
华中科技大学

课程实验报告

课程名称： 算法设计与分析

专业班级： 计算机科学与技术校交 1601 班

学 号： U201610504

姓 名： 刘逸帆

指导老师： 赵 峰

报告日期： 2018-11-18

计算机科学与技术学院

目录

一、 实验情况总览	3
二、 解题报告	3
2.1 实验一：最近点对问题	3
2.1.1 实验题目	3
2.1.2 设计思路	3
2.1.3 程序源代码.....	4
2.1.4 运行演示	6
2.2 实验二：大整数加减乘运算	8
2.2.1 实验题目	8
2.2.2 设计思路	8
2.2.3 程序源代码.....	8
2.2.4 运行演示	14
2.3 实验三：最优二分查找树.....	15
2.3.1 实验题目	15
2.3.2 设计思路	15
2.3.3 程序源代码.....	16
2.3.4 运行演示	18
2.4 实验四：FLOYD-WARSHALL 算法.....	19
2.4.1 实验题目	19
2.4.2 设计思路	19
2.4.3 程序源代码.....	20
2.4.4 运行演示	22
三、 心得体会	23

一、实验情况总览

在本报告中共完成了如下实验：

第一部分：最近点对问题、大整数加减乘运算、最优二分查找树、FLOYD-WARSHALL 算法。

第二部分（提高部分）：无。

二、解题报告

2.1 实验一：最近点对问题

2.1.1 实验题目

已知平面上分布着点集 P 中的 n 个点 p_1, p_2, \dots, p_n ，点 p_i 的坐标记为 (x_i, y_i) ， $1 \leq i \leq n$ 。两点之间的距离使用欧式距离进行计算，记为

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

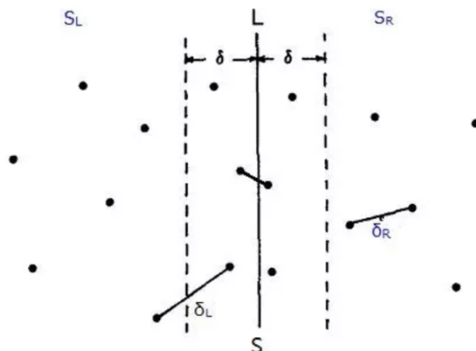
要求设计算法，找到点集 P 中一对距离最近的点。

2.1.2 设计思路

使用分治思想进行算法设计，求解最近点对问题。设计的具体算法如下：

1. 按照所有点的 x 坐标对点集 P 进行排序。排序的时间复杂度为 $O(n \log n)$ 。
2. 在 x 坐标中位数对应的点处将点集所在平面划分为两半。划分后左右两半平面分别记为 S_L 和 S_R ，此时问题被分为两个子问题。问题一、分别在平面 S_L 和 S_R 中求得最短距离点对；问题二、求跨越分割线的最短距离点对。

3. 对问题进行划分后，分别对两个子问题进行处理。对于子问题一，用递归进行第二步操作的方式不断对子平面进行划分，直至子问题小到能够分别直接求出 S_L 与 S_R 两个平面内的最短距离点对，记此时求得的最短距离分别为 δ_L 和 δ_R 并设 $\delta = \min(\delta_L, \delta_R)$ ；对于子问题二，在平面上距离划分线两边的 δ 处做两条平行线，从而得到一个带状区域。由于在 S_L 、 S_R 区域中的点对最短距离为 δ ，在跨越划分线的点对中若有更短距离点对，则两点必然均在带状区域中，如图所示。



将带状区域的点加入一个新的数组中，并对这些点按照 y 轴坐标升序排列。此时对于带状区域中的每个点 p_i ，可以证明在最坏情况下至多只需要考虑随后的另外 7 个点与它的距离。设此时求得的最短距离为 dm ，比较 dm 与 δ 的大小，取其中的最小值作为最近点对距离。

对于算法的时间复杂度：因为在分治过程中每次递归调用算法时，对子问题进行合并时都需要对带状区域中点集按 y 坐标进行排序，这一部分算法的时间复杂度为 $O(n\log n)$ 。则有 $T(n)=2T(n/2)+O(n\log n)$ 。由主方法可知算法的时间复杂度为 $O(n\log^2 n)$ 。

对于算法的空间复杂度：递归时每次所需的辅助空间大小为 n ，且递归的深度为 $\log^2 n$ 。又因为递归算法的空间复杂度=每次所需辅助空间*递归深度。所以，该算法的空间复杂度为 $O(n\log n)$ 。

