实验二

一、 实验题目：动态规划算法——编程实现矩阵链乘法

二、实验目的与内容

1、实验目的：

1) 了解动态规划算法的一般设计思路，学习如何构造状态转移方程和保存状态。
2) 了解在进行矩阵链乘法时，不同的运算方案（运算先后）所带来的性能差距。
3) 掌握矩阵链乘法问题的具体含义，理解其在实现最优解表时的运算方案和转移方程以及其推导过程。

2、实验内容：

1) 设计并编程实现拥有计算最小乘法次数的矩阵链乘法匹配方案：
   a) 输入：第一行输入一个数 N，代表要处理的矩阵个数；下面跟着 N 行，每一行两个数，为矩阵的行数和列数。（p[i].first，p[i].second）
   b) 输出：最小的乘法次数，匹配方案。
2) 自行设计测试用例测试数据，测试程序的正确性。
3) 分析算法的正确性与复杂度。

三、算法设计

1、算法描述

实验中我们维护两个动态规划中用于存储状态的表，一个表记为 opt[i][j]，用于存储从第 i 号矩阵到第 j 号矩阵所用的最小的乘法次数，另一个表记为 record[i][j]，存储使得从第 i 号元素到第 j 号元素产生最小乘法次数的断点的编号 k。设计状态方程和边界条件完成两个表的存储。递归查找 record 表就能够输出匹配方案。表项 opt[0][N-1]存入的是问题解。

下面具体讲解该算法该如何实现：

STEP 1: 初始化最优解矩阵 opt[N][N]和断点矩阵 record[N][N]。

最优解矩阵的初始化策略是，对于 i=0 到 i=N-1，有：

$$opt[i][j] = \begin{cases} 0 & if(i = j) \\ INF & if(i \neq j) \end{cases}$$

即对于最优解矩阵，初始化时，对于只有一个矩阵情况(i=j)，即不用计算，开销为 0，其余的要开销的情况，需要后期进行计算，先置位为无穷。

$$record[i][j] = \begin{cases} i & if(i = j) \\ INF & if(i \neq j) \end{cases}$$

对于断点矩阵，首先对于每个矩阵自己(i=j)，断点就是自己；其余情况不知道断点是什么，需要后期进行计算，先置位为无穷。

STEP 2: 构造最优解矩阵和断点矩阵。

这里我们要用到三重循环，每一重循环的意义如下：

1) 第一重：从 i=2 开始，到 i=N，i 代表当前处理的矩阵数链长。（由于对于一个矩阵的情况（i=1），上一步已经初始化为 0，这里便可以直接从 2 开始循环）

2) 第二重：从 j=0 开始，到 j=N-1，j 代表当前处理的矩阵链开始点的下标。这里由于链长为 i，我们可以计算到矩阵链结尾点的下标，记为 z，有 z=j+1-1。

3) 第三重，从 k=j 开始，到 k=z-1，k 代表断点的下标。

这样，对于最内层循环，我们实际上要处理的是，下标 j 开始到下标 z 这个长度为 i 的矩阵链的最优解，以及其对应的断点，如下公式所示：

$$opt[j][z] = \min_{k=j\ to\ z-1}(opt[j][k] + opt[k+1][z] + p[j] \times p[k+1] \times p[z+1])$$

同时我们要记录下断点:

$$record[j][z] = k \quad (for\ which\ k\ minimize\ opt[j][z])$$

所以，我们可以写出伪代码：

```
from i=2 to N{
    form j=0 to N-1{
        z=j+i-1;
        from k=j to z-1{
            temp_opt=opt[j][k]+opt[k+1][z]+p[j]*p[k+1]*p[z+1];
            if temp_opt<opt[j][z]{
                opt[j][z]=temp_opt;
                record[j][z]=k;
            }
        }
    }
```

STEP 3: 递归查询 record 函数，构造最优解的匹配方案函数 print。

具体地，我们函数三个参数为 record、i 和 j，其中 record 是我们前面所提到的断点矩阵，i 和 j 是我们目前在 record 矩阵中处理的串的左下标和右下标。

实现这个算法伪代码如下：

```
print(record[][], i, j){
    if(i=j){
        output(Ai);     //输出 Ai
    }
    else{
        Output('(');     //输出左括号
        print(record[][], i, record[i][j]);   //在左边递归
        print(record[][], record[i][j]+1, j);   //在右边递归
        output(')'); //输出右括号
    }
```

STEP　4: 输出最少的乘法步骤和匹配方案。

最小的乘法步骤储存在 opt[0][N-1]中，直接调用即可；输出匹配方案，直接调用函数 print(record, 0, N-1) 即可。

2、算法流程图

算法描述的流程图如图 3.1：

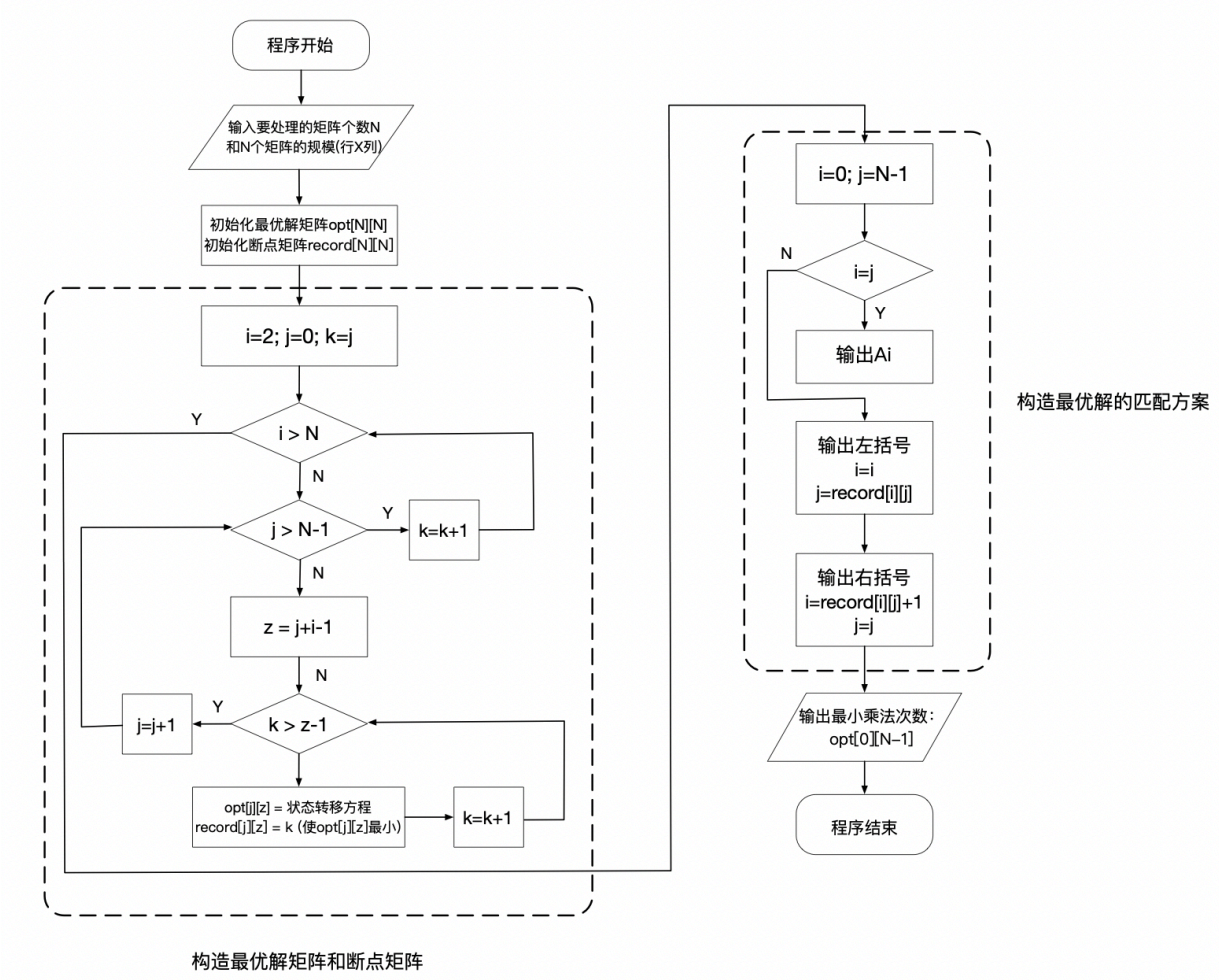


图 3.1 矩阵链乘法算法流程图

四、实验环境

操作系统：Windows 10
编译环境：Dev C++ 5.11
编译器:　TDM-GCC 4.9.2 64-bit Release

五、实验过程

通过构造最优解表 opt 和断点记录表 record 实现的动态规划算法的源代码如下所示，其中每一块代码的释义、变量名的解释在注释中均有体现。

动态规划实现矩阵链乘法

```
#include <iostream>
#include <utility>
#define INF 100000
using namespace std;

int N;
int opt[1000][1000], record[1000][1000]; //opt：最优解矩阵，record：用于记录括号位置
pair<int, int> p0[1000]; //记录每个矩阵的大小
int p[1000]; //存储矩阵大小序列

void opt_generate(int p[1000], int opt[1000][1000], int record[1000][1000], int N){
    for(int i=0;i<N;i++){
        opt[i][i]=0; //对角线开销置 0
    }
    for(int i=0;i<N;i++){
        record[i][i]=i; //每个元素开始时括号为自己（边界情况，链长为 1 的情况）
    }
    /*构造最优解矩阵*/
    for(int i=2;i<=N;i++){    //i 代表当前所处理的矩阵链的长度
        for(int j=0;j<=N-i;j++){ //j 代表当前链开始的横坐标
            int z=j+i-1; //z 代表当前链结束的横坐标
            opt[j][z]=INF; //初始值为无穷大
            for(int k=j;k<z;k++){ //选择断点 k，寻找最优解
                int opt_t=opt[j][k]+opt[k+1][z]+p[j]*p[k+1]*p[z+1];
                if(opt_t<opt[j][z]){
                    opt[j][z]=opt_t;
                    record[j][z]=k;     //保存断点 k
                }
            }
        }
    }
}

//打印出括号序列
void print(int record[1000][1000], int i,int j){
    if(i==j){
        cout<<"A"<<i;
    }
    else{
        cout<<"(";
        print(record,i,record[i][j]); //左边寻找
```

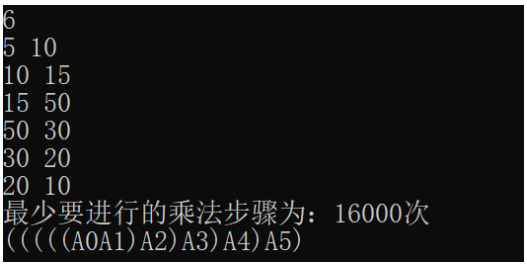
```
        print(record,record[i][j]+1,j); //右边寻找
        cout<<")";
    }
}


int main(){
    cin>>N;
    for(int i=0;i<N;i++){
        cin>>p0[i].first;
        cin>>p0[i].second;
    }

    //如果输入不合法，汇报错误
    for(int i=1;i<N;i++){
        if(p0[i].first!=p0[i-1].second){
            cout<<"This matrix can not multiply."<<endl;
            return 0;
        }
    }

    //生成矩阵大小序列
    for(int i=0;i<N;i++){
        p[i]=p0[i].first;
    }
    p[N]=p0[N-1].second;

    opt_generate(p,opt,record,N);

    cout<<"最少要进行的乘法步骤为："<<opt[0][N-1]<<"次"<<endl;
    print(record,0,N-1);
    return 0;
}
```

六、算法测试

为了保证算法的正确性，我们设置三组测试样例，设置意义如下（我们的输入输出已经在第二节实验内容中作出了约定）：

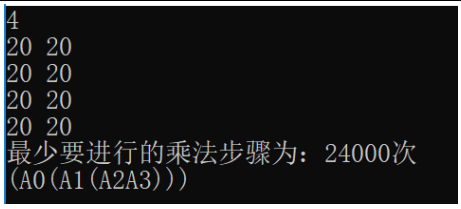
1) 第一组：普通算例测试，矩阵数 N=6，矩阵中没有相同的矩阵。
2) 第二组：边界条件测试，矩阵数 N=4，所有矩阵都相同。
3) 第三组：边界条件测试，矩阵数 N=6，但是矩阵链不满足能相乘的条件。（其中有相邻两个矩阵不满足矩阵乘法的定义）

测试样例 1

样例输入	理论输出	样例输出
6 5 10 10 15 15 50 50 30 30 20 20 10	16000 (((((A0A1)A2)A3)A4)A5)	

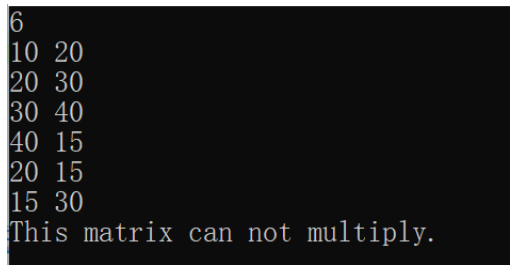
由于理论输出与样例输出相符，所以测试样例 1 验证成功。

测试样例 2

样例输入	理论输出	样例输出
4 20 20 20 20 20 20 20 20	24000 (A0(A1(A2A3)))	

由于理论输出与样例输出相符，所以测试样例 2 验证成功。

测试样例 3

样例输入	理论输出	样例输出
6 10 20 20 30 30 40 40 15 20 15 15 30	This matrix can not multiply.	

由于理论输出与样例输出相符，所以测试样例 3 验证成功。

综上，算法通过所有样例的测试。

七、结果分析

1. 算法正确性证明：

我们输出的乘法的最少次数是 opt[0][N-1]，这个数值是由状态转移方程得到的，我们要证明算法的正确性，就是要证明状态转移方程的正确性，而这个问题的状态转移方程如下：

$$opt[j][z] = \min_{k=j\ to\ z-1}(opt[j][k] + opt[k+1][z] + p[j] \times p[k+1] \times p[z+1])$$

用反证法证明该状态转移方程的正确性如下：

假设这个状态转移方程所求得的断点 k_{min}，求出的 $opt[j][z]$ 并不是最优解。

（即从下标 k 到下标 z 所用的最少的乘法次数）那么我们假设有一个点 k'_{min}，它能够求出 $opt'[j][z]$，使得$opt'[j][z] < opt[j][z]$。那么其实上我们有：

$$opt'[j][z] = (opt[j][\,k'_{min}] + opt[\,k'_{min}+1][z] + p[j] \times p[\,k'_{min}+1] \times p[z+1])$$
$$\geq \min_{k=j\ to\ z-1} (opt[j][k] + opt[k+1][z] + p[j] \times p[k+1] \times p[z+1])$$
$$= (opt[j][\,k_{min}] + opt[\,k_{min}+1][z] + p[j] \times p[\,k_{min}+1] \times p[z+1])$$
$$= opt[j][z]$$

即我们得到$opt'[j][z] \geq opt[j][z]$，这与我们前面假设的$opt'[j][z] < opt[j][z]$相违背，故我们有通过这个状态转移方程所求得的确实是从下标 j 到下标 z 的最小乘法次数，k 确实为该最小次数的断点。

2. 复杂度分析：

这个算法求了两个结果，分别是最小乘法次数和最优匹配方案。我们分开论述这两个部分的复杂度：

1) 求得最小乘法次数的复杂度：

可以看出，最小乘法次数即为 opt[0][N-1]，这个过程中我们维护了最优解表 opt[N][N]，我们运用了三重循环，分别遍历链的长度 i（从 2 到 N）、链的开始节点 j（从 0 到 N-i）、链的断点 k（从 j 到 j+i-1）。所以对于每一次循环调用了状态转移方程，其复杂度为 O(1)。

故我们获得最优解所用的复杂度为 $O(n^3)$。（三重循环，每一重都为 O(n)）

2) 打印最佳匹配情况的复杂度（在我们已经求得 record 数组情况下）：

我们采用递归的方式完成最优匹配情况的打印，在每一次调用中，我们分别左递归和右递归，求得最佳解（详情见前面的算法分析和源代码）。递归边界为 i=j，若这个条件成立，我们输出对应的字母和括号即可。即我们有复杂度为：

$$T_{最小乘法次数}(n) = 2\left(\frac{n}{2}\right) + O(1) = O(n)$$

3) 整体上来看算法复杂度：

综上，整个程序由求得最小乘法次数（同时求得 opt 和 record 数组）和打印匹配情况两部分组成，整个程序的复杂度为：

$$T_{总复杂度}(n) = T_{最小乘法次数}(n) + T_{打印匹配情况}(n) = O(n^3) + O(n) = O(n^3)$$

八、总结

本次实验中我用动态规划算法构造了状态转移方程，实现了最优解矩阵、断点矩阵，以及用递归调用断点矩阵打印出了最优匹配情况，了解到了动态规划问题的一般解决方法。同时，通过这个实验，了解到同样一个矩阵链的乘法，运算顺序的不同会产生很大的性能差异。。

实验中我觉得有一个不好处理的点是，以什么样的顺序去完成最优解表 opt，先开始我的想法是三层循环分别遍历链开始的点，链结束的点以及该链中的断点。但是后来我在实际操作的时候发现，这样做是不可取的，因为链长的情况需要依赖于链短的情况所求得的结果，所以我们只能以链长作为最外层循环。所以后来

我的三层循环分别变为链长、链开始的点、链中的断点，便解决了问题。

实验二（选做）

一、实验题目：动态规划算法——编程实现 0-1 背包问题求解算法

二、实验目的与内容

1、实验目的：

1) 掌握动态规划算法的设计思路，学习构造状态转移方程和动态规划表的方法。
2) 了解 0-1 背包问题的具体含义以及其状态转移方程和状态表的含义和推导过程。

2、实验内容：

1) 设计并编程实现 0-1 背包问题的算法：
   a) 输入：第一行输入两个数 n 和 M，n 代待选表物品的件数，M 代表背包容量；下面给出 2 行，一行给出每个背包的重量 w[i]，一行给出每个背包的价值 p[i]。
   b) 输出：要求求出背包物品所能装入的最大价值。
2) 自行设计测试用例测试数据，测试程序的正确性。
3) 分析算法的复杂度。

三、算法设计

1、算法描述

在实验中，我们构造一个表 KNAP[i][j]，表示前 i 件物品装入 j 容量背包的最大价值，通过状态转移方程和边界条件完成这张表。背包能装入的最大价值就存储在 KNAP[n][M]中。

STEP 1: 构造表 KNAP[n][n]的状态转移方程。

KNAP[i][j]表示前 i 件物品恰好装入容量为 j 的背包中获得的最大价值，其中有：$1 \le i \le n, 0 \le j \le M$。

对第 i 件物品有两种选择方式，放或者不放：

1) 如果放第 i 件物品，问题转化为前 i-1 件物品装入容量 j-w[i]的背包所能获得的最大价值。

2) 如果不放第 i 件物品，问题就是前 i-1 件物品装入容量 j 的背包所获得的最大价值。

即我们要求这两种情况的较大值，即状态转移方程如下：

$$KNAP[i][j]=\begin{cases}KNAP[i-1][j] & if\ j<w[i]\\ max(KNAP[i-1][j],KNAP[i-1][j-w[i]]+p[i]) & if\ j\geq w[i]\end{cases}$$

STEP　2: 初始化表 KNAP[n][n]。

这个问题的边界为：前 0 件商品的最大价值和一定为 0，所以我们只要满足一下初始化条件即可：

$$KNAP[0][j]=0\ \ for\ j=0,1,2\ldots,M$$

其余表项均会被状态转移方程填充，不用初始化。

STEP　3: 完成表 KNAP[n][n]

二重循环调用状态转移方程即可，i 从 1 循环到 n，j 从 0 循环到 M，调用 n *M 次状态转移方程即可。

STEP　4: 输出背包所能装入的最大价值。

即输出前 n 件物品装入 M 容量的背包所获得的最大价值，即 KNAP[n][M]。

2、算法流程图

算法描述的流程图如图 3.1：

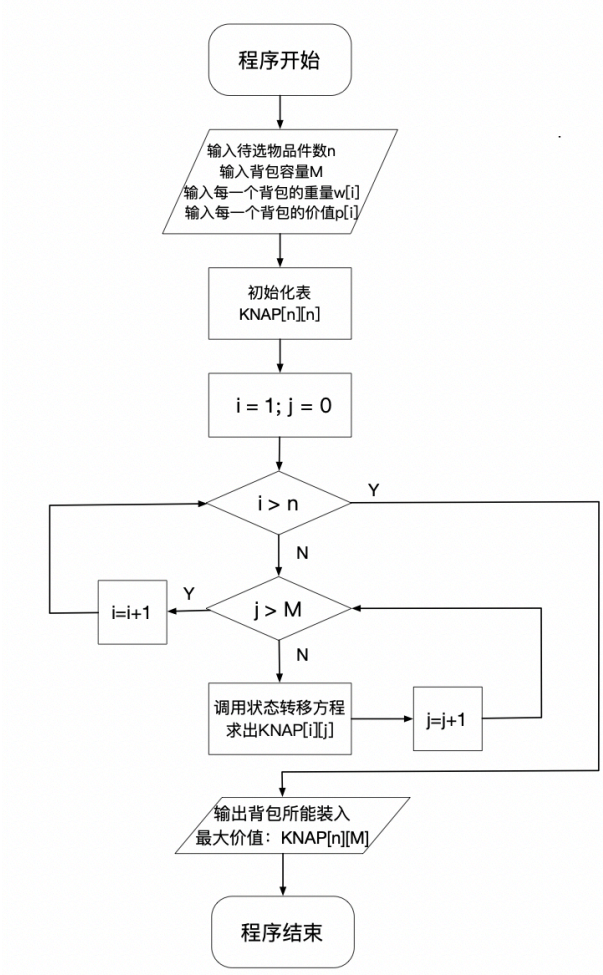


图 3.1 0-1 背包问题算法流程图

四、实验环境

操作系统：Windows 10
编译环境：Dev C++ 5.11
编译器　：TDM-GCC 4.9.2 64-bit Release

五、实验过程

利用动态规划实现 0-1 背包问题的源代码如下所示，其中每一部分的意义和变量名释义的已经写在了注释中：

动态规划实现 0-1 背包问题

```
#include <iostream>
#include <algorithm>
using namespace std;

int main(){
    int M; //背包总容量
    int n; //物品的件数
    cin>>n>>M;

    int w[n+5],p[n+5]; //每件物品的重量 w[i],价值 p[i]
    int KNAP[n+5][M+5]; //前 i 件物品装入 j 容量的最大价值

    for(int i=1;i<=n;i++){ //读入物品重量
        cin>>w[i];
    }
    for(int i=1;i<=n;i++){ //读入物品价值
        cin>>p[i];
    }

    for(int j=0;j<=M;j++){ //初始化边界条件，前 0 件商品最大价值为 0
        KNAP[0][j]=0;
    }

    for(int i=1;i<=n;i++){
        for(int j=0;j<=M;j++){
            if(w[i]>j){ //如果背包容量不够第 i 件商品，则一定不装入
                KNAP[i][j]=KNAP[i-1][j];
            }
            else{   //否则选择是否装第 i 件物品
            KNAP[i][j]=max(KNAP[i-1][j],KNAP[i-1][j-w[i]]+p[i]);
            }
        }
```

```
    }

    cout<<"背包能装入的最大价值为："<<KNAP[n][M];
    return 0;
}
```

六、算法测试

为了保证算法的正确性，我们设置三组测试样例，设置意义如下（我们的输入输出已经在第二节实验内容中做出了约定）：

1) 第一组：普通算例测试， 物品数量 n=5，背包容量 M=8，每个背包的价值和重量均大于 0 且不相等。
2) 第二组：边界条件测试，物品数量 n=4，背包容量 M=3，有价值为 0 的背包，背包的价值和重量均有重复。
3) 第三组：边界条件测试，物品数量 n=5，背包容量 M=3，有价值为 0 的背包和重量为 0 的背包，背包的价值和重量均有重复。

测试样例 1

样例输入	理论输出	样例输出
5 8 2 4 6 3 5 4 2 5 1 3	9	5 8 2 4 6 3 5 4 2 5 1 3 背包能装入的最大价值为：9

由于理论输出与样例输出相符，所以测试样例 1 验证成功。

测试样例 2

样例输入	理论输出	样例输出
4 3 2 1 1 2 1 0 0 1	1	4 3 2 1 1 2 1 0 0 1 背包能装入的最大价值为：1

由于理论输出与样例输出相符，所以测试样例 2 验证成功。

测试样例 3

样例输入	理论输出	样例输出
4 3 2 1 0 2 3 0 1 1 1 3	4	4 3 2 1 0 2 3 0 1 1 1 3 背包能装入的最大价值为：4

由于理论输出与样例输出相符，所以测试样例 3 验证成功。

综上，算法通过所有样例的测试。

七、结果分析

1. 复杂度分析：

求得背包最大价值的过程中，我们主要维护了一个表 KNAP[i][j]，代表前 i 件物品装入容量为 j 的背包中所获得的最大价值，最后输出 KNAP[n][M]即可。所以，复杂度为维护 KNAP 数组的复杂度。我们对于其有两重循环，第一重将物品编号 i 从 1 循环到 n，第二重将背包容量 j 从 1 循环到 M。在每一次循环中调用状态转移方程。而状态转移方程的复杂度为 O(1)，遍历 i 的复杂度为 O(n)，遍历 j 的复杂度为 O(M)。

故总复杂度为 O(nM)。

八、总结

通过这个实验，我深化了对背包问题的理解。实验中构造了状态转移方程，用该方程实现了存储矩阵 KNAP[][]，解决了问题。在这个过程中，进一步熟悉了动态规划算法的一般解决方法以及状态转移方程的构造。

实验中要注意的一个点是，在构造动态规划的状态转移方程时，如果当前的背包容量不能装下当前物品，即(j<w[i])，我们直接不放这个商品即可，等于 KNAP[i-1][j]。其余的情况才需要考虑放与不放当前物品。