马上开始

35tang-C++竞赛系列三阶课程

《35tang-C++竞赛系列三阶课程》

本节目标

- 背包问题

经典0/1背包问题

- 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，使得装入背包中的物品的总价值最大?每种物品只有一个。

价值：int v[N]={0,8,10,6,3,7,2};
重量：int w[N]={0,4,1,2,2,5,1};
背包容量：int n=6, c=12;
数组从1号元素开始使用，上面数组6个物品

例如：背包容量12的情况下，每个物品只有一个。
可以放6，5，4，3，2号物品，总重量1+5+2+2+1=11,不超过12，价值是2+7+3+6+10=28
也可以放6，5，4，1号物品，总重量1+5+2+4=12,不超过12，价值是2+7+3+8=20

类似有序枚举

排列的搜索

```
int ans=0;
void dfs(int i,int j,int sumv)
//i表示当前要选第几个，j表示当前背包剩下的容量，sumv表示当前选择得物品的总价值
{
        if (i==n+1) {//打擂并返回（一条路径到头了）
                ans=max(ans,sumv);
                return;
        }
        //不选i项
        dfs(i+1,j,sumv);
        //选i项
        if( w[i]<=j) dfs(i+1,j-w[i],sumv+v[i]);
}
int main()
{
        dfs(1,c,0);
        cout<<ans<<endl;
        return 0;
}
```

dfs 用函数返回值的方法返回答案

```
int getvalue(int i,int j)
//i表示当前要选第几个，j表示当前背包剩下的容量
//函数返回值表示当前从当前第i个物品往后选可能的最大值
//第i项选和不选往后走返回一个最大的
{
        if (i==n+1) {//一条路径到头了
            return 0;
        }
        //不选i项 和选i项往后走返回的最大值
       //不选，容量还是j；选了，容量变成j-w[i]
        if( w[i]<=j) return max(getvalue (i+1,j),v[i]+getvalue (i+1,j-w[i]));
        else return getvalue (i+1,j);
       // if( w[i]<=j) 这个判断是保证背包剩下容量放得下物品i
}
int main()
{
       cout<<getvalue(1,c)<<endl;
       return 0;
}
```

dfs 用函数返回值的方法返回答案：这次倒过来，从最后一个开始选，道理是一样的，只不过第一次调用传入参数不同

```
int getvalue(int i,int j)
//i表示当前要选第几个，j表示当前背包剩下的容量
//函数返回值表示当前从当前第i个物品往前选可能的最大值
//第i项选和不选往前走返回一个最大的
{
        if (i==0||j==0) {//打擂并返回（一条路径到头了）
                return 0;
        }
        //不选i项 和选i项往前走返回的最大值
        if( w[i]<=j) return max(getvalue (i-1,j),v[i]+getvalue (i-1,j-w[i]));
        else return getvalue (i-1,j);
}
int main()
{
        cout<<getvalue(n,c)<<endl;
        return 0;
}
```

也可以这么理解搜索的算法

- 总容量是C，一共i个物品，如果C>W[i]，我们就看看放物品i和不放物品i哪个价值更大，如果放，背包容量变成了C-W[i],然后这个容量放前面的i-1个物品的总价值+物品i的value；如果不放，就是用容量C去放前面i-1个物品，这两种情况看看哪一个的value最大。这就是两个相邻节点，两个分支情况。

```
//j是重量，i是当前可以放的最大的一个物品编号
int getvalue(int j, int i)
{
        if (j==0 ||i==0) return 0;//找完了或者容量为0了
        if (j>=w[i]) return max(getvalue(j,i-1), getvalue(j-w[i],i-1)+v[i]) ;
        else return getvalue(j,i-1);
}
```