2.1.3 程序源代码

```
/* problem: closest points */
#include<iostream>
#include <ctime>
#include <cmath>
#include <algorithm>

using namespace std;

class POINT{
public:
    double x;
    double y;
    POINT(){}
    POINT(double x, double y){//make POINT
        POINT::x=x;
        POINT::y=y;
    }
};

//平面上点对距离
double distance(POINT a, POINT b){
    return sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y));
}

//合并时比较  $\delta$  距离
double min(double a, double b){
    return (a<b)?a:b;
}

//自定义排序规则：依照结构体中的 x 成员变量升序排序
bool compareX(POINT a, POINT b){
    return a.x < b.x;
}

//自定义排序规则：依照结构体中的 y 成员变量升序排序
bool compareY(POINT a, POINT b){
    return a.y < b.y;
}

//求出最近点对记录，并将两点记录再 a、b 中
double cal_mindistance(POINT points[], int length, POINT &a, POINT &b){
    double mindistance;           //记录集合 points 中最近两点距离
    double distance1, distance2, distance3;           //记录分割后两个子集中各自最小点
对距离
    int i = 0, j = 0, k = 0, x = 0;           //用于控制 for 循环的循环变量
    POINT a1, b1, a2, b2;           //保存分割后两个子集中最小点对
```

```

if (length < 2) return 100000;    //若子集长度小于 2，定义为最大距离，表示不可达

else if (length == 2) { //若子集长度等于 2，直接返回该两点的距离
    a = points[0];
    b = points[1];
    return distance(points[0], points[1]);
}

else { //子集长度大于 3，进行分治求解
    //分解
    POINT *pts1 = new POINT[length/2];    //开辟两个子集
    POINT *pts2 = new POINT[length-length/2];

    double mid = points[(length - 1) / 2].x;    //排完序后的中间下标值，即中位数

    for (i = 0; i < length / 2; i++)
        pts1[i] = points[i];
    for (int j = 0, i = length / 2; i < length; i++)
        pts2[j++] = points[i];

    distance1 = cal_mindistance(pts1, length / 2, a1, b1);    //分治求解左半部分子集的
最近点
    distance2 = cal_mindistance(pts2, length - length / 2, a2, b2);    //分治求解右半部分子集的最近
点

    if (distance1 < distance2) { mindistance = distance1; a = a1; b = b1; }    //记录最近点，
最近距离
    else { mindistance = distance2; a = a2; b = b2; }

    //合并：求解跨分割线并在  $\delta \times 2\delta$  区间内的最近点对
    POINT *pts3 = new POINT[length];

    for (i = 0, k = 0; i < length; i++)    //取得中线  $2\delta$  宽度的所有点对共 k
个
        if (abs(points[i].x - mid) <= mindistance)
            pts3[k++] = points[i];

    sort(pts3, pts3 + k, compareY);    // 以 y 排序矩形阵
内的点集合

    for (i = 0; i < k; i++)
    {
        for (j = i + 1; j <= i + 6 && j < k; j++)    //只需与有序的邻接的 6 个点进行比
较
        {
            distance3 = distance(pts3[i], pts3[j]);
            if (distance3 < mindistance)
            { //如果跨分割线的两点距离小于已知最小距离，则记录该距离和两点
                a = pts3[i];
                b = pts3[j];
                mindistance = distance3;
            }
        }
    }
    return mindistance;
}
}

```

```

#define COORDINATE_RANGE 100
void SetPoints(POINT *points, int length)
{
    //随机函数对点数组 points 中的二维点进行初始化
    srand(unsigned(time(NULL)));
    double digital;
    for (int i = 0; i < length; i++)
    {
        digital = (rand() % int(COORDINATE_RANGE * 200)) / COORDINATE_RANGE -
COORDINATE_RANGE;
        digital /= 100;
        points[i].x = (rand() % int(COORDINATE_RANGE * 200)) / COORDINATE_RANGE -
COORDINATE_RANGE;
        points[i].x += digital;
        digital = (rand() % int(COORDINATE_RANGE * 200)) / COORDINATE_RANGE -
COORDINATE_RANGE;
        digital /= 100;
        points[i].y = (rand() % int(COORDINATE_RANGE * 200)) / COORDINATE_RANGE -
COORDINATE_RANGE;
        points[i].y += digital;
    }
}

int main(void)
{
    int num;           //随机生成的点对个数
    POINT a, b;        //最近点对
    double diatance;   //点对距离

    cout << "请输入二维点对个数:";
    cin >> num;
    if (num < 2)
        cout << "请输入两个以上点" << endl;
    else
    {
        cout << endl << "随机生成的" << num << "个二维点对如下: " << endl;
        POINT *points = new POINT[num];

        SetPoints(points, num);
        for (int i = 0; i < num; i++)
            cout << "(" << points[i].x << ", " << points[i].y << ")" << endl;

        sort(points, points + num, compareX);    //调用 algorithm 库中的 sort 函数对 points 进行排序,
compareX 为自定义的排序规则

        diatance = cal_mindistance(points, num, a, b);
        cout << endl << endl << "按横坐标排序后的点对:" << endl;
        for (int i = 0; i < num; i++)
            cout << "(" << points[i].x << ", " << points[i].y << ")" << endl;

        cout << endl << "最近点对为: " << "(" << a.x << ", " << a.y << ")" 和 " << "(" << b.x << ", " << b.y
<< ")" << endl << "最近点对距离为: " << diatance << endl;

        delete(points);
    }
}

