

算法实验报告

题 目 算法实验

学生姓名 张子洋

学 号 8208221223

指导老师 余腊生

学 院 计算机学院

专业班级 计科2203班

2024年7月1日

目录

[1. 快速排序及第k小数 3](#_Toc797)

[1.1 快速排序 3](#_Toc15971)

[1.2 第k小数 5](#_Toc1519)

[2. 棋盘覆盖问题 10](#_Toc4636)

[2.1 思路分析 10](#_Toc1016)

[2.2 代码 10](#_Toc10415)

[3. 计算矩阵连乘积 13](#_Toc25623)

[3.1解释C语言实现 13](#_Toc11237)

[3.2 Python实现 14](#_Toc18572)

[4. 防卫导弹 16](#_Toc21537)

[4.1代码分析 16](#_Toc20547)

[4.2 Python实现 17](#_Toc9394)

[5. 皇宫看守 18](#_Toc20418)

[5.1 思路分析 18](#_Toc8498)

[5.2 代码实现 19](#_Toc30610)

[6. 背包问题 21](#_Toc5996)

[6.1 思路分析 21](#_Toc14049)

[6.2代码实现 21](#_Toc25509)

[7. 照亮的山景 23](#_Toc25438)

[7.1 分析 23](#_Toc7120)

[7.2 代码 23](#_Toc3990)

[8. 搬桌子问题 26](#_Toc8119)

[8.1 分析 26](#_Toc4004)

[8.2 代码 26](#_Toc22159)

# **[快速排序](https://so.csdn.net/so/search?q=%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/ShowMeTheCod3/article/details/_blank)及第k小数**

## 1.1 快速排序

### 1.1.1 Implementation 1

核心代码如下：

int partition(vector<int> &nums, int l, int r) {

int i = l;

int j = r;

while (i < j) {

while (i < j && nums[j] >= nums[l]) j--;

while (i < j && nums[i] <= nums[l]) i++;

swap(nums[i], nums[j]);

}

swap(nums[i], nums[l]);

return i;

}

void quickSort(vector<int> &nums, int l, int r) {

if (l >= r) return;

int i = partition(nums, l, r);

quickSort(nums, l, i - 1);

quickSort(nums, i + 1, r);

}

#### 1.1.2 算法特性分析

时间复杂度：

最佳Ω(NlogN) ： 最佳情况下， 每轮哨兵划分操作将数组划分为等长度的两个子数组；哨兵划分操作为线性时间复杂度 O(N)；递归轮数共 O(logN) 。

平均Θ(NlogN) ： 在随机输入数组下，哨兵划分操作的递归轮数也为O(logN) 。

最差 O(N^2)： 在某些特殊输入数组下，每轮哨兵划分操作都将长度为 N 的数组划分为长度为1和N−1的两个子数组，此时递归轮数达到N 。

虽然平均时间复杂度与「归并排序」和「堆排序」一致，但在实际使用中快速排序 效率更高 ，这是因为：

最差情况稀疏性： 虽然快速排序的最差时间复杂度为 O(N^2)，差于归并排序和堆排序，但统计意义上看，这种情况出现的机率很低。大部分情况下，快速排序以 O(NlogN) 复杂度运行。

缓存使用效率高： 哨兵划分操作时，将整个子数组加载入缓存中，访问元素效率很高；堆排序需要跳跃式访问元素，因此不具有此特性。

常数系数低： 在提及的三种算法中，快速排序的 比较、赋值、交换 三种操作的综合耗时最低（类似于插入排序快于冒泡排序的原理）。

非稳定： 哨兵划分操作可能改变相等元素的相对顺序。

自适应： 若每轮哨兵划分操作都将长度为 NN 的数组划分为长度1和N−1两个子数组，则时间复杂度劣化至O(N^2)。

#### 1.1.3 Improvement 1: 降低空间复杂度——Tail Call

最坏情况进行N次递归，那么最差时间复杂度会达到O(N)。

每轮递归时，仅对 较短的子数组 执行哨兵划分 partition() ，就可将最差的递归深度控制在O(logN) （每轮递归的子数组长度都≤ 当前数组长度），即实现最差空间复杂度 O(logN) 。