记忆化搜索：

设置dp[i][j]为前i个物品里面去选择，容量是j的时候的最大价值,也就是函数getvalue的结果

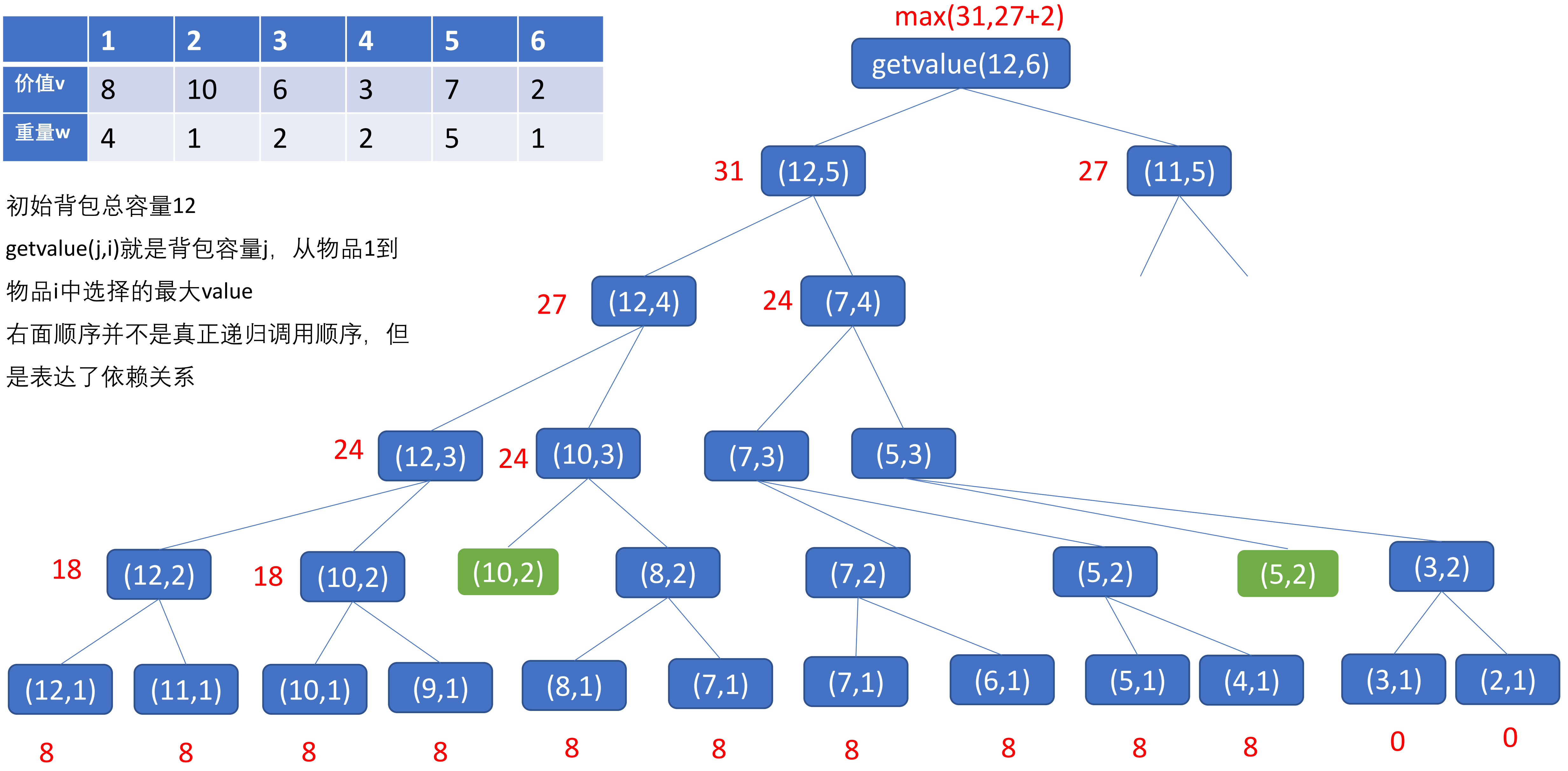
```
int dp[7][13]={0};

int getvalue(int i,int j)
//i表示当前要选第几个，j表示当前背包剩下的容量
//函数返回值表示当前从当前第i个物品往前选可能的最大值
//第i项选和不选往前走返回一个最大的
{
        if (dp[i][j]..) return dp[i][j];
        if (i==0||j==0) {//打擂并返回（一条路径到头了）
                return 0;
        }
        if( w[i]<=j) return dp[i][j]=max(getvalue(i-1,j),v[i]+getvalue (i-1,j-w[i]));
        else return dp[i][j]=getvalue (i-1,j);
}
 int main()
{

        cout<<getvalue(n,c)<<endl;
        //cout<<dp[n][c]<<endl;//注意，所有的结果都存放在dp数组
        return 0;
}
```

