

**课 程 实 验 报 告**

**课程名称： 算法设计与分析**

|  |  |
| --- | --- |
| 专业班级： | CS1601 |
| **学 号：** | U201614515 |
| 姓 名： | 吴阳民 |
| 指导教师： | 华强胜 |
| 报告日期： | 2018年12月9日 |
| 计算机科学与技术学院 | |

目 录

[一、 实验情况总览 1](#_Toc532484034)

[二、 解题报告 3](#_Toc532484035)

[2.1 最近点对问题解题报告 3](#_Toc532484036)

[2.1.1 题目描述 3](#_Toc532484037)

[2.1.2 算法设计 4](#_Toc532484038)

[2.1.3 源程序 6](#_Toc532484039)

[2.1.4 本题小结 13](#_Toc532484040)

[2.2 大整数计算 14](#_Toc532484041)

[2.2.1 题目描述 14](#_Toc532484042)

[2.2.2 算法设计 14](#_Toc532484043)

[2.2.3 源程序 17](#_Toc532484044)

[2.2.4 本题小结 18](#_Toc532484045)

[2.3 最优二分查找树 18](#_Toc532484046)

[2.3.1 题目描述 18](#_Toc532484047)

[2.3.2 算法设计 19](#_Toc532484048)

[2.3.3 源程序 21](#_Toc532484049)

[2.3.4 本题小结 21](#_Toc532484050)

[2.4 Floyd-Warshall算法 21](#_Toc532484051)

[2.4.1 题目描述 21](#_Toc532484052)

[2.4.2 算法设计 21](#_Toc532484053)

[2.4.3源程序 22](#_Toc532484054)

[2.4.4 本题小结 22](#_Toc532484055)

[三、 心得体会 22](#_Toc532484056)

[四、 附录 22](#_Toc532484057)

[附录1.1 closest\_point\_search.h 22](#_Toc532484058)

[附录1.2 closest\_point\_search.cpp 24](#_Toc532484059)

[附录2.1 BigInteger.h 32](#_Toc532484060)

[附录3.1 optimal\_binary\_search\_tree.h 38](#_Toc532484061)

[附录3.2 optimal\_binary\_search\_tree.cpp 38](#_Toc532484062)

[附录4.1 Floyd\_Warshall.h 40](#_Toc532484063)

[附录4.2 Floyd\_Warshall.cpp 41](#_Toc532484064)

1. 实验情况总览

本次实验中共有8道题目：4道必做，4道选作。我完成了4道必做题。完成的四道必做题为：

1. 最近点对问题
2. 大整数相计算
3. 最优二分查找树
4. Floyed-Warshall算法

本地测试：

1.最近点对问题的测试如图 1，图 2所示。

测试数据

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| case 1 | | | | | | | | | |
| (1, 0) | (2, 5) | (8, 1) | (6, 9) | (5, 10) | (3, 20) | (10, 80) | (20, 60) | (30, 70) | (4, 26) |
| 理论结果 | | (1, 0) (2, 5) dis=5.0992 | | | 实际结果 | | (1, 0) (2, 5) dis=5.0992 | | |
| case 2 | | | | | | | | | |
| (2, 2) | (6, 7) | (20, 65) | (32, 96) | (27, 16) | (2, 60) | (3, 8) | (9, 56) | (68, 45) | (21, 75) |
| 理论结果 | | (6,7) (3, 8) dis=3.16228 | | | 实际结果 | | 6,7) (3, 8) dis=3.16228 | | |

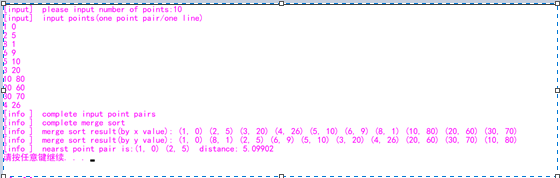


图 1

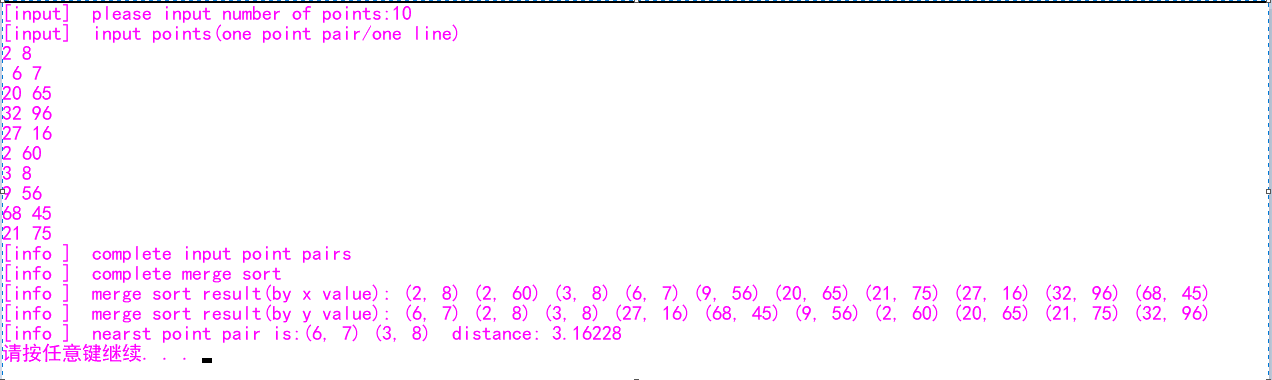


图 2

2.大整数计算的测试如图 3所示。

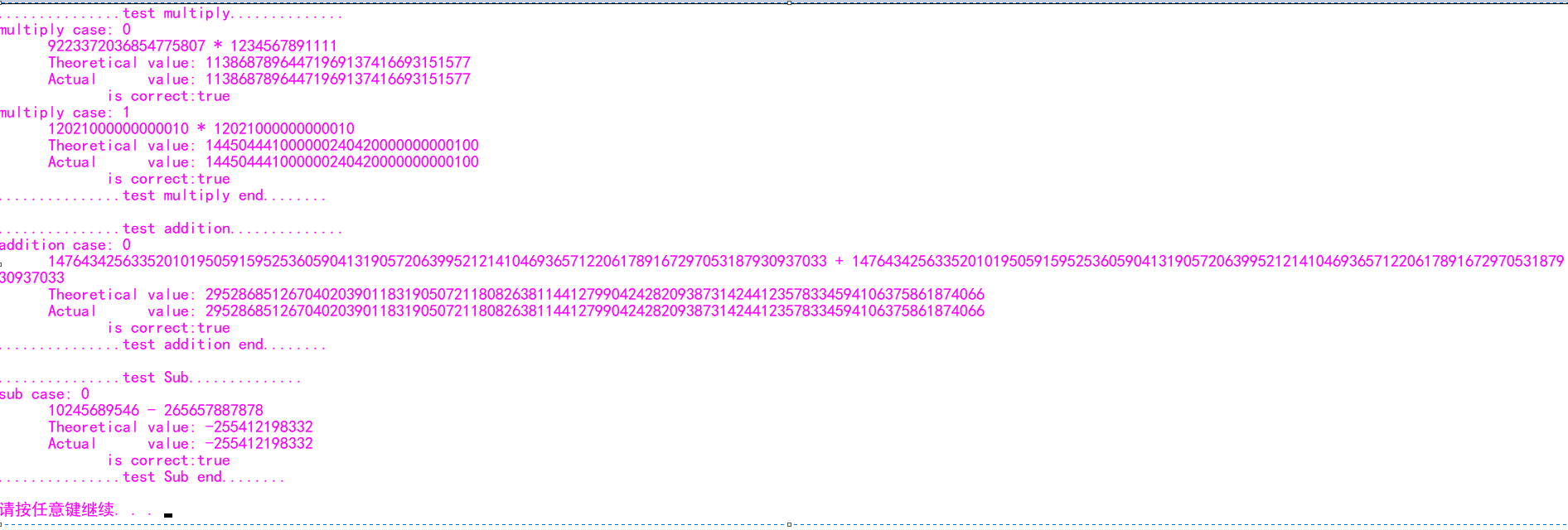


图 3

3.最优二分查找树的测试如图 4所示。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n = 5 | | | | | | | | | | |
| 0.15 | 0.10 | 0.05 | 0.10 | 0.20 | 0.05 | 0.10 | 0.05 | 0.05 | 0.05 | 0.10 |

