目录

一、实验情况总览.....2
二、解题报告.....9
2.1 最近点对问题解题报告.....9
2.1.1 题目描述.....9
2.1.2 算法设计.....9
2.1.3 源程序及注释.....12
2.1.4 本题小结.....16
2.2 大整数乘法问题解题报告.....17
2.2.1 题目描述.....17
2.2.2 算法设计.....17
2.2.3 源程序及注释.....20
2.2.4 本题小结.....33
2.3 最优二分查找树问题解题报告.....35
2.3.1 题目描述.....35
2.3.2 算法设计.....35
2.3.3 源程序及注释.....36
2.3.4 本题小结.....38
2.4 每对结点之间的最短路径问题解题报告.....39
2.4.1 题目描述.....39
2.4.2 算法的设计.....39
2.4.3 源程序及注释.....41
2.4.4 本题小结.....44
三、心得体会.....46

一、实验情况总览

完成：第一部分的四道题。

通过：全部通过。

第一题：

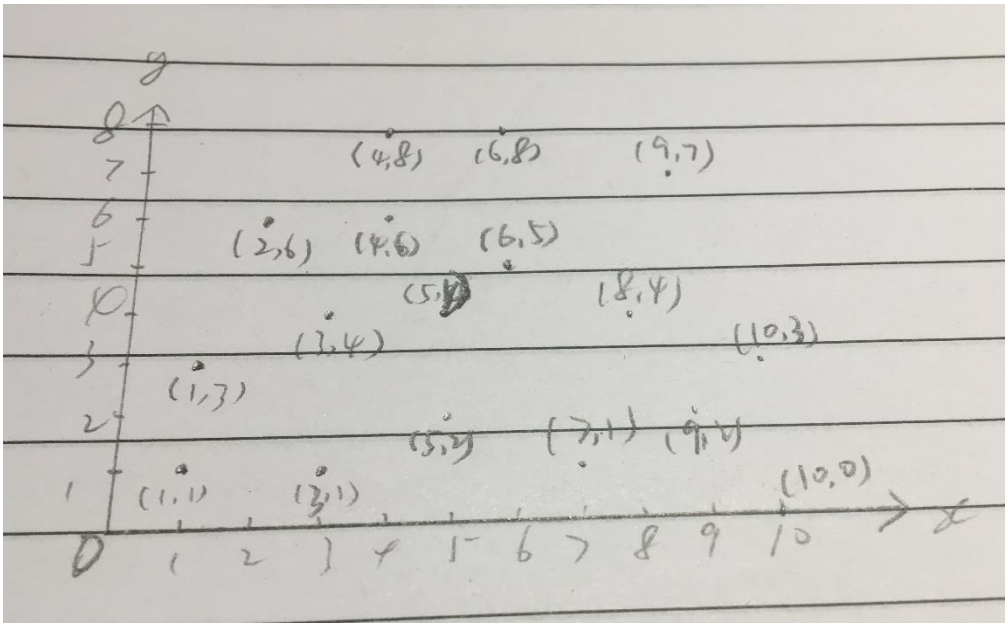

图 1-1 最近点对

```
The smallest distance is 1.414214
请按任意键继续. . .
```

图 1-2 最近点对

第二题：

```
Please input a: 9223372036854775807

Please input b: 1234567891111

Which operator ? (+, -, *) *
a * b = 11386878964471969137416693151577

Continue ? (y/n)
y

Please input a: 11386878964471969137416693151577

Please input b: 11386878964471969137416693151577

Which operator ? (+, -, *) *
a * b = 129661012551534224181237491966943866835916846940669172697586929

Continue ? (y/n)
y

Please input a: 129661012551534224181237491966943866835916846940669172697586929

Please input b: 11386878964471969137416693151577

Which operator ? (+, -, *) *
a * b = 1476434256335201019505915952536059041319057206399521214104693657122061789167297053187930937033

Continue ? (y/n)
y

Please input a: 1476434256335201019505915952536059041319057206399521214104693657122061789167297053187930937033

Please input b: 1476434256335201019505915952536059041319057206399521214104693657122061789167297053187930937033

Which operator ? (+, -, *) +
a + b = 29528685126704020390118319050721180826381144127990424282093873142441235783345941063758618740666

Continue ? (y/n)
y
```

图 1-3 大整数运算

```
Please input a: 1476434256335201019505915952536059041319057206399521214104693657122061789167297053187930937033

Please input b: 29528685126704020390118319050721180826381144127990424282093873142441235783345941063758618740666

Which operator ? (+, -, *) -
a - b = -1476434256335201019505915952536059041319057206399521214104693657122061789167297053187930937033
```

图 1-4 大整数运算

第三题：

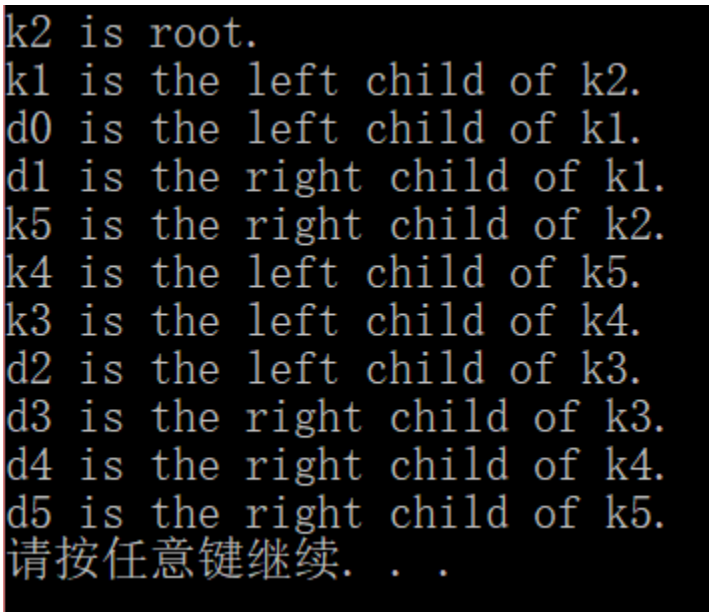

图 1-5 最优二分查找树

第四题：

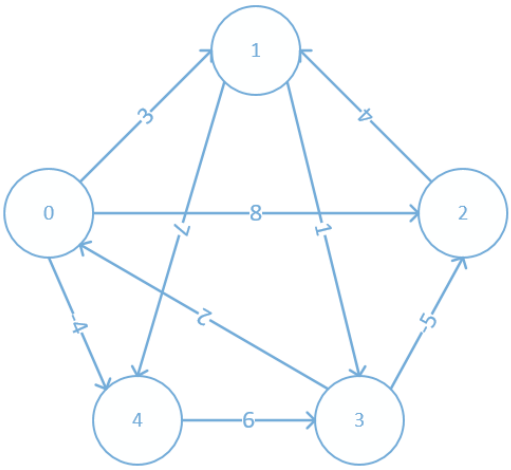

图 1-6 Flyod-Warshall

```
The shortest path distance from 0 to 1 : 1
0->4->3->2->1

The shortest path distance from 0 to 2 : -3
0->4->3->2

The shortest path distance from 0 to 3 : 2
0->4->3

The shortest path distance from 0 to 4 : -4
0->4

The shortest path distance from 1 to 0 : 3
1->3->0

The shortest path distance from 1 to 2 : -4
1->3->2

The shortest path distance from 1 to 3 : 1
1->3

The shortest path distance from 1 to 4 : -1
1->3->0->4

The shortest path distance from 2 to 0 : 7
2->1->3->0

The shortest path distance from 2 to 1 : 4
2->1

The shortest path distance from 2 to 3 : 5
2->1->3

The shortest path distance from 2 to 4 : 3
2->1->3->0->4

The shortest path distance from 3 to 0 : 2
3->0

The shortest path distance from 3 to 1 : -1
3->2->1
```

