算法实验

1. 分治问题

1.1 快排与 topk

题目分析

1. 快速排序：基于分治法，通过将数组划分为较小的子数组，然后递归地对子数组进行排序，最终将整个数组排序。基本步骤是先选择数组中的一个元素作为比较的枢轴，根据比较元素将数组划分为两个子数组，使得左子数组中的元素小于等于枢轴，右子数组中的元素大于枢轴。之后对左右子数组递归调用quickSort函数，最后合并左子数组、枢轴和右子数组，得到排序后的数组。
    - 平均时间复杂度为 O(n log n)，其中 n 是待排序数组的长度。在最坏情况下，时间复杂度为 $O(n^2)$。
    - 空间复杂度为 O(log n)，由于有递归栈的空间消耗。在最坏情况下，空间复杂度为 O(n)，即如果每次选择的枢轴都是最小或最大元素，栈的深度将达到O(n)的量级。
2. topK 问题：
    - 可以使用快速选择，通过每次选择一个枢轴，并根据枢轴的位置将数组划分为两部分，然后根据 k 的位置递归地在一个部分中查找。如果枢轴的位置等于 k-1，则找到了最小的 k 个元素。
        - 时间复杂度的期望值为 O(n)，在最坏情况下为 O(n^2)
    - 或者使用优先队列（构造一个大小为 n 的堆或一个大小为 k 的堆）

代码实现

```
ElementType Median3( ElementType A[], int Left, int Right )
{
  int Center = (Left+Right) / 2;
  if ( A[Left] > A[Center] )
    Swap( &A[Left], &A[Center] );
  if ( A[Left] > A[Right] )
    Swap( &A[Left], &A[Right] );
  if ( A[Center] > A[Right] )
    Swap( &A[Center], &A[Right] );
  /* 此时A[Left] <= A[Center] <= A[Right] */
  Swap( &A[Center], &A[Right-1] ); /* 将基准Pivot藏到右边*/
  /* 只需要考虑A[Left+1] … A[Right-2] */
  return  A[Right-1];  /* 返回基准Pivot */
}
```

```
void Qsort( ElementType A[], int Left, int Right )
{ /* 核心递归函数 */
    int Pivot, Cutoff, Low, High;

    if ( Cutoff <= Right-Left ) { /* 如果序列元素充分多，进入快排 */
        Pivot = Median3( A, Left, Right ); /* 选基准 */
        Low = Left; High = Right-1;
        while (1) { /*将序列中比基准小的移到基准左边，大的移到右边*/
            while ( A[++Low] < Pivot ) ;
            while ( A[--High] > Pivot ) ;
            if ( Low < High ) Swap( &A[Low], &A[High] );
            else break;
        }
        Swap( &A[Low], &A[Right-1] );   /* 将基准换到正确的位置 */
        Qsort( A, Left, Low-1 );    /* 递归解决左边 */
        Qsort( A, Low+1, Right );   /* 递归解决右边 */
    }
    else InsertionSort( A+Left, Right-Left+1 ); /* 元素太少，用简单排序 */
}

void QuickSort( ElementType A[], int N )
{ /* 统一接口 */
    Qsort( A, 0, N-1 );
}


#include<iostream> #include<vector> #include<queue> #include"QuickSort.h"
#include"topK.h" using namespace std; void testTopk() {
vector<int> arr = { 12, 21, 1, 22, 342, 23, 452, 11, 34, 45, 4554, 9, 809,
29, 89, 14, 66, 33 };
priority_queue<int,vector<int>,greater<int>>  pq; for (auto it = arr.begin(); it != arr.end(); it++)
{
pq.push(*it);
}
int res = 0; int k = 3;
cout << "请输入需要的 k ： ";
cin >> k;
```