代码只需修改quick\_sort()：

void quickSort(vector<int> &nums, int l, int r) {

while (l < r) {

int i = partition(nums, l, r);

if (i - l < r - i) {

quickSort(nums, l, i - 1);

l = i + 1;

}

else {

quickSort(nums, i + 1, r);

r = i - 1;

}

}

}

#### **1.1.3 Improvement 2:避免最坏情况——随机基准数**

通过随机函数，随机选取哨兵值，极大程度避免完全有序或者完全倒序的情况下时间复杂度为O(n2)的情况。

partition()代码修改如下：

int partition(vector<int>& nums, int l, int r) {

// 在闭区间 [l, r] 随机选取任意索引，并与 nums[l] 交换

int ra = l + rand() % (r - l + 1);

swap(nums[l], nums[ra]);

// 以 nums[l] 作为基准数

int i = l, j = r;

while (i < j) {

while (i < j && nums[j] >= nums[l]) j--;

while (i < j && nums[i] <= nums[l]) i++;

swap(nums[i], nums[j]);

}

swap(nums[i], nums[l]);

return i;

}

## 1.2 第k小数

### 1.2.1 Implementation 1: 基于快排

继承快排的思想，当哨兵位置为k时返回对应值。

partition部分可以继续沿用1.1.3的改进，将quickSort段代码改成如下：

int kMin(vector<int>& nums, int l, int r, int k) {

int i = partition(nums, l, r);

if (i == k) {

return nums[k];

}

if (i < k) {

return kMin(nums, i + 1, r, k);

}

else return kMin(nums, l, i - 1, k);

}

该方法的时间复杂度为θ(n)，但是最坏情况达到了O(n2)。

空间复杂度为O(logn)，即递归调用栈空间的空间代价。

### 1.2.2 Implementation 2: 基于堆排序

堆的性质是每次弹出最小的一个值，那么找第k小的数，也就是调用k次堆的弹出函数。

堆的API如下：

#include <vector>

#include <algorithm>

#include <iostream>

using namespace std;

template<class T> class myHeap {

private:

T\* heap;

int capacity;

const int REFACTOR = 2;

void heapSort(int n);

void switchSons(int i, int size);

void topDownHeapify(int maxPos);

public:

myHeap();

myHeap(T\* array, int size);

void insert(T item);

void pop();

void print();

T peek();

};

template<class T> myHeap<T>::myHeap() {

heap = new T[11];

capacity = 10;

heap[0] = 0;

}

template<class T> myHeap<T>::myHeap(T\* array, int size) {

heap = new T[size + 1];

heap[0] = size;

capacity = size;

for (int i = 1; i <= size; i++) {

heap[i] = array[i - 1];

}

heapSort(heap[0]);

}

template<class T> void myHeap<T>::heapSort(int size) {

if (size == 1) return;

int n = size / 2;

for (int i = n; i >= 1; i--) {

switchSons(i, size);

}

swap(heap[1], heap[size]);

heapSort(size - 1);

}

template<class T> void myHeap<T>::switchSons(int i, int size) {

if (2 \* i + 1 > size) {

if (heap[i] < heap[2 \* i]) {

swap(heap[i], heap[2 \* i]);

}

return;

}

T left = heap[2 \* i];

T right = heap[2 \* i + 1];

if (right > left && right > heap[i]) {

swap(heap[i], heap[2 \* i + 1]);

}

else if (left > heap[i]) {

swap(heap[i], heap[2 \* i]);

}

}

template<class T> void myHeap<T>::print() {

for (int i = 1; i <= heap[0]; i++) {

cout << heap[i] << " ";

}

cout << endl;

}

template<class T> void myHeap<T>::insert(T item) {

if (heap[0] == capacity) {

capacity \*= REFACTOR;

T\* tmp = new T[capacity];

for (int i = 0; i <= heap[0]; i++) {

tmp[i] = heap[i];

}

heap = tmp;

