

# 实验一：递归与分治

## 一、快速排序及第k小数

### 1.1 快速排序

#### 1.1.1 问题分析与算法思路

快速排序的基本思想为：

确定一个基准值 $p$ ，从序列的两端开始遍历，直到在左侧找到一个大于 $p$ 的数，在右侧找到一个小于 $p$ 的数，然后交换两数位置，循环此操作一直到指针相交。通过这样的操作可以保证 $p$ 左侧的数均小于 $p$ ，右侧的数均大于 $p$ 。结果可以将原序列分割为两个子序列，通过同样的方式对子序列进行操作直到子序列个数为1，即可实现序列排序。

这里我们采用 $i$ ， $j$ 指针分别指向第一个和最后一个，并分别向右，向左移动实现。

#### 1.1.2 算法设计与代码

```
void quick_sort(int q[], int l, int r)
{
    if (l >= r) return; //判边界
    int x = q[l], i = l - 1, j = r + 1;
    while (i < j)
    {
        do i++; while (q[i] < x);
        do j--; while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    quick_sort(q, l, j), quick_sort(q, j + 1, r);
}
```

#### 1.1.3 结果测试



多次测试运行后，结果均正确。

### 1.2 第K小数

#### 1.2.1 减治

本题与课外思考题Top\_k很类似。在本题中我们需要找到第 $k$ 小的数，可以使用上述快速排序的方法。已知我们在一次快速排序后，基准值 $x$ 左侧的数均小于 $x$ ，所以我们可以返回该基准值的位置 $location$ ，如果该位置小于 $k$ 则说明我们需要在右侧找第 $k - location$ 的值，如果该位置等于 $k$ ，则说明找到数字，如果该位置大于 $k$ 则说明我们需要在左侧找 $k$ 的值。由此，我们可以每次舍去一部分获得一个子问题，直到得

到k的值。

算法如下：

```
int Partition(int a[], int l, int r)
{
    int pivot = a[l];                // 取序列第一个元素为pivot
    int i = l - 1, j = r + 1;
    do
    {
        do { i++; } while (a[i] < pivot);
        do { j--; } while (a[j] > pivot);
        if (i < j) swap(a[i], a[j]);
    } while (i < j);

    return j;
}

void Top_k(int a[], int l, int r, int k)
{
    if (l >= r) return;
    int j = Partition(a, l, r);
    int count = j - l + 1;            // [l,j]区间共有多少数
    if (count == k)
    {
        return;
    }
    else if (count > k)
    {
        Top_k(a, l, j, k);            // 前半部分第k小的数
    }
    else
    {
        Top_k(a, j + 1, r, k - count); // 后半部分第k - i小的数
    }
}
```

## 1.2.2 冒泡排序

冒泡排序也是一种常见的排序方式，我们在这里可以采用一种优化方式来实现。即每次冒泡将最小的冒到最前面，只需要冒k次将第k小的数字排到前面即可，而不用全部排序。

以下为代码实现：

```
void bubblesort(int a[], int k, int n)
{
    for (int i = 0; i < k; i++)
    {
        for (int j = n - 2; j >= 0; j--)            // j控制每轮需要比较的次数
        {
            if (a[j + 1] < a[j])                    // 不满足升序要求，交换顺序
            {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}
```

```
}  
}  
}
```

### 1.2.3 堆排序

堆排序在思考题中已实现，不再赘述。

### 1.2.4 结果展示



## 1.3 算法思考

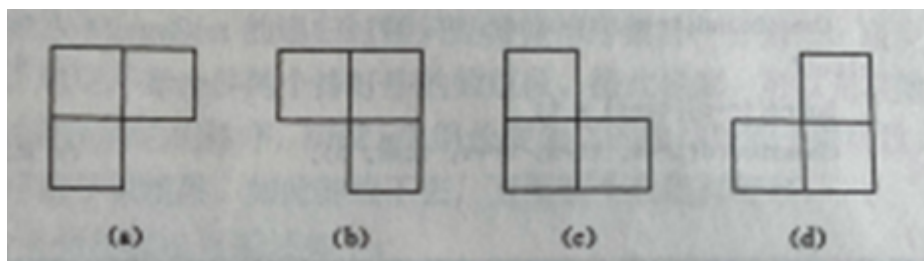
时间复杂度：

- 快速排序： $O(n\log n)$
- 减治法： $O(n)$
- 冒泡排序： $O(nk)$
- 堆排序： $O(n\log n)$

## 二、 棋盘覆盖问题

### 2.1 问题描述

在一个 $2k \times 2k$ 个方格组成的棋盘中，恰有一个方格与其它方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖；



