《程序设计艺术与方法》课程实验报告

实验名称	实验一 STL 的熟悉与使用						
姓　　名		系院专业		班　　级		学　　号	
实验日期		指导教师	徐本柱			成　　绩	

一、实验目的和要求

1．掌握 C++中 STL 的容器类的使用;
2．掌握 C++中 STL 的算法类的使用.

二、实验预习内容

1．预习 ICPC 讲义，大致了解 STL 的相关内容。
2．了解 STL 中一些类 vector list 类的使用方法
3．了解泛型算法的使用

三、实验项目摘要

1．练习 vector 和 list 的使用
2．练习泛型算法的使用

四、实验结果与分析（源程序及相关说明）

实验内容：

1. 练习 vector 和 list 的使用

①实验要求：

定义一个空的 vector，元素类型为 int，生成 10 个随机数插入到 vector 中，用迭代器遍历 vector 并输出其中的元素值。在 vector 头部插入一个随机数，用迭代器遍历 vector 并输出其中的元素值。用泛型算法 find 查找某个随机数，如果找到便输出，否则将此数插入 vector 尾部。用泛型算法 sort 将 vector 排序，用迭代器遍历 vector 并输出其中的元素值。删除 vector 尾部的元素，用迭代器遍历 vector 并输出其中的元素值。将 vector 清空。

思路：按要求编写程序，编写时需要注意 STL 的使用规则。

②源程序：

```
# include <algorithm>
# include <vector>
# include <list>
# include <iostream>

using namespace std;

void vectorTest()
{
	vector<int> vec1;
	cout << "Vector*********************************************" << endl;
	cout << "生成 10 个随机数存入 vector" << endl;

	for (int i = 0; i < 10; ++i)
	{
		vec1.push_back(rand());
	}

	vector<int>::iterator it;
	for (it = vec1.begin(); it != vec1.end(); ++it)
	{
		cout << *it << ' ';
	}
	cout << "\n";

	cout << "\n 头部插入随机数" << endl;

	vec1.insert(vec1.begin(), rand());

	for (it = vec1.begin(); it != vec1.end(); ++it)
	{
		cout << *it << ' ';
	}
	cout << "\n";

	cout << "\nfind 查找某个随机数" << endl;

	int x = rand();
	it = find(vec1.begin(), vec1.end(), x);

	if (*it == x)
	{
		cout << *it << endl;
```

```
    }
    else
    {
        vec1.push_back(x);
    }

    for (it = vec1.begin(); it != vec1.end(); ++it)
    {
        cout << *it << ' ';
    }
    cout << "\n";

    cout << "\n 用 sort 排序" << endl;

    sort(vec1.begin(), vec1.end());

    for (it = vec1.begin(); it != vec1.end(); ++it)
    {
        cout << *it << ' ';
    }
    cout << "\n";

    cout << "\n 删除 vector 尾部元素" << endl;

    vec1.pop_back();

    for (it = vec1.begin(); it != vec1.end(); ++it)
    {
        cout << *it << ' ';
    }
    cout << "\n";

    cout << "\n 将 vector 清空" << endl;

    vec1.erase(vec1.begin(), vec1.end());

    for (it = vec1.begin(); it != vec1.end(); ++it)
    {
        cout << *it << ' ';
    }
    cout << "\n";

    cout << "Vector*********************************************" << endl;
}
```

```
void listTest()
{
	list<int> list1;

	cout << "\nList**********************************************" << endl;
	cout << "生成 10 个随机数存入 list" << endl;

	for (int i = 0; i < 10; ++i){
		list1.push_back(rand());
	}

	list<int>::iterator it;
	for (it = list1.begin(); it != list1.end(); ++it){
		cout << *it << ' ';
	}
	cout << "\n";

	cout << "\n 头部插入随机数" << endl;

	list1.push_front(rand());

	for (it = list1.begin(); it != list1.end(); ++it){
		cout << *it << ' ';
	}
	cout << "\n";

	cout << "\nfind 查找某个随机数" << endl;

	int x = rand();
	it = find(list1.begin(), list1.end(), x);

	if (*it == x)
	{
		cout << *it << endl;
	}
	else
	{
		list1.push_back(x);
	}

	for (it = list1.begin(); it != list1.end(); ++it)
	{
		cout << *it << ' ';
```

```
    }
    cout << "\n";

    cout << "\n 用 sort 排序" << endl;

    list1.sort();

    for (it = list1.begin(); it != list1.end(); ++it)
    {
        cout << *it << ' ';
    }
    cout << "\n";

    cout << "\n 删除尾部元素" << endl;

    // 删除 list 尾部元素
    list1.pop_back();

    for (it = list1.begin(); it != list1.end(); ++it){
        cout << *it << ' ';
    }
    cout << "\n";

    cout << "\n 将 list 清空" << endl;

    //清空
    list1.erase(list1.begin(), list1.end());

    for (it = list1.begin(); it != list1.end(); ++it){
        cout << *it << ' ';
    }
    cout << "\n";

    cout << "\nList*********************************************" << endl;
}

int main(int argc, char *argv[]){
    //cout << "Vector*********************************************" << endl;
    vectorTest();
    //cout << "Vector*********************************************" << endl;
    //cout << "List*********************************************" << endl;
    listTest();
    //cout << "List*********************************************" << endl;l;
```

```
    return 0;
}
```

③实验结果：

```
Vector*********************************************
生成10个随机数存入vector
41 18467 6334 26500 19169 15724 11478 29358 26962 24464

头部插入随机数
5705 41 18467 6334 26500 19169 15724 11478 29358 26962 24464

find查找某个随机数
5705 41 18467 6334 26500 19169 15724 11478 29358 26962 24464 28145

用sort排序
41 5705 6334 11478 15724 18467 19169 24464 26500 26962 28145 29358

删除vector尾部元素
41 5705 6334 11478 15724 18467 19169 24464 26500 26962 28145

将vector清空

Vector*********************************************
```

实验结果

```
List***********************************************
生成10个随机数存入list
23281 16827 9961 491 2995 11942 4827 5436 32391 14604

头部插入随机数
3902 23281 16827 9961 491 2995 11942 4827 5436 32391 14604

find查找某个随机数
3902 23281 16827 9961 491 2995 11942 4827 5436 32391 14604 153

用sort排序
153 491 2995 3902 4827 5436 9961 11942 14604 16827 23281 32391

删除尾部元素
153 491 2995 3902 4827 5436 9961 11942 14604 16827 23281

将list清空


List***********************************************
```

实验结果

2.练习泛型算法的使用。

①实验要求：

定义一个 vector，元素类型为 int，插入 10 个随机数，使用 sort 按升序排序，输出每个元素的值，再按降叙排序，输出每个元素的值。练习用 find 查找元素。用 min 和 max 找出容器中的最小元素个最大元素，并输出。

