实验五：贪心算法和随机算法

背包问题

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

问题分析及算法思路

首先计算每种物品单位重量的价值，然后依照贪心策略，将**尽可能多的单位重量价值最高的物品**装入背包。将某种物品全部装入背包，背包内的物品总重量未超过C，则选择单位重量价值次高的物品并尽可能地多装入背包

使用贪心算法解决背包问题的思路如下：

1. 计算每个物品的单位价值：对于每个物品，计算其单位重量的价值（即价值除以重量），并将其降序排序。
2. 初始化结果变量：创建一个结果变量，用于记录当前已选择的物品的总重量和总价值，初始值为0。
3. 遍历物品：按照单位价值降序的顺序，依次遍历每个物品。
4. 判断是否可以放入背包：对于当前遍历到的物品，判断是否可以放入背包中。如果当前物品的重量小于等于背包剩余容量，则将该物品放入背包，并将其重量和价值加到结果变量中。否则，跳过该物品。
5. 返回结果：遍历完所有物品后，返回结果变量中记录的总重量和总价值。

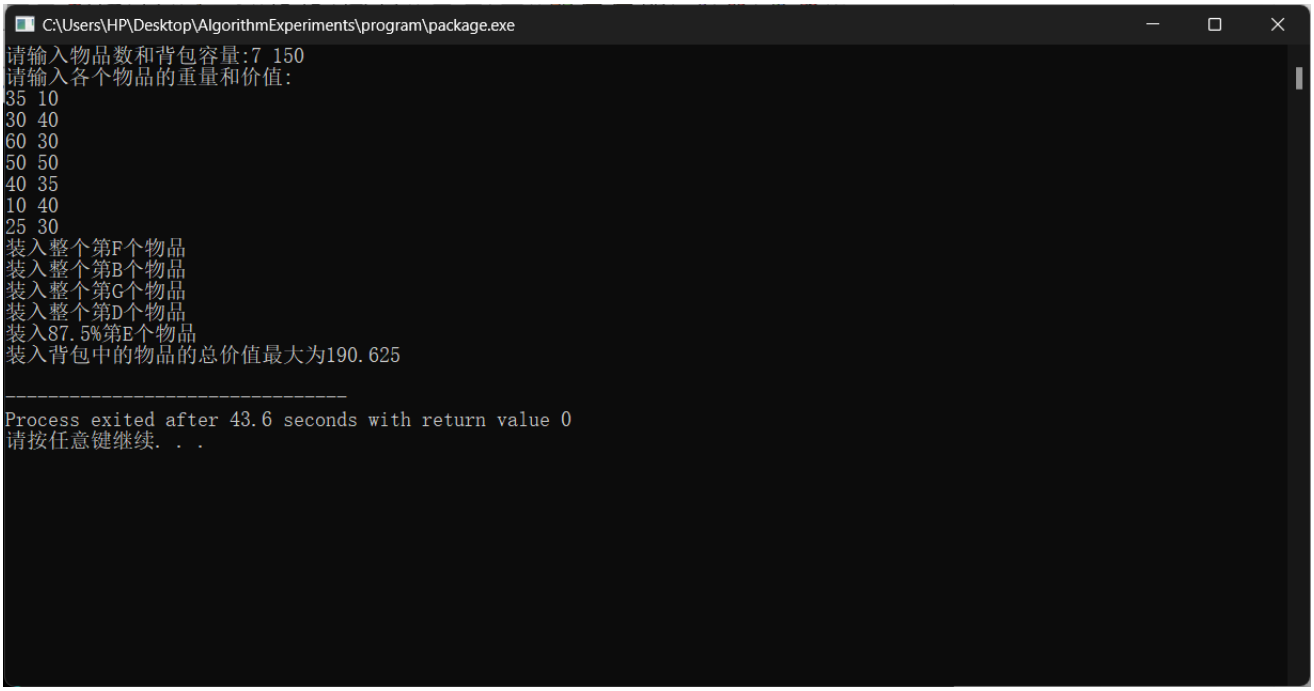
贪心算法的核心思想是每次选择当前最优的解决方案，即单位价值最高的物品放入背包。若某次不按照最大单位重量价值的物品进行放置，所得到的总价值一定不是最大的，则可以确定**贪心解就是最优解**，保证了贪心策略的正确性。

算法设计与代码实现

```
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;
}
```

算法演示:



算法讨论

背包问题有多个不同的变体，其中一些常见的种类包括：

1. 0/1背包问题（0/1 Knapsack Problem）：在这种问题中，每个物品要么完全放入背包，要么完全不放入背包，不能进行分割。即每个物品只有两种选择：放入背包或不放入背包。
2. 分数背包问题（Fractional Knapsack Problem）：这个问题允许物品被分割放入背包，即可以选择物品的一部分放入背包。每个物品有一个对应的重量和价值，目标是找到使总价值最大化的物品组合。
3. 多重背包问题（Multiple Knapsack Problem）：多重背包问题与0/1背包问题类似，但是每个物品有多个可用的实例（数量不限），可以选择将多个相同的物品放入背包中。
4. 无限背包问题（Unbounded Knapsack Problem）：在这个问题中，每个物品有无限个可用实例，可以无限次地选择物品放入背包。
5. 有限背包问题（Bounded Knapsack Problem）：这个问题介于0/1背包问题和无限背包问题之间，每个物品有一定数量的可用实例，可以选择将物品放入背包的次数受限。
6. 值约束背包问题（Knapsack Problem with Value Constraints）：在这个问题中，除了背包的容量限制外，还有对总价值的限制。目标是找到在满足总重量不超过背包容量的前提下，使总价值最大化的物品组合。

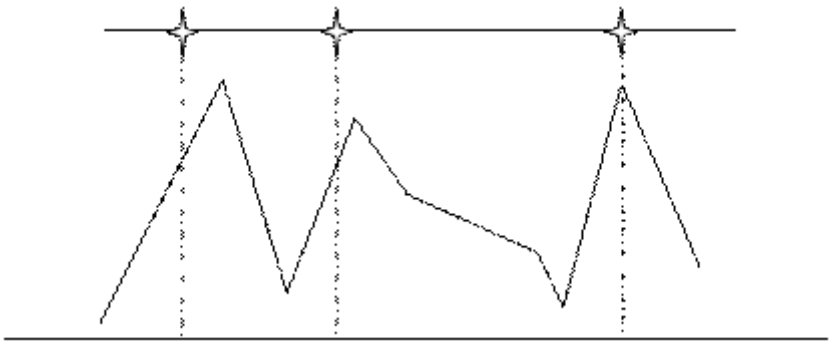
通常来说，背包问题使用动态规划来解决，但是在本题中，物品可以分割成任意大小，故可以通过贪心策略，从最大单位重量价值的物品开始装入背包，是背包的纵价值最大，算法的时间复杂度主要消耗在排序上，使用了C++自带的sort排序，时间复杂度是$O(nlogn)$

照亮的山景

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。



问题分析及算法思路

一座山想要能被照亮，那么把这座山的两侧分别延长，与灯所在的高度相交于两个点，在这个区间内如果有一盏灯，就可以照亮这座山，如果没有，就必须在区间两侧各有一盏灯。那么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，同时将已经照亮的山移出集合，标记灯为已使用，直到集合为空，所求的灯的数量就是最小的灯的数量。

这是在笔算时的解决方案，在计算时应该将其数据化。

首先找到**每个顶点能被灯照到的左端点和右端点**，可以采用遍历每个灯的做法，计算灯到顶点的直线的斜率 k 和截距 b，然后计算该灯到顶点这段距离区间内**所有顶点的横坐标投影到该直线的纵坐标**是否小于**该顶点的纵坐标**：

- 若成立，则认为该灯是无法照射到该顶点的，转而判断下一个灯；
- 若这段距离区间中所有顶点都存在以上条件，则认为该灯可以照射到该顶点，由该灯在顶点处的左右关系对顶点能被灯照到的左、右端点进行更新，并判断下一个灯。

若判断的灯已经在该顶点的右侧且该灯无法照到该顶点，则无需继续判断后续灯是否能照射到该顶点。

经过上述过程，我们得到了每个顶点被哪些灯照到, 记录到l和r中，接着根据**贪心策略**获得开灯最少的数量：

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，由贪心策略可知，其是我们要找的灯的序号。点亮该灯后，将能照射到的顶点删除，然后对剩余顶点重复上述过程，直到顶点被全部照亮。