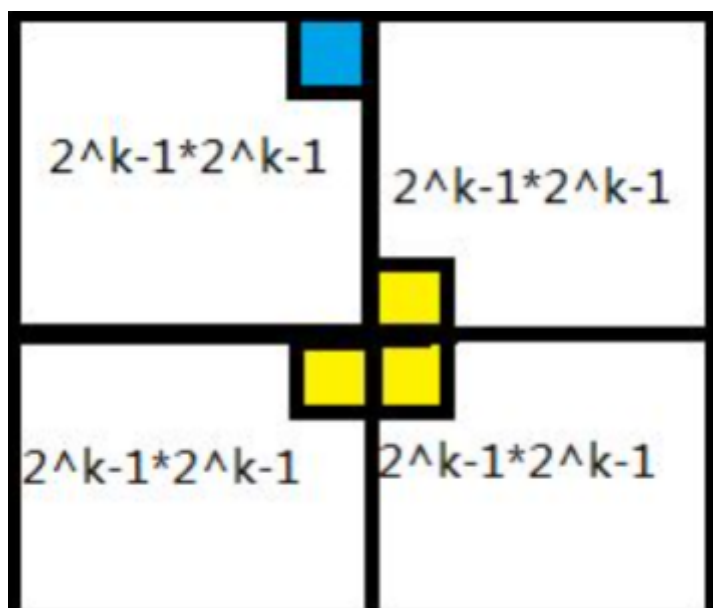
## 2.2 问题分析及算法思路

本题我们采用分治的方式求解。先对棋盘维数分类：

- 当  $k = 0$ ，为1维棋盘时，棋盘无L骨牌
- 当  $k > 0$ ，为  $2^k \times 2^k$  棋盘时，我们将其均分为4个子棋盘。而特殊方格在4个子棋盘之中，其余子棋盘无特殊方格。我们将一个L型骨牌放到汇合点。这样其余三个子棋盘也类似于大棋盘也有一个特殊方格。这样一来，我们形成了递归的思想。

由此我们进入一个棋盘时，首先要判断该特殊方格的位置，如果在左上，则进入新子棋盘，否则将右下角设置为特殊方格，进入该棋盘。类似的，如果在右上，则进入子棋盘，否则将左下角设为特殊方格，进入该棋盘。....

直到最后我们可以为所有的方格都设为特殊方格，只是我们在设置的时候有所标注，即起始的特殊方格为0，其余的特殊方格根据递归次数决定。



## 2.3 算法设计及代码

在算法中涉及以下变量，具体意义如下：

- tr: 当前棋盘左上角的行号
- tc: 当前棋盘左上角的列号
- dr: 当前特殊方格所在的行号
- dc: 当前特殊方格所在的列号
- size: 当前棋盘的大小  $2^k$

```
void chessBoard(int tr, int tc, int dr, int dc, int size)
{
    if (size == 1)           // 棋盘方格大小为1,说明递归到最里层
        return;
    int t = num++;           // 每次递增1
    int s = size / 2;        // 棋盘中间的行/列号
    // 检查特殊方块是否在左上角子棋盘中
    if (dr < tr + s && dc < tc + s)                // 在
        chessBoard(tr, tc, dr, dc, s);
    else                    // 不在,将该子棋盘右下角的方块视为特殊方块
```

```

{
    board[tr + s - 1][tc + s - 1] = t;
    chessBoard(tr, tc, tr + s - 1, tc + s - 1, s);
}
// 检查特殊方块是否在右上角子棋盘中
if (dr < tr + s && dc >= tc + s) // 在
    chessBoard(tr, tc + s, dr, dc, s);
else // 不在，将该子棋盘左下角的方块视为特殊方块
{
    board[tr + s - 1][tc + s] = t;
    chessBoard(tr, tc + s, tr + s - 1, tc + s, s);
}
// 检查特殊方块是否在左下角子棋盘中
if (dr >= tr + s && dc < tc + s) // 在
    chessBoard(tr + s, tc, dr, dc, s);
else // 不在，将该子棋盘右上角的方块视为特殊方块
{
    board[tr + s][tc + s - 1] = t;
    chessBoard(tr + s, tc, tr + s, tc + s - 1, s);
}
// 检查特殊方块是否在右下角子棋盘中
if (dr >= tr + s && dc >= tc + s) //在
    chessBoard(tr + s, tc + s, dr, dc, s);
else // 不在，将该子棋盘左上角的方块视为特殊方块
{
    board[tr + s][tc + s] = t;
    chessBoard(tr + s, tc + s, tr + s, tc + s, s);
}
}
}