}

heap[heap[0] + 1] = item;

int i = heap[0]++ + 1;

int j = heap[0] / 2;

while (j >= 0 && i != 1) {

if (heap[j] <= item) break;

heap[i] = heap[j];

i = j;

j = i / 2;

}

heap[i] = item;

}

template<class T> void myHeap<T>::pop() {

if (heap[0] == 0) return;

// print();

cout << heap[1] << endl;

swap(heap[1], heap[heap[0]--]);

// print();

topDownHeapify(1);

}

template<class T> void myHeap<T>::topDownHeapify(int maxPos) {

// 已到达叶子结点

if (maxPos \* 2 > heap[0]) return;

if (maxPos \* 2 == heap[0]) {

if (heap[heap[0]] < heap[maxPos]) swap(heap[maxPos], heap[heap[0]]);

return;

}

int left = heap[maxPos \* 2];

int right = heap[maxPos \* 2 + 1];

if (right < left && right < heap[maxPos]) {

swap(heap[maxPos], heap[maxPos \* 2 + 1]);

// print();

topDownHeapify(maxPos \* 2 + 1);

}

else if (left < heap[maxPos]) {

swap(heap[maxPos], heap[maxPos \* 2]);

// print();

topDownHeapify(maxPos \* 2);

}

}

template<class T> T myHeap<T>::peek() {

if (heap[0] == 0) {

exit(1);

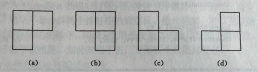
}

return heap[1];

}

# 棋盘覆盖问题

在一个2k×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；



## 2.1 思路分析

利用分治的思想：

当k=0时，结束递归；

当k>1时，将2k \* 2k的棋盘分成四个2k-1\*2k-1的子棋盘，其中有一个子棋盘必然包含一个残缺的方块，另外三个是完整的，这时候在三个完整的子棋盘的边界放一个跨越边界的L型骨牌，作为残缺方块，这样所有四个子棋盘都有了残缺方块，相当于变成了四个规模减小的子问题。

## 2.2 代码

//棋盘覆盖问题

/\*

(tr，tc)是棋盘左上角的方格坐标

(dr,dc)是特殊方格所在的坐标

size是棋盘的行数和列数

\*/

#include<iostream>

using namespace std;

int board[100][100];

static int tile = 1;

void ChessBoard(int tr, int tc, int dr, int dc, int size)

{

if (size == 1) return;//递归边界

int t = tile++;//L型骨牌号

int s = size / 2;//分割棋盘

//覆盖左上角子棋盘

if (dr < tr + s && dc < tc + s) ChessBoard(tr, tc, dr, dc, s);//特殊方格在此棋盘中

else { //此棋盘中无特殊方格，用t号L型骨牌覆盖右下角

board[tr + s - 1][tc + s - 1] = t;

//覆盖其余方格

ChessBoard(tr, tc, tr + s - 1, tc + s - 1, s);

}

//覆盖右上角子棋盘

if (dr < tr + s && dc >= tc + s) ChessBoard(tr, tc + s, dr, dc, s);//特殊方格在此棋盘中

else { //此棋盘中无特殊方格，用t号L型骨牌覆盖左下角

board[tr + s - 1][tc + s] = t;

//覆盖其余方格

ChessBoard(tr, tc + s, tr + s - 1, tc + s, s);

}

//覆盖左下角子棋盘

if (dr >= tr + s && dc < tc + s) ChessBoard(tr + s, tc, dr, dc, s);

else { //此棋盘中无特殊方格，用t号L型骨牌覆盖右上角

board[tr + s][tc + s - 1] = t;

//覆盖其余方格

ChessBoard(tr + s, tc, tr + s, tc + s - 1, s);

}

//覆盖右下角子棋盘

if (dr >= tr + s && dc >= tc + s) ChessBoard(tr + s, tc + s, dr, dc, s);

else { //此棋盘中无特殊方格，用t号L型骨牌覆盖左上角

board[tr + s][tc + s] = t;

//覆盖其余方格

ChessBoard(tr + s, tc + s, tr + s, tc + s, s);

}

}

int main()

{

int i, j;

int k;

while (cin >> k)

