0.1 背包DP

多重背包

多重背包也是0-1背包的一个变式。与0-1背包的区别在于每种物品有k_i个，而非一个。一个很朴素的想法就是：把「每种物品选k_i次」等价转换为「有k_i个相同的物品，每个物品选一次」。这样就转换成了一个0-1背包模型，套用上文所述的方法就可已解决。状态转移方程如下：

```
for(int i=0;i<n;i++)
{
    for(int j=v;j>=0;j--)
    {
        for(int k=0;k<=s[i]&&k*w[i]<=j;k++)
        dp[j]=max(dp[j],dp[j-k*w[i]]+k*val[i]);
    }
}
```

二进制分组优化

对于一个数量为M_i件的物品，将其分成若干见，令这些系数分别为$1,2,2^2...2^{k-1},M_i-2^k+1$，继而转化为01背包，就使得本为$O(V\sum M_i)$时间复杂度的问题变为了$O(V\sum logM_i)$。

单调队列优化

对于价值为v，重量为w，数量为k的物品，$dp[m]$的状态只与$dp[m-w]+v,dp[m-2*w]+2*v,\cdots,dp[m-k*v]+k*v$有关。所以我们将这个问题等价与$w$个同模的单调队列，在其上进行长度为$k$的滑动窗口问题。每次加入队列的值为$dp[j+k*v]-k*w$。时间复杂度$O(NV)$。

$$dp[j]=dp[j]$$
$$dp[j+v]=max(dp[j],dp[j+v]-w)+w$$
$$dp[j+2v]=max(dp[j],dp[j+v]-w,dp[j+2v]-2w)+2w$$
$$dp[j+3v]=max(dp[j],dp[j+v]-w,dp[j+2v]-2w,dp[j+3v]-3w)+3w$$

```
for(int i=1;i<=N;i++)
{
    scanf("%d%d%d",&v,&w,&s);
    int u=i%2;
    for(int j=0;j<=V;j++) dp[u][j]=dp[u^1][j];
```

```
    for(int j=0;j<v;j++)
    {
        int head=0,tail=-1;
        for(int k=j;k<=V;k+=v)
        {
            if(head<=tail && k-s*v>q[head]) ++head;
            while(head<=tail&&dp[u^1][q[tail]]-q[tail]/v*w<=dp[u^1][k]-k/v*w)
            --tail;
            if(head<=tail)
            dp[u][k]=max(dp[u][k],dp[u^1][q[head]]+(k-q[head])/v*w);
            q[++tail]=k;
        }
    }
}
printf("%d\n",dp[N%2][V]);
```

混合背包

混合背包就是将前面三种的背包问题混合起来，有的只能取一次，有的能取无限次，有的只能取k次。

```
for循环物品种类()
{
    if是背包(0-1) 套用背包代码0-1;
    else if是完全背包() 套用完全背包代码;
    else if是多重背包() 套用多重背包代码;
}
```

二维费用背包

有n个任务需要完成，完成第i个任务需要花费t_i分钟，产生c_i元的开支。现在有T分钟时间，W元钱来处理这些任务，求最多能完成多少任务。

选一个物品会消耗两种价值（经费、时间），只需在状态中增加一维存放第二种价值即可。

```
for(int k=1;k<=n;k++)
{
    for(int i=m;i>=mi;i--)    // 对经费进行一层枚举
        for(int j=t;j>=ti;j--)  // 对时间进行一层枚举
```

```
            dp[i][j]=max(dp[i][j],dp[i-mi][j-ti]+1);
}
```

分组背包

有n件物品和一个大小为m的背包，第i个物品的价值为v_i，体积为w_i。同时，每个物品属于一个组，同组内最多只能选择一个物品。求背包能装载物品的最大总价值。

这种题怎么想呢？其实是从「在所有物品中选择一件」变成了「从当前组中选择一件」，于是就对每一组进行一次0-1背包就可以了。再说一说如何进行存储。我们可以将$t_{k,i}$表示第k组的第i件物品的编号是多少，再用cnt_k表示第k组物品有多少个。

```
for(int k=1;k<=ts;k++) //循环每一组
    for(int i=m;i>=0;i--) //循环背包容量
        for(int j=1;j<=cnt[k];j++) //循环该组的每一个物品
        if(i>=w[t[k][j]])
        dp[i]=max(dp[i],dp[i-w[t[k][j]]]+c[t[k][j]]); //像0-1背包一样状态转移
```