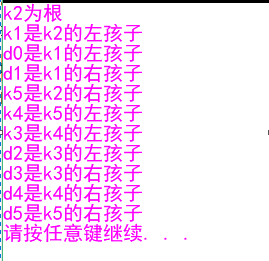


图 4

4.Floyed-Warshall算法的测试如图 5所示。

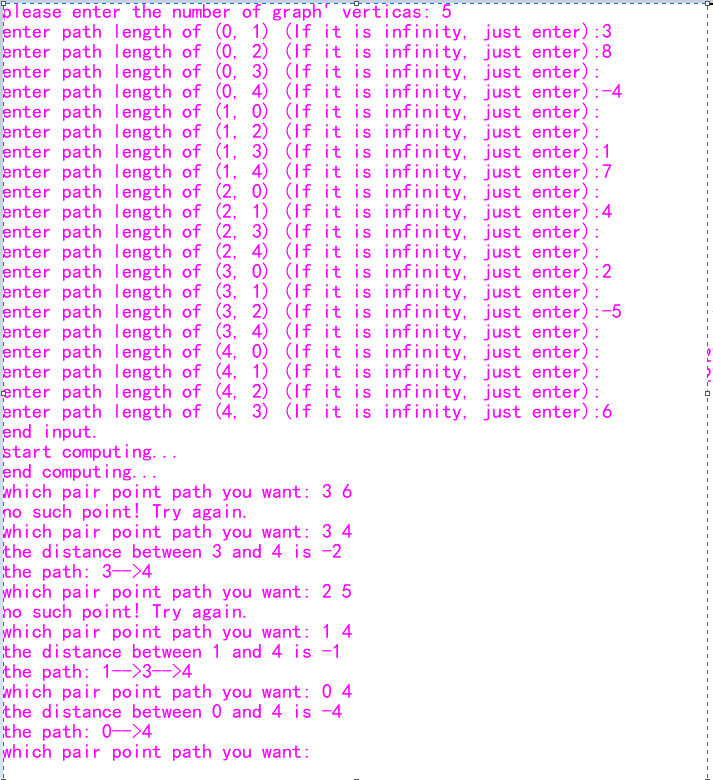


图 5

1. 解题报告

2.1 最近点对问题解题报告

2.1.1 题目描述

已知平面上分布着点集P中的n个点。点i的坐标记为。两点之间的距离取其欧式距离，记为

问题：找出一对距离最近的点。

注：允许两个点位于同一个位置，此时两点之间的距离为0

2.1.2 算法设计

Approach 1: Brute Force

要求所有点对中相距最近的点对，可以将所有的点对的距离计算出来，然后比较。该算法时间复杂度较高。

Approach 2: Divide and Conquer

a.算法分析：该题目使用分治法求解。类比于最大子数组问题（在最大子数组问题中我们将输入划分递归的划分为两部分，最大子数组为子数组中的一个，而子数组有三种情况：位于划分的右边，位于划分的左边，横跨划分点），在求解该问题时，依然使用分治方法尝试将问题递归地划分为相等规模的两部分。那么最短距离必然存在于划分完成的左边或右边或横跨划分点。

记输入序列为A[left, right]

算法执行：

1.预处理：将所有点对分别按照x，y坐标进行排序，得到两个序列：（为输入数据按照x坐标排序结果，为输入数据按照y坐标排序结果），方便后面的划分过程。

2.分解

使用类似于最大子数组问题的划分方法，递归的将输入数据划分为两个相等大小规模的序列（可以按照x坐标划分，也可以按照y坐标划分，在这里选定为按照x划分）。选定划分位置为mid，划分得到.

3.求解

本次划分完成后，距离最近的两个点, 所处位置必然为以下三种情况之一：

* 完全位于序列中，因此

。

* 完全位于序列中，因此
* 跨越了中点，因此

对于第一二种情况，可以递归求解，这两个问题依然是更小规模的最近点对问题。因此求解剩下的工作就是：递归的求解得到第一二种情况的解，求解第三种情况，选择三者最小的值然后返回。之后我们再来讨论如何求解第三种情况。

4.合并

合并过程并没有额外的操作，在返回过程中我们已经得到了需要的信息，不需要额外的操作。

5.求解第三种递归过程中的第三种情况（跨越中点）。

在此之前我们已经通过递归过程得到了左右两部分的最短距离，记为同时记，记为跨越中点的点对的最短距离。我们可以很容易得知我们只需要在mid周围x距离为的范围内寻找最近点对即可。遍历该区域所有的点，即可得到最近点对，但是这种算法并不稳定：存在这样一种可能，所有的点都集中在该区域内（），这样的话，遍历求解与暴力求解的时间复杂度一致。

优化的方法：我们在之前已经根据x与y坐标分别排序，我们依然可以使用类似于之前的划分方法，在x限定区域后再对y进行分治求解，方法与之前的方法一致。

b.时间复杂度分析

排序过程，使用归并排序或快速排序，时间复杂度为

给出递归求解的递归式：

上述递归式求解结果为：

故时间复杂度为

2.1.3 源程序

首先给出数据类型定义说明：

1.point\_array为点对存储数据类型，num表示点数量，每个点包括x坐标值与y坐标值。

1. **typedef** **struct** POINT\_ARRAY {
2. **int**\*\* point;
3. **int** num;
4. }point\_array;

2.nearst\_dis为递归过程中的返回值类型，模拟算法返回的三元组，其中first\_point，second\_point分别表示构成最短距离的第一二个点distance存储这两个点的欧式距离。

1. **typedef** **struct** DIS\_NEARST {
2. **int** first\_point[2];
3. **int** second\_point[2];
4. **double** distance;
5. }nearst\_dis;

下面给出源程序中所有函数定义说明：

1.排序方法为归并排序，输入参数：left，right限定排序范围，result\_array为排序结果，sort\_methond为排序方法（0表示按照x坐标排序，1表示按照y坐标排序），还有一个隐含的全局变量input给出待排序序列。merge\_sort为调用入口，merge为归并排序辅助函数。

1. **void** merge\_sort(**int** left, **int** right, point\_array\* result\_array, **int** sort\_methond) {
2. **if** (left < right - 1) {
3. **int** mid = (left + right) / 2;
4. merge\_sort(left, mid, result\_array, sort\_methond);
5. merge\_sort(mid, right, result\_array, sort\_methond);
6. merge(left, right, mid, result\_array, sort\_methond);
7. }
8. }
10. **void** merge(**int** left, **int** right, **int** mid, point\_array\* result\_array, **int** sort\_methond) {
11. **int** front = mid - left;
12. **int** back = right - mid;
13. **int**\*\* left\_array = (**int** \*\*)malloc(**sizeof**(**int** \*) \* (front + 1));
14. **int**\*\* right\_array = (**int** \*\*)malloc(**sizeof**(**int** \*) \* (back + 1));
15. **for** (**int** i = 0; i < front; ++i) {
16. left\_array[i] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
17. left\_array[i][0] = result\_array->point[left + i][0];
18. left\_array[i][1] = result\_array->point[left + i][1];
19. }
20. left\_array[front] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
21. left\_array[front][0] = INT\_MAX;
22. left\_array[front][1] = INT\_MAX;
23. **for** (**int** i = 0; i < back; ++i) {
24. right\_array[i] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
25. right\_array[i][0] = result\_array->point[mid + i][0];
26. right\_array[i][1] = result\_array->point[mid + i][1];
27. }
28. right\_array[back] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
29. right\_array[back][0] = INT\_MAX;
30. right\_array[back][1] = INT\_MAX;
31. **for** (**int** i = 0, j = 0; i <= front && j <= back && left < right; ++left) {
32. **if** (left\_array[i][sort\_methond] <= right\_array[j][sort\_methond]) {
33. result\_array->point[left][0] = left\_array[i][0];
34. result\_array->point[left][1] = left\_array[i][1];
35. ++i;
36. }
37. **else** {
38. result\_array->point[left][0] = right\_array[j][0];
39. result\_array->point[left][1] = right\_array[j][1];
40. ++j;
41. }
42. }
44. **for** (**int** i = 0; i <= front; ++i) {
45. free(left\_array[i]);
46. }
47. **for** (**int** i = 0; i <= back; ++i) {
48. free(right\_array[i]);
49. }
50. }

2.get\_dis函数返回给定两点之间的欧式距离

1. **double** get\_dis(**int** first\_point[2], **int** sec\_point[2]) {
2. **return** sqrt(pow(abs(first\_point[0] - sec\_point[0]), 2) + pow(abs(first\_point[1] - sec\_point[1]), 2));
3. }

3.min\_three与min\_two给出三点之间与两点之间最短的一个。

1. nearst\_dis\* min\_three(nearst\_dis\* left, nearst\_dis\* right, nearst\_dis\* mid) {
2. nearst\_dis\* min\_two\_result = min\_two(left, right);
3. **return** min\_two\_result->distance < mid->distance ? min\_two\_result : mid;
4. }
5. nearst\_dis\* min\_two(nearst\_dis\* left, nearst\_dis\* right) {
6. **return** left->distance < right->distance ? left : right;
7. }

3.min\_three\_point在给定的三个点中求的最短距离

1. nearst\_dis\* min\_three\_point(**int** point\_1[2], **int** point\_2[2], **int** point\_3[2]) {
2. nearst\_dis\* result = (nearst\_dis \*)malloc(**sizeof**(nearst\_dis));
3. **double** dis[3];
4. dis[0] = get\_dis(point\_1, point\_2);
5. dis[1] = get\_dis(point\_2, point\_3);
6. dis[2] = get\_dis(point\_3, point\_1);
7. **if** (dis[0] < dis[1]) {
8. result->distance = dis[0];
9. result->first\_point[0] = point\_1[0];
10. result->first\_point[1] = point\_1[1];
11. result->second\_point[0] = point\_2[0];
12. result->second\_point[1] = point\_2[1];
13. }
14. **else** {
15. result->distance = dis[1];
16. result->first\_point[0] = point\_2[0];
17. result->first\_point[1] = point\_2[1];
18. result->second\_point[0] = point\_3[0];
19. result->second\_point[1] = point\_3[1];
20. }
22. **if** (dis[2] < result->distance) {
23. result->distance = dis[2];
24. result->first\_point[0] = point\_3[0];
25. result->first\_point[1] = point\_3[1];
26. result->second\_point[0] = point\_1[0];
27. result->second\_point[1] = point\_1[1];
28. }
29. **return** result;
30. }