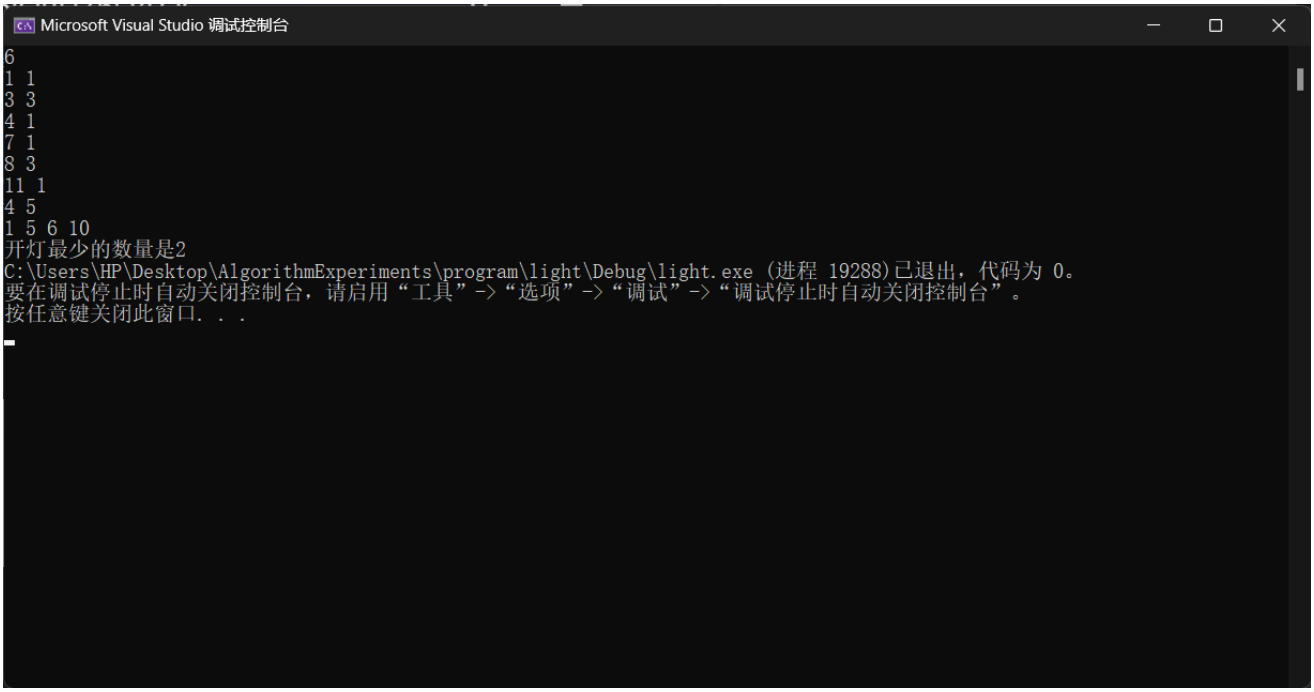
若贪心策略不是最优解，我们可以通过换部分灯的开关情况使其变为最优解，且不增加要亮灯的数目，则该贪心策略是正确的。

算法设计与代码实现

```
struct Mou
{
    int x;
    int y;
    int l;              // l 为顶点能被灯照到的左端点
    int r;              // r 为顶点能被灯照到的右端点
};
vector<Mou> mou;

void FindLightRegion(int m, int n, int t)
{
    for (int i = 0; i < m; i++)         // 遍历每个顶点
    {
        for (int j = 0; j < n; j++)     // 遍历每个灯
        {
            bool flag = true;

            if (l[j] != mou[i].x)
            {
                // 计算直线的斜率k和截距b
                float k = (float)(t - mou[i].y) / (l[j] - mou[i].x);
                float b = t - k * l[j];
                int s = i;
                while ((l[j] < mou[i].x && --s && l[j] < mou[s].x)
                    || (l[j] > mou[i].x && ++s && l[j] > mou[s].x))     // 遍历灯到该点区间内的所有点
                {
                    if (k * mou[s].x + b < mou[s].y)
                    {
                        flag = false;
                        break;
                    }
                }
            }
```

```
            if (flag)
            {
                if (mou[i].l == -1)
                {
                    mou[i].l = mou[i].r = j;
                }
                else
                {
                    mou[i].r++;
                }
            }
            else
            {
                if (l[j] > mou[i].x)
                    break;              // 无需继续遍历
            }
        }
    }
}

int FindMinLight(int m, int n, int t)
{
    int res = 0;
    FindLightRegion(m, n, t);   // 得到每个顶点被哪些灯照到，记录到 l 和 r 中
    while (mou.size() != 0)
    {
        // 统计区间每个数出现的次数
        int max = 0, cishu = 0;
        for (auto t : mou)
        {
            map<int, int> nums;
            for (int i = t.l; i <= t.r; i++)
            {
                nums[i]++;
            }
            // 找出区间中出现次数最多的数
            for (auto num : nums)
            {
                if (num.second > cishu)
                {
                    max = num.first;
                    cishu = num.second;
                }
            }
        }
        // 将出现次数最多的数设为当前需要的灯，然后删除所有有关的顶点
        res++;
        for (auto it = mou.begin(); it != mou.end();)
        {
            if (it[0].l <= max && it[0].r >= max)
                it = mou.erase(it);
            else
                ++it;
        }
    }
    return res;
}
```

