** **

**课程实验报告**

课程名： 算法分析与设计实验

学 院： 信息工程学院 系 计算机科学与技术系

专 业： 计算机科学与技术(卓越工程师)

班 级： 计算机科学与技术(卓越工程师) 221班

学 号： 5418122020

姓 名： 马星

任课教师： 刘萍

授课学期： 2024 年~~~~ 2025 年 秋季 学期

**目 录**

[实验一 递归深度与递归的非递归实现 4](#_Toc186320981)

[一、实验项目名称 4](#_Toc186320982)

[二、实验目的 4](#_Toc186320983)

[三、实验基本原理 4](#_Toc186320984)

[四、主要仪器设备及耗材 6](#_Toc186320985)

[五、实验步骤（完整内容见光盘） 6](#_Toc186320986)

[第一题：递归函数 6](#_Toc186320987)

[第二题：二叉树深度 8](#_Toc186320988)

[六、实验数据及处理结果 12](#_Toc186320989)

[七、思考讨论题或体会或对改进实验的建议 13](#_Toc186320990)

[八、参考资料 13](#_Toc186320991)

[实验二 分治法 14](#_Toc186320992)

[一、实验项目名称 14](#_Toc186320993)

[二、实验目的 14](#_Toc186320994)

[三、实验基本原理 14](#_Toc186320995)

[四、主要仪器设备及耗材 15](#_Toc186320996)

[五、实验步骤（完整内容见光盘） 15](#_Toc186320997)

[第1题：改写分治算法 15](#_Toc186320998)

[第2题：求第K小 17](#_Toc186320999)

[六、实验数据及处理结果 19](#_Toc186321000)

[第一题 改写分治算法(数据对拍): 19](#_Toc186321001)

[第二题 求第K小(数据对拍): 21](#_Toc186321002)

[七、思考讨论题或体会或对改进实验的建议 23](#_Toc186321003)

[八、参考资料 23](#_Toc186321004)

[实验三 贪心法 24](#_Toc186321005)

[一、实验项目名称 24](#_Toc186321006)

[二、实验目的 24](#_Toc186321007)

[三、实验基本原理 24](#_Toc186321008)

[四、主要仪器设备及耗材 25](#_Toc186321009)

[五、实验步骤（完整内容见光盘） 25](#_Toc186321010)

[第1题 背包问题 25](#_Toc186321011)

[第2题 带时限的作业排序问题 27](#_Toc186321012)

[第3题 Minimum-Cost Spanning Tree 30](#_Toc186321013)

[六、实验数据及处理结果 37](#_Toc186321014)

[第1题 背包问题 37](#_Toc186321015)

[第2题 带时限的作业排序问题 37](#_Toc186321016)

[第3题 Minimum-Cost Spanning Tree 41](#_Toc186321017)

[七、思考讨论题或体会或对改进实验的建议 42](#_Toc186321018)

[八、参考资料 43](#_Toc186321019)

[实验四 动态规划法 45](#_Toc186321020)

[一、实验项目名称 45](#_Toc186321021)

[二、实验目的 45](#_Toc186321022)

[三、实验基本原理 45](#_Toc186321023)

[四、主要仪器设备及耗材 46](#_Toc186321024)

[五、实验步骤（完整内容见光盘） 46](#_Toc186321025)

[第1题 多段图问题 46](#_Toc186321026)

[第2题 矩阵连乘问题 49](#_Toc186321027)

[第3题 最长公共子序列问题 52](#_Toc186321028)

[六、实验数据及处理结果 54](#_Toc186321029)

[第1题 多段图问题 54](#_Toc186321030)

[第2题 矩阵连乘问题 56](#_Toc186321031)

[第3题 最长公共子序列问题 57](#_Toc186321032)

[七、思考讨论题或体会或对改进实验的建议 58](#_Toc186321033)

[(1)实验体会： 58](#_Toc186321034)

[(2)改进建议： 59](#_Toc186321035)

[八、参考资料 60](#_Toc186321036)

# 实验一 递归深度与递归的非递归实现

学生姓名：马星 学 号：5418122020专业班级：计算机科学与技术(卓越工程师)221班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期： 2024/12/5 实验成绩：

## 一、实验项目名称

递归深度与递归的非递归实现

## 二、实验目的

1．掌握递归实现的机理

2．掌握堆内存和栈内存的概念及应用

3．掌握递归的非递归实现方法

## 三、实验基本原理

**1．递归的原理**

一、递归编程：递归（recursion）就是子程序（或函数）直接调用自己或通过一系列调用语句间接调用自己，是一种描述问题和解决问题的基本方法。

1，采用递归方法的原因：

递归通常用来解决结构自相似的问题。所谓结构自相似，是指构成原问题的子问题与原问题在结构上相似，可以用类似的方法解决。具体地，整个问题的解决，可以分为两部分：第一部分是一些特殊情况，有直接的解法；第二部分与原问题相似，但比原问题的规模小。实际上，递归是把一个不能或不好解决的大问题转化为一个或几个小问题，再把这些小问题进一步分解成更小的问题，直至每个小问题都可以直接解决。

2，递归的两个基本要素：

边界条件：确定递归到何时终止，也称为递归出口。

递归模式：大问题是如何分解为小问题的，也称为递归体。递归函数只有具备了这两个要素，才能在有限次计算后得出结果

二、递归原理：

1，递归函数的内部执行过程：

一个递归函数的调用过程类似于多个函数的嵌套的调用，只不过调用函数和被调用函数是同一个函数。为了保证递归函数的正确执行，系统需设立一个工作栈。具体地说，递归调用的内部执行过程如下：

（1）运行开始时，首先为递归调用建立一个工作栈，其结构包括值参、局部变量和返回地址；

（2）每次执行递归调用之前，把递归函数的值参和局部变量的当前值以及调用后的返回地址压栈；

（3）每次递归调用结束后，将栈顶元素出栈，使相应的值参和局部变量恢复为调用前的值，

然后转向返回地址指定的位置继续执行。

2，递归调用有层级关系：如果把调用递归函数的主函数称为第0层，进入函数后，首次递归调用自身称为第1层调用；从第i层递归调用自身称为第i+1层。反之，退出第i+1层调用应该返回第i层。

每一级的函数调用都有它自己的变量。每一次函数调用都会有一次返回，并且是某一级递归返回到调用它的那一级，而不是直接返回到main()函数中的初始调用部分。

递归函数中，位于递归调用前的语句和各级被调函数具有相同的执行顺序。位于递归调用语句之前，它按照递归调用的顺序被执行，即依次为第一级、第二级、第三级、第四级……。与上面不同，位于递归调用后的语句的执行顺序和各个被调函数的顺序相反。位于递归调用语句之后，其执行顺序依次是：….第四级、第三级、第二级、第一级。

3. 由于栈内存的容量限制，递归深度是有限制的。当递归深度超过了这个限制，就不能用递归。此时就必须把递归方式改为非递归实现。非递归实现有两种方式：一种，是根据递归算法，把它改为循环方式，这种方法没有统一的框架；另一种，模拟递归的机器实现方式，用栈（数组或STL的栈）来实现，这个有可借鉴的框架，可以参考教材。

## 四、主要仪器设备及耗材

PC微机

DOS操作系统或 Windows 操作系统

CLion 集成开发环境

## 五、实验步骤（完整内容见光盘）

### 第一题：递归函数

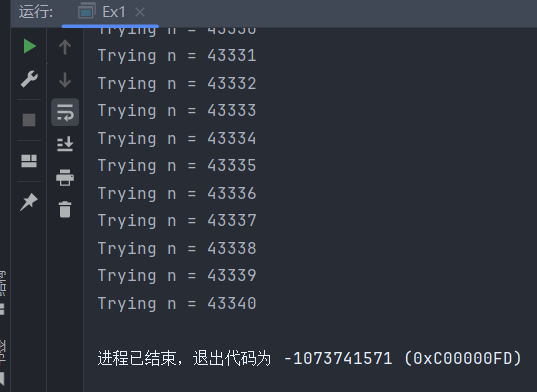
递归函数f(n)=n+f(n-1)，f(1)=1。用递归实现计算，检查可以运算的最大的n，及递归深度。然后在代码中增加一些无用的变量，以便大幅度减少递归深度。

#### (1) 检查最大递归深度

#include <bits/stdc++.h>  
  
using namespace std;  
  
int f(int n) {  
 if (n == 1) return 1;  
 return n + f(n - 1);  
}  
  
int main() {  
 int n = 1;  
 while (true) {  
 cout << "Trying n = " << n << endl;  
 f(n);  
 ++n;  
 }  
 return 0;  
}

使用while循环不断增加n的值, 直到f(n)报错栈溢出

f函数的时间复杂度为O(n)

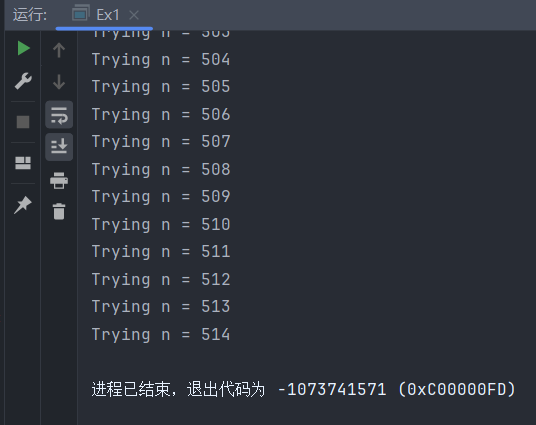


当n到达43340时栈溢出了, 最大递归深度为43339

#### (2)在递归中创建一些无用的变量

在递归函数中创建大小为1000的int数组

#include <bits/stdc++.h>  
  
using namespace std;  
  
int f(int n) {  
 int a[1000];  
 if (n == 1) return 1;  
 return n + f(n - 1);  
}  
  
int main() {  
 int n = 1;  
 while (true) {  
 cout << "Trying n = " << n << endl;  
 f(n);  
 ++n;  
 }  
 return 0;  
}



这次n到达514就栈溢出了, 最大递归深度变为了513

### 第二题：二叉树深度

求二叉树中每个结点的深度，递归的代码如下：

int depth(BNode \*bt){

if(bt==nullptr) return 0;

bt->depth = max(depth(bt->lchild),depth(bt->rchild))+1;

return bt->depth;