思路：和上一题一样，按实验要求操作，注意 max_element 和 min_element 的使用。

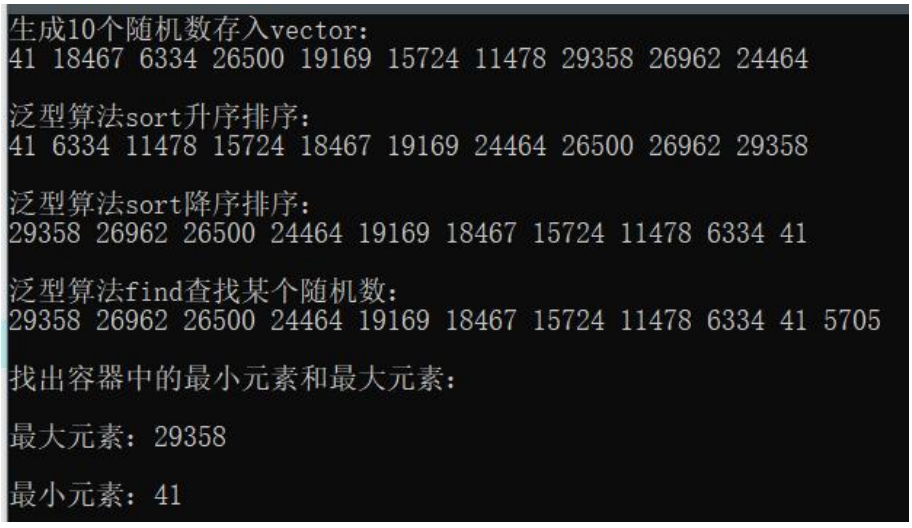
②源程序：

```
# include <vector>
# include <algorithm>
```

```
# include <iostream>

using namespace std;
vector<int> vector2;

void print()
{
	vector<int>::iterator it;
	for (it = vector2.begin(); it != vector2.end(); it++)
	{
		cout << (*it) << ' ';
	}
	cout << "\n";
}

void vectorTest2()
{
	int max, min;

	vector<int>::iterator it;
	cout << "生成 10 个随机数存入 vector：" << endl;

	for (int i = 0; i < 10; i++)
	{
		vector2.push_back(rand());
	}

	print();

	cout << "\n 泛型算法 sort 升序排序：" << endl;

	sort(vector2.begin(), vector2.end());

	print();

	cout << "\n 泛型算法 sort 降序排序：" << endl;

	sort(vector2.begin(), vector2.end(), greater<int>());

	print();

	cout << "\n 泛型算法 find 查找某个随机数：" << endl;

	int x = rand();
```

```
    it = find(vector2.begin(), vector2.end(), x);

    if (*it == x)
    {
        cout << *it << endl;
    }
    else
    {
        vector2.push_back(x);
    }

    print();

    cout << "\n 找出容器中的最小元素和最大元素：" << endl;

    max = *max_element(vector2.begin(), vector2.end());

    cout << "\n 最大元素：" << max << endl;

    min = *min_element(vector2.begin(), vector2.end());

    cout << "\n 最小元素：" << min << endl;
}

int main(){
    vectorTest2();
    return 0;
}
```

③实验结果：



实验结果

实验分析：

本次实验的主要内容是熟悉 STL，熟悉其中的容器类和算法类，并练习使用它们。

总体上来说，STL 的很多功能是非常便于使用的。从容器类这方面来说，STL 相对于自己编写的数据结构，采用了一种叫做迭代器的特殊结构，类似于指针，但适用范围更广；STL 还提供了一些常用容器，经过严格的测试，考虑的得更周全，稳定性更佳。用户使用时不必过多关心具体的实现细节，能更专注于算法本身。

从算法类这方面来说，STL 把一些非常通用的算法，如查找、排序、拷贝等写成通用函数，提供了一个易于使用的接口。用户就不需要再花大力气去写这些算法并检验正确性，只需直接调用。在实验中我练习使用了一些泛型算法，调用简单方便，完成的代码也非常简洁易懂。

《程序设计艺术与方法》课程实验报告

实验名称	实验二 搜索算法的实现						
姓　　名		系院专业		班　　级		学　　号	
实验日期			指导教师	徐本柱		成　　绩	

一、实验目的和要求

1. 掌握宽度优先搜索算法。
2. 掌握深度优先搜索算法。

二、实验预习内容

1．预习 ICPC 讲义中的搜索的内容
2. 了解什么是深度优先搜索和广度优先搜索。

三、实验项目摘要

1. 将书上的走迷宫代码上机运行并检验结果，并注意体会搜索的思想。
2.八皇后问题：在一个国际象棋棋盘上放八个皇后，使得任何两个皇后之间不相互攻击，求出所有的布棋方法。上机运行并检验结果。
3. 骑士游历问题：在国际棋盘上使一个骑士遍历所有的格子一遍且仅一遍，对于任意给定的顶点，输出一条符合上述要求的路径。
4.倒水问题：给定 2 个没有刻度容器，对于任意给定的容积，求出如何只用两个瓶装出 L 升的水，如果可以，输出步骤，如果不可以，请输出 No Solution。

四、实验结果与分析（源程序及相关说明）

实验内容：

1.将书上的走迷宫代码上机运行并检验结果，并注意体会搜索的思想。

①实验要求：

（1）按照实验要求运行代码，注意体会搜索的思想。

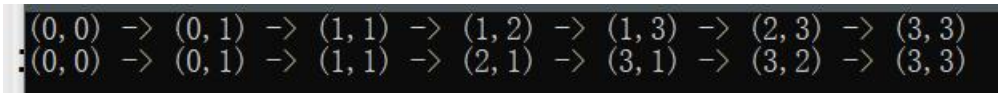
②源程序：

```
#include<algorithm>
#include<vector>
#include<list>
#include<iostream>
#include<vector>
```

```
using namespace std;

int maze[4][6] = { {1, 1, 0, 0, 0, 0},
                   {0, 1, 1, 1, 0, 0},
                   {1, 1, 0, 1, 0, 0},
                   {0, 1, 1, 1, 0, 0} };
vector<pair<int, int> > path;

int dir[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};//下 右 上 左

void printvector(vector<pair<int, int> > path)
{
    vector<pair<int , int > >::iterator it;
    for(it = path.begin();it != path.end(); it++)
    {
        cout << "("<< it->first << "," << it->second << ")" << " -> ";
    }
    cout << "(3,3)" << endl;
}

void search(vector<pair<int, int> > tpath, int x, int y)
{
    if(x < 0 || y < 0 || x > 5 || y > 3)//越界返回
        return;

    if(x == 3 && y == 3)
    {
        path = tpath; //如果找到了出口,则记录下路径
        printvector(path);//每个路径都打印
        return;
    }

    for(int ix = 0; ix < 4; ix++)//四个方向搜索
    {
        if(maze[x+dir[ix][0]][ y+dir[ix][1]] == 1)
        {
            tpath.push_back(make_pair(x, y));

            maze[x][y] = -1;//标志已走过

            search(tpath, x+dir[ix][0], y+dir[ix][1]);
            tpath.pop_back();//删除最后一个元素
        }
    }
```

```
}
int main()
{
    vector<pair<int, int> > tpath;
    search(tpath, 0, 0);//从开始点找起
}
    //代码结束
```

③实验结果：



④实验分析：

书上的走迷宫代码，缺少输出，对走过的格子没有标记，并且有一处括号有误，改正之后才能正常运行。

这个代码的思想类似于树的深度优先遍历，使用了递归搜索的思想。在每一个格子朝四个方向搜索，确定邻接格子是否能走通。如果能走通，则把这个格子加入序列，进入到这个格子继续搜索；如果走不通，则退回上一个格子，并把这个格子从序列中删除。由于已知出口的位置，当搜索到出口坐标时，程序返回。

