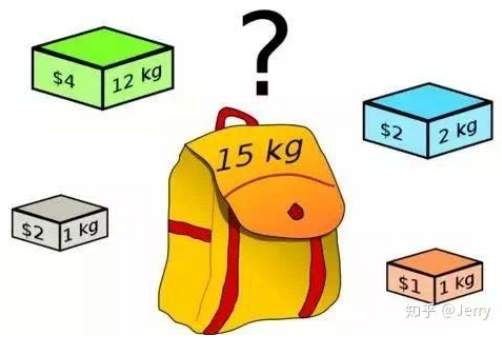
背包专题



关键：

- 01背包与完全背包模型
- 背包问题的多种形式与求解策略
- 动态规划优化技巧

3.9.0 动态规划算法设计与实现

在设计动态规划方法与应用时需注意：

- 问题的状态表示对能否用动态规划进行求解是至关重要的，不恰当的状态表示将使问题的描述不具有最优子结构性质，从而无法建立最优值的递归关系，动态规划的应用也就无从谈起。因此，状态表示和最优子结构性质的分析，是最关键的一步。
- 在算法的程序设计中，应充分利用子问题重叠性质来提高效率。更具体地说，应采用递推(迭代)的方法来编程计算由递归式定义的最优值，而不采用直接递归的方法。

3.9.0 动态规划算法设计与实现

(1)划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段，划分后的阶段一定要是有序的或者是可排序的，否则问题就无法求解。

(2)确定状态和状态变量：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来，状态的选择要满足无后效性。

(3)确定决策并写出状态转移方程：因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。如果确定了决策，状态转移方程也就可写出。但事实上常常是反过来做，根据相邻两段各状态之间的关系来确定决策。

3.9.0 动态规划算法设计与实现

(4)寻找边界条件：给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。

(5)程序设计实现：动态规划的主要难点在于理论上的设计，一旦设计完成，实现部分就会非常简单。

DP解题的框架如下:

① 初始化(边界条件)

② for i=2 to n (**顺推法**) 或 for i=n-1 to 1(**逆推法**)

对i阶段的每一个决策点求局部最优

③ 确定和输出结束状态的值.

3.9.0 背包问题

背包问题(Knapsack problem)是一种组合优化的NP完全问题。问题可以描述为：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，如何选择才能使得物品的总价值最高。

问题的名称来源于如何选择最合适的物品放置于给定背包中。相似问题经常出现在商业、组合数学、应用数学、计算复杂性理论、密码学等领域中。

也可以将背包问题描述为决定性问题，即在总重量不超过W的前提下，总价值是否能达到V？它是在1978年由Merkel和Hellman提出的。

典型的背包问题包括：01背包；部分背包；完全背包；多重背包；分组背包；混合背包；等

3.9.1 01背包

翻转

01背包

有N件物品和一个容量为C的背包。第i件物品的费用是w[i]，价值是v[i]，求将哪些物品装入背包可使价值总和最大。

如果在选择装入背包的物品时，对每种物品i只有两种选择：装入背包或不装入背包，则称为01背包问题。

0-1背包问题是一个特殊的整数规划问题。

$$\max \sum_{i=1}^{n} v_i x_i \quad \begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

3.9.1 01背包

算法描述

设所给0-1背包问题的子问题的最优值为m(i,j)，即m(i,j)是背包容量为j，可选择物品为1，2，…，i时0-1背包问题的最优值。由0-1背包问题的最优子结构性质，可以建立计算m(i,j)的递归式如下。

$$m(i,j)=\begin{cases}\max\{m(i-1,j),m(i-1,j-w_i)+v_i\} & j\geq w_i\\ m(i-1,j) & 0\leq j<w_i\end{cases}$$

$$m(1,j)=\begin{cases}v_1 & j\geq w_1\\ 0 & 0\leq j<w_1\end{cases}$$

3.9.1 01背包

01背包

特点：每种物品仅有一件，可以选择放或不放。

用子问题定义状态：即m[i][j]表示前i件物品恰放入一个容量为j的背包可以获得的最大价值。则其状态转移方程便是：

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

基本上所有跟背包相关的问题的方程可由此衍生出

“将**前i件物品**放入容量为j的背包中”，若只考虑第i件物品（放或不放），那么就可以转化为一个只牵扯前i-1件物品的问题。

- 如果不放第i件物品，那么问题就转化为“前i-1件物品放入容量为j的背包中”，价值为m[i-1][j];
- 如果放第i件物品，就转化为“前i-1件物品放入剩下的容量为j-c[i]的背包中”，此时能获得的最大价值就是m[i-1][j-w[i]]+v[i] （通过放入第i件物品获得的价值）。