	1	2	3	4	5	6
价值v	8	10	6	3	7	2
重量w	4	1	2	2	5	1

初始背包总容量12

getvalue(j,i)就是背包容量j，从物品1到物品i中选择的最大value

右面顺序并不是真正递归调用顺序，但是表达了依赖关系



1.绿色是不是重复了？这些用记忆化搜索就可以不去搜索第二遍

2.如果先算出最下面一排，是不是可以一层一层往上推，直到推导出最上面的（12，6）也就是容量12，从物品1到6中选择的最大价值

- 倒着推，先算出小的，推大的，不知道哪些容量都会用到，干脆就都推出来，用不到的也没有关系。空间换时间。
- 关键是getvalue(j, i)的值，getvalue(j,i)不就是dp[i][j]?:

```
if (j>=w[i])
    dp[i][j]=max(getvalue1(j,i-1), getvalue1(j-w[i],i-1)+v[i]) ;
else dp[i][j]=getvalue1(j,i-1);
```

于是，正着推，先算出i，j比较小的dp值，再推导出后面的：

```
if(j>=w[i])
        dp[i][j]=max(dp[i-1][j],dp[i-1][j-w[i]]+v[i]);
    else
        dp[i][j]=dp[i-1][j];
```

i从小到大，j从小到大，现在要计算dp[5][7]，也就是背包容量7，前5个物品的最大价值？

根据公式，dp[5][7]=max(dp[4][2]+7,dp[4][7]
背包容量7，前5个物品的最大价值=max（选择第5个物品：剩下背包容量2去选择前4个物品的最大值，不选择第5个物品：背包容量7选择前4个物品）

	1	2	3	4	5	6
价值v	8	10	6	3	7	2
重量w	4	1	2	2	5	1

dp[][]

背包容量j

可选的最后1个物品i

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	8	8	8	8	8	8	8	8	8
2	10	10	10	10	18	18	18	18	18	18	18	18
3	10	10	16	16	18	18	24	24	24	24	24	24
4	10	10	16	16	19	19	24	24	27	27	27	27
5	10	10	16	16	19	19	?					
6												

递推的理解：

i，j从小到大循环，把dp[i][j]从小到大依次都计算出来

设置dp[i][j]为前i个物品里面去选择，背包剩余容量是j的时候的最大价值。

对于第i个物品，如果他的容量超过了j，那么最大总价值是前面i-1个物品总容量是j的最大价值；

如果物品i的容量不超过j，那么对于物品i，如果选了它，他占用了w[i]的容量，贡献了v[i]的价值，这个时候的总价值就是从前面j-1个物品里面选择出j –w[i]这么大的容量所对应的最大价值；如果不选i，这时候的总价值就是总容量j放前面i-1个物品。比较两个哪个大。

```
for(int i=1;i<=n;i++)
  {
     for(int j=1;j<=c;j++)
     {
        if(j>=w[i])
           dp[i][j]=max(dp[i-1][j],dp[i-1][j-w[i]]+v[i]);
        else
           dp[i][j]=dp[i-1][j];
     }
  }
```

0/1背包问题

- 给定n种物品和一背包。物品i的重量（容量）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，使得装入背包中的物品的总价值最大?**每种物品只有一个**。
- 用dp[i][j]表示前i件物品放入一个容量为j的背包可以获得的最大价值，我们最后要找的就是dp[n][C]。其状态转移方程（就是如何一步一步推导dp[i][j]的公式）为：

```
if(j>=w[i])
        dp[i][j]=max(dp[i-1][j],dp[i-1][j-w[i]]+v[i]);
else
        dp[i][j]=dp[i-1][j];
```

  也就是说对于容量j的时候，从i个物品里面选择放入背包，最大的value在两个可能性里面选一个：要么不选物品i，要么选物品i，选和不选都对应前面一个dp数组的元素或者说状态