路径序列是用一个存储 pair 类型的 vector 保存的。运用了上一讲中讲到的 STL 知识。

2.八皇后问题：在一个国际象棋棋盘上放八个皇后，使得任何两个皇后之间不相互攻击，求出所有的布棋方法。上机运行并检验结果。

①实验要求：

（1）求出所有的布棋方法。上机运行并检验结果。

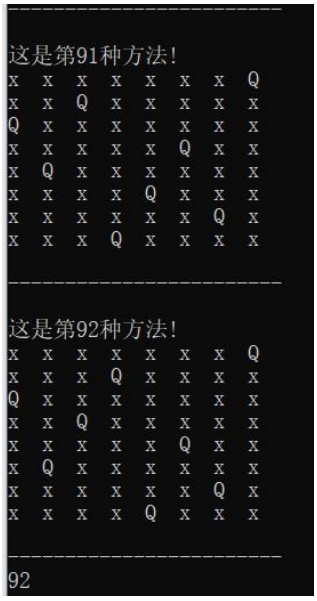
②源程序：

```
#include<algorithm>
#include<vector>
#include<list>
#include<iostream>
#include<cstdio>
#include<cmath>

using namespace std;

int x[9] = {-1, -1, -1, -1, -1, -1, -1, -1, -1}, sum = 0;//x[i]的下标表示 queen 所在行数，i 表示 queen 所在列数

int count;

void printMethod()//图形化打印方法
{
    cout << "\n 这是第" << sum << "种方法!\n" ;
    for(int i = 1; i < 9; i++)
    {
```

```
            for(int j = 1; j < 9; j++)
            {
                if(j == x[i])
                {
                    cout << "Q" << "    ";
                }
                else
                {
                    cout << "x" << "    ";
                }
                //cout << "\n";
            }
            cout << "\n";
        }
        cout << "\n" << "-------------------------" << "\n";
}

void dfs(int num)
{
        if(num >= 8)//检查完了
        {
            sum++;
            printMethod();
            return;
        }
        else
        {
            for(int j = 1;j <= 8; j++)
            {
                x[num + 1] = j;//每一列都放一放看，找出成立的位置

                bool flag = true;

                for(int g = num;g > 0; g--)//检查前面已经布好的 i-1 行
                {
                    if( x[g] == x[num + 1]
                    || (abs(x[num + 1] - x[g]) == abs(num + 1 - g)) )   //在同一列/在对角线上
                    flag = false;
                }
                if( flag )//判断是否符合条件
                    dfs(num + 1);
            }
        }
}
```

```
int main()
{
	dfs(0);
	cout << sum;
	//printMethod();
}
```

③实验结果：



这个结果是正确的。

④实验分析：

八皇后问题可以采用类似于对树进行 DFS 的思想。

任何一个皇后都不能位于其他皇后的同行、同列和左上、右上这四个方向。为了简化问题，我们在这里逐行安放皇后。皇后可以处于该行的任何一列，所以对于每一列，我们都检查放在这里会不会和之前安放过的皇后冲突。如果不冲突，则考察下一行；如果冲突，则返回。整个八皇后问题可以视为大量上述小问题的递归组合。所以我们采用 DFS 算法进行求解。当最后一行也放上了皇后，说明所有的子问题都解决了，问题结束。

在这种算法中，对于某一行，皇后放在任何一列（决策）都会使问题产生新分支，会生成一棵逻辑上的“决策树”。需要根据上一步的选择，把下一步所有可能的结果都列出来进行检验。

打印输出：当每个子问题递归到第 8 行时，x[]中就会存放这个子问题求得的排列方式，只要按照这个标号来进行输出。

3. 骑士游历问题：在国际棋盘上使一个骑士遍历所有的格子一遍且仅一遍，对于任意给定顶点，输出一条符合上述要求的路径。

①实验要求：

（1）输出一条符合上述要求的路径。

②源程序：

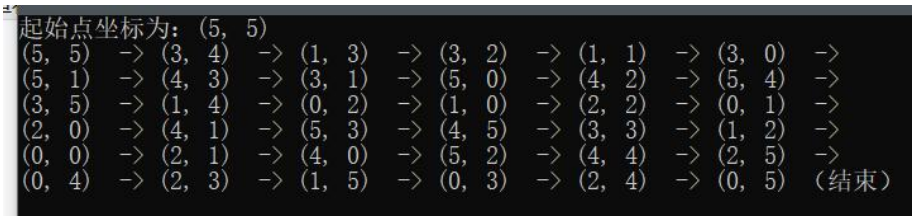
```
#include<algorithm>
#include<vector>
```

```
#include<list>
#include<iostream>
#include<vector>
#include<iostream>
#include<cstring>
#include<stack>

// 定义最大棋盘
# define MAXROW 6
# define MAXCOL 6

using namespace std;

stack<pair<int, int> > route;//用栈存储
bool board[MAXROW][MAXCOL];//棋盘标记
pair<int, int>dir[9];

void initdir()
{
	dir[1] = make_pair(-2, -1);
	dir[2] = make_pair(-2, 1);
	dir[3] = make_pair(2, -1);
	dir[4] = make_pair(2, 1);
	dir[5] = make_pair(-1, -2);
	dir[6] = make_pair(-1, 2);
	dir[7] = make_pair(1, -2);
	dir[8] = make_pair(1, 2);
}

bool inBound(int x, int y)
{
	if((0 <= x && x < MAXROW) && (0 <= y && y < MAXCOL))
		return true;
	return false;
}

bool check()//检查是否每一点都走过
{
	for(int i = 0; i < MAXROW; i++)
	{
		for(int j = 0; j < MAXCOL; j++)
		{
			if(!board[i][j])
				return false;
```

```
            }
        }
        return true;
}


void printRoute()
{
        stack<pair<int, int> > stackR;//倒置
        int flag = 0;

        while(!route.empty())//整理栈内元素顺序
        {
                pair<int, int> p = route.top();
                stackR.push(p);
                route.pop();
        }

        while(!stackR.empty())//输出
        {
                pair<int, int> p = stackR.top();
                cout << "(" << p.first << ", " << p.second << ") ";
                stackR.pop();
                if(!stackR.empty())
                        cout << " -> ";
                else
                        cout << "（结束）";

                flag++;
                if(flag % MAXCOL == 0)
                     cout << "\n";
        }
}

void findRoute(int x, int y)
{
        board[x][y] = true;
        route.push(make_pair(x, y));

        if(check())//都走完了
        {
              printRoute();
              exit(0); //只要一条，直接结束
        }
```

```
    for(int i = 1; i < 9; ++i)
    {

        pair<int, int> p1;
        p1 = make_pair(x + dir[i].first, y + dir[i].second) ;

        if( !board[p1.first][p1.second] && inBound(p1.first, p1.second) )
        {
            findRoute(p1.first, p1.second);

            board[p1.first][p1.second] = false;//如果走不通返回，会把放进去的全部弹出
                                               //恢复访问标志，回到上一次
           route.pop();
        }
    }
}

int main()
{
    int x, y;
    x = 5;
    y = 5;
    cout << "起始点坐标为：(" << x << ", " << y << ")" << endl;

    initdir();
    findRoute(x, y);
    return 0;
}
```

③实验结果：



这里用 6 * 6 棋盘进行演示。更改棋盘尺寸宏定义就可以对 8 * 8 棋盘进行求解。

④实验分析：

这个问题更类似于走迷宫。

相对于迷宫的 4 个方向，骑士在一般情况下有 8 个方向可走，但在棋盘的边角位置，他的走法会受到限制。位置和走法数目的对应关系如下表：

2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

表 1 骑士所在位置和走法的对应关系

和走迷宫类似，在每个位置，我们都向可以走的各个方向进行试验，记录路线（把路线坐标入栈），并对走过的格子进行标记。到了无路可走的时候，检查这时是否已经走过全部格子，如果已经全部走过，说明路线正确，退出程序；如果没有走遍，说明路线错误，恢复访问标志并把弹出栈顶元素，返回之前的格子，选择别的方向进行试验。

