北京郵電大學

Beijing University of Posts and Telecommunications

《算法设计与实践》
课程实验报告

学　　院 计算机学院

姓　　名 曾世茂　顾天阳

学　　号 2019211532　2019211539

班　　级 2019211308

贡　　献 50%　50%

最长公共子序列

一、 实验内容

1、利用附件 1 给出的字符串 A、B、C、D，分别找出下列两两字符串间的最长公共子序列，并输出结果

A-B 、 C-D 、 A-D、 C-B

2、利用最长公共子序列求解下列最长递减子序列问题：

给定由 n 个整数 a_1, a_2, …, a_n 构成的序列，在这个序列中随意删除一些元素后可得到一个子序列 a_i, a_j, a_k, …, a_m, 其中 1≤i≤m≤n，并且 a_i≥a_j≥a_k≥…≥a_m, 则称序列 a_i, a_j, a_k, …, a_m 为原序列的一个递减子序列，长度最长的递减子序列即为原序列的最长递减子序列。利用“附件 2. 最大子段和输入数据-序列 1”、“附件 2. 最大子段和输入数据-序列 2”，求这两个序列中的最长递减子序列

二、算法实现

1、设计思路

首先证明原问题具有最有子结构的性质。根据两个序列最后一位的情况，原问题可以分为以下几种情形：

1. X(m)与 Y(n)最后一位相同的情况：则表明最后一位应该在最长公共子序列中，所以原问题的最长公共子序列可以归结为 X(m-1)与 Y(n-1)的最长公共子序列加上 Xm

2. 两个序列最后一位不同的情况：则表明其中至少有一位不在最长公共子序列中。则原问题的最长公共子序列可以归结为 X(m-1)与 Y(n) 或 X(m)与 Y(n-1)的两个最长公共子序列中的长度更长者

3. 特别的，当 X 或 Y 长度为 0 时，X 与 Y 的最长公共子序列长度为 0;

由以上证明的最优子结构性质,可以设计一个 dp 数组,dp[i][j]表示 X(i)与 Y(j)的最长公共子序列的长度。

dp[i][j]的递归关系如下：

$$\mathrm{dp[i][j]} = \begin{cases} 0,\ i = 0\ or\ j = 0 \\ dp[i-1][j-1] + 1,\ x(i) = y(j) \\ max(dp[i][j-1], dp[i-1][j]),\ x(i) != y(j) \end{cases}$$

为了根据最后能推断出两个序列的最长公共子序列，还需添加一个 action 数组，记录 dp 递推时选择了哪一个递推关系。后续根据 action 数组可以推断出最长公共子序列。

2、程序实现

（1） 数据结构

```
1.  int **dp = new int *[x.length() + 1];
2.  int **action = new int *[x.length() + 1];
3.  for (int i = 0; i <= x.length(); i++)
```

```
{
    dp[i] = new int[y.length() + 1];
    action[i] = new int[y.length() + 1];
}
```

（2） 核心算法

① 利用递推关系求 dp 数组以及 action 数组

```
for (int i = 0; i <= x.length(); i++)
{
    for (int j = 0; j <= y.length(); j++)
    {

        if (i == 0 || j == 0)
        {
            dp[i][j] = 0;
            action[i][j] = 1;
        }
        else
        {
            if (x[i - 1] == y[j - 1])
            {
                dp[i][j] = dp[i - 1][j - 1] + 1;
                action[i][j] = 1;
            }
            else if (dp[i - 1][j] >= dp[i][j - 1])
            {
                dp[i][j] = dp[i - 1][j];
                action[i][j] = 2;
            }
            else
            {
                dp[i][j] = dp[i][j - 1];
                action[i][j] = 3;
            }
        }
    }
}
```

② 利用 action 数组推断出最长公共子序列

```
int i = x.length() - 1;
int j = y.length() - 1;
string res = "";
while (i >= 0 && j >= 0)
{
```

```
        if (action[i + 1][j + 1] == 1)
        {
            res += x[i];
            i--;
            j--;
        }
        else if (action[i + 1][j + 1] == 2)
        {
            i--;
        }
        else if (action[i + 1][j + 1] == 3)
        {
            j--;
        }
    }
    reverse(res.begin(), res.end());
    cout << res << endl;
```

3、运行结果分析

A-B 的最长公共子序列：

```
3741a169n084+932a9681840g035o126r160i539t644h812m737+375i919s815+210a465n302y234+360w485e02411741698d312e454f02
5i006n494e376d677+651c373o160m418p173u900t034a542t144i134o107n610a6281715+277p841r630o097c046e000d407u018r082e4
39+439t714h726a096t284+739t123a143k178e039s852+378s958o919m728e300+478v048a3551260u637e480s348+252a562s569+581i
767n340p107u404t588+428a174n336d647+257p541r832o112d830u085c018e645s292+782s463o834m765e312+358v268a1491808u058
e434s632+134a726s023+317o512u829t311p602u526t829627007156527547003539349050502050257951314758377499746144882015
9492726382418921359620098912229671250429570695693782427742773730909157342588582838794752465921014615482215549712
2479338193618289487105808564763154017047233215666695224509245643049733450843369173923255018200425418297776291475
5671183277874463942273549539106609606561154500446757298189273897879096379223487329984826615837517215450462541857
2974286527180333254951214699929767844857466793016576251068550814957631337704065556642009127525671711370415746051
0603081909420648675368792652070a1b2c3d4e5f6g7h8i9j
&(!#a%$(n@*%+!*&a)*$1%&&g))*o)($r8@$i%#!t)$$h^#$m@!!+@&!i$%!s!#(+!)&a)(%n(()y*)@+%()w^%&e$!!1%@&l*@(d@*)e!(*f#)
@i%)^n*%*e^@(d^*%+&^)c@*#o()(m!(^p##)u&%#t#(^a*&%t@%$i%!@o@*%n^@&a$%)l@($+*%^p##^r@&%o%!$c#^!e%!(d!)!u%$^r%^%e$
@)+@$^t^&&h&!*a!!%t$^*+&^!t#$@a(^!k)(#e$(#s#!(+!#*s%!)o!^%m#!(e$*^+#($v*!*a)#%l#$#u#)#e&%&s%&$+#^(a@$(s&**+(!@i
^$)n$@)p!!*u%%$t&#@+%*!a#%%n!^#d@$$+!*)p@$%r$$@o*!%d#&(u@&&c!#*e^^%s&$%+@!%s&!^o@(&m&(&e$!*+$!&v%)!a#%^l#(&u&%$
e$@&s)@*+^(*a^!^s)!!+#%#o^%%u&($t%&%p#()u&^#t&^!^@@($*@$(&)&(*!#%^@!^^$*&!^*!!))##$*$)#$!^$@$!^&$#(#%(*^@!@!*)^
!&$*@$%$%%!(&&^@)&*))&^&&@&)#(^*$&#^$#%^%#^&@&*#&!(#*%#**$@#)^(*()@(^&!@&(#^#@%$&)^^($!(($(!#^)&!%#@!&%&@#%!!%(
^^&!!#(%*$##&()@!(!*#%*^^!*!*^&*&()#%^*&(&!^@$(@^)@@$$%!$$*$$@)*#@($(%(*$$##)$!@!)&@@!$!8@&&)()(^&&&&%((!@$%^!@
^(!&$@$@^$$#$$!^@))##($*@$&@^$^$@*^^^!%@$&!$)^$)*#%^!@(@$%*#&&&(^(**)$^)(!)&$^!*$&$@%(^**$%&@%#^@@^#(!(%^))*@@#
##*@^@&*$(#&)%(%!^&!*)^%!@^)%&!$@$$#!)&!!&*!!%)@@8@(*@(^!$&$#!))))!!^^%@%(**)$*^!!*($!*!)!#^@%*8$$%!%)#$!($()%(%
)!#!@#$%^&*()_+\
最长公共子序列为：
an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+input+and+produces+some+value
s+as+output
最长公共子序列的长度为：
122
```

A-D 的最长公共子序列：

C-D 的最长公共子序列：

C-B 的最长公共子序列：

最长公共子序列为：
an+algorithm+is+any+welldefined+computational+procedure+that+takes+some+values+as+input+and+produces+some+value
s+as+output
最长公共子序列的长度为：
122

运行结果分析： 该算法在四种输入情况下均能正常工作，正确输出两个字符串的最长公共子序列

三、利用最长公共子序列算法结局最长递减子序列问题

1、实验内容

给定由 n 个整数 a_1, a_2, …, a_n 构成的序列，在这个序列中随意删除一些元素后可得到一个子序列 a_i, a_j, a_k, …, a_m, 其中 1≤i≤m≤n, 并且 a_i≥a_j≥a_k≥…≥a_m, 则称序列 a_i, a_j, a_k, …, a_m 为原序列的一个递减子序列, 长度最长的递减子序列即为原序列的最长递减子序列。

例如，序列{1，7，2，3，6，5}，它的一个最长递减子序列为{7，6，5}。利用“附件 2. 最大子段和输入数据-序列 1”、“附件 2. 最大子段和输入数据-序列 2”，求这两个序列中的最长递减子序列

2、设计思路

一个序列的最长递减子序列即为该序列与其按照递减顺序排序后的的序列的最长公共子序列。

证明如下：

设原数列为 X(m)：$x_1x_2x_3...x_m$ ，Y(m)为 X(m)按照递减顺序排序后的数列 $y_1y_2y_3...y_m$ ，Z(n)为 X(m)的最长递减子序列。则需证明 Z(n)为 X(m)的子序列，Z(n)为 Y(m)的子序列。且 Z(n)是最长的。

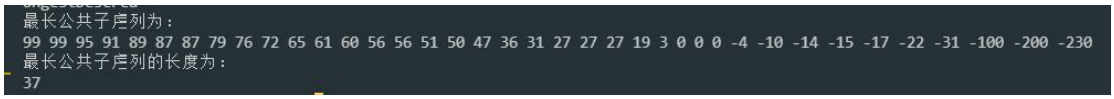
1. Z(n)是 X(m)的最长递减子序列，故自然为 X(m)的子序列。Z(n)为 Y(m)的子序列。

2. 假设存在一个长度大于 Z(n)的公共子序列 S(n)，由于 S(k)为 Y(m)的子序列，故 S(n)为递减子序列。则 Z(n)就不是 X(m)的最长递减子序列，与假设矛盾，故不存在这样的 S(n), Z(n)即为最长的公共子序列。

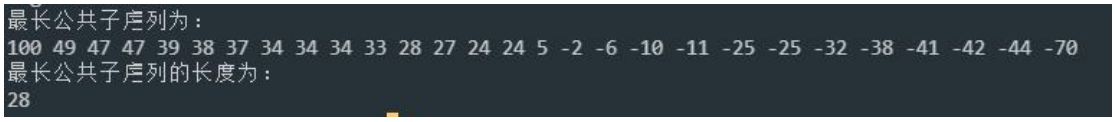
故只需找出 X(m)与 Y(m)的最长公共子序列，即为所求。

3、结果分析

序列 1 的最长递减子序列为：



序列 2 的最长递减子序列为：



通过分析可知，该算法可以正确运行

四、算法复杂度分析

时间复杂度：计算 dp 数组中每一个数的时间为 O(1)，dp 数组的大小为 m*n，故整体时间复杂度为 O(mn)

空间复杂度：dp 数组与 action 数组的大小均为(m*n),故整体空间复杂度为 O(mn)

五、算法改进与优化

如果仅需要计算最长公共子序列的长度的话，不仅可以省略 action 数组，在计算 c[i][j]时，只用到了第 i 行和第 i-1 行，因此用两行的数组空间就可以计算出最长公共子序列的长度。

而在求解最长递减子序列时，若不使用最长公共子序列的方式求解且只需要求出最长子序列长度,可将 dp[i]定义为长度为 i 的递减子序列的结尾最大数字，并配合二分搜索，则可在 O(nlogn)时间复杂度，O(n)空间复杂度内解决。

最大子段和

一、实验内容

1. 针对“附件 2. 最大子段和输入数据-序列 1”“附件 2. 最大子段和输入数据-序列 2”中给出的序列 1、序列 2，分别计算其最大子段和

序列 1：长度在 300-400 之间，由(-100,100)内的数字组成

序列 2：长度在 100-200 之间，由(-50,50)内的数字组成

要求

1. 指出最大子段和在原序列中的位置
2. 给出最大子段和具体值

二、算法实现

1、 设计思路

首先证明原问题具有最有子结构的性质。设 dp[i]为 X(m)中 X1 ~ Xi ，以 Xi 结尾的的最大字段和。则若 dp[i-1]小于小于等于 0 ， dp[i]即为 num[i]，若 dp[i-1]大于 0，dp[i]则为 dp[i-1]+num[i]。由此得到 dp[i]的递推公式为：

$$dp[i]=\begin{cases} num[i] & ,dp[i-1] \le 0 \\ dp[i-1]+num[i] & ,dp[i-1] > 0 \end{cases}$$

则整体的最长字段和的长度则为 max(dp[i]),其中 0≤i<n

为了最终能得出最长的递减子序列，还需要创建一个索引 startIndex ，startIndex[i]用于记录以第 i 位结尾的递减子序列的起始元素的下标。每次 dp 递推时，若递推动作为 dp[i] = num[i] ，则记录 startIndex[i] = i,表示该递减子序列只有一个元素。 若递推动作为 dp[i] = dp[i − 1] + num[i]，则记录 startIndex[i] = startIndex[i-1]，表示继承前一个递减子序列的起始下标。

2、 程序实现

（1） 数据结构

```
1.  int *dp = new int[array.size()];
2.  int *startIndex = new int[array.size()];
3.  dp[0] = array.at(0);
4.  startIndex[0] = 0;
```

（2） 算法实现

① 利用递推关系求 dp 数组以及 startIndex 数组

```
1.  Int max = 0;
2.  for (int i = 1; i < array.size(); i++)
3.  {
4.      if (dp[i - 1] >= 0)
5.      {
6.          dp[i] = dp[i - 1] + array.at(i);
7.          startIndex[i] = startIndex[i - 1];
8.      }
9.      else
10.     {
11.         dp[i] = array.at(i);
12.         startIndex[i] = i;
13.     }
14.     if (dp[i] > max)
15.     {
16.        Max = dp[i];
17.         maxIndex = i;  //记录最长字段和序列的末尾元素的下标
18.     }
19. }
```

最大字段和为：max

最大字段和序列的区间为：(startIndex[maxIndex] , maxIndex);

3、 运行结果分析

1. 序列 1

得到的最大字段和及其区间为：（下标从 0 开始）

2. 序列 2

得到的最长字段和及其区间为：（下标从 0 开始）

4. 算法复杂度分析：

时间复杂度：计算 dp 数组中每一个数的时间为 O(1)，dp 数组的大小为 n，故整体时间复杂度为 O(n)

空间复杂度：dp 数组与 startIndex 数组的大小均为(n),故整体空间复杂度为 O(n)

凸多边形最优三角剖分

一、实验内容

1、根据给出的 TD-LTE 网络配置数据，选取全部基站 eNodeB，并以这些基站作为平面点，构造平面点集的凸包，得到具有 21（29）个顶点的凸 21（29）多边形；
2、分别利用教科书上 $O(n^3)$ 算法、近似 $O(n^2)$ 的算法，实现凸多边形的最优、次优三角剖分，同时计算最优、次优三角剖分对应的目标值——边长弦长总和。

二、算法实现

1、设计思路

由矩阵连乘积有关的知识，我们发现表达式语法树和三角剖分之间的对应关系，其中，根结点对应多边形的一条边，非叶子结点对应多边形的一条弦，叶子结点对应多边形的一条边，因此我们可以类比矩阵连乘积的最优完全加括号方式进行计算。

下面证明凸多边形的最优三角剖分具有最优子结构性质。凸多边形$P=\{v_0,v_1,\dots,v_n\}$的最优三角剖分 T 包含三角形 $v_0v_kv_n$，则 T 的权为三角形 $v_0v_kv_n$ 的权、子多边形$\{v_0,v_1,\dots,v_k\}$和$\{v_k,v_{k+1},\dots,v_n\}$的权之和。可以断言，由 T 确定的这两个子多边形的三角剖分也是最优的。因为若有更小权的子多边形的三角剖分，将导致 T 不是最优三角剖分的矛盾。由此，凸多边形的最优三角剖分问题可以采用动态规划解决。

首先，定义$t[i][j]$为凸子多边形$\{v_{i-1},v_i,\dots,v_j\}$的最优三角剖分对应的权函数值，即其最优值。其中，设退化的多边形$\{v_{i-1},v_i\}$具有权值 0。据此，要计算的 $n+1$变形 P 的最优权值为$t[1][n]$。当$j-i\geq 1$时，凸子多边形$\{v_{i-1},v_i,\dots,v_j\}$至少有 3 个顶点。由最优子结构性质，$t[i][j]$的值应为$t[i][k]$的值加上$t[k+1][j]$的值，再加上三角形 $v_{i-1}v_kv_n$的权值。由于在计算时还不知道 k 的确切位置，而 k 的所有可能位置只有$j-i$个，因此可以在这$j-i$个位置中选出使$t[i][j]$达到最小的位置。由此，$t[i][j]$可定义为：

$$t[i][j]=\begin{cases}0 & i=j\\ min_{i\leq k\leq j}\{t[i][k]+t[k+1][j]+w(v_{i-1}v_kv_j)\} & i<j\end{cases}$$

2、程序实现

（1）全局变量及结构体

```
1.  //基站结构体
2.  struct eNodeB
3.  {
4.      int enodebid;
5.      double longitude, latitude;
```

```
    int id;
};

int idx;            //基站个数
double t[N][N][2]; //权函数，后面 2 分别对应最优值和次优值
int s[N][N][2];     //三角剖分中的断点信息，后面 2 分别对应最优值和次优值
eNodeB enodebs[N]; //基站数组
```

（2）函数定义

```
double w(int a, int b, int c);//求三角形的权值和
double dist(int a, int b);//求两个基站间的距离
double radian(double a);//将经纬度转化成弧度

void min_weight_triangulation(int n, double t[][N][2], int s[][N][2])
;//最优三角剖分 dp 函数
void find_breakpoints(int l, int r, int u);//递归寻找断点
```

（3）核心算法

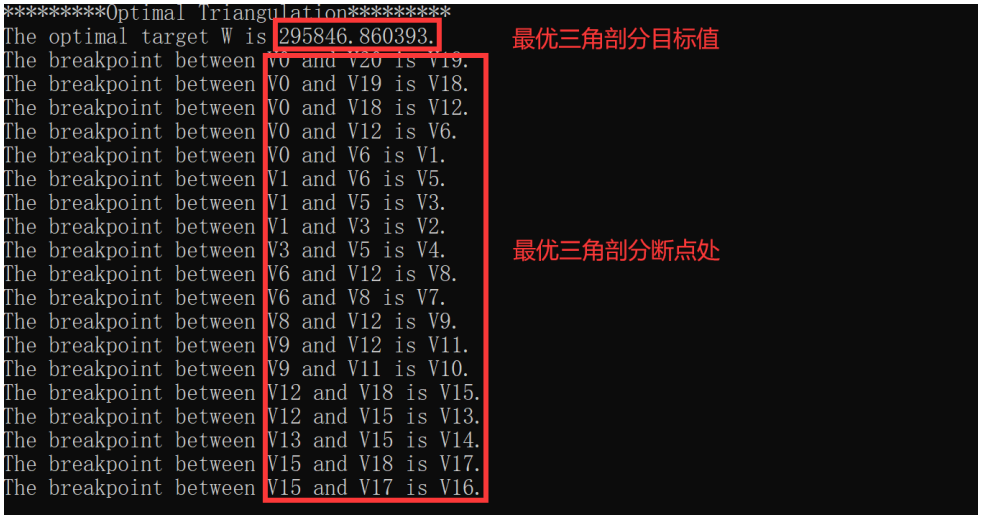
利用动态规划计算最优三角剖分为本问题的核心算法：

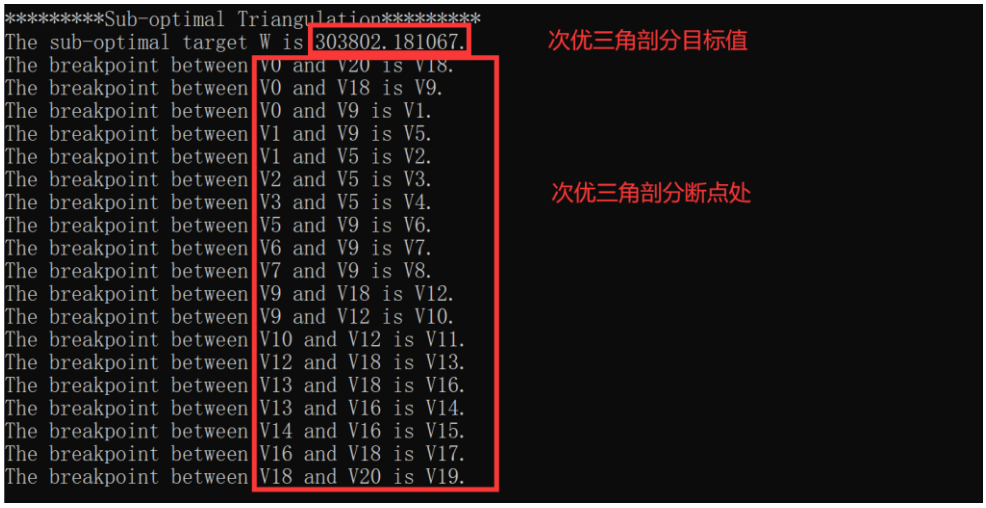
```
   void min_weight_triangulation(int n, double t[][N][2], int s[][N][
2])
{
    for (int i = 1; i <= n; i++)
        t[i][i][0] = t[i][i][1] = 0;
    for (int r = 2; r <= n; r++)
    {
        for (int i = 1; i <= n - r + 1; i++)
        {
            int j = i + r - 1;
            t[i][j][0] = t[i][j][1] = t[i + 1][j][0] + w(i - 1, i, j)
;
            s[i][j][0] = s[i][j][1] = i;
            for (int k = i + 1; k < i + r - 1; k++)
            {
                double u = t[i][k][0] + t[k + 1][j][0] + w(i - 1, k,
j);
                if (u < t[i][j][0])
                {
                    t[i][j][1] = t[i][j][0];
                    s[i][j][1] = s[i][j][0];
                    t[i][j][0] = u;
                    s[i][j][0] = k;
                }
            }
```

```
23.         }
24.     }
25. }
```

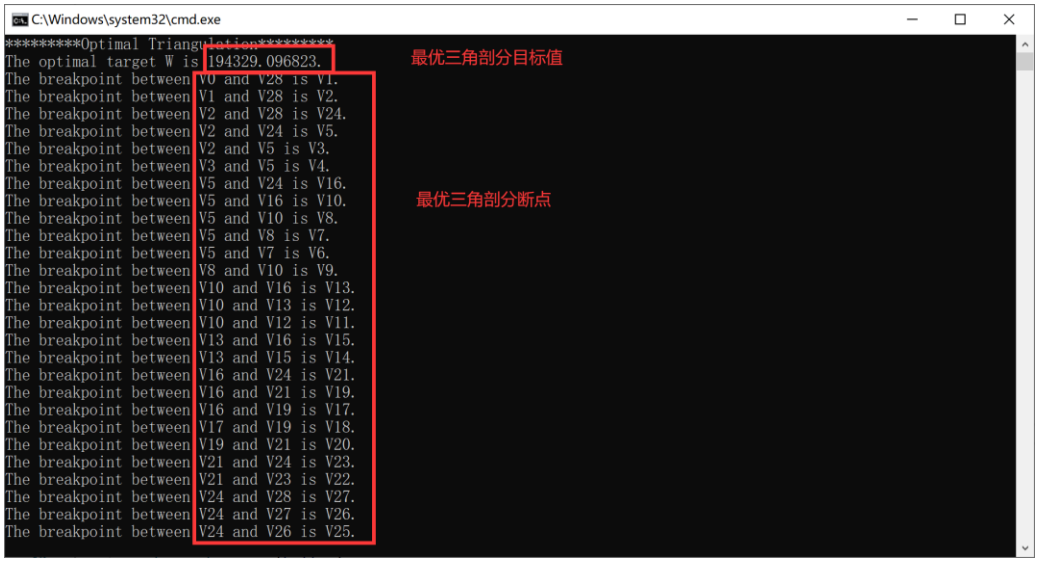
（4）结果输出

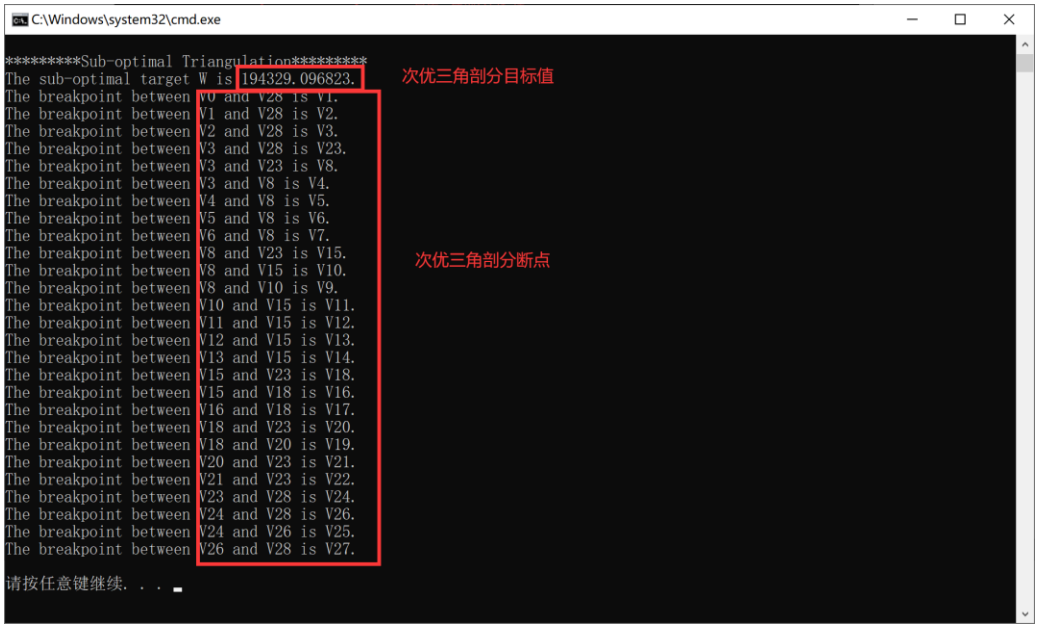
凸 21 边形：





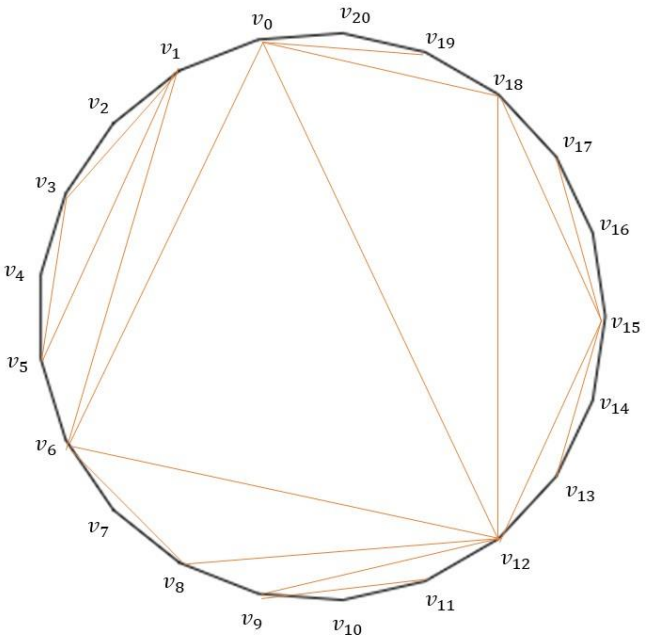
凸 29 边形：



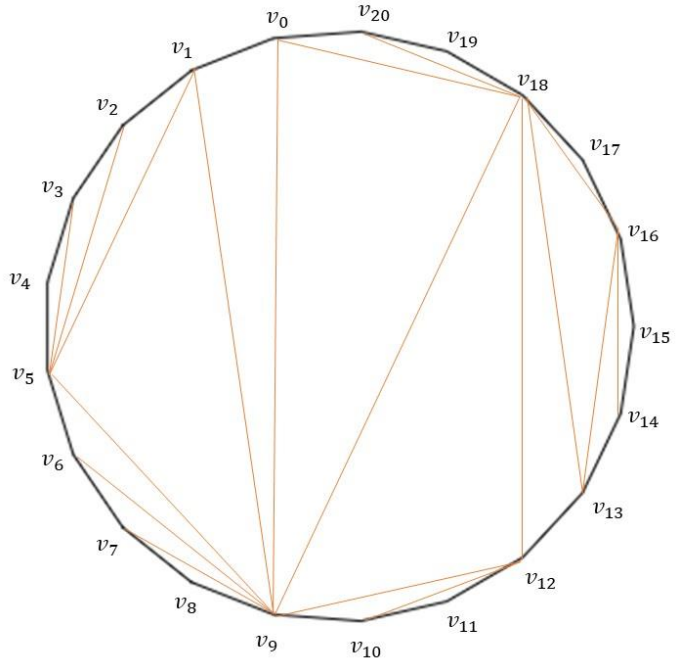


（5）绘制结果

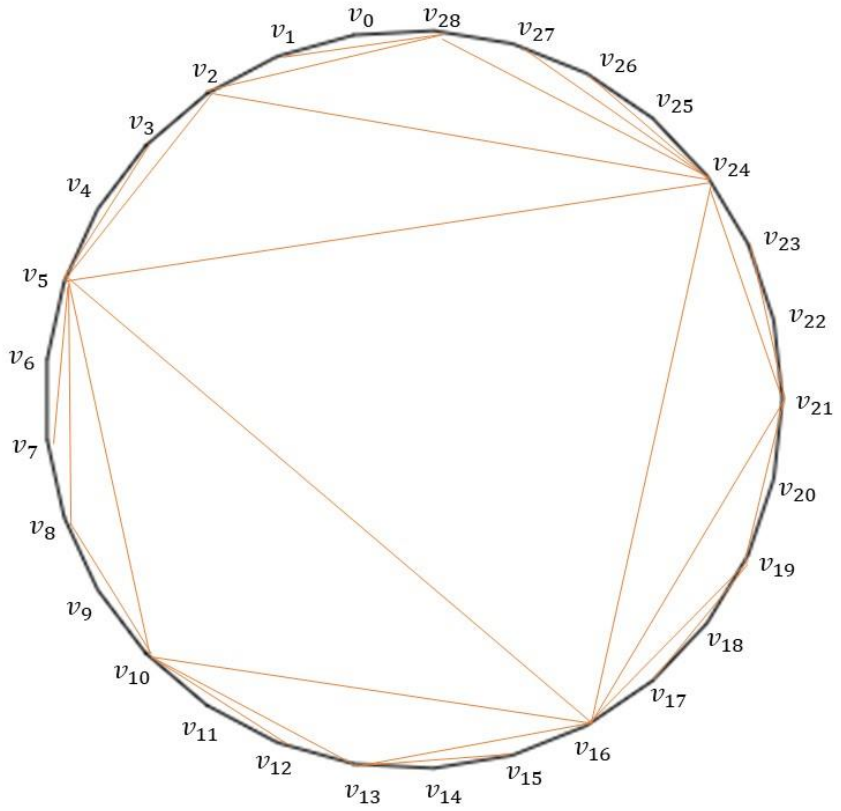
凸 21 边形最优三角剖分绘制：



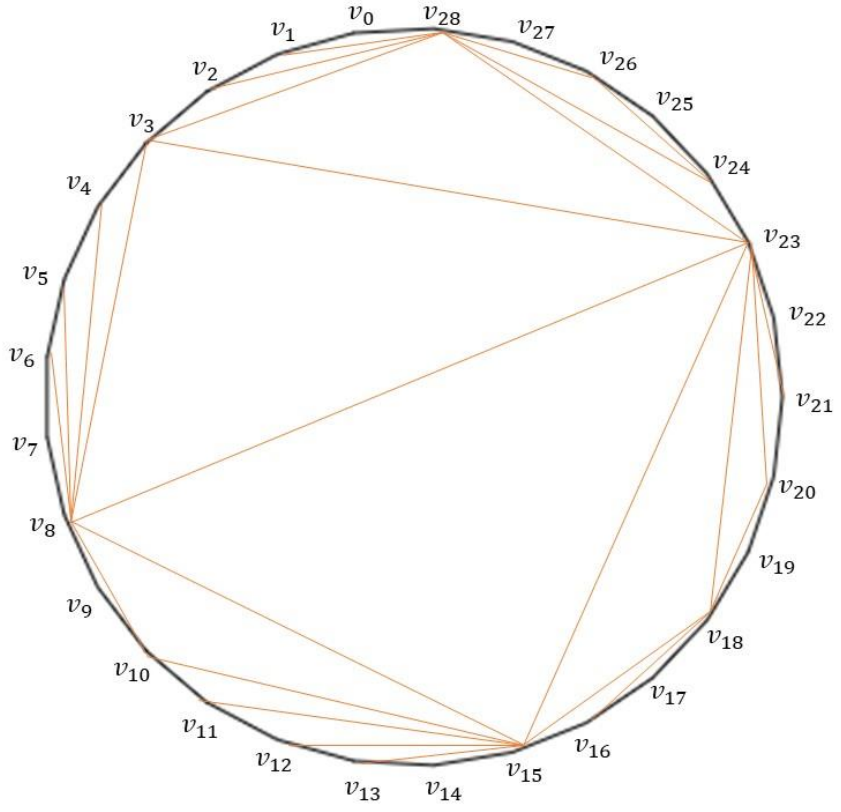
凸 21 边形次优三角剖分绘制：



凸 29 边形最优三角剖分绘制：


v_0
v_{28}
v_1
v_{27}
v_2
v_{26}
v_3
v_{25}
v_4
v_{24}
v_5
v_{23}
v_6
v_{22}
v_7
v_{21}
v_8
v_{20}
v_9
v_{19}
v_{10}
v_{18}
v_{11}
v_{17}
v_{12}
v_{16}
v_{13}
v_{15}
v_{14}

凸 29 边形次优绘制：


v_0
v_{28}
v_1
v_{27}
v_2
v_{26}
v_3
v_{25}
v_4
v_{24}
v_5
v_{23}
v_6
v_{22}
v_7
v_{21}
v_8
v_{20}
v_9
v_{19}
v_{10}
v_{18}
v_{11}
v_{17}
v_{12}
v_{16}
v_{13}
v_{15}
v_{14}

（6）时空复杂度分析

该算法内部有 3 层循环，故时间复杂度为$O(n^3)$；定义了 3 纬数组，但第 3 纬仅有 2 个空间，故空间复杂度为$O(n^2)$。

三、算法改进

考虑上述算法时间复杂度为$O(n^3)$，当 n 较大时，算法运行时间较长，为此我们提出一种启发式的三角剖分算法，可以将时间复杂度降至$O(n^2)$附近。

所谓启发式算法，它表示一个基于直观或经验构造的算法，在可接受的花费（指计算时间和空间）下给出待解决组合优化问题每一个实例的一个可行解，该可行解与最优解的偏离程度一般不能被预计。考虑该算法核心部分有 3 层循环，可将第 3 层循环进行优化，相比搜索和比较每一个v_k，这里我们采用随机选取 5 个剖分点v_k进行最优比较，这样便避免了最内层循环，将算法时间复杂度降为$O(n^2)$附近。

1、程序实现

（1）结构体和全局变量

```
//基站结构体
struct eNodeB
{
    int enodebid;                //基站 id
    double longitude, latitude; //基站经纬度
    int id;                      //逆时针序号
};

int idx;           //基站数量
double t[N][N];    //最优三角剖分的目标值
int s[N][N];       //最优三角剖分的剖分点信息
eNodeB enodebs[N]; //基站数组
```

（2）函数定义

```
double w(int a, int b, int c);//求三角形的权值和
double dist(int a, int b);//求两个基站间的距离
double radian(double a);//将经纬度转化成弧度

void min_weight_triangulation(int n, double t[][N], int s[][N]);//最优三角剖分 dp 函数
void find_breakpoints(int l, int r, int u);//递归寻找断点
```

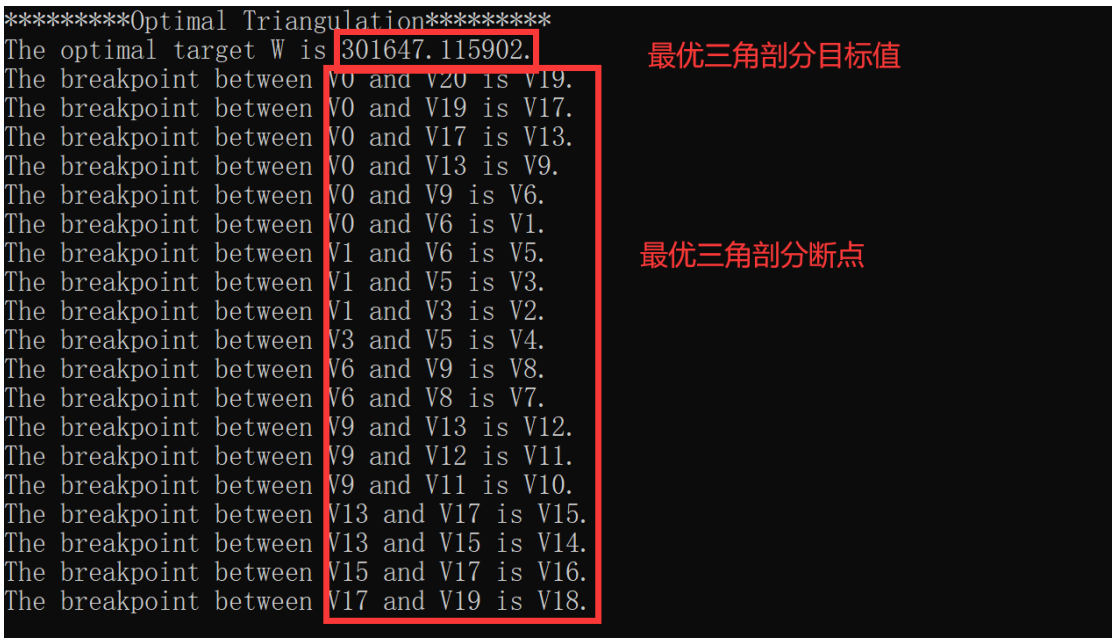
（3）核心算法

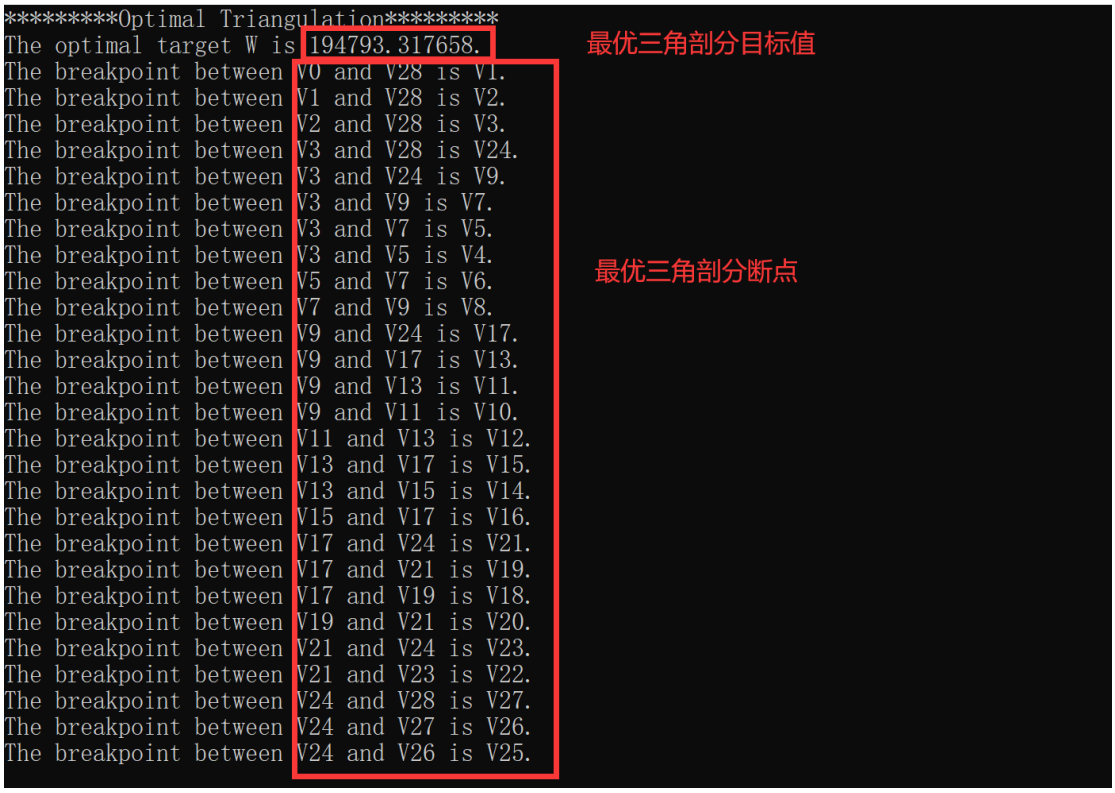
```
   void min_weight_triangulation(int n, double t[][N], int s[][N])
{
    for (int i = 1; i <= n; i++)
        t[i][i] = 0; //退化的多边形权值为 0

```

```
6.      //控制子问题规模 r=2,3,...n，自下而上，逐步求解
7.      for (int r = 2; r <= n; r++)
8.      {
9.          //子问题起点 i 循环，规模为 r 的多个子问题 t[i][j]求解
10.         for (int i = 1; i <= n - r + 1; i++)
11.         {
12.             int j = i + r - 1;
13.             //选取离起点最近的第一个断点 i
14.             t[i][j] = t[i + 1][j] + w(i - 1, i, j);
15.             s[i][j] = i;
16.             double u;
17.             //如果待比较断点数目不超过 5 个，则直接循环求解
18.             if (r - 1 <= 5)
19.             {
20.                 for (int k = i + 1; k < i + r - 1; k++)
21.                 {
22.                     u = t[i][k] + t[k + 1][j] + w(i - 1, k, j);
23.                     if (u < t[i][j])
24.                     {
25.                         t[i][j] = u;
26.                         s[i][j] = k;
27.                     }
28.                 }
29.             }
30.             //如果待比较断点数目超过 5 个，则利用随机数选取 5 个断点进行比
    较
31.             else
32.             {
33.                 for (int k = 0; k < 15; k++)
34.                 {
35.                     int rdm = i + rand() % (r - 1);
36.                     u = t[i][rdm] + t[rdm + 1][j] + w(i - 1, rdm, j);

37.                     if (u < t[i][j])
38.                     {
39.                         t[i][j] = u;
40.                         s[i][j] = rdm;
41.                     }
42.                 }
43.             }
44.         }
45.     }
46. }
```

（4）输出结果

对比原算法求得的结果 295846.860393 和 194329.096823，启发式算法求得的 301647.115902 和 194793.317658 与其基本一致，可以作为一个可行解。





（5）时空复杂度分析

该算法由于采用了启发式优化，避免了最内层循环，故时间复杂度为 $O(n^2)$；定义了 2 维数组，故空间复杂度也为 $O(n^2)$。

0-1 背包问题

一、实验要求

1、附件给出两组背包数据（背包容量、物品重量、物品价值），计算最优物品装载方案，第 1 组数据为 50 个物品，第 2 组数据为 100 个物品；
2、给出最优方案中，各个物品是否被放入以及物品放入后的背包总重量、总价值。

二、算法实现

1、设计思路

设 f[i][j]表示从前 i 个物品中进行选取，总重量不超过 j 的所有选法中的最大价值，对于每一个 f[i][j]，我们可以分为选择第 i 个物品和不选择第 i 个物品两种情况，而由 f[i][j]所表示的含义可知，其为选择第 i 物品和不选择第 i 个物品中的较大者，即：

$$f[i][j] = \max(f[i-1][j], f[i-1][j-v[i]+w[i]])$$

进行一个完整的自下而上的迭代过程后，我们便可以得到最优物品装载价值。对于各个物品是否被放入，我们采用自上而下的方法，在同等重量下，如果在前 i 个物品中进行选取的最大价值等于在前 i-1 个物品中进行选取的最大价值，则表明整体的最优方案不选择第 i 个物品，否则选择第 i 个物品并从当前重量中减去第 i 个物品的重量，然后进行下一个物品的选择判断。

2、程序实现

（1）全局变量定义

```
int n, m;        // n 表示物品个数，m 表示背包重量
int v[N], w[N]; // v 数组保存物品的重量，w 数组保存物品的价值
int f[N][N];     // f[i][j]表示从前 i 个物品中选取总重量不超过 j 的所有选法中的最大价值
int weight;      //最优选取方案的物品总重量
bool flag[N];    //存放最优选取方案中每个物品是否被放入
```

（2）核心算法

```
//从第 i 个物品，总重量不超过 j 开始遍历
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
    {
        //首先假设不选取第 i 个物品
        f[i][j] = f[i - 1][j];
        //若第 i 个物品的重量不超过当前剩余重量，则考虑选取第 i 个物品的情况（取较大者）
        if (j >= v[i])
            f[i][j] = max(f[i][j], f[i - 1][j - v[i]] + w[i]);
```

```
    }

int c = m;
//判断最优选取方案中各个物品是否被选取
for (int i = n; i > 0; i--)
{
    //若选取第 i 个物品的最大价值等于不选取第 i 个物品的最大价值，则表示第 i 个物品未选取
    if (f[i][c] == f[i - 1][c])
        flag[i] = false;
    //否则选取第 i 个物品，标志为 true
    else
    {
        flag[i] = true;
        c -= v[i];
        weight += v[i];
    }
}
```

（3）程序输出

第 1 组数据：

```
Goods 1 is put.
Goods 2 is put.
Goods 3 is not put.
Goods 4 is put.
Goods 5 is not put.
Goods 6 is not put.
Goods 7 is not put.
Goods 8 is put.
Goods 9 is put.
Goods 10 is not put.
Goods 11 is put.
Goods 12 is not put.
Goods 13 is not put.
Goods 14 is not put.
Goods 15 is not put.
Goods 16 is not put.
Goods 17 is not put.
Goods 18 is put.
Goods 19 is not put.
Goods 20 is put.
Goods 21 is put.
Goods 22 is not put.
Goods 23 is put.
```

```
Goods 24 is put.
Goods 25 is put.
Goods 26 is put.
Goods 27 is not put.
Goods 28 is not put.
Goods 29 is not put.
Goods 30 is not put.
Goods 31 is not put.
Goods 32 is put.
Goods 33 is put.
Goods 34 is not put.
Goods 35 is not put.
Goods 36 is not put.
Goods 37 is not put.
Goods 38 is put.
Goods 39 is not put.
Goods 40 is not put.
Goods 41 is not put.
Goods 42 is not put.
Goods 43 is put.
Goods 44 is put.
Goods 45 is put.
Goods 46 is not put.
Goods 47 is not put.
Goods 48 is not put.
Goods 49 is put.
Goods 50 is put.

The total weight is 340.
The total value is 1173.
```

第 2 组数据：

```
Goods 1 is put.
Goods 2 is not put.
Goods 3 is not put.
Goods 4 is not put.
Goods 5 is put.
Goods 6 is put.
Goods 7 is not put.
Goods 8 is not put.
Goods 9 is put.
Goods 10 is not put.
Goods 11 is put.
```

```
Goods 12 is not put.
Goods 13 is not put.
Goods 14 is not put.
Goods 15 is not put.
Goods 16 is not put.
Goods 17 is not put.
Goods 18 is not put.
Goods 19 is not put.
Goods 20 is not put.
Goods 21 is not put.
Goods 22 is put.
Goods 23 is put.
Goods 24 is not put.
Goods 25 is not put.
Goods 26 is not put.
Goods 27 is put.
Goods 28 is not put.
Goods 29 is not put.
Goods 30 is not put.
Goods 31 is put.
Goods 32 is not put.
Goods 33 is not put.
Goods 34 is put.
Goods 35 is not put.
Goods 36 is not put.
Goods 37 is not put.
Goods 38 is not put.
Goods 39 is not put.
Goods 40 is not put.
Goods 41 is not put.
Goods 42 is put.
Goods 43 is not put.
Goods 44 is not put.
Goods 45 is not put.
Goods 46 is not put.
Goods 47 is not put.
Goods 48 is not put.
Goods 49 is put.
Goods 50 is not put.
Goods 51 is not put.
Goods 52 is put.
Goods 53 is not put.
Goods 54 is put.
Goods 55 is put.
```

```
Goods 56 is not put.
Goods 57 is not put.
Goods 58 is put.
Goods 59 is not put.
Goods 60 is put.
Goods 61 is put.
Goods 62 is not put.
Goods 63 is not put.
Goods 64 is not put.
Goods 65 is not put.
Goods 66 is not put.
Goods 67 is not put.
Goods 68 is not put.
Goods 69 is not put.
Goods 70 is put.
Goods 71 is not put.
Goods 72 is not put.
Goods 73 is not put.
Goods 74 is not put.
Goods 75 is not put.
Goods 76 is not put.
Goods 77 is put.
Goods 78 is not put.
Goods 79 is not put.
Goods 80 is put.
Goods 81 is put.
Goods 82 is put.
Goods 83 is put.
Goods 84 is put.
Goods 85 is not put.
Goods 86 is not put.
Goods 87 is not put.
Goods 88 is put.
Goods 89 is put.
Goods 90 is not put.
Goods 91 is not put.
Goods 92 is not put.
Goods 93 is not put.
Goods 94 is not put.
Goods 95 is put.
Goods 96 is not put.
Goods 97 is not put.
Goods 98 is not put.
Goods 99 is not put.
```

```
100.   Goods 100 is put.
101.
102.   The total weight is 649.
103.   The total value is 1661.
```

（4）时空复杂度分析

核心算法有两层循环，对于一个物品数量为 n、背包容量为 m 的问题，其时间复杂度为$O(nm)$；定义了一个 f 二维数组，空间复杂度为$O(nm)$。

三、算法改进

观察算法的两层循环，我们发现每次循环只计算第 i 层和第 i-1 层的 f 值，若考虑每次只记录当前层和当前层下一层的 f 信息，可将 f[n][m]缩减为 f[m]，大大减小了空间复杂度。

1、算法实现

（1）全局变量定义

```
1.  int n = 100, m = 650; // n 表示物品个数，m 表示背包重量
2.  int v[N], w[N];       // v 数组保存物品的重量，w 数组保存物品的价值
3.  int f[N];             // f[j]表示总重量不超过 j 的所有选法中的最大价值
4.  int weight;           //最优选取方案的物品总重量
```

（2）核心算法

```
1.  //从第 i 个物品，总重量不超过 j 开始遍历
2.  for (int i = 1; i <= n; i++)
3.      //这里 j 需要从背包容量开始遍历，防止第 i-1 层的 f 被第 i 层的 f 所覆盖
4.      for (int j = m; j >= v[i]; j--)
5.          f[j] = max(f[j], f[j - v[i]] + w[i]);
```

（3）结果输出

第 1 组数据：

```
The total value is 1173.
```

第 2 组数据：

```
The total value is 1661.
```

（4）时空复杂度分析

核心算法有两层循环，对于一个物品数量为 n、背包容量为 m 的问题，其时间复杂度为$O(nm)$；定义了一个 f 一维数组，空间复杂度为$O(m)$。