- 初始化：没有物品可以放的时候或者背包容量是0的时候价值都是0。所以dp数组开始可以初始化都为0。

- 压缩

dp[i][j]=max(dp[i-1][j],dp[i-1][j-w[i]]+v[i]); //注意外循环i从小到大

当我们计算i=4的时候，图中绿色一行，所有的dp[4][j]其实都只依赖于dp[3][j], 注意我们外面循环i是从小到大的，所以i=3的时侯j循环走完的时候，dp[3][j]都计算出来了，换个思路如果把数组变成dp[j]不记录i这一个维度了，那么i=3计算完的时候dp[j]是不是计算的就是所有的dp[3][j],当我们计算i=4的时候，dp[j]的value（原来的dp[4][j]）分别依赖于这个时候的dp[j]，因为这个时候的dp[j]就是原来的dp[3][j]。

所以压缩成dp [j] = max{dp[j], dp[j - w[i]] + v[i]}

	1	2	3	4	5	6
价值v	8	10	6	3	7	2
重量w	4	1	2	2	5	1

背包容量j

可选的最后1个物品i

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	8	8	8	8	8	8	8	8	8
2	10	10	10	10	18	18	18	18	18	18	18	18
dp[3][j] 3	dp[1]	dp[2]	dp[3]	dp[4]	dp[5]	dp[6]	dp[7]	dp[8]	dp[9]	dp[10]	dp[11]	dp[12]
dp[4][j] 4	dp[1]	dp[2]	dp[3]	dp[4]	dp[5]	dp[6]	dp[7]	dp[8]	dp[9]	dp[10]	dp[11]	dp[12]
5	10	10	16	16	19	19	24	24	27	27	27	31
6	10	12	16	18	19	21	24	26	27	29	29	31

	1	2	3	4	5	6
价值v	8	10	6	3	7	2
重量w	4	1	2	2	5	1

- 但是，当我们计算第四行的dp[12]的时候，应该是

dp [12] = max(dp[12], dp[12- w[4]] + v[4]) =max(dp[12], dp[10]+3)

问题出来了，我们这里的红色的dp[10]应该是i=3的时候计算出来的dp[10]也就是蓝色的，但是如果我们的j从1到12循环，计算i=4的时候的dp[12]的时候dp[10]已经计算过了，我们要的紫色的，但是现在是白色的，或者说我们希望现在的dp[10]是原来二维的dp[3][10],而不是原来的dp[4][10]。

怎么办？j循环倒着来。

背包容量j

可选的最后1个物品i

		1	2	3	4	5	6	7	8	9	10	11	12
	1	0	0	0	8	8	8	8	8	8	8	8	8
	2	10	10	10	10	18	18	18	18	18	18	18	18
dp[3][j]	3	dp[1]	dp[2]	dp[3]	dp[4]	dp[5]	dp[6]	dp[7]	dp[8]	dp[9]	dp[10]	dp[11]	dp[12]
dp[4][j]	4	dp[1]	dp[2]	dp[3]	dp[4]	dp[5]	dp[6]	dp[7]	dp[8]	dp[9]	dp[10]	dp[11]	dp[12]
	5	10	10	16	16	19	19	24	24	27	27	27	31
	6	10	12	16	18	19	21	24	26	27	29	29	31

0/1背包问题压缩空间

```
for(int i=1;i<=n;i++)
 {
        for(int j=c;j>=w[i];j--)
        {
                              dp [j]=max(dp[j],dp[j-w[i]]+ v[i]);
        }
    }
cout<<dp[c]<<endl;
```

循环到i的时候，现在的每一个dp[j]里面放的就是刚才的i-1时候的所有的dp[j]，相当于dp[i-1][j]，如果不倒着来，计算dp[i][j]时候依赖的dp[i][j - w[i]] 就有可能不是dp[i-1][j - w[i]]， 而是dp[i][j - w[i]] 了。

- 上面简化公式，也可以这么理解：假设dp[j]表示容量为j 从n个物品选择任意个的最大价值。对于每一个容量j，他的最大值等于max{每一个物品放和不放的最大值}

```
for(int i=1;i<=n;i++)
{
    for(int j=c;j>=w[i];j--)
    {
            dp [j]=max(dp[j],dp[j-w[i]]+ v[i]);
    }
}
```