3.9.1 01背包

算法描述

n=5, c=10, w={2,2,6,5,4}, v={6,3,5,4,6}。

算法复杂度分析：
从m(i,j)的递归式容易看出，算法需要O(nc)计算时间。当背包容量c很大时，算法需要的计算时间较多。例如，当$c>2^n$时，算法需要$\Omega(n2^n)$计算时间。

	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7	j=8	j=9	j=10
i=0	0	0	0	0	0	0	0	0	0	0	0
i=1	0	0	6	6	6	6	6	6	6	6	6
i=2	0	0	6	6	9	9	9	9	9	9	9
i=3	0	0	6	6	9	9	9	9	9	9	14
i=4	0	0	6	6	9	9	9	10	10	13	14
i=5	0	0	6	6	9	9	12	12	15	15	15

3.9.1 01背包

参考代码

n=5, c=10, w={2,2,6,5,4}, v={6,3,5,4,6}。

```
#include<stdio.h>
#include<algorithm>
using namespace std;
const int N=5,V=10;
int w[N+1]={0,2,2,6,5,4};
int v[N+1]={0,6,3,5,4,6};
int c[N+1][V+1]={0},x[N+1]={0};
int main(){
  int i,j; i=j=1;
  for(;i<=N;i++)
    for(j=1;j<=V;j++)
      if(j<w[i]) c[i][j]=c[i-1][j];
      else c[i][j]=max(c[i-1][j],c[i-1][j-w[i]]+v[i]);
  printf("%d: ",c[N][V]);
  for(i=N,j=V;i>0;i--){
    if(c[i][j]==c[i-1][j]) x[i]=0;
    else x[i]=1,j=j-w[i];
  }
  for(j=1;j<=N;j++) printf("%d",x[j]);
}
```

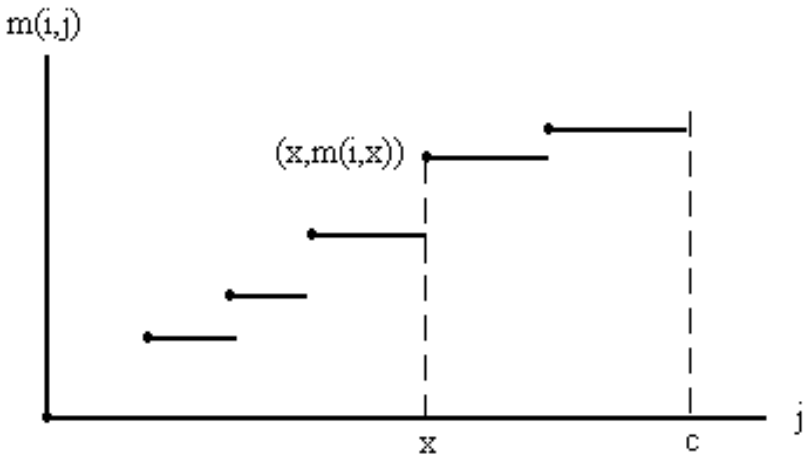
j

i

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	**9**	**9**	**9**	**9**	**9**
3	0	0	6	6	9	9	9	9	9	9	14
4	0	0	6	6	9	9	9	10	10	13	14
5	0	0	6	6	9	9	12	12	15	15	15

算法改进

由m(i,j)的递归式容易证明，在一般情况下，对每一个确定的i(1≤i≤n)，函数m(i,j)是关于变量j的阶梯状单调不减函数。跳跃点是这一类函数的描述特征。在一般情况下，函数m(i,j)由其全部跳跃点惟一确定。



对每一个确定的i(1≤i≤n)，用p[i]存储函数m(i,j)的全部跳跃点。p[i]可依计算m(i,j)的递归式递归地由p[i-1]计算，初始时p[0]={(0,0)}。

算法改进

- m(i,j)是由m(i-1,j)与$m(i-1,j-w_i)+v_i$作max运算得到的。故m(i,j)的全部跳跃点包含于m(i-1,j)的跳跃点集p[i-1]与$m(i-1,j-w_i)+v_i$的跳跃点集q[i-1]的并集中。
- 易知，$(s,t)\in q[i-1]$当且仅当$w_i\le s\le c$且$(s-w_i,t-v_i)\in p[i-1]$。
- 因此，容易由p[i-1]确定跳跃点集q[i-1]如下：

$$q[i-1]=p[i-1]\oplus(w_i,v_i)=\{(j+w_i,m(i,j)+v_i)\mid(j,m(i,j))\in p[i-1]\}$$

算法改进

- 另一方面，设(a,b)和(c,d)是p[i-1]∪q[i-1]中的2个跳跃点，则当c≥a且d<b时，(c,d)受控于(a,b)，从而(c,d)不是p[i]中的跳跃点。除受控跳跃点外，p[i-1]∪q[i-1]中的其他跳跃点均为p[i]中的跳跃点。
- 由此可见，在递归地由p[i-1]计算p[i]时，可先由p[i-1]计算出q[i-1]，然后合并p[i-1]和q[i-1]，并清除其中的受控跳跃点得到p[i]。