```

## 2.4 结果展示

C# Microsoft Visual Studio 调试控制台

输入棋盘的size(大小必须是2的n次幂): 8  
 输入特殊方格位置的坐标: (3, 3)
 

3	3	4	4	8	8	9	9
3	2	2	4	8	7	7	9
5	2	0	6	10	10	7	11
5	5	6	6	1	10	11	11
13	13	14	1	1	18	19	19
13	12	14	14	18	18	17	19
15	12	12	16	20	17	17	21
15	15	16	16	20	20	21	21

相同数字形成的形状为L骨牌。

## 2.5 算法思考与讨论

分治即“分而治之”，一个规模很大的问题若要直接求解起来是非常困难的，将一个复杂的问题分解为若干个规模较小但是类似于原问题的子问题，子问题可以再分为更小的子问题，最终达到子问题可以简单的直接求解的目的，那么原问题的解即子问题的解的并集。分治算法可以缩小问题的规模，使得问题的求解变得十分容易。

# 实验二： 动态规划

## 一、计算矩阵连乘积

### 1.1 问题描述

在科学计算中经常要计算矩阵的乘积。矩阵A和B可乘的条件是矩阵A的列数等于矩阵B的行数。若A是一个 $p \times q$ 的矩阵，B是一个 $q \times r$ 的矩阵，则其乘积 $C=AB$ 是一个 $p \times r$ 的矩阵。由该公式知计算 $C=AB$ 总共需要 $pqr$ 次的数乘。其标准计算公式为：

$$C_{ij} = \sum_{k=1}^q A_{ik} B_{kj} \quad \text{其中 } 1 \leq i \leq p, 1 \leq j \leq r$$

现在的问题是，给定 $n$ 个矩阵 $\{A_1, A_2, \dots, A_n\}$ 。其中 $A_i$ 与 $A_{i+1}$ 是可乘的， $i=1, 2, \dots, n-1$ 。要求计算出这 $n$ 个矩阵的连乘积 $A_1 A_2 \dots A_n$ 。

递归公式：

$$m[i][j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{m[i][k] + m[k+1][j] + p_{i-1} p_k p_j\} & i < j \end{cases}$$

### 1.2 问题分析和算法思路

我们不仅要计算所有矩阵所需要的最小计算次数是多少，还要了解如何将矩阵结合才能得到最小的计算次数。因此，如何合并是一个重要问题，在这里使用的方法是逐次计算比较。比如从矩阵 $i$ 到矩阵 $j$ 的连乘，我们可以看作从 $i$ 到 $k$ ，再从 $k$ 到 $j$ 加上 $pqr$ 。在不同的 $k$ 值中选取最小的一个作为我们从 $i$ 到 $j$ 的最优解。从 $r$ 为2（两个连乘）到 $r$ 为 $n$ （ $n$ 个连乘），这样得到的 $m$ 矩阵记录了从 $i$ 到 $j$ 矩阵的最小计算次数。我们根据计算过程中记录的隔断点就可以得知最好的连乘顺序。

### 1.3 算法设计与代码

```
void MatrixChain(int n)
{
    int r, i, j, k;
    for (i = 0; i <= n; i++) // 初始化对角线
    {
        m[i][i] = 0;
    }
    for (r = 2; r <= n; r++) // r 个矩阵连乘
    {
        for (i = 1; i <= n - r + 1; i++) // 依次计算每r个矩阵相连乘的最优解情况
        {
            j = i + r - 1;
            m[i][j] = m[i][i] + m[i + 1][j] + p[i - 1] * p[i] * p[j];
        }
    }
}
```


```

        s[i][j] = i; // 分隔位置
        for (k = i + 1; k < j; k++) // 变换分隔位置，逐一测试
        {
            int t = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
            if (t < m[i][j]) // 如果变换后的位置更优，则替换原来的分隔方
            {
                m[i][j] = t;
                s[i][j] = k;
            }
        }
    }
}

void print(int i, int j) // 输出连乘顺序
{
    if (i == j)
    {
        cout << "p[" << i << "]";
        return;
    }
    cout << "(";
    print(i, s[i][j]);
    print(s[i][j] + 1, j);
    cout << ")";
}