}

**请用非递归方法构造两种退化的二叉树**：一种是完全退化的树，除叶结点外，所有结点只有左子树；一种是类似完全退化的树，所有右子树都是叶结点，除叶结点外，所有结点都有左子树和右子树。这两种树如图所示：

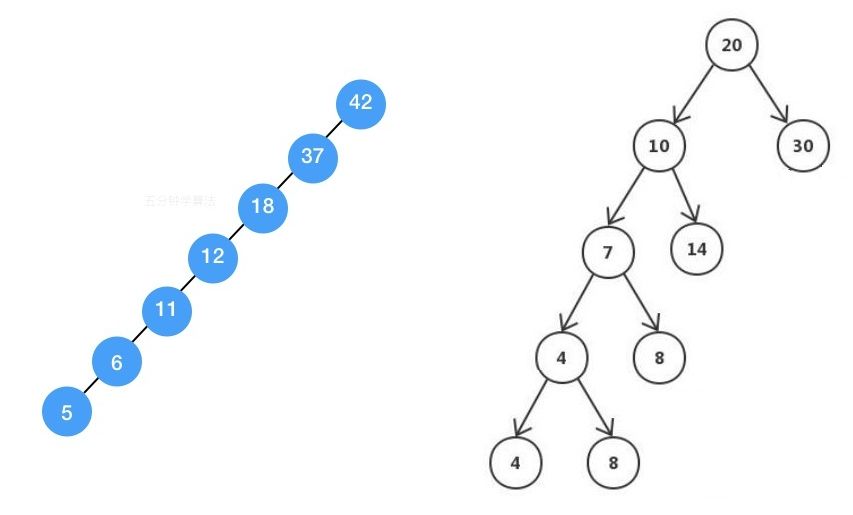


图4.1 两种退化的树示意图

两种树的深度为100000，测试递归可以到达的深度；然后把递归程序改为非递归实现，采用STL的stack来实现。

#### (1) 构造两种退化的树

#define TREE\_DEPTH 100000

struct BNode {  
 int id;  
 int depth;  
 BNode \*lchild;  
 BNode \*rchild;  
};

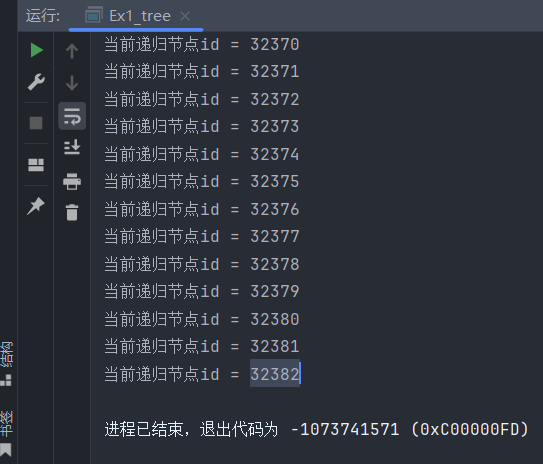
BNode \*createTree1() {  
 BNode \*root = new BNode{0, 0, nullptr, nullptr};  
 BNode \*current = root;  
 for (int i = 1; i < TREE\_DEPTH; ++i) {  
 // 只有左子树  
 current->lchild = new BNode{i, 0, nullptr, nullptr};  
 current = current->lchild;  
 }  
 return root;  
}  
  
BNode \*createTree2() {  
 BNode \*root = new BNode{0, 0, nullptr, nullptr};  
 BNode \*current = root;  
 for (int i = 1; i < TREE\_DEPTH; ++i) {  
 // 右子树为叶子节点  
 current->lchild = new BNode{i, 0, nullptr, nullptr};  
 current->rchild = new BNode{i + TREE\_DEPTH, 0, nullptr, nullptr};  
 current = current->lchild;  
 }  
 return root;  
}

#### (2) 使用递归计算最大递归深度

int depth(BNode \*node) {  
 if (!node) return 0;  
 cout << "当前递归节点id = " << node->id << endl;  
 node->depth = max(depth(node->lchild), depth(node->rchild)) + 1;  
 cout << "节点 " << node->id << " 深度为" << node->depth << endl;  
 return node->depth;  
}

depth函数的时间复杂度为O(n), n为节点个数

第一种退化的树没有右子树, 递归左子树最深到达32382



第二种退化的树右子树只有叶子节点, 叶子节点编号从100001开始, 递归深度达到32365



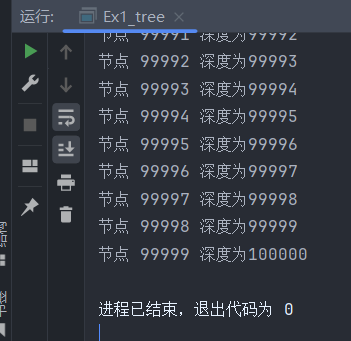
所以最大递归深度大约在32300左右

#### (3) 使用栈代替递归计算每个节点的深度

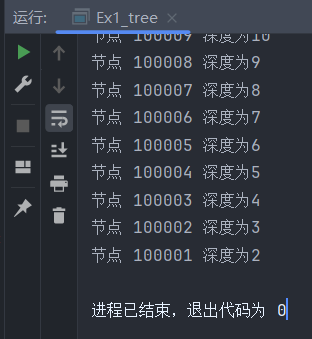
void calculateDepthNonRecursive(BNode \*root) {  
 if (!root) return;  
 stack<pair<BNode \*, int>> stk;  
 stk.emplace(root, 1);  
 while (!stk.empty()) {  
 auto [node, depth] = stk.top();  
 stk.pop();  
 node->depth = depth;  
 cout << "节点 " << node->id << " 深度为" << node->depth << endl;  
 if (node->rchild) stk.emplace(node->rchild, depth + 1);  
 if (node->lchild) stk.emplace(node->lchild, depth + 1); // 左子树先出栈  
 }  
}

depth函数的时间复杂度为O(n), n为节点个数

第一种树, 成功计算出每个节点的深度



第一种树, 成功计算出每个节点的深度, 因为每次是左子树先出栈, 遍历的顺序是先序遍历, 最后计算的是右子树



## 六、实验数据及处理结果

在本次实验中，首先通过实现递归函数f(n)=n+f(n-1)来测试递归深度。我们发现当没有额外变量时，最大递归深度为43339；而当在递归函数中添加一个大小为1000的int数组作为无用变量时，最大递归深度降至513。这显著表明额外的局部变量会占用更多的栈空间，从而减少可达到的最大递归深度。

对于二叉树的深度计算，构造了两种退化的二叉树：一种是只有左子树的完全退化树，另一种是右子树都是叶节点且所有非叶节点都有左右子树的类似完全退化树。在递归方法中，这两种树的最大递归深度分别约为32382和32365。使用非递归方法（即栈实现）成功计算出每个节点的深度，显示非递归方法可以有效避免递归深度限制的问题。

## 七、思考讨论题或体会或对改进实验的建议

实验体会:

通过这次实验，我深刻体会到了递归与非递归实现的区别及其应用场景。递归虽然代码简洁，易于理解，但受限于系统栈的大小，不能处理太深的递归。而非递归实现，尤其是使用栈的模拟方式，虽然代码相对复杂，但能够处理更深的结构，更适合大规模数据处理。

对于改进实验的建议：

1. 可以考虑引入更多类型的递归问题，如斐波那契数列、树的遍历等，以加深对递归的理解和应用。
2. 在实验指导书中增加关于如何优化递归函数以避免栈溢出的章节，例如通过尾递归优化或转换为迭代方法。
3. 提供更多关于内存管理和栈操作的深入讨论，帮助学生更好地理解底层机制。

通过这些改进，可以帮助学生更全面地掌握递归和非递归的知识，提高编程能力和问题解决能力。

## 八、参考资料

《算法设计与分析》实验指导书

# 实验二 分治法

学生姓名：马星 学 号：5418122020专业班级：计算机科学与技术(卓越工程师)221班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期： 2024/12/12 实验成绩：

## 一、实验项目名称

分治法

## 二、实验目的

1．掌握分治法的基本思想

2．掌握相关问题（求最大最小元、二分搜索、排序问题、选择问题等）的分治算法

## 三、实验基本原理

1．分治法的基本思想

SolutionType DandC(ProblemType P) {

ProblemType P1，P2，…，Pk;

if(Small(P))

return S(P);

else {

Divide(P，P1，P2，…，Pk);

Return Combine(DandC(P1)，DandC(P2)，…，DandC(Pk));

}

}

2．一个问题能够用分治法求解的要素

（1）问题能够按照某种方式分解成若干个规模较小、相互独立且与原问题类型相同的子问题；

（2）子问题足够小时可以直接求解；

（3）能够将子问题的解组合成原问题的解。由于分治法要求分解成同类子问题，并允许不断分解，使问题规模逐步减小，最终可用已知的方法求解足够小的问题，因此，分治法求解很自然导致一个递归算法。

## 四、主要仪器设备及耗材

PC微机

DOS操作系统或 Windows 操作系统

CLion 集成开发环境

## 五、实验步骤（完整内容见光盘）

### 第1题：改写分治算法

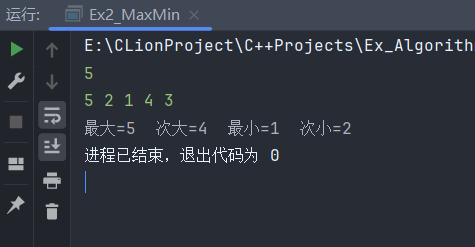
请改写教材中的分治算法MaxMin（p91），使其可以输出n个数中的最大值和次最大值以及最小值和次最小值（注意：不能用排序）。

输入：第一行输入数的个数n，第二行输入n个数。

输出：第一行输出最大值和次最大值，第二行输出最小值和次最小值。

#include <bits/stdc++.h>  
  
#define inf INT32\_MAX  
using namespace std;  
  
void MaxMin(const vector<int> &a, int l, int r, int &max1, int &max2, int &min1, int &min2) {  
 if (l == r) {// 只有1个数  
 max1 = a[l];  
 max2 = -inf;  
 min1 = a[l];  
 min2 = inf;  
 } else if (l + 1 == r) {// 有两个数  
 max1 = max(a[l], a[r]);  
 max2 = min(a[l], a[r]);  
 min1 = min(a[l], a[r]);  
 min2 = max(a[l], a[r]);  
 } else {// 有两个以上的数  
 // 求左右区间各自的最大、次大、最小、次小  
 int mid = l + ((r - l) >> 1);  
 int Lmax1, Lmax2, Lmin1, Lmin2;  
 int Rmax1, Rmax2, Rmin1, Rmin2;  
 MaxMin(a, l, mid, Lmax1, Lmax2, Lmin1, Lmin2);  
 MaxMin(a, mid + 1, r, Rmax1, Rmax2, Rmin1, Rmin2);  
 max1 = max(Lmax1, Rmax1);// 最大 = max(左最大,右最大)  
 min1 = min(Lmin1, Rmin1);// 最小 = min(左最小,右最小)  
 if (Lmax1 > Rmax1) {// Lmax1最大, 次大为左次大或右最大  
 max2 = max(Lmax2, Rmax1);  
 } else {// Rmax1最大, 次大为左最大或右次大  
 max2 = max(Lmax1, Rmax2);  
 }  
 if (Lmin1 < Rmin1) {// 同理  
 min2 = min(Lmin2, Rmin1);  
 } else {  
 min2 = min(Lmin1, Rmin2);  
 }  
 }  
}  
  