3.9.1 01背包

n=5，c=10，w={2,2,6,5,4}，v={6,3,5,4,6}。

p[0]={(0,0)},(w_1,v_1)=(2,6);q[0]=p[0]⊕(w_1,v_1)={(2,6)}。

p[1]={(0,0),(2,6)};q[1]=p[1]⊕(w_2,v_2)={(2,3),(4,9)}。

p[1]∪q[1]={(0,0),(2,6),(2,3),(4,9)}

p[2]={(0,0),(2,6),(4,9)}; q[2]=p[2]⊕(6,5)={(6,5),(8,11),(10,14)}

p[3]={(0,0),(2,6),(4,9),(8,11),(10,14)};

q[3]=p[3]⊕(5,4)={(5,4),(7,10),(9,13)}

p[4]={(0,0),(2,6),(4,9),(7,10),(8,11),(9,13),(10,14)}

q[4]=p[4]⊕(4,6)={(4,6),(6,12),(8,15)}

p[5]={(0,0),(2,6),(4,9),(6,12),(8,15)}

p[5]的最后的那个跳跃点(8,15)给出所求的最优值为m(5,c)=15。

3.9.1 01背包

参考代码

```
const int N = 5;
template<class Type>
int Knapsack(int n,Type c,Type v[],Type w[],int **p,int x[]);
template<class Type>
void Traceback(int n,Type w[],Type v[],Type **p,int *head,int x[]);
int main(){
        int c=10;
        int v[]= {0,6,3,5,4,6},w[]= {0,2,2,6,5,4}; //下标从1开始
        int x[N+1];
        int **p = new int *[50];
        for(int i=0; i<50; i++)        p[i] = new int[2];
        cout<<"背包能装的最大价值为："<<Knapsack(N,c,v,w,p,x)<<endl;
        cout<<"背包装下的物品编号为："<<endl;
        for(int i=1; i<=N; i++){
                if(x[i]==1) cout<<i<<" ";
        }
        cout<<endl;
        for(int i=0; i<50; i++)        delete p[i];
        delete[] p;
        return 0;
}
```

01背包

参考代码

```
int Knapsack(int n,int c,int v[],int w[],int **p,int x[]){
  int *head = new int[n+2];
  head[n+1]=0; head[n]=1;  p[0][0]=0, p[0][1]=0;
  int left = 0,right = 0,next = 1;
  for(int i=n; i>=1; i--){
    int k = left;
    for(int j=left; j<=right; j++){
      if(p[j][0]+w[i]>c) break;
      int y = p[j][0] + w[i],m = p[j][1] + v[i];
      while(k<=right && p[k][0]<y){
         p[next][0]=p[k][0]; p[next++][1]=p[k++][1]; }
      if(k<=right && p[k][0]==y)
        if(m<p[k][1]){ m=p[k][1]; k++; }
      if(m>p[next-1][1]){ p[next][0]=y; p[next++][1]=m; }
      while(k<=right && p[k][1]<=p[next-1][1]) k++;
    }
    while(k<=right){ p[next][0]=p[k][0]; p[next++][1]=p[k++][1]; }
    left = right + 1; right = next - 1; head[i-1] = next;
  }
  Traceback(n,w,v,p,head,x);
  return p[next-1][1];
}
```

3.9.1 01背包

n=5, c=10

变量说明

重量w[]= {0,2,2,6,5,4}, 价值v[]= {0,6,3,5,4,6}

head[i]：物品i的最前跳跃点在p[]中的位置为head[i]

p[][]：存储所有物品的跳跃点(p[][0],p[][1])，p[][0]为重量， p[][1]为价值

i：倒序第i个物品(n→1)

left：上一个物品的跳跃点在p[]中的开始位置

right：上一个物品的跳跃点在p[]中的结束位置

[left,right]为上个物品跳跃点在p[]中的窗口

j：遍历p[left]→p[right]中上一个物品对应的每个跳跃点，加上(如果重量不超过背包重量)当前物品的重量和价值，得到y与m

(y,m)：以某个跳跃点为基础，增加新物品后产生的新跳跃点

k：逐次遍历上一个物品的跳跃点，判断是否受控：如果不受控则产生新的跳跃点并将其存储在p[next] 中；如果受控则跳过去，不产生新的跳跃点

next：逐次从前往后产生新物品跳跃点，并将其保存在p[next]中

3.9.1 01背包

算法剖析

重量w[]= {0,2,2,6,5,4}, 价值v[]= {0,6,3,5,4,6}

初始

head

0	1	2	3	4	5	6
					1	0

p[]