```

## 1.4 结果展示

 Microsoft Visual Studio 调试控制台

```

请输入矩阵的数目：5
请输入各个矩阵的维度(相邻维度只需输入一个即可):20 15 30 56 25 10
最佳添加括号的方式为：(p[1] (p[2] (p[3] (p[4]p[5]))))
最小计算量的值为：38300
    
```

## 1.5 算法思考与讨论

矩阵的连乘积是动态规划中一个经典问题，用动态规划的方法得到的时间复杂度为 $O(n^3)$ ，在需要大规模的矩阵连乘时，选择最优的连乘顺序，可以大幅度减小工作量，提高效率。

# 二、防卫导弹

## 2.1 问题描述

一种新型的防卫导弹可截击多个攻击导弹。它可以向前飞行，也可以用很快的速度向下飞行，可以毫无损伤地截击进攻导弹，但不可以向后或向上飞行。但有一个缺点，尽管它发射时可以达到任意高度，但它只能截击比它上次截击导弹时所处高度低或者高度相同的导弹。现对这种新型防卫导弹进行测试，在每一次测试中，发射一系列的测试导弹（这些导弹发射的间隔时间固定，飞行速度相同），该防卫导弹所能获得的信息包括各进攻导弹的高度，以及它们发射次序。

现要求编一程序，求在每次测试中，该防卫导弹最多能截击的进攻导弹数量，一个导弹能被截击应满足下列两个条件之一：

- a) 它是该次测试中第一个被防卫导弹截击的导弹；
- b) 它是在上一次被截击导弹的发射后发射，且高度不大于上一次被截击导弹的高度的导弹。

输入数据：第一行是一个整数 $n$ ，以后的 $n$ 各有一个整数表示导弹的高度。

输出数据：截击导弹的最大数目。

## 2.2 问题分析与算法思路

$L[i]$ 为选择拦截第 $i$ 个导弹后从此开始最多能截击的导弹数目。

由于选择第 $i$ 枚导弹开始拦截，因此下一个需要拦截的导弹的高度要小于等于它的高度，所以 $L[i]$ 应该为从 $i+1$ 到 $n$ 的每一个 $j$ ，满足 $h[j] \leq h[i]$ 的 $j$ 中 $L[j]$ 的最大值（即从 $i$ 之后开始拦截都不如从 $i$ 开始拦截的多）。

如果到最后一个才拦截，那么即 $L[n-1] = 1$ ，只能拦截一个。往前判断，如果 $h[n-2]$ 的高度大于 $h[n-1]$ 且当前能拦截的最大值小于之前能拦截的最大值，说明如果我们现在改变拦截起点，可以在原基础上多拦截一个。按照这样的思路进行到起点，我们就可以知道最多可以拦截多少导弹了。这是一种动态规划的思想。

## 2.3 算法设计与代码

```
#include<iostream>
using namespace std;
int main()
{
    int i, j, n, max, h[100], l[100];
    cout << "请输入导弹数量" << endl;
    cin >> n;
    cout << "请输入导弹高度" << endl;
    for (i = 0; i < n; i++)
        cin >> h[i];
    l[n - 1] = 1;
    for (i = n - 2; i >= 0; i--)
    {
        max = 0;
        for (j = i + 1; j < n; j++)
            if (h[i] > h[j] && max < l[j])
                max = l[j];
        l[i] = max + 1;
    }
    cout << "最多可以拦截的导弹个数为: ";
    cout << l[0];
    return 0;
}
```