有依赖的背包

对于一个主件和它的若干附件，有以下几种可能：只买主件，买主件+某些附件。因为这几种可能性只能选一种，所以可以将这看成分组背包。如果是多叉树的集合，则要先算子节点的集合，最后算父节点的集合。

泛化物品

这种背包，没有固定的费用和价值，它的价值是随着分配给它的费用而定。在背包容量为V的背包问题中，当分配给它的费用为v_i时，能得到的价值就是$h(v_i)$。这时，将固定的价值换成函数的引用即可。

杂项

输出方案

```
for(int i=N;i>=1;i--)
{
    for(int j=0;j<=V;j++)
```

```
    {
        f[i][j]=f[i+1][j];
        if(j>=v[i]) f[i][j]=max(f[i][j],f[i+1][j-v[i]]+w[i]);
    }
}
int j=V;
for(int i=1;i<=N;i++)
{
    if(j>=v[i] && f[i][j]==f[i+1][j-v[i]]+w[i])
    {
        j-=v[i];printf("%d ",i);
    }
}
```

求方案数

对于给定的一个背包容量、物品费用、其他关系等的问题，求**装到一定容量**的方案总数。这种问题就是把求最大值换成求和即可。例如0-1背包问题的转移方程就变成了：

$$dp_i = \sum(dp_i, dp_{i-c_i})$$

初始条件：$dp_0 = 1$，因为当容量为0时也有一个方案，即什么都不装。

求最优方案总数

要求最优方案总数，我们要对0-1背包里的dp数组的定义稍作修改，DP状态$f_{i,j}$为在只能放前i个物品的情况下，容量为j的背包「正好装满」所能达到的最大总价值。

这样修改之后，每一种DP状态都可以用一个$g_{i,j}$来表示方案数。

$f_{i,j}$表示只考虑前i个物品时背包体积「正好」是j时的最大价值。

$g_{i,j}$表示只考虑前i个物品时背包体积「正好」是j时的方案数。

如果$f_{i,j} = f_{i-1,j}$且$f_{i,j} \neq f_{i-1,j-v} + w$说明我们此时不选择把物品放入背包更优，方案数由$g_{i-1,j}$转移过来，

如果$f_{i,j} \neq f_{i-1,j}$且$f_{i,j} = f_{i-1,j-v} + w$说明我们此时选择把物品放入背包更优，方案数由$g_{i-1,j-v}$转移过来，

如果$f_{i,j} = f_{i-1,j}$且$f_{i,j} = f_{i-1,j-v} + w$说明放入或不放入都能取得最优解，方案数由$g_{i-1,j}$和$g_{i-1,j-v}$转移过来。

```
for(int i=1;i<=n;i++)
```

```
{
    int v,w;cin>>v>>w;
    for(int j=m;j>=v;j--)
    {
        int maxx=max(f[j],f[j-v]+w),cnt=0;
        if(maxx==f[j]) cnt+=g[j];
        if(maxx==f[j-v]+w) cnt+=g[j-v];
        g[j]=cnt%mod;f[j]=maxx;
    }
}
```

第k优解

$dp_{i,j,k}$记录了前i个物品中，选择的物品总体积为j时，能够得到的第k大的价值和。转移时，与仍然采用背包原来的转移方式。不同的是现在需要记录的是所有可能情况的一个序列，使用双指针维护即可。

```
for(int i=1;i<=n;i++)
{
    for(int j=V;j>=w[i];j--)
    {
        for(int p=1;p<=k;p++)
        {
            a[p]=f[j-w[i]][p]+v[i];b[p]=f[j][p];
        }
        int x=1,y=1,z=1;a[k+1]=b[k+1]=-1;
        while(z<=k&&(a[x]!=-1||b[y]!=-1))
        {
            if(a[x]>b[y]) f[j][z]=a[x++];
            else f[j][z]=b[y++];
            if(f[j][z]!=f[j][z-1]) z++;
        }
    }
}
printf("%d\n",f[V][k]);
```