图 1-7 Flyod-Warshall

```
The shortest path distance from 3 to 2 : -5
3->2

The shortest path distance from 3 to 4 : -2
3->0->4

The shortest path distance from 4 to 0 : 8
4->3->0

The shortest path distance from 4 to 1 : 5
4->3->2->1

The shortest path distance from 4 to 2 : 1
4->3->2

The shortest path distance from 4 to 3 : 6
4->3
```

图 1-8 Flyod-Warshall

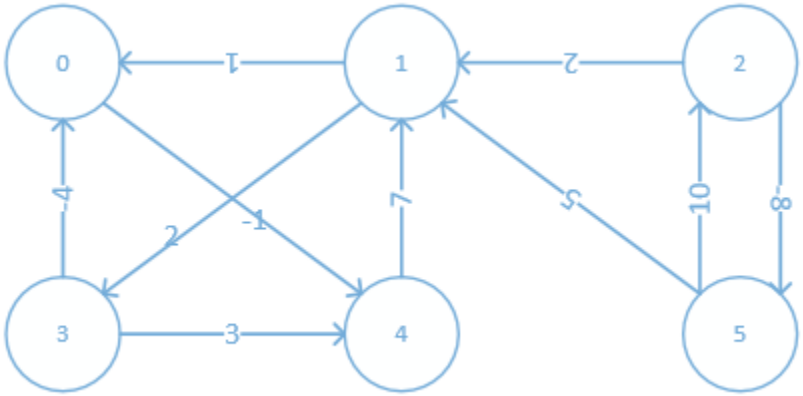

图 1-9 Flyod-Warshall

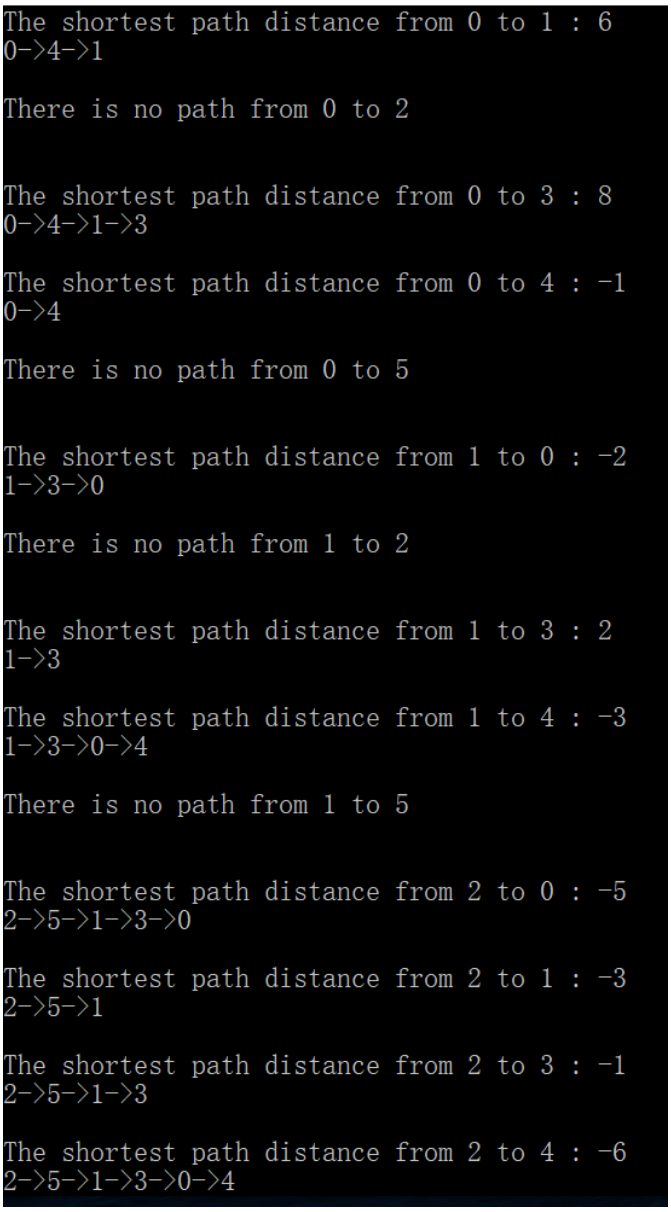

图 1-10 Flyod-Warshall

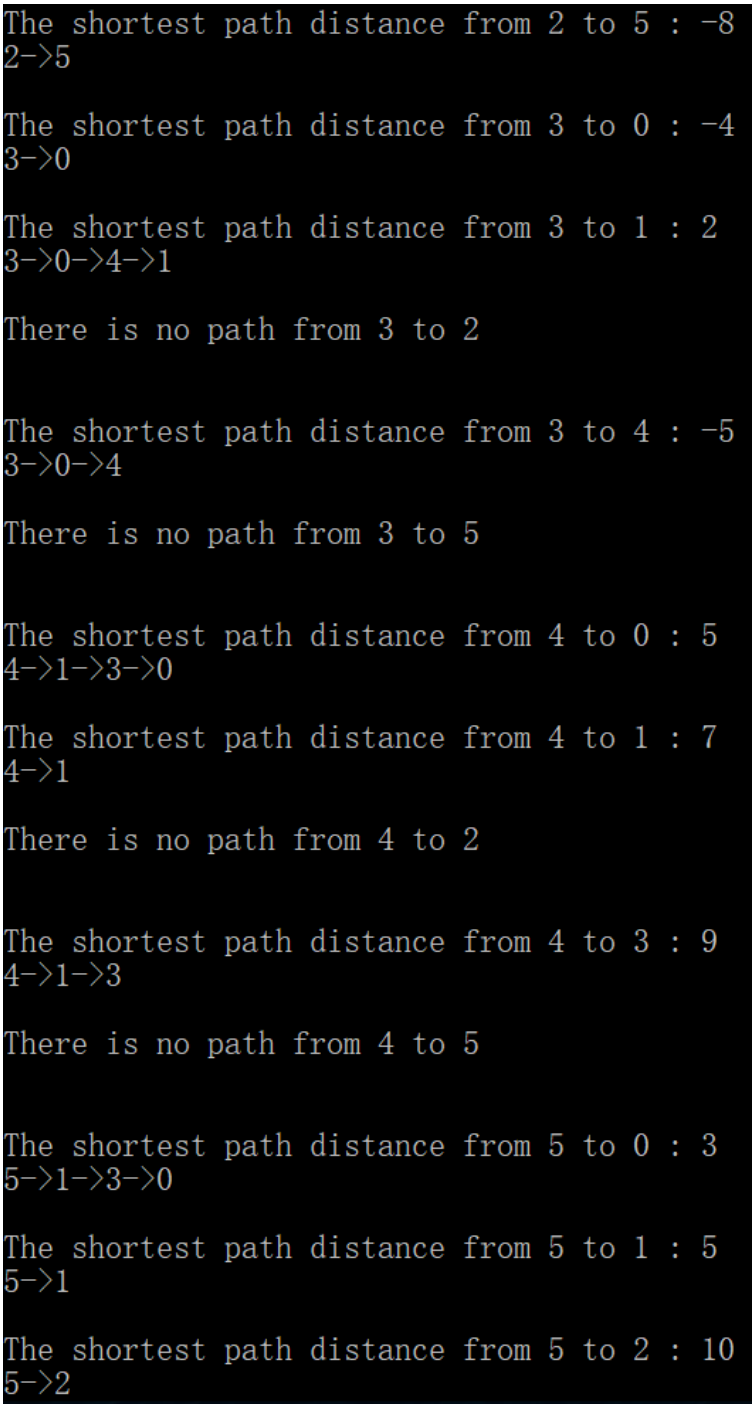

图 1-11 Flyod-Warshall

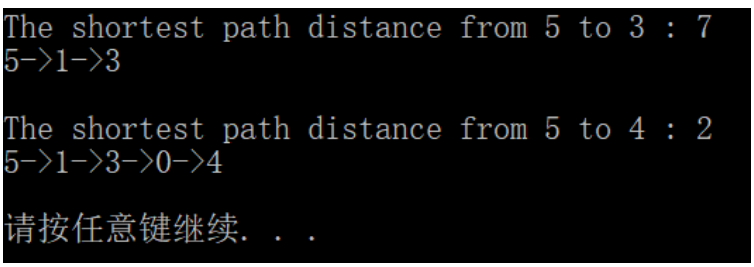

图 1-12 Flyod-Warshall

二、解题报告

2.1 最近点对问题解题报告

2.1.1 题目描述

在与联盟的战斗中连续失败后，帝国撤退到最后一个据点。 根据其强大的防御系统，帝国击退了联盟攻击的六波浪潮。 经过几个不眠之夜，联盟将军亚瑟注意到，防御系统唯一的弱点就是能源供应。 该系统由 N 个核电站充电，其中任何一个都会使系统失效。

这位将军很快就开始命令 N 个特工人员进行突袭，这些特工人员进入了据点。 不幸的是，由于帝国空军的袭击，他们未能降落在预期位置。作为一名经验丰富的将军，亚瑟很快意识到他需要重新安排计划。他现在想知道的第一件事就是哪个特工离任何一个发电站最近。你是否可以帮助将军计算特工和核电站之间的最小距离？（POJ3714）

2.1.2 算法设计

题目的理解与分析：

本题就是要解决最近点对问题，这个最近距离指的是欧几里得距离。

算法设计思路：

采用分治算法来解决这个问题，

首先输入为 P（包含点），X（点的横坐标），Y（点的纵坐标）。我们要对 X 和 Y 中的点进行排序，是 x，y 坐标单调递增。而且要使用“预排序”来维持 X，Y 中点的次序，而不是每次递归调用中都进行排序。

当 P 中的元素个数小于等于 3 时，采用直接遍历的方式求出任意两个点间的距离，找出最近距离。当 P 中的元素个数大于 3 时，采用如下分治模式：

分解：找出一条垂直线 l，将 P 集合分成 Pl 和 Pr，使得两个集合中点的个数基本相等，Pl 中的点都在 l 上或者 l 的左侧，类似地，Pr 中的点都在 l 上或者 l

的右侧。同理，X 被划分为两个数组 Xl 和 Xr，Xl 和 Xr 分别包含 Pl，Pr 中的点，类似地，Y 也是如此。