## 2.4 结果展示

```
C:\> Microsoft Visual Studio 调试控制台
请输入导弹数量
8
请输入导弹高度
50 45 48 60 55 54 53 40
最多可以拦截的导弹个数为： 3
```

## 2.5 算法思考与讨论

本题是动态规划中一个经典的最长上升子序列问题，时间复杂度为 $O(n^2)$ 。

# 三、皇宫看守

## 3.1 问题描述

太平王世子事件后，陆小凤成了皇上特聘的御前一品侍卫。皇宫以午门为起点，直到后宫嫔妃们的寝宫，呈一棵树的形状；某些宫殿间可以互相望见。大内保卫森严，三步一岗，五步一哨，每个宫殿都要有人全天候看守，在不同的宫殿安排看守所需的费用不同。可是陆小凤手上的经费不足，无论如何也没法在每个宫殿都安置留守侍卫。

请你编程计算帮助陆小凤布置侍卫，在看守全部宫殿的前提下，使得花费的经费最少。

输入数据：输入数据由文件名为input.txt的文本文件提供。输入文件中数据表示一棵树，描述如下：

第1行  $n$ ，表示树中结点的数目。

第2行至第 $n+1$ 行，每行描述每个宫殿结点信息，依次为：该宫殿结点标号 $i$  ( $0 < i \leq n$ )，在该宫殿安置侍卫所需的经费 $k$ ，该边的儿子数 $m$ ，接下来 $m$ 个数，分别是这个节点的 $m$ 个儿子的标号 $r_1, r_2, \dots, r_m$ 。

对于一个 $n$  ( $0 < n \leq 1500$ ) 个结点的树，结点标号在1到 $n$ 之间，且标号不重复。

输出数据：输出到output.txt文件中。输出文件仅包含一个数，为所求的最少的经费。

如右图的输入数据示例：↵

Sample Input↵

6↵

1 30 3 2 3 4↵

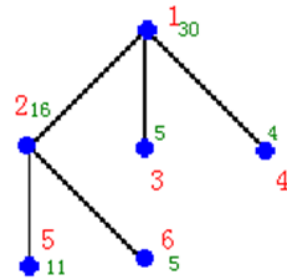
2 16 2 5 6↵

3 5 0↵

4 4 0↵

5 11 0↵

6 5 0↵



Sample Output↵

25↵

## 3.2 问题分析与算法思路

本题要求图中每个点都被观察到：

- 父节点 **放置** 哨兵，所有子节点都 **可放可不放** 哨兵
- 父节点 **不放** 哨兵，但是他至少有一个 **子节点** 放置哨兵，观察住了他
- 父节点 **不放** 哨兵，但 父节点 的 **父节点** 放置哨兵观察，则 **子节点** 可放可不放 哨兵

则每个节点有以下三种情况：

- 被父节点观察 (0)
- 被子节点观察 (1)
- 被自己来观察 (2)

状态计算：

- 在  $i$  上被父节点观察 (0)：

$$f_{i,0} = \sum_{i_{ch}} \min(f_{i_{ch},1}, f_{i_{ch},2}), \text{ 其中 } i_{ch} \in \text{ver} \mid \text{ver 是 } i \text{ 的子节点}$$

- 在  $i$  上被子节点观察 (1)：

$$f_{i,1} = \min \left( f_{i,2} + \sum_{\text{other } i_{ch}} \min(f_{\text{other } i_{ch}}, 1, f_{\text{other } i_{ch},2}) \right),$$

- 在  $i$  上被自己来观察 (2)：

$$f_{i, 2} = \sum_{h \in \text{children}(i)} \min(f_{h, 0}, f_{h, 1}, f_{h, 2}) + w_i,$$

我们先从根节点开始DFS，然后遍历当前节点的所有子节点，递归得到当前节点的 $f[u][0], f[u][1], f[u][2]$ 最后对根节点取被子节点观察和被自己观察的方案中代价最小的方案 $\min(f[\text{root}][1], f[\text{root}][2])$ 即可。

### 3.3 算法设计与代码

```
void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}

void dfs(int u)
{
    f[u][2] = w[u];

    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];          // 遍历所有子节点
        dfs(j);
        f[u][0] += min(f[j][1], f[j][2]);
        f[u][2] += min(min(f[j][0], f[j][1]), f[j][2]);
    }

    f[u][1] = 1e9;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        // f[u][0]为所有子节点的摆放方案代价之和，减去 min(f[j][1], f[j][2]) 即是除了j节点其余节点的代价之和
        f[u][1] = min(f[u][1], f[j][2] + f[u][0] - min(f[j][1], f[j][2]));
    }
}
```