如此递归运行，程序退出时，栈中就保存了一条正确的路线，将栈中元素逆置，就是一条从给定点出发、遍历整个棋盘的正序路线了。

4. **倒水问题：给定 2 个没有刻度容器，对于任意给定的容积，求出如何只用两个瓶装出 L 升的水，如果可以，输出步骤，如果不可以，请输出 No Solution。**

①实验要求：

（1）求出如何只用两个瓶装出 L 升的水，如果可以，输出步骤，如果不可以，请输出 No Solution。

②源程序：

```
//倒水问题
#include<iostream>
#include<queue>
using namespace std;

struct node//状态节点，储存 A、B 水量和操作
{
    int a;
    int b;
    int step;
};

int a,b,c;

int vis[200][200];
```

```
node path[200][200];

void judge_step(int a,int b);

int bfs()
{
    queue<node>q;//使用队列实现的 BFS
    node now, next;

    now.a = 0;
    now.b = 0;
    now.step = 0;

    q.push(now);//入队

    while(q.size())
    {
        now = q.front();
        q.pop();

        if(now.a == c || now.b == c)//说明找到了方案
        {
            cout << now.step << "\n";
            judge_step(now.a,now.b);
            return 1;
        }

        node pre;                          //记录路径
        pre.a = now.a,pre.b = now.b;

        if(now.a < a && !vis[a][now.b])//可以倒满 A
        {
            next.a = a;
            next.b = now.b;
            next.step = now.step + 1;

            q.push(next);
            vis[next.a][next.b] = 1;
            pre.step = 1;
            path[next.a][next.b] = pre;
        }

        if(now.b < b && !vis[now.a][b])//可以倒满 B
        {
```

```
            next.a = now.a;
            next.b = b;
            next.step = now.step + 1;

            q.push(next);
            vis[next.a][next.b] = 1;
            pre.step = 2;
            path[next.a][next.b] = pre;
        }

        if(now.a > 0 && !vis[0][now.b])
        {
            next.a = 0;
            next.b = now.b;
            next.step = now.step + 1;

            q.push(next);
            vis[next.a][next.b] = 1;
            pre.step = 3;
            path[next.a][next.b] = pre;
        }

        if(now.b > 0 && !vis[now.a][0])
        {
            next.a = now.a;
            next.b = 0;
            next.step = now.step + 1;

            q.push(next);
            vis[next.a][next.b] = 1;
            pre.step = 4;
            path[next.a][next.b] = pre;
        }

        //A 往 B 倒
        if(now.a && now.b < b)
        {
            pre.step = 5;

            if(now.a >= b - now.b && !vis[now.a - (b - now.b)][b])//能倒满
            {
                next.a = now.a - (b - now.b);
                next.b = b;
                next.step = now.step + 1;
```

```
            q.push(next);
            vis[next.a][next.b] = 1;
            path[next.a][next.b] = pre;
        }
        else if(!vis[0][now.b + now.a])//不能倒满
        {
            next.a = 0;
            next.b = now.b + now.a;
            next.step = now.step + 1;

            q.push(next);
            vis[next.a][next.b] = 1;
            path[next.a][next.b] = pre;
        }
    }

    //B 往 A 倒
    if(now.b && now.a < a)
    {
        pre.step = 6;

        if(now.b >= a - now.a && !vis[a][now.b - (a - now.a)])//能倒满
        {
            next.a = a;
            next.b = now.b - (a - now.a);
            next.step = now.step + 1;

            q.push(next);
            vis[next.a][next.b] = 1;
            path[next.a][next.b] = pre;
        }

        else if(!vis[now.a + now.b][0])//不能倒满
        {
            next.a = now.a + now.b;
            next.b = 0;
            next.step = now.step + 1;

            q.push(next);
            vis[next.a][next.b] = 1;
            path[next.a][next.b] = pre;
        }
    }
```

```
    }
    return 0;
}

void judge_step(int a,int b)
{
    if(a == 0 && b == 0)
    {
        return;
    }

    judge_step(path[a][b].a,path[a][b].b);

    if(path[a][b].step == 1)//这代表了 BFS 的六个方向
    {
        cout << "FILL(A)" << "\n";
    }
    if(path[a][b].step == 2)
    {
        cout << "FILL(B)" << "\n";
    }
    if(path[a][b].step == 3)
    {
        cout << "EMPTY(A)" << "\n";
    }
    if(path[a][b].step == 4)
    {
        cout << "EMPTY(B)" << "\n";
    }
    if(path[a][b].step == 5)
    {
        cout << "POUR(A, B)" << endl;
    }
    if(path[a][b].step == 6)
    {
        cout << "POUR(B, A)"<< "\n";
    }
}

int main()
{
    cout << "请输入 A、B 杯容量和目标水量：";
    cin >> a >> b >> c;
    cout << "\n";
```

```
    if(!bfs())
        cout << "impossible" << "\n";

    return 0;
}

/*
3 5 4
2 8 5
4 9 6

*/
```

③实验结果：

```
请输入A、B杯容量和目标水量：4 9 6

8
FILL(B)
POUR(B, A)
EMPTY(A)
POUR(B, A)
EMPTY(A)
POUR(B, A)
FILL(B)
POUR(B, A)
```

④实验分析：

这一题可以使用 BFS 的思想来做。

对于 A、B 两个瓶子，在任何时刻，“倒水”无非 6 种操作：

1.倒满 A；

2.倒满 B；

3.倒空 A；

4.倒空 B；

5.A 往 B 倒；

6.B 往 A 倒；

对于后两种情况，按源容器是否倒空，还可以再分成两种子情况。

BFS 搜索的方向就是这几种情况。需要注意的是，在这里需要输出步骤，即记录方向。所以应在节点中记录 A、B 的当前水量和执行的操作，并在一般 BFS 访问函数的位置输出执行的操作。

按照 BFS 通用模板建立标志、队列等辅助变量，将这 6 种情况转换成相应的数学表达式。一旦发现任意时刻找到了方案（队列不空），则返回；若始终找不到，则输出 impossible.

实验名称	实验三 计算几何算法的实现						
姓　　名		系院专业		班　　级		学　　号	2018217784
实验日期			指导教师	徐本柱		成　　绩	

一、实验目的和要求

1. 理解线段的性质、叉积和有向面积。
2. 掌握寻找凸包的算法。
3. 综合运用计算几何和搜索中的知识求解有关问题。

二、实验预习内容

1. 预习 ICPC 讲义中的计算几何的内容
2. 了解凸包的概念。

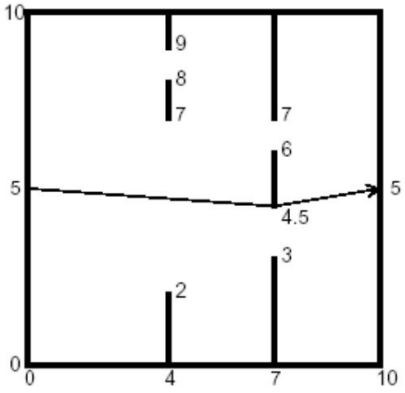
三、实验项目摘要

1. 将讲义第三章第三节中的凸包代码上机运行并检验结果。
2. 完成讲义第三章的课后习题，上机运行并检验结果。
3. 思考：

判线段相交时，如果有个线段的端点在另一条线段上，注意可能与另一条线段上的端点重合，思考这样的情况怎么办。

4. 房间最短路问题：

给定一个内含阻碍墙的房间，求解出一条从起点到终点的最最短路径。房间的边界固定在 $x=0, x=10, y=0$ 和 $y=10$。起点和终点固定在(0,5)和(10,5)。房间里还有 0 到 18 个墙，每个墙有两个门。输入给定的墙的个数，每个墙的 x 位置和两个门的 y 坐标区间，输出最短路的长度。下图是个例子：