- 注意：dp初始化是0。绿色部分是容量j小于物品i时候的，对于每一个i不会计算，实际上在计算当前i的时候如果用到了用的是前面的比较小的i计算出来的结果。

	1	2	3	4	5	6
价值v	8	10	6	3	7	2
重量w	4	1	2	2	5	1

i从小到大，j从大到小，现在要计算i是5的时候的dp[7]，也就是背包容量7，前5个物品的最大价值？

根据公式，dp [7]=max(dp[7],dp[2]+7)
背包容量7，前5个物品的最大价值=max（前4个物品容量7的最大价值，选择物品5的价值7+前4个物品容量2的最大价值）

背包容量j

可选的最后1个物品i

	1	2	3	4	5	6	7	8	9	10	11	12
1				8	8	8	8	8	8	8	8	8
2	10	10	10	10	18	18	18	18	18	18	18	18
3		10	16	16	18	18	24	24	24	24	24	24
4		10	16	16	19	19	24	24	27	27	27	27
5							?	24	27	27	27	31
6												

完全背包问题

- 完全背包和0/1背包问题的不同就在于每一种物品可以有很多个，而不像上面的0/1背包，每一个物品只有一个，要么放，要么不放，完全背包对于每一个物品的数量没有限制，每一个物品可能是0个，1个，2个……。

完全背包

- 有N种物品和一个容量为C的背包，每种物品都有**无限**件可用。第i种物品的重量(费用)是w[i]，价格是v[i].求解将哪些物品装入背包可使这些物品的重量（费用）总和不超过背包容量，且价值总和最大。
- 令dp[i][j]表示前i种物品恰好放入一个容量为j的背包的最大值。三重循环：

  dp[i][j] = max{dp[i-1][j- k * w[i]] + k * v[i]} //放k（0,1,2…）个物品i的情况打擂找最大

  i从1到n，j从w[i] 到C， 0 <= k * w[i]<= j
- 比0/1背包多出来一个k的循环。就是说容量j从前i个物品里面选择，第i个物品可以选k个(0个，1个，2个…)，多出来的这个循环就是所有情况去打擂（0/1背包实际上只能选择0个和1个，所以不需要这个k循环，直接用max函数取两种情况的最大）

完全背包简化：注意：红色部分和0/1背包的区别？

另一种双循环的做法：

- dp [i][j] = max(dp[i-1][j],dp[i][j-w[i]]+v[i])

就是说i个物品里面选择，背包容量是j时侯的最大价值，等于：
不选第i个物品，容量j从前面i-1个物品里面选择的最大值；
和选择了第i个物品，剩下的容量在i个物品里面选择的最大值。

```
for(int i=1;i<=n;i++)
   {
      for(int j=1;j<=c;j++)
      {
         if(j>=w[i])
            dp[i][j]=max(dp[i-1][j],dp[i][j-w[i]]+v[i]);
         else
           dp[i][j]=dp[i-1][j];
      }
   }
```

完全背包压缩成一维数组

简化：

- dp [j] = max{dp[j], dp[j - w[i]] + v[i]}

  i从1到n，j从w[i]到C

```
for(int i=1;i<=n;i++)
{
    for(int j=w[i];j<=c;j++)
    {
        // dp[i][j]=max(dp[i-1][j],dp[i][j-w[i]]+v[i]);
           dp [j]=max(dp[j],    dp[j-w[i]]+ v[i]);
    }
}
cout<<dp[c]<<endl;
```

```
for(int i=1;i<=n;i++)
for(int j=w[i];j<=c;j++)
        // dp[i][j]=max(dp[i-1][j],dp[i][j-w[i]]+v[i]);
          dp [j]=max(dp[j],    dp[j-w[i]]+ v[i]);
```

	1	2	3	4	5	6
价值v	8	10	6	3	7	2
重量w	4	1	2	2	5	1

- 这里和前面0/1背包不一样的：j要从小到大，比如计算第一行的dp[12]的时候，需要保证dp[8]已经计算过了。或则说这个时候的dp[i]需要用到的比i小的dp是当前物品选择时候的dp而不像0/1背包那样由i-1的计算出来的