/\*  
5  
5 2 1 4 3  
 \*/  
int main() {  
 int n;  
 cin >> n;  
 vector<int> a(n);  
 for (int i = 0; i < n; i++)cin >> a[i];  
 int max1, max2, min1, min2;  
 MaxMin(a, 0, n - 1, max1, max2, min1, min2);  
 cout << "最大=" << max1 << " 次大=" << max2 << " 最小=" << min1 << " 次小=" << min2;  
 return 0;  
}

运行结果:



### 第2题：求第K小

用分治法编程解决在n个数当中找第K小元素问题（注意：不能用排序）。

输入：第一行输入n的值，第二行输入n个数，第三行输入K的值。

输出：n个数中的第K小元素。

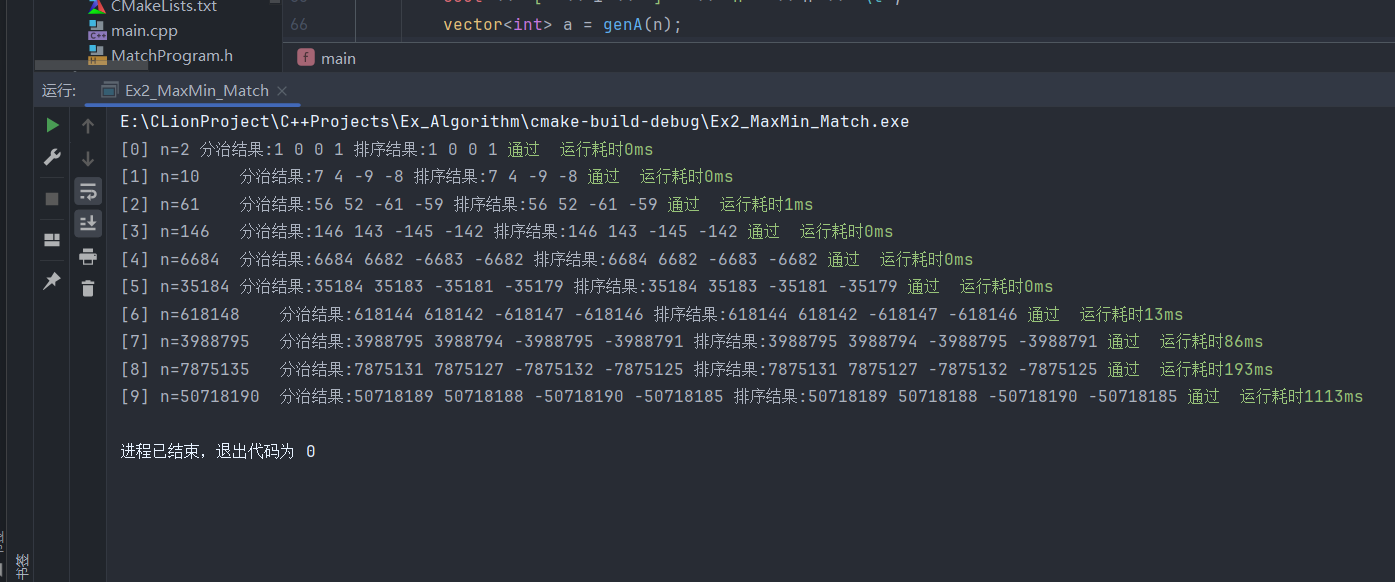
#include <bits/stdc++.h>  
  
using namespace std;  
  
/\*\*  
 快排-随机基准点分区  
  
 @param a 排序数组  
 @param left 区间左界索引  
 @param right 区间右界索引  
 @return 基准点索引  
 \*/  
int partition(vector<int> &a, int left, int right) {  
 int index = rand() % (right - left + 1) + left;  
 swap(a[index], a[left]);  
 int pv = a[left];//基准点元素值  
 int i = left, j = right;//双边快排   
 while (i < j) {  
 while (i < j && a[j] > pv) {  
 j--;//寻找小的(或等于)  
 }  
 while (i < j && a[i] <= pv) {  
 i++;//寻找大的  
 }  
 swap(a[i], a[j]);//找到则交换  
 }  
 //退出循环时,i==j,指向的是最右侧的小于等于基准值的元素  
 swap(a[left], a[i]);//将基准值交换过来  
 return i;  
}  
  
int quick(vector<int> &a, int left, int right, int k) {  
 int p = partition(a, left, right);//基准点元素的索引  
 if (p == k) {//选择的基准点就是排第i的元素  
 return a[p];  
 } else if (k < p) {//在左半区  
 return quick(a, left, p - 1, k);  
 } else {//在右半区  
 return quick(a, p + 1, right, k);  
 }  
}  
  
  
/\*\*   
 使用快速排序思想,求sorted(a)[k]  
 \*/  
int choose(vector<int> &a, int k) {  
 return quick(a, 0, a.size() - 1, k);  
}  
  
/\*  
5 2  
5 2 1 4 3  
 \*/  
int main() {  
 int n, k;  
 cin >> n >> k;  
 vector<int> a(n);  
 for (int i = 0; i < n; i++)cin >> a[i];  
 cout << "第" << k << "小的元素为" << choose(a, k - 1);// 第几 ---> 索引  
 return 0;  
}

## 六、实验数据及处理结果

### 第一题 改写分治算法(数据对拍):

#include <bits/stdc++.h>  
  
#define inf INT32\_MAX  
#define NONE "\e[0m"  
#define RED "\e[31m"  
#define GREEN "\e[32m"  
using namespace std;  
  
void MaxMin(const vector<int> &a, int l, int r, int &max1, int &max2, int &min1, int &min2){...}

vector<int> test(const vector<int> &a) {  
 int max1, max2, min1, min2;  
 MaxMin(a, 0, a.size() - 1, max1, max2, min1, min2);  
 return **{**max1, max2, min1, min2**}**;  
}  
  
int randInt(int a, int b) {  
 int r = (rand() << 16) | rand();  
 return r % (b - a + 1) + a;  
}  
  
vector<int> genA(int n) {// 生成长度为n的a数组  
 vector<int> a(n);  
 for (int i = 0; i < n; i++)a[i] = randInt(-n \* 10, n \* 10);  
 return a;  
}  
   
  
int main() {  
 srand(time(nullptr));  
 vector<int> ns = **{**2, 10, 61, 146, 6684, 35184, 618148, 3988795, 7875135, 50718190**}**;  
 for (int i = 0; i < ns.size(); i++) {  
 int n = ns[i];  
 cout << "[" << i << "] \t" << "n=" << n << "\t";  
 vector<int> a = genA(n);  
 // 分治计算  
 auto start = clock();  
 const vector<int> &calAns = test(a);  
 auto end = clock();  
 // 排序计算  
 sort(a.begin(), a.end());  
 const vector<int> &ans = **{**a[n - 1], a[n - 2], a[0], a[1]**}**;  
 // 比对结果  
 cout << "分治结果:";  
 for (int k: calAns) cout << k << " ";  
 cout << "排序结果:";  
 for (int k: ans) cout << k << " ";  
 auto useTime = (double) 1000 \* (end - start) / CLOCKS\_PER\_SEC;  
  
 if (calAns == ans) {  
 cout << GREEN << "通过\t" << "运行耗时" << useTime << "ms" << NONE << endl;  
 } else {  
 cout << RED << "失败\t" << "运行耗时" << useTime << "ms" << NONE << endl;  
 }  
 }  
 return 0;  
}



通过实现的MaxMin函数处理输入向量，并与排序后的结果比对验证正确性。测试了不同大小的数组，结果显示算法能够正确找到最大值、次大值、最小值和次小值。

时间复杂度: T(n) = 2T(n/2), T(1)=1, 时间复杂度O(n)

### 第二题 求第K小(数据对拍):

#include <bits/stdc++.h>  
  
#define NONE "\e[0m"  
#define RED "\e[31m"  
#define GREEN "\e[32m"  
using namespace std;  
  
/\*\*  
 快排-随机基准点分区  
  
 @param a 排序数组  
 @param left 区间左界索引  
 @param right 区间右界索引  
 @return 基准点索引  
 \*/  
int partition(vector<int> &a, int left, int right) {...}  
  
int quick(vector<int> &a, int left, int right, int k) {...}  
  
  
/\*\*  
 使用快速排序思想,求sorted(a)[k]  
 \*/  
int choose(vector<int> &a, int k) {  
 return quick(a, 0, a.size() - 1, k);  
}  
  
int randInt(int a, int b) {  
 int r = (rand() << 16) | rand();  
 return r % (b - a + 1) + a;  
}  
  
vector<int> genA(int n) {// 生成长度为n的a数组  
 vector<int> a(n);  
 for (int i = 0; i < n; i++)a[i] = randInt(-n, n);  
 return a;  
}  
  
/\*  
5 2  
5 2 1 4 3  
 \*/  
int test(vector<int> a, int k) {  
 return choose(a, k - 1);  
}  
  
int main() {  
 srand(time(nullptr));  
 vector<int> ns = **{**1, 10, 61, 146, 6684, 35184, 618148, 3988795, 7875135, 56718190**}**;  
 for (int i = 0; i < ns.size(); i++) {  
 int n = ns[i];  
 int k = randInt(1, n);  
 cout << "[" << i << "] " << "n=" << n << "\t" << "k=" << k << "\t";  
 vector<int> a = genA(n);  
 // 分治计算  
 auto start = clock();  
 int calAns = test(a, k);  
 auto end = clock();  
 // 排序计算  
 sort(a.begin(), a.end());  
 int ans = a[k - 1];  
 // 比对结果  
 cout << "分治结果:" << calAns << "\t";  
 cout << "排序结果:" << ans << "\t";  
 auto useTime = (double) 1000 \* (end - start) / CLOCKS\_PER\_SEC;  
  
 if (calAns == ans) {  
 cout << GREEN << "通过\t" << "运行耗时" << useTime << "ms" << NONE << endl;  
 } else {  
 cout << RED << "失败\t" << "运行耗时" << useTime << "ms" << NONE << endl;  
 }  
 }  
 return 0;  
}



使用快速选择算法寻找第k小的元素，并与排序后的结果进行比较。测试了不同规模的数组和不同的k值，结果均显示算法有效且准确。

时间复杂度: T(n) = T(n/2) + n, T(1)=1, 时间复杂度为O(n)