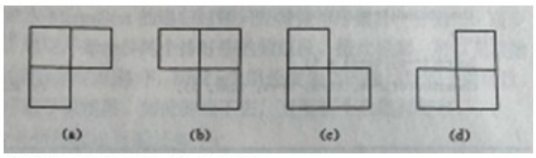
```
for (int i = 0; i < k - 1; ++i) { pq.pop();
}
cout <<"第 "<<k<<" 小的数为："<< pq.top();


}

int main() {
testTopk();
}
```

效果

1.2 棋盘覆盖问题

题目描述

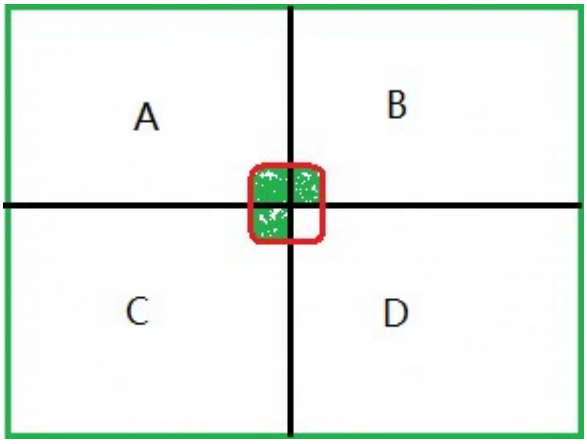
在一个 $2^k \times 2^k$ 个方格组成的棋盘中，恰有一个方格与其它方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的 4 种不同形态的 L 型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且



任何 2 个 L 型骨牌不得重叠覆盖；↵

题目分析

棋盘的大小为$2^k \times 2^k$，如果通过递归的方法来解决这个问题，如图，将棋盘分为A、B、C、D 4个大小为$2^{(k-1)} \times 2^{(k-1)}$子棋盘，特殊方格只可能位于A、B、C、D中的一个子棋盘中。



1. 如果特殊方格位于A中，可以将B的左下角、C的右上角、D的左上角的方格作为特殊方格，用递归 调用直接处理（特殊方格位于 B, C, D 则同理）
2. 对于位于棋盘中央的 四个方格特殊处理，不含有特殊方格的3个子棋盘的角合并起来可以用一个L 型的骨牌覆盖，而含有特殊方格的子棋盘的靠近中央的角则会有递归调用填满
3. 递归的 base case：当棋盘大小只有 2×2 时可以直接处理，除去特殊方格，其余 3 个方格使用恰当的
   L 型骨牌直接覆盖即可

总体上，每次递归将棋盘分割成4个子棋盘，并根据特殊方格的位置选择合适的L型骨牌。最终可以覆盖 整个棋盘上除特方格以外的所有方格。

代码实现

```
private void cheCover(int x, int y, int dx, int dy, int size){
        //递归出口
        if(size == 1) {
                return;
        }

        //不同类型的三格骨牌
        int i = this.type++;
        //将棋盘分割为四个象限
        size = size/2;

        //处理左上角
        if(dx < x+size && dy < y+size) {
                //特殊点若在左上角
                cheCover(x, y, dx, dy, size);
        }else {
                //特殊点若不在左上角，对该象限右下角进行填充
                this.board[x+size-1][y+size-1] = i;
                cheCover(x, y, x+size-1, y+size-1, size);
        }

        //处理右上角
        if(dx < x+size && dy >= y+size) {
                //特殊点若在右上角
                cheCover(x, y+size, dx, dy, size);
        }else {
                //特殊点若不在右上角，对该象限左下角进行填充
                this.board[x+size-1][y+size] = i;
                cheCover(x, y+size, x+size-1, y+size, size);
        }

        //处理左下角
        if(dx >= x+size && dy < y+size) {
                //特殊点若在左下角
                cheCover(x+size, y, dx, dy, size);
        }else {
```

```
            //特殊点若不在左下角，对该象限右下角进行填充
            this.board[x+size][y+size-1] = i;
            cheCover(x+size, y, x+size, y+size-1, size);
        }

        //处理右下角
        if(dx >= x+size && dy >= y+size) {
            //特殊点若在右下角
            cheCover(x+size, y+size, dx, dy, size);
        }else {
            //特殊点若不在右下角，对该象限左下角进行填充
            this.board[x+size][y+size] = i;
            cheCover(x+size, y+size, x+size, y+size, size);
        }
}
```