解决：将 P 分解为 Pl 和 Pr 后，进行两次递归调用。一次找出 Pl 中的最近点对，一次找出 Pr 中的最近点对。得到 Pl 和 Pr 中最近点对距离 δl 和 δr。并且取 δ=min（δl，δr）。

合并：算法确定是否存在距离小于 δ 的一个点对，一个点位于 Pl 中，另一个点位于 Pr 中。如果存在这样一个点对，那么点对中的两个点与直线 l 的距离必定在 δ 单位内。为了找出这样的点对：

1. 建立数组 Y’，使得它只包含区域内的点。
2. 对 Y’中的点 p，算法试图找出 Y’中距离 p 所在单位在 δ 内的点。（仅需考虑紧随 p 后的 7 个点）。记录 Y’的所有点对中最近的距离 δ’。
3. 如果 δ’＜ δ，则返回 δ’。否则返回 δ。

如下图蓝色区域内至多有八个点。

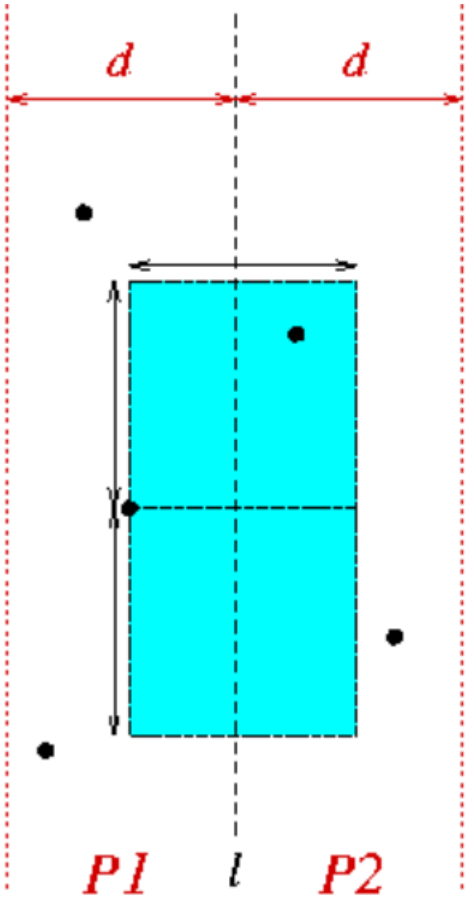

图 2-1 中间区域图解

算法的正确性证明在算法导论上给的很详细，在此不赘述。

算法时间复杂度：

排序算法时间复杂度为 O（nlogn），进行递归时，每次对 Y 排序，因此 T（n）可以表示为：

$$T（n）=2T（n/2）+O（nlogn）$$

可以看出，算法时间复杂度为 O（$nlogn^2$）

伪代码：

```
Closest_Pair_Points (Px, Py)
Px 和 Py 按照 x，y 坐标增序排列
if N ≤ 3 then
   return closestdist points of Px，Py using brute-force algorithm
else
   Xl = { points of Px from 1 to ⌈N/2⌉ }
   Xr = { points of Px from ⌈N/2⌉+1 to N }
   Xm = Px(⌈N/2⌉)x
   Yl = { p ∈ Py : px ≤ Xm }
   Yr = { p ∈ Py : px > Xm }
   dL = Closest_Pair_Points (Xl, Yl)
   dR = Closest_Pair_Points (Xr, Yr)
   dmin = min(dL, dR)
   Y' = { p ∈ Py : |Xm - px| < dmin }
   closestdist = dmin
   for i = 1 to |Y'| - 1
      k = i + 1
      while k ≤ |Y'| and Y'(k)y - Y'(i)y < dmin
         if |Y'(k) - Y'(i)| < closestdist then
            closestdist = |Y'(k) - Y'(i)|
         k = k + 1
```

return closestdist

2.1.3 源程序及注释

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>

struct Point
{
    int x, y;
};