## 七、思考讨论题或体会或对改进实验的建议

1. 理解到分治法将大问题分解为多个小问题，并递归求解，最后合并结果的策略。
2. 学习到如何通过分治法解决求最大最小元素、二分搜索、排序等问题，以及如何改写现有算法以满足特定需求。
3. 体会到分治法在实际应用中的高效性和实用性，尤其是在处理大数据量时的优势。
4. 认识到编写高效、准确的代码需要深入理解算法原理和数据结构。
5. 建议增加更多实际应用场景和优化技巧的介绍，以提高学生的应用能力和编程技能。

## 八、参考资料

《算法设计与分析》实验指导书

# 实验三 贪心法

学生姓名：马星 学 号：5418122020专业班级：计算机科学与技术(卓越工程师)221班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期： 2024/12/17 实验成绩：

## 一、实验项目名称

贪心法

## 二、实验目的

1．掌握贪心法的基本思想

2．掌握相关问题（背包问题、带时限的作业排序、最佳合并模式、最小代价生成树、单源最短路径、磁带最优存储等）的贪心算法

## 三、实验基本原理

1．贪心法的基本思想

贪心法是通过分步决策（stepwisedecision）的方法来求解问题的。贪心法每一步上用作决策依据的选择准则被称为最优量度标准（optimizationcriterion）。

SolutionTypeGreedy(STypea[],int n) {

Solution

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

SType x=Select(a);

if(Feasiable(solution,x))

solution=Union(solution,x);

}

return solution;

}

2．贪心法的基本要素

（1）最优量度标准

选择最优量度标准是使用贪心法求解问题的核心问题。贪心算法每一步作出的选择可以依赖以前作出的选择，但不依赖将来的选择，也不依赖于子问题的解。对于一个贪心算法，必须证明所采用的量度标准能够导致一个整体最优解。

（2）最优子结构

优子结构特性是关于问题最优解的特性。当一个问题的最优解中包含了子问题的最优解，则称该问题具有最优子结构特性（optimalsubstructure）。一般而言，如果一个最优化问题的解结构具有元组形式，并具有最优子结构特性，我们可以尝试选择量度标准。

## 四、主要仪器设备及耗材

PC微机

DOS操作系统或 Windows 操作系统

CLion 集成开发环境

## 五、实验步骤（完整内容见光盘）

### 第1题 背包问题

已知一个载重为M的背包和n件物品，第i件物品的重量为wi，如果将第i件物品全部装入背包，将有收益pi，这里，wi>0，pi>0，0<i<n。所谓背包问题是指求一种最佳装载方案，使得收益最大。

输入：第一行物品个数n和背包载重M，以下n行输入物品编号i，物品收益pi，物品重量wi。

输出：x1，x2，…，xn，0<xi<1，0<i<n，每个xi是第i件物品装入背包中的部分（小数位保留二位）。

按每单位的收益进行排序, 单位收益高的最优,全部装入, 直到某物品无法全部, 剩下的容量都装该物品, 后面单位收益低的均不装入

#include <bits/stdc++.h>  
  
using namespace std;  
  
struct Item {  
 int id;  
 double w;// 总重量  
 double p;// 全部装入可获得的收入  
};  
  
/\*  
10 3  
0 12 6  
1 6 6  
2 3 6  
  
-> 6.00,4.00,0.00  
 \*/  
int main() {  
 double M;  
 int n;  
 cin >> M >> n;  
 vector<Item> items(n);  
 for (int i = 0; i < n; i++) {  
 cin >> items[i].id >> items[i].p >> items[i].w;  
 }  
 sort(items.begin(), items.end(), [&](const Item &a, const Item &b) {  
 return (a.p / a.w) > (b.p / b.w);  
 });// 按照单位收益从大到小排序

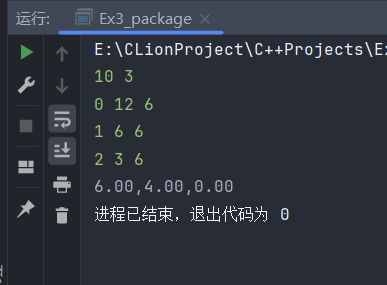
vector<double> ans(n, 0.0);  
for (Item &item: items) {  
 if (item.w <= M) {// 第i个可以全部装入  
 M -= item.w;  
 ans[item.id] = item.w;  
 } else {// 不能全部装入   
 ans[item.id] = M;  
 break;  
 }  
}

for (int i = 0; i < n; i++) {  
 cout << fixed << setprecision(2) << ans[i];  
 if (i != n - 1)cout << ",";  
 }  
  
 return 0;  
}

测试:

背包容量为10 ,共有三种物品, 物品0单位收益=12/6=2, 物品1单位收益=6/6=1, 物品2单位收益=3/6=0.5

按单位收益排序, 物品0总共有6个单位重量, 全部装入, 物品1总共有6个单位重量, 但此时背包容量只剩余4个单位, 所以装入4个单位的物品1



### 第2题 带时限的作业排序问题

设有一个单机系统、无其它资源限制且每个作业运行相等时间，不妨假定每个作业运行1个单位时间。现有n个作业，每个作业都有一个截止期限di>0，di为整数。如果作业能够在截止期限之内完成，可获得pi>0的收益。问题要求得到一种作业调度方案，该方案给出作业的一个子集和该作业子集的一种排列，使得若按照这种排列次序调度作业运行，该子集中的每个作业都能如期完成，并且能够获得最大收益。

输入：第一行输入n的值，以下n行输入作业号i，收益pi，截止期限di。

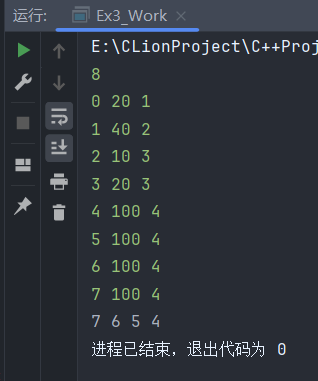
输出：n个作业的一个最优子集。

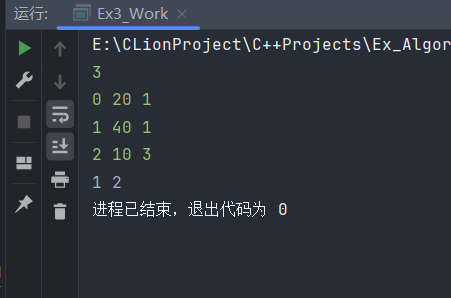
思路:

按照价值排序, 将作业放在它的截止时间的末尾, 如果时间段被占用, 则向前搜索未被占用的时间, 如果全部被占用则放弃这个作业

#include <bits/stdc++.h>  
  
using namespace std;  
  
struct Work {  
 int id;  
 int p;// 收益  
 int d;// 期限  
};  
  
/\*  
8  
0 20 1  
1 40 2  
2 10 3  
3 20 3  
4 100 4  
5 100 4  
6 100 4  
7 100 4  
  
-> 5 6 7  
 \*/  
/\*  
3  
0 20 1  
1 40 1  
2 10 3  
  
-> 1 2  
 \*/  
int main() {  
 int n;  
 cin >> n;  
 vector<Work> works(n);  
 int maxD = 0;  
 for (int i = 0; i < n; i++) {  
 cin >> works[i].id >> works[i].p >> works[i].d;  
 maxD = max(maxD, works[i].d);  
 }  
 sort(works.begin(), works.end(), [&](const Work &a, const Work &b) {  
 return a.p > b.p;  
 });// 按收益降序  
  
 vector<int> use(maxD + 1, -1);//use[i]: [i-1,i]是否占用  
 for (int i = 0; i < n; i++) {  
 Work &work = works[i];  
 // 从work.d向前找第一个没有被占用的时间  
 int t = work.d;  
 while (t >= 1 && use[t] != -1)t--;  
 if (t == 0)continue;// 全部被占用  
 // [t-1,t] 执行作业work  
 use[t] = work.id;  
 }  
 //vector<int> res;  
 for (int i: use) {// 按顺序返回  
 if (i != -1) {  
 //res.push\_back(i);  
 cout << i << " ";  
 }  
 }  
 return 0;  
}

测试:





### 第3题 Minimum-Cost Spanning Tree

一个无向连通图的生成树是一个极小连通子图，它包括图中全部结点，并且有尽可能少的边。一棵生成树的代价是树中各条边上的代价之和。一个网络的各生成树中，具有最小代价的生成树称为该网络的最小代价生成树（minimum-cost spanning tree）。

输入：第一行输入结点个数n和边的个数m，以下m行输入各边的两个结点u、v及该边上的代价。

输出：如果有生成树，则输出最小生成树的代价；如果没有生成树，则输出nospanningtree。

#### (1) Kruskal

##### 1. 过程

对边进行排序, 每次选择最小代价的边. 若边的两个顶点未连通, 则选择这条边; 若边的两个顶点已连通,则不选择这条边. 直到选择了n-1条边, 若选不出n-1条边则说明图不连通

##### 2. 正确性证明

数学归纳法

① n<=2时, 显然成立

②设n=a时 和 n=b 时成立, 则当n=a+b时:

将图划分为 子图A(a) 和 B(b)

∵ n=a时该算法成立

∴ A(a)可以构造最小生成树, 设为TA

同理, B(b) 可以构造最小生成树, 设为TB

TA与TB有若干条边

注意到:

∵ TA是不含回路的, TB也是不含回路的, TA-TB只选择一条边

∴ TA-TB 的其中任意一条边被选择都不会形成回路

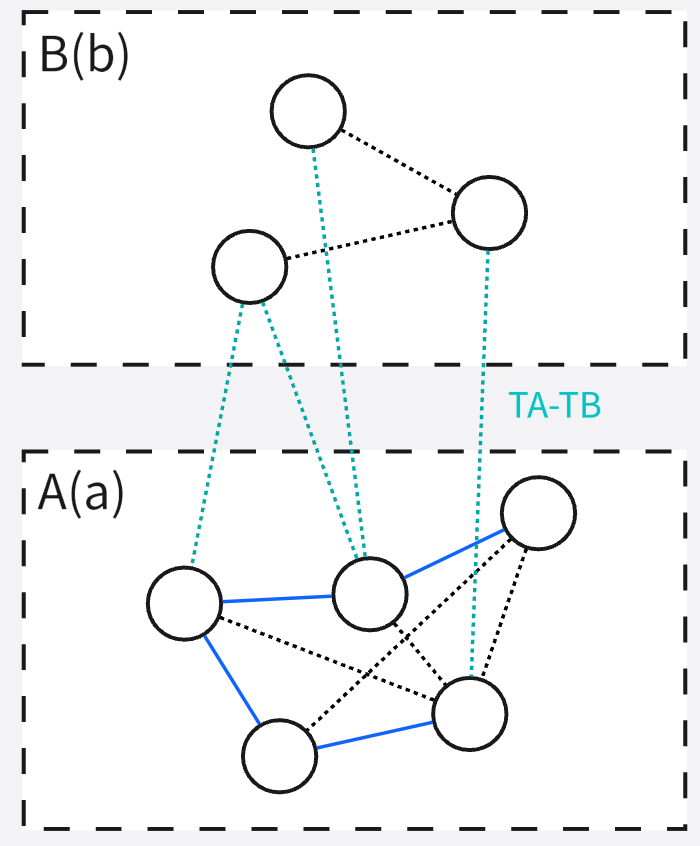
如果选择TA-TB中最短的一条边

∵ TA是最小的, TB也是最小的, TA-TB也是最小的

∴ 选出的树在**全局**上是**最小**的(且**不含回路**)