```

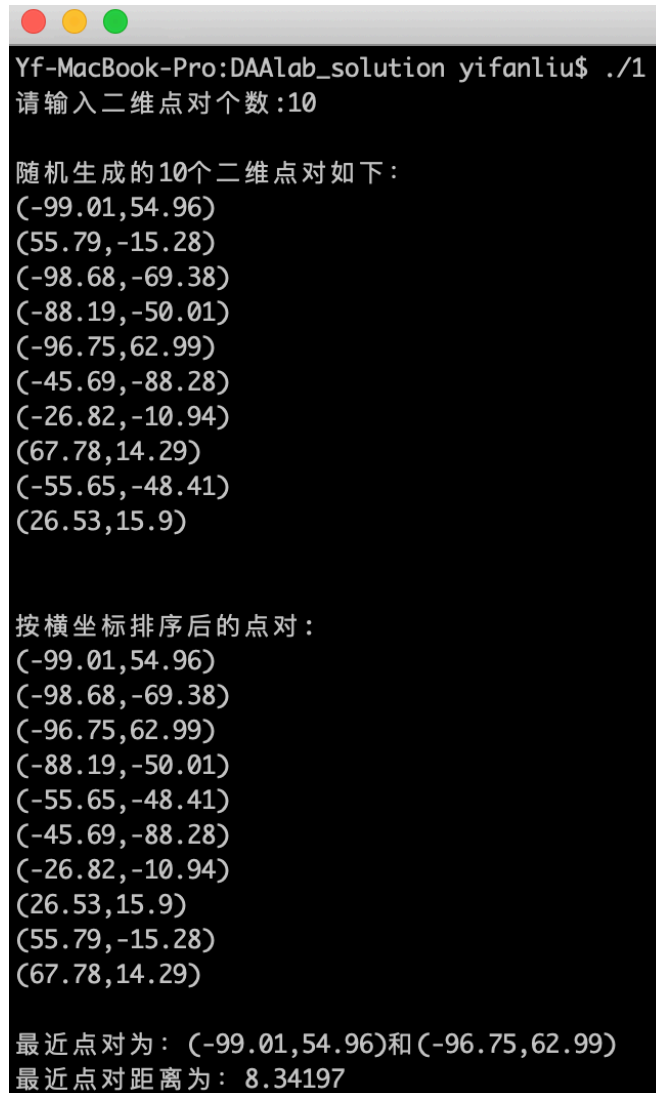
2.1.4 运行演示

在本次实验的程序设计中设计了一个函数 SetPoints, 能够根据用户输入随机

生成指定个数的点。同时在程序中设置了一个常量 `COORDINATE_RANGE`，用于限制点集的范围，即对于每个点 $p(x,y)$ ， $|x|$ 、 $|y|$ 须小于该常量，本次测试中该常量的值取为 100。

运行演示的过程中输入点个数为 10，程序将首先随机生成制定空间范围内的 10 个点对。使用 2 中设计的算法进行求解时，程序首先对所有点按 x 坐标进行排序，随后递归调用算法进行计算，最后得到并输出最近点对及其距离。

运行结果如图所示：



```
Yf-MacBook-Pro:DAAlab_solution yifanliu$ ./1
请输入二维点对个数:10

随机生成的10个二维点对如下:
(-99.01,54.96)
(55.79,-15.28)
(-98.68,-69.38)
(-88.19,-50.01)
(-96.75,62.99)
(-45.69,-88.28)
(-26.82,-10.94)
(67.78,14.29)
(-55.65,-48.41)
(26.53,15.9)

按横坐标排序后的点对:
(-99.01,54.96)
(-98.68,-69.38)
(-96.75,62.99)
(-88.19,-50.01)
(-55.65,-48.41)
(-45.69,-88.28)
(-26.82,-10.94)
(26.53,15.9)
(55.79,-15.28)
(67.78,14.29)

最近点对为: (-99.01,54.96)和(-96.75,62.99)
最近点对距离为: 8.34197
```

程序运行结果说明当随机生成 10 个点时，分治算法能够计算出 10 个点中的最近点对并输出该点对以及两点间距离。程序运行结果符合预期，验证了算法的正确性。

2.2 实验二：大整数加减乘运算

2.2.1 实验题目

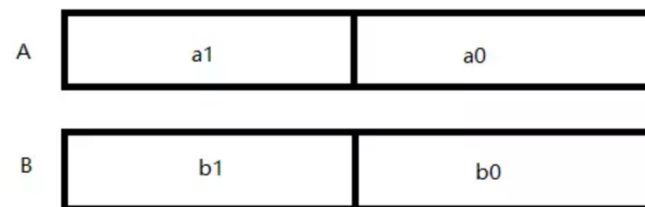
大整数 (big integer)：位数很多的整数，普通的计算机不能直接处理，如：9834975972130802345791023498570345。对大整数进行的算术运算，显然常规程序语言是无法直接表示的。

利用分治法设计一个计算两个 n 位的大整数相乘的算法，要求计算时间低于 $O(n^2)$ 。同时编程实现大整数的加减运算，需考虑操作数为 0、负数、任意位等各种情况。

2.2.2 设计思路

对于大整数加减的问题，对大整数 A、B 对应的字符串人工进行判断并实现进借位逻辑即可，算法的时间复杂度将被限制在 $O(n)$ 内。

对于大整数相乘的问题，使用分治思想进行算法设计：设目前有 A、B 两个大数且均为 n 位（在数字 A、B 的最高位前添加 0 使数字位数为 2 的指数倍，此后若数字 A 和 B 位数不同，则在其中一个数字的最高位前添加前置 0 直至相同），需要计算 $A*B$ 的值，则需要将 A 和 B 划分成两等份，如图所示：



其中，整数 A 被分成 $a1$ 和 $a0$ 两等份，整数 B 被分成 $b1$ 和 $b0$ 两等份，每个子串长度均为 $n/2$ 。则为了减少乘运算从而降低算法运行时的时间复杂度，可推导出关系式如下：

$$A = a1 * 10^{(n/2)} + a0$$

$$B = b1 * 10^{(n/2)} + b0$$

$$A * B = c2 * 10^n + c1 * 10^{(n/2)} + c0$$

其中：

$$c2 = a1 * b1$$

$$c1 = a0 * b1 + b0 * a1 = (a1 + a0) * (b1 + b0) - (c2 + c0)$$

经过以上操作后即进行了一次划分，问题被拆解为两个规模更小的子问题，此后需要递归调用算法对每个子问题继续进行划分，且每次划分都按照以上式子进行合并，从而能够在规定的时间复杂度内求解问题。