4.get\_nearst\_dis\_mid求的跨越重点的最短距离（即：第三步）。

1. nearst\_dis\* get\_nearst\_dis\_mid(point\_array\* sorted\_array\_x, point\_array\* sorted\_array\_y, nearst\_dis\* nearst\_dis\_c, **int** mid) {
2. nearst\_dis\* result = (nearst\_dis \*)malloc(**sizeof**(nearst\_dis));
3. result->distance = nearst\_dis\_c->distance;
4. result->first\_point[0] = nearst\_dis\_c->first\_point[0];
5. result->first\_point[1] = nearst\_dis\_c->first\_point[1];
6. result->second\_point[0] = nearst\_dis\_c->second\_point[0];
7. result->second\_point[1] = nearst\_dis\_c->second\_point[1];
8. **for** (**int** i = mid + 1; i < sorted\_array\_x->num; ++i) {
9. **if** (sorted\_array\_x->point[i][0] - sorted\_array\_x->point[mid][0] < result->distance
10. && abs(sorted\_array\_x->point[i][1] - sorted\_array\_x->point[mid][1]) < result->distance
11. && get\_dis(sorted\_array\_x->point[i], sorted\_array\_x->point[mid])) {
12. result->distance = get\_dis(sorted\_array\_x->point[i], sorted\_array\_x->point[mid]);
13. result->first\_point[0] = sorted\_array\_x->point[i][0];
14. result->first\_point[1] = sorted\_array\_x->point[i][1];
15. result->second\_point[0] = sorted\_array\_x->point[mid][0];
16. result->second\_point[1] = sorted\_array\_x->point[mid][1];
17. }
18. **else** {
19. **break**;
20. }
21. }
22. **for** (**int** i = mid; i >= 0; --i) {
23. **if** (sorted\_array\_x->point[mid][0] - sorted\_array\_x->point[i][0] < result->distance
24. && abs(sorted\_array\_x->point[i][1] - sorted\_array\_x->point[mid][1]) < result->distance
25. && get\_dis(sorted\_array\_x->point[i], sorted\_array\_x->point[mid])) {
26. result->distance = get\_dis(sorted\_array\_x->point[i], sorted\_array\_x->point[mid]);
27. result->first\_point[0] = sorted\_array\_x->point[i][0];
28. result->first\_point[1] = sorted\_array\_x->point[i][1];
29. result->second\_point[0] = sorted\_array\_x->point[mid][0];
30. result->second\_point[1] = sorted\_array\_x->point[mid][1];
31. }
32. **else** {
33. **break**;
34. }
35. }
36. **return** result;
37. }

5.recursive函数根据输入的按照x坐标排序的序列以及按照y坐标排序的序列求的最短距离

1. nearst\_dis\* recursive(point\_array\* array\_sorted\_x, point\_array\* array\_sorted\_y) {
2. **if** (array\_sorted\_x->num > 3) {
3. **int** mid = array\_sorted\_x->num / 2;
4. point\_array\* left\_array\_sorted\_x\_next = (point\_array \*)malloc(**sizeof**(point\_array));
5. left\_array\_sorted\_x\_next->point = (**int** \*\*)malloc(**sizeof**(**int**)\*(mid));
6. left\_array\_sorted\_x\_next->num = 0;
8. point\_array\* right\_array\_sorted\_x\_next = (point\_array \*)malloc(**sizeof**(point\_array));
9. right\_array\_sorted\_x\_next->point = (**int** \*\*)malloc(**sizeof**(**int**)\*(array\_sorted\_x->num - mid));
10. right\_array\_sorted\_x\_next->num = 0;
11. /\*
12. 构建传递给下一次递归调用的数组参数表，即左右分割数组
13. a.以x排序数组
14. \*/
15. **for** (**int** i = 0, j = 0, k = 0; i < array\_sorted\_x->num; ++i) {
16. **if** (i < mid) {
17. left\_array\_sorted\_x\_next->point[j] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
18. left\_array\_sorted\_x\_next->point[j][0] = array\_sorted\_x->point[i][0];
19. left\_array\_sorted\_x\_next->point[j][1] = array\_sorted\_x->point[i][1];
20. ++left\_array\_sorted\_x\_next->num;
21. ++j;
22. }
23. **else** {
24. right\_array\_sorted\_x\_next->point[k] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
25. right\_array\_sorted\_x\_next->point[k][0] = array\_sorted\_x->point[i][0];
26. right\_array\_sorted\_x\_next->point[k][1] = array\_sorted\_x->point[i][1];
27. ++right\_array\_sorted\_x\_next->num;
28. ++k;
29. }
30. }
31. /\*
32. b.以y排序数组
33. \*/
34. point\_array\* left\_array\_sorted\_y\_next = (point\_array \*)malloc(**sizeof**(point\_array));
35. left\_array\_sorted\_y\_next->point = (**int** \*\*)malloc(**sizeof**(**int**)\*(mid));
36. left\_array\_sorted\_y\_next->num = 0;
37. point\_array\* right\_array\_sorted\_y\_next = (point\_array \*)malloc(**sizeof**(point\_array));
38. right\_array\_sorted\_y\_next->point = (**int** \*\*)malloc(**sizeof**(**int**)\*(array\_sorted\_y->num - mid));
39. right\_array\_sorted\_y\_next->num = 0;
40. **for** (**int** i = 0, j = 0, k = 0; i < array\_sorted\_y->num; ++i) {
41. **if** (array\_sorted\_y->point[i][0] >= left\_array\_sorted\_x\_next->point[0][0]
42. &&
43. array\_sorted\_y->point[i][0] <= left\_array\_sorted\_x\_next->point[left\_array\_sorted\_x\_next->num - 1][0]) {
44. left\_array\_sorted\_y\_next->point[j] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
45. left\_array\_sorted\_y\_next->point[j][0] = array\_sorted\_y->point[i][0];
46. left\_array\_sorted\_y\_next->point[j][1] = array\_sorted\_y->point[i][1];
47. ++left\_array\_sorted\_y\_next->num;
48. ++j;
49. }
50. **else** {
51. right\_array\_sorted\_y\_next->point[k] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
52. right\_array\_sorted\_y\_next->point[k][0] = array\_sorted\_y->point[i][0];
53. right\_array\_sorted\_y\_next->point[k][1] = array\_sorted\_y->point[i][1];
54. ++right\_array\_sorted\_y\_next->num;
55. ++k;
56. }
57. }

60. nearst\_dis\* nearst\_dis\_left = recursive(left\_array\_sorted\_x\_next, left\_array\_sorted\_y\_next);
61. nearst\_dis\* nearst\_dis\_right = recursive(right\_array\_sorted\_x\_next, right\_array\_sorted\_y\_next);
62. nearst\_dis\* nearst\_dis\_mid = get\_nearst\_dis\_mid(array\_sorted\_x, array\_sorted\_y, min\_two(nearst\_dis\_left, nearst\_dis\_right), mid);
63. **return** min\_three(nearst\_dis\_left, nearst\_dis\_right, nearst\_dis\_mid);
64. }
65. **else** **if** (array\_sorted\_x->num == 2) {
66. nearst\_dis\* result = (nearst\_dis \*)malloc(**sizeof**(nearst\_dis));
67. result->first\_point[0] = array\_sorted\_x->point[0][0];
68. result->first\_point[1] = array\_sorted\_x->point[0][1];
69. result->second\_point[0] = array\_sorted\_x->point[1][0];
70. result->second\_point[1] = array\_sorted\_x->point[1][1];
71. result->distance = get\_dis(array\_sorted\_x->point[0], array\_sorted\_x->point[1]);
72. **return** result;
73. }
74. **else** **if** (array\_sorted\_x->num == 3) {
75. **return** min\_three\_point(array\_sorted\_x->point[0], array\_sorted\_x->point[1], array\_sorted\_x->point[2]);
76. }
77. **else** {
78. nearst\_dis\* result = (nearst\_dis \*)malloc(**sizeof**(nearst\_dis));
79. result->distance = INT\_MAX;
80. result->first\_point[0] = INT\_MAX;
81. result->first\_point[1] = INT\_MAX;
82. result->second\_point[0] = INT\_MAX;
83. result->second\_point[1] = INT\_MAX;
84. **return** result;
85. }
86. }

完整代码见

* 附录1.1 closest\_point\_search.h
* 附录1.2 closest\_point\_search.cpp

2.1.4 本题小结

本题的实际实现过程与理论分析的过程有较大差距，在理论分析过程中，我们使用一定的间隔来标识两点所在区间，但是在实际中，只能根据点去划分，不能根据区域去划分。

2.2 大整数计算

2.2.1 题目描述

1.用字符串给定两个非负整数，计算两数相乘的结果。

2.用字符串给定两个非负整数，计算两数相加的结果。

3.用字符串给定两个非负整数，计算两数相减的结果。

2.2.2 算法设计

1.大整数相乘

对于这个问题想到寻找到分治方法解决的话，首先得找到最优子结构，我们尝试从一个竖式算法找到子问题的形式。如图 6。

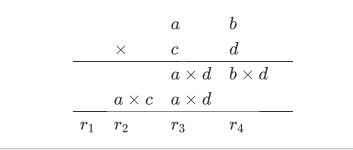


图 6

a,b,c,d假设均为一位数，那么这便是最基本的竖式计算，扩展到多位数，这个算式依然成立。这便是在大整数乘法中使用分治算法的前提。使用一个例子来进一步分析这个算式，如图 7。

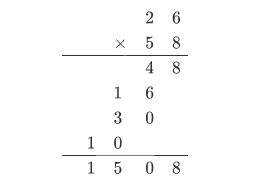


图 7

由于n\*m的乘积最多有m+n位，假设，那么n\*m会留下n位作为积，剩下的（称为部分积）可以看作是一位乘法中的进位即要与下一次乘法相加的部分。在上面这个例子中6\*8=48，将8作为积，4作为与下一次乘法（即：2\*8=16）相加。我们可以很容易在这个算式中得知如何对齐相加，在具体实现的时候，对齐的位置也是根据n与m的长度决定的。在这个例子中len（26）=2， len（58）=2，使用分治方法，我们得到四部分分别为2，6，5，8.四部分按照乘法相乘结果为：6\*8=48，2\*8=16，6\*5=30，2\*5=10.要计算部分积必须要考虑对齐，但是我们可以做一下转换，不使用对齐。如图 8所示

