背包问题

1. 01背包问题

问题：n件物品，第i件物品重量和价值分别w[i],v[i]，在`总重量不超过m`[1]的情况下，求所有挑选方案中价值总和的最大值。

思路：01背包问题中，每件物品只有选和不选两种状态。

设dp[i][j]:在前i中当中选取总重量不超过j时的总价值的最大值。

那么在选取第i件物品时，dp[i][j]可以从何处转移而来?

思考得到：因为每件物品只有选和不选两种状态，在选取第i件时，dp[i][j]=dp[i-1][j-w[i]]+v[i]，即是：放入第i件物品后还能得到的价值(从i-1当中得到)+放入第i件物品得到的价值；不选取第i件物品时，dp[i][j]=dp[i-1][j]。最终的答案就是两者取max。

从而状态转移方程:dp[i][j]=max(dp[i-1][j],dp[i][j]=dp[i-1][j-w[i]]+v[i]);

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

一维数组优化：dp[j]:在总重不超过j时的总价值的最大值。由上面的思路，我们应该考虑如何得到:dp[j]=max(dp[j],dp[[j-w[i]]+v[i])。在二维数组当中每次均使用到前一个背包(dp[i-1])，如果要优化，那我们必须在一位数组中也达到使用到前一个背包的效果。于是我们可以考虑将第二层循环倒置，这样，我们在上一次i的循环当中的结果就可以得以保存重复使用。(`如果不倒置的话前，当前物品就会被选n次`)

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

2. 完全背包问题

问题：n种物品，第i件物品重量和价值分别w[i],v[i]，在总重量不超过m的情况下，求所有挑选方案中价值总和的最大值。这里每种物品可以选取任意多个。

思路：设dp[i][j]:在前i中当中选取总重量不超过j时的总价值的最大值。

由01背包的基础，可以想到递推关系：dp[i-1][j]=max{dp[i-1][j-k*w[i]]+k*v[i]|k>=0}，于是得到最简单三重循环。

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

三重循环的复杂度明显不够好，在这个算法中，使用测试点画一下图的话，可以发现在算法中有很多的重复递推，因此考虑如何去掉多余的计算。

我们发现：在dp[i][j]的递推中，k>=1的部分其实是可以由dp[i][j-w[i]]递推得到，在j不断增大时，j-w[i]其实也在增大，所以可以得到1...k的值，就相当于在dp[i]上的平行的倍增递推。

> 例如：设w[i]=1. 当j=1时，dp[1]=j-w=0为k=1. 当j=2时，dp[2]=j-w=1=dp[1]+w(即可以由前 j - w 项递推到，以w倍增)

dp[i][j]还可以由dp[i-1][j]推导，就直接获取到前i-1时总重不超过j时的最优解。则最终为两者取max。

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

一维数组优化：同01背包的一维数组优化，考虑得到dp[j] = max(dp[j],dp[j-w[i]]+v[i]);由01背包，可以想到不将 j 倒置就可以将每一种物品取n次(平行倍增递推)。

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

滚动数组：因为每次使用到的都是dp[i][j]和dp[i-1][j]，就可以使用dp[2][m+1]来实现。

例如：

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

3. 多重背包问题

问题：n种物品，第i件物品重量和价值分别w[i],v[i]，每种物品的数量有限为s[i]，在总重量不超过m的情况下，求所有挑选方案中价值总和的最大值。

思路：多重背包也是从01背包扩展而来。

朴素算法：将s[i]个物品逐个拆分，转化为01背包求解，在数据量不大的情况下是可以的的。

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

二进制思想优化

给定一个数n，如何使用将n拆分为最少个数，使他们可以表示0~n内的所有数字。

首先对n以2为底取log并且向上取整，得到需要使用数字的个数k。然后确定数字：前k-1个数字为1,2,...2^(k-1),第k个数字为n-2^(k-1)+1(其实就是n-∑(前k-1个数字))。

> 例如n=10，则k=4，拆分为：1,2,4,3；n=7,拆分为1,2,4。

在01背包中，每件物品只有取和不取两种状态，每次循环都是去询问这两种状态，每一次递推的结果都是在前一次取或者是不取的最优的情况下再去询问当前状态，因此在多重背包中我们将物品分堆，在每一次的询问下就可以组合出n种情况，从而降低询问次数到logn

```
struct good{
    int v,w;
};
int main(){
    vector<good> goods;
    for(int i = 0 ;i < n; i++){
        int v,w,s;
        cin >> v >> w >> s;
        //分堆
        for(int k = 1; k <= s; k<<=1){
            s -= k;
            goods.push_back({v*k,w*k});
        }
        if(s > 0)
            goods.push_back({s*k,s*k});
    }
    for(auto g:goods)
        for(int j = m; j >= g.w ; j--)
            dp[j] = max(dp[j], dp[j - g.w] + g.v);
    cout << dp[m] << endl;
    return 0;
}
```

