背包九讲

《背包九讲》

前言：

背包问题：有n件物品，每件物品有一定的价值，获取每件物品都需要一定的代价，背包问题就是在遵守一定的规则的情况下，获取最高的价值。

1，01背包

最基本的背包问题，其规则为每件物品要么选，要么不选。

定义状状态数组dp[i][j]表示前i个物品，当背包的容量为j时，背包可以容纳的最大价值。第i件物品可以不选可以选，如果不选的话，就相当于从当背包容量为j的时候，前i-1件物品里价值最大的物品。也就是说dp[i][j]=dp[i-1][j]。如果选择的话，（设物品i所占的容量为w[i]）从背包中拿出w[i]的容量用来装物品i，那么前i-1件物品所占用的容量就为j-w[i]。那么dp[i][j]=dp[i-1][j-w[i]]+v[i]（v[i]表示的是物品i的价值）

关于初始化。这里的初始化有一些讲究。和我们如何定义数组dp[i][j]的含义有关。假如说我们定义dp[i][j]为钱i件物品，当背包容量为j时可获得的价值，那么开始时候dp[i][j]全部初始化为0就可以了，但是如果问的是前i件物品，容量为j时，恰好可以装多少，也就是说我们必须装满，如果不能装满的话，价值再大也没有用。 这两种定义有什么区别呢？看个例子，设有n件物品， 并且可获得的最大价值为x，需要的背包容量为k，而你的背包总容量为k+1。对于第一种情况dp[n][k+1]=dp[n][k]=x，但是对于第二种情况就不一定了，因为我们要求必须装满才行。

Code:

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

01背包的空间优化

01背包可以从二维优化为一维的背包，当然可以直接通过滚动数组来降维，但还有一更优的优化方式，定义dp[i]为背包容量为i时可获取的最大的价值。如果不选的话，还是考虑到第i个物品。当容量为j时，此时的最大价值为dp[j]，注意我们这里是考虑到第i个物品，所以这里的dp[j]的最大价值应该是前i-1个物品，背包容量为j时的最大价值。所以如果我们不选第i个物品的话dp[j]直接不用做任何操作，如果选的话dp[j-weight[i]]+value[i]

但是这里就出现了一个问题，该怎么保证dp[j-weight[i]]是前i-1个物品的最大价值呢？可以倒着从m到weight[i]进行遍历。（挺难想的）

```
        for(int i=1;i<=n;i++){
        for(int j=m;j>=weight[i];j--)
            dp[j]=max(dp[j],dp[j-weight[i]]+value[i]);
    }
    return dp[m];
```

2，完全背包

规则：每件物品可以选择任意次数。

完全背包和01背包的区别就是完全背包没件物品可以选择任意次，在代码上和01背包的区别也是比较小的，只需要再加一个for，看当前物品选择多少个才是最合适的。这种做法比较简单直观，但是时间复杂度是O(n^3)，空间复杂度是O（n^2）

```
        for(int i=1;i<=n;i++){
        for(int j=0;j<=m;j++)
            for(int k=0;k*v[i]<=j;k++){
                dp[i][j]=max(dp[i][j],dp[i-1][j-k*v[i]]+w[i]*k);
            }
    }
    return dp[n][m];
```

显然这种复杂度并不好，首先优化空间复杂度，可以把通过类似于01背包的方法把空间复杂度优化成一维的。再优化时间复杂度，假设考虑到第i件物品的容量为j时的状态，即dp[j]，这里的dp[j]保存的值是前i-1件物品的最大价值。该怎么转移呢？因为这里对物品选择的个数是没有限制的，所以转移的时候，如果选择1个，那么就从i-1个物品进行转移，选择两个就从再转移一次。。。所以说转移的dp[j]可以是上一层的也可以是这一层的。，所以遍历的时候，体积正序遍历就可以了。

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

3，多重背包

规则：每件物品最多选k次

多重背包也是以01背包为基础的背包问题，按照朴素的做法就是枚举该物品的个数，思路和完全背包的朴素枚举基本上是完全一样的，只需要加一个物品个数限制就可以了。

```
        for(int i=1;i<=n;i++){
        for(int j=0;j<=m;j++)
            for(int k=0;k*v[i]<=j&&k<=num[i];k++){
                dp[i][j]=max(dp[i][j],dp[i-1][j-k*v[i]]+w[i]*k);
            }
    }
    return dp[n][m];
```