- 最关键：关键是dp[j]不是依赖于前i-1个物品的dp，而是依赖于当前i个物品时候的dp [k] {k<j}。

背包容量j

可选的最后1个物品i

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	8	8	8	8	16	16	16	16	24
2	10	20	30	40	50	60	70	80	90	100	110	120
3	10	20	30	40	50	60	70	80	90	100	110	120
4	10	20	30	40	50	60	70	80	90	100	110	120
5	10	10	16	16	50	60	70	80	90	100	110	120
6	10	20	30	40	50	60	70	80	90	100	110	120

完全背包新的理解方法

新的理解：dp[j]表示每一个容量j在所有物品里面去选择的最大值,

dp[j]=max{选择一个物品i之后剩下的容量可以得到的最大值+物品i的价值}，这里i从1到n都去打擂把最大的留在dp[j]里面。

注意容量从前往后选，从而保证容量j小的都计算过了。

注意：这里找最大不是从前i个物品去选择，而是在所有物品中选择。

```
 for(int j=1;j<=c;j++)
 for(int i=1;i<=n;i++)//尝试选择物品i，打擂得到dp[j]的最大值
              if (j>=w[i]) dp [j]=max(dp[j],dp[j-w[i]]+ v[i]);
cout<<dp[c]<<endl;
```

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

	1	2	3	4	5	6
价值v	8	10	6	3	7	2
重量w	4	1	2	2	5	1

dp[j]表示每一个容量j在所有物品里面去选择的最大值,dp[j]=max{选择一个物品i之后剩下的容量可以得到的最大值+物品i的价值}，注意容量从前往后选，从而保证容量j小的都计算过了。

j从小到大，i从小到大，现在要计算j是5的时侯的dp[5]，也就是背包容量5的最大价值，就是挨个物品尝试选择，找最大。

当j为5，i为2的时侯，如果选择了物品2，剩下背包容量5-1=4，最大价值dp[4]为40，这个价值加上物品2的价值10为50，是当前最大的dp[5]。注意：这个时侯的dp[4]是考虑了所有物品任意去选的最大可能。

背包容量j

挨个尝试选择1个物品i之后的最大值

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	8	18							
2	10	20	30	40	?							
3	10	20	30	40	?							
4	10	20	30	40								
5	10	20	30	40								
6	10	20	30	40								

dp[j]所有i都尝试后的最大值

硬币的完全背包问题

- 给定若干coin的面额，比如，1，2，5，计算凑成一个amount的最小的硬币数目，硬币是无限多的。
- 这里硬币的面值相当于背包里面的物品重量，amount就是背包总容量，value都是1。
- 分析，如果设定dp[i]表示总价值为i的需要的最小硬币数目。

理解

- 这里是对每一个j，打播找dp[i-coins[j]]+1里面最小的放在dp[i]里面.

dp[i]=min(dp[i-coins[j]]+1, dp[i]); //j去尝试每一个硬币

  - 这么理解：如果面额是i，这个时候的最小硬币数目就是依次尝试减去每一个硬币的面额，看看剩下的最小需要硬币的数目里面的最小值，+1就是说需要的数目要加上当前这个硬币。

按照压缩后的完全背包去理解:

假设amount是10，给出3种硬币面值为1 2 5.

这个时候的最小硬币数目就是依次尝试减去每一个硬币的面额，看看剩下的最小需要硬币的数目里面的最小值，+1就是说需要的数目要加上当前这个硬币。

- dp[10]=min(dp[9],dp[8],dp[5])+1;
- dp[9]=min(dp[8],dp[7],dp[4])+1;
- dp[8]=min(dp[7],dp[6],dp[3])+1;
- dp[7]=min(dp[6],dp[5],dp[2])+1;
- dp[6]=min(dp[5],dp[4],dp[1])+1;
- dp[5]=min(dp[4],dp[3],dp[0])+1;
- dp[4]=min(dp[3],dp[2])+1;
- dp[3]=min(dp[2],dp[1])+1;
- dp[2]=min(dp[1],dp[0])+1;
- dp[1]=min(dp[0])+1;
- dp[0]=0;