0	1	2	3	4	5	6	7	8	9	10	11	…
0												
0												

left=right=k=j=0,next=1

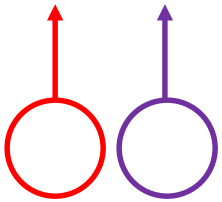
i=5

head

0	1	2	3	4	5	6
				3	1	0

p[]

0	1	2	3	4	5	6	7	8	9	10	11	…
0	0	4										
0	0	6										



left=1,right=2,next=3

3.9.1 01背包

算法剖析

重量w[]= {0,2,2,6,5,4}, 价值v[]= {0,6,3,5,4,6}

head

0	1	2	3	4	5	6
			6	3	1	0

i=4

p[ ]

0	1	2	3	4	5	6	7	8	9	10	11	...
0	0	4	0	4	9							
0	0	6	0	6	10							

left=3,right=5,next=6

head

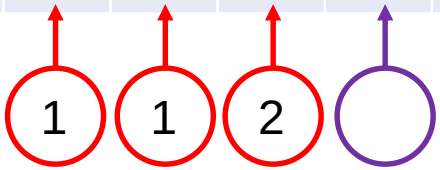
0	1	2	3	4	5	6
			6	3	1	0

i=3

p[ ]

0	1	2	3	4	5	6	7	8	9	10	11	...
0	0	4	0	4	9	0	4	9	10			
0	0	6	0	6	10	0	6	10	11			

left=6,right=9,next=10



3.9.1 01背包

参考代码

```
//x[]数组存储对应物品0-1向量,0不装入背包，1表示装入背包
template<class Type>
void Traceback(int n,Type w[],Type v[],Type **p,int *head,int x[]){
  //初始化j,m为最后一个跳跃点对应的第0列及第1列
  Type j = p[head[0]-1][0],m=p[head[0]-1][1];
  for(int i=1; i<=n; i++)     {
    x[i]=0;// 初始化数组;
    for(int k=head[i+1]; k<=head[i]-1; k++){
      if(p[k][0]+w[i]==j && p[k][1]+v[i]==m){
        x[i]=1;//物品i被装入，则x[i]置1
        j=p[k][0];// j和m值置为满足if条件的跳跃点对应的值
        m=p[k][1];// 如上例j=6,m=9
        break;//再接着判断下一个物品
      }
    }
  }
}
```

算法复杂度分析

上述算法的主要计算量在于计算跳跃点集p[i](1≤i≤n)。由于q[i-1]=p[i-1]⊕(w_i,v_i)，故计算q[i-1]需要O(|p[i-1]|)计算时间。合并p[i-1]和q[i-1]并清除受控跳跃点也需要O(|p[i-1]|)计算时间。从跳跃点集p[i]的定义可以看出，p[i]中的跳跃点相应于$x_1,\cdots,x_{i-1}$的0/1赋值。因此，p[i]中跳跃点个数不超过2^{i-1}。由此可见，算法计算跳跃点集p[i]所花费的计算时间为

$$O\left(\sum_{i=1}^{n-1}|p[i-1]|\right)=O\left(\sum_{i=1}^{n-1}2^{i-1}\right)=O(2^n)$$

从而，改进后算法的计算时间复杂性为O(2^n)。当所给物品的重量w_i(1≤i≤n)是整数时，|p[i]|≤c+1，(1≤i≤n)。在这种情况下，改进后算法的计算时间复杂性为O(min{nc,2^n})。

3.9.1 01背包

空间复杂度

01背包求解的空间复杂度可以优化到O(N)。

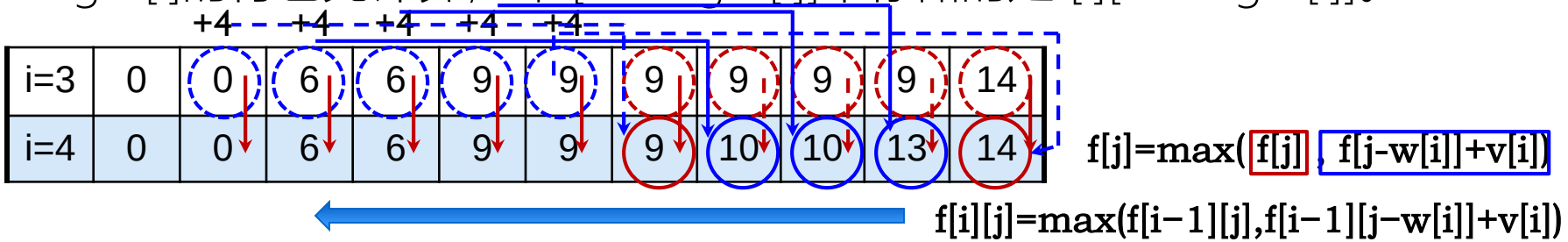
考虑滚动数组：如果只用一个数组f[0…V]，只要在每次主循环中以j=V…0的顺序推f[j]，这样能保证推f[j]时f[j−w[i]]保存的是状态f[i−1][j−w[i]]的值。于是f[j]=max(f[j],f[j−w[i]])相当于f[i][j]=max(f[i−1][j],f[i−1][j−w[i]])，因为f[j−w[i]]就相当于f[i−1][j−w[i]]。