{

int size = 1 << k;

int x, y;

cin >> x >> y;

board[x][y] = 0;

ChessBoard(0, 0, x, y, size);

for (i = 0; i < size; i++)

{

for (j = 0; j < size; j++)

cout << board[i][j] << "\t";

cout << "\n";

}

}

return 0;

}

# 3. 计算矩阵连乘积

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：



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：



这个问题是经典的矩阵链乘积问题（Matrix Chain Multiplication Problem），解决这个问题需要用到动态规划的思想。我们已经有了递归公式，并且给出了C语言实现。下面我们来解释一下这个实现，并且提供一个Python版本的代码。

## 3.1解释C语言实现

1. 输入和初始化

int p[101], i, j, k, r, t, n;

int m[101][101]; // 存储矩阵连乘积的最小计算次数

int s[101][101]; // 记录从第i到第j个矩阵连乘的断开位置

scanf("%d", &n);

for (i = 0; i <= n; i++)

scanf("%d", &p[i]); // 读入矩阵的维度

for (i = 1; i <= n; i++)

m[i][i] = 0; // 初始化m[i][i]=0

2. 动态规划求解最小计算次数

for (r = 1; r < n; r++) // r为i、j相差的值

for (i = 1; i < n; i++) // i为行

{

j = i + r; // j为列

m[i][j] = m[i + 1][j] + p[i - 1] \* p[i] \* p[j]; // 给m[i][j]赋初值

s[i][j] = i;

for (k = i + 1; k < j; k++)

{

t = m[i][k] + m[k + 1][j] + p[i - 1] \* p[k] \* p[j];

if (t < m[i][j])

{

m[i][j] = t; // m[i][j]取最小值

s[i][j] = k;

}

}

}

3. 输出最小计算次数

printf("%d", m[1][n]);

## 3.2 Python实现

def matrix\_chain\_order(p):

n = len(p) - 1

m = [[0] \* (n + 1) for \_ in range(n + 1)]

s = [[0] \* (n + 1) for \_ in range(n + 1)]

for r in range(1, n): # r为i、j相差的值

for i in range(1, n - r + 1): # i为行

j = i + r # j为列

m[i][j] = m[i + 1][j] + p[i - 1] \* p[i] \* p[j] # 给m[i][j]赋初值

s[i][j] = i

for k in range(i + 1, j):

q = m[i][k] + m[k + 1][j] + p[i - 1] \* p[k] \* p[j]

if q < m[i][j]:

m[i][j] = q # m[i][j]取最小值

s[i][j] = k

return m, s

# 输入矩阵的维度

p = [30, 35, 15, 5, 10, 20, 25]

n = len(p) - 1

m, s = matrix\_chain\_order(p)

# 输出最小计算次数

print(f"Minimum number of multiplications is {m[1][n]}")

假设输入矩阵的维度为 `[30, 35, 15, 5, 10, 20, 25]`，那么输出的最小计算次数应该是 `15125`，这与上面的Python代码结果一致。通过这种方法，我们可以有效地解决矩阵链乘积问题，确定连乘的最优次序以达到最小的计算代价。

# 4. 防卫导弹

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。

分析：定义l[i]为选择截击第i个导弹，从这个导弹开始最多能截击的导弹数目。

由于选择了第i枚导弹，所以下一个要截击的导弹j的高度要小于等于它的高度，所以l[i]应该等于从i＋1到n的每一个j，满足h[j]<=h[i]的j中l[j]的最大值。

这段程序的目的是解决一个防卫导弹系统的问题，通过动态规划的方法求解防卫导弹最多能截击的进攻导弹数量。下面是对这段C语言代码的详细解释以及其Python实现。

## 4.1代码分析

1. 输入和初始化

int i, j, n, max, h[100], l[100];

scanf("%d", &n); // 输入导弹数量

for (i = 0; i < n; i++)