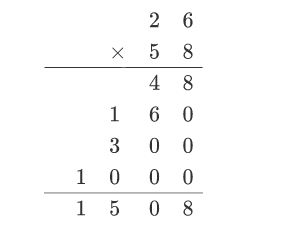


图 8

我们只需要在每次的乘积之后补0，然后做加法即可。所以问题就转化为了该补多少个0.补0的个数与n和m的长度有关，这里给出表达式。

假设a,b,c,d为分解完成的四部分，如下伪代码描述

1. length\_1 = len(num1)
2. length\_2 = len(num2)
4. a = num1[0 : length\_1 / 2]
5. b = num1[length\_1 / 2: length\_1]
6. c = num1[0 : length\_2 / 2]
7. d = num1[length\_2 / 2: length\_2]

t1, t2, t3, t4分别乘积，如下伪代码描述

1. t1 = multiply(a,c) + '0' \* (len(d)+len(b))
2. t2 = multiply(b,c) + '0' \* len(d)
3. t3 = multiply(a,d) + '0' \*len(b)
4. t4 = multiply(b,d)

这便是分治方法解决大整数相乘的基本方法，下面阐述分治方法的三个步骤在该问题中具体情况。

a.分解。分解按照上面所阐述的，将两个整数用mid分割。

b.求解。分别求解得到4个积，并补上合适的0.

c.合并。使用加法将四部分加起来，在这里需要用到大整数加法。

2.大整数相加

大整数相加的计算方法与竖式计算一致，唯一需要处理的部分为对齐。

如何对齐：

假设我们尝试计算如图 9所示的加法。



图 9

但是实际上输入得到的是如图 10所示的样子。



图 10

我们可以找出两个数中长度最大的一个，然后在另一个数字前面补上相应数目的0即可。补完0之后，如图 11所示。



图 11

在此之后就可以直接按位计算了。

3.大整数相减

大整数相减与大整数相加做法大致类似。首先判断减数与被减数的大小并使用sign记录差的符号。如果被减数大于或等于减数，则sign=1（表示差为正）；如果被减数小于减数，则sign=-1（表示差为负），并交换减数与被减数。大于等于时不交换与小于时交换时为了保证一个大数减一个小数（方便计算）。得到差之后再根据sign增加符号。

2.2.3 源程序

类BigInteger拥有三个方法分别为multiply，add，subtract为分别用于计算乘法，加法，减法。

该文件还拥有一个函数run\_bigInteger用于测试，测试分为三部分，分别为乘法，加法，减法。每个测试拥有一个循环用于从字符串数组中读取值，并计算，比较。输出格式如图 12所示。

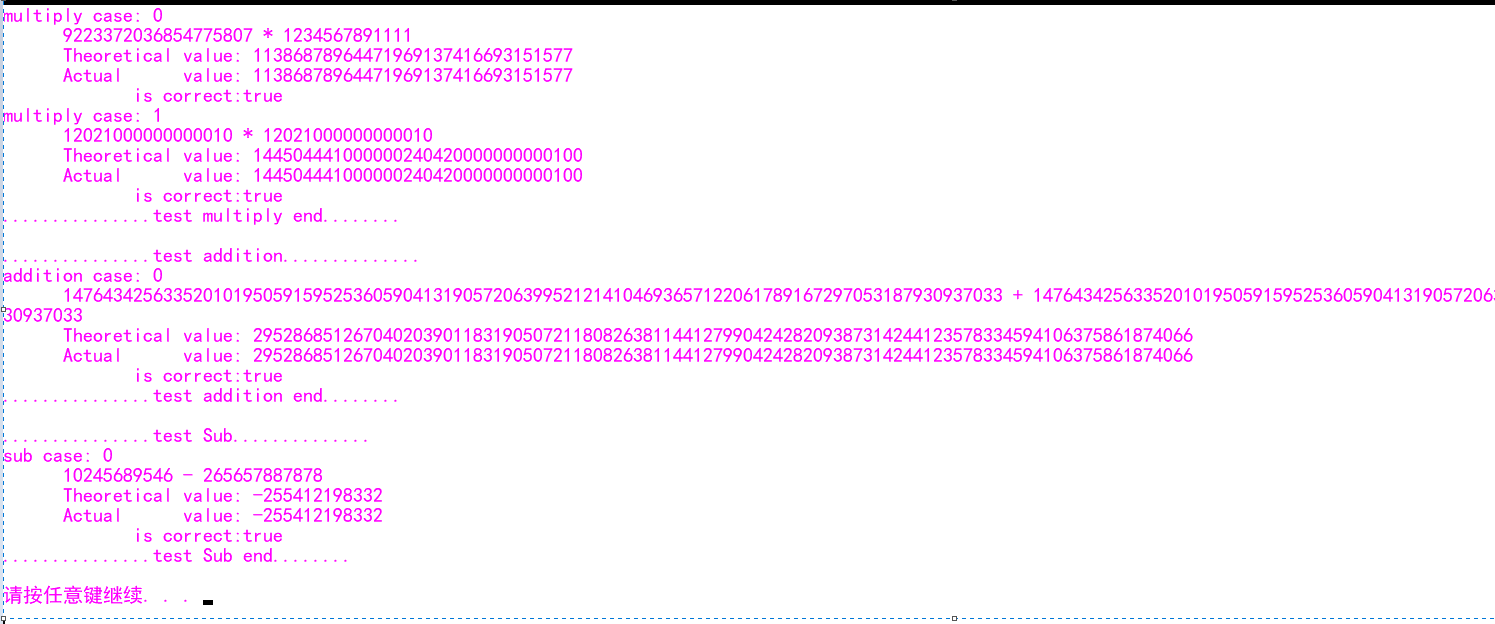


图 12

源程序见附录2.1。

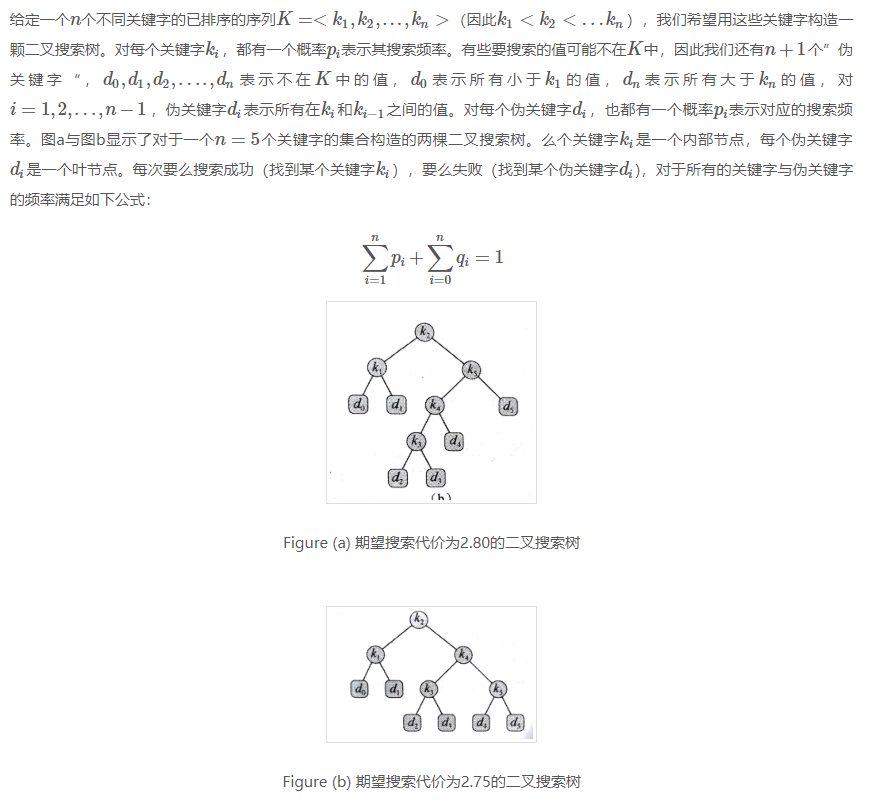
2.2.4 本题小结

本题大数相加以及相减都不难，只有大数相乘比较难。解决这个问题的关键在于从竖式乘法中找到规律，并且能够使用。

2.3 最优二分查找树

2.3.1 题目描述

最优二分查找树也成为最优二叉搜索树。该问题的形式化定义如下：



构造二叉搜索树使得所有搜索操作访问的结点总数最少。

2.3.2 算法设计

1.最优二叉搜索树的递归式

在这里给出书上推导的最优解值的递归定义。选取子问题域为找到一个包含关键字的最优二叉查找树，其中而且。（当时没有真实的关键字；只有虚拟键）。定义为搜索一棵包含关键字的最优二叉查找树的期望代价，最终，我们要计算。

当时出现简单情况。此时只有虚拟键。期望的搜索代价为。

当时，需要从中选取一个根，然后用关键字来构造一棵最优二叉查找树作为其左子树，并用关键字来构造一棵最优二叉搜索树作为其右子树。当一棵树成为一个结点的子树时，子树的每个结点的深度增加1.整个子树的期望搜索代价增加为子树中所有概率之和。对于一棵有关键字的子树，定义概率总和为

因此，如果是一棵包含关键字的最优子树的根，则有

注意

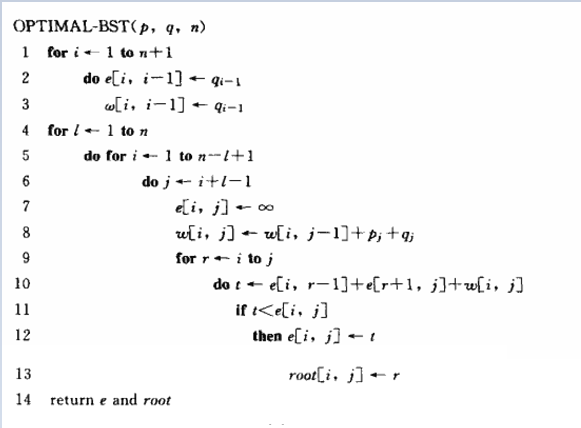
将重写为

最终的递归式为

2.最优二叉搜索树的解法

使用额外的数组root来记录根结点，w数组用来存储代价。

给出该算法的伪代码。



2.3.3 源程序

源程序请查看附录3。

2.3.4 本题小结

本题关键在于能够根据分析出的递归表达式设计实现，以及辅助存储空间的设计，构造最优解。

2.4 Floyd-Warshall算法

2.4.1 题目描述

Floyd-Warshall算法是解决任意两点间的最短路劲的一种算法，可以正确处理有向图或负权（但不可存在负权回路）的最短路径问题，同时也被用于计算有向图的传递闭包。编程实现该算法。

