算法基础（三十二）：用01背包为例来分析动态规划

我们以`0 1`背包问题来总结动态规划的一般流程：

> 有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。
>
> 第 i 件物品的体积是 v_i，价值是 w_i。
>
> 求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。
> 输出最大价值。
>
> **输入格式**
>
> 第一行两个整数，N，V，用空格隔开，分别表示物品数量和背包容积。
>
> 接下来有 N 行，每行两个整数 v_i, w_i，用空格隔开，分别表示第 i 件物品的体积和价值。
>
> **输出格式**
>
> 输出一个整数，表示最大价值。
>
> **数据范围**
>
> $0 < N, V \le 1000$
> $0 < v_i, w_i \le 1000$
>
> **输入样例**
>
> ```
> 4 5
> 1 2
> 2 4
> 3 4
> 4 5
> ```
>
> **输出样例：**
>
> ```
> 8
> ```

总的来说`dp`问题包括这三大部分：

状态表示：

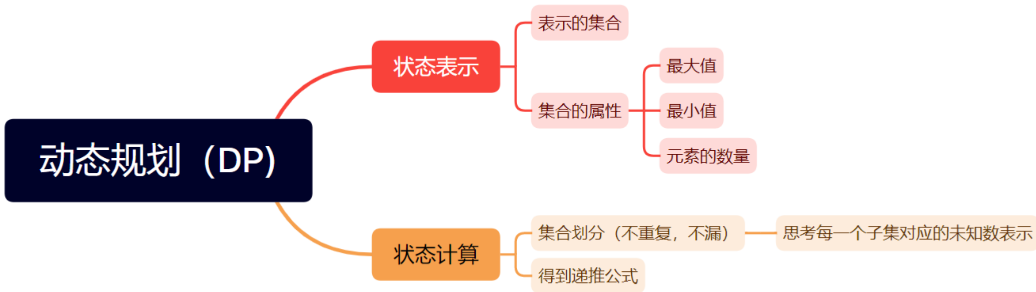
也就是一个未知数，思考用**几维**来表示我们需要解决的问题，背包问题一般是两维：`f[i, j]`

状态计算：

我们如何一步步将我们的状态计算出来，即如何算出我们的`f[i, j]`

DP优化：

一般是对动态规划的代码或者是方程进行等价变形，一般是先写出最基本的形式，然后再进行优化



接下来我们来思考这个未知数的含义是什么，未知数的含义一般有两层，第一是未知数对应的集合是什么，第二是这个未知数的值表示的是这个集合中的哪个属性：

状态对应的集合：

一般而言，每一个状态未知数都对应着一堆数值，这些数值在一起表示的是整个集合，比如**在背包问题中`f[i，j]`就对应的是一堆选法对应价值的集合，这些选法满足的条件是：物品的编号不超过`i`，物品的体积不超过`j`**

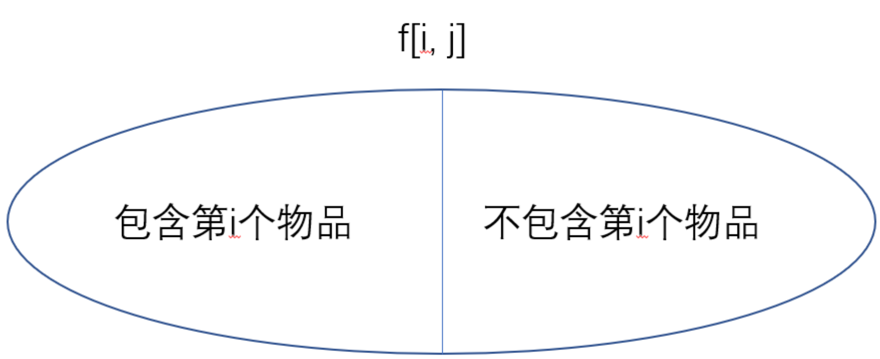
集合的属性：

这个未知数存的数其实就是它对应集合的某个属性，常见的属性有最大值，最小值，集合元素数量，在背包问题中，`f[i，j]`的值表示的就是集合的最大值

然后就是状态计算，我们在这里用集合划分的方式来进行计算

集合划分：

如何将当前集合划分成若干个更小子集，使得每一个子集我们都可以算出来，对于背包问题来说我们可以将这个状态对应的集合划分成两类：不包含第`i`个物品选法对应的价值以及包含第`i`个物品对应的价值，划分如下：