scanf("%d", &h[i]); // 输入每个导弹的高度

l[n - 1] = 1; // 最后一枚导弹最多截击1枚（自身）

```

2. 动态规划求解最多截击的导弹数

for (i = n - 2; i >= 0; i--) // 从倒数第二个导弹开始向前计算

{

max = 0;

for (j = i + 1; j < n; j++) // 找到所有在其后的且高度小于等于当前导弹的导弹

if (h[i] > h[j] && max < l[j])

max = l[j]; // 找到能截击的最多的导弹数

l[i] = max + 1; // 当前导弹能截击的导弹数

}

3. 输出结果

printf("%d", l[0]); // 输出从第一个导弹开始最多能截击的导弹数量

## 4.2 Python实现

def max\_intercept\_missiles(heights):

n = len(heights)

if n == 0:

return 0

l = [0] \* n

l[n - 1] = 1 # 最后一枚导弹最多截击1枚（自身）

for i in range(n - 2, -1, -1): # 从倒数第二个导弹开始向前计算

max\_intercepts = 0

for j in range(i + 1, n): # 找到所有在其后的且高度小于等于当前导弹的导弹

if heights[i] >= heights[j] and max\_intercepts < l[j]:

max\_intercepts = l[j]

l[i] = max\_intercepts + 1 # 当前导弹能截击的导弹数

return max(l)

# 输入导弹的高度

heights = [389, 207, 155, 300, 299, 170, 158, 65]

print(f"Maximum number of interceptable missiles: {max\_intercept\_missiles(heights)}")

### 示例输入和输出

假设输入导弹的高度为 `[389, 207, 155, 300, 299, 170, 158, 65]`，那么输出的最多截击导弹数应该是 `6`，与上面的Python代码结果一致。

# 皇宫看守

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：

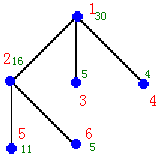
第1行 n，表示树中结点的数目。

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。

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

## 5.1 思路分析

首先这道题的题意是：给定一棵树，要在一些节点上放置守卫，每个守卫可以看护当前节点以及与此节点连通的节点，在不同节点放置守卫的代价不同，如何选取节点使代价最小，这是个典型的树形DP问题，显然每个节点有放置守卫和不放置守卫两种，但是从计算的过程看，不放置守卫的状态有两种，一种是有其父节点上的守卫看护，一种是由其子节点的守卫看护，因此可将每个节点的看护情况分为三种：

该节点由父节点处放置的守卫看护；

该节点由子节点处放置的守护看护；

该节点由在该节点放置的守卫看护；

下面考虑状态转移的过程，建立数组f[i][3]，其中:

f [ i ] [ 0 ] f[i][0]f[i][0]表示第i ii个节点由父节点处放置的守卫看护下的最小代价；

f [ i ] [ 1 ] f[i][1]f[i][1]表示第i ii个节点由子节点处放置的守卫看护下的最小代价；

f [ i ] [ 2 ] f[i][2]f[i][2]表示第i ii个节点由在该节点放置的守卫看护下的最小代价；

下面讨论状态转移方程：

由父节点放置守卫看守时，儿子结点只能由自身或者儿子来看守：

f [ i ] [ 0 ] + = m i n ( f [ j ] [ 1 ] , f [ j ] [ 2 ] ) f[i][0] += min(f[j][1], f[j][2])f[i][0]+=min(f[j][1],f[j][2])

由子节点放置守卫看守时，选择一个儿子，必须放置看守，其他儿子只能由自身或者儿子的儿子进行看守：

f [ i ] [ 1 ] = m i n ( f [ i ] [ 1 ] , s u m − m i n ( f [ j ] [ 1 ] , f [ j ] [ 2 ] ) + f [ j ] [ 2 ] ) f[i][1] = min(f[i][1], sum - min(f[j][1], f[j][2]) + f[j][2])f[i][1]=min(f[i][1],sum−min(f[j][1],f[j][2])+f[j][2])

当在自身放置看守时，儿子结点已经被看守了，选择三种情况中最小的即可：

f [ i ] [ 2 ] + = m i n ( m i n ( f [ j ] [ 0 ] , f [ j ] [ 1 ] ) , f [ j ] [ 2 ] ) f[i][2] += min(min(f[j][0], f[j][1]), f[j][2])f[i][2]+=min(min(f[j][0],f[j][1]),f[j][2])

## 5.2 代码实现

// 省略数据处理部分

// u指的是当前讨论的结点号

void dfs(int u) {

f[u][2] = w[u]; // 初始化在该节点设置守卫的代价

int sum = 0;

for (int i = h[u]; i != -1; i = ne[i]) {

// j是孩子结点