### 3.4 结果展示

 Microsoft Visual Studio 调试控制台

```

6
1 30 3 2 3 4
2 16 2 5 6
3 5 0
4 4 0
5 11 0
6 5 0
25
    
```

### 3.5 算法思考与讨论

本题是树形DP的典型应用，这种DP方式涉及到了树中父节点、子节点之间的邻接关系，根据不同的条件进行分别讨论，确定状态表示和状态计算的策略，并运用树（图）的DFS/BFS算法求解。

## 实验三：贪心算法和随机算法

### 一、背包问题

#### 1.1 问题描述

有一个背包，背包容量是  $M=150$ 。有 7 个物品，物品可以分割成任意大小。

要求尽可能让装入背包中的物品总价值最大，但不能超过总容量。

物品	A	B	C	D	E	F	G
重量	35	30	60	50	40	10	25
价值	10	40	30	50	35	40	30

#### 1.2 问题分析与算法思路

这题是经典背包问题，需要使用贪心算法求解。

我们按照每个物品单位重量的价值进行排序，然后从最大单位重量价值的物品开始装入背包，当剩余背包容量大于等于该物品的重量时，直接放入，并将该物品的价值加入总价值；否则，获取剩余比例，将该比例的物品放入背包中（题目允许切割），并将该比例的价值加入总价值，以此得到的就是总价值最大的方案。

容易证明该贪心策略的正确性（主观感受也可得到）。

#### 1.3 算法设计与代码

```
struct Obj
{
    int id;           // 物品序号
    int w;           // w为各物品的重量
    int v;           // v为各物品的价值
    float unit;      // 单位重量的价值
    bool operator< (const Obj& w)const
    {
        return unit < w.unit;
    }
}obj[N];

void FindMaxValue(int n, int m)
{
    float value = 0;
    sort(obj, obj + n);           // 按照单位重量的价值对物品进行升序排序
    for (int i = n - 1; i >= 0; i--)
    {
        if (m - obj[i].w >= 0)           // 存在剩余容量
        {
            m -= obj[i].w;               // 去掉这部分的背包容量
            value += obj[i].v;           // 加入这部分的价值
        }
    }
}
```

```

        cout << "装入整个第" << l[obj[i].id] << "个物品" << endl;
        if (m == 0) break;
    }
    else
    {
        float ratio = (float) m / obj[i].w;
        cout << "装入" << ratio * 100 << "%第" << l[obj[i].id] << "个物品" <<
endl;

        value += ratio * obj[i].v;
        break;
    }
}
cout << "装入背包中的物品的总价值最大为" << value << endl;
}

```

## 1.4 结果展示

```

请输入物品数和背包容量:7 150
请输入各个物品的重量和价值:
35 10
30 40
60 30
50 50
40 35
10 40
25 30
装入整个第F个物品
装入整个第B个物品
装入整个第G个物品
装入整个第D个物品
装入87.5%第E个物品
装入背包中的物品的总价值最大为190.625

```

## 1.5 算法思考与讨论

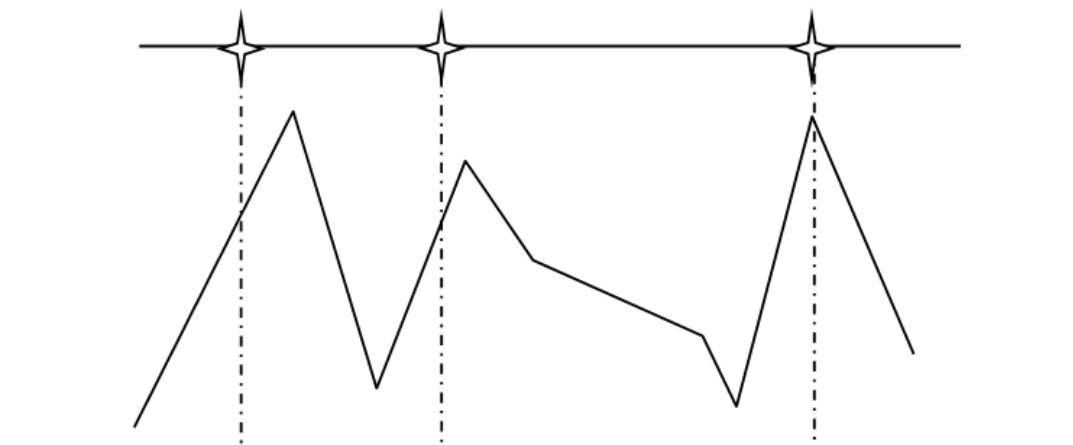
背包问题有多种，分为01背包，完全背包，多重背包，分组背包等类型，主要通过动态规划去求解，但是这里的物品可以分割成任意大小，因此本题采用贪心策略。