//对X排序
int compare_x(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->x - p2->x);
}

//对Y排序
int compare_y(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->y - p2->y);
}

//计算两个点的距离
float distance(Point p1, Point p2)
```

```
{
        return sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y));
}

//找到两个浮点数中的最小值
float min(float x, float y)
{
    return (x < y) ? x : y;
}

//遍历点（小于等于3个），求最小距离
float check_few_points(Point P[], int n)
{
    float min = FLT_MAX;//初始化min
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            if (distance(P[i], P[j]) < min)
                min = distance(P[i], P[j]);
    return min;
}

//找到最近点对（区域限制在中间的长条）
float mid_closest(Point m_points[], int size, float d)
{
    int i,j;
    float min = d; //初始化min

    qsort(m_points, size, sizeof(Point), compare_y);//对Y排序
```

```
    //找到最近点对
    for (i = 0; i < size; i++)
        for (j = i + 1; j < size && (m_points[j].y - m_points[i].y) < min; j++)
            if (distance(m_points[i], m_points[j]) < min)
                min = distance(m_points[i], m_points[j]);

    return min;
}

//将P按照x增序排列
float closest_dist(Point P[], int n)
{
    //少于3个点
    if (n <= 3)
        return check_few_points(P, n);

    int mid = n / 2;
    Point midPoint = P[mid];

    //在左边和右边各自找到最近点对
    float dl = closest_dist(P, mid);
    float dr = closest_dist(P + mid, n - mid);

    //找出dl和dr中的较小值
    float d = min(dl, dr);

    //建立数组保存中间区域的点
    Point m_points[10];
    int i,j = 0;
    for (i = 0; i < n; i++)
```

```
            if (abs(P[i].x - midPoint.x) < d)
            {
                m_points[j] = P[i];
                j++;
            }

        //找到中间区域的最近点对
        return min(d, mid_closest(m_points, j, d));
}

//找出最近点对
float closest(Point P[], int n)
{
        qsort(P, n, sizeof(Point), compare_x);//按照x增序排列

        return closest_dist(P, n);
}



int main()
{
        Point P[] = { { 1, 1 },{ 1, 3 },{ 2, 6 },{ 3, 1 },{ 3, 4 },{ 4, 6 },
                                { 4, 8 },{ 5, 2 },{ 5, 4 },{ 6, 5 },{ 6, 8 },{ 7, 1 },
                                { 8, 4 },{ 9, 2 },{ 9, 7 },{ 10, 0 },{ 10, 3 }
                        };//点的集合
        int n = sizeof(P) / sizeof(P[0]);
        printf("The smallest distance is %f \n", closest(P, n));
        return 0;
}
```

2.1.4 本题小结

本题主要是分治算法的应用。在求解原问题时，将原问题分为两个子问题，求解左边和右边的最小点对。当求出左边和右边之中最小点对的距离后，再考虑跨越边界的点。而考虑跨越边界的点又有很强的技巧性，我们只考虑少量跨越边界的点，这一点已经在上面的解析中呈现出来了。通过分治算法，我们有效地将算法时间复杂度降低。实现技术上的要点首先就是要先对 X,Y 数组进行排序，在找到左边和右边之中最小点对的距离后，我们再来考虑跨越边界的点。在考虑边界点时，我们不需要对所有的点对进行遍历，我们对每个点，最多仅仅考虑其后的 7 个点。这一点非常重要，因为我们可以在常数时间内完成这个操作。

本次实验加深了我对分治算法的理解，分治算法将原问题的规模缩小，使得子问题能在常数时间内解决。而在实现分治算法的过程中，一些必要的细节也能将算法的时间复杂度降低。书本上得来的分治思想只是知识，通过实验能够将这种思想实现出来，变成自己的东西。

2.2 大整数乘法问题解题报告

2.2.1 题目描述

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(n2)。

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：98349759721308023457910234985703 45。对大整数的算术运算，显然常规程序语言是无法直接表示的。编程实现大整数的加、减、乘运算，需考虑操作数为 0、负数、任意位等各种情况。

2.2.2 算法设计

题目的理解与分析：

题目要求实现大整数的运算，大整数的表示用字符串来表示，加减运算的时候将运算变成位运算，最后结果转换成字符串输出即可。对于乘法运算，运用分治算法的思想，将原问题规模缩小，使得其时间复杂度低于 O(n2)。

算法的设计思路：

对于加法运算，首先算出每位（保证先加到两个数中位数少的那个数的最高位）相加后的结果，将结果保存在一个数组 S 中，carry 表示进位，S[i]=（A[i]+B[i]+carry）%10,carry=（A[i]+B[i]+carry）/10.然后就只对位数高的那个数做相似的运算即可。

对于减法运算，首先算出每位（保证先加到两个数中位数少的那个数的最高位）相减后的结果，将结果保存在一个数组 S 中，carry 表示借位。如果 A[i]＜B[i]+carry，那么 S[i]=（A[i]+10-carry-B[i]）%10,carry=1；否则 S[i]=（A[i] -carry-B[i]）%10,carry=0. 然后就只对位数高的那个数做相似的运算即可。

对于乘法运算，采用分治法的思想，将要处理的数分成左右两个部分，然后对左右两个部分相乘，得到最后三部分的结果，最后直到要处理的数只有一位的时候。具体过程如下所示：（Karatsuba 乘法）

$$
\begin{aligned}
x1 &= aL \;\; bL \\
x2 &= aR \;\; bR \\
x3 &= (aL + aR) \;\; (bL + bR)
\end{aligned}
$$

$$
\begin{array}{lll}
 & aL & aR \\
x & bL & bR \\
\hline
aL \;\; bL & aL \;\; bR + aR \;\; bL & aR \;\; bR \\
x1 & x3 - x1 - x2 & x2
\end{array}
$$

图 2-2 Karatsuba 乘法

伪代码和流程图：

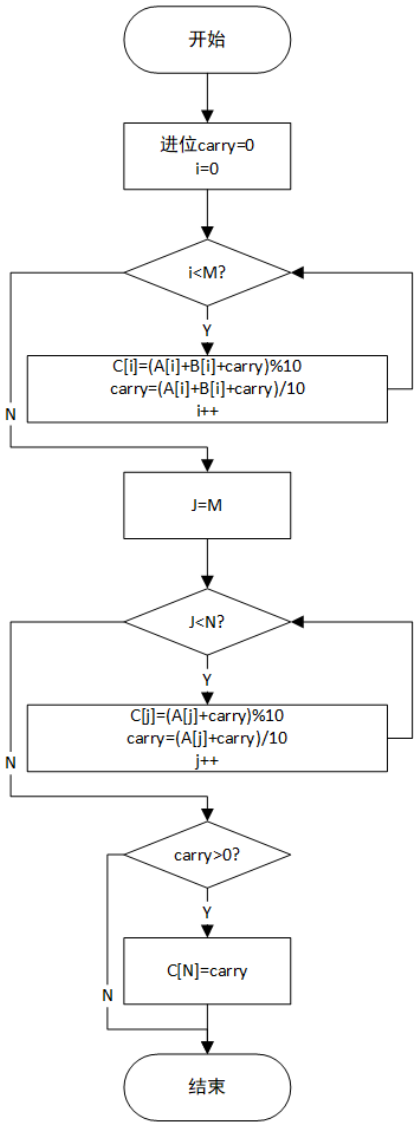

图 2-3 大整数加法

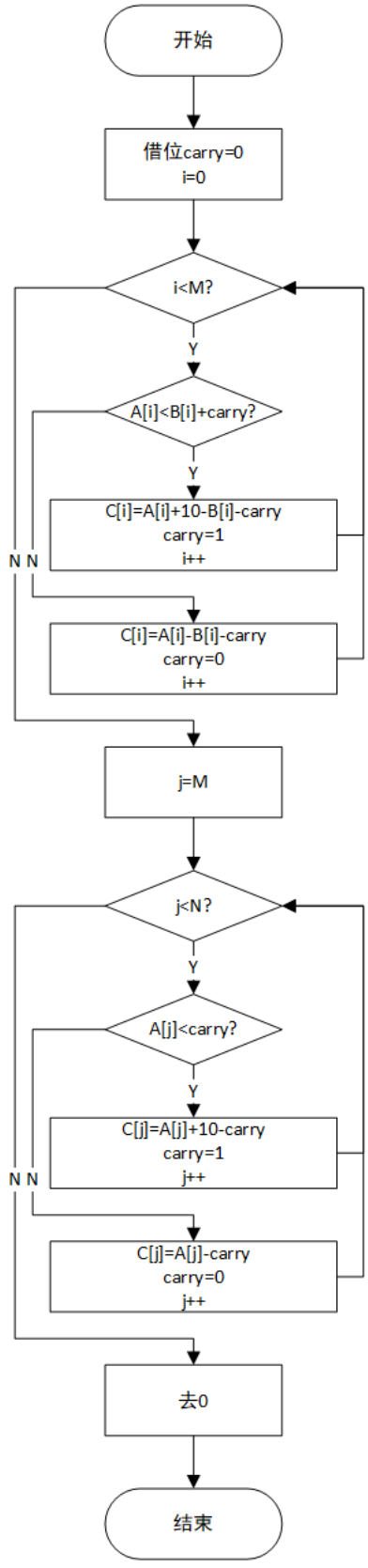
开始
借位carry=0
i=0
i<M?
Y
A[i]<B[i]+carry?
Y
C[i]=A[i]+10-B[i]-carry
carry=1
i++
N
N
C[i]=A[i]-B[i]-carry
carry=0
i++
j=M
j<N?
Y
A[j]<carry?
Y
C[j]=A[j]+10-carry
carry=1
j++
N
N
C[j]=A[j]-carry
carry=0
j++
去0
结束

图 2-4 大整数减法

大整数乘法：

Algorithm Karatsuba(a,b)**:**

if a or b has one digit, **then:**

return a * b.

else:

Let n be the number of digits in max{a, b}.

Let a_L and a_R be left and right halves of a.

Let b_L and b_R be left and right halves of b.

Let x_1 hold Karatsuba(a_L, b_L).

Let x_2 hold Karatsuba($a_L + a_R$, $b_L + b_R$).

Let x_3 hold Karatsuba(a_R, b_R).

return $x_1*10^n + (x_2 - x_1 - x_3)*10^{n/2} + x_3$.

end of if

算法时间复杂度为：O（$3n^{log3}$）≈O（$3n^{1.585}$）

2.2.3 源程序及注释