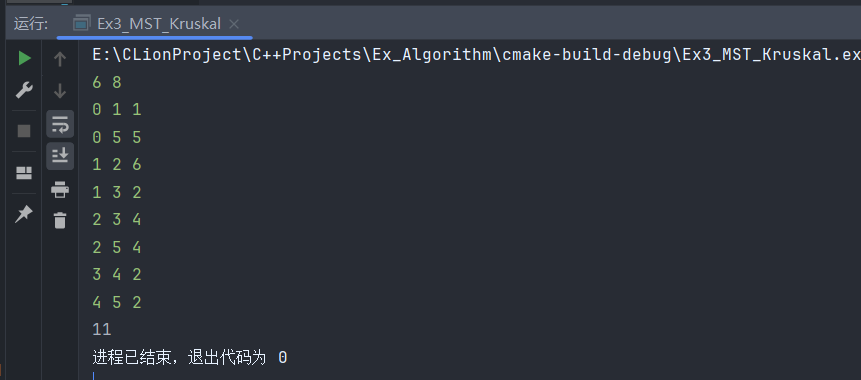
所以 n=a+b 时该算法成立

这种划分与Kruskal算法相同: 每次选择不会形成回路的最短边, 就是在考虑局部的两个子树的合并, 局部最优递推出全局最优



##### 3. 代码

#include <bits/stdc++.h>  
  
using namespace std;  
  
class UnionSet {// 并查集  
public:  
 vector<int> f;  
  
 explicit UnionSet(int n) {  
 f.resize(n + 1);  
 for (int i = 0; i <= n; i++) f[i] = i;  
 }  
  
 int find(int x) {  
 return x == f[x] ? x : (f[x] = find(f[x]));  
 }  
  
 void Union(int x, int y) {  
 f[find(x)] = find(y);  
 }  
  
 bool isConnect(int x, int y) {  
 return find(x) == find(y);  
 }  
};  
  
struct Edge {  
 int u, v, w;  
  
 bool operator<(const Edge &rhs) const {  
 return w > rhs.w;  
 }  
};  
  
/\*  
 2 2  
 v1------ v3 v1 ---- v3  
 1/ 6\ 4/ 2\ 1/ 2\  
v0 v2 v4 --> v0 v2 v4  
 5\ 4\ /2 4\ /2  
 ----- v5 v5  
6 8  
0 1 1  
0 5 5  
1 2 6  
1 3 2  
2 3 4  
2 5 4  
3 4 2  
4 5 2  
  
11  
\*/  
  
int main() {  
 int n, m;  
 cin >> n >> m;  
 priority\_queue<Edge> q;  
 for (int i = 0; i < m; i++) {  
 Edge e{0, 0, 0};  
 cin >> e.u >> e.v >> e.w;  
 q.push(e);// 将所有边入队排序  
 }  
 UnionSet s(n);  
 int ans = 0;  
 int k = 0;// 已选择的边数  
 while (k < n - 1 && !q.empty()) {  
 Edge e = q.top();// 选择最小的边  
 q.pop();  
 int x = e.u, y = e.v, w = e.w;  
 if (s.isConnect(x, y))continue;// 已连通, 跳过  
 s.Union(x, y);// 选择这条边  
 ans += w;  
 k++;  
 }  
 if (k != n - 1) {  
 cout << "nospanningtree";  
 } else {  
 cout << ans;  
 }  
 return 0;  
}



#### (2) Prim

##### 1. 过程

类似于Dijkstra算法

①选择一个顶点为起始顶点

②除起始顶点外, 将所有顶点标记为不可见

③每次从可见的未访问顶点中选择一个距离树最近的点. 若不存在这样的点, 则说明无法构造最小生成树

④选择[上一次被选择的顶点]-[当前被选择的顶点]这条边

⑤根据当前被选择的顶点更新与树的距离

直到n个点均被访问

##### 2. 正确性证明

数学归纳法

① n<=2时, 显然成立

② 设n = k时成立, 则当n = k+1时:

将图划分为 子图A(k) 和 B(1)

∵ n = k时成立

∴ A(k)可以构造最小生成树, 设为T

对于子图B(1), 它与树T有若干条边

注意到:

∵ T是不含回路的, B(1)-T只选择一条边

∴ 选择 B(1)-T 的其中任意一条边都不会形成回路

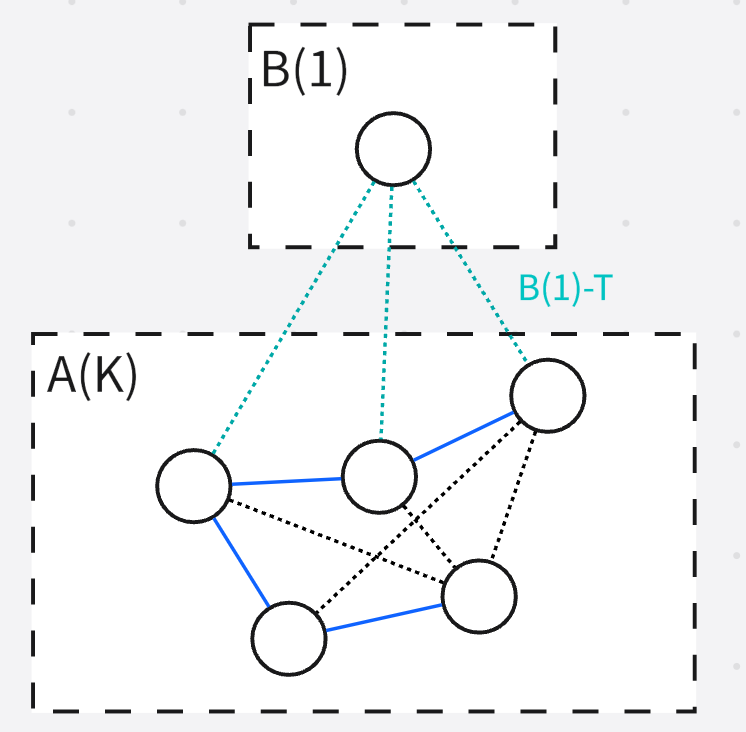
如果选择B(1)-T中最短的一条边

∵ T是最小的, B(1)-T也是最小的

∴ 选出的树在全局上是最小的(且不含回路)

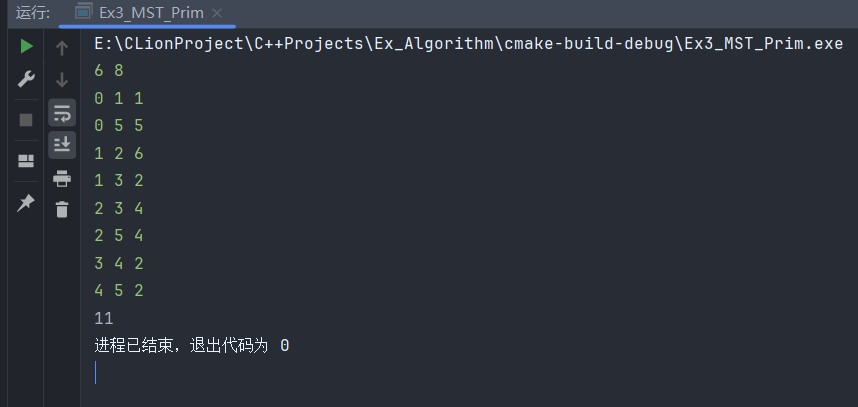
所以 n=k+1 时成立

这种划分与Prim算法相同: 每次选择与树相连的最近节点, 就是在考虑当前子树和树外一个节点的合并, 局部最优递推出全局最优



##### 3. 代码

#include <bits/stdc++.h>  
  
using namespace std;  
  
int main() {  
 int n, m;  
 cin >> n >> m;  
 vector<vector<int>> graph(n, vector<int>(n, INT32\_MAX));  
 for (int i = 0; i < m; i++) {  
 int u, v, w;  
 cin >> u >> v >> w;  
 graph[u][v] = w;  
 graph[v][u] = w;  
 }  
  
 vector<bool> isVisited(n, false); // 节点i是否已选择  
 vector<int> distance(n, INT32\_MAX);// 可见节点到起点0的最短距离  
 distance[0] = 0;  
 int preNode = -1;// 上一个选择的节点  
 int k = 0;// 已选节点数  
 int ans = 0;// 选择边的代价之和  
 while (k < n) {  
 // 选择最近的未访问顶点v  
 int minCost = INT32\_MAX, v = -1;  
 for (int i = 0; i < n; i++) {  
 if (isVisited[i])continue;  
 if (distance[i] < minCost) {  
 minCost = distance[i];  
 v = i;  
 }  
 }  
 if (v == -1) {  
 cout << "nospanningtree";  
 return 0;  
 }  
 isVisited[v] = true;  
 if (k > 0) {  
 ans += graph[preNode][v];  
 }  
 preNode = v;  
 k++;  
 // 更新可访问节点距离  
 for (int i = 0; i < n; i++) {  
 if (isVisited[i])continue;  
 if (graph[v][i] < distance[i]) {  
 distance[i] = graph[v][i];  
 }  
 }  
 }  
 cout << ans;  
 return 0;  
}



## 六、实验数据及处理结果

### 第1题 背包问题

排序是O(nlogn)的, z遍历Item进行装入是O(n)的, 整体时间复杂度为O(nlogn)

本题数据为浮点数无法进行枚举和划分, 无法对拍

### 第2题 带时限的作业排序问题

排序是O(nlogn)的

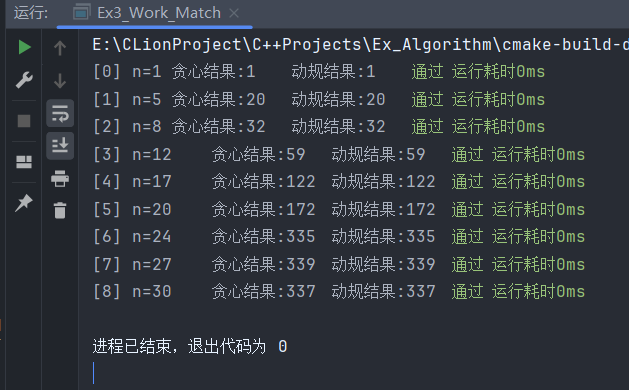
扫描每个作业为O(n), 其中需要为每个作业寻找未占用时间, 最坏情况下每个作业时间都相同, 需要1+2+...n次查找, 所以为O(n^2)