实现效果

```
3       3       4       4       8       8       9       9
3       2       2       4       -1      8       7       9
5       2       6       6       10      7       7       11
5       5       6       1       10      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
```

2. 动态规划问题

2.1 计算矩阵连乘积

题目描述

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×q 的矩阵，B 是一个 q×r 的矩阵，则其乘积 C=AB 是一个 p×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：

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

现在的问题是，给定 n 个矩阵{A1,A2,…,An}。其中 Ai 与 Ai+1 是可乘的，i=1,2,…,n-1。要求计算出这 n 个矩阵的连乘积 A1A2…An。

递归公式：$m[i][j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} \{m[i][k] + m[k+1][j] + p_{i-1} p_k p_j\} & i < j \end{cases}$

题目分析

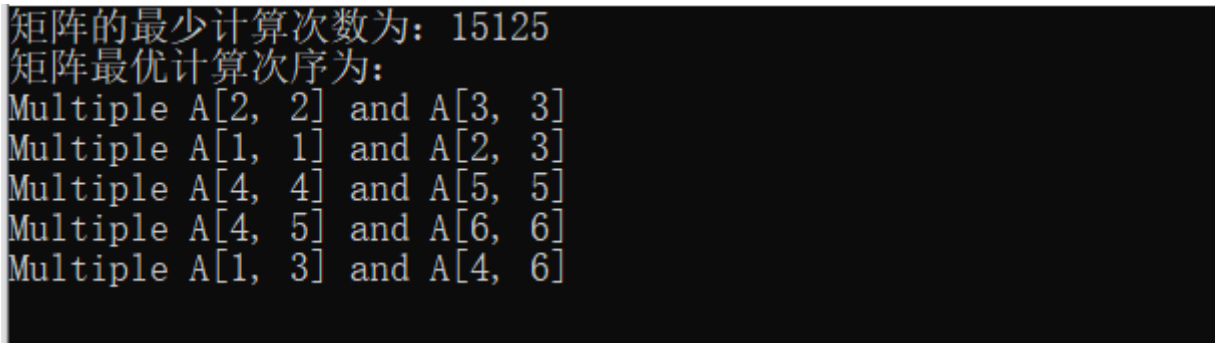
1. 子问题划分：
    - 需要将原问题划分为多个子问题，以便通过求解子问题来解决原问题。
    - 假设原问题的矩阵序列为 A1, A2, ..., An。
    - 可以选择一个位置 k (1 ≤ k ≤ n-1)，将矩阵序列划分为两部分：A1A2...Ak 和
    - Ak+1Ak+2...An。则原问题的最优解可以通过求解子问题 A1A2...Ak 和 Ak+1Ak+2...An 的最优解得到。
2. 定义状态和状态转移方程：
    - 定义一个状态数组 dp，其中 dp[i][j] 表示计算矩阵$A_i A_{i+1} \dots A_j$的最小代价。
    - 状态转移方程为：
        - 当 i = j 时，dp[i][j] = 0，表示单个矩阵的乘法代价为 0。
        - 当 i < j 时，dp[i][j] = min{dp[i][k] + dp[k+1][j] + $p_{i-1} p_k p_j$}，其中 i ≤ k < j，表示划分位置为 k 时的最小代价。
3. 计算最优解：
    - 使用动态规划的方法，按照子问题划分和状态转移方程，从小规模的子问题开始，逐步计算出大规模问题的最优解。
    - 根据状态转移方程，可以使用两层循环来计算 dp 数组的每个元素。
4. 求解结果：
    - 最终的最优解为 dp[1][n]，即计算矩阵 A1A2...An 的最小代价。