四、实验结果与分析（源程序及相关说明）

实验内容：

1.将讲义第三章第三节的代码上机运行并检验结果。

①实验要求：

（1）按照实验要求运行代码，并检验结果。

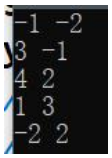
②源程序：

```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
using namespace std;
typedef pair < double, double > POINT;
// function dirction determines the direction that the segment
//p1p turns to p2p with respect to point p
//if return value is positive, means clockwise;
//if return value is negative, menas counter-clockwise;
// naught means on the same line;
vector<POINT> vec;

double direction(POINT p, POINT p1, POINT p2) {
	POINT v1, v2;
	v1.first = p2.first - p.first ;
	v1.second = p2.second - p.second;
	v2.first = p1.first - p.first;
	v2.second = p1.second - p.second;
	return v1.first * v2.second - v1.second * v2.first;
}
//function on_segment determines whether the point p is on the segment p1p2
bool on_segment(POINT p, POINT p1, POINT p2) {
	double min_x = p1.first < p2.first ? p1.first : p2.first ;
	double max_x = p1.first > p2.first ? p1.first : p2.first ;
	double min_y = p1.second < p2.second ? p1.second : p2.second;
	double max_y = p1.second > p2.second ? p1.second : p2.second;
	if (p.first >= min_x && p.first <= max_x && p.second >= min_y && p.second <= max_y) {
		return true;
	}
	else {
		return false;
	}
}
//point startPoint is the polor point that is needed for comparing two other points;
POINT startPoint;
//function sortByPolorAngle provides the realizing of comparing two points, which support
```

```
//the STL function sort();
bool sortByPolorAngle(const POINT & p1, const POINT & p2) {
	double d = direction(startPoint, p1, p2);
	if (d < 0) {
		return true;
	}
	if (d > 0) {
		return false;
	}
	if (d == 0 && on_segment(startPoint, p1, p2) ) {
		return true;
	}
	if (d == 0 && on_segment(p2, startPoint, p1) ) {
		return true;
	}
	return false;
}
//here realizes the process of finding convex hull
void find_convex_hull(vector < POINT > & point) {
	POINT p0 = point[0];
	int k = 0;
	for (int i = 1; i < point.size(); i++) {
		if (point[i].second < p0.second || point[i].second == p0.second && point[i].first < p0.first) {
			p0 = point[i];
			k = i;
		}
	}
	point.erase(point.begin() + k);
	point.insert(point.begin(), p0);
	vector < POINT > convex_hull;
	do {
		convex_hull.push_back(point[0]);
		startPoint = point[0];
		point.erase(point.begin());
		sort(point.begin(), point.end(), sortByPolorAngle);
		if (point[0] == convex_hull[0]) {
			break;
		}
		point.push_back(convex_hull[convex_hull.size() - 1]);
	}
	while (1);
	for (int i = 0; i < convex_hull.size(); i++) {
		cout << convex_hull[i].first << ' ' << convex_hull[i].second << endl;
	}
```

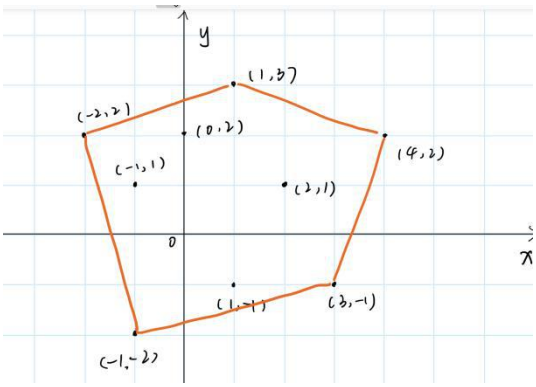
```
}

void initVector()
{
	vec.push_back(POINT(-1, -2));
	vec.push_back(POINT(1, -1));
	vec.push_back(POINT(3, -1));
	vec.push_back(POINT(2, 1));
	vec.push_back(POINT(4, 2));
	vec.push_back(POINT(1, 3));
	vec.push_back(POINT(0, 2));
	vec.push_back(POINT(-1, 1));
	vec.push_back(POINT(-2, 2));
}
int main()
{
	initVector();
	find_convex_hull(vec);
}
```

③实验结果：



程序输出

将输入数据标在坐标系中，可以看出，程序求得的凸包是正确的。



手工检验

④实验分析：

示例凸包代码缺少主函数和数据点集，添加后运行即可得出结果。

2.完成第三章的课后习题，运行代码并检验结果。

①实验要求：

（1）完成课后习题，上机运行并检验结果。

②源程序：

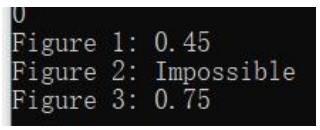
```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
#include<bits/stdc++.h>

using namespace std;

const int maxn = 1010;
int n;
double esp = 1e-8;

template<class Ty1, class Ty2> //减号重载，变成求向量 AB
inline const pair<Ty1, Ty2> operator-(const pair<Ty1, Ty2>&p1, const pair<Ty1, Ty2>&p2)
{
    pair<Ty1, Ty2> ret;
    ret.first = p1.first - p2.first;
    ret.second = p1.second - p2.second;
    return ret;
}

typedef pair<double, double> POINT;

POINT p[maxn];

int sgn(double x)//判定符号
{
    if (fabs(x) < esp)
    {
        return 0;
    }
    else {
        return x > 0 ? 1 :- 1;
    }
}

double CP(POINT A, POINT B) //叉积（这里指的其实是向量。
                                    //这个问题中点和向量形式是一样的，用的时候要注意）
{
    return A.first * B.second - A.second * B.first;
}

bool intersec(POINT A1, POINT B1, POINT A2 , POINT B2)//判断两次跨立相交
```

```
{
    //判定向量 A1B1,A2B2 是否相交
    double C1, C2, C3, C4;

    C1 = CP(A2 - A1, B1 - A1);
    C2 = CP(B1 - A1, B2 - A1);

    C3 = CP(A1 - A2, B2 - A2);
    C4 = CP(B2 - A2, B1 - A2);

    if (sgn(C1 * C2) >= 0 && sgn(C3 * C4) >= 0)//互相跨立
    {
        return true;
    }
    return false;
}

bool segInter() //判定有无非法交点
{
    //对于每一条线段
    //除了与它直接相连的两条
    //检查它是否与剩下的其他线段有交点

    for (int i = 2; i < n - 1; i++)
    {
        for (int j = 1; j < i; j++)
        {
            if (intersec(p[i], p[i + 1], p[j - 1], p[j]))
            {
                return true;
            }
        }
    }

    //最后一根单独比较

    for (int i = 1; i < n - 2; i++)
    {
        if (intersec(p[n - 1], p[0], p[i], p[i + 1]))
        {
            return true;
        }
    }
    return false;
```

```
}

double Area()
{
	double S = 0;
	for (int i = 1; i < n - 1; i++)
	{
		S += CP(p[i] - p[0], p[i + 1] - p[0]);
	}
	return fabs(S) * 0.5;
}

int main()
{
	int num = 0;
	int flag[400] = {0};
	double area[400] = {0};

	while((cin >> n) && (n != 0))
	{
		double a, b;
		for (int i = 0; i < n; i++)
		{
			cin >> a >> b;
			p[i] = POINT(a, b);
		}
		if (n < 3)
		{
			flag[num] = -1;
			num++;
			continue;
		}
		if(segInter())
		{
			flag[num] = -1;
			num++;
			continue;
		}
		else
		{
			flag[num] = 1;
			area[num] = Area();
			num++;
		}
```

```
    }

    for(int i = 0 ; i < 400 ; i++)
    {
     if(flag[i] == -1)
          cout << "Figure " << i + 1 << ": Impossible" << "\n";
     if(flag[i] == 1)
          cout << "Figure " << i + 1 << ": " << setprecision(2) << area[i] << "\n";
    }

    return 0;
}
/*
5
0 0.6
0 1
0.5 0.5
1 1
1 0
4
0 0
0 1
1 0
1 3
5
0 0
0 1
0.5 0.5
1 1
1 0
0
*/
```