## 二、照亮的山景

## 2.1 问题描述

在一片山的上空，高度为T处有N个处于不同位置的灯泡，如图。如果山的边界上某一点于某灯i的连线不经过山的其它点，我们称灯i可以照亮该点。开尽量少的灯，使得整个山景都被照亮。山被表示成有m个转折点的折线。

提示：照亮整个山景相当于照亮每一个转折点。



## 2.2 问题分析与算法思路

一座山要想被照亮，那么把这座山的两侧分别延长，与灯所在的高度交到两点，在这个区间内如果有灯，就可以照亮这座山，如果没有，就必须在该区间的两侧各有一盏灯。我们把每座山的区间放到一个集合中，遍历所有的灯，每次贪心的寻找覆盖区间最多的灯，铜山将已经照亮的山移除集合，标记灯为已使用，直到山集合为空，所求的灯的数量就是最小的灯的数量。

## 2.3 算法设计与代码

首先用斜率的计算公式求每座山的区间。

```
for(int i=2;i<x.length;i+=2){//每座山峰山顶之间隔一个点
    x1 = x[i-2];
    x2=x[i-1];
    x3=x[i];
    y1 = y[i-2];
    y2=y[i-1];
    y3=y[i];
    //double k1 = (y2-y1)/(x2-x1+0.0),k2=(y3-y2)/(x3-x2+0.0);
    //(x,height),
    //(height-y2)/(xi-x2)=ki
    sections[i-2][0] = (height*(x2-x1) +x1*y2 -x2*y1)/(y2-y1);
    sections[i-2][1] = (height*(x2-x3) +x3*y2 -x2*y3)/(y2-y3);
}
```

max为最大的覆盖区间个数，count记录尚未点亮的山的个数

ans为灯的覆盖区间。

```

int max = 0, index = 0;
int count = sections.length;
ArrayList<Integer> ans = new ArrayList<>();

```

灯的判断条件

```
lights[i] > sections[j][1] && lights[i] < sections[j][0]
```

总代码：

```

public void solve(int[] x, int[] y, int height, int[] lights) {
    int[][] sections = new int[x.length-2][2];
    int x1, x2, x3, y1, y2, y3;
    //只要构成山峰，数组一定是奇数
    for(int i=2; i<x.length; i+=2) { //每座山峰山顶之间隔一个点
        x1 = x[i-2];
        x2 = x[i-1];
        x3 = x[i];
        y1 = y[i-2];
        y2 = y[i-1];
        y3 = y[i];
        //double k1 = (y2-y1)/(x2-x1+0.0), k2 = (y3-y2)/(x3-x2+0.0);
        //((x,height),
        //(height-y2)/(xi-x2)=ki
        sections[i-2][0] = (height*(x2-x1) + x1*y2 - x2*y1)/(y2-y1);
        sections[i-2][1] = (height*(x2-x3) + x3*y2 - x2*y3)/(y2-y3);
    }
    int max = 0, index = 0;
    int count = sections.length;
    ArrayList<Integer> ans = new ArrayList<>();
    while(count > 0) {
        ArrayList<Integer> tmp = new ArrayList<>(); //存选出的灯所关联的区间
        for(int i=0; i<lights.length; i++) {
            ArrayList<Integer> related = new ArrayList<>(); //存储当前灯关联的区间
            //int num = 0;
            for(int j=0; j<sections.length; j++) {
                if(lights[i] > sections[j][1] && lights[i] < sections[j][0])
                {
                    //num++;
                    related.add(j);
                }
            }
            int num = related.size();
            if(num > max) {
                max = num;
                index = i;
                tmp = related;
            }
        }
        //此时index是覆盖区间最多的灯的下标
        count -= tmp.size();
        System.out.println(lights[index]);
        Iterator itr = tmp.iterator();
    }
}

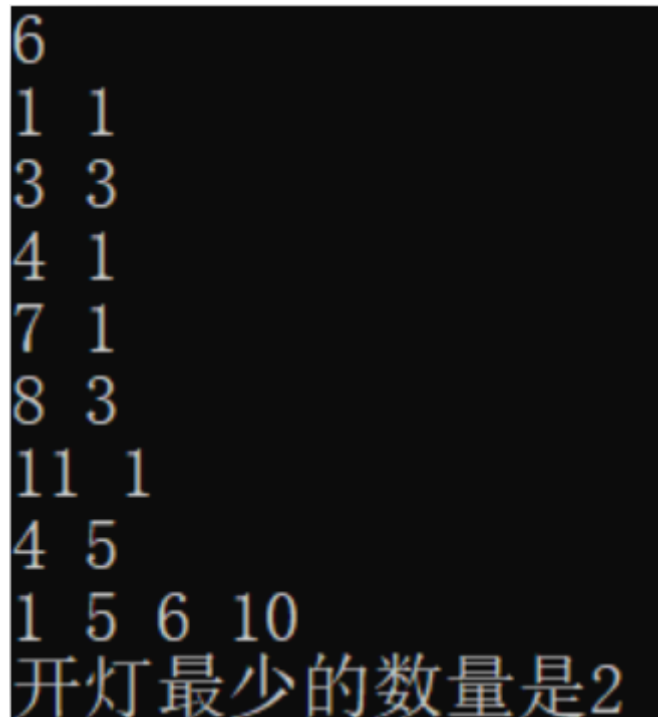
```

```

        while(itr.hasNext()){
            Integer t = (Integer) itr.next();
            sections[t][0] = sections[t][1] = -1;
        }
    }
}