算法演示：



算法讨论

在本题的求解中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$O(m*n^2)$。

搬桌子问题

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。

> **sample input**
>
> ```
> 10  5
> 1  3
> 3  9
> 4  6
> 6  10
> 7  8
> ```

sample output

```
3
```

问题分析及算法实现

使用贪心策略，尽可能在一趟从左向右的移动中移动尽可能多的桌子。

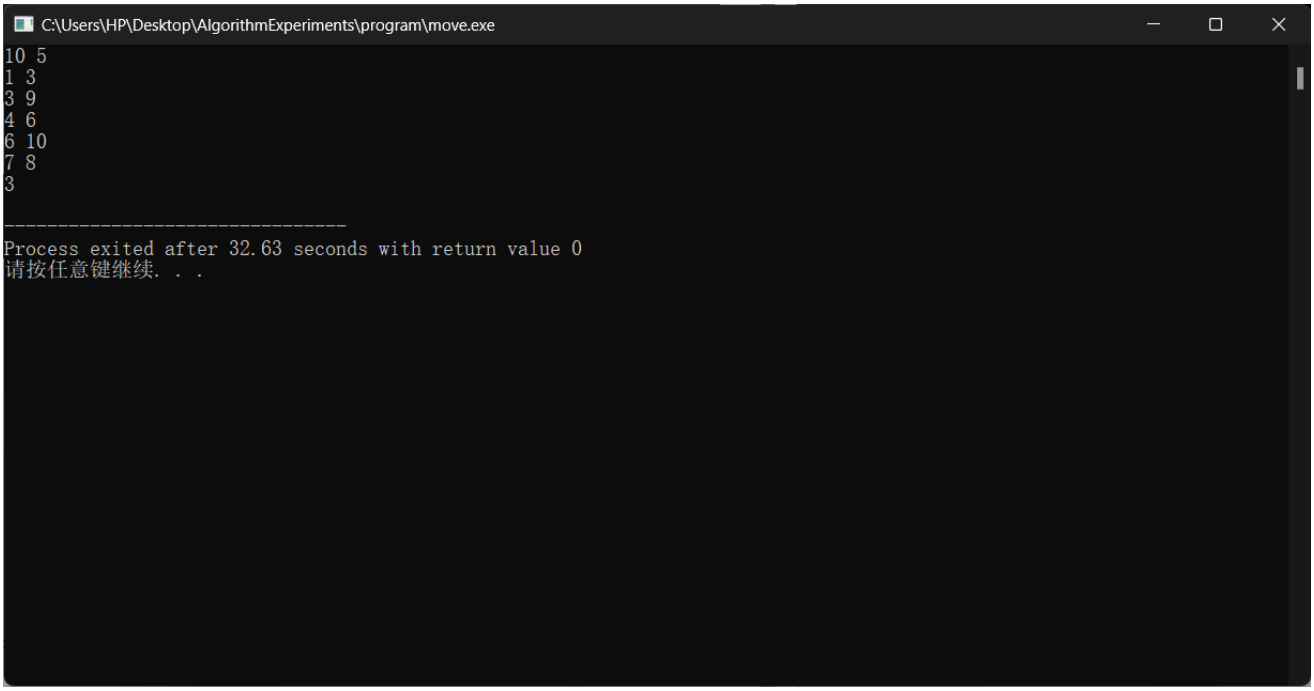
将每一次移动桌子的起始教室和目的教室作为一个区间，对剩余卓子的区间进行比较

- 如果存在剩余桌子的左界限大于上一张桌子的右界限，则可以进行下一张桌子的移动，进行上一步操作
- 反之，则结束这一趟移动，返回下一张桌子的左界限，重复上一个操作