代码实现

```
#include<iostream>
#include<vector>
using namespace std;

const int L = 7;

int matrixMultiple(int n, vector< vector<int> > &m, vector< vector<int> > &s, int p[]){

   for(int i = 1; i < n; i++){
      //对角线设为0，没有自己和自己乘
      m[i][i] = 0;
   }

   for(int r = 2; r <= n; r++){ //r为规模

      for(int i = 1; i <= n-r+1; i++){  //i:首矩阵编号

         int j = i + r - 1;  //尾矩阵编号
         m[i][j] = m[i+1][j] + p[i-1]*p[i]*p[j];    //将链ij划分为A(i) *(A[i+1 : j])
         s[i][j] = i;

         for(int k = i + 1; k < j; k++){  //k:断开的位置
            //将链ij划分为(A[i:k] + A[k+1 : j])
            int temp = m[i][k] + m[k + 1][j] + p[i-1]*p[k]*p[j];
            if(temp < m[i][j]){
               m[i][j] = temp;
               s[i][j] = k;
            }
         }
      }
   }
   return m[1][L-1];
}

void traceBack(int i, int j, vector< vector<int> > &s){
```

```
    if(i == j)
        return ;

    traceBack(i, s[i][j], s);
    traceBack(s[i][j] + 1, j, s);
    cout << "Multiple A[" << i << ", " << s[i][j] << "]";
    cout << " and A[" << (s[i][j] + 1) << ", " << j << "]" << endl;
}

int main(){

//A1:30*35 A2:35*15 A3:15*5 A4:5*10 A5:10*20 A6:20*25
//p[0-6]={30,35,15,5,10,20,25}

        int p[L]={30, 35, 15, 5, 10, 20, 25};

    vector< vector<int> > m(L, vector<int>(L));
    vector< vector<int> > s(L, vector<int>(L));

        cout<<"矩阵的最少计算次数为："<<matrixMultiple(6,m,s,p)<<endl;
        cout<<"矩阵最优计算次序为："<<endl;
        traceBack(1,6,s);  //输出A[1:6]的最优计算此序
        return 0;
}
```

效果分析



2.2 防卫导弹

题目描述

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。

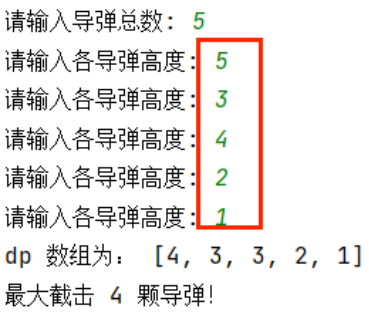
题目分析

题目使用动态规划解决，可以定义一个 dp 数组，其中 `dp[i]` 表示在如果拦截第 `i` 颗导弹则之后能够截击的最大导弹数量。从后向前进行一次遍历，对于每一颗导弹，扫描在它之后发射的导弹，找到之后发射的导弹的最大拦截数目，在此数目上 + 1 得到拦截当前导弹时的最大拦截数目。

代码实现

```
def missile_interception(n, heights):
    dp = [1] * n
    dp[n-1] = 1
    for i in range(n-2, -1, -1):
        max_def = 0
        for j in range(i+1,n):
            if heights[i] > heights[j] and max_def < d
                max_def = dp[j]
        dp[i] = max_def + 1
    print("dp 数组为：",dp)
    return max(dp)

n = int(input("请输入导弹总数："))
heights = []
for _ in range(n):
    height = int(input("请输入各导弹高度："))
    heights.append(height)

max_def = missile_interception(n, heights)

print("最大截击", max_def,"颗导弹！")
```

效果


请输入导弹总数： 5
请输入各导弹高度： 5
请输入各导弹高度： 3
请输入各导弹高度： 4
请输入各导弹高度： 2
请输入各导弹高度： 1
dp 数组为： [4, 3, 3, 2, 1]
最大截击 4 颗导弹!