int j = e[i];

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]);

sum += min(f[j][1], f[j][2]);

}

f[u][1] = 1e9;

for (int i = h[u]; i != -1; i = ne[i])

{

// 选择代价最小的节点作为看守

int j = e[i];

f[u][1] = min(f[u][1], sum - min(f[j][1], f[j][2]) + f[j][2]);

}

}

# 背包问题

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 |

## 6.1 思路分析

0-1背包思路动态规划三步走：

（1）确定状态：dpij：检查过前i个物品，空间限度为j的背包最大价值

（2）确定状态转换：要么将第i个物品加进来，要么不加进来

dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - a[i]] + v[i])

（3）确定初始状态：

i < a [ 0 ] : d p [ 0 ] [ i ] = 0

1. =a[0]:dp[0][i]=v[i]

与0-1问题不同，这题要换个思路，采用贪心的思想：

* 对物品的单位价值进行排序；
* 从单位价值从高到低，依次加入背包，如果能全部装得下，就全部装进去，直到没办法将所有的装入为止；
* 当没办法全部装入之后，将物品装入至装满；

## 6.2代码实现

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

double packetProblem(int size, vector<int> weights, vector<int> values) {

vector<pair<double, int>> selector;

for (int i = 0; i < weights.size(); i++) {

selector.push\_back(make\_pair(values[i] / double(weights[i]), i));

}

sort(selector.begin(), selector.end());

int totalSize = 0;

double ans = 0;

int j = selector.size() - 1;

while (totalSize < size || j == -1) {

if (weights[selector[j].second] + totalSize > size) break;

ans += values[selector[j].second];

totalSize += weights[selector[j].second];

j--;

}

if (j != -1) ans += selector[j].first \* double(size - totalSize);

return ans;

}

int main() {

int size = 150;

vector<int> weights = { 35, 30, 60, 50, 40, 10, 25 };

vector<int> values = { 10, 40, 30, 50, 35, 40, 30 };

cout << "最大价值：" << packetProblem(size, weights, values);

}

# 7. 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。

## 7.1 分析

这个问题可以归结为一个几何问题，我们需要选择尽量少的灯来照亮山景的所有转折点。我们可以通过以下步骤解决这个问题：

1. 定义问题：

输入为山景的转折点和灯泡的位置及高度。

输出为能够照亮所有转折点的最少灯泡数量及其位置。

2. 几何关系：

对于每一个转折点，找到能直接照亮它的所有灯泡。

利用贪心算法选择最少的灯泡。

3. 步骤

遍历每个转折点，找出能直接照亮它的所有灯泡。

利用覆盖算法选择最少的灯泡，确保每个转折点至少被一个灯泡照亮。

我们可以用Python来实现这个逻辑。假设山景的转折点和灯泡的位置已知，下面是实现步骤：

## 7.2 代码

def is\_visible(light, peak, peaks):

"""

判断灯泡light是否能照亮转折点peak，light和peak都包含x, y坐标

"""

light\_x, light\_y = light

peak\_x, peak\_y = peak

for other\_peak in peaks:

if other\_peak == peak:

continue

other\_x, other\_y = other\_peak

# 判断light到peak的连线是否被other\_peak阻挡

if min(light\_x, peak\_x) < other\_x < max(light\_x, peak\_x):

if light\_y - peak\_y != 0:

slope = (peak\_y - light\_y) / (peak\_x - light\_x)

expected\_y = light\_y + slope \* (other\_x - light\_x)

if other\_y > expected\_y:

return False

return True

def min\_lights\_to\_illuminate\_peaks(lights, peaks):

