

**北京邮电大学软件学院**  
**2019-2020 学年第一学期实验报告**

课程名称: 算法分析与设计

项目名称: 实验五 动态规划法

项目完成人:

姓名: 平雅霓 学号: 2017211949

指导教师: 李朝晖

日 期: 2019 年 12 月 17 日

## 一、 实验目的

- 1、 深刻理解并掌握动态规划法的设计思想；
- 2、 提高应用动态规划法设计算法的技能

## 二、 实验内容

1. 0—1 背包问题
2. 带权重的区间调度问题
3. 汽车加油行驶问题
4. 子序列问题

## 三、 实验环境

Eclipse、 Windows10

## 四、 实验结果

### 1. 0—1 背包问题

**测试数据：**对于  $n=5$  的 0-1 背包问题，背包容量为 10。

	重量	价值
物体 1	2	6
物体 2	2	3
物体 3	6	5
物体 4	5	4
物体 5	4	6

**测试结果：**

0	0	0	0	0	0	0	0	0	0	0
0	0	6	6	6	6	6	6	6	6	6
0	0	6	6	9	9	9	9	9	9	9
0	0	6	6	9	9	9	9	11	11	14
0	0	6	6	9	9	9	10	11	13	14
0	0	6	6	9	9	12	12	15	15	15

最大价值为15

最大价值为 15

**时间复杂度：**  $O(n \times C)$

## 2. 带权的区间调度问题

**测试数据：** 如下表：

	开始时间	结束时间	权重
任务 1	0	4	2
任务 2	1	6	4
任务 3	5	7	4
任务 4	2	9	7
任务 5	8	10	2
任务 6	8	11	1

**测试结果：**

dp 结果： 2 4 6 7 8 7  
可执行的任务数为：3  
执行的任务为：  
( 0, 4, 2)  
( 5, 7, 4)  
( 8, 10, 2)

**时间复杂度：**  $O(n \log n)$

## 3. 汽车加油行驶问题

**测试数据：** 如下图所示

Sample Input

9 3 2 3 6

0 0 0 0 1 0 0 0 0

0 0 0 1 0 1 1 0 0

1 0 1 0 0 0 0 1 0

0 0 0 0 0 1 0 0 1

1 0 0 1 0 0 1 0 0

0 1 0 0 0 0 0 1 0

0 0 0 0 1 0 0 0 1

1 0 0 1 0 0 0 1 0

0 1 0 0 0 0 0 0 0

测试结果:

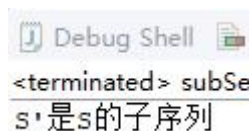
```
Debug Shell Coverage Console
<terminated> test (3) [Java Application] C:\Progr
9 3 2 3 6
0 0 0 0 1 0 0 0 0
0 0 0 1 0 1 1 0 0
1 0 1 0 0 0 0 1 0
0 0 0 0 0 1 0 0 1
1 0 0 1 0 0 1 0 0
0 1 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0 1
1 0 0 1 0 0 0 1 0
0 1 0 0 0 0 0 0 0
12
```

#### 4. 子序列问题

测试数据 (1) :

```
String S[] = {"买Yahoo股票","买eBay股票","买Yahoo股票","买Oracle股票"};
String S_[] = {"买Yahoo股票","买Oracle股票"};
```

测试结果 (1) :

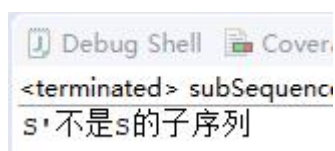


```
Debug Shell
<terminated> subSe
s'是s的子序列
```

**测试数据 (2) :**

```
String S[] = {"买Yahoo股票", "买eBay股票", "买Yahoo股票", "买Oracle股票"};
String S_[] = {"买Yahoo股票", "买Oracle股票", "买eBay股票"};
```

**测试结果 (2) :**



```
Debug Shell
<terminated> subSequence
s'不是s的子序列
```

**时间复杂度:**  $O(m+n)$

## 五、 附录

### 1. 0-1 背包问题

#### • 问题分析

0-1 背包问题是给定  $n$  种物品和一背包, 物品  $i$  的重量是  $w_i$ , 其价值为  $v_i$ , 背包的容量为  $C$ 。问应如何选择装入背包的物品, 使得装入背包中物品的总价值最大。此题的解法在于找出动态规划的目标函数, 设  $x_i$  表示物品  $i$  装入背包的情况, 则当  $x_i = 0$  时, 表示物品  $i$  没有被装入背包,  $x_i = 1$  时, 表示物品  $i$  被装入背包。根据问题的要求, 有如下约束条件和目标函数:

$$\sum_{i=1}^n w_i x_i \leq C$$

$$x_i \in \{0,1\} (1 \leq i \leq n)$$

于是, 问题归结为寻找一个满足约束条件式, 并使目标函数式达到最大的解向量  $X=(x_1, x_2, \dots, x_n)$ 。

## • 设计方案

0/1 背包问题可以看作是决策一个序列( $x_1, x_2, \dots, x_n$ ), 对任一变量  $x_i$  的决策是决定  $x_i=1$  还是  $x_i=0$ 。在对  $x_{i-1}$  决策后, 已确定了 ( $x_1, \dots, x_{i-1}$ ), 在决策  $x_i$  时, 问题处于下列两种状态之一:

(1) 背包容量不足以装入物品  $i$ , 则  $x_i=0$ , 背包不增加价值;

(2) 背包容量可以装入物品  $i$ , 则  $x_i=1$ , 背包的价值增加了  $v_i$ 。这两种情况下背包价值的最大者应该是对  $x_i$  决策后的背包价值。令  $V(i, j)$  表示在前  $i(1 \leq i \leq n)$  个物品中能够装入容量为  $j(1 \leq j \leq C)$  的背包中的物品的最大值, 则可以得到如下动态规划函数

$$V(i, 0) = V(0, j) = 0$$

$$V(i, j) = \begin{cases} V(i-1, j) \\ \max \{V(i-1, j), V(i-1, j-w_i) + v_i\} & j > w_i \end{cases}$$

## • 算法

根据动态规划函数, 用一个  $(n+1) \times (C+1)$  的二维表  $V$ ,  $V[i][j]$  表示把前  $i$  个物品装入容量为  $j$  的背包中获得的最大价值。按下述方法来划分阶段:  
第一阶段, 只装入前 1 个物品, 确定在各种情况下的背包能够得到的最大价值; 第二阶段, 只装入前 2 个物品, 确定在各种情况下的背包能够得到的最大价值; 依此类推...

直到第  $n$  个阶段。最后,  $V(n, C)$  便是在容量为  $C$  的背包中装入  $n$  个物品时取得的最大价值。

```

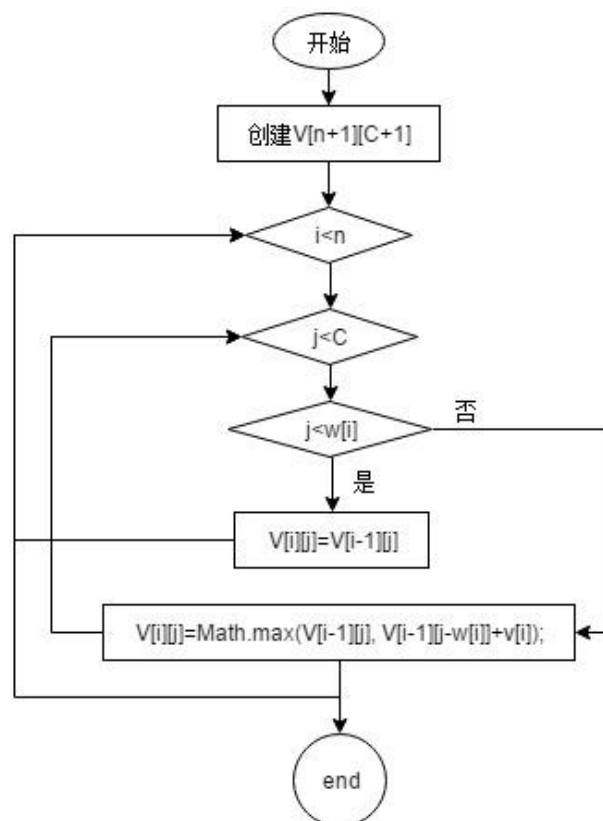
int w[] = {0,2,2,6,5,4}; //每个物体的重量
int v[] = {0,6,3,5,4,6}; //每个物体的价值
int V[][]= new int[6][11];
for(int i=1;i<w.length;i++)
    for(int j=1;j<=limited_weight;j++) {
        if(j<w[i])
            V[i][j]=V[i-1][j];
        else
            V[i][j]=Math.max(V[i-1][j], V[i-1][j-w[i]]+v[i]);
    }

```

其中  $\text{Math.max}(V[i-1][j], V[i-1][j-w[i]]+V[i])$  为算法的核心部分, 即给二维表赋值的语句。

		0	1	2	3	4	5	6	7	8	9	10	
	0	0	0	0	0	0	0	0	0	0	0	0	
$w_1=2 \ v_1=6$	1	0	0	6	6	6	6	6	6	6	6	6	$x_1=1$
$w_2=2 \ v_2=3$	2	0	0	6	6	9	9	9	9	9	9	9	$x_2=1$
$w_3=6 \ v_3=5$	3	0	0	6	6	9	9	9	9	11	11	14	$x_3=0$
$w_4=5 \ v_4=4$	4	0	0	6	6	9	9	9	10	11	13	14	$x_4=0$
$w_5=4 \ v_5=6$	5	0	0	6	6	9	9	12	12	12	15	15	$x_5=1$

## ● 设计图



## ● 程序

```
package fifth_lab;

public class _01bag {
    public static int V[][];
    public static void main(String[] args) {
        int limited_weight=10;
        int w[] = {0,2,2,6,5,4}; //每个物体的重量
        int v[] = {0,6,3,5,4,6}; //每个物体的价值
        int V[][]= new int[6][11];
        for(int i=1;i<w.length;i++)
            for(int j=1;j<=limited_weight;j++) {
                if(j<w[i])
                    V[i][j]=V[i-1][j];
                else
                    V[i][j]=Math.max(V[i-1][j], V[i-1][j-w[i]]+v[i]);
            }
        //打印表格
        for(int i=0;i<w.length;i++) {
            for(int j=0;j<=limited_weight;j++) {
                System.out.print(V[i][j]+"\\t");
            }
            System.out.print("\\n");
        }
        System.out.print("最大价值为"+V[w.length-1][limited_weight]);
    }
}
```

## ● 调试心得

通过本次实验，我对 0-1 背包问题有了更深的了解，并且对动态规划法的思想有了初步的认识，对目标函数的设计更加熟练，这个问题的难点也在于约束条件和目标函数的设计，设计时要合理。



## 2. 带权的区间调度问题

### • 问题分析

存在单一资源  $R$ ，有  $n$  个需求  $\{1, 2, \dots, n\}$ ，每个需求指定一个开始时间  $b_i$  与一个结束时间  $e_i$ ，在时间区间  $[b_i, e_i]$  内该需求想要占用资源  $R$ ，资源  $R$  一旦被占用则无法被其他需求利用。每个需求  $i$  带有一个权值  $v_i$ ，代表该需求被满足之后能够带来的价值或者贡献。如果两个需求的时间区间不重叠，那么它们是相容的。带权值的区间调度问题即，对上面所描述的情境，求出一组相容子集  $S$  使得  $S$  中的区间权值之和最大。

在此问题中，我们还是每次选取结束时间最早的为最优策略，采用动态规划法，每次比较与当前任务的相容的前一个任务的  $dp$  值与当加上当前任务的权重的大小，取较大的值，最后选出  $dp$  值最大的一个解。

目标函数为：

$$dp[i] = \max \{dp[j], dp[j] + jobs[i].weight\}$$

### • 设计方案

设计一个类存储工作的属性，属性包括开始时间和结束时间，之后针对所有工作的结束时间从小到大进行排序，与当前任务的相容的前一个任务的  $dp$  值与当加上当前任务的权重的大小，取较大的值，最后选出  $dp$  值最大的一个解。

### • 算法

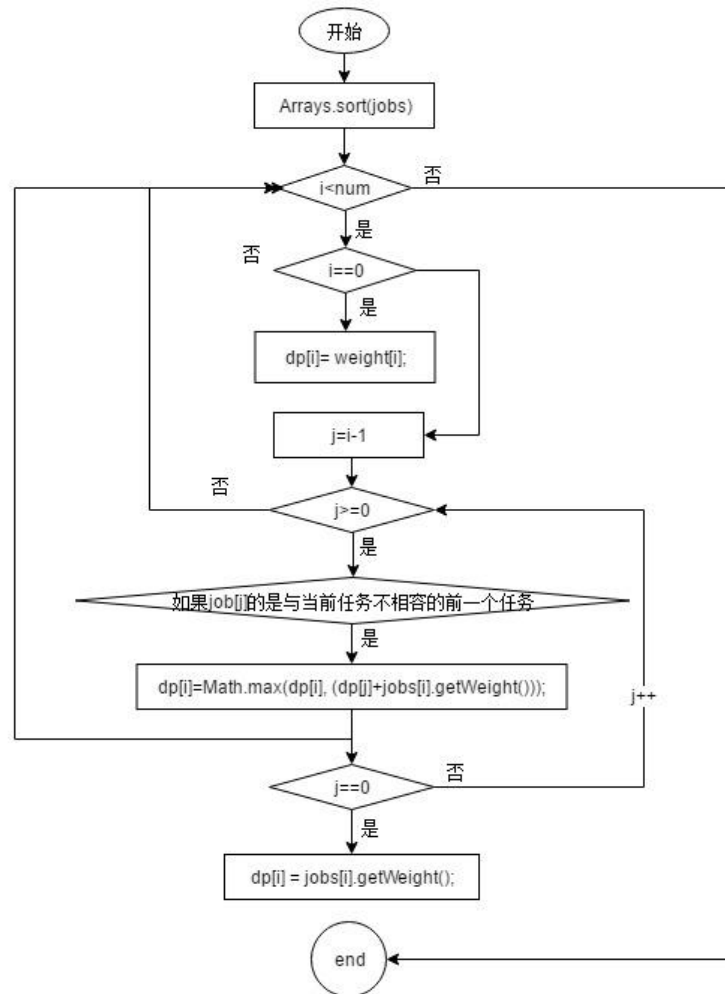
使用 Arrays.sort()函数对人物按结束时间排序，然后通过一个二重循环寻找与每一个当前任务相容的前一个任务，通过 Math.Max(dp[i],dp[j]+jobs[i].getWeight())进行比较，更新当前任务的 dp 值。

```
//对工作的结束时间进行排序
Arrays.sort(jobs);

int dp[]= new int[num];

//动态规划表赋值
for(i=0;i<num;i++) {
    if(i==0) {
        dp[i]= weight[i];
        (solveSpace[0]).add(i);
    }
    else {
        for(int j=i-1;j>=0;j--) {
            if(jobs[j].getEnd()<jobs[i].getStart()) {
                dp[i]=Math.max(dp[i], (dp[j]+jobs[i].getWeight()));
                for(int k=0;k<solveSpace[j].size();k++)
                    solveSpace[i].add(solveSpace[j].get(k));
                solveSpace[i].add(i);
                break;
            }
        }
        if(j==0) {
            dp[i] = jobs[i].getWeight();
            solveSpace[i].add(i);}
    }
}
```

- 设计图



## • 程序

```

package fifth_lab;
import java.util.Arrays;
import java.util.Vector;

public class interval {

    public static void main(String[] args) {
        int num=6;
        int []startTime = {0,1,5,2,8,8};
        int []endTime = {4,6,7,9,10,11};
        int []weight = {2,4,4,7,2,1};
        Job[] jobs = new Job[num];        //创建的工作
        Vector<Integer> solveSpace[]=new Vector[6];

        int maxWeight;

```

```

int i = 0;
while(i<num){
    jobs[i]=new Job(startTime[i],endTime[i],weight[i]);
    solveSpace[i]=new Vector<Integer>();
    i++;
}

//对工作的结束时间进行排序
Arrays.sort(jobs);

int dp[]= new int[num];

//动态规划表赋值
for(i=0;i<num;i++) {
    if(i==0) {
        dp[i]= weight[i];
        (solveSpace[0]).add(i);
    }
    else {
        for(int j=i-1;j>=0;j--) {
            if(jobs[j].getEnd()<jobs[i].getStart()) {
                dp[i]=Math.max(dp[i], (dp[j]+jobs[i].getWeight()));
                for(int k=0;k<solveSpace[j].size();k++)
                    solveSpace[i].add(solveSpace[j].get(k));
                solveSpace[i].add(i);
                break;
            }
            if(j==0) {
                dp[i] = jobs[i].getWeight();
                solveSpace[i].add(i);}
        }
    }
}

System.out.print("dp 结果:  ");
for(i=0;i<num;i++){
    System.out.print(dp[i]+" ");
}

int flag = 0;
maxWeight = dp[0];
for(i=1;i<num;i++){
    if(dp[i]>maxWeight) {
        maxWeight = dp[i];
    }
}

```

```

        flag = i;
    }
}
System.out.println("\n 可执行的任务数为:"+solveSpace[flag].size());
System.out.println("执行的任务为:  ");
for(i=0;i<solveSpace[flag].size();i++) {
    System.out.println("( "+jobs[solveSpace[flag].get(i)].getStart()+" "+
jobs[solveSpace[flag].get(i)].getEnd()+" "+jobs[solveSpace[flag].get(i)].getWeight()+"");

}

}
}
}

```

```

@SuppressWarnings("rawtypes")
class Job implements Comparable{
    private int start;
    private int end;
    private int weight;
    public Job(int start, int end,int weight) {
        this.start = start;
        this.end = end;
        this.weight = weight;
    }
    public int getStart() {
        return this.start;
    }
    public int getEnd() {
        return this.end;
    }
    public int getWeight() {
        return this.weight;
    }
    @Override
    public int compareTo(Object o) {
        Job job = (Job) o;
        if (this.end > job.getEnd())
            return 1;
        else if (this.end == job.getEnd())
            return 0;
        else
            return -1;
    }
}

```

```
}
```

### • 调试心得

通过本次实验，我实现了区间调度问题的动态规划算法，以及如何设计目标函数，对设计约束条件和目标函数更加熟练，此算法的难点依旧在于寻找与当前任务相容的前一个任务与设计目标函数。

## 3. 汽车加油行驶

### • 问题分析

给定一个  $N \times N$  的方形网格，设其左上角为起点，坐标(1, 1)，X 轴向右为正，Y 向下为正，每个方格边长为 1。一辆汽车从起点出发驶向右下终点，其坐标为 (N,N)。在若干个网格交叉点处，设置了油库，可供汽车在行驶途中加油。汽车在行驶过程中应该遵守如下规则：

(1) 汽车只能沿网格边行驶，装满油后能行驶  $K$  条网格边。出发时汽车已装满油，起点与终点处不设置油库；

(2) 当汽车行驶经过一条网格边的时候，若其  $X$  坐标或者  $Y$  坐标减小，则应付费  $B$ ，否则免付费用；

(3) 汽车行驶过程中遇油库应该加油并付加油费用  $A$ ；(4) 在需要时可在网格点处增设油库，并付增设费用  $C$ (不含加油费用  $A$ )

(5) 上述(1)~(4)中的各数都是正整数请找一条总费用最少的最优行驶路线。

### • 算法

$f[i][j][0]$ 表示汽车从网格点(1,1)行驶至网格点(i,j)所需的最少费用

$f[i][j][1]$ 表示汽车行驶至网格点(i,j)后还能行驶的网格边数

$s[i][0]$ 表示 x 轴方向

$s[i][1]$ 表示 y 轴方向

$s[i][2]$ 表示行驶费用

$f[i][j][0] = \min\{f[i+s[k][0]][j+s[k][1]][0] + s[k][2]\}$

$f[i][j][1] = f[i+s[k][0]][j+s[k][1]][1] - 1$

$f[1][1][0] = 0$

$f[1][1][1] = K$

$f[i][j][0] = f[i][j][0] + A, f[i][j][1] = K$  如果(i, j)是油库

$f[i][j][0] = f[i][j][0] + C + A, f[i][j][1] = K$  (i, j)不是油库,且  $f[i][j][1]$   
 $= 0$

## ● 程序

```
package fifth_lab;

import java.util.Scanner;

public class carDriving {
    private static int N ;
    private static int A;
    private static int C;
    private static int B;
    private static int K;

    private static int[][][] f = new int[50][50][2];
    private static int[][] s = {{-1,0,B},{0,-1,B},{1,0,0},{0,1,0}};

    private static int[][] a = new int[50][50]; //方形网络
```

```

private static int MAX = 100000;
private static int leastFee;

public static void main(String[] args){
    Scanner input = new Scanner(System.in);
    N = input.nextInt();
    K = input.nextInt();
    A = input.nextInt();
    B = input.nextInt();
    C = input.nextInt();

    for(int i=1; i<=N; i++){//输入方形网络
        for(int j=1; j<=N; j++){
            a[i][j] = input.nextInt();
        }
    }

    leastFee = dynamic();
    System.out.println(leastFee);//最优行驶路线所需的费用，即最小费用
}

private static int dynamic(){
    int i, j, k;
    for (i=1;i<=N;i++){
        for (j=1;j<=N;j++){
            f[i][j][0]=MAX;
            f[i][j][1]=K;
        }
    }

    f[1][1][0] = 0;
    f[1][1][1] = K;

    boolean finish = false;
    int tx, ty;
    while(!finish){
        finish = true;
        for(i=1; i<=N; i++){
            for(j=1; j<=N; j++){
                if(i==1 && j==1)
                    continue;
                int minFee = MAX;
                int driveEdges = MAX;
                int fee, canDriveEdges;

```



```

        for(k=0; k<4; k++){ //可走的四个方向
            tx = i + s[k][0];
            ty = j + s[k][1];
            if(tx<1 || ty<1 || tx>N || ty>N) //如果出界
                continue;

            fee = f[tx][ty][0] + s[k][2];
            canDriveEdges = f[tx][ty][1] - 1;
            if(a[i][j] == 1){ //如果是油库
                fee += A;
                canDriveEdges = K;
            }
            if(a[i][j]==0 && canDriveEdges == 0 && (i!=N||j!=N)){ //如果不是油库,且油已经用完

                fee += A + C;
                canDriveEdges = K;
            }
            if(fee < minFee){ //记录更少费用
                minFee = fee;
                driveEdges = canDriveEdges;
            }
        }

        if(f[i][j][0] > minFee){ //如果有更优解
            finish = false;
            f[i][j][0] = minFee;
            f[i][j][1] = driveEdges;
        }
    }
}

return f[N][N][0]; //汽车从网格点(1,1)行驶至网格点(N,N)所需的最少费用(亦即从起点到终点), 此为所求
}
}

```

## • 调试心得

通过本次实验，我掌握了汽车加油行驶问题的动态规划法求解，本问题的难点在于设计目标函数和约束条件。

## 4. 子序列问题

### • 问题分析

存在具有时间自然顺序的一种数据资源，给定这类事件的一个长序列  $S$ 。你的朋友想要有一种有效的方式来发现其中的某些模式，下面四个事件出现在这个系列  $S$  中，按顺序出现，但是不一定连续出现，例如："买 Yahoo 股票","买 eBay 股票","买 Yahoo 股票","买 Oracle 股票"

从一组可能的事件和  $n$  个这种事件的序列  $S$  开始，一个给定的事件在  $S$  中可能出现多次。如果存在一种方式，从  $S$  中删除某些事件，已使得留下的时间按照顺序等于序列  $S'$ ，我们就说序列  $S'$  的是  $S$  的子序列。

此题要求给出时间复杂度为  $O(m+n)$  的算法解题。

由要求的时间复杂度可以看出，我们可以采用双重循环来做此题，即序列  $S$  的长度  $m$ ，子序列  $S'$  长度  $n$ 。

### • 设计方案

外循环为序列，按序遍历序列中每一个字符串，使之与子序列中的字符串逐个比较，如果相同，则继续向后比较，不同，则跳过子序列中当前的字符串取比较下一个，记录子序列中符合条件的字符串数，若等于母序列中的字符串数，则该  $S'$  是  $S$  的子序列。

### • 算法

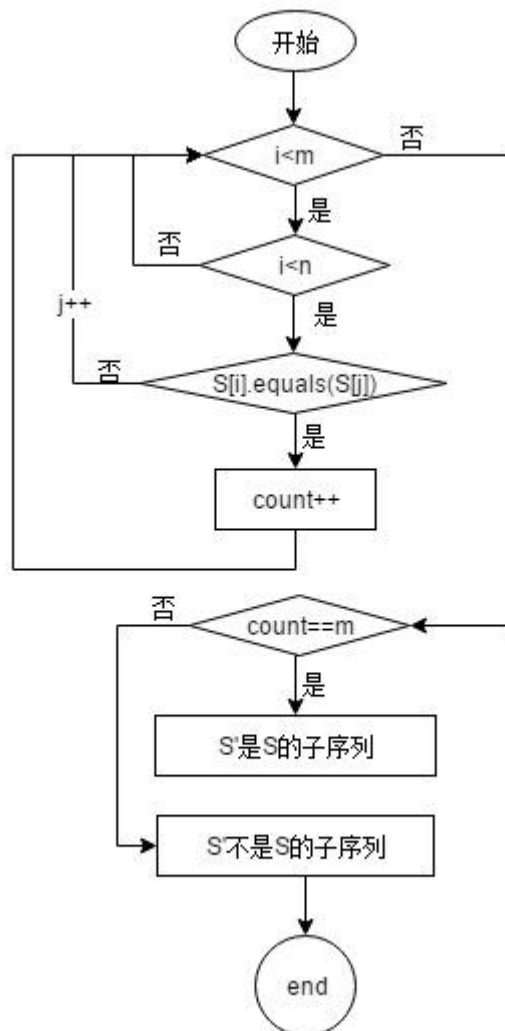
母序列和子序列同时遍历，记录  $count$ ，最后判断  $count$  是否与母序列的字符串个数相同。相同则  $S'$  是一个子序列，否则不是子序列。

```

for(int i=0;i<S.length;i++)
    for(;j<S_.length;) {
        if(S[i].equals(S[j])) {
            count++;
            break;
        }
        j++;
    }
if(count==S.length)
    System.out.println("s'是s的子序列");
else {
    System.out.println("s'不是s的子序列");
}

```

## • 设计图



## • 程序

```
package fifth_lab;
```

```

public class subSequence {
    public static int count=0;
    public static void main(String[] args) {
        String S_[] = {"买 Yahoo 股票","买 eBay 股票","买 Yahoo 股票","买 Oracle 股票"};
        String S[] = {"买 Yahoo 股票","买 Oracle 股票"};
        int j=0;

        for(int i=0;i<S.length;i++)
            for(;j<S_.length;) {
                if(S[i].equals(S[j])) {
                    count++;
                    break;
                }
                j++;
            }
        if(count==S.length)
            System.out.println("S'是 S 的子序列");
        else {
            System.out.println("S'不是 S 的子序列");
        }
    }
}

```

## ● 调试心得

通过本次实验，我了解了子序列问题的动态规划法求法，此问题相比背包问题和区间调度问题较为简单，不需要过于复杂的设计。