2.3 皇宫看守

题目描述

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 intput.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<=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 r1，r2，...，rm。

对于一个 n（0 < n <= 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。

题目分析

给定一个树形结构的宫殿，题目要求相当于要求解结点带权的最小支配集

题目可以使用动态规划求解，大致想法如下：

1. 定义状态：设$f[i][j]$表示在以节点 i 为根的子树中，安排侍卫的最小经费，其中 i 表示节点的编号，j 表示节点 i 是否安排了侍卫。j 的取值有三种情况：j = 0 表示节点 i 没有安排侍卫，j = 1 表示节点 i 安排了侍卫，j = 2 表示节点 i 的某个子节点安排了侍卫。
2. 确定状态转移方程：
   - 当 j = 0 时，表示节点 i 没有安排侍卫。则节点 i 的所有子节点可以安排侍卫也可以不安排，选择经费更小的状态。状态转移方程为：$f[i][0] = \Sigma min(f[s][1], f[s][2])$，其中 s 为节点 i 的子节点。
   - 当 j = 1 时，表示节点 i 安排了侍卫。则节点 i 的所有子节点可以按照任意方式安排侍卫，选择经费更小的状态。状态转移方程为：$f[i][1] = w[i] + \Sigma min(f[s][0], f[s][1], f[s][2])$，其中 w[i] 为节点 i 安排侍卫的经费，s 为节点 i 的子节点。
   - 当 j = 2 时，表示节点 i 的某个子节点安排了侍卫。则可以枚举哪个子节点安排了侍卫，其他子节点可以按照任意方式安排侍卫，选择经费更小的状态。状态转移方程为：$f[i][2] = min f[s][1] + \Sigma min(f[t][1], f[t][2])$，其中 s 为节点 i 的某个子节点，t 为节点 i 的其他子节点。
3. 确定初始状态：对于叶子节点（没有子节点的节点），初始状态为 $f[i][0] = 0$，$f[i][1] = w[i]$，$f[i][2] = INF$（表示不可行）。
4. 采用递归+备忘录的方式计算，适合规模较小的问题
5. 最终的答案是$f[root][1]$和$f[root][2]$中的较小值，其中root为根节点。

代码实现：

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1510, INF = 0x3f3f3f3f;
int n;
int h[N], e[N], ne[N], w[N], idx;
bool st[N];
int f[N][3];
/*
f[i][0]表示i节点能被他的父节点看到的情况下所需要的士兵数
f[i][1]表示i节点能被他的子节点看到的情况下所需要的士兵数
f[i][2]表示i节点有兵的情况下所需要的士兵数
*/
void add(int a,int b){
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx ++ ;
}
void dfs(int u){
    f[u][1] = INF;
    f[u][2] = w[u];
    int sum = 0;
    for(int i = h[u]; i != -1; i = ne[i]){
        int j = e[i];
        dfs(j);
        f[u][0] += min(f[j][2], f[j][1]);
        //u点能被看到的状态是由 u的子节点j有守卫/j能被j的子节点看到 的状态转移而来
        f[u][2] += min(min(f[j][0], f[j][2]), f[j][1]);
        //u点有守卫是由 u的子节点j有守卫/j能被j的子节点看到/j能被j的父节点看到 的状态转移而来
        sum += min(f[j][2], f[j][1]);//记录u节点被他至少一个子节点看到的情况
    }
    for(int i = h[u]; i != -1; i = ne[i]){
        int j = e[i];
        f[u][1] = min(f[u][1], sum - min(f[j][1], f[j][2]) + f[j][2]);//遍历每一个子节点有士兵的情况，找出其中最小值
```