```
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

string Add(string a, string b);
string Sub(string a, string b);
string Mul(string a, string b);
void show_result(string a, string b, char opt);
string make_equal_length(string num, int n);
void swap(string& a, string& b);
```

```
int main(int argc, char* argv[])
{
    string a, b;
    char opt, flag;
    while (true)
    {
        a = "";
        b = "";
        cout << "Please input a: ";
        cin >> a;
        cout << endl;
        cout << "Please input b: ";
        cin >> b;
        cout << endl;
        cout << "Which operator ? (+, -, *) ";
        cin >> opt;
        cout << "a " << opt << " b = ";
        show_result(a, b, opt);
        cout << endl << "Continue ? (y/n)" << endl;
        cin >> flag;
        cout << endl;
        if (flag == 'n' || flag == 'N')
            break;
    }
    return 0;
}

string make_equal_length(string a, int n)
{
```

```
    if (a == "0")
        return a;
    else
    {
        for (int i = 0; i < n; i++)//后面补0
            a += '0';
    }
    return a;
}

void swap(string& a, string& b)
{
    string temp;
    temp = a;
    for (int i = 0; i < a.length(); i++)
        temp[i] = a[i];
    for (int j = 0; j < a.length(); j++)
        a[j] = b[j];
    for (int k = 0; k < a.length(); k++)
        b[k] = temp[k];
}

string Mul(string a, string b)
{
    string mul = "";
    bool is_neg = false;
    int first_a = 0, first_b = 0, n;
    int size_a = a.length(), size_b = b.length();
    //不计算符号
    if (a[0] == '-')
```

```
        size_a--, first_a++;
if (b[0] == '-')
        size_b--, first_b++;

int size = (size_a > size_b) ? size_a : size_b;//实际位数

if (a[0] == '-' && b[0] != '-')
        is_neg = true;
else if (a[0] != '-' && b[0] == '-')
        is_neg = true;

if (size == 0)
        return NULL;
else if (size == 1)
{
        int i = (a[first_a] - '0') * (b[first_b] - '0');
        stringstream stream;
        string str1;
        stream << i;
        stream >> str1;
        mul += str1;
        if (is_neg)
                mul = '-' + mul;
        return mul;
}
//Karatsuba乘法
string aL = a.substr(first_a, size / 2);
string aR = a.substr(size / 2 + first_a, size - size / 2);
string bL = b.substr(first_b, size / 2);
string bR = b.substr(size / 2 + first_b, size - size / 2);
```