算法设计与代码实现

```
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;
}
```

算法演示：



算法讨论

本算法需要进行遍历m张桌子的移动区间，消耗的时间复杂度$O(m^2)$，贪心的策略可以让单趟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。

附录

背包问题源代码

```
#include<iostream>
#include<algorithm>
using namespace std;
const int N = 20;
char l[35] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
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;
}

int main()
{
    int n, m;       // n为物品数, m为背包容量
    cout << "请输入物品数和背包容量:";
    cin >> n >> m;
    cout << "请输入各个物品的重量和价值:\n";
    for (int i = 0; i < n; i++)
    {
        cin >> obj[i].w >> obj[i].v;
        obj[i].id = i;
        obj[i].unit = (float) obj[i].v / obj[i].w;
    }
    FindMaxValue(n, m);
    return 0;
}
```

照亮的山景源代码

```
#include<iostream>
#include<map>
#include<vector>
using namespace std;
const int N = 20;
int l[N], top[N];       // top[N] 表示山峰的编号
struct Mou
{
    int x;
    int y;
    int l;              // l 为顶点能被灯照到的左端点
    int r;              // r 为顶点能被灯照到的右端点
};
vector<Mou> mou;
```

```
void FindLightRegion(int m, int n, int t)
{
    for (int i = 0; i < m; i++)         // 遍历每个顶点
    {
        for (int j = 0; j < n; j++)     // 遍历每个灯
        {
            bool flag = true;

            if (l[j] != mou[i].x)
            {
                // 计算直线的斜率k和截距b
                float k = (float)(t - mou[i].y) / (l[j] - mou[i].x);
                float b = t - k * l[j];
                int s = i;
                while ((l[j] < mou[i].x && --s && l[j] < mou[s].x)
                    || (l[j] > mou[i].x && ++s && l[j] > mou[s].x))     // 遍历灯到该点区间内的所
有点
                {
                    if (k * mou[s].x + b < mou[s].y)
                    {
                        flag = false;
                        break;
                    }
                }
            }

            if (flag)
            {
                if (mou[i].l == -1)
                {
                    mou[i].l = mou[i].r = j;
                }
                else
                {
                    mou[i].r++;
                }
            }
            else
            {
                if (l[j] > mou[i].x)
                    break;              // 无需继续遍历
            }
        }
    }
}

int FindMinLight(int m, int n, int t)
{
    int res = 0;
    FindLightRegion(m, n, t);   // 得到每个顶点被哪些灯照到, 记录到 l 和 r 中
    while (mou.size() != 0)
    {
        // 统计区间每个数出现的次数
        int max = 0, cishu = 0;
        for (auto t : mou)
        {
            map<int, int> nums;
            for (int i = t.l; i <= t.r; i++)
```

```
            {
                nums[i]++;
            }
            // 找出区间中出现次数最多的数
            for (auto num : nums)
            {
                if (num.second > cishu)
                {
                    max = num.first;
                    cishu = num.second;
                }
            }
        }
        // 将出现次数最多的数设为当前需要的灯，然后删除所有有关的顶点
        res++;
        for (auto it = mou.begin(); it != mou.end();)
        {
            if (it[0].l <= max && it[0].r >= max)
                it = mou.erase(it);
            else
                ++it;
        }
    }
    return res;
}

int main()
{
    int m, n, t;            // m为山棱转折点的个数，n为灯泡个数，t为灯泡的高度
    cin >> m;
    for (int i = 0; i < m; i++)
    {
        int x, y;
        cin >> x >> y;
        mou.push_back({ x,y,-1,-1 });   // 转折点的水平坐标和垂直海拔高度，并预初始化区间
    }
    cin >> n >> t;
    for (int i = 0; i < n; i++)
    {
        cin >> l[i];
    }
    cout << "开灯最少的数量是" << FindMinLight(m, n, t);
    return 0;
}
```

搬桌子问题源代码

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

const int N = 20;
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;
}

int main()
{
    int n, m;   // n为教室数, m为要搬运的工作数
    cin >> n >> m;
    for (int i = 0; i < m; i++)
    {
        cin >> mov[i].start >> mov[i].end;
        mov[i].use = false;
    }
    cout << runnum(n, m) << endl;
    return 0;
}
```