2.4.2 算法设计

Floyed-Warshall算法的原理是动态规划。该算法的运行时间为.采用与最优二分查找树相似的分析方法，我们假设中间结点k，即通过k，能够得到从i到j的最短路径。同样的令为从结点i到结点k，且满足所有中间结点皆属于集合{1, 2, …, k}的一条最短路径的权值。当k=0时，从顶点i到顶点j的路径中，没有编号大于0的中间顶点；亦即，根本不存在中间顶点，这样的路径至多包含一条边，因此。根据以上分析，得出以下的递归式。

因为对于任意路径，所有中间顶点都在集合中，所以矩阵给出了最终的解答：对所有的，有。

构建最短路径：

在计算过程中定义前驱矩阵用于构建最短路径。

2.4.3源程序

源程序在附录4

2.4.4 本题小结

本题关键在于能够根据分析出的递归表达式设计实现，以及辅助存储空间的设计，构造最优解。

1. 心得体会

本次实验的题目都不是特别难，但是在实现过程中我发现：理论分析的结果于实际实现还是有一定区别的，理论分析只是给出“可以这样做”实际设计算法需要设计具体的实现形式。

1. 附录

附录1.1 closest\_point\_search.h

1. #pragma once
2. #define \_CRT\_SECURE\_NO\_WARNINGS
4. #ifndef CLOST\_POINT\_SEARCH
5. #define CLOST\_POINT\_SEARCH
7. #include<limits.h>
8. #include<stdio.h>
9. #include<stdlib.h>
10. #include<math.h>
11. **typedef** **struct** POINT\_ARRAY {
12. **int**\*\* point;
13. **int** num;
14. }point\_array;
16. **typedef** **struct** DIS\_NEARST {
17. **int** first\_point[2];
18. **int** second\_point[2];
19. **double** distance;
20. }nearst\_dis;
22. **void** get\_input();
24. **void** merge\_sort(**int** left, **int** right, point\_array\* result\_array, **int** sort\_methond);
26. **void** merge(**int** left, **int** right, **int** mid, point\_array\* result\_array, **int** sort\_methond);
28. **void** destructor(point\_array \* array);
30. **double** get\_dis(**int** first\_point[2], **int** sec\_point[2]);
32. **void** main\_logic();
34. nearst\_dis\* recursive(point\_array\* array\_sorted\_x, point\_array\* array\_sorted\_y);
36. nearst\_dis\* get\_nearst\_dis\_mid(point\_array\* sorted\_array\_x, point\_array\* sorted\_array\_y, nearst\_dis\* nearst\_dis\_c, **int** mid);
38. nearst\_dis\* min\_three(nearst\_dis\* left, nearst\_dis\* right, nearst\_dis\* mid);
40. nearst\_dis\* min\_two(nearst\_dis\* left, nearst\_dis\* right);
42. nearst\_dis\* min\_three\_point(**int** point\_1[2], **int** point\_2[2], **int** point\_3[2]);

45. #endif // CLOST\_POINT\_SEARCH

附录1.2 closest\_point\_search.cpp

1. #pragma once
2. #include"closest\_point\_search.h"
4. point\_array\* input\_point;
5. point\_array\* sort\_x;
6. point\_array\* sort\_y;

9. **void** get\_input() {
10. input\_point = (point\_array \*)malloc(**sizeof**(input\_point));
11. sort\_x = (point\_array \*)malloc(**sizeof**(input\_point));
12. sort\_y = (point\_array \*)malloc(**sizeof**(input\_point));
13. input\_point->num = 0;
14. sort\_x->num = 0;
15. sort\_y->num = 0;
16. **int** num = 0;
17. printf("[input]  please input number of points:");
18. scanf("%d", &num);
19. **if** (num <= 0) {
20. printf("[error]  wrong number");
21. **return**;
22. }
23. **else** {
24. input\_point->num = num;
25. input\_point->point = (**int** \*\*)malloc(**sizeof**(**int**\*) \* input\_point->num);
26. sort\_x->num = num;
27. sort\_x->point = (**int** \*\*)malloc(**sizeof**(**int**\*) \* sort\_x->num);
28. sort\_y->num = num;
29. sort\_y->point = (**int** \*\*)malloc(**sizeof**(**int**\*) \* sort\_y->num);
30. }
31. printf("[input]  input points(one point pair/one line)\n");
32. **for** (**int** i = 0; i < num; ++i) {
33. input\_point->point[i] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
34. sort\_x->point[i] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
35. sort\_y->point[i] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
36. scanf("%d %d", &input\_point->point[i][0], &input\_point->point[i][1]);
37. sort\_x->point[i][0] = input\_point->point[i][0];
38. sort\_x->point[i][1] = input\_point->point[i][1];
40. sort\_y->point[i][0] = input\_point->point[i][0];
41. sort\_y->point[i][1] = input\_point->point[i][1];
42. }
43. printf("[info ]  complete input point pairs\n");
44. main\_logic();
45. system("pause");
46. **return**;
47. }
49. **void** merge\_sort(**int** left, **int** right, point\_array\* result\_array, **int** sort\_methond) {
50. **if** (left < right - 1) {
51. **int** mid = (left + right) / 2;
52. merge\_sort(left, mid, result\_array, sort\_methond);
53. merge\_sort(mid, right, result\_array, sort\_methond);
54. merge(left, right, mid, result\_array, sort\_methond);
55. }
56. }
58. **void** merge(**int** left, **int** right, **int** mid, point\_array\* result\_array, **int** sort\_methond) {
59. **int** front = mid - left;
60. **int** back = right - mid;
61. **int**\*\* left\_array = (**int** \*\*)malloc(**sizeof**(**int** \*) \* (front + 1));
62. **int**\*\* right\_array = (**int** \*\*)malloc(**sizeof**(**int** \*) \* (back + 1));
63. **for** (**int** i = 0; i < front; ++i) {
64. left\_array[i] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
65. left\_array[i][0] = result\_array->point[left + i][0];
66. left\_array[i][1] = result\_array->point[left + i][1];
67. }
68. left\_array[front] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
69. left\_array[front][0] = INT\_MAX;
70. left\_array[front][1] = INT\_MAX;
71. **for** (**int** i = 0; i < back; ++i) {
72. right\_array[i] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
73. right\_array[i][0] = result\_array->point[mid + i][0];
74. right\_array[i][1] = result\_array->point[mid + i][1];
75. }
76. right\_array[back] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
77. right\_array[back][0] = INT\_MAX;
78. right\_array[back][1] = INT\_MAX;
79. **for** (**int** i = 0, j = 0; i <= front && j <= back && left < right; ++left) {
80. **if** (left\_array[i][sort\_methond] <= right\_array[j][sort\_methond]) {
81. result\_array->point[left][0] = left\_array[i][0];
82. result\_array->point[left][1] = left\_array[i][1];
83. ++i;
84. }
85. **else** {
86. result\_array->point[left][0] = right\_array[j][0];
87. result\_array->point[left][1] = right\_array[j][1];
88. ++j;
89. }
90. }
92. **for** (**int** i = 0; i <= front; ++i) {
93. free(left\_array[i]);
94. }
95. **for** (**int** i = 0; i <= back; ++i) {
96. free(right\_array[i]);
97. }
98. }
100. **void** destructor(point\_array \* array) {
101. **for** (**int** i = 0; i < array->num; ++i) {
102. free(array->point[i]);
103. }
104. free(array->point);
105. free(array);
106. }
108. **double** get\_dis(**int** first\_point[2], **int** sec\_point[2]) {
109. **return** sqrt(pow(abs(first\_point[0] - sec\_point[0]), 2) + pow(abs(first\_point[1] - sec\_point[1]), 2));
110. }
112. **void** main\_logic() {
113. /\*
114. 归并排序，x，y
115. \*/
116. merge\_sort(0, sort\_x->num, sort\_x, 0);
117. merge\_sort(0, sort\_y->num, sort\_y, 1);
118. printf("[info ]  complete merge sort\n");
119. printf("[info ]  merge sort result(by x value): ");
120. **for** (**int** i = 0; i < sort\_x->num; ++i) {
121. printf("(%d, %d) ", sort\_x->point[i][0], sort\_x->point[i][1]);
122. }
123. printf("\n");
124. printf("[info ]  merge sort result(by y value): ");
125. **for** (**int** i = 0; i < sort\_y->num; ++i) {
126. printf("(%d, %d) ", sort\_y->point[i][0], sort\_y->point[i][1]);
127. }
128. printf("\n");


