 下
        addNeighborNodeInOpen(mapInfo,current, x, y + 1, DIRECT_VALUE);
        // 左上
        addNeighborNodeInOpen(mapInfo,current, x - 1, y - 1, OBLIQUE_VALUE );
        // 右上
        addNeighborNodeInOpen(mapInfo,current, x + 1, y - 1, OBLIQUE_VALUE );
        // 右下
        addNeighborNodeInOpen(mapInfo,current, x + 1, y + 1, OBLIQUE_VALUE );
        // 左下
        addNeighborNodeInOpen(mapInfo,current, x - 1, y + 1, OBLIQUE_VALUE );
    }

    /**
     * 添加一个邻结点到open表
     */
    private void addNeighborNodeInOpen(MapInfo mapInfo,Node current, int x, int
y, double value)
    {
        if (canAddNodeToOpen(mapInfo,x, y))
        {
            Node end=mapInfo.end;
            Coord coord = new Coord(x, y);
            double G = current.G + value; // 计算邻结点的G值
            Node child = findNodeInOpen(coord);
            if (child == null)
            {
                double H=calcH(current.coord,coord); // 计算H值
                if(isEndNode(end.coord,coord))
                {
                    child=end;
                    child.parent=current;
                    child.G=G;
                    child.H=H;
                }
                else
                {
                    child = new Node(coord, current, G, H);
                }
                openList.add(child);
            }
            else if (child.G > G)
            {
                child.G = G;
                child.parent = current;
                openList.add(child);
            }
        }
    }

    /**
     * 从Open列表中查找结点
     */
    private Node findNodeInOpen(Coord coord)
    {
```

```
        if (coord == null || openList.isEmpty()) return null;
        for (Node node : openList)
        {
            if (node.coord.equals(coord))
            {
                return node;
            }
        }
        return null;
    }


    /**
     * 计算H的估值：“曼哈顿”法，坐标分别取差值相加
     */
    private double calcH(Coord end,Coord coord)
    {
        return (Math.abs(end.x - coord.x) + Math.abs(end.y - coord.y)) *
DIRECT_VALUE;
    }

    /**
     * 判断结点是否是最终结点
     */
    private boolean isEndNode(Coord end,Coord coord)
    {
        return coord != null && end.equals(coord);
    }

    /**
     * 判断结点能否放入Open列表
     */
    private boolean canAddNodeToOpen(MapInfo mapInfo,int x, int y)
    {
        // 是否在地图中
        if (x < 0 || x >= mapInfo.width || y < 0 || y >= mapInfo.hight) return
false;
        // 判断是否是不可通过的结点
        if (mapInfo.maps[y][x] == BAR) return false;
        // 判断结点是否存在close表
        if (isCoordInClose(x, y)) return false;

        return true;
    }

    /**
     * 判断坐标是否在close表中
     */
    private boolean isCoordInClose(Coord coord)
    {
        return coord!=null&&isCoordInClose(coord.x, coord.y);
    }

    /**
     * 判断坐标是否在close表中
     */
    private boolean isCoordInClose(int x, int y)
    {
```

```
        if (closeList.isEmpty()) return false;
        for (Node node : closeList)
        {
            if (node.coord.x == x && node.coord.y == y)
            {
                return true;
            }
        }
        return false;
    }
}
```

Dijkstra

```
import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Queue;

/**
 *
 * ClassName: Dijkstra
 * @Description: Dijkstra算法
 * @author
 */
public class Dijkstra
{
    public final static int BAR = 1; // 障碍值
    public final static int PATH = 2; // 路径
    public final static double DIRECT_VALUE = 1.0; // 横竖移动代价
    public final static double OBLIQUE_VALUE = 1.4; // 斜移动代价

    Queue<Node> openList = new PriorityQueue<Node>(); // 优先队列(升序)
    List<Node> closeList = new ArrayList<Node>();

    /**
     * 开始算法
     */
    public void start(MapInfo mapInfo)
    {
        if(mapInfo==null) return;
        // clean
        openList.clear();
        closeList.clear();
        // 开始搜索
        openList.add(mapInfo.start);//在openList添加起点
        moveNodes(mapInfo);
    }

    /**
     * 移动当前结点
     */
    private void moveNodes(MapInfo mapInfo)
    {
        while (!openList.isEmpty())
        {
```