完全背包

dp[0]初始化为0

其他dp[i]初始化为最大值，因为要打擂找最小。

```
int selectcoins(int flagcoin[])
{
    int dp[102]={0};//dp[i],金额为i的时候的最小硬币数目
    for (int i=1;i<=amount;i++){
        dp[i]=INT_MAX; //打擂，初始化为最大
        for (int j=0;j<coinslength;j++){
            if (coins[j]<=i){
                dp[i]=min(dp[i-coins[j]]+1,dp[i]);
                if (dp[i]==dp[i-coins[j]]+1)     flagcoin[i]=coins[j];
            }
        }
    }
    return (dp[amount]);
}
```

里面引入了一个flagcoin数组，flagcoin[i]是金额为i的最小coin数目时候的第一个coin的面额。这样的做法是为了后面能够回溯出最小数目的硬币组成。有兴趣可以课后研究程序。

```
for(int i=amount;i>0;i-=flagcoin[i])  cout<<flagcoin[i]<<"\t";
```

多重背包问题

- 有N种物品和一个容量为V的背包。第i种物品**最多有m[i]件**，每件重量(费用)是w[i]，价值是v[i]。求解将哪些物品装入背包可使这些物品的重量(费用)总和不超过背包容量，且价值总和最大。就是说物品的数量不是1，也不是随便用，而是有一个最大量。
- 这个就没有办法按照完全背包的做法压缩成二重循环了：容量j从前i个物品里面选择，第i个物品可以选k个(0个，1个，2个…m[i]个)，所有情况的最大值。0/1背包实际上多重背包的特列，只有0个和1个两种情况，所以用一个max函数比较这两种情况就够了。而多重背包这里，需要对i选0个，1个，2个….m[i]个的情况打擂，所以多出来一重循环。
  - dp[i][j] = max{dp[i-1][j- k * w[i]] + k * v[i]}

    i从1到n，j从1 (w[i])到C， 0 <= k <=m[i]

    需要保证j> k * w[i]

写一下？

```
#include <iostream>
using namespace std;
int main()
{
   int c = 10;//背包容量
   int n=5;//5个物品
   int w[6] = { 0,5,4,3,2,1 };//物品重量
   int v[6] = { 0,1,2,3,4,5 };//物品价值
   int m[6] = { 0,1,2,1,2,1 };//物品数量
   int dp[11][11] = { 0 };

  for(int i = 1; i <= n; i++)
  for(int j = w[i]; j <= c;j++)//这里j从0或者w[i]开始都可以
  for(int k = 0; k <= m[i]; k++)
     if (j>k*w[i])
       dp[i][j] = max(dp[i][j],dp[i-1][j- k * w[i]] + k * v[i]);
    //也可以类似0/1背包的压缩，压缩为1维数组，节省空间，但是时间复杂度一样的。

   cout << dp[n][c] << endl;
   return 0;}
```

尝试理解细节，为什么压缩，顺序等等

理解以后记住模板程序

区别几个不同的背包问题的不同写法。当作模板！以后遇到类似问题关键就是去找什么是“背包容量”，什么是“物品重量”，最后求解的是什么？是最大价值？最少方案数？最多方案数？

0/1背包和完全背包分别压缩成一维数组后的程序非常类似，只不过是容量循环的顺序不同。一般情况下，如果不压缩，循环顺序取决于你的公式，如果你计算i项需要用到的是i-1项，i循环就应该从小到大。

下面循环一般情况都是从小到大：

0/1背包：双重循环（背包容量，物品）

完全背包：三重循环（背包容量，物品，物品可选的个数）

双重循环（背包容量，物品）

多重背包：三重循环（背包容量，物品，物品可选的个数）

作业1： NOIP 2005 采药 medicine.cpp

- 0/1背包

挑战作业2 money.cpp

- USACO training Section 2.3　PROB Money Systems
- https://www.luogu.com.cn/problem/P1474
  - 这是标准的完全背包。

挑战作业3：

NOIP2012 摆花 flower.cpp 多重背包问题

- 小明的花店新开张，为了吸引顾客，他想在花店的门口摆上一排花，共 m 盆。通过调 查顾客的喜好，小明列出了顾客最喜欢的 n 种花，从 1 到 n 标号。为了在门口展出更多种花， 规定第 i 种花不能超过 a_i盆，摆花时同一种花放在一起，且不同种类的花需按标号的从小到 大的顺序依次摆列。 试编程计算，一共有多少种不同的摆花方案。