集合的划分也有两个标准：

- 不重复，集合中的某个元素必须只能属于划分的子集中的一个，但是这个原则不一定什么时候都需要满足
- 不漏，集合中的某个元素必须属于划分的子集中的一个

然后再看每一个子集的表示

对于上面划分左边的子集，它的意思就是已经选了第`i`个物品，然后在剩下的物品中选择对应的价值集合，很显然不管怎样都含有第`i`个物品的价值，那么我们剩下只能从`1 ~ i - 1`个物品中选择，并且由于已经选择了第`i`个物品，背包的剩下空间就是`j - vi`，于是这个集合对应的未知数表示就是`wi + f[i-1, j-vi]`；

对于右半边的集合，它的表示就是从`1 ~ i - 1`个物品中选物品，体积不超过`j`的选法对应的价值，那么这个集合它对应的未知数就是`f[i-1, j]`

最后得到递推公式：

很显然，`f[i, j]`的值就是左右两个集合取最大，即`f[i, j] = max(wi + f[i-1, j-vi], f[i-1, j])`

代码实现

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

const int N = 1010;
//n表示所有物品的个数
//m表示背包的容积
int n, m;
//v[i]表示物品i的体积，w[i]表示物品i的价值
int v[N], w[N];

//f[i][j]表示状态
int f[N][N];

int main()
{
    //读入物品个数和背包容量
    cin >> n >> m;

    //读入所有的物品体积和价值
    for(int i = 1; i <= n; i ++) cin >> v[i] >> w[i];

    //i = 0的所有价值都为0，不用计算
    //j = 0的所有价值都为0，不用计算
    for(int i = 1; i <= n; i ++)
        for(int j = 0; j <= m; j ++)
        {
            //不装入第i个物品的情况
            f[i][j] = f[i - 1][j];
            //对于装入第i个物品的情况，只有背包的体积大于这个物品时才能装入，否则这个情况就是一个空集
            //值为0
            if(j >= v[i]) f[i][j] = max(f[i][j], f[i - 1][j - v[i]] + w[i]);
        }

    cout << f[n][m] << endl;

    return 0;
```

```
}
```

代码优化

首先我们看到这个递推公式:

```
//不装入第i个物品的情况
f[i][j] = f[i - 1][j];
//对于装入第i个物品的情况，只有背包的体积大于这个物品时才能装入，否则这个情况就是一个空集
//值为0
if(j >= v[i]) f[i][j] = max(f[i][j], f[i - 1][j - v[i]] + w[i]);
```

我们以`i`作为层数，我们可以发现再计算第`i`层的`f[i][j]`的时候我们只使用的第`i - 1`层的`j`，所以我们其实可以只用一维的数组`f[j]`，当计算完第`i`层的`f[j]`之后，进入第`i + 1`层时，此时的`f[j]`仍然是第`i`层的值，我们直接对`f[j]`使用自己的值进行改变，变成第`i + 1`层

```
    for(int i = 1; i <= n; i ++)
        for(int j = 0; j <= m; j ++)
        {
            //不装入第i个物品的情况
            f[j] = f[j];
            //对于装入第i个物品的情况，只有背包的体积大于这个物品时才能装入，否则这个情况就是一个空集
            //值为0
            //if(j >= v[i]) f[i][j] = max(f[i][j], f[i - 1][j - v[i]] + w[i]);
            if(j >= v[i]) f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
```

由于`j`是从小到大开始遍历，所以当计算到`j = k`的时候使用的值是`f[j - v[i]]`，其中`j - v[i]`一定是小于`k`的，也就说明`f[j - v[i]]`在之前已经更新过，说明此时使用的`f[j - v[i]]`不是原来的`i - 1`层的值，所以还需要进行进一步更改，那么此时我们`j`从后往前遍历即可:

```
    for(int i = 1; i <= n; i ++)
        for(int j = m; j >= 0; j --)
        {
            //f[i][j] = f[i - 1][j];
            f[j] = f[j];
            //if(j >= v[i]) f[i][j] = max(f[i][j], f[i - 1][j - v[i]] + w[i]);
            if(j >= v[i]) f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
```

进一步将`if`语句进行优化可得:

```
    for(int i = 1; i <= n; i ++)
        for(int j = m; j >= v[i]; j --)
            f[j] = max(f[j], f[j - v[i]] + w[i]);
```