2.2.3 程序源代码

```
/* problem: big integer */
/**
    分治降低复杂度原理：
    按如下规则划分乘数和被乘数 A、B：
    A = a1 * 10^(n/2) + a0
    B = b1 * 10^(n/2) + b0
    则乘法 A * B 可表示为：
    A * B = c2 * 10^n + c1 * 10^(n/2) + c0
    其中：
    c2 = a1 * b1
    c1 = a0 * b1 + b0 * a1 = (a1 + a0) * (b1 + b0) - (c2 + c0)
    (尽量减少乘法运算，需要将 c1 变形为后面的式子)
```

```

**/

#include <iostream>
#include <stdlib.h>
#include <algorithm>
#include <sstream>

using namespace std;

//string 字符串变整数型例如 str="1234", 转换为整数的 1234.
int str2Int(string k) {
    int back;
    stringstream instr(k);
    instr >> back;
    return back;
}

string int2Str(int intValue) {
    string result;
    stringstream stream;
    stream << intValue;//将 int 输入流
    stream >> result;//从 stream 中抽取前面放入的 int 值
    return result;
}

void removePreZero(string &str) {
    //去掉前置 0,需要考虑只有一个 0 或者全部是 0 的情况
    if (str.length() == 1)return;
    int k = 0;
    for (int i = 0; i < str.length(); i++) {
        if (str.at(i) == '0') {
            k++;
        } else {
            break;
        }
    }
    if (k == str.length())str = "0";
    else {
        str = str.substr(k);
    }
}

/**
 * 大数相加，不考虑前置 0 的情况
 * @param x
 * @param y
 * @return
 */
string add(string x, string y) {
    string result = "";

    //去掉前置 0
    removePreZero(x);
    removePreZero(y);

    //反转字符串方便相加
    reverse(x.begin(), x.end());
    reverse(y.begin(), y.end());

```

```

    for (int i = 0, j = 0; j || i < max(y.length(), x.length()); i++) {
        int t = j;
        if (i < x.length()) t += (x.at(i) - '0');
        if (i < y.length()) t += (y.at(i) - '0');
        //c.s[c.len++] = t % 10;
        int q = t % 10;
        result.insert(0, int2Str(q));
        j = t / 10;
    }
    return result;
}

```

```

string subtract(string &x, string &y) {
    string result = "";

    //去掉前置 0
    removePreZero(x);
    removePreZero(y);

    int len_x = x.length();
    int len_y = y.length();
    int len = len_x > len_y ? len_x : len_y;
    int *tempResult = (int *) malloc(sizeof(int) * len);

    string sign;
    if (len_x > len_y) { // x - y 为正数
        sign = "+";
    } else if (len_x < len_y) { // x-y 为负数
        sign = "-";
    } else { //长度相同则判断各位的大小
        int i;
        for (i = 0; i < len_x; i++) {
            if (x.at(i) == y.at(i)) continue;
            else if (x.at(i) > y.at(i)) {
                sign = "+";
                break;
            } else {
                sign = "-";
                break;
            }
        }
    }

    //说明没有 break, 说明 x == y
    if (i == len_x) {
        return "0";
    }

    //反转字符串方便相减
    reverse(x.begin(), x.end());
    reverse(y.begin(), y.end());

    int q = 0;
    //若 x 大, 则直接相减, 否则用 y-x, 并且最终加上负号
    for (int i = 0; i < len; i++) {
        int aint = i < len_x ? x.at(i) - '0' : 0;
        int bint = i < len_y ? y.at(i) - '0' : 0;
        if (sign.at(0) == '+') {

```

```

        tempResult[q++] = aint - bint;
    } else {
        tempResult[q++] = bint - aint;
    }
}

for (int i = 0; i < q; i++) {
    if (tempResult[i] < 0) {
        tempResult[i + 1] -= 1;
        tempResult[i] += 10;
    }
}
q--;
while (tempResult[q] == 0)q--;
for (int i = q; i >= 0; i--) {
    result += int2Str(tempResult[i]);
}

if (sign.at(0) == '-')return sign + result;
return result;
}

/**
 * 添加前置 0
 * @param str
 * @param zero_num
 */
void addPreZero(string &str, int zero_num) {
    for (int i = 0; i < zero_num; i++)str.insert(0, "0");
}

string addLastZero(string str, int zero_num) {
    string s = str;
    for (int i = 0; i < zero_num; i++)s += "0";
    return s;
}

/**
 * 计算 x 和 y 的和
 */
string madd(string x, string y) {

    bool flag_x = false; //正
    bool flag_y = false;
    bool flag; // 最终结果的正负号

    //先处理正负号
    if (x.at(0) == '-') { //负
        flag_x = true;
        x = x.substr(1);
    }
    if (y.at(0) == '-') {
        flag_y = true;
        y = y.substr(1);
    }

    string result;
    if (!flag_x && !flag_y){

```

```

        result = add(x,y);
    }
    else if(!flag_x && flag_y){
        result = subtract(x,y);
    }
    else if(flag_x && !flag_y){
        result = subtract(y,x);
    }
    else{
        result = add(x,y);
        result.insert(0, "-");
    }

    return result;
}

/**
 * 计算 x 和 y 的差
 */
string msub(string x, string y) {

    bool flag_x = false; //正
    bool flag_y = false;
    bool flag; // 最终结果的正负号

    //先处理正负号
    if (x.at(0) == '-') { //负
        flag_x = true;
        x = x.substr(1);
    }
    if (y.at(0) == '-') {
        flag_y = true;
        y = y.substr(1);
    }

    string result;
    if(!flag_x && !flag_y){
        result = subtract(x,y);
    }
    else if(!flag_x && flag_y){
        result = add(x,y);
    }
    else if(flag_x && !flag_y){
        result = add(x,y);
        result.insert(0, "-");
    }
    else{
        result = subtract(y,x);
    }

    return result;
}

/**
 * 计算 x 和 y 的乘积, 重载后可接收 int 型的和 string 类型的
 * @param x
 * @param y
 * @return