```
    string x1 = Mul(aL, bL);
    string x2 = Mul(aR, bR);
    //(aL-aR)(bR-bL)+aLbL+aRbR=aLbR+aRbL
    string temp_1 = Sub(aL, aR);
    string temp_2 = Sub(bR, bL);
    string temp_3 = Mul(temp_1, temp_2);
    string temp_4 = Add(temp_3, x1);
    string mid = Add(temp_4, x2);
    string Left = make_equal_length(x1, size);
    string Mid = make_equal_length(mid, size / 2);
    string Left_t = Add(Left, Mid);
    mul = Add(Left_t, x2);
    if (is_neg)
        mul = '-' + mul;
    return mul;
}

string Add(string a, string b)
{
    string temp1 = a, temp2 = b;
    string sum, temp;
    int size = (a.length() > b.length()) ? a.length() : b.length();//较长字符串
    int sum_t, carry = 0;
    bool overflow = false, is_neg = false;
    int i, j;
    //符号位
    if (a[0] == '-' && b[0] == '-')
    {
        a[0] = '0';
        b[0] = '0';
```

```
        is_neg = true;
    }
    else if (a[0] == '-' && b[0] != '-')
    {
        a[0] = '0';
        return Sub(b, a);//换减法
    }
    else if (b[0] == '-' && a[0] != '-')
    {
        b[0] = '0';
        return Sub(a, b);//a-b
    }
    //倒置
    for (i = a.length() - 1, j = 0; i >= 0; i--, j++)
        temp1[j] = a[i];
    for (i = b.length() - 1, j = 0; i >= 0; i--, j++)
        temp2[j] = b[i];
    //高位补零
    if (a.length() < b.length())
    {
        sum = b;//结果先设为b
        temp = b;
        for (i = a.length(); i < b.length(); i++)
            temp1 += '0';
    }
    else if (a.length() > b.length())
    {
        sum = a;//结果先设为a
        temp = a;
        for (i = b.length(); i < a.length(); i++)
```

```
            temp2 += '0';
    }
    else
    {
        sum = a;
        temp = a;
    }

    //每一位相加
    for (i = 0; i < size; i++)
    {
        sum_t = temp1[i] - '0' + temp2[i] - '0' + carry;

        //如果产生进位
        if (sum_t > 9)
        {
            //如果产生溢出
            if (i == (size - 1))
                overflow = true;
            carry = 1;
            sum[i] = sum_t - 10 + '0';
        }
        else
        {
            carry = 0;
            sum[i] = sum_t + '0';
        }
    }
    if (overflow)
    {
```

```
            string str;//进位
            str = carry + '0';
            sum += str;
            temp += '0';//保持位数相同
            size++;
        }
        //再将sum倒置
        for (i = 0, j = sum.length() - 1; i < sum.length(), j >= 0; i++, j--)
        {
            temp[i] = sum[j];
        }
        for (i = 0; i < sum.length(); i++)
            sum[i] = temp[i];
        //如果sum是负数
        if (is_neg)
        {
            return "-" + sum;
        }
        return sum;
}

string Sub(string a, string b)
{
        string temp1 = a, temp2 = b;
        string mins, temp;
        int    size = (a.length() > b.length()) ? a.length() : b.length();//较长字符串
        int minus_t, dec = 0;
        bool overflow = false, is_neg = false;
        int     i, j;
```

```
//判断符号位
if (a[0] == '-' && b[0] == '-')
{
    a[0] = '0';
    b[0] = '0';
    return Sub(b, a);//-a+b
}
else if (a[0] == '-' && b[0] != '-')
{
    string str = "-";
    b = str + b;
    return Add(a, b);//-a-b
}
else if (a[0] != '-' && b[0] == '-')
{
    b[0] = '0';
    Add(a, b);//a+b
}
//倒置
for (i = a.length() - 1, j = 0; i >= 0; i--, j++)
    temp1[j] = a[i];
for (i = b.length() - 1, j = 0; i >= 0; i--, j++)
    temp2[j] = b[i];
//高位补零
if (a.length() < b.length())
{
    mins = b;//作为被减数
    temp = b;
    for (i = a.length(); i < b.length(); i++)
        temp1 += '0';
```

```
}
else if (a.length() > b.length())
{
    mins = a;//作为被减数
    temp = a;
    for (i = b.length(); i < a.length(); i++)
        temp2 += '0';
}
else
{
    mins = a;
    temp = a;
}
//判断大小
for (i = size - 1; i >= 0; i--)
{
    if (temp1[i] > temp2[i])
    {
        break;
    }
    else if (temp1[i] < temp2[i])
    {
        is_neg = true;//结果为负
        swap(temp1, temp2);
        break;
    }
}
//每一位相减
for (i = 0; i < size; i++)
{
```

```
            minus_t = (temp1[i] - '0') - (temp2[i] - '0') + dec;
            //如果产生退位
            if (minus_t < 0)
            {
                //如果结果是负数
                if (i == (size - 1))
                    overflow = true;
                dec = -1;
                mins[i] = minus_t + 10 + '0';
            }
            else
            {
                dec = 0;
                mins[i] = minus_t + '0';
            }
        }
        //再将mins倒置，得出结果
        for (i = 0, j = size - 1; i < size, j >= 0; i++, j--)
            temp[i] = mins[j];

        for (i = 0; i < size; i++)
            mins[i] = temp[i];

        if (is_neg)//为负
        {
            return "-" + mins;
        }
        return mins;
    }
```

```
void show_result(string a, string b, char opt)
{
    string result;
    int first_a = 0, first_b = 0;
    int i, n;
    //处理负号
    if (a[0] == '-')
    {
        first_a = 1;
        a[0] = '0';
    }
    if (b[0] == '-')
    {
        first_b = 1;
        b[0] = '0';
    }
    int size = (a.length() - first_a > b.length() - first_b) ? a.length() - first_a :
b.length() - first_b;
    //较小数高位补零
    if (a.length() - first_a < b.length() - first_b)
    {//a小
        for (i = a.length() - first_a; i < b.length() - first_b; i++)
            a = '0' + a;
    }
    else if (a.length() - first_a > b.length() - first_b)
    {//b小
        for (i = b.length() - first_b; i < a.length() - first_a; i++)
            b = '0' + b;
    }
```

```
//高位补零
for (i = 0; i < size; i++)
{
    if (size < pow(2, i) || size == pow(2, i))
    {
        n = i;
        break;
    }
}
for (i = size; i < pow(2, n); i++)
{
    a = '0' + a;
    b = '0' + b;
}
//改回负号
if (first_a == 1)
    a[0] = '-';
if (first_b == 1)
    b[0] = '-';
switch (opt)
{
case '+':
    result = Add(a, b);
    break;
case '-':
    result = Sub(a, b);
    break;
case '*':
    result = Mul(a, b);
    break;
```