两种优化方法

第一种是我比较喜欢的优化方法， 叫做二进制优化，二进制真的是一个特别神奇的东西。假如说num=10.通过二进制拆分，可以拆分为1+2+4+3。并且这四个数可以组成小于等于10的任意一个数字。

因此我们可以把10个物品拆成四份，每一份为的数目为1,2,4,3。接下来就按照01背包处理就可以了。时间复杂度大约为o(m) （m是容量，n是物品的个数，c[i]指的是物品i的个数）

Code

```
int solve1(){
    int n,m;
    cin>>n>>m;
    int pos=0,x,y,z;
    for(int i=1;i<=n;i++){
        cin>>x>>y>>z;
        for(int j=1;z>=j;j<<=1){
            z-=j;
            volume[pos]=j*x;
             value[pos++]=j*y;
         }
         if(z>0) {
             value[pos]=z*y;
             volume[pos++]=z*x;
         }
     }
     for(int i=0;i<pos;i++){
         for(int j=m;j>=volume[i];j--){
             dp[j]=max(dp[j],dp[j-volume[i]]+value[i]);
         }
     }
     return dp[m];
 }
```

第二种优化是借用一种数据结构—单调队列优化，这种优化是比较难的，《男人八题》中就有一个单调队列优化的裸多重背包问题。

待补...

4，混合背包

将01背包，完全背包，多重背包三类背包放在一起。

这类问题的解决方法就是属于哪种背包就按照哪种背包的方式算，这类问题算不上是新问题

例题 https://www.acwing.com/problem/content/7/

Code

```
int dp[N]; //dp[i]表示当容量为i时，可获取的最大价值
int volume[N],value[N];//体积和价值
bool mark[N];//标记哪个物品属于哪类背包问题 mark[i]=1 表示物品i为完全背包问题，0表示01背包问题
int solve(){
    int n,m;
    cin>>n>>m;//n件物品，容量限制为m
    int x,y,z;
    int pos=0;
    for(int i=1;i<=n;i++){
         cin>>x>>y>>z;
         if(z==0){
             mark[pos]=1;
             volume[pos]=x;
             value[pos++]=y;
         }
         else if(z==-1){
             volume[pos]=x;
             value[pos++]=y;
         }
         else {//如果是多重背包问题，通过二进制拆分成01背包
             for(int j=1;z>=j;j<<=1){
                 z-=j;
                 volume[pos]=j*x;
                 value[pos++]=j*y;
             }
             if(z>0){
                 volume[pos]=z*x;
                 value[pos++]=z*y;
             }
         }
     }
     for(int i=0;i<pos;i++){
         if(mark[i]) {//完全背包正序，01背包逆序
             for(int j=volume[i];j<=m;j++)
                 dp[j]=max(dp[j],dp[j-volume[i]] + value[i]);
         }
         else{
```

```
            for(int j=m;j>=volume[i];j--){
                dp[j]=max(dp[j],dp[j-volume[i]] + value[i]);
            }
        }
    }
    return dp[m];
}
```

5，二维费用背包

例题：https://www.acwing.com/problem/content/8/

这里的代价是二维的，比如说同时有重量和体积的限制。

定义状态数组dp[i][j][k]表示考虑到第i个物品，当背包的容量为j,体积为k时可以获得的最大价值。

状态转移:考虑到第i个物品，如果选的话：

dp[i][j][k]=dp[i-1][j-volume[i]][k-weight[i]]+value[i];

如果不选的话直接就dp[i][j][k]=dp[i-1][j][k];

我们可以对空间进行优化，类似于01背包的优化，可以优化到二维.

```
int dp[N][N];
int volume[N],weight[N],value[N];
int solve(){
    int n,m,v;
```

```
        cin>>n>>v>>m;
        for(int i=1;i<=n;i++)
            cin>>volume[i]>>weight[i]>>value[i];
        for(int i=1;i<=n;i++){
            for(int j=m;j>=weight[i];j--){
                for(int k=v;k>=volume[i];k--){
                    dp[j][k]=max(dp[j][k],dp[j-weight[i]][k-volume[i]]+value[i]);
                }
            }
        }
        return dp[m][v];
    }
```