```

## 2.4 结果展示



```

6
1 1
3 3
4 1
7 1
8 3
11 1
4 5
1 5 6 10
开灯最少的数量是2

```

## 2.5 算法思考与讨论

我通过求解每个顶点能被灯照到的左端点和右端点，然后运用贪心策略依次计算区间内每个数出现的次数，找出区间中出现次数最多的数，开相应序号的灯，从而求解开灯最少的数量，算法的时间复杂度是  $O(n^2 \cdot m)$

# 三、搬桌子问题

## 3.1 问题描述

某教学大楼一层有  $n$  个教室，从左到右依次编号为 1、2、...、 $n$ 。现在要把一些课桌从某些教室搬到另外一些教室，每张桌子都是从编号较小的教室搬到编号较大的教室，每一趟，都是从左到右走，搬完一张课桌后，可以继续从当前位置或往右走搬另一张桌子。

输入数据：先输入  $n$ 、 $m$ ，然后紧接着  $m$  行输入这  $m$  张要搬课桌的起始教室和目标教室。

输出数据：最少需要跑几趟。



## 3.2 问题分析与算法思路

本题只需要将任务按照起始教师的编号排序，然后从当前已经遍历到的教师开始找到后面最近的教师开始任务；当当前后面教师没有可以开始任务的教师时，折返。

易知贪心策略是正确的：如果不按照此策略，我们可以改变任务执行的顺序，使得跑的趟数不发生改变，则贪心策略是最优的。

## 3.3 算法设计与代码

```
struct Move {
    int start;
    int end;
    bool use;
    bool operator< (const Move& w) const
    {
        return start < w.start;
    }
}mov[N];

int runnum(int n, int m)
{
    sort(mov, mov + m);           // 按照任务起始教室的编号排序
    int res = 0, num = 0, work = 0; // res为趟数
    while (work < m)
    {
        int num = 0;               // num为当前到的教室编号
        for (int i = 0; i < m; i++)
        {
            if (mov[i].use == false && mov[i].start >= num)
            {
                mov[i].use = true;
                work++;
                num = mov[i].end;
                if (num == n) break;
            }
        }
        res++;
    }
    return res;
}
```

## 3.4 结果展示

```
10 5
1 3
3 9
4 6
6 10
7 8
3
```

### 3.5 算法思考与讨论

本题算是贪心算法的一种典型应用，当每次从当前已经遍历到的教师开始找到后面最近的教室开始任务时，可以让一次遍历到的教室属于任务范围内的比例最大，从而使得整个算法性能达到最优。