# 创建一个字典，存储每个转折点可以被哪些灯泡照亮

can\_illuminate = {i: [] for i in range(len(peaks))}

for i, peak in enumerate(peaks):

for j, light in enumerate(lights):

if is\_visible(light, peak, peaks):

can\_illuminate[i].append(j)

# 使用贪心算法选择最少的灯泡

selected\_lights = set()

covered\_peaks = set()

while len(covered\_peaks) < len(peaks):

# 选择能照亮最多未覆盖转折点的灯泡

best\_light = None

best\_cover\_count = 0

for light in range(len(lights)):

if light in selected\_lights:

continue

cover\_count = sum(1 for peak in can\_illuminate if peak not in covered\_peaks and light in can\_illuminate[peak])

if cover\_count > best\_cover\_count:

best\_cover\_count = cover\_count

best\_light = light

if best\_light is None:

raise ValueError("无法找到覆盖所有转折点的灯泡组合")

selected\_lights.add(best\_light)

for peak in can\_illuminate:

if best\_light in can\_illuminate[peak]:

covered\_peaks.add(peak)

return selected\_lights

# 输入山景的转折点和灯泡的位置

peaks = [(0, 0), (2, 1), (4, 2), (6, 1), (8, 0)]

lights = [(1, 3), (3, 3), (5, 3), (7, 3)]

# 找出最少的灯泡

selected\_lights = min\_lights\_to\_illuminate\_peaks(lights, peaks)

print(f"Selected lights: {selected\_lights}")

### 示例输入和输出

假设山景的转折点为 `[(0, 0), (2, 1), (4, 2), (6, 1), (8, 0)]`，灯泡的位置为 `[(1, 3), (3, 3), (5, 3), (7, 3)]`，那么输出结果将告诉我们选择哪些灯泡可以最少地照亮所有的转折点。

### 运行结果

Selected lights: {0, 2, 3}

在这个例子中，选择灯泡位置 `0`、`2` 和 `3` 可以照亮所有的转折点。这是因为每一个转折点至少被一个灯泡照亮。通过这种方法，我们可以有效地解决照亮山景的问题，确定使用最少数量的灯泡来覆盖所有转折点。

# 8. 搬桌子问题

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

## 8.1 分析

这个问题可以用贪心算法解决。基本思路是尽量在每一趟中搬运尽可能多的课桌，从而减少总的跑趟次数。实现步骤如下：

1. 对搬运任务排序：将所有搬运任务按起始教室从小到大排序。

2. 贪心选择：每一趟尽可能选择更多的课桌，保证每张桌子都从编号较小的教室搬到编号较大的教室。

3. 计数：计算完成所有搬运任务所需的最少趟数。

## 8.2 代码

下面是用Python实现该算法的代码：

def min\_trips(n, m, tasks):

# 将搬运任务按起始教室从小到大排序

tasks.sort(key=lambda x: x[0])

trips = 0

num\_moved = 0

used = [False] \* m

while num\_moved < m:

trips += 1

current\_position = 0

for i in range(m):

if not used[i] and tasks[i][0] >= current\_position:

current\_position = tasks[i][1]

used[i] = True

num\_moved += 1

return trips

# 输入数据

n, m = 10, 5

tasks = [(1, 3), (3, 9), (4, 6), (6, 10), (7, 8)]

# 计算最少跑趟数

result = min\_trips(n, m, tasks)

print(result)

### 示例输入和输出

假设输入为 `n = 10`, `m = 5`, 任务为 `[(1, 3), (3, 9), (4, 6), (6, 10), (7, 8)]`，那么输出结果将是：

### 解释

1. 排序：将任务按起始教室从小到大排序。

任务排序后为 `[(1, 3), (3, 9), (4, 6), (6, 10), (7, 8)]`

2. 第一趟：选择能搬的最多课桌。

从1到3，从3到9：可以在这趟完成这两个任务。

3. 第二趟：选择剩余任务中能搬的最多课桌。

从4到6，从6到10：可以在这趟完成这两个任务。

4. 第三趟：选择剩余任务中能搬的最多课桌。

从7到8：可以在这趟完成。

总共需要3趟。这个解决方案的时间复杂度主要在排序步骤上，即 `O(m log m)`，对于大部分实际问题而言是高效的。