整体时间复杂度为O(n^2)

数据对拍: 使用暴力搜索, 每个作业都有做与不做两种选择, 按照时限排序作业,枚举每个作业的状态(如果已经做了k次作业, 则期限<=k的作业无法再做), 对拍程序时间复杂度O(2^n)

#include <bits/stdc++.h>  
#include <ostream>  
  
#define NONE "\e[0m"  
#define RED "\e[31m"  
#define GREEN "\e[32m"  
using namespace std;  
  
struct Work {  
 int id;  
 int p;// 收益  
 int d;// 期限  
 friend ostream &operator<<(ostream &os, const Work &work) {  
 os << "{id:" << work.id << " p:" << work.p << " d:" << work.d << "}";  
 return os;  
 }  
};  
  
/\*  
8  
0 20 1  
1 40 2  
2 10 3  
3 20 3  
4 100 4  
5 100 4  
6 100 4  
7 100 4  
  
-> 5 6 7  
 \*/  
/\*  
3  
0 20 1  
1 40 1  
2 10 3  
  
-> 1 2  
 \*/  
vector<Work> test(vector<Work> &works) {  
 int maxD = 0;  
 for (auto & work : works) {  
 maxD = max(maxD, work.d);  
 }  
 sort(works.begin(), works.end(), [&](const Work &a, const Work &b) {  
 return a.p > b.p;  
 });// 按收益降序  
  
 vector<Work \*> use(maxD + 1, nullptr);//use[i]: [i-1,i]是否占用  
 for (auto & work : works) {  
 // 从work.d向前找第一个没有被占用的时间  
 int t = work.d;  
 while (t >= 1 && use[t])t--;  
 if (t == 0)continue;// 全部被占用  
 // [t-1,t] 执行作业work  
 use[t] = &work;  
  
 }  
 vector<Work> res;  
 for (Work \*work: use) {  
 if (work)res.push\_back(\*work);  
 }  
 return res;  
}  
  
int randInt(int a, int b) {  
 int r = (rand() << 16) | rand();  
 return r % (b - a + 1) + a;  
}  
  
vector<Work> genWork(int n) {  
 vector<Work> works(n);  
 for (int i = 0; i < n; i++) {  
 works[i] = {i, randInt(1, n), randInt(1, n)};  
 }  
 return works;  
}  
  
void f(vector<Work> &works, int i, int k, int &value, vector<Work> &curr, int &ansValue, vector<Work> &ans) {  
 if (value > ansValue) {  
 ans = curr;  
 ansValue = value;  
 }  
 if (i == works.size()) return;  
 Work &work = works[i];  
 // (1) 不做  
 f(works, i + 1, k, value, curr, ansValue, ans);  
 // (2) 做  
 if (k < work.d) {  
 curr.push\_back(work);  
 value += work.p;  
 f(works, i + 1, k + 1, value, curr, ansValue, ans);  
 value -= work.p;  
 curr.pop\_back();  
 }  
}  
  
vector<Work> dp(vector<Work> &works) {  
 sort(works.begin(), works.end(), [&](const Work &a, const Work &b) {  
 if (a.d == b.d) {  
 if (a.p == b.p)return a.id < b.id;  
 return a.p > b.p;  
 }  
 return a.d < b.d;  
 });// 期限升序排序, 期限相同, 按收益大到小排序  
  
 vector<Work> ans;  
 vector<Work> curr;  
 int value = 0, ansValue = 0;  
 f(works, 0, 0, value, curr, ansValue, ans);  
 return ans;  
}  
  
  
long sum(vector<Work> &ans) {  
 int s = 0;  
 for (int i = 0; i < ans.size(); i++) {  
 Work &work = ans[i];  
 if (i < work.d)s += work.p;  
 }  
 return s;  
}  
  
int main() {  
 srand(time(nullptr));  
 vector<int> ns = **{**1, 5, 7, 10, 14, 17, 20, 25, 28**}**;  
 for (int i = 0; i < ns.size(); i++) {  
 int n = ns[i];  
 cout << "[" << i << "] " << "n=" << n << "\t";  
 vector<Work> works = genWork(n);  
 // 贪心计算  
 auto start = clock();  
 vector<Work> calAns = test(works);  
 auto end = clock();  
 // 动规计算  
 vector<Work> ans = dp(works);  
 // 比对结果  
 cout << "贪心结果:";  
 long calSum = sum(calAns);  
 cout << calSum << "\t";  
 cout << "动规结果:";  
 long ansSum = sum(ans);  
 cout << ansSum << "\t";  
 auto useTime = (double) 1000 \* (end - start) / CLOCKS\_PER\_SEC;  
  
 if (calSum == ansSum) {  
 cout << GREEN << "通过\t" << "运行耗时" << useTime << "ms" << NONE << endl;  
 } else {  
 cout << RED << "失败\t" << "运行耗时" << useTime << "ms" << NONE << endl;  
 }  
 }  
 return 0;  
}

由于对拍程序是O(2^n), n的取值无法太大, 最大只能取log(1e9) ~= 30



### 第3题 Minimum-Cost Spanning Tree

#### (1) Kruskal

对边排序时间复杂度O(ElogE), 算法需要遍历所有边O(E)

在对一条边判断两端点连通时需要调用并查集的find函数, find函数的时间复杂度为α(V), 增长速度极为缓慢, 时间忽略不计

整体时间复杂度为O(ElogE)

#### (2) Prim

需要选择V个顶点, 对于每个顶点需要遍历它的邻居节点, 最坏情况下每对顶点都相连, 整体时间复杂度O(V^2)

可以根据图的稠密度选择使用Kruskal还是Prim算法

## 七、思考讨论题或体会或对改进实验的建议

#### (1) 思考讨论题或体会

通过这次实验，我深刻体会到了贪心法的基本思想和其在实际问题中的应用。在解决背包问题、带时限的作业排序以及最小代价生成树等问题时，贪心算法展示了其高效性与实用性。以下是具体的体会：

**1. 贪心法的基本思想**

* 贪心法通过分步决策的方法来求解问题，每一步的选择基于当前的最优量度标准。这种方法在处理一些复杂问题时非常有效，因为它能够将大问题分解成多个小问题，并逐步求解。

**2. 贪心法的应用**

* 在背包问题中，通过单位收益对物品进行排序，然后依次选择收益最高的物品装入背包，直到背包装满。这种方法简单且易于实现，但在处理大规模数据时可能不够高效。
* 带时限的作业排序问题中，通过按价值排序和向前搜索未被占用的时间，实现了高效的作业调度方案。这种方法在实际应用中非常有用，尤其是在资源有限的情况下。
* 最小代价生成树问题中，分别采用了Kruskal和Prim算法来实现。Kruskal算法通过边排序选择最小代价的边，而Prim算法则从起始顶点开始，逐步扩展生成树。这两种方法各有优缺点，但都能有效地求解最小代价生成树问题。

**3. 贪心法的局限性**

* 贪心法的一个主要局限是它依赖于局部最优解的选择。在某些情况下，贪心选择可能会导致全局最优解的丢失。例如，在背包问题中，如果按照单位收益对物品进行排序后，某些高价值物品被优先选择，可能导致低价值但重量较大的物品无法装入背包。
* 另一个局限是贪心法在处理复杂约束条件的问题时可能不够灵活。例如，在带时限的作业排序问题中，如果作业之间的依赖关系较为复杂，简单的贪心策略可能无法找到最优解。

#### (2) 对改进实验的建议

为了更全面地掌握贪心法及其应用，提出以下改进建议：

**1. 引入更多实际应用场景**

* 可以增加更多类型的贪心算法问题，如斐波那契数列计算、图的遍历等。这些实际应用场景能够帮助学生更好地理解贪心法的适用性和局限性。

**2. 优化贪心算法的性能**

* 在实验指导书中增加关于如何优化贪心算法以避免局部最优解的章节。例如，可以通过动态规划、回溯法等方法结合贪心策略，提高算法的全局最优性。
* 提供关于内存管理和优化技巧的深入讨论，帮助学生更好地理解和实现高效的贪心算法。

**3. 提供更多案例分析与讨论**

* 在课程中增加更多的案例分析和讨论环节，让学生有机会动手实践并分享他们的解决方案。通过这种方式，学生可以从他人的解决方案中学到更多经验和技巧。
* 鼓励学生进行团队合作项目，利用贪心法解决复杂的实际问题。这样可以培养他们的团队协作能力和实际应用能力。

综上所述，通过引入更多实际应用场景、优化算法性能并提供深入的案例分析与讨论，可以帮助学生更全面地掌握贪心法及其应用，提高他们的编程能力和问题解决能力

## 八、参考资料

《算法设计与分析》实验指导书

# 实验四 动态规划法

学生姓名：马星 学 号：5418122020专业班级：计算机科学与技术(卓越工程师)221班

实验类型：□ 验证 □ 综合 □ 设计 □ 创新 实验日期： 2024/12/17 实验成绩：

## 一、实验项目名称

动态规划法

## 二、实验目的

1．掌握动态规划法的基本思想

2．掌握相关问题（多段图、矩阵连乘、最长公共子序列、最优二叉搜索树、0/1背包、流水作业调度等）的动态规划算法

## 三、实验基本原理

1．动态规划的基本思想

动态规划法的实质也是将较大问题分解为较小的同类子问题，这一点上它与分治法和贪心法类似。但动态规划法有自己的特点。分治法的子问题相互独立，相同的子问题被重复计算，动态规划法解决这种子问题重叠现象。贪心法要求针对问题设计最优量度标准，但这在很多情况下并不容易。动态规划法利用最优子结构，自底向上从子问题的最优解逐步构造出整个问题的最优解，动态规划则可以处理不具备贪心准则的问题。

2．最优性原理

最优性原理指出，一个最优策略具有这样的性质，不论过去状态和决策如何，对前面的决策所形成的状态而言，其余决策必定构成最优策略。这便是最优决策序列的最优子结构特性。

3．动态规划算法的步骤

（1）刻画最优解的结构特性；

（2）递归定义最优解值；

（3）以自底向上方式计算最优解值；

（4）根据计算得到的信息构造一个最优解。

其中，第（1）至（3）步是动态规划算法的基本步骤。最优解值是最优解的目标函数的值。

## 四、主要仪器设备及耗材

PC微机

DOS操作系统或 Windows 操作系统

CLion 集成开发环境

## 五、实验步骤（完整内容见光盘）

### 第1题 多段图问题

多段图G=(V,E)是一个带权有向图，它具有如下特性：图中的结点被划分成k>=2个互不相交的子集Vi，1<=i<=k。其中V1和Vk分别只有一个结点，V1包含源点（source）s，Vk包含汇点（sink）t。对所有边<u,v>属于E，多段图要求若u属于Vi，则v属于V＋，1<=i<k，每条边的权值为c(u,v)。从s到t的路径长度是这条路径上边的权值之和，多段图问题（multi stage graph problem）是求从s到t的一条长度最短的路径。