132. nearst\_dis\* result = recursive(sort\_x, sort\_y);
133. printf("[info ]  nearst point pair is:(%d, %d) (%d, %d)  distance: %.5lf\n",
134. result->first\_point[0],
135. result->first\_point[1],
136. result->second\_point[0],
137. result->second\_point[1],
138. result->distance);
139. **return**;
140. }
142. nearst\_dis\* recursive(point\_array\* array\_sorted\_x, point\_array\* array\_sorted\_y) {
143. **if** (array\_sorted\_x->num > 3) {
144. **int** mid = array\_sorted\_x->num / 2;
145. point\_array\* left\_array\_sorted\_x\_next = (point\_array \*)malloc(**sizeof**(point\_array));
146. left\_array\_sorted\_x\_next->point = (**int** \*\*)malloc(**sizeof**(**int**)\*(mid));
147. left\_array\_sorted\_x\_next->num = 0;
149. point\_array\* right\_array\_sorted\_x\_next = (point\_array \*)malloc(**sizeof**(point\_array));
150. right\_array\_sorted\_x\_next->point = (**int** \*\*)malloc(**sizeof**(**int**)\*(array\_sorted\_x->num - mid));
151. right\_array\_sorted\_x\_next->num = 0;
152. /\*
153. 构建传递给下一次递归调用的数组参数表，即左右分割数组
154. a.以x排序数组
155. \*/
156. **for** (**int** i = 0, j = 0, k = 0; i < array\_sorted\_x->num; ++i) {
157. **if** (i < mid) {
158. left\_array\_sorted\_x\_next->point[j] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
159. left\_array\_sorted\_x\_next->point[j][0] = array\_sorted\_x->point[i][0];
160. left\_array\_sorted\_x\_next->point[j][1] = array\_sorted\_x->point[i][1];
161. ++left\_array\_sorted\_x\_next->num;
162. ++j;
163. }
164. **else** {
165. right\_array\_sorted\_x\_next->point[k] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
166. right\_array\_sorted\_x\_next->point[k][0] = array\_sorted\_x->point[i][0];
167. right\_array\_sorted\_x\_next->point[k][1] = array\_sorted\_x->point[i][1];
168. ++right\_array\_sorted\_x\_next->num;
169. ++k;
170. }
171. }
172. /\*
173. b.以y排序数组
174. \*/
175. point\_array\* left\_array\_sorted\_y\_next = (point\_array \*)malloc(**sizeof**(point\_array));
176. left\_array\_sorted\_y\_next->point = (**int** \*\*)malloc(**sizeof**(**int**)\*(mid));
177. left\_array\_sorted\_y\_next->num = 0;
178. point\_array\* right\_array\_sorted\_y\_next = (point\_array \*)malloc(**sizeof**(point\_array));
179. right\_array\_sorted\_y\_next->point = (**int** \*\*)malloc(**sizeof**(**int**)\*(array\_sorted\_y->num - mid));
180. right\_array\_sorted\_y\_next->num = 0;
181. **for** (**int** i = 0, j = 0, k = 0; i < array\_sorted\_y->num; ++i) {
182. **if** (array\_sorted\_y->point[i][0] >= left\_array\_sorted\_x\_next->point[0][0]
183. &&
184. array\_sorted\_y->point[i][0] <= left\_array\_sorted\_x\_next->point[left\_array\_sorted\_x\_next->num - 1][0]) {
185. left\_array\_sorted\_y\_next->point[j] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
186. left\_array\_sorted\_y\_next->point[j][0] = array\_sorted\_y->point[i][0];
187. left\_array\_sorted\_y\_next->point[j][1] = array\_sorted\_y->point[i][1];
188. ++left\_array\_sorted\_y\_next->num;
189. ++j;
190. }
191. **else** {
192. right\_array\_sorted\_y\_next->point[k] = (**int** \*)malloc(**sizeof**(**int**) \* 2);
193. right\_array\_sorted\_y\_next->point[k][0] = array\_sorted\_y->point[i][0];
194. right\_array\_sorted\_y\_next->point[k][1] = array\_sorted\_y->point[i][1];
195. ++right\_array\_sorted\_y\_next->num;
196. ++k;
197. }
198. }

201. nearst\_dis\* nearst\_dis\_left = recursive(left\_array\_sorted\_x\_next, left\_array\_sorted\_y\_next);
202. nearst\_dis\* nearst\_dis\_right = recursive(right\_array\_sorted\_x\_next, right\_array\_sorted\_y\_next);
203. nearst\_dis\* nearst\_dis\_mid = get\_nearst\_dis\_mid(array\_sorted\_x, array\_sorted\_y, min\_two(nearst\_dis\_left, nearst\_dis\_right), mid);
204. **return** min\_three(nearst\_dis\_left, nearst\_dis\_right, nearst\_dis\_mid);
205. }
206. **else** **if** (array\_sorted\_x->num == 2) {
207. nearst\_dis\* result = (nearst\_dis \*)malloc(**sizeof**(nearst\_dis));
208. result->first\_point[0] = array\_sorted\_x->point[0][0];
209. result->first\_point[1] = array\_sorted\_x->point[0][1];
210. result->second\_point[0] = array\_sorted\_x->point[1][0];
211. result->second\_point[1] = array\_sorted\_x->point[1][1];
212. result->distance = get\_dis(array\_sorted\_x->point[0], array\_sorted\_x->point[1]);
213. **return** result;
214. }
215. **else** **if** (array\_sorted\_x->num == 3) {
216. **return** min\_three\_point(array\_sorted\_x->point[0], array\_sorted\_x->point[1], array\_sorted\_x->point[2]);
217. }
218. **else** {
219. nearst\_dis\* result = (nearst\_dis \*)malloc(**sizeof**(nearst\_dis));
220. result->distance = INT\_MAX;
221. result->first\_point[0] = INT\_MAX;
222. result->first\_point[1] = INT\_MAX;
223. result->second\_point[0] = INT\_MAX;
224. result->second\_point[1] = INT\_MAX;
225. **return** result;
226. }
227. }
229. nearst\_dis\* get\_nearst\_dis\_mid(point\_array\* sorted\_array\_x, point\_array\* sorted\_array\_y, nearst\_dis\* nearst\_dis\_c, **int** mid) {
230. nearst\_dis\* result = (nearst\_dis \*)malloc(**sizeof**(nearst\_dis));
231. result->distance = nearst\_dis\_c->distance;
232. result->first\_point[0] = nearst\_dis\_c->first\_point[0];
233. result->first\_point[1] = nearst\_dis\_c->first\_point[1];
234. result->second\_point[0] = nearst\_dis\_c->second\_point[0];
235. result->second\_point[1] = nearst\_dis\_c->second\_point[1];
236. **for** (**int** i = mid + 1; i < sorted\_array\_x->num; ++i) {
237. **if** (sorted\_array\_x->point[i][0] - sorted\_array\_x->point[mid][0] < result->distance
238. && abs(sorted\_array\_x->point[i][1] - sorted\_array\_x->point[mid][1]) < result->distance
239. && get\_dis(sorted\_array\_x->point[i], sorted\_array\_x->point[mid])) {
240. result->distance = get\_dis(sorted\_array\_x->point[i], sorted\_array\_x->point[mid]);
241. result->first\_point[0] = sorted\_array\_x->point[i][0];
242. result->first\_point[1] = sorted\_array\_x->point[i][1];
243. result->second\_point[0] = sorted\_array\_x->point[mid][0];
244. result->second\_point[1] = sorted\_array\_x->point[mid][1];
245. }
246. **else** {
247. **break**;
248. }
249. }
250. **for** (**int** i = mid; i >= 0; --i) {
251. **if** (sorted\_array\_x->point[mid][0] - sorted\_array\_x->point[i][0] < result->distance
252. && abs(sorted\_array\_x->point[i][1] - sorted\_array\_x->point[mid][1]) < result->distance
253. && get\_dis(sorted\_array\_x->point[i], sorted\_array\_x->point[mid])) {
254. result->distance = get\_dis(sorted\_array\_x->point[i], sorted\_array\_x->point[mid]);
255. result->first\_point[0] = sorted\_array\_x->point[i][0];
256. result->first\_point[1] = sorted\_array\_x->point[i][1];
257. result->second\_point[0] = sorted\_array\_x->point[mid][0];
258. result->second\_point[1] = sorted\_array\_x->point[mid][1];
259. }
260. **else** {
261. **break**;
262. }
263. }
264. **return** result;
265. }
266. nearst\_dis\* min\_three(nearst\_dis\* left, nearst\_dis\* right, nearst\_dis\* mid) {
267. nearst\_dis\* min\_two\_result = min\_two(left, right);
268. **return** min\_two\_result->distance < mid->distance ? min\_two\_result : mid;
269. }
270. nearst\_dis\* min\_two(nearst\_dis\* left, nearst\_dis\* right) {
271. **return** left->distance < right->distance ? left : right;
272. }
274. nearst\_dis\* min\_three\_point(**int** point\_1[2], **int** point\_2[2], **int** point\_3[2]) {
275. nearst\_dis\* result = (nearst\_dis \*)malloc(**sizeof**(nearst\_dis));
276. **double** dis[3];
277. dis[0] = get\_dis(point\_1, point\_2);
278. dis[1] = get\_dis(point\_2, point\_3);
279. dis[2] = get\_dis(point\_3, point\_1);
280. **if** (dis[0] < dis[1]) {
281. result->distance = dis[0];
282. result->first\_point[0] = point\_1[0];
283. result->first\_point[1] = point\_1[1];
284. result->second\_point[0] = point\_2[0];
285. result->second\_point[1] = point\_2[1];
286. }
287. **else** {
288. result->distance = dis[1];
289. result->first\_point[0] = point\_2[0];
290. result->first\_point[1] = point\_2[1];
291. result->second\_point[0] = point\_3[0];
292. result->second\_point[1] = point\_3[1];
293. }
295. **if** (dis[2] < result->distance) {
296. result->distance = dis[2];
297. result->first\_point[0] = point\_3[0];
298. result->first\_point[1] = point\_3[1];
299. result->second\_point[0] = point\_1[0];
300. result->second\_point[1] = point\_1[1];
301. }
302. **return** result;
303. }

附录2.1 BigInteger.h

1. #pragma once
2. #define \_CRT\_SECURE\_NO\_WARNINGS
4. #ifndef BIG\_INTEGER
5. #define BIG\_INTEGER
7. #define max\_value 300
8. #include<iostream>
9. #include<algorithm>
10. #include<string>
11. #include<cmath>
12. **class** BigInteger {