6，分组背包

在一个组内，只能选择一类物品，该类物品的选择可以满足01背包，完全背包或者多重背包的原则。

例题：https://www.acwing.com/problem/content/9/

在01背包的基础上，对物品进行分组，每个组最多选择一个物品。定义状态dp[i][j]为考虑到第i组，当前背包容量为j时的最大价值。状态转移也比较简单，如果选择，直接枚举第i组的所有状态，dp[i][j]=dp[i-1][j-volume[i]]+value[i]，如果不选的dp[i][j]=dp[i-1][j]。所以该类问题较01背包问题只是增加了一个for，用来枚举每个组的物品。

空间优化后的Code：

```
int s[N], v[N][N], w[N][N];//分别表示每一组的个数，v[i][j]表示第i组第j个的volume,w[i][j]表示的是....value
int dp[N];
int solve(){
    int n,m;
    cin >> n >> m;
    for (int i = 1; i <= n; ++i) {
        cin >> s[i];
        for (int j = 1; j <= s[i]; ++j) {
            cin >> v[i][j] >> w[i][j];
        }
    }
```

```

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

7，背包问题的具体方案

该类问题可以归结为对背包问题的路径的记录。

对于这类问题有两种方法来解决。

1　我们可以用一个数组path[i][j]来记录当第i个物品当容量为j时的选择。以01背包为例子。考虑到第i个物品，当背包容量为j的时候，第i个物品如果选的话，

dp[j]<dp[j-volume[i]]+value[i]，　path[i][j]=1

否则的话，第i个物品可以不选，

假设有n个物品，背包的总容量为m，那么最终的状态一定dp[n][m],如果第n个物品选了的话，path[n][m]=1，那么上一个状态就是path[n-1][m-volume[n]]...如果没选的话，　上一个状态为path[n-1][m].

如果是完全背包的话，如果第n个物品选了的话，上一个状态就是path[n][m-volume[i]]，这里是n，不是n-1，因为第n个物品可能选了多次....

完全背包---code:

```
for (int i = 1; i <= n; i++)
    for (int j = w[i]; j <= m; j++)
       if (dp[j] < dp[j - w[i]] + v[i]){
           dp[j] = dp[j - w[i]] + v[i];
           path[i][j]=1;//表示当容量为j的时候选择了第i个物品。
```

```
        }

int i=n,j=m;
while(i>=1&&j){
    if(path[i][j]){
        cout<<i<<" ";
        j-=w[i];
    }
    else i--;
}
```

多重背包---code：

```
for (int i = 1; i <= n; i++)
    for (int k = 1; k <= cnt[i]; k++)
        for (int j = 10; j >= w[i]; j--)
            if (dp[j] < dp[j - w[i]] + v[i]){
                dp[j] = dp[j - w[i]] + v[i];
                path[i][j] = 1;
            }
int i=n,j=m;
while(i>=1&&j){
    if(path[i][j]&&cnt[i]){
        cnt[i]--;
        j-=w[i];
        cout<<i<<" ";
    }
    else i--;
}
```

2 如果第i个物品可以被选，那么它一定满足两个条件，首先是背包要有足够多的容量，然后是满足dp[i-1][j]<=dp[i-1][j-weight[i]]+value[i]，即选第i个物品可以获得更高的利益，我们可以根据这一原则来寻找符合条件的路径。

Code：

ACwing上的一个例题： https://www.acwing.com/problem/content/12/

```
for(int i=n;i>=1;i--){
    for(int j=m;j>=0;j--){
        if(j>=weight[i]) dp[i][j]=max(dp[i+1][j],dp[i+1][j-weight[i]]+value[i]);
        else dp[i][j]=dp[i+1][j];
    }
}

for(int i=1;i<=n&&m>=0;i++){
    if(weight[i]<=m&&dp[i+1][m]<=dp[i+1][m-weight[i]]+value[i]){
        m-=weight[i];
        cout<<i<<" ";
    }
}
```

8，背包问题的方案数。

这类问题又可以分为两种，第一种是当获得最大价值时的方案数目，第二种是当装有一定体积时的方案数目。

1、定义状态数组dp[i]表示当容量为i时可以获得的最大价值，cnt[i]当容量为i时且获得最大价值的方案数。

首先是初始化,dp[i]（0àm）=0，和01背包初始化是一样的，cnt[i]的初始化全部为1，表示不装任何物品。然后状态转移，当dp[j]=dp[j-weight[i]]+value[i]的时候，此时选与不选获得的价值都是一样的，所有cnt[j]=cnt[j]+cnt[j-weight[i]]，当后者大的时候，cnt[j]=cnt[j-weight[i]]，否则cnt[j]=cnt[j].

Code (01背包为例)