输入：第一行输入结点个数n和边的个数m，以下m行输入各边的两个结点u、v及该边上的代价。

输出：从s到t的最短的路径的长度。

#### 思路:

(1) dp定义: dis[i]表示s到i的最短路径长度, 则dis[t]为最终答案

(2) 初态: 初始距离全为INF, dis[s]为0

(3) 如果有u→v, 则dis[v] = min(dis[v], dis[u] + g[u][v])

因为输入的点是多段有序的, 所以在计算v的时候, u一定已经计算过了

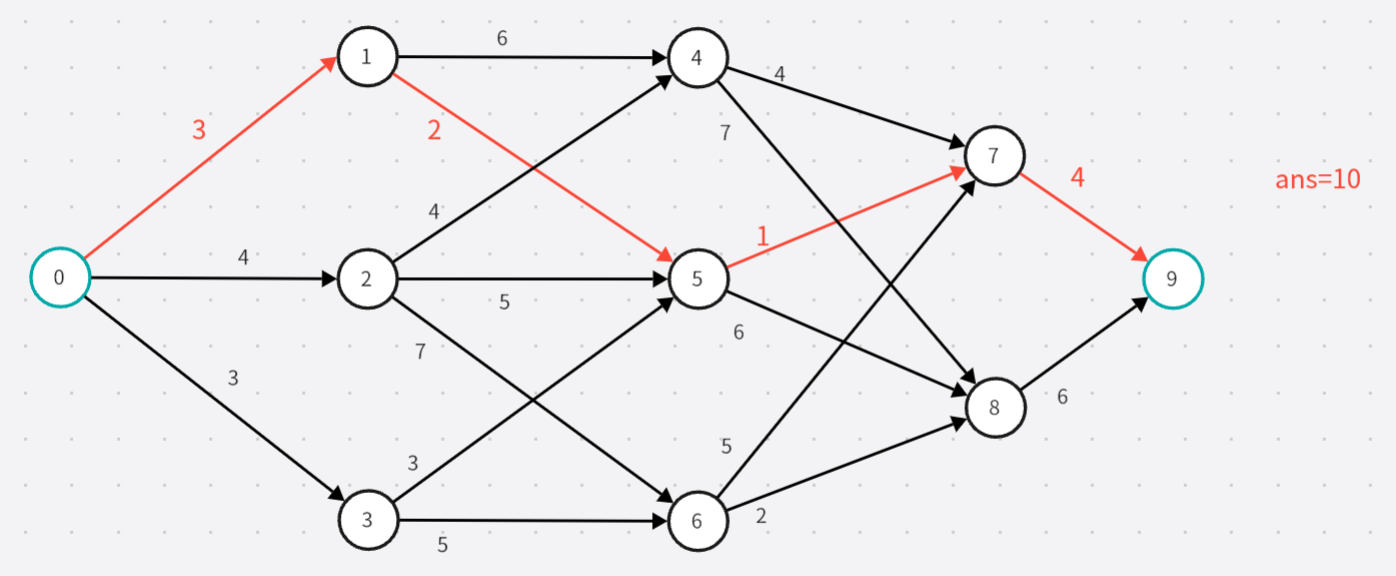
所以双重循环, 外层遍历v=1~n, 内层遍历u=0~v, 计算min(dis[v], dis[u] + g[u][v])

#### 代码:

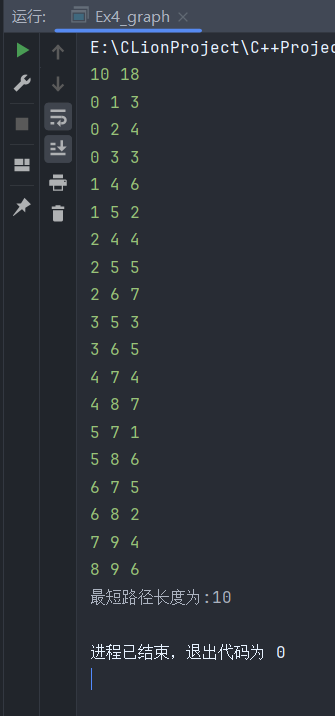
#include <bits/stdc++.h>  
  
using namespace std;  
  
#define inf 0x3f3f3f3f  
   
int main() {  
 int n, m;  
 cin >> n >> m;  
 vector<vector<int>> g(n, vector<int>(n, inf));  
  
 for (int i = 0; i < m; i++) {  
 int u, v, w;  
 cin >> u >> v >> w;  
 g[u][v] = w;  
 }  
  
 vector<int> dis(n, inf);//dis[i]:顶点i距离起点0的最短路径长度  
 dis[0] = 0;  
 //dp  
 for (int v = 1; v < n; v++) {  
 for (int u = 0; u < v; u++) {  
 if (dis[u] + g[u][v] < dis[v]) {  
 dis[v] = dis[u] + g[u][v];  
 }  
 }  
 }  
 cout << "最短路径长度为:" << dis[n - 1] << endl;  
  
 return 0;  
}

#### 运行结果:

输入:

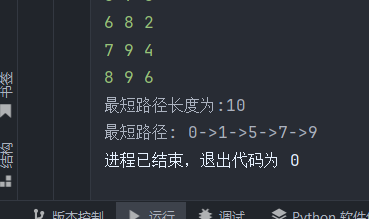


输出:



如果在取min操作时记录path路径path[v] = u, 最后从终点回溯路径, 即可得到最短路径

int t = n - 1;  
string P = to\_string(t);  
while (path[t] != -1) {  
 P = to\_string(path[t]) + "->" + P;  
 t = path[t];  
}  
cout << "最短路径: "<<P;



### 第2题 矩阵连乘问题

给定n个矩阵{A0,A1,…,An-1}，其中Ai，i=0,…,n-1的维数为pi\*pi+1，并且Ai与Ai+1是可乘的。考察这n个矩阵的连乘积A0A1…An-1，由于矩阵乘法满足结合律，所以计算矩阵的连乘可有许多不同的计算次序。矩阵连乘问题是确定计算矩阵连乘积的计算次序，使得按照这一次序计算矩阵连乘积，需要的“数乘”次数最少。

输入：第一行输入n的值，第二行输入n个矩阵的维数pi（i=0，…，n）。

输出：最少乘法次数。

#### 思路:

两个矩阵相乘, 设A\*B, A为m\*p, B为p\*n, 则经过m\*p\*n次数乘后会变为m\*n的矩阵

所以{A0,A1,…,An-1}的矩阵连乘, 对于它的维数数组[p0,p1,...pn], 等价于在[1,n-1]上执行n-1次删除操作, 删除pi的代价为p[i-1]\*p[i]\*p[i+1]

(1) dp定义: dp[i][j]表示删除(i,j)需要的最少代价, 则最终答案为dp[0][n]

(2) 初态: 当(i,j)区间不超过2个数时不需要操作,代价为0, 其余情况置为INF

(3) 转移方程: 考虑[i,j]这个区间的最后一次删除, 假设最后剩余的为pk, 则需要先删除(i,k)和(k,j), 最后删除p[k], 总代价为dp[i][k]+dp[k][j]+p[i]\*p[k]\*p[j]

所以枚举k, 进行取min即可

#### 代码:

#include <bits/stdc++.h>  
  
#define inf 0x3f3f3f3f  
using namespace std;  
  
  
int main() {  
 int n;  
 cin >> n;  
 vector<int> p(n + 1);  
 for (int i = 0; i < n + 1; i++)cin >> p[i];  
  
 vector<vector<int>> dp(n + 1, vector<int>(n + 1, inf));  
 for (int i = 0; i <= n; i++) {  
 for (int j = i; j <= n && j < i + 2; j++) {  
 dp[i][j] = 0;  
 }  
 }  
  
 for (int len = 3; len <= n + 1; len++) {// 子区间长度 [i,i+1,...i+len-1] = [i,...j]  
 for (int i = 0; i <= n - len + 1; i++) {  
 int j = i + len - 1;  
 for (int k = i + 1; k < j; k++) {// 枚举k找最优划分  
 dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j] + p[i] \* p[k] \* p[j]);  
 }  
 }  
 }  
 cout << dp[0][n] << endl;  
  
 return 0;  
}

#### 运行结果:

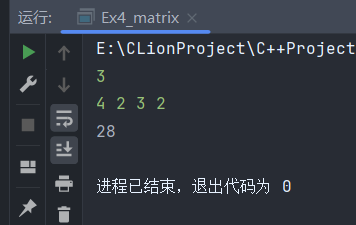
3

4 2 3 2

删除顺序2 3 4\*2\*3 + 4\*3\*2 = 48

删除顺序3 2 2\*3\*2 + 4\*2\*2 = 28

28



4

1 2 3 4 5

删除顺序 2 3 4 1\*2\*3 + 1\*3\*4 + 1\*4\*5 = 38

删除顺序 2 4 3 1\*2\*3 + 3\*4\*5 + 1\*3\*5 = 81

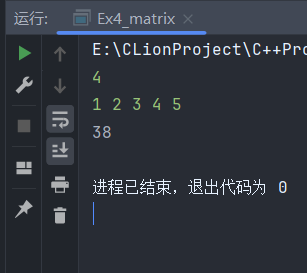
删除顺序 3 2 4 2\*3\*4 + 1\*2\*4 + 1\*4\*5 = 52

删除顺序 3 4 2 2\*3\*4 + 2\*4\*5 + 1\*2\*5 = 74

删除顺序 4 2 3 3\*4\*5 + 1\*2\*3 + 1\*3\*5 = 81

删除顺序 4 3 2 3\*4\*5 + 2\*3\*5 + 1\*2\*5 = 100

38



### 第3题 最长公共子序列问题

给定2个序列X={x1,x2,…,xm}和Y={y1,y2,…,yn}，找出X和Y的最长公共子序列。

输入：第一行输入序列X，第二行输入序列Y。

输出：X和Y的最长公共子序列的长度。

#### 思路:

(1)dp定义: dp[i][j]表示X的前i个字符与Y的前j个字符的最长公共子序列长度, 则最终答案为dp[len(X)][len(Y)]

(2)初态: 初始值均为0

(3)转移方程:

若X[i]==Y[j], 则LCS(X[0,i], Y[0,j]) = LCS(X[0,i-1], Y[0,j-1]) + X[i], 长度dp[i][j]=dp[i-1][j-1]+1

若X[i]!=Y[j], 则LCS(X[0,i], Y[0,j]) = LCS(X[0,i-1], Y[0,j]) || LCS(X[0,i], Y[0,j-1]), 长度dp[i][j]=max(dp[i-1][j],dp[i][j-1])

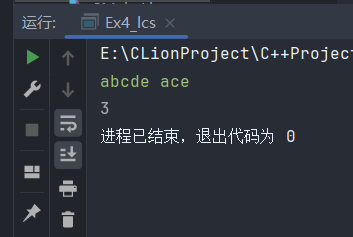
#### 代码:

#include <bits/stdc++.h>  
  
using namespace std;  
  
  
int LCS(string A, string B) {  
 int la = A.length(), lb = B.length();  
  
 vector dp(la + 1, vector<int>(lb + 1, 0));  
 for (int i = 1; i <= la; i++) {  
 for (int j = 1; j <= lb; j++) {  
 if (A[i - 1] == B[j - 1]) {  
 dp[i][j] = dp[i - 1][j - 1] + 1;  
 } else {  
 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);  
 }  
 }  
 }  
 return dp[la][lb];  
}  
  
int main() {  
 string A, B;  
 cin >> A >> B;  
 cout << LCS(A, B);  
 return 0;  
}

#### 运行结果:

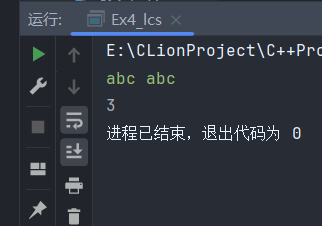
abcde ace

len(ace)=3



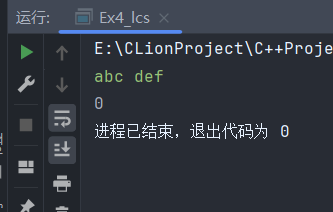
abc abc

len(abc)=3



abc def

0

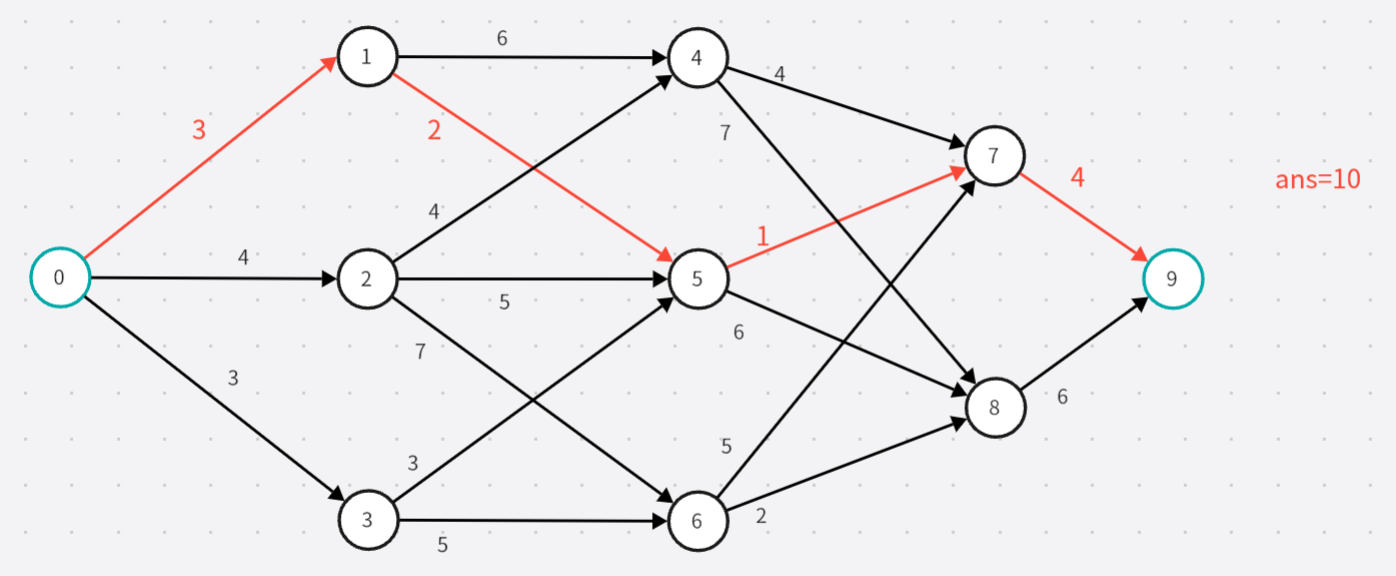
****

## 六、实验数据及处理结果

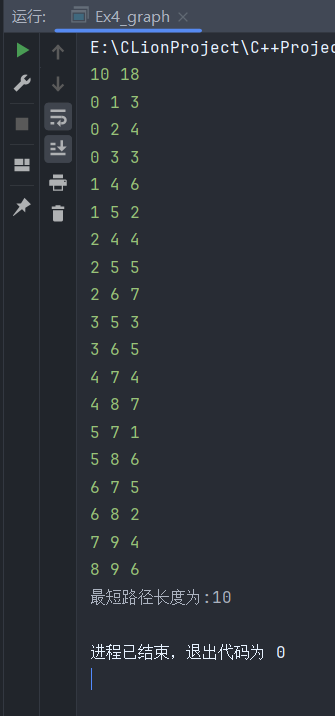
### 第1题 多段图问题

双重循环, 外层遍历每个顶点, 内层遍历前面的顶点, 时间复杂度为O(n^2)

输入:



输出:



### 第2题 矩阵连乘问题

区间dp, 每次枚举一个区间O(n^2), 区间内枚举最优划分, 最终时间复杂度为O(n^3)

输入:

3

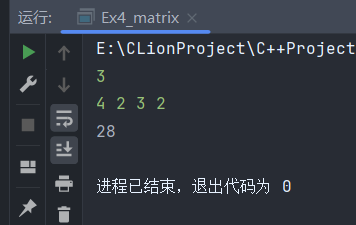
4 2 3 2

删除顺序2 3 4\*2\*3 + 4\*3\*2 = 48

删除顺序3 2 2\*3\*2 + 4\*2\*2 = 28

最终结果应为28

程序输出:



输入:

4

1 2 3 4 5

删除顺序 2 3 4 1\*2\*3 + 1\*3\*4 + 1\*4\*5 = 38

删除顺序 2 4 3 1\*2\*3 + 3\*4\*5 + 1\*3\*5 = 81

删除顺序 3 2 4 2\*3\*4 + 1\*2\*4 + 1\*4\*5 = 52

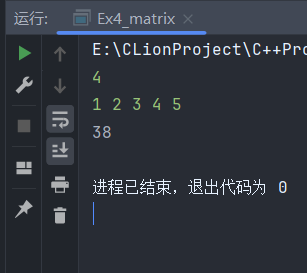
删除顺序 3 4 2 2\*3\*4 + 2\*4\*5 + 1\*2\*5 = 74

删除顺序 4 2 3 3\*4\*5 + 1\*2\*3 + 1\*3\*5 = 81

删除顺序 4 3 2 3\*4\*5 + 2\*3\*5 + 1\*2\*5 = 100

最终结果应为38

程序输出:



### 第3题 最长公共子序列问题

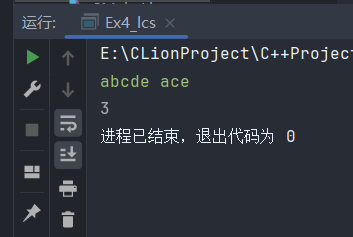
双重循环, 时间复杂度为O(n^2)

输入:

abcde ace

结果应为len(ace)=3

程序输出:

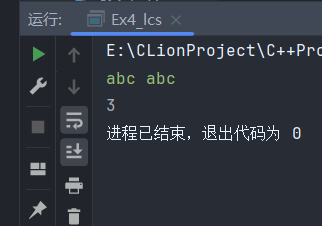


输入:

abc abc

结果应为len(abc)=3

程序输出:

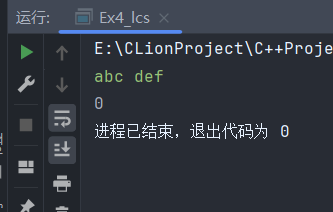


输入:

abc def

结果应为0

程序输出:

****

## 七、思考讨论题或体会或对改进实验的建议

### (1)实验体会：

#### 1. 动态规划的基本思想：

动态规划法通过将较大的问题分解为较小的同类子问题，自顶向下逐步求解。这种方法有效解决了分治法中子问题相互独立的问题，避免了重复计算的缺陷。

在实验中，通过定义最优解的结构特性，以递归方式计算最优解值，最终构造出整个问题的最优解。这体现了动态规划的自底向上求解过程。

#### 2. 动态规划的应用：

多段图问题：在解决多段图最短路径问题时，通过双重循环遍历所有顶点对，利用动态规划的方法计算出从源点到汇点的最短路径长度。这一过程中体现了动态规划算法的时间复杂度和具体实现步骤。

矩阵连乘问题：在处理矩阵连乘问题时，通过区间动态规划的方法，枚举区间内的不同划分方式，找出最少乘法次数的计算次序。这种方法有效地解决了矩阵连乘中的优化问题。

最长公共子序列问题：通过动态规划的定义和转移方程，求解两个序列的最长公共子序列长度。这一过程展现了动态规划在序列匹配问题上的广泛应用。

### (2)改进建议：

#### 1. 引入更多实际应用场景：

增加实际问题类型：可以引入更多类型的动态规划问题，如斐波那契数列计算、图的遍历等。这些实际应用场景能够帮助学生更好地理解动态规划法的适用性和局限性。

提升实践能力：鼓励学生进行团队合作项目，利用动态规划法解决复杂的实际问题。这样可以培养他们的团队协作能力和实际应用能力。

#### 2. 优化算法性能：

提高算法效率：在实验指导书中增加关于如何优化动态规划算法以避免局部最优解的章节。例如，可以通过结合贪心策略或使用更高效的数据结构来提高算法的全局最优性。

深入讨论内存管理：提供关于内存管理和优化技巧的深入讨论，帮助学生更好地理解和实现高效的动态规划算法。

#### 3. 提供更多案例分析与讨论：

增加案例分析环节：在课程中增加更多的案例分析和讨论环节，让学生有机会动手实践并分享他们的解决方案。通过这种方式，学生可以从他人的解决方案中学到更多经验和技巧。

团队合作项目：鼓励学生进行团队合作项目，利用动态规划法解决复杂的实际问题。这样可以培养他们的团队协作能力和实际应用能力。

## 八、参考资料

《算法设计与分析》实验指导书