```
            Node current = openList.poll();//删除第一个元素
            closeList.add(current);
            addNeighborNodeInOpen(mapInfo,current);
            if (isCoordInClose(mapInfo.end.coord))
            {
                drawPath(mapInfo.maps, mapInfo.end);
                break;
            }
        }
    }

    /**
     * 在二维数组中绘制路径
     */
    private void drawPath(int[][] maps, Node end)
    {
        if(end==null||maps==null) return;
        System.out.println("总代价: " + end.G);
        while (end != null)
        {
            Coord c = end.coord;
            maps[c.y][c.x] = PATH;//地图对应的位置改为2
            end = end.parent;//?
        }
    }

    /**
     * 添加所有邻结点到open表
     */
    private void addNeighborNodeInOpen(MapInfo mapInfo,Node current)
    {
        int x = current.coord.x;
        int y = current.coord.y;
        // 左
        addNeighborNodeInOpen(mapInfo,current, x - 1, y, DIRECT_VALUE);
        // 上
        addNeighborNodeInOpen(mapInfo,current, x, y - 1, DIRECT_VALUE);
        // 右
        addNeighborNodeInOpen(mapInfo,current, x + 1, y, DIRECT_VALUE);
        // 下
        addNeighborNodeInOpen(mapInfo,current, x, y + 1, DIRECT_VALUE);
        // 左上
        addNeighborNodeInOpen(mapInfo,current, x - 1, y - 1, OBLIQUE_VALUE );
        // 右上
        addNeighborNodeInOpen(mapInfo,current, x + 1, y - 1, OBLIQUE_VALUE );
        // 右下
        addNeighborNodeInOpen(mapInfo,current, x + 1, y + 1, OBLIQUE_VALUE );
        // 左下
        addNeighborNodeInOpen(mapInfo,current, x - 1, y + 1, OBLIQUE_VALUE );
    }

    /**
     * 添加一个邻结点到open表
     */
    private void addNeighborNodeInOpen(MapInfo mapInfo,Node current, int x, int
y, double value)
    {
        if (canAddNodeToOpen(mapInfo,x, y))
```

```
        {
            Node end=mapInfo.end;
            Coord coord = new Coord(x, y);
            double G = current.G + value; // 计算邻结点的G值
            Node child = findNodeInOpen(coord);
            if (child == null)
            {
                double H=calcH(current.coord,coord); // 计算H值
                if(isEndNode(end.coord,coord))
                {
                    child=end;
                    child.parent=current;
                    child.G=G;
                    child.H=H;
                }
                else
                {
                    child = new Node(coord, current, G, H);
                }
                openList.add(child);
            }
            else if (child.G > G)
            {
                child.G = G;
                child.parent = current;
                openList.add(child);
            }
        }
    }

    /**
     * 从Open列表中查找结点
     */
    private Node findNodeInOpen(Coord coord)
    {
        if (coord == null || openList.isEmpty()) return null;
        for (Node node : openList)
        {
            if (node.coord.equals(coord))
            {
                return node;
            }
        }
        return null;
    }


    /**
     * 计算H的估值：“曼哈顿”法，坐标分别取差值相加
     */
    private double calcH(Coord end,Coord coord)
    {
        return (Math.abs(end.x - coord.x) + Math.abs(end.y - coord.y)) *
DIRECT_VALUE;
    }

    /**
     * 判断结点是否是最终结点
```

```
     */
    private boolean isEndNode(Coord end,Coord coord)
    {
        return coord != null && end.equals(coord);
    }

    /**
     * 判断结点能否放入Open列表
     */
    private boolean canAddNodeToOpen(MapInfo mapInfo,int x, int y)
    {
        // 是否在地图中
        if (x < 0 || x >= mapInfo.width || y < 0 || y >= mapInfo.hight) return
false;
        // 判断是否是不可通过的结点
        if (mapInfo.maps[y][x] == BAR) return false;
        // 判断结点是否存在close表
        if (isCoordInClose(x, y)) return false;

        return true;
    }

    /**
     * 判断坐标是否在close表中
     */
    private boolean isCoordInClose(Coord coord)
    {
        return coord!=null&&isCoordInClose(coord.x, coord.y);
    }

    /**
     * 判断坐标是否在close表中
     */
    private boolean isCoordInClose(int x, int y)
    {
        if (closeList.isEmpty()) return false;
        for (Node node : closeList)
        {
            if (node.coord.x == x && node.coord.y == y)
            {
                return true;
            }
        }
        return false;
    }
}
```