```
    }
}
int main()
{
    cin>>n;
    memset(h, -1, sizeof h);
    for(int i = 1; i <= n; i ++ ){
        int id, k, cnt;
        cin>>id>>k>>cnt;
        w[id] = k;
        for(int j = 0; j < cnt; j ++ ){
            int ver;
            cin>>ver;
            add(id, ver);
            st[ver] = true;
        }
    }

    int root = 1;//节点的下标从1开始，所以应该从1节点进行遍历
    while(st[root]) root ++ ;
    dfs(root);

    cout<<min(f[root][1], f[root][2])<<endl;
    return 0;

}
```

经OJ评判结果正确。

3.贪心问题

3.1 背包问题

题目描述

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

题目分析

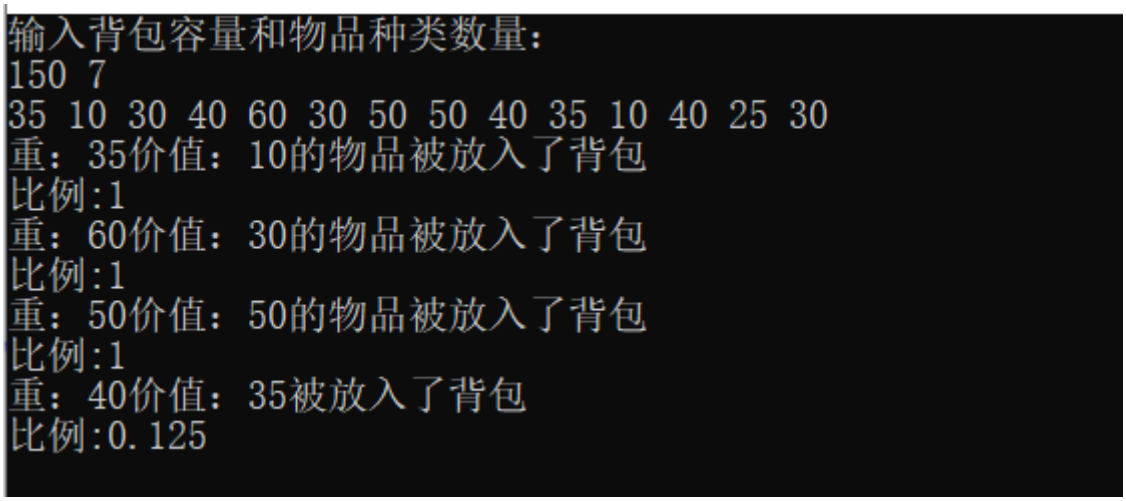
这是连续背包问题，可以使用贪心算法来解决。贪心策略为优先选择单位重量价值最高的物品放入背包，当容量不够是进行切分，直到背包容量用完

代码实现

```
#include <iostream>
#include <algorithm>
using namespace std;
struct bag{
    int weight;
    int value;
    float bi;
    float bili;
}bags[100];
bool compare(const bag &bag1,const bag &bag2);
int main()
{
    int sum=0,n;
    float M;
    int j=0;
    cout<<"输入背包容量和物品种类数量："<<endl;
    cin>>M>>n;
    for(int i=0;i<n;i++){
        cin>>bags[i].weight>>bags[i].value;
        bags[i].bi=bags[i].weight/bags[i].value;
    }
    for(int i=0;i<n;i++){
        bags[i].bili=0;
    }
    sort(bags,bags+n,compare);
   for(j=0;j<n;j++){
        if(bags[j].weight<=M){
            bags[j].bili=1;
            sum+=bags[j].weight;
            M-=bags[j].weight;
            cout<<"重："<<bags[j].weight<<"价值："<<bags[j].value<<"的物品被放入了背包
"<<endl<<"比例:"<<bags[j].bili<<endl;
        }
        else break;
```