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

3.9.1 01背包

逆序枚举的原因：n=5, c=10, w={2,2,6,5,4}, v={6,3,5,4,6}。

- 注意，我们是由第 i-1 次循环的两个状态推出第 i 个状态的，而且 v>v-weight[i]，则对于第i次循环，背包容量只有当V..0循环时，才会先处理背包容量为v的状况，后处理背包容量为 v-weight[i] 的情况。
- 具体来说，由于，在执行v时，还没执行到v - weight[i]，因此，f[v-weight[i]]保存的还是第i-1次循环的结果。即在执行第i次循环且背包容量为v时，此时的f[v]存储的是 f[i-1][v] ，此时f[v-weight[i]]存储的是f[i-1][v-weight[i]]。
- 相反，如果在执行第 i 次循环时，背包容量按照0..V的顺序遍历检测第 i 件物品是否能放，则背包容量为v时，f[v]存储的是f[i-1][v]，但此时f[v-weight[i]]存储的是f[i][v-weight[i]]。因为，v>v-weight[i]，第i次循环中v-weight[i]的背包先计算，即f[v-weight[i]]中存储的是f[i][v-weight[i]]。



3.9.1 01背包

参考代码

```
#define  V 1500
unsigned int f[V];//全局变量，自动初始化为0
unsigned int weight[10];
unsigned int value[10];
#define  max(x,y)   (x)>(y)?(x):(y)
int main(){
    int N,M;
    cin>>N;//物品个数
    cin>>M;//背包容量
    for (int i=1;i<=N; i++)  cin>>weight[i]>>value[i];
    for (int i=1; i<=N; i++)
        for (int j=M; j>=1; j--)
            if (weight[i]<=j) f[j]=max(f[j],f[j-weight[i]]+value[i]);
    cout<<f[M]<<endl;//输出最优解

}
```

3.9.1 01背包

小结

01背包是最基本的背包问题，包含了背包问题中设计状态、方程的最基本思想，其他类型背包问题往往也可以转换成01背包问题求解。

针对性练习

Luogu 2925 干草出售

Luogu 1616 疯狂的采药

HDU 3466 Proud Merchants

3.9.2 完全背包

有N种物品和一个容量为V的背包，每种物品都有无限件可用。第i种物品的重量是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费用总和不超过背包容量，且价值总和最大。

将01背包问题的基本思路加以改进，得到状态与转移方程：

$$f[i][j]=max(f[i-1][j-k*w[i]]+k*v[i]) \mid 0<=k*w[i]<=j$$

这跟01背包问题一样有O(VN)个状态需要求解，但求解每个状态的时间已经不是常数了，求解状态f[i][j]的时间是O(V/w[i])，总的复杂度是超过O(VN)的。

3.9.2 完全背包

一般优化思路：

1.若两件物品i、j满足w[i]<=w[j]且v[i]>=v[j]，则将物品j去掉，不用考虑。任何情况下都可将价值小重量大的物品j换成物美价廉的i，得到至少不会更差的方案。对于随机生成的数据，这个方法往往会大大减少物品的件数，从而加快速度。然而这个并不能改善最坏情况的复杂度，因为针对特别设计的数据，有可能一件物品也去不掉。这个优化可以简单的$O(N^2)$实现。