```

```

*/
string multiply(string &x, string &y) {

    bool flag_x = false; //正
    bool flag_y = false;
    bool flag; // 最终结果的正负号

    //先处理正负号
    if (x.at(0) == '-') { //负
        flag_x = true;
        x = x.substr(1);
    }

    if (y.at(0) == '-') {
        flag_y = true;
        y = y.substr(1);
    }
    //两个都为负或者两个都为正，则最终为正
    if ((flag_x && flag_y) || (!flag_x && !flag_y)) {
        flag = false;
    } else {
        flag = true; //否则结果为负
    }

    /**
     * 预处理，将 x 和 y 处理成相同位数的两个数
     * x 和 y 的长度必须是 2 的指数倍，这样才能递归分治计算
     * 所以需要将 x 和 y 添加前置 0，将长度处理为可行的最小的长度
     *
     * 处理过后 x 和 y 的长度将统一
     */
    int init_len = 4;
    if (x.length() > 2 || y.length() > 2) { // 长度大于 2 时，最小长度应该为 4，故初始值为 4
        if (x.length() >= y.length()) {
            while (init_len < x.length()) init_len *= 2; //计算两个数最终的长度
            //添加前置 0
            if (x.length() != init_len) {
                addPreZero(x, init_len - x.length());
            }
            addPreZero(y, init_len - y.length());
        } else {
            while (init_len < y.length()) init_len *= 2;
            //添加前置 0
            addPreZero(x, init_len - x.length());
            if (y.length() != init_len) {
                addPreZero(y, init_len - y.length());
            }
        }
    }

    if (x.length() == 1) {
        addPreZero(x, 1);
    }
    if (y.length() == 1) {
        addPreZero(y, 1);
    }

    int n = x.length();

```

```

    string result;

    string a1, a0, b1, b0;
    if (n > 1) {
        a1 = x.substr(0, n / 2);
        a0 = x.substr(n / 2, n);
        b1 = y.substr(0, n / 2);
        b0 = y.substr(n / 2, n);
    }

    if (n == 2) { //长度为 2 时，结束递归
        int x1 = str2Int(a1);
        int x2 = str2Int(a0);
        int y1 = str2Int(b1);
        int y2 = str2Int(b0);
        int z = (x1 * 10 + x2) * (y1 * 10 + y2);
        result = int2Str(z);
    }
    else {
        string c2 = multiply(a1, b1);
        string c0 = multiply(a0, b0);
        string temp_c1_1 = add(a0, a1);
        string temp_c1_2 = add(b1, b0);
        string temp_c1_3 = add(c2, c0);
        string temp_c1 = multiply(temp_c1_1, temp_c1_2);
        string c1 = subtract(temp_c1, temp_c1_3);
        string s1 = addLastZero(c1, n / 2);
        string s2 = addLastZero(c2, n);
        result = add(add(s1, s2), c0);
    }

    if (flag) { //结果为负数
        result.insert(0, "-");
    }

    return result;
}

int main() {
    string a, b;
    cout<<"input a: ";
    cin>>a;
    cout<<"input b: ";
    cin>>b;

    cout << "+: " << madd(a, b) << endl;
    cout << "-: " << msub(a, b) << endl;
    cout << "*: " << multiply(a, b) << endl;

    return 0;
}

```

2.2.4 运行演示

在测试用例一中，取数字 A: 123456789123456789，数字 B: 90，程序运行结果如图：

```
Yf-MacBook-Pro:DAAlab_solution yifanliu$ ./2
input a: 123456789123456789
input b: 9
+: 123456789123456798
-: 123456789123456780
*: 111111110211111101
```

经过手算验证，大数乘法执行结果正确，验证了算法的正确性。

测试用例二用于测试程序能否对负数进行运算并输出正确结果，其中取数字 A: 123456789，数字 B: -90，程序运行结果如图：

```
Yf-MacBook-Pro:DAAlab_solution yifanliu$ ./2
input a: 123456789
input b: -90
+: 123456699
-: 123456879
*: -1111111010
```

经过手算验证，程序对负数进行运算的结果也是正确的，验证了算法的正确性。

2.3 实验三：最优二分查找树

2.3.1 实验题目

对于一个给定的概率集合，由于我们知道每个关键字和伪关键字的搜索概率，因而可以确定在一棵给定的二叉搜索树 T 中进行一次搜索的期望代价 E 。

我们希望构造一棵期望搜索代价最小的二叉搜索树，这样的树我们称之为最优二分查找树（最优二叉搜索树）。

2.3.2 设计思路

对最优二叉树的形式化定义采用《算法导论》课本中 15.5 节的内容定义。

穷举法构造最优二叉搜索树需要检查指数棵二叉搜索树，故使用动态规划的方法设计对应求解算法：

1. 容易证明最优二叉搜索树具有最优子结构，所以可以用子问题的最优解构造原问题的最优解。

2. 递归定义最优解。选取子问题域为求解包含关键字 k_i, \dots, k_j 的最优二叉搜索树，其中 $i \geq 1, j \leq n$ 且 $j \geq i-1$ 。定义 $e[i, j]$ 为在包含关键字 k_i, \dots, k_j 的最优二叉搜索树中进行一次搜索的期望代价，最终希望计算 $e[1, n]$ 。当 $j=i-1$ 时只包含伪关键字，此时期望的搜索代价为 $e[i, i-1]=q_{i-1}$ ；当 $j \geq i$ 时需要从 k_i, \dots, k_j 中选择一个根 k_r ，然后用其两侧的关键字分别构造两棵最优二叉搜索树作为其左右子树，此时这棵树的期望搜索代价增加，对应的增加函数为：