```
for(int i=1;i<=n;i++) cin>>weight[i]>>value[i];
for(int i=0;i<=m;i++) cnt[i]=1;

for(int i=1;i<=n;i++){
    for(int j=m;j>=weight[i];j--){
        if(dp[j]==dp[j-weight[i]]+value[i]){
            cnt[j]=(cnt[j]+cnt[j-weight[i]])%mod;
        }
        else if(dp[j]>dp[j-weight[i]]+value[i]){
            cnt[j]=cnt[j]%mod;
        }  //这一步可以省略
        else {
            dp[j]=dp[j-weight[i]]+value[i];
            cnt[j]=cnt[j-weight[i]]%mod;
        }
    }
}
cout<<cnt[m]%mod<<endl;
```

2、定义状态数组dp[i]表示当装有容量为i的物品时的方案数目,初始化的时候,dp[0]=1.其他的都是0.

状态转移方程dp[j]=dp[j]+dp[j-weight[i]].

Code :(01背包为例)

```
for(int i=1;i<=n;i++){
    for(int j=m;j>=weight[i];j--){
        dp[j]=dp[j]+dp[j-weight[i]];
    }
}
```

9，有依赖性的背包问题

当选择一件物品的时候，必须执行另外一种规则。

有依赖的背包问题，也可以说是树形背包问题，这类问题是比较难的，我学的也不扎实，唉，算了，放俩例题吧~。

推荐几个例题：https://www.luogu.com.cn/problem/P2014 （洛谷 选课）

Code：

```
#include<bits/stdc++.h>
using namespace std;
const int N=300+7;
struct stu
{
    int nxt,to;
}edge[N];
int n,m;
int head[N],tol=1;
int value[N];
int volume[N];
void add(int u,int v){
    edge[tol].to=v;
    edge[tol].nxt=head[u];
    head[u]=tol++;
}
int dp[N][N];
void dfs(int u){
    // cout<<u<<endl;
    for(int i=head[u];i;i=edge[i].nxt){
        int y=edge[i].to;
        dfs(y);
        for(int j=m-volume[u];j>=0;j--){
            for(int i=0;i<=j;i++){
                dp[u][j]=max(dp[u][j],dp[u][j-i]+dp[y][i]);
            }
        }
    }
    for(int i=m;i>=volume[u];i--){
        dp[u][i]=dp[u][i-volume[u]]+value[u];
    }
```

```
    dp[u][0]=0;
}
int main(int argc, char const *argv[]){

    cin>>n>>m;
    int s;
    for(int i=1;i<=n;i++){
        cin>>s>>value[i];
        add(s,i);
        volume[i]=1;
    }
    dfs(0);
    cout<<dp[0][m]<<endl;

    return 0;
}
```

ACwing10. 有依赖的背包问题https://www.acwing.com/problem/content/10/

```
#include<bits/stdc++.h>
using namespace std;
const int N=1e2+7;
struct stu{
    int to,nxt;
}edge[N];
int head[N],tol=1,n,m;
void add(int u,int v){
    edge[tol].to=v;
    edge[tol].nxt=head[u];
```

```
    head[u]=tol++;
}
int volume[N],value[N];
int dp[N][N];
int mark[N];
void dfs(int x){
    for(int i=head[x];i;i=edge[i].nxt){
        int y=edge[i].to;
        dfs(y);
        for(int j=m-volume[x];j>=0;j--){
            for(int k=0;k<=j;k++){
                dp[x][j]=max(dp[x][j],dp[x][j-k]+dp[y][k]);
            }
        }
    }
    for(int i=m;i>=volume[x];i--)
        dp[x][i]=dp[x][i-volume[x]]+value[x];
    for (int i = 0; i < volume[x]; i++) dp[x][i] = 0;
}
int main(int argc, char const *argv[]){
    cin>>n>>m;
    int z,root;
    for(int i=1;i<=n;i++){
        cin>>volume[i]>>value[i]>>z;
        if(z==-1) root=i;
        else add(z,i);
    }
    dfs(root);
    cout<<dp[root][m]<<endl;
    return 0;
}
```