```
    default:
        result = Mul(a, b);
        break;
    }
    //去掉高位多余的零
    string::iterator it = result.end();
    //如果结果为零
    if (result[result.length() - 1] == 0)
        result = "0";
    else
    {
        string::iterator it = result.begin();
        if (result[0] == '-')
            it++;
        for (; *it == '0';)
        {
            if (*it == '0')
                it = result.erase(it);
            else ++it;
        }
    }
    cout << result << endl;
}
```

2.2.4 本题小结

题目要求实现大整数的运算，大整数的表示用字符串来表示，加减运算的时候将运算变成位运算，最后结果转换成字符串输出即可。对于乘法运算，运用分治算法的思想，将原问题规模缩小，使得其时间复杂度低于 O(n^2)。

对于加法运算，首先算出每位（保证先加到两个数中位数少的那个数的最高位）相加后的结果，将结果保存在一个数组 S 中，carry 表示进位，S[i]=（A[i]+B[i]+carry）%10,carry=（A[i]+B[i]+carry）/10.类似地，在上面的算法设计思想已经阐述过，在此不赘述。对于减法运算，在上面的设计思想中已经说明过了。

对于乘法运算，设计要点就是采用分治法的思想，其中必要的细节就是还要在其中加入字符串相加的算法，采用分治法的思想，能够有效地将算法的时间复杂度降低到小于 O（n^2）。

经过这次实验，我对分治法有了更深一层的理解，又是一次基于分治法的实验，经过几次的实践，我的能力得到了一定的提升，对于以后类似的问题，都能通过这一思想来解决。

2.3 最优二分查找树问题解题报告

2.3.1 题目描述

最优二叉搜索树，有时称为权重平衡二叉树，它是二叉搜索树，并且它为给定的访问序列提供尽可能小的搜索时间（或访问概率）。即对于一个给定的概率集合，构造的一棵期望搜索代价最小的二叉搜索树称之为最优二叉搜索树。

2.3.2 算法设计

题目的理解与分析：

题目要求构造一棵二叉搜索树，并且要使它的搜索代价最小，这个在题目描述中已经呈现。

算法的设计思路：

我们可以看出二叉搜索树问题具有最优子结构：如果一棵最优二叉搜索树 T 有一棵包含关键字 ki，…，kj 的子树 T’，那么 T’必然是包含关键字 ki，…，kj。

从 root[1][n]开始，里面的值即为根节点。然后开始递归，每次递归时，取当前数组元素，设其为 temp。若 temp 不等于 i，则 root[i][temp-1]为 t 的左孩子节点，然后递归调用，参数为 i，temp-1；若 temp 等于 i，则表明 k 节点已经结束，下面只有 d 节点，说明 d[temp-1]为 k[temp]的左孩子节点。类似地，若 temp 不等于 j，则 root[temp+1][j]为 t 的右孩子节点，然后递归调用，参数为 temp+1，j；若 temp 等于 j，则表明 k 节点已经结束，下面只有 d 节点，说明 d[temp]为 k[temp]的右孩子节点。

伪代码：

```
Optimal_bst(i, j)
    if i == 1 and j == N
        print(k root[i][j] is root)
    if i > j
```

```
        return
    temp = root[i][j]
    if temp!= i
        print(k root[i][temp - 1] is the left child of k(temp))
        Optimal_bst (i, temp - 1)
    else
        print(d(temp-1) is the left child of k(temp))
    if temp!= j
        print(k root[temp + 1][j] is the right child of k(temp))
        Optimal_bst (temp + 1, j)
    else
        print(d (temp) is the right child of k(temp))
```

2.3.3 源程序及注释

```
#include<iostream>

using namespace std;

const int N = 5;

int root[6][6] = {//root矩阵
    0,0,0,0,0,0,
    0,1,1,2,2,2,
    0,0,2,2,2,4,
    0,0,0,3,4,5,
    0,0,0,0,4,5,
    0,0,0,0,0,5
};
```

```
void optimal_bst(int i, int j) {
    if (i > j)
        return;
    if (i == 1 && j == N)
        cout << "k" << root[i][j] << " is root." << endl;//找到根
    int t = root[i][j];
    if (i != t)//将树分为两部分
    {
        cout << "k" << root[i][t - 1] << " is the left child of k" << t <<"."<< endl;
        optimal_bst(i, t - 1);//搜索左树
    }
    else
    {
        cout << "d" << t - 1 << " is the left child of k" << t << "." << endl;//k节点已经找完
    }
    if (j != t)//将树分为两部分
    {
        cout << "k" << root[t + 1][j] << " is the right child of k" << t << "." << endl;
        optimal_bst(t + 1, j);//搜索右树
    }
    else
    {
        cout << "d" << t << " is the right child of k" << t << "." << endl;//k节点已经找完
    }
}

int main()
```

```
{
    optimal_bst(1, N);
    return 0;
}
```

2.3.4 本题小结

本题主要是了解递归的过程。根据题目中给出的 root 矩阵，来构造出一棵树。

题目要求我们对 root 矩阵存的元素有较清晰的认识，能够根据已有的矩阵表来反向推理出各个节点的位置。在对 root 矩阵有了较深的理解后，还要设计算法来打印出结果。解决本题主要就是要用好递归，清楚下一次递归的条件。技术上的要点就是要清楚 root 矩阵当前元素将问题划分成了哪两块，这对下一次递归是至关重要的。

本题虽然比较简单，但是也要深入理解课本中的最优二叉树是从何而来的，不能仅仅满足于根据 root 矩阵来重构最优二叉树。课本中前面讲述的最优二叉书问题很好地运用到了动态规划，这是我认为更为重要的部分。动态规划作为算法重要的一部分，能够很好地解决很多问题。而本题让我理解了 root 矩阵是如何来的，因为要从 root 矩阵反推回去。

2.4 每对结点之间的最短路径问题解题报告

2.4.1 题目描述

Floyd-Warshall 算法是在具有正或负边权重（但没有负环路）的加权图中寻找最短路径的算法。单次执行算法将找到所有顶点对之间的最短路径的长度。虽然它不返回路径本身的细节，但可以通过对算法的简单修改来重建路径。本题要求打印最短路径。

2.4.2 算法的设计

题目的理解与分析：

本题要求实现 Floyd-Warshall 算法，并在此基础上，最终能够打印最短路径，要求对书中的伪代码进行修改。

算法设计思路：

使用动态规划的方法解决问题。

注意到，若考虑从 i 到 j 的所有中间节点均取自于集合{1,2，…，k}的路径，设 p 为其中权重最小的路径。算法利用路径 p 和从 i 到 j 的所有中间节点均取自于集合{1,2，…，k}的路径之间的关系，该关系依赖于 k 是否是 p 上的一个节点。若 k 不是，则显然最短路径是取自于集合{1,2，…，k}的路径；若 k 是，则将路径分解为 i 到 k（p1），k 到 j（p2），得到 p1 是 i 到 k 中间节点均取自于{1,2，…，k-1}的路径，同理，p2 是 k 到 j 中间节点均取自于{1,2，…，k-1}的路径。设 dij 为从 i 到 j 的所有中间节点均取自于集合{1,2，…，k}的一条最短路径的权重，有：

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

对任何路径，所有的中间节点均属于集合{1,2，…，k}，矩阵 D（n）=（dij（n））给出的就是最终答案。

为了打印出最短路径，需要构造前驱矩阵 Π，这个过程在计算 D（k）的同时完成，Π=Π（n）并且定义 πij（k）为从结点 i 到结点 j 的一条所有中间结点都取自于集合{1,2，…，k}的最短路径上 j 的前驱结点。

当 k=0 时，从 i 到 j 的一条最短路径没有中间结点，因此：

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

对于 k≥1，如果考虑路径 i 到 k 到 j（k≠j），则所选 j 的前驱与 k 到 j 的一条中间结点都取自于集合{1,2，…，k}的最短路径 j 的前驱是一样的。否则，j 的前驱与 i 到 j 的一条中间结点都取自于集合{1,2，…，k}的最短路径 j 的前驱是一样的：

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

构造完了前驱矩阵 Π，可以通过递归打印路径。如果 i 等于 j，则打印 i；如果 πij 等于 NIL，则 i 到 j 间的路径不存在。否则递归调用，参数为 Π，i，πij，打印 j。

伪代码：