$$w(i, j) = \sum_{l=i}^j p_l + \sum_{l=i-1}^j q_l$$

则 $e[i, j]$ 可重写为：

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j)$$

假设我们知道该采用哪个节点 k_r 作为根，可以得到最终的递归公式：

$$e[i, j] = \begin{cases} q_{i-1} & \text{如果 } j = i-1 \\ \min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{如果 } i \leq j \end{cases}$$

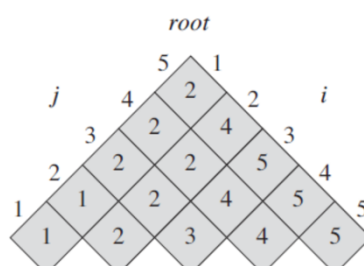
3. 计算求解最优二叉搜索树的期望搜索代价。 $e[i,j]$ 的值给出了最优二叉搜索树的期望代价，为了记录最优二叉搜索树的结构，定义 $root[i, j]$ 保存根节点 k_r 的下标 r 。同时为了避免每次计算 e 时重新计算 $w(i,j)$ ，将 w 保存在表 $w[1..n+1, 0..n]$ 中。最后可设算法伪代码如下：

```

OPTIMAL-BST( $p, q, n$ )
1  for  $i \leftarrow 1$  to  $n+1$ 
2      do  $e[i, i-1] \leftarrow q_{i-1}$ 
3      do  $w[i, i-1] \leftarrow q_{i-1}$ 
4  for  $l \leftarrow 1$  to  $n$ 
5      do for  $i \leftarrow 1$  to  $n-l+1$ 
6          do  $j \leftarrow i+l-1$ 
7               $e[i, j] \leftarrow \infty$ 
8               $w[i, j] \leftarrow w[i, j-1] + p_j + q_j$ 
9              for  $r \leftarrow i$  to  $j$ 
10                  do  $t \leftarrow e[i, r-1] + e[r+1, j] + w[i, j]$ 
11                  if  $t < e[i, j]$ 
12                      then  $e[i, j] \leftarrow t$ 
13                       $root[i, j] \leftarrow r$ 
14  return  $e$  and  $root$ 

```

对结果进行输出时，根据 $root$ 表的定义， $root$ 保存包含关键字 k_1, k_2, \dots, k_3 的根节点 k_r 的下标 r ，如图所示。



对于规模为 n 的二叉搜索树，从 $root$ 开始进行递归操作，每次进行递归时，取出当前各点的根的下标 r 并将 r 分别与 i, j 进行比较。如果 r 和 i 相等，说明已经触达了最底层的关键字，此时只需打印出 d_{r-1} 为 k_r 的左子树即可。若 r 与 i 不相等，则下标为 $root$ 的关键字为 k_r 的左子树，并以同样的方式递归处理左子树即可。

2.3.3 程序源代码

```

/* problem: optimal binary search tree */

#include <iostream>
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <limits>

using namespace std;

int check_sum(int n, double p[], double q[]){
    double sum=0;

```

```

//calculate sum
for(int i=0;i<=n;i++){
    if(i==0){
        sum+=q[i];
    }
    else{
        sum+=q[i];
        sum+=p[i];
    }
    // cout<<sum<<"\n";
}
//check the result
if(abs(sum-1)<0.0000001)return 1;
return 0;
}

void CONSTRUCT_OPTIMAL_BST(int root[][10000], int i, int j, int last){
    if(i>j){
        if(j<last)
            cout<<"d'<j<<" is the left child of k"<<last<<"\n";
        else
            cout<<"d'<j<<" is the right child of k"<<last<<"\n";
        // cout<<i<<" "<j<<" "<last<<"\n";
        return;
    }
    if(last==0)
        cout<<"k'<root[i][j]<<" is the root.\n";
    else if(j<last)
        cout<<"k'<root[i][j]<<" is the left child of k"<<last<<"\n";
    else
        cout<<"k'<root[i][j]<<" is the right child of k"<<last<<"\n";

    CONSTRUCT_OPTIMAL_BST(root,i,root[i][j]-1,root[i][j]);
    CONSTRUCT_OPTIMAL_BST(root,root[i][j]+1,j,root[i][j]);
}

int main(void){
    int n;

    cout<<"number of key:";
    cin>>n;

    //input
    double p[n+1],q[n+1];
    //result
    double e[n+1][n],w[n+1][n];
    int root[n+1][10000];

    cout<<"p("<n<<"): ";
    for(int i=1;i<=n;i++){
        cin>>p[i];
    }
    cout<<"q("<n+1<<"): ";
    for(int i=0;i<=n;i++){
        cin>>q[i];
    }

    //p+q=1?