```
    }
    if(j<n){
     bags[j].bili=M/bags[j].weight;
     sum+=bags[j].bili*bags[j].weight;
     cout<<"重：  "<<bags[j].weight<<"价值：  "<<bags[j].value<<"被放入了背包"<<endl<<"
比例:"<<bags[j].bili<<endl;
    }


    return 0;
}
bool compare(const bag &bag1,const bag &bag2){
return  bag1.bi>bag2.bi;
}
```

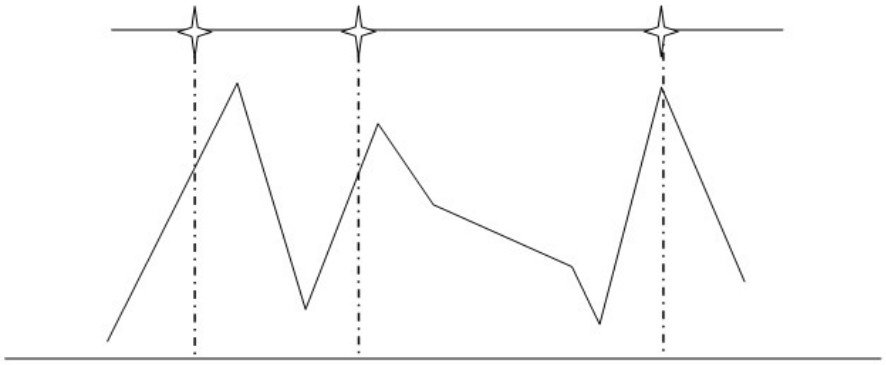
效果



3.2 照亮的山景

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。↵



题目分析

1. 问题可以抽象为给定没给转折点的横纵坐标，表示每座山，选择合适位置的灯将其点亮，使之能点亮所有的山，并使需要点亮的灯最少
2. 如果把一座山的两边分别延长，会与灯所在的高度交于两个点，两个点的横坐标所构成的区间内的所有的灯就可以把整座山照亮
3. 如果采用贪心策略，优先点亮覆盖区间数目最多的灯，直到所有山景被点亮，或者出现还有山景未点亮但是所有尚未被点亮灯都不处在任何一个区间内（即没有一种方案使得能点亮所有的山景）

代码实现

```
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;
        }
    }
```

效果

经OJ评测，正确

3.3 搬桌子问题

题目描述

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。↵

题目分析

反证法可以证明问题满足满足最优子结构的性质，适合使用贪心法。贪心策略：把课桌按起点从小到大排序，每次都是搬离当前位置最近的课桌。

代码实现

```
public int solve(int m,int n,int[][] works){
    int ans = 0;
    Arrays.sort(works, new Comparator<int[]>() {
      @Override
      public int compare(int[] o1, int[] o2) {
        return o1[1] < o2[1] ? -1 : 1;
      }
    });
//    System.out.println(Arrays.deepToString(works));
    //按目的地升序排序
    int start = 0,end = 0;//这一趟开始的任务和结束的任务编号
    //int startTime=0,endTime=0;
    int count = m,perMov=0;//循环条件count，每次完成的任务次数perMov
    while(count > 0){
      System.out.println(Arrays.deepToString(works));
      start = end = 0;//重新找
      perMov = 0;
      while(start < m){
        while(start <m && works[start][0] <= 0){
          start++;
        }
```

```
            //找到第一个没有执行的任务
            works[start][0]=-1;
            //执行过的任务开始时间为-1
            while (end < m && works[end][0] < works[start][1]){
                end++;
            }
            //找到第一个可以接在后面执行的任务,更新位置
            start = end;
            perMov++;
        }
        count-=perMov;
        ans++;
    }
    return ans;
}
```

效果

```
输入几个教室
10
输入几个桌子要搬
5
输入数据(01)
1 3
输入数据(11)
3 9
输入数据(21)
4 6
输入数据(31)
6 10
输入数据(41)
7 8
[[1, 3], [4, 6], [7, 8], [3, 9], [6, 10]]
[[-1, 3], [-1, 6], [-1, 8], [3, 9], [6, 10]]
[[-1, 3], [-1, 6], [-1, 8], [-1, 9], [6, 10]]
至少需要 3次
```