```
Floyd-Warshall(W)
    n=W.rows
    D(0)=W
    for k=1 to n
        let D(k)=(dij(k)) be a new n*n matrix
        for i=1 to n
            for j=1 to n
                dij(k)=min(dij(k-1),dik(k-1)+dkj(k-1))
```

```
                    Pij=(Dij > Dik+ Dkj)? Pkj:Pij
    return D(n)

Print-All-Pairs-Shortest-Path(Π,i,j)
    if i==j
        print i
    else if πij==NIL
        print “there is no path”
    else Print-All-Pairs-Shortest-Path(Π,i, πij)
        print j
```

2.4.3 源程序及注释

```
#include<iostream>
#include<fstream>

using namespace std;

const int N = 6;//点的个数
const int INF = 10000000;//无限大
const int NIL = -1;//无前驱节点

int D[N][N];//最短路径矩阵
int Prior[N][N];//前驱矩阵

void initD(char *filename, int n) {//初始化D
    int i, j, k;
    int edge;//边的数量

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

```
            for (j = 0; j < n; j++) {
                if (i == j) {
                    D[i][j] = 0;//对角线
                }
                else {
                    D[i][j] = INF;
                }
            }
        }

        ifstream file(filename);//使用文件流输入
        file >> edge;//读取边的数量
        for (k = 0; k < edge; k++) {//读取边的端点以及权值
            file >> i >> j;
            file >> D[i][j];
        }
    }

    void initPrior(int n) {//初始化前驱矩阵
        int i, j;
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                if (i == j || D[i][j] == INF)//没有路径
                    Prior[i][j] = NIL;
                if (i != j&&D[i][j] < INF)//有路径
                    Prior[i][j] = i;
            }
        }
    }
```

```
void Floyd_Warshall(int n) {//Floyd-Warshall算法
    int i, j, k;
    for (k = 0; k < n; k++) {
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                if (D[i][j] > D[i][k] + D[k][j]) {//k是否在路径中
                    D[i][j] = D[i][k] + D[k][j];
                    Prior[i][j] = Prior[k][j];//计算前驱矩阵
                }
            }
        }
    }
}

void print_all_path(int i, int j) {//打印路径
    if (i == j)
        cout << i;
    else if (Prior[i][j] == NIL)//没有路径
        cout << "There is no path from " << i << " to " << j << endl;
    else {
        print_all_path(i, Prior[i][j]);
        cout << "->" << j;
    }
}

int main() {
    int i, j;

    initD("test1.txt",N);//测试文件路径 第一行为边的数量n，接下来有n行，
```

```
每行三个输入，依次为i,j,边的权值
    initPrior(N);
    Floyd_Warshall(N);

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            if (i != j&&D[i][j] < INF-1000) {//存在路径
                cout << "The shortest path distance from " << i << " to " << j
<< " : " << D[i][j] << endl;
                print_all_path(i, j);
                cout << endl << endl;
            }
            else if(i != j){//不存在路径
                cout << "There is no path from " << i << " to " << j << endl;
                cout << endl << endl;
            }
        }
    }
}
```

2.4.4 本题小结

本题要求实现 Floyd-Warshall 算法，并在此基础上，最终能够打印最短路径。在设计上，采用动态规划的思想来解决问题。算法首先利用动态规划的思想，将原问题分为两个子问题，一个为这条最短路径经过 k，另一个为这条最短路径不经过 k。在这两种情况下，最短路径取从 i 到 j 不经过 k 的最短路径和从 i 到 k 加上从 k 到 j 的路径见的最小值。这样递归求解即可得到最后最短路径的权值。然后打印路径最重要的就是如何计算前驱矩阵，同样地，计算前驱结点也被分为两种情况，当 k=0 时，从 i 到 j 的一条最短路径没有中间结点，对于 k≥1，如果考虑路径 i 到 k 到 j（k≠j），则所选 j 的前驱与 k 到 j 的一条中间结点都取自于集合

{1,2，…，k}的最短路径 j 的前驱是一样的。否则，j 的前驱与 i 到 j 的一条中间结点都取自于集合{1,2，…，k}的最短路径 j 的前驱是一样的。这样，计算出了前驱矩阵，打印路径就变得十分简单了。

本次实验加深了我对动态规划的理解，实验不仅仅要求实现书中所讲的 Floyd-Warshall 算法，还要求对其进行扩展，打印路径，在书本的基础上进行了一定的扩展，而不是局限于书本上的知识。尽管学习了动态规划，实践起来还是有一点复杂的。在实验的过程中，我能够更好地理解动态规划的思想，更重要的是如何把它运用到实践中去。

三、心得体会

算法其实是一门难度较大，技巧较多的课程，仅仅凭借上课的一点点课时是不够的，课堂上我们学到的东西十分有限，所以我们要花大量的课外时间来学好这门课程。这门课仅仅只是让我们了解一些基本的最重要的思想，实践得我们自己来做。而算法实验正好就是一种实践的形式。但由于课时的限制，算法实验的内容也十分有限，如果要想熟练掌握一些基本算法，这些课时是远远不够的。如果可能的话，算法课程应该尽量多增加一点课时，并且重在实践，只有自己亲自去尝试了，才能知道自己是否掌握了。

在学习算法的过程中，我不仅仅只满足于这些算法的具体步骤，而且对这些算法正确性的证明感兴趣。证明算法的正确性其实是一个很难的过程，我们需要严谨的逻辑以及一些适当的技巧。我认为仅仅了解算法是如何解决问题是不够的，我们还要思考这种算法是否是正确的，这种算法运用到了什么基本思想。只有当我们这样去思考了，才有可能真正地用算法的思想去解决实际问题，而不是套用已有的算法。当学习很多著名的算法时，我发现它们背后很可能有着共通的特征，比如它们可能运用了某种共同的思想，比如很多算法背后都运用了动态规划的思想。也许很多算法让我们很混乱，但是它们都是基于最基本的东西。

在做实验的过程中，这和学习算法又是两种体验了。仅仅是知道了算法如何实现的，给出了伪代码，这是不够的。因为实践才能检验你是否真正的理解了算法的核心，当你觉得自己懂了的时候，应该实现书中所给的算法来检验自己，因为写代码和了解思想是不同的事情，有时代码的实现要求你自己去实现一些必要的细节，如果你不清楚怎样去实现，很可能这个算法不能让程序的时间复杂度降低，这种算法就失去了本来的效用。在做算法实验的过程中，多多少少都遇到了一些棘手的问题，这并不是算法带来的问题，而是那些伪代码要求你自己去实现一些细节，如果你并不清楚如何实现的话，你很可能就会陷入难关。

算法是一门很深的课程，但它确实有趣，这种解决问题的思想能被运用在各种各样的实际问题中去。