```

```

if(check_sum(n,p,q)==0){
    cout<<"check your input plz.\n";
    return 0;
}

//initial
for(int i=1;i<=n+1;i++){
    e[i][i-1]=q[i-1];
    w[i][i-1]=q[i-1];
}
//main
for(int l=1;l<=n;l++){
    for(int i=1;i<=n-l+1;i++){
        int j=i+l-1;
        e[i][j]=DBL_MAX;
        w[i][j]=w[i][j-1]+p[j]+q[j];
        for(int r=i;r<=j;r++){
            double t=e[i][r-1]+e[r+1][j]+w[i][j];
            if(t<e[i][j]){
                e[i][j]=t;
                root[i][j]=r;
            }
        }
    }
}

// for(int i=1;i<=n;i++){
//     for(int j=i;j<=n;j++){
//         cout<<root[i][j]<<' ';
//     }
//     cout<<"\n";
// }

CONSTRUCT_OPTIMAL_BST(root,1,n,0);

return 0;
}

```

2.3.4 运行演示

测试用例使用了《算法导论》15.5 节中图 15-9 对应的搜索概率，程序运行结果如图所示。

```

Yf-MacBook-Pro:DAAlab_solution yifanliu$ ./3
number of key:5
p(5): 0.15 0.10 0.05 0.10 0.20
q(6): 0.05 0.10 0.05 0.05 0.05 0.10
k2 is the root.
k1 is the left child of k2
d0 is the left child of k1
d1 is the right child of k1
k5 is the right child of k2
k4 is the left child of k5
k3 is the left child of k4
d2 is the left child of k3
d3 is the right child of k3
d4 is the right child of k4
d5 is the right child of k5

```

将程序执行结果与《算法导论》中对应用例的最优二叉搜索树进行对比后，验证了程序中算法设计的正确性。

2.4 实验四：Floyd-Warshall 算法

2.4.1 实验题目

输入一个带权图，找到该图中每个节点之间的最短路径，并输出相应的最短路径。其中进行计算时使用 Floyd-Warshall 算法求解节点之间的最短距离。求解最短距离的同时记录前驱矩阵 π ，求解完成后使用 print_all 算法推导并输出每条最短路径的长度以及具体线路。

2.4.2 设计思路

对于一个图，其最短路径具有最优子结构。因此，可以使用动态规划的方式来进行所有点对的最短路径的求解，即可以使用 Floyd-Warshall 算法求解问题。

Floyd-Warshall 算法应用了最短路径具有最优子结构的递归公式：

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$

公式中 $d_{ij}^{(k)}$ 表示的是从节点 i 到节点 j 的中间节点取自 $\{1, 2, \dots, k\}$ 的集合。特别地，当 $k=0$ 时，两个点之间没有任何中间节点，其最短路径位连接两个节点的边的权（如果没有相连，即距离为无穷大）。

算法的核心，在于构造 $d_{ij}^{(k)}$ 的最短路径权重矩阵 $D^{(k)}$ 。在初始时， $D^{(0)}$ 和输入的矩阵相同。每一次循环，都增加一个中间节点，并对矩阵中的所有点对进行上述递归操作。实现方式的伪代码如下：

```
FLOYD-WARSHALL( $W$ )
1   $n = W.rows$ 
2   $D^{(0)} = W$ 
3  for  $k = 1$  to  $n$ 
4      let  $D^{(k)} = (d_{ij}^{(k)})$  be a new  $n \times n$  matrix
5      for  $i = 1$  to  $n$ 
6          for  $j = 1$  to  $n$ 
7               $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
8  return  $D^{(n)}$ 
```

由于经过三层 for 循环的嵌套，该算法的时间复杂度为 $O(n^3)$ 。

完成上述过程后，仅能算出所有点对的最小距离。为了能打印出最短的路径还需要在算法中加入一个前驱矩阵 π 。

当 $k=0$ 时， i 到 j 的一条路径上不存在任何中间节点，如果 i 、 j 相连，那么 j 的前驱节点即 i 。

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty. \end{cases}$$

当 $k>0$ 时，如果加入了 k 节点后 i 到 j 的路径长度变大了，说明 j 的前驱节点和之前选择的是一样的。如果加入 k 节点后 i 到 j 的路径长度减小了，则 j 的前驱节点将改为 k_j 路径上的 j 的前驱节点。

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

根据前驱矩阵打印点之间的最短路径时，算法如下：若 $\pi_{ij} = \text{NIL}$ ，则 i 到 j 之间不存在路径；若 $i \neq j$ ，先调用递归尝试打印 i 到 j 的前驱节点间的路径，即 i 到 π_{ij} 之间的路径，再打印出 j ；若 $i = j$ ，说明递归到了底部，打印出起始节点 i 即可。打印的最坏时间复杂度为 $O(n)$ 。

```
PRINT-ALL-PAIRS-SHORTEST-PATH( $\Pi, i, j$ )
1  if  $i == j$ 
2      print  $i$ 
3  elseif  $\pi_{ij} == \text{NIL}$ 
4      print "no path from"  $i$  "to"  $j$  "exists"
5  else PRINT-ALL-PAIRS-SHORTEST-PATH( $\Pi, i, \pi_{ij}$ )
6      print  $j$ 
```