2.首先将费用大于V的物品去掉，然后计算出费用相同的物品中价值最高的是哪个，可以O(V+N)地完成这个优化。

3.9.2 完全背包

把完全背包问题转化为01背包：

考虑到第i种物品最多选V/w[i]件，于是可以把第i种物品转化为V/w[i]件重量与价值均不变的物品，然后求解这个01背包问题。将一种物品拆成多件物品。

更高效的转化方法：

把第i种物品拆成费用为$w[i]*2^k$、价值为$v[i]*2^k$的若干件物品，其中k满足$w[i]*2^k<=V$，运用二分[倍增]思想。

因为不管最优策略选几件第i种物品，总可以表示成若干个2^k件物品的和。这样把每种物品拆成O(log(V/w[i]))件物品，是一个很大的改进。

3.9.2 完全背包

O(VN)的算法

完全背包问题的一般转移方程可以写为：

$$dp[i][j]=max(dp[i-1][j-k*w[i]]+k*v[i])|0<=k*w[i]<=j$$

其中：f[i-1][j]表示不选第i种物品；后者则表示第i种物品至少选择1件。

第1种滚动数组（二维）： 求前i件物品最大价值只需前i-1件物品的最大值，即求dp[2][v]只需dp[1][v]，因此只需定义二维数组dp[2][v]，此时：dp[i%2][j]=max(dp[(i-1)%2][j],dp[(i-1)%2][j-w[i]]+v[i])

```
for (int i = 1; i <= n; i++)
    for (int j = w[i]; j <= V; j++)
        for ( int k=0; k*w[i]<=j; k++)
            dp[i%2][j]=max(dp[(i-1)%2][j],dp[(i-1)%2][j-k*w[i]]+k*v[i]);
```

3.9.2 完全背包

O(VN)的算法

- 第2种滚动数组：
- 一维数组 `f[i][j]=max{f[i-1][j],f[i][j-w[i]]+v[i]}`

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3
2	0	0	3	3	6	6	6	6
3	0	0	3	3	6	6	9	9

背包体积

物品数量

- 当前行改动的值是由上一行改动的值产生的，当前行没改动的值会保留上一行的值；

0	1	2	3	4	5	6	7	背包体积
0	0							先计算背包体积为0和1的情况
0	0	3	3					再计算体积为2和3的情况，是基于上一次计算的
0	1	3	3	6	6			
0	0	3	3	6	6	9	9	

所以此时的状态转移方程是`dp[j]=max(dp[j],dp[j-weight[i]]+value[i])`

3.9.2 完全背包

O(VN)的算法

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

这个代码与01背包的代码只有j的循环次序不同而已。

01背包中要按照j=V…0的逆序来循环，是因为要保证第i次循环中的状态f[i][j]是由状态f[i−1][j−w[i]]递推而来。正是为了保证每件物品只选一次，保证在考虑“选入第i件物品”这件策略时，依据的是一个未选入第i件物品的f[i−1][j−w[i]]。

而完全背包的特点是每种物品可选无限件，所以在考虑“加选一件第i种物品”这种策略时，却正需要一个可能已选入第i种物品的f[i][j−w[i]]（至少1个），所以就可以并且必须采用j=w[i]…V的顺序循环。

3.9.2 完全背包

小结

完全背包问题 也 是基础背包问题，它有两个状态转移方程，分别在“基本思路”以及“O(VN)的算法” 中给出。这两个状态转移方程 要 仔细地体会，要弄明白它们是怎么得出来的，最好能够自己想一种得到这些方程的方法。事实上，对每一道动态规划题目都思考其方程的意义以及如何得来，是加深对动态规划的理解、提高动态规划功力的好方法。

针对性练习

HDU 1114 Piggy-Bank

Luogu 1853 投资的最大效益

3.9.3 多重背包

有N种物品和一个容量为V的背包。第i种物品最多有p[i]件可用，每件费用是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费用总和不超过背包容量，且价值总和最大。

算法

在完全背包状态转移方程的基础上，对第i种物品采取p[i]+1种策略：取0件，取1件……取p[i]件。

令f[i][j]表示前i种物品恰放入一个容量为j的背包的最大权值，则有状态转移方程：

$$f[i][j]=\max\{f[i-1][j-k*w[i]]+k*v[i]\mid 0<=k<=p[i]\}$$

复杂度是O(V*Σp[i])。

3.9.3 多重背包

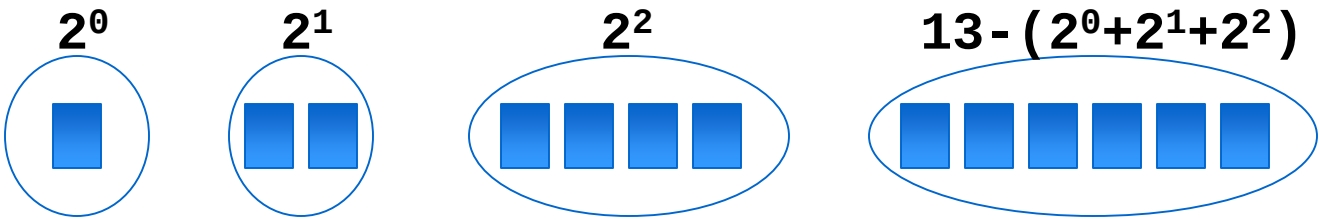
转化为01背包问题

把第i种物品换成p[i]件01背包中的物品，则得到了物品数为Σp[i]的01背包问题，使得原问题中第i种物品可取的每种策略——取0…p[i]件——均能等价于取若干件代换以后的物品直接求解，复杂度仍然是O(V*Σp[i])。