③实验结果：



这个结果是正确的。

④实验分析：

多边形面积很容易求，只要用向量积 / 2 就可以得到三角形面积，将这些面积相加就可以得到整个图形的面积。

这一题的重点在于判定多边形是否合法，也就是判断不直接相邻的边会不会相交。参考计算几何知识，可以通过向量的跨立来判断线段的相交。当两个线段对应向量互相跨立，它们对应的线段就相交。只要多边形中有一组边出现这种情况，这多边形就不合法。

需要注意的一点是，这一题中，点是用 pair 的形式表示的。同时，点和向量在形式上是相同的。所以，由两点求向量要通过重载 pair 的减号运算来实现。

3. 判线段相交时，如果有个线段的端点在另一条线段上，注意可能与另一条线段上的端点重合，思考这样的情况怎么办？

①实验要求：

（1）思考这个问题。

②实验分析：

这个时候分三种情况：

其一，两条线段有一部分重合：这时候两线段一定平行、共线，只要检查端点，看它是是在另一条线段上即可。

其二，有个端点和另一条线段端点重合：而且它们不共线。只要检查端点坐标是否重合，很容易就可以确定是否出现了这种情况。

其三，这个端点在另一条线段上，但不与那条线段的端点重合。

设这个端点为 A，由另一条线段的两个端点分别连接这个端点 A，得到两个向量 F1,F2，它们是反向的，并且夹角为 180°。F1F2 的内积等于它们模乘积的相反数。所以，这时候计算 F1F2 的内积，符合上述条件，说明出现了这种情况。

4. 房间最短路问题。

①实验要求：

（1）求出一条最短路线。

②源代码：

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cstring>
#include <vector>
#include <cmath>
#define eps 1e-8
#define INF 1e40
#define N 100

using namespace std;

struct Point //点或者向量还是混用的
{
    double x, y;
    Point() {};
    Point(double x, double y) : x(x), y(y) {};
    //运算符重载

    Point operator - (const Point & e) const
    {
        return Point(x - e.x, y - e.y);
```

```
    }
    double operator ^ (const Point & e) const //叉乘
    {
        return x * e.y - y * e.x;
    }
    double operator * (const Point & e) const
    {
        return x * e.x + y * e.y;
    }
};

struct Line        //直线
{
    Point a, b;
    Line() {};
    Line(Point a, Point b) : a(a), b(b) {}
};

int n, cntEdge, cntPoint;
double path[N][N];
Line line[N];
Point p[N];

double dis(Point a, Point b)//ab 的距离
{
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

int sgn(double x)//符号函数 sgn
{
    if (fabs(x) < eps)
    {
        return 0;
    }
    else {
        return x < 0 ? - 1 : 1;
    }
}

bool onSeg(Point P, Point A1, Point A2)//p 是否在线段 a1a2 上
{
    if (sgn((A1 - P) ^ (A2 - P)) == 0 && sgn((A1 - P) * (A2 - P)) < 0)
    {
        return 1;
```

```
    }
    else {
        return 0;
    }
}

bool ifInter(Line L1, Line L2)//判断线段是否相交
{
    double c1 = (L1.b - L1.a) ^ (L2.a - L1.a);
    double c2 = (L1.b - L1.a) ^ (L2.b - L1.a);
    double c3 = (L2.b - L2.a) ^ (L1.a - L2.a);
    double c4 = (L2.b - L2.a) ^ (L1.b - L2.a);
    return sgn(c1) * sgn(c2) < 0 && sgn(c3) * sgn(c4) < 0;
}

void getEdges()//连接任意两点，没有相交的情况就加入图
{
    for (int i = 1; i <= cntPoint; i++)
    {
        for (int j = 1; j <= cntPoint; j++)
        {
            if (i == j)
            {
                path[i][j] = path[j][i] = 0;
            }
            else
            {
                path[i][j] = path[j][i] = INF;
            }
        }
    }

    bool flag;
    int to, from;

    for (int i = 1; i <= cntPoint - 2; i++)
    {
        for (int j = i + 1; j <= cntPoint - 2; j++)
        {
            from = (i + 3)  / 4, to = (j + 3)  / 4;
            if (from == to)
            {
                continue;
            }
```

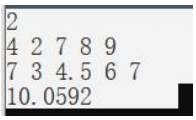
```
            //同一列 跳过
            flag = true;

            for (int k = 3 * from - 2; k <= 3 * to && flag; k++)
            {
                if (ifInter(Line(p[i], p[j]), line[k]))
                {
                    flag = false;
                }
                //如果与任意一条线段相交
            }

            if (flag)
            {
                path[j][i] = path[i][j] = dis(p[i], p[j]);
            }
        }
    }

    for (int i = 1; i <= cntPoint - 2; i++)
    {
        //与中间的点连边
        {
            flag = true;
            to = (i + 3)    / 4 - 1;

            for (int j = 1; j <= 3 * to && flag; j++)
            {
                if (ifInter(Line(p[i], p[cntPoint - 1]), line[j]))
                {
                    flag = false;
                }
            }

            if (flag)
            {
                path[cntPoint - 1][i] = dis(p[i], p[cntPoint - 1]);
                path[i][cntPoint - 1] = dis(p[i], p[cntPoint - 1]);
            }

            to = (i + 3)    / 4 + 1;
            flag = true;

            for (int j = 3 * to - 2; j <= cntEdge && flag; j++)
```

```
                {
                    if (ifInter(Line(p[i], p[cntPoint]), line[j]))
                    {
                        flag = false;
                    }
                }

                if (flag)
                {
                    path[cntPoint][i] = dis(p[i], p[cntPoint]);
                    path[i][cntPoint] = dis(p[i], p[cntPoint]);
                }
            }
            flag = true;
        }

        for (int i = 1; i <= cntEdge; i++)
        {
            if (ifInter(Line(p[cntPoint], p[cntPoint - 1]), line[i]))
            {
                flag = false;
            }
        }

        if (flag)
        {
            path[cntPoint][cntPoint - 1] = dis(p[cntPoint - 1], p[cntPoint]);
            path[cntPoint - 1][cntPoint] = dis(p[cntPoint - 1], p[cntPoint]);
        }

}
void Floyd()//求最短路
{
        for (int i = 1; i <= cntPoint; i++)
        {
            for (int j = 1; j <= cntPoint; j++)
            {
                for (int k = 1; k <= cntPoint; k++)
                {
                    if (path[i][k] < INF && path[k][j] < INF && path[i][k] + path[k][j] < path[i][j])
                    {
                        path[i][j] = path[i][k] + path[k][j];
                    }
                }
```

```
            }
        }
}

int main()
{
        while ( cin >> n && n != - 1)
        {
                cntEdge = 0;
                cntPoint = 0;

                for (int i = 0; i < n; i++)
                {
                        double x, y1, y2, y3, y4;
                        cin >> x >> y1 >> y2 >> y3 >> y4;

                        line[++cntEdge] = Line(Point(x, 0), p[++cntPoint] = Point(x, y1));

                        line[++cntEdge] = Line(p[++cntPoint] = Point(x, y2), p[++cntPoint] = Point(x, y3));

                        line[++cntEdge] = Line(p[++cntPoint] = Point(x, y4), Point(x, 10));
                }

                p[++cntPoint] = Point(0, 5);
                p[++cntPoint] = Point(10, 5);

                getEdges();
                Floyd();
                cout << path[cntPoint - 1][cntPoint];
        }
        return 0;
}

/*
1
5 4 6 7 8
2

4 2 7 8 9
7 3 4.5 6 7
-1

10
10.06
```

*/

③实验结果：



程序输出

这和 POJ 1556 的示例相符。结果是正确的。

④实验分析：

这道题是一个最短路问题。

若将每一堵墙上的门的两端视为可能的节点，那么如果有 n 堵墙，节点就有 4n 个。我们要找的是从起点到终点的最短路径，**这个路径一定是两个节点的连线或是由几条这样的连线组成的。而如果两点的连线与墙相交（端点除外），那么该连线是不能存在的。**

所以首先尝试将所有的点两两连线，对于每一条连线，检查它是否与墙相交。如果相交，则删除这条连线；如果不相交，则保留该连线。最后留下的所有连线就能构成一张图。使用任意一种最短路算法对该图求一遍起点到终点的最短路径长度，就可以得到最短路径长。

《程序设计艺术与方法》课程实验报告

实验名称	实验四 动态规划算法的实现						
姓　　名		系院专业		班　　级		学　　号	
实验日期		指导教师	徐本柱			成　　绩	

一、实验目的和要求

1. 理解动态规划的基本思想、动态规划算法的基本步骤。
2. 掌握动态规划算法实际步骤。

二、实验预习内容

1．预习 ICPC 讲义中的动态规划内容。

三、实验项目摘要

1. 求两个字符串的最长公共子序列。

X 的一个子序列是相应于 X 下标序列{1, 2, ..., m}的一个子序列，求解两个序列的所有子序列中长度最大的，例如输入：pear, peach 输出：pea。

2. 求能够将串 a 变为串 b 最少修改次数。

给定两个字符串 a 和 b，现将串 a 通过变换变为串 b，可用的操作为，删除串 a 中的一个字符；在串 a 的某个位置插入一个元素；将串 a 中的某个字母换为另一个字母。对于任意的串 a 和串 b，输出最少多少次能够将串变为串 b。

思考：输出变换的步骤。

3. 输入一个矩阵，计算所有的子矩阵中和的最大值。

例如，输入

0 -2 -7 0

9 2 -6 2

-4 1 -4 1

-1 8 0 -2

输出为：15

思考：当矩阵很大时，比如 100*100 的矩阵，你的程序还能够很快的得出结果吗，如果不能，请思考如何用动态规划的思想解决。

四、实验结果与分析（源程序及相关说明）

实验内容：

1. 求两个字符串的最长公共子序列。

①实验要求：

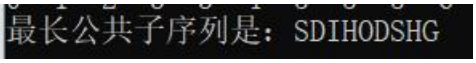
（1）按照实验要求编写程序，并检验结果。

②源程序：

```
//求最长公共子序列
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
using namespace std;

void getLCS(string S1, string S2)
{
    int L1 = S1.length();
    int L2 = S2.length();

    int C[L2 + 1][L1 + 1] ;

    for(int i = 0; i < L2 + 1; i++)//初始化 第一行第一列代表与空串的 LCS，一定是 0
    C[i][0] = 0;
    for(int i = 0; i < L1 + 1; i++)
    C[0][i] = 0;

    for (int i = 1; i <= L1; i++)//填 LCS 表格
    {
        for (int j = 1; j <= L2; j++)
        {
            if (S1[i - 1] == S2[j - 1])
            {
                C[i][j] = C[i - 1][j - 1] + 1;
            }
            else
            {
                C[i][j] = max(C[i][j - 1], C[i - 1][j]);
            }
        }
    }

    for(int i = 0; i < L1 + 1; i++)
    {
    for(int j = 0; j < L2 + 1; j++)
```

```
    {
        cout << C[i][j] << "    ";
        }
        cout << "\n";
    }

    string LCS = "";

    while (L1 != 0)
    {
        if (C[L1][L2] == C[L1 - 1][L2])
        {
            L1--;
            continue;
        }
        if (C[L1][L2] == C[L1][L2 - 1])
        {
            L2--;
            continue;
        }
        if (C[L1][L2] == C[L1 - 1][L2 - 1] + 1)
        {
            LCS += S1[L1 - 1];
            L1--;
            L2--;
        }
    }

    reverse(LCS.begin(), LCS.end());
    cout << "最长公共子序列是："<< LCS << endl;
}

int main()
{
//  string S1, S2;
//  cin >> S1;
//  cin >> S2;

    string S1 = "ABCPDSFJGODIHJOFDIUSHGD";
    string S2 = "OSDIHGKODGHBLKSJBHKAGHI";

    getLCS(S1, S2);
}
```

③实验结果：



程序输出

手工检查输入的两个字符串，可以看到程序输出的结果是正确的。

④实验分析：

对于长度分别为 m、n 的两个字符序列 X_m、Y_n，本题要求出它们的最长公共子序列 LCS(X_m, Y_n)。这是一个最优化问题。

1.如果它们的最后一个字符相同，即 $X_m == Y_n$，那么这个字符一定是 LCS 的最后一个字符。

2.如果它们最后一个字符不相同，这个时候最后一个字符一定不包含在 LCS 中。也就是说 LCS(X_m, Y_n) 一定等于 LCS(X_{n-1}，Y_m) 和 LCS(X_n，Y_{m-1}) 两者中比较长的一个。

这两个问题覆盖了所有的子情况。由上面的分析，设二维数组 C[i+1][j+1] 的元素 C[i, j] = LCS(X_i, Y_j) ，则它的取值具有如下的形式：

$$C[i,j]=\begin{cases}0 & 若\ i=0或j=0\\ C[i-1,j-1]+1 & 若\ i,j>0,x_i=y_j\\ \max\{C[i,j-1],C[i-1,j]\} & 若\ i,j>0,x_i\neq y_j\end{cases}$$

（当 i 或 j 为 0 时，其中一个字符串一定是空串，LCS 长度为 0）

实际运算后二维数组 C 的情况如下：

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

这个数组右下角的元素值就是 LCS 的长度。

若要输出 LCS，则应从右下角元素回溯，直到第 0 行或者第 0 列。研究矩阵 C 和 LCS 的关系，可以得出回溯的原则是：

1.若该元素和其左边相邻元素相等，左移一格，并转下一次循环；

2.若不符合（1），且该元素和其上边相邻元素相等，上移一格，并转下一次循环；

3.若不符合（1）（2），且该元素和其左上方元素差 1，则把左上方的元素加入栈，向左上方移动一格；

回溯结束后将栈中元素依次弹出，就是两个字符串的一个 LCS 了。

2.求能够将串 a 变为串 b 的最少修改次数。

①实验要求：

（1）给定两个字符串 a 和 b，现将串 a 通过变换变为串 b，可用的操作为，删除串 a 中的一个字符；在串 a 的某个位置插入一个元素；将串 a 中的某个字母换为另一个字母。对于任

意的串 a 和串 b，输出最少多少次能够将串变为串 b。

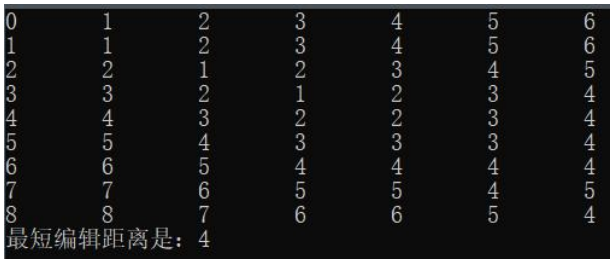
②源程序：

```
//求最小修改次数
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
using namespace std;

int dis[1000][1000];

int min(int a, int b, int c)
{
	int temp;
	if(a > b)
		temp = b;
	else
		temp = a;

	if(temp > c)
		return c;
	else
		return temp;
}

void print(int L1, int L2)
{
	for(int i = 0; i < L1 + 1; i++)
	{
	for(int j = 0; j < L2 + 1; j++)
	{
		cout << dis[i][j] << "\t";
		}
		cout << "\n";
	}

}

void getEditDis(string S11, string S22)
{
	int i, j ;

	string S1 = "0" + S11;//string 下标是从 0 开始的，而代价表是从 1 开始的
	string S2 = "0" + S22;//需要在前面加个 0 防止第一个字被忽略
```

```
int L1 = S1.length();
int L2 = S2.length();

for(i = 0; i <= L1; i++)
	dis[i][0] = i ;
for(j = 0; j <= L2; j++)
	dis[0][j] = j ;

for(i = 1; i <= L1; i++)
{
	for(j = 1; j <= L2; j++)
	{

		if(S1[i] == S2[j])//当新增的一对字符相等
		{
			//cout << S1[i] << "    " << S2[j] << endl;
			dis[i][j] = dis[i - 1][j - 1] ;//不修改，继承
		}
		else
		{
			//cout << S1[i] << "    " << S2[j] << endl;
			dis[i][j] = dis[i - 1][j - 1] + 1 ;//修改  +1
		}
		//状态转移方程  比较修改  删除  插入各自的代价
		dis[i][j] = min( dis[i][j], dis[i - 1][j] + 1, dis[i][j - 1] + 1 ) ;

		if(dis[i][j] == dis[i - 1][j] + 1)//删除
		{

		}
		else if(dis[i][j] == dis[i][j - 1] + 1)//插入
		{

		}
		else if(dis[i][j] == dis[i - 1][j - 1] + 1)//修改
		{

		}
		else//未改动
		{

		}
```

```
            }
        }

        print(L1, L2);
        cout << "最短编辑距离是：" << dis[L1][L2];
}

int main()
{
//    string S1, S2;
//    cin >> S1;
//    cin >> S2;

        string S1 = "sfdxbqw";
        string S2 = "gfdgw";

        getEditDis(S1, S2);
}
```

③实验结果：



实验结果

这个结果是正确的。

④实验分析：

这一题和上一题类似，可以通过动态规划，并填代价表的方式来解决。

把串 A 修改成串 B 有三种可能的途径：增加、删除、改动。这三种方法都会消耗一次操作次数。并且，无论修改哪个串，例如“给 A 增加一个字符变成 B”和“删掉 B 的一个字符变成 A”，它们都是等价的操作，只不过方向反过来了。

假设要求的字符序列是 A[1…m]、B[1…n]。设最短编辑距离为 SED。

这里以第一个字符作为例子，并且假设是把 A 串变成 B 串。

如果它们一样，则总的 SED = A[2…m]、B[2…n]的 SED。

而如果不一样，就会有以下几种情况：

1.把 A1 改成 B1，总的 SED = A[2…m]、B[2…n]的 SED+1；

2.删掉 A1，总的 SED = A[2…m]、B[1…n]的 SED；

3.把 B1 复制一个插到 A1 前面，总的 SED = A[1…m]、B[2…n]的 SED。

以此类推，可以得出判定第 i 个字符的情况。

使用填代价表的方式解题。代价表就是一个矩阵 C，长宽分别是 B、A 串的长度+1。C[i][j]表示从 B[1…i]变到 A[1…j]的 SED。

初始化时需要将 0 行初始化成 0-m，0 列初始化成 0-n。代表了由空串变成另一个串需要的修改次数（即全部添加）。

对于其他格子，结合上面的分析，设要填的格子为 X，则三种字符串操作分别对应的填表方式如下：

1.删除。将 X 左边相邻格子的值+1 填入 X。

2.添加。将 X 上边相邻格子的值+1 填入 X。

3.修改。将 X 左上方格子的值+1 填入 X。

填表完成后，矩阵 C 右下角的元素值就是总的 SED。

```
0       1       2       3       4
1       1       2       3       4
2       2       2       3       4
3       3       3       3       4
4       3       4       4       4
5       4       4       5       4
最短编辑距离是：4
```

填表的结果

3. 输入一个矩阵，计算所有的子矩阵中和的最大值。

①实验要求：

（1）思考：当矩阵很大时，比如 100*100 的矩阵，你的程序还能够很快的得出结果吗，如果不能，请思考如何用动态规划的思想解决。

②源程序：

```
#include <iostream>
#include <cstring>

using namespace std;
int mat[1000][1000];

int maxSubSeq(int a[],int n)//求一维数组最大子序列
{
    int sum = 0;
    int b = 0;

    for(int i = 0;i < n;i++)
    {
        if(b < 0)
            b = a[i];
        else
            b += a[i];
        if(b > sum)
            sum = b;
    }
    return sum;
```

```
}

int main()
{
	int m,n;
	cout << "输入矩阵的行列数 m, n:";
	cin >> m;
	cin >> n;

	for(int i = 0; i < m; i++)
	{
		for(int j = 0; j < n; j++)
		{
			cin >> mat[i][j];
		}
	}

	int maxSubArr = 0;
	int temp[n];

	for(int i = 0; i < m; i++)
	{
		for(int j = 0; j < n; j++)
		{
			temp[j] = 0;
		}
		for(int j = i; j < m; j++)
		{
			for(int k = 0; k < n; k++)
				temp[k] += mat[j][k];

			int sum = maxSubSeq(temp, n);

			if(sum > maxSubArr)
				maxSubArr = sum;
		}
	}

	cout << maxSubArr;
	system("pause");
}


/*
```

```
4 3
1 2 3
4 5 6
7 8 9
10 11 12

4 3
1 1 1
1 1 1
1 1 1
1 1 1
*/
```

③实验结果：

```
输入矩阵的行列数m, n:4 3
1 2 3
4 5 6
7 8 9
10 11 12
最大子矩阵和：78
```

程序输出

这个结果是正确的。

④实验分析：

这个问题是课上最大子段和问题的二维拓展。

在一维的最大子段和问题中，对于元素 a[j]，如果最大子段和包含它，则该子段和等于 a[j]之前元素的最大子段和加上 a[j]。这就可以转化成动态规划问题。

而对于最大子矩阵和，如果先确定其中的一维，就可以对另一维使用最大子段和的方法求解。所以可用两个外层循环枚举行数的组合，对于这个确定的行范围，将每一列累加起来，划归成一行，再求出这一行最大的子段和，记录下来。当行数枚举完成，最大子矩阵的和也就求出来了。