0.2 树形DP

树上背包

现在有n门课程，第i门课程的学分为a_i，每门课程有零门或一门先修课，有先修课的课程需要先学完其先修课，才能学习该课程。一位学生要学习m门课程，求其能获得的最多学分数。

我们可以新增一门0学分的课程（设这个课程的编号为0），作为所有无先修课课程的先修课，这样我们就将森林变成了一棵以0号课程为根的树。

使用上下界优化后的时间复杂度为$O(nm)$。

```
int dfs(int u)
{
    int sz=1;
    f[u][1]=w[u];
    for(int i=head[u];~i;i=r[i].nex)
    {
        int v=r[i].b;
        int cnt=dfs(v);
        for(int j=min(sz,m+1);j>=1;j--) //上下界优化
        { //m+1是因为加上新建的根结点一共选了m+1门课
            for(int k=1;k<=cnt&&j+k<=m+1;k++) //这里k从1开始,
              如果需要从0则最后处理
                f[u][j+k]=max(f[u][j+k],f[u][j]+f[v][k]);
        }
        sz+=cnt;
    }
    return sz;
}
```

0.3 数位DP

对于多组数据且状态数较多的数位DP，如何保证可以重复使用DP数组且不清空就成为了提升效率的关键。

发现有一个东西在阻止我们复用DP数组，那就是limit。对于不同的输入，limit的意义本质上是不同的。不过我们注意到limit等于1的状态出现的频率远远小于limit等于0的状态，所以我们可以选择只记忆化limit为0的状态，这样每次DP数组的意义就完全相同了。

```
ll dfs(int pos,int lead, int limit)
{
```

```
    ll ans = 0;
    if (!pos) return ...;
    ll &d = dp[pos][...];
    if (!limit && d != -1) return d;
    for (int v = 0;v<=(limit ? A[pos] : 9); ++v)
        ans += dfs(pos - 1, lead&&v==0, limit && A[pos] == v
            );
    if (!limit) d = ans;
    return ans;
}
ll f(ll x)
{
    int len = 0;
    while (x) A[++len] = x % 10, x /= 10;
    return dfs(len, ..., true);
}
```

0.4 概率DP

一个软件有s个子系统，会产生n种bug。某人一天发现一个bug，这个bug属于某种bug分类，也属于某个子系统。每个bug属于某个子系统的概率是$\frac{1}{s}$，属于某种bug分类的概率是$\frac{1}{n}$。求发现n种bug，且s个子系统都找到bug的期望天数。

令$f_{i,j}$为已经找到i种bug分类，j个子系统的bug，达到目标状态的期望天数。考虑$f_{i,j}$的状态转移：

- $f_{i,j}$，发现一个bug属于已经发现的i种bug分类，j个子系统，概率为$p_1 = \frac{i}{n} \cdot \frac{j}{s}$。
- $f_{i,j+1}$，发现一个bug属于已经发现的i种bug分类，不属于已经发现的子系统，概率为$p_2 = \frac{i}{n} \cdot (1 - \frac{j}{s})$。
- $f_{i+1,j}$，发现一个bug不属于已经发现bug分类，属于j个子系统，概率为$p_3 = (1 - \frac{i}{n}) \cdot \frac{j}{s}$。
- $f_{i+1,j+1}$，发现一个bug不属于已经发现bug分类，不属于已经发现的子系统，概率为$p_4 = (1 - \frac{i}{n}) \cdot (1 - \frac{j}{s})$。

再根据期望的线性性质，就可以得到状态转移方程：

$$
\begin{aligned}
f_{i,j} &= p_1 \cdot f_{i,j} + p_2 \cdot f_{i,j+1} + p_3 \cdot f_{i+1,j} + p_4 \cdot f_{i+1,j+1} + 1 \\
&= \frac{p_2 \cdot f_{i,j+1} + p_3 \cdot f_{i+1,j} + p_4 \cdot f_{i+1,j+1} + 1}{1 - p_1}
\end{aligned}
$$