二进制优化：

将第i种物品分成若干件物品，其中每件物品有一个系数，这件物品的费用和价值均是原来的费用和价值乘以这个系数。使这些系数分别为1,2,4,...,2^{k-1},p[i]-2^k+1，且k是满足p[i]-2^k+1>0的最大整数。例如，如果p[i]为13，就将这种物品分成系数分别为1,2,4,6的四件物品。分成的这几件物品的系数和为p[i]，表明不可能取多于p[i]件的第i种物品。

13



3.9.3 多重背包

转化为01背包问题的二进制优化：

另外这种方法也能保证对于0…p[i]间的每一个整数，均可以用若干个系数的和表示，这个证明可以分0…2^{k-1}和2^k…p[i]两段来分别讨论得出。这样就将第i种物品分成了O(log(p[i]))种物品，将原问题转化为了复杂度为O(V*Σlog(p[i]))的01背包问题，是很大的改进。

```
for(int i=1; i<=n; i++) {
    int num=min(p[i], V/w[i]);
    for(int k=1; num>0; k<<=1) {
        if(k>num) k=num;
        num-=k;
        for(int j=V; j>=w[i]*k; j--)
            f[j]=max(f[j], f[j-w[i]*k]+v[i]*k);
    }
}
```

3.9.3 多重背包

O(VN)的算法：

多重背包问题同样有O(VN)的算法，虽然采用二进制表示的形式已经使复杂度降低很多，但是根据基本算法的状态转移方程，应用单调队列的方法仍可使复杂度进一步降低（每个状态O(1)时间）。

单调队列是单调递增或递减的队列，满足：1）必须从队头到队尾的严格单调性；2）排在队列前面的比排在队列后面的要先进队。常用于求区间移动（滑动窗口）时动态维护区间的最值。

分析多重背包转移方程：f[i][j]=max(f[i-1][j],f[i-1][j-k*w[i]]+k*v[i])，再结合二进制表示形式，单调队列优化的主要思想就是分组更新，因为w[i]是成倍增加的。f[i-1][j]只会更新f[i-1][j+k*w[i]](这里是从前往后看的，所以是+）。对于当前为w的体积，可以按照余数将它分为w组，也就是0...w-1。同一个剩余系的数在一组。如在模3意义下，1,4,7,10是一组，2,5,8,11是一组，3,6,9,12是一组。每组的转移互不影响，也就是单独转移。

3.9.3 多重背包

O(VN)的算法：

针对多重背包的最原始的状态转移方程，令c[i]=min(p[i]，j/w[i])，f[i][j]=max(f[i-1][j-k*w[i]]+k*v[i]) (0<=k<=c[i]) 这里的k是指取第i种物品k件。

如果令 a=j/w[i] , b=j % w[i] 那么 j=a*w[i]+b.则：

f[i][j]=max(f[i-1][b+(a-k)*w[i]]-(a-k)*v[i])+a*v[i] (a-c[i]<=a-k<=a)

令k'=a-k，则其表示取第 i 种物品的件数比 a 少几件。

可以发现，f[i-1][b+k'*w[i]]-k'*v[i] 只与k'有关，而这个k'是一段连续的。我们要做的就是求出 f[i-1][b+k'*w[i]]-k'*v[i] 在 k'取可行区间内时的最大值，而这就可以使用单调队列优化。

3.9.3 多重背包

```
const int M=20010,N=1010;
int f[M],g[M],que[M],n,m,v,w,s;
int main(){
  cin>>n>>m;
  for(int i=1;i<=n;i++){        //遍历n种物品
    cin>>v>>w>>s;
    memcpy(g,f,sizeof(f));      //备份f数组，以便正推
    for(int r=0;r<v;r++){       //遍历余数（分组）
       int head=0,tail=-1;
       for(int k=0;r+k*v<=m;k++){     //遍历每个分组
         if(head<=tail && k-que[head]>s) head++;  //超出最大长度，队首出队
         while(head<=tail && g[r+k*v]-k*w>=g[r+que[tail]*v]-que[tail]*w)
              tail--;           //新值比队尾大，队尾出队
         que[++tail]=k;         //新值入队
         f[r+k*v]=g[r+que[head]*v]+(k-que[head])*w;  //更新f[]
       }
    }
  }
  cout<<f[m]<<endl;
  return 0;
}
```

3.9.3 多重背包

小结

这是将一个算法的复杂度由O(V*Σp[i])改进到O(V*Σlog(p[i]))的过程，以及O(VN)算法。需要特别注意"拆分物品"的思想和方法。

针对性练习

HDU 1059 Dividing

Luogu P1776 宝物筛选

3.9.4 混合背包

如果将前面三个背包混合起来，也就是说，有的物品只可以取一次（01背包），有的物品可以取无限次（完全背包），有的物品可以取的次数有一个上限（多重背包），应该怎么求解呢?