15. **public**:
16. std::string num1;
17. std::string num2;
18. std::string multiply(std::string n1, std::string n2) {
19. **int** n1Length = n1.length();
20. **int** n2Length = n2.length();
21. **if** (n1Length == 0 || n2Length == 0 || n1 == "0" || n2 == "0") {
22. **return** std::string("0");
23. }
25. **if** (n1Length < 5 && n2Length < 5) {
26. **char** r[max\_value];
27. **return** std::string(\_itoa(atoi(n1.c\_str()) \* atoi(n2.c\_str()), r, 10));
28. }
30. std::string a(n1.substr(0, n1Length / 2));
31. std::string b(n1.substr(n1Length / 2, n1Length - n1Length / 2));
32. std::string c(n2.substr(0, n2Length / 2));
33. std::string d(n2.substr(n2Length / 2, n2Length - n2Length / 2));
35. std::string t1 = multiply(a, c) + std::string(d.length() + b.length(), '0');
36. std::string t2 = multiply(b, c) + std::string(d.length(), '0');
37. std::string t3 = multiply(a, d) + std::string(b.length(), '0');
38. std::string t4 = multiply(b, d);
39. **return** add(add(add(t1, t2), t3), t4);
41. }
42. std::string add(std::string s1, std::string s2) {
43. **if** (s1.length() == 0 && s2.length() == 0) {
44. **return** std::string("0");
45. }
46. **else** **if** (s1.length() == 0) {
47. **return** s2;
48. }
49. **else** **if** (s2.length() == 0) {
50. **return** s1;
51. }
52. **if** (s1 == "0") {
53. **return** s2;
54. }
55. **if** (s2 == "0") {
56. **return** s1;
57. }
58. //较短的字符串放在第二位。
59. **if** (s1.size() < s2.size())
60. {
61. std::string temp = s1;
62. s1 = s2;
63. s2 = temp;
64. }
65. std::string sum;
67. s2 = std::string(s1.size() - s2.size(), '0') + s2;
68. **int** a = 0;
69. **for** (**int** i = s1.size() - 1; i >= 0; i--)
70. {
71. **int** b = (s1[i] - '0') + (s2[i] - '0') + a;
72. **if** (b > 9) {
73. sum.push\_back((b - 10) + '0');
74. a = 1;
75. }
76. **else** {
77. sum.push\_back(b + '0');
78. a = 0;
79. }
81. }
82. sum.push\_back(a + '0');
84. std::string::iterator it = sum.end() - 1;
85. **while** (it != sum.begin()) {
86. **if** ((\*it) == '0') {
87. it = sum.erase(it);
88. --it;
89. }
90. **else** {
91. **break**;
92. }
93. }
94. std::reverse(sum.begin(), sum.end());
95. **return** sum;
96. }
97. std::string subtract(std::string n1, std::string n2) {
99. **int** sign = 1;//n1-n2大于等于0    sign = 1否则sign = -1；
100. **int** n1Length = n1.length();
101. **int** n2Length = n2.length();
102. **if** (n1Length < n2Length) {
103. std::string tmp = n1;
104. n1 = n2;
105. n2 = tmp;
106. sign = -1;
107. }
108. //n1与n2长度相等时，从高位比较，找出较大的一个
109. **else** **if**( n1Length == n2Length) {
110. **int** i = 0;
111. **for** (; i < n1Length; ++i){
112. **int** subResult = n1[i] - n2[i];
113. **if** (subResult< 0) {
114. std::string tmp = n1;
115. n1 = n2;
116. n2 = tmp;
117. sign = -1;
118. **break**;
119. }
120. **else** **if** (subResult == 0) {
121. **continue**;
122. }
123. **else** {
124. **break**;
125. }
126. }
127. }
128. n2 = std::string(n1.length() - n2.length(), '0') + n2;
130. std::string result;
132. **int** borrow = 0;
133. **for** (**int** i = n1.length()-1; i >= 0; --i) {
134. **int** tmp = n1[i] - n2[i] - borrow;
135. **if** (tmp < 0) {
136. tmp = 10 + n1[i] - n2[i] - borrow;
137. borrow = 1;
138. }
139. **else** {
140. borrow = 0;
141. }
142. result.push\_back(tmp + '0');
143. }
144. std::string::iterator it = result.end() - 1;
145. **while** (it != result.begin()) {
146. **if** ((\*it) == '0') {
147. it = result.erase(it);
148. --it;
149. }
150. **else** {
151. **break**;
152. }
153. }
154. **if** (sign == -1) {
155. result.push\_back('-');
156. }
157. std::reverse(result.begin(), result.end());
158. **return** result;
159. }
161. **public**:
162. BigInteger(std::string n1, std::string n2) {
163. **this**->num1 = n1;
164. **this**->num2 = n2;
165. }
166. ~BigInteger() {}
168. };
170. **void** run\_mult() {
172. std::cout << "...............test multiply.............." << std::endl;
173. std::string testSetMultiply[2][3] =
174. {
175. { "9223372036854775807" ,"1234567891111", "11386878964471969137416693151577"},
176. { "12021000000000010", "12021000000000010", "144504441000000240420000000000100" }
177. };
178. **for** (**int** i = 0; i < 2; ++i) {
179. std::cout << "multiply case: " << i << std::endl;
180. BigInteger test(testSetMultiply[i][0], testSetMultiply[i][1]);
181. std::cout
182. << "      " << testSetMultiply[i][0] << " \* " << testSetMultiply[i][1] << std::endl
183. << "      Theoretical value: " << testSetMultiply[i][2] << std::endl
184. << "      Actual      value: " << test.multiply(test.num1, test.num2) << std::endl
185. << "             is correct:" << (test.multiply(test.num1, test.num2) == testSetMultiply[i][2] ? "true" : "false")
186. << std::endl;
188. }
189. std::cout << "...............test multiply end........" << std::endl << std::endl;

192. std::cout << "...............test addition.............." << std::endl;
193. std::string testSetAddition[1][3] =
194. {
195. { "1476434256335201019505915952536059041319057206399521214104693657122061789167297053187930937033" ,"1476434256335201019505915952536059041319057206399521214104693657122061789167297053187930937033", "2952868512670402039011831905072118082638114412799042428209387314244123578334594106375861874066" },
197. };
198. **for** (**int** i = 0; i < 1; ++i) {
199. std::cout << "addition case: " << i << std::endl;
200. BigInteger test(testSetAddition[i][0], testSetAddition[i][1]);
201. std::cout
202. << "      " << testSetAddition[i][0] << " + " << testSetAddition[i][1] << std::endl
203. << "      Theoretical value: " << testSetAddition[i][2] << std::endl
204. << "      Actual      value: " << test.add(test.num1, test.num2) << std::endl
205. << "             is correct:" << (test.add(test.num1, test.num2) == testSetAddition[i][2] ? "true" : "false")
206. << std::endl;
208. }
209. std::cout << "...............test addition end........" << std::endl << std::endl;

212. std::cout << "...............test Sub.............." << std::endl;
213. std::string testSetSub[1][3] =
214. {
215. {"10245689546", "265657887878", "-255412198332"}
216. };
217. **for** (**int** i = 0; i < 1; ++i) {
218. std::cout << "sub case: " << i << std::endl;
219. BigInteger test(testSetSub[i][0], testSetSub[i][1]);
220. std::cout
221. << "      " << testSetSub[i][0] << " - " << testSetSub[i][1] << std::endl
222. << "      Theoretical value: " << testSetSub[i][2] << std::endl
223. << "      Actual      value: " << test.subtract(test.num1, test.num2) << std::endl
224. << "             is correct:" << (test.subtract(test.num1, test.num2) == testSetSub[i][2] ? "true" : "false")
225. << std::endl;
227. }
228. std::cout << "...............test Sub end........" << std::endl << std::endl;
230. system("pause");
231. }
232. #endif // !BIG\_INTEGER

附录3.1 optimal\_binary\_search\_tree.h

1. #pragma once
2. #define \_CRT\_SECURE\_NO\_WARNINGS
4. #ifndef OPTIMAL\_BINARY\_SEARCH\_TREE
5. #define OPTIMAL\_BINARY\_SEARCH\_TREE
7. #include <iostream>
8. #include <vector>
10. **using** **namespace** std;
12. **int** run\_OST();
14. **void** optimal\_bst(
15. vector<**double**>&p,
16. vector<**double**>&q,
17. vector<vector<**double**>>&e,
18. vector<vector<**double**>>&w,
19. vector<vector<**int**>>&root);
20. #endif // !OPTIMAL\_BINARY\_SEARCH\_TREE

附录3.2 optimal\_binary\_search\_tree.cpp

1. #include"optimal\_binary\_search\_tree.h"
2. **void** optimal\_bst(
3. vector<**double**>&p,
4. vector<**double**>&q,
5. vector<vector<**double**>>&e,
6. vector<vector<**double**>>&w,
7. vector<vector<**int**>>&root)
8. {
9. **int** n = p.size() - 1;
10. **for** (**int** i = 1; i <= n + 1; i++)
11. {
12. e[i][i - 1] = q[i - 1];
13. w[i][i - 1] = q[i - 1];
14. }
15. **for** (**int** l = 1; l <= n; l++)
16. {
17. **for** (**int** start = 1; start <= n - l + 1; start++)
18. {
19. **int** end = start + l - 1;
20. w[start][end] = w[start][end - 1] + p[end] + q[end];
21. e[start][end] = 1000;
22. **for** (**int** r = start; r <= end; r++)
23. {
24. **double** t = e[
25. start][r - 1] + e[r + 1][end] + w[start][end];
26. **if** (t<e[start][end])
27. {
28. e[start][end] = t;
29. root[start][end] = r;
30. }
31. }
32. }
33. }
34. }
36. **void** print(vector<vector<**int**>>root, **int** a, **int** b, **int** p)
37. {
38. **if** (p == -1)
39. printf("k%d为根\n", root[a][b]);
41. **int** r = root[a][b];
42. //左子树
43. **if** (r == a)
44. {
45. printf("d%d是k%d的左孩子\n", a - 1, a);
46. }
47. **else**
48. {
49. printf("k%d是k%d的左孩子\n", root[a][r - 1], r);
50. print(root, a, r - 1, 0);
51. }
52. //右子树
53. **if** (r == b)
54. {
55. printf("d%d是k%d的右孩子\n", b, b);
56. }
57. **else**
58. {
59. printf("k%d是k%d的右孩子\n", root[r + 1][b], r);
60. print(root, r + 1, b, 0);
61. }
62. }
63. **int** run\_OST()
64. {
66. **double** p1[] = { 0,   0.15,0.10,0.05,0.10,0.20 };
67. **double** q1[] = { 0.05,0.10,0.05,0.05,0.05,0.10 };
68. vector<**double**>p;
69. vector<**double**>q;
70. **for** (**int** i = 0; i<**sizeof**(p1) / **sizeof**(**double**); i++)
71. {
72. p.push\_back(p1[i]);//尾部加入元素
73. q.push\_back(q1[i]);
74. }
75. **int** n = p.size() - 1;
76. vector<vector<**double**>>e(n + 2);//二重容器保存搜索费用
77. vector<vector<**int**>>root(n + 2);
78. vector<vector<**double**>>w(n + 2);
79. **for** (**int** i = 0; i<n + 2; i++)
80. {
81. e[i].resize(n + 2);
82. root[i].resize(n + 2);
83. w[i].resize(n + 2);
84. }
85. optimal\_bst(p, q, e, w, root);
86. print(root, 1, n, -1);
87. system("pause");
88. **return** 0;
89. }

附录4.1 Floyd\_Warshall.h

1. #ifndef FLOYD\_WARSHALL
2. #define FLOYD\_WARSHALL
3. #define \_CRT\_SECURE\_NO\_WARNINGS
5. #include<string>
6. #include<iostream>
8. **namespace** floyd\_warshall {
9. **typedef** **struct** GRAPH {
10. **int** graph\_vertices;
11. **int** \*\*graph\_martix;
12. }graph;
13. **class** solution {
14. **private**:
15. graph ins\_graph;
16. graph shortest\_path;
17. **void** compute();
18. **int** add(**int** a, **int** b);
19. **int** min(**int** a, **int** b);
20. **void** get\_shortest\_path(**int** first, **int** second);
21. **void** free\_graph(graph \* g);
22. **public**:
23. solution();
24. **virtual** **void** set\_graph();
25. **virtual** graph\* get\_graph();
26. **virtual** **void** get\_path(**int** first, **int** second);
27. **virtual** **void** show\_graph(graph \* **const** g);
28. };
29. **void** run();
30. }
31. #endif // !FLOYD\_WARSHALL

附录4.2 Floyd\_Warshall.cpp

1. #include "floyd\_warshall.h"
3. **void** floyd\_warshall::solution::compute()
4. {
5. std::cout << "start computing..." << std::endl;
6. **for** (**int** k = 0; k < **this**->ins\_graph.graph\_vertices; ++k) {
7. **for** (**int** i = 0; i < **this**->ins\_graph.graph\_vertices; ++i) {
8. **for** (**int** j = 0; j < **this**->ins\_graph.graph\_vertices; ++j) {
9. **if** (add(**this**->ins\_graph.graph\_martix[i][k], **this**->ins\_graph.graph\_martix[k][j]) < **this**->ins\_graph.graph\_martix[i][j]) {
10. **this**->ins\_graph.graph\_martix[i][j] = add(**this**->ins\_graph.graph\_martix[i][k], **this**->ins\_graph.graph\_martix[k][j]);
11. **this**->shortest\_path.graph\_martix[i][j] = k;
12. }
13. }
14. }
15. }
16. std::cout << "end computing..." << std::endl;
17. **return**;
18. }
20. **int** floyd\_warshall::solution::add(**int** a, **int** b)
21. {
22. **return** (a == INT\_MAX || b == INT\_MAX) ? INT\_MAX : a + b;
23. }
25. **int** floyd\_warshall::solution::min(**int** a, **int** b)
26. {
27. **return** a < b ? a : b;
28. }
30. floyd\_warshall::solution::solution()
31. {
32. **this**->set\_graph();
33. **this**->compute();
34. **return**;
35. }
37. **void** floyd\_warshall::solution::set\_graph()
38. {
39. std::cout << "please enter the number of graph' verticas: ";
40. std::cin >> **this**->ins\_graph.graph\_vertices;
41. getchar();
42. **this**->shortest\_path.graph\_vertices = **this**->ins\_graph.graph\_vertices;
43. **this**->ins\_graph.graph\_martix = **new** **int**\*[**this**->ins\_graph.graph\_vertices];
44. **this**->shortest\_path.graph\_martix = **new** **int**\*[**this**->shortest\_path.graph\_vertices];
45. **for** (**int** i = 0; i < **this**->ins\_graph.graph\_vertices; ++i) {
46. **this**->ins\_graph.graph\_martix[i] = **new** **int**[**this**->ins\_graph.graph\_vertices];
47. **this**->shortest\_path.graph\_martix[i] = **new** **int**[**this**->shortest\_path.graph\_vertices];
48. **for** (**int** j = 0; j < **this**->ins\_graph.graph\_vertices; ++j) {
49. **this**->ins\_graph.graph\_martix[i][j] = INT\_MAX;
50. **this**->shortest\_path.graph\_martix[i][j] = 0;
51. }
52. **this**->ins\_graph.graph\_martix[i][i] = 0;
53. }
54. **for** (**int** i = 0; i < **this**->ins\_graph.graph\_vertices; ++i) {
55. **for** (**int** j = 0; j < **this**->ins\_graph.graph\_vertices; ++j) {
56. **if** (i != j) {
57. std::cout << "enter path length of (" << i << ", " << j << ") (If it is infinity, just enter):";
58. **char** tmp[100];
59. std::cin.getline(tmp, 100);
60. **if** (strcmp(tmp, "")) {
61. **this**->ins\_graph.graph\_martix[i][j] = atoi(tmp);
62. //getchar();
63. }
64. }
65. }
66. }
67. std::cout << "end input." << std::endl;
68. **return**;
69. }
71. **void** floyd\_warshall::solution::free\_graph(graph \* g)
72. {
73. **if** (g == NULL) {
74. **return**;
75. }
76. **for** (**int** i = 0; i < g->graph\_vertices; ++i) {
77. **delete** g->graph\_martix[i];
78. }
79. **delete** g->graph\_martix;
80. **delete** g;
81. g = NULL;
82. **return**;
83. }
85. floyd\_warshall::graph\* floyd\_warshall::solution::get\_graph()
86. {
87. graph \*result = **new** graph;
88. result->graph\_vertices = **this**->ins\_graph.graph\_vertices;
89. result->graph\_martix = **new** **int**\*[result->graph\_vertices];
90. **for** (**int** i = 0; i < **this**->ins\_graph.graph\_vertices; ++i) {
91. result->graph\_martix[i] = **new** **int**[result->graph\_vertices];
92. **for** (**int** j = 0; j < result->graph\_vertices; ++j) {
93. result->graph\_martix[i][j] = **this**->ins\_graph.graph\_martix[i][j];
94. }
95. }
96. **return** result;
97. }
99. **void** floyd\_warshall::solution::get\_shortest\_path(**int** first, **int** second)
100. {
101. **if** (first == second) {
102. **return**;
103. }
104. **else** **if** (**this**->shortest\_path.graph\_martix[first][second] == 0) {
105. std::cout << "-->" << second;
106. **return**;
107. }
108. **else** {
109. get\_shortest\_path(first, **this**->shortest\_path.graph\_martix[first][second]);
110. get\_shortest\_path( **this**->shortest\_path.graph\_martix[first][second],second);
111. }
112. }
114. **void** floyd\_warshall::solution::get\_path(**int** first, **int** second)
115. {
116. **if** (first >= **this**->ins\_graph.graph\_vertices || second >= **this**->ins\_graph.graph\_vertices) {
117. std::cout << "no such point! Try again." << std::endl;
118. **return**;
119. }
120. **if** (**this**->ins\_graph.graph\_martix[first][second] == INT\_MAX) {
121. std::cout << first << " and " << second << " No path" << std::endl;
122. **return**;
123. }
124. **else**   **if** (first == second) {
125. std::cout << first << " and " << second << " is " << "the same Node" << std::endl;
126. **return**;
127. }
128. **else**
129. {
130. std::cout << "the distance between " << first << " and " << second << " is " << **this**->ins\_graph.graph\_martix[first][second] << std::endl;
131. std::cout << "the path: ";
132. std::cout << first;
133. get\_shortest\_path(first, second);
134. std::cout << std::endl;
135. }
136. }
138. **void** floyd\_warshall::solution::show\_graph(graph \* **const** g)
139. {
140. **for** (**int** i = 0; i < g->graph\_vertices; ++i) {
141. **for** (**int** j = 0; j < g->graph\_vertices; ++j) {
142. **if** (g->graph\_martix[i][j] == INT\_MAX) {
143. std::cout << " N ";
144. }
145. **else** **if** (g->graph\_martix[i][j] >= 0) {
147. std::cout << " " << g->graph\_martix[i][j] << " ";
148. }
149. **else** {
150. std::cout << g->graph\_martix[i][j] << " ";
151. }
152. }
153. std::cout << std::endl;
154. }
155. }
157. **void** floyd\_warshall::run()
158. {
159. solution s = solution();
160. **while** (**true**) {
161. std::cout << "which pair point path you want: ";
162. **int** i = 0;
163. **int** j = 0;
164. std::cin >> i;
165. std::cin >> j;
166. s.get\_path(i, j);
168. }
169. **return**;
170. }