单调队列优化

先看看最朴素的算法：`dp[j] = max(dp[j],dp[j - k*w[i]] + k*v[i])`(这里使用的一维数组优化，实际上是与`dp[i-1][]`)，可以发现随着k的增大，dp[j]都与 [j - w] + v, [j - 2w] + 2v , [j - 3w] + 3v...有关;dp[j - w]与[j - 2w] + v , [j - 3w] + 2v...有关(其中w为重量w[i]，v为价值v[i])，

> dp[j] = dp[j - w] + v , dp[j - 2w] + 2v , dp[j - 3w] + 3v ... ①
> // j 从m倒序推
> dp[j - w] = dp[j - 2w] + v , dp[j - 3w] + 2v ...
> dp[j - w] + v = dp[j - 2w] + 2v , dp[j - 3w] + 3v ... ②
> dp[j - k * w] + k * v = ...
> 从中发现规律若第②加上一个v，则第①式的最大值即是第②式的最大值
> (整体加上一个数对整体的max没有影响)
> 所以问题转化为求前一个序列中的最大值 → 单调队列(求区间内的最大值)
> // j 从0正序推 似乎更好理解
> dp[j] = dp[j - w] + v , dp[j - 2w] + 2v , dp[j - 3w] + 3v ...
> dp[j + w] = dp[j] + v , dp[j - w] + 2v ...
> dp[j + w] - v = dp[j] , dp[j - w] + v ...
> dp[j + k * w] - k * v = ...
> 最大值为前一个序列中最大值加上k * w
> 同一个序列中每项减去对应的k * w才能进行比较

> 若第一个序列为k=1,即 + 1 * w 第二个序列为 k=2,即 + 2 * w
> 那么第一个序列和第二个序列的比较只需要 第二个序列在第一个序列的基础上 + 1 * w
> 此时第二个序列的最大值就是第一个序列的最大值加上(2 - 1) * v
> 这点对应于下面代码中的f[k] = max(f[k], g[q[h]] + (k - q[h]) / c * w)

那么如何确定这个序列，我们重新将`j + k*w`看为一个整体j，从0...m取值，在同一序列中的数减去`k*w`得到的值相同，其实就是`j % w`相同。如何确定区间长度，对于一件物品能取的范围为`min(s*c , m)`,所以我们的区间长度就为`min(s*c , m)`。

代码描述：

```
for : '物品循环'
    for : '余数枚举'
        '同一序列中使用单调队列'
```

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

const int N = 20010;
int n, m;
int f[N], g[N], q[N];//g存储前一次结果 q存储单调队列最大值下标

int main() {
    cin >> n >> m;
    for (int i = 0; i < n; i++) {
        int c, w, s;
        cin >> c >> w >> s;
        memcpy(g, f, sizeof(f));
        for (int j = 0; j < c; j++) {
            int h = 0, t = -1;
            for (int k = j; k <= m; k += c) {//这里k=思路中j+k*w
                f[k] = g[k];
                if (h <= t && k - s * c > q[h]) h++;//区间滑动
                if (h <= t) f[k] = max(f[k], g[q[h]] + (k - q[h]) / c * w);//选取
最大值
                while (h <= t && g[q[t]] - (q[t] - j) / c * w <= g[k] - (k - j)
/ c * w) t--;//同一个序列中每项减去对应的k * w进行比较
                q[++t] = k;
            }
        }
    }
    cout << f[m] << endl;
    return 0;
}
```

4. 混合背包问题

问题：将n种物品，第i件物品**重量和价值分别v[i],w[i]**(交换了上文v和w的意义)，在总重量不超过m的情况下，求所有挑选方案中价值总和的最大值。其中有些物品只能取1次，有些物品可以取无数次，有些物品取有限次。

思路：在前几种背包问题的解决办法下，将问题进行分类(01，完全，多重(通常使用二进制优化))，然后根据对应算法进行计算。

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

const int V = 1010;

struct  Good
{
    int v, w;
    bool type;
};

int n, m;
int dp[V];

int main() {
    vector<Good> goods;
    cin >> n >> m;
    for (int i = 0; i < n; i++) {
        int v, w, s;
        cin >> v >> w >> s;
        if (s == -1) {
            goods.push_back({ v,w,true });
        }
        else if (s > 0) {
            for (int k = 1; k <= s; k <<= 1) {
                s -= k;
                goods.push_back({ k*v,k*w,true });
            }
            if (s > 0)goods.push_back({ s*v,s*w,true });
        }
        else {
            goods.push_back({ v,w,false });
        }
    }
    for (auto good : goods) {
        if (good.type)
            for (int j = