背包问题实验报告

学号：2111033　　姓名：艾明旭　　专业：信息安全

01 背包是在 M 件物品取出若干件放在空间为 W 的背包里，每件物品的体积为 W1，W2 至 Wn，与之相对应的价值为 P1,P2 至 Pn。01 背包是背包问题中最简单的问题。01 背包的约束条件是给定几种物品，每种物品有且只有一个，并且有权值和体积两个属性。在 01 背包问题中，因为每种物品只有一个，对于每个物品只需要考虑选与不选两种情况。如果不选择将其放入背包中，则不需要处理。如果选择将其放入背包中，由于不清楚之前放入的物品占据了多大的空间，需要枚举将这个物品放入背包后可能占据背包空间的所有情况。

暴力枚举代码如下：

```
#include <iostream>
#include<Windows.h>
#include <algorithm>
using namespace std;
const int N = 100;
int a[N] = { 0 };
int temp[N] = { 0 };

void BagProblem(int len, int* w, int* v, int capacity)
{
    int i, j, k;
    int maxValue = 0;
    int Value = 0;
    int Weight = 0;
    int flag = 0;
    long long  num = 1l << len; // num 为 2 的 len 次方
    for (i = 0; i < num; i++)
    {
        j = i;
        k = 0;
        Value = 0;
        Weight = 0;
        flag = 0;
        while (j)
        {
            if (j & 1)
            {
                a[k] = 1;
                Value += v[k];
                Weight += w[k];
            }
```

```
            j >>= 1;
            k++;
        }
        if (Weight <= capacity && Value > maxValue)
        {
            maxValue = Value;
            flag = 1;
        }
        if (flag == 1)
        {
            for (int q = 0; q < len; q++)
            {
                temp[q] = a[q];
            }
        }
        for (int q = 0; q < len; q++)
        {
            a[q] = 0;
        }
    }
    for (int q = 0; q < len; q++)
    {
        //cout << temp[q] << " ";
    }
   // cout << endl;
    cout << maxValue << endl;
}

int main()
{
    DWORD start,end;
    start = GetTickCount();
    int n, c;
    cin >>c >> n;
    int w[N];
    int v[N];
    for (int i = 0; i < n; i++)
    {
        cin >> w[i]>>v[i];
    }
    BagProblem(n, w, v, c);
    end = GetTickCount()-start;
    cout <<endl<< end;
    return 0;
```

```
}
```

动态规划代码如下：

```
#include<iostream>
#include<math.h>
#include<Windows.h>
using namespace std;
#define N 1005
int dp[N];
int n, m, v[N], w[N];

int main()
{
    DWORD start, end;
    start = GetTickCount();
    cout << "请输入物品的数量和背包的重量" << endl;
    cin >> m >> n;
    cout << "请分别输入物品" << endl;
    for (int i = 1; i <= n; i++)
    {
        cin >> v[i] >> w[i];
        for (int j = m; j >= v[i]; j--)
        {
            dp[j] = max(dp[j], dp[j - v[i]] + w[i]);
        }
    }
    cout << dp[m] << endl;
    end = GetTickCount() - start;
    cout << end;
    return 0;
}
```

动态规划求解 01 背包问题的结果：

我们有两种选择方法的集合

①不选第 i 个物品，并且总体积不大于 j 的集合所达到的最大值:f[i-1][j]

②选择 1~i 个物品，并且总体积不大于 j 的集合所达到的最大值 f[i][j]

对于第二种情况我们很难计算，因此需要思考从另一个角度解决问题。当选择 1~i 个物品，总体积不大于 j 的集合的最大值可以转化成选择 1~i-1 个物品，总体积不大于 j-V[i]的集合+最后一个物品的价值：f[i-1][j-V[i]]+w[i]

因此总结：f[i][j]= Max{f[i-1][j],f[i-1][j-v[i]]+w[i]}!!!

对于 dp 数组的定义：

dp 数组是什么？其实就是描述问题局面的一个数组。换句话说，我们刚才明确问题有什么「状态」，现在需要用 dp 数组把状态表示出来。

首先看看刚才找到的「状态」，有两个，也就是说我们需要一个二维 dp 数组，

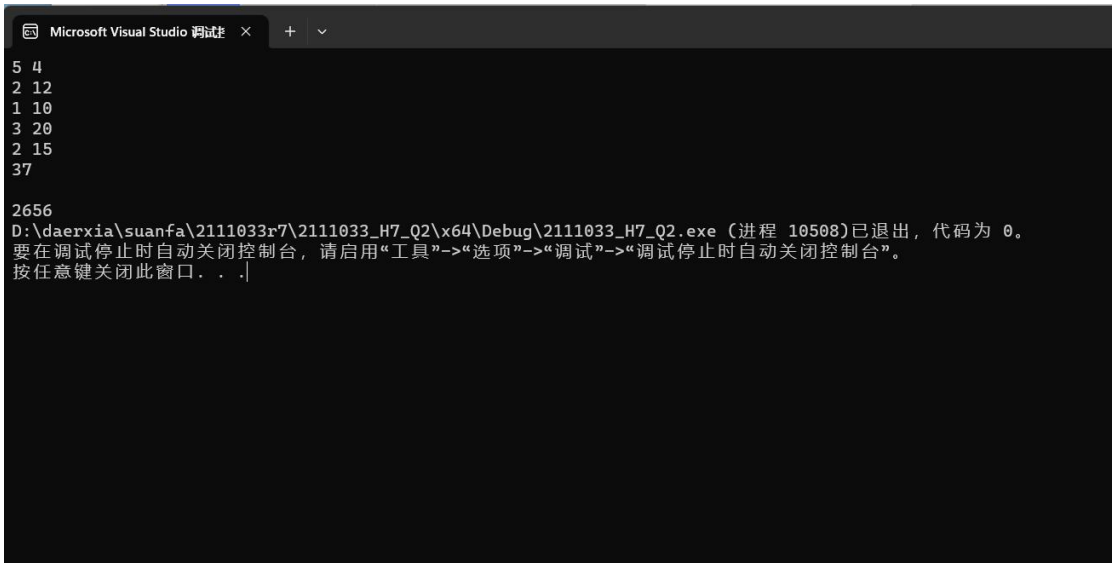
一维表示可选择的物品，一维表示背包的容量。

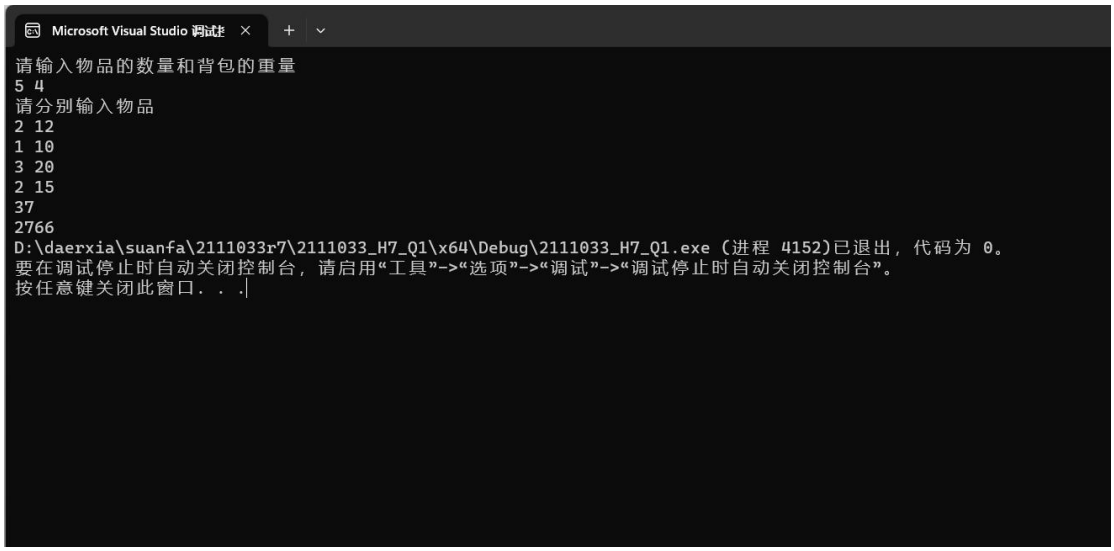
dp[i][w]的定义如下：对于前 i 个物品，当前背包的容量为 w，这种情况下可以装的最大价值是 dp[i][w]。

比如说，如果 dp[3][5] = 6，其含义为：对于给定的一系列物品中，若只对前 3 个物品进行选择，当背包容量为 5 时，最多可以装下的价值为 6。

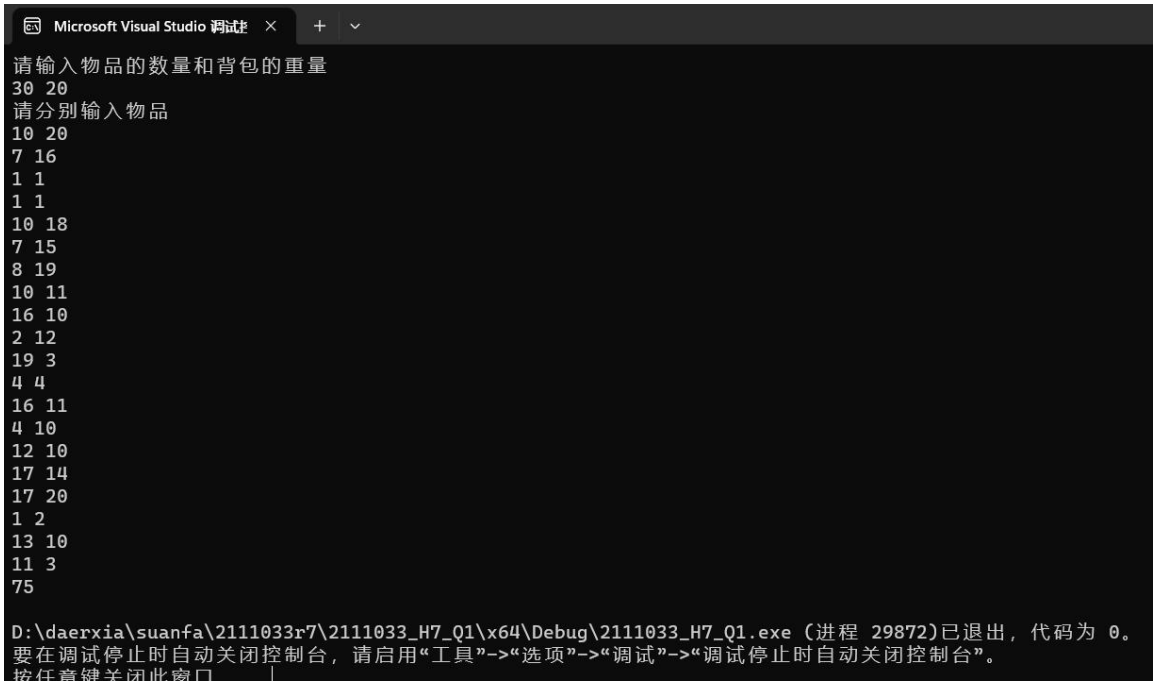
PS：为什么要这么定义？便于状态转移，或者说这就是套路，记下来就行了。建议看一下我们的动态规划系列文章，几种动规套路都被扒得清清楚楚了。

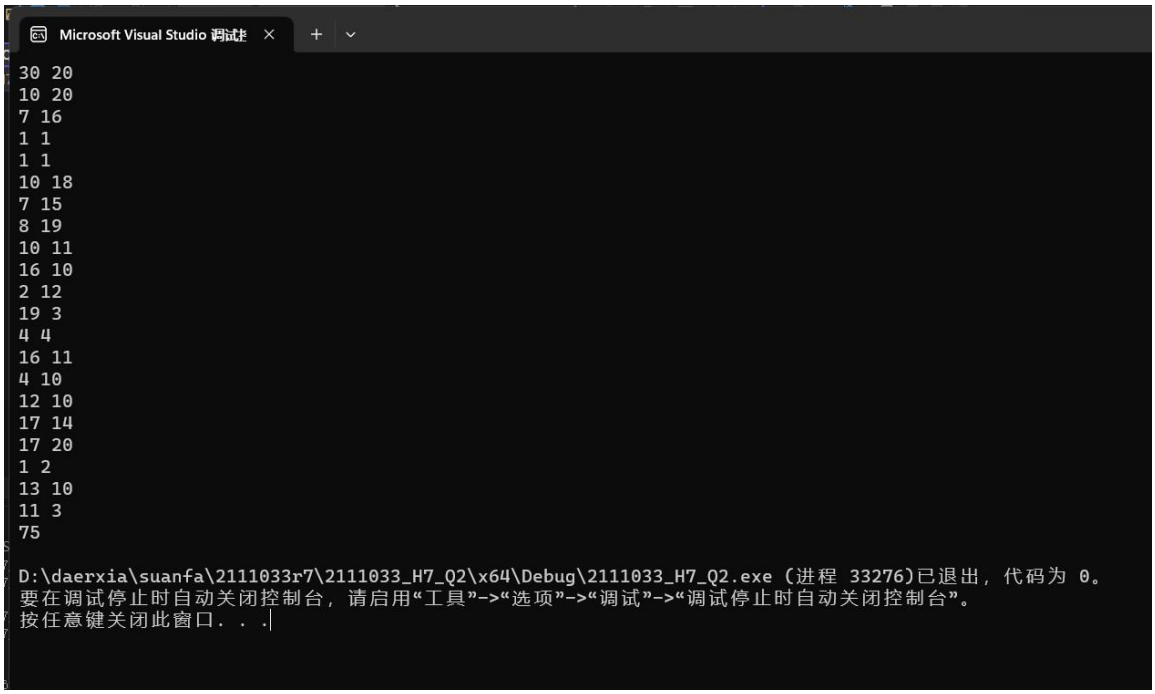
根据这个定义，我们想求的最终答案就是 dp[N][W]。base case 就是 dp[0][..] = dp[..][0] = 0，因为没有物品或者背包没有空间的时候，能装的最大价值就是 0。





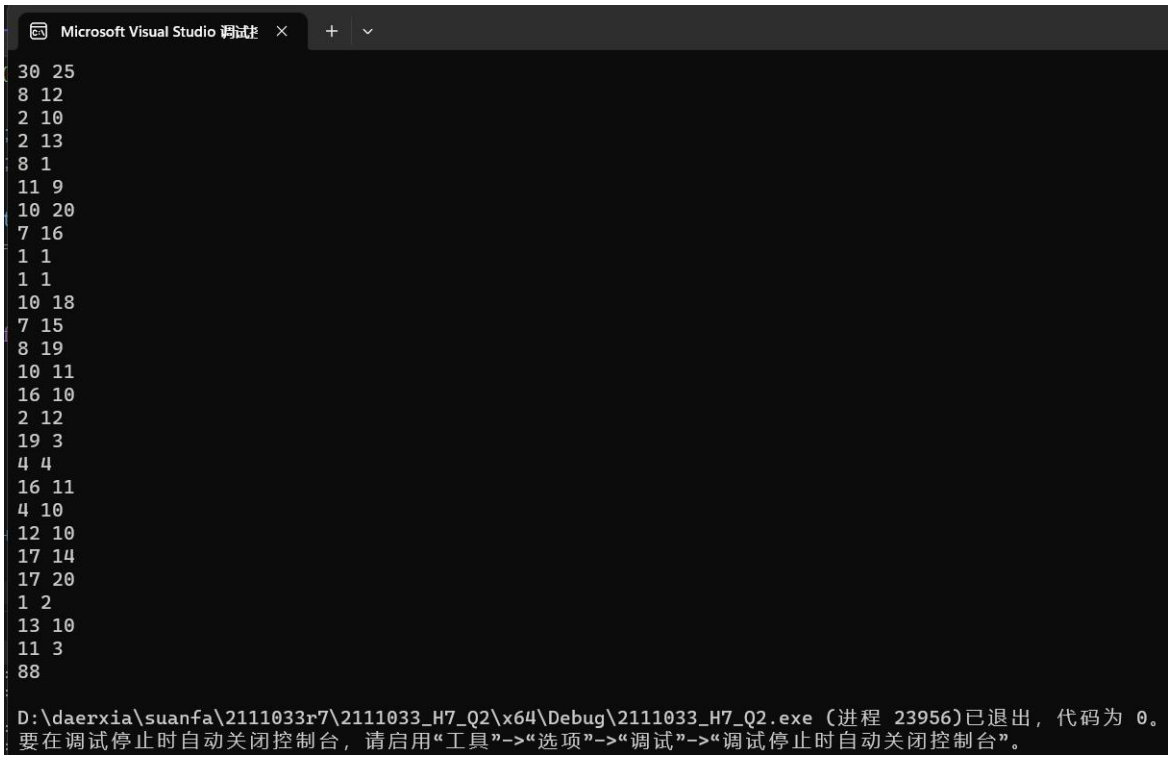
在测试样例背包容量为 4 的情况下，两种方法都可以得到正确的结果
即正确的输出 37
同时二者在时钟周期的消耗上，也并无明显的差别。

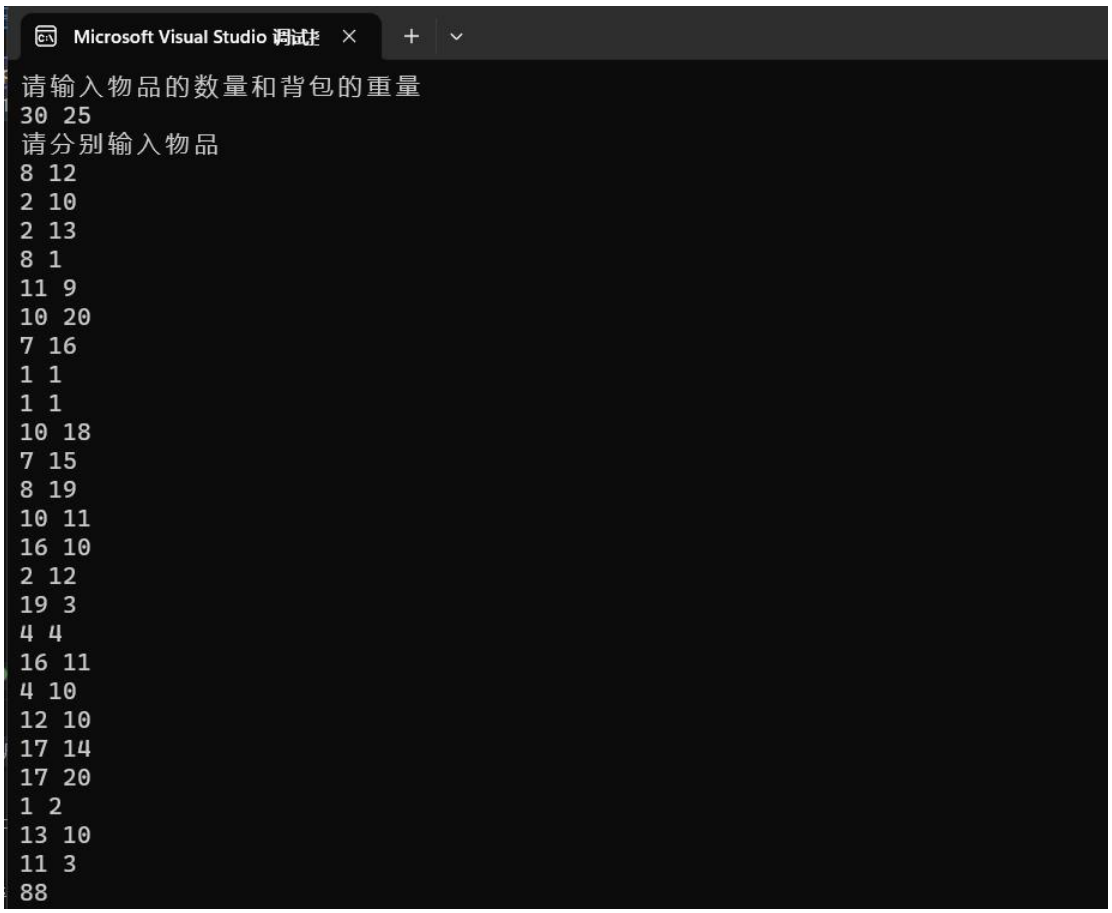




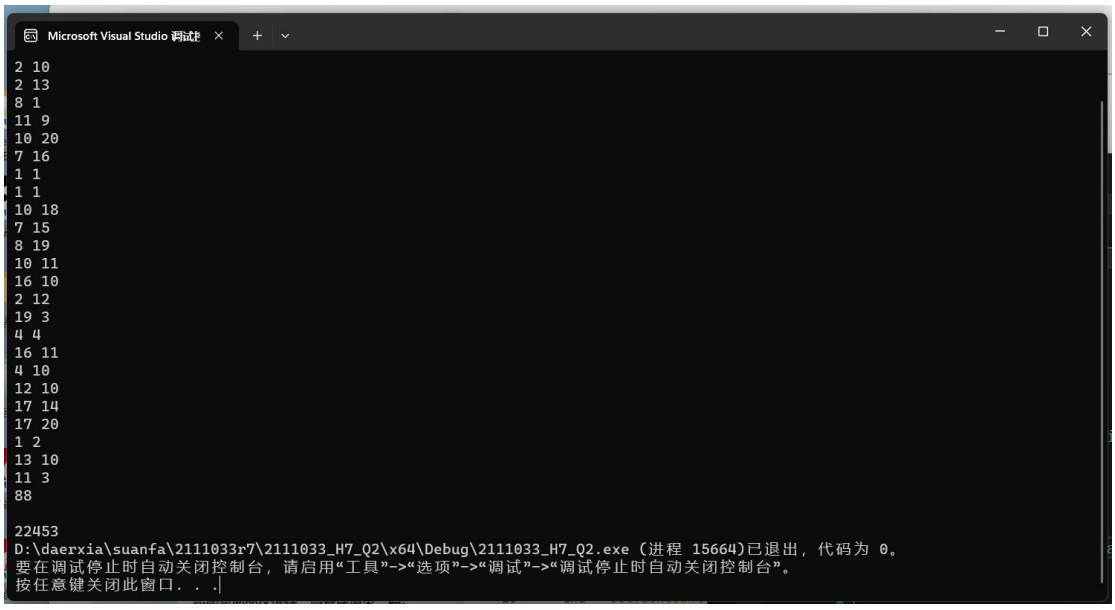
由上面两图的结果可知，在 20 的范围内，仍然得到正确的结果，可以得到正确的输出

在 25 的范围内进行计算，我们的程序经过了几秒的延迟之后，还是输出了正确的结果，与动态规划的计算结果进行验证，我们也可以发现得到了正确的结果，即 25 的范围两者都能得到正确的输出，但是动态规划的时间复杂度很低，能够更好的满足快速得到正确结果。

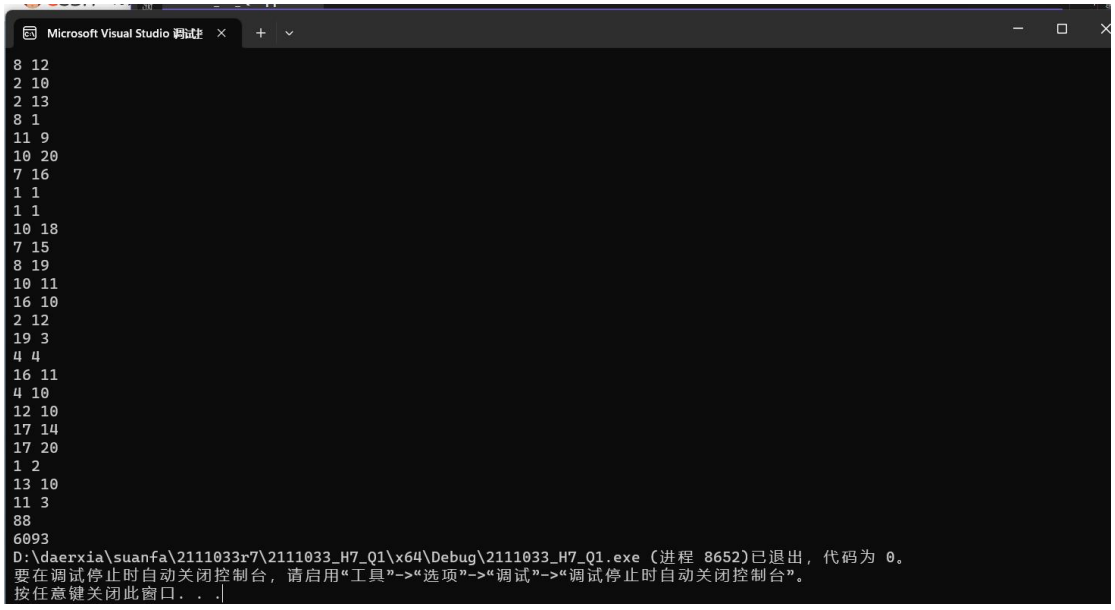




当我们更改程序，添加上时间的输出之后，我们就可以明显的看见输出时间上的差距，作为 25 数量级的算法，使用暴力枚举法需要的时钟周期是 22453

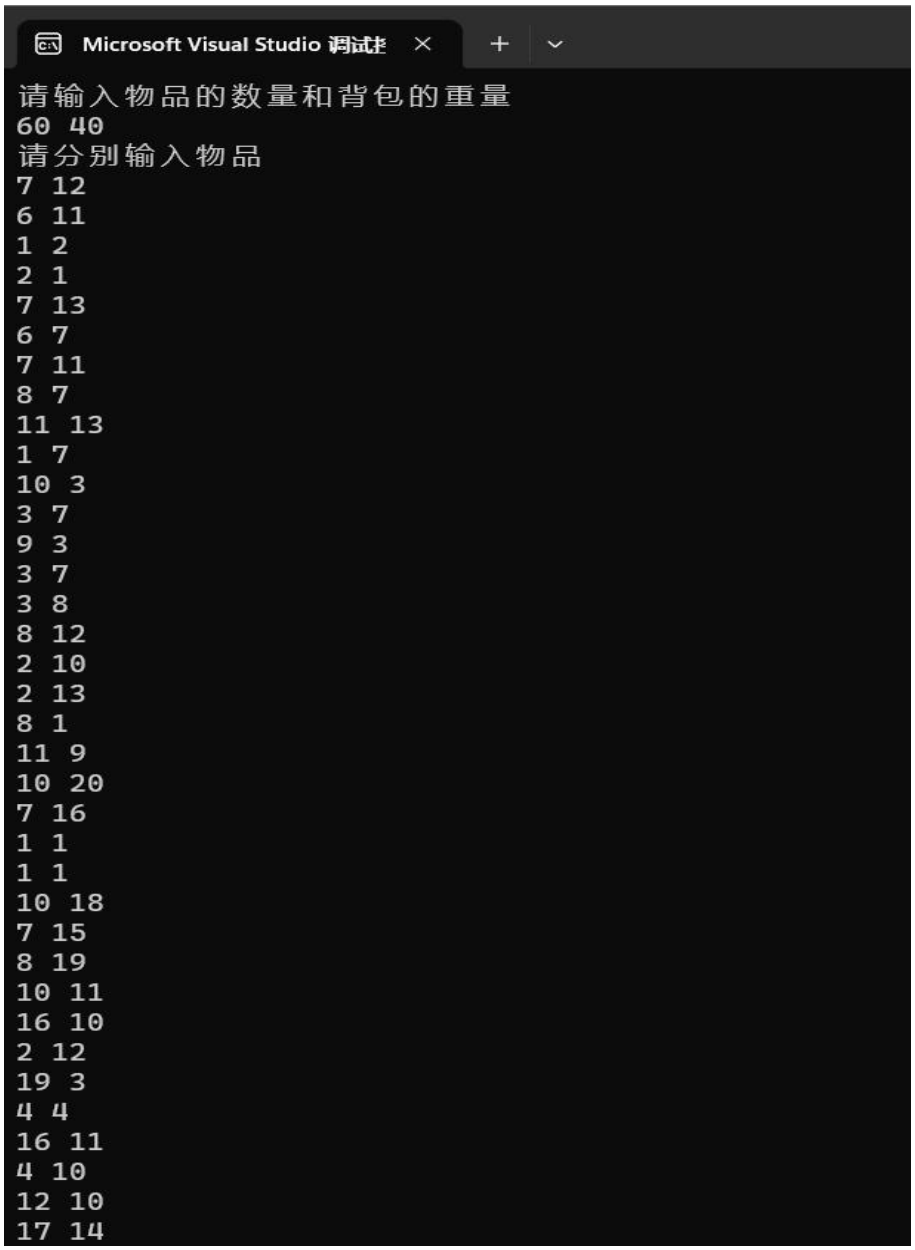


而对于动态规划法，我们可以看到时钟周期明显缩小，就能得到正确的解



当我们运算的数量级达到 40，即使我们用 long long 来定义变量，依然不能在暴力求解当中得到正确的解。

而我们在动态规划问题当中可以很容易得到正确的解


Microsoft Visual Studio 调试控
请输入物品的数量和背包的重量
60 40
请分别输入物品
7 12
6 11
1 2
2 1
7 13
6 7
7 11
8 7
11 13
1 7
10 3
3 7
9 3
3 7
3 8
8 12
2 10
2 13
8 1
11 9
10 20
7 16
1 1
1 1
10 18
7 15
8 19
10 11
16 10
2 12
19 3
4 4
16 11
4 10
12 10
17 14


8 1
11 9
10 20
7 16
1 1
1 1
10 18
7 15
8 19
10 11
16 10
2 12
19 3
4 4
16 11
4 10
12 10
17 14
17 20
1 2
13 10
11 3
159

对于动态规划，可以求得正确的解。
原因分析：可能是由于所定义的单位太大，导致数组在存储的时候位数超限。而我们在计算的过程当中，会出现超出部分的数量没有被计算的情况，在此基础上可能会出现计算只停留在部分数位的结果。

总结

动态规划算法的思想对于 01 背包问题的求解具有很好的思想价值和实际应用意义。我们使用动态规划问题在求解的时候，会发现原本指数级的算法最终只有线性的时间复杂度就可以求解出来。通过上述实验，我对动态规划问题在解决实际应用场景当中的复杂问题有了更好的理解。联想到本学期所学的其他科目，对于背包密钥的设计，我们无法通过动态规划问题求得简单的解，只能通过遍历。所以最终我们需要构造更加复杂的向量。

本次实验的过程让我对算法的理解有了更深的认识，还对代码的编写有了更多的思考的方向，还对算法优化与其他实际应用场景联系有了更加充足的认识。