若要打印出所有点对之间的路径，则需要对所有点对进行遍历，即打印的时间复杂度提升到 $O(n^3)$ 。

2.4.3 程序源代码

```
/* problem: Floyd-Warshall */
#include <iostream>
#include <fstream>
#include <algorithm>

using namespace std;

//max of int
const int INF=100000;
//max size of graph
const int N=10;

int v_num,e_num;
int Graph[N][N];
int Weight[N][N];
int Pi[N][N];

int initial_graph(string filename){
    //initial here
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++){
            Pi[i][j]=-1;
            if(i==j)Graph[i][j]=0;
            else Graph[i][j]=INF;
        }
    }
    //read file
    ifstream in(filename);
    if(!in){
        cout<<"not exist such file!";
        return -1;
    }
    in>>v_num>>e_num;

    //make Graph
    for(int k=0;k<e_num;k++){
```

```

        int i,j,weight;
        in>>i>>j>>weight;
        Graph[i][j]=weight;
    }
    //make Pi
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++){
            if(i!=j && Graph[i][j]<INF) Pi[i][j]=i;
            else Pi[i][j]=-1;
        }
    }
    return 0;
}

void floyd_warshall(){
    for(int k=0;k<v_num;k++){
        for(int i=0;i<v_num;i++){
            for(int j=0;j<v_num;j++){
                if(Graph[i][j]>Graph[i][k]+Graph[k][j]){
                    Graph[i][j]=Graph[i][k]+Graph[k][j];
                    Pi[i][j]=Pi[k][j];
                }
            }
        }
    }
    return;
}

void print_path(int i,int j){
    if(i==j){
        cout<<i;
    }
    else if(Pi[i][j]==-1){
        cout<<"\nno path from "<<i<<" to "<<j<<endl;
    }
    else{
        print_path(i,Pi[i][j]);
        cout<<"->"<<j;
    }
}

void print_all(){
    int i,j;
    for(i=0;i<v_num;i++){
        for(j=0;j<v_num;j++){
            if(Graph[i][j]!=INF && i!=j){
                cout<<"点"<<i<<"到点"<<j<<"的最短距离为:"<<Graph[i][j];
                cout<<"  路径为:";
                print_path(i,j);
                cout<<endl;
            }
            else if(i!=j){
                cout<<"点"<<i<<"到点"<<j<<"没有可达路径"<<endl;
            }
        }
    }
    return;
}
}

```

```

void test_graph(){
    for(int i=0;i<v_num;i++){
        for(int j=0;j<v_num;j++){
            if (Graph[i][j]==INF)
                cout<<"# ";
            else
                cout<<Graph[i][j]<<" ";
        }
        cout<<"\n";
    }
    cout<<"\n";

    for(int i=0;i<v_num;i++){
        for(int j=0;j<v_num;j++){
            cout<<Pi[i][j]<<" ";
        }
        cout<<"\n";
    }
    return ;
}

int main(void){
    string filename("./graph.txt");
    if (initial_graph(filename)==-1) return 0;
    // test_graph();
    floyd_warshall();
    print_all();
    // test_graph();
    return 0;
}

```

2.4.4 运行演示

程序运行时从文件 `graph.txt` 中导入图的数据，其中包括图的顶点数、边数、以及有向边的起点终点和权重。本次测试使用的图为本上的图 25-1，对应文本文件 `graph.txt` 的内容如下所示：

```

5 9
0 1 3
0 2 8
0 4 -4
1 3 1
1 4 7
2 1 4
3 2 -5
3 0 2
4 3 6

```

执行程序后，成功打印出节点之间的最短路径长度和具体路径，如图所示。

```
Yf-MacBook-Pro:DAAlab_solution yifanliu$ ./4
点0到点1的最短距离为:1 路径为:0->4->3->2->1
点0到点2的最短距离为:-3 路径为:0->4->3->2
点0到点3的最短距离为:2 路径为:0->4->3
点0到点4的最短距离为:-4 路径为:0->4
点1到点0的最短距离为:3 路径为:1->3->0
点1到点2的最短距离为:-4 路径为:1->3->2
点1到点3的最短距离为:1 路径为:1->3
点1到点4的最短距离为:-1 路径为:1->3->0->4
点2到点0的最短距离为:7 路径为:2->1->3->0
点2到点1的最短距离为:4 路径为:2->1
点2到点3的最短距离为:5 路径为:2->1->3
点2到点4的最短距离为:3 路径为:2->1->3->0->4
点3到点0的最短距离为:2 路径为:3->0
点3到点1的最短距离为:-1 路径为:3->2->1
点3到点2的最短距离为:-5 路径为:3->2
点3到点4的最短距离为:-2 路径为:3->0->4
点4到点0的最短距离为:8 路径为:4->3->0
点4到点1的最短距离为:5 路径为:4->3->2->1
点4到点2的最短距离为:1 路径为:4->3->2
点4到点3的最短距离为:6 路径为:4->3
Yf-MacBook-Pro:DAAlab_solution yifanliu$
```

对照书本上算法执行结束时的结果矩阵后，发现结果与预期符合，验证了算法的正确性。

三、 心得体会

在算法设计与分析的的实验课程中，实验的选题贴近课堂教学且覆盖范围较广，在完成实验的过程中能够有效的对课内知识进行复习。

具体来说，在进行实验一（最近点对问题）和实验二（大整数加减乘）的过程中，我对分治算法的思想已经使用时的经典步骤都有了更好的理解和领会，对分治思想的运用更加熟练；在进行实验三（最优二分查找树）时，对算法中的一大难点动态规划有了更深的体会；在进行实验四（FLOYD-WARSHALL 算法）的实践后，我对与图相关的问题有了一定的了解，对于降低算法复杂度的方式有了新的思路。

总的来说，经历过两次实验课程后，收获颇丰，通过亲自动手实践的方式学习算法也提高了我对算法设计的好奇心与兴趣，为日后的学习打下了基调。