01背包与完全背包的混合

考虑到在01背包和完全背包中给出的伪代码只有一处不同，故如果只有两类物品：一类物品只能取一次，另一类物品可以取无限次，那么只需在对每个物品应用转移方程时，根据物品的类别选用顺序或逆序的循环即可，复杂度是O(VN)。

再加上多重背包

如果再加上有的物品最多可以取有限次，那么原则上也可以给出O(VN)的解法：遇到多重背包类型的物品用单调队列解即可。但用多重背包中将每个这类物品分成O(log(p[i]))个01背包的物品的方法也已经很优了。当然，更清晰的写法是调用前面给出的三个相关过程。

3.9.4 混合背包

```
for i=1..N
    if 第i件物品是01背包  ZeroOnePack(c[i],w[i])
    else if 第i件物品是完全背包 CompletePack(c[i],w[i])
    else if 第i件物品是多重背包 MultiplePack(c[i],w[i],n[i])
```

```
procedure ZeroOnePack(cost,weight)
    for v=V..cost
        f[v]=max{f[v],f[v-cost]+weight}
```

```
procedure CompletePack(cost,weight)
    for v=cost..V
        f[v]=max{f[v],f[v-c[i]]+w[i]}
```

```
procedure MultiplePack(cost,weight,amount)
    if cost*amount>=V
        CompletePack(cost,weight); return;
    integer k=1
    while k<num
        ZeroOnePack(k*cost,k*weight)
        amount=amount-k; k=k*2;
    ZeroOnePack(amount*cost,amount*weight)
```

3.9.4 混合背包

小结

困难的问题往往都是由简单的问题叠加而来。但只要领会三种基本背包问题的思想，就可以做到把困难的混合背包拆分成三种情况来解决。

针对性练习

Luogu P1833 樱花

HDU 3535 AreYouBusy

3.9.6 背包问题的代码示例

```
/*---------0-1背包----------*/
int knapsack01(int n, int V) {
    memset(f, 0xc0c0c0c0, sizeof f); f[0] = 0; //需要装满
    memset(f, 0, sizeof f); //不需要装满
    for (int i = 1; i <= n; i++)
        for (int j = V; j >= w[i]; j--)
            f[j] = max(f[j], f[j - w[i]] + v[i]);
    return f[V];
}

/*----------完全背包----------*/
int Fullbackpack(int n, int V) {
    for (int i = 1; i <= n; i++)
        for (int j = w[i]; j <= V; j++)
            f[j] = max(f[j], f[j - w[i]] + v[i]);
    return f[V];
}
```

3.9.6 背包问题的代码示例

```
/*-------多重背包二进制拆分-------*/
int number[A];
int MultiplePack1(int n, int V) {
    for (int i = 1; i <= n; i++) {
        int num = min(number[i], V / w[i]);
        for (int k = 1; num > 0; k <<= 1) {
            if (k > num) k = num;
            num -= k;
            for (int j = V; j >= w[i] * k; j--)
            f[j] = max(f[j], f[j - w[i] * k] + v[i] * k);
        }
    }
    return f[V];
}
```

3.9.6 背包问题的代码示例

```
/*------------多重背包单调队列优化------------*/
void MultiPack(int p, int w, int v) {
    for (int j = 0; j < cost; j++) {
        int head = 1,tail = 0;
        for (int k = j, i = 0; k <= V / 2; k += w, i++) {
            int r = f[k] - i * v;
            while (head <= tail and r >= q[tail].v) tail--;
            q[++tail] = node(i, r);
            while (q[head].id < i - num) head++;
            f[k] = q[head].v + i * v;
        }
    }
}
```

3.9.7 背包问题关键总结

1）01背包：

-转移方程：f[i][j]=max{f[i-1][j],f[i-1][j-w[i]]+v[i]}

-优化后：f[j]=max{f[j],f[j-w[i]]+v[i]}，逆序

2）完全背包：

-转移方程:f[i][j]=max{f[i-1][j-k*w[i]]+k*v[i]|0<=k*w[i]<=C}

-优化后：f[j]=max{f[j],f[j-w[i]]+v[i]}，顺序

3）多重背包：

-转移方程：f[i][j]=max{f[i-1][j-k*w[i]]+k*v[i]|0<=k<=p[i]}

-优化后：f[i][j]=max(f[i-1][b+k'*v[i]]-k'*w[i])+a*w[i]

(a-c[i]<=k'<=a,c[i]=min(p[i],j/v[i]),a=j/v[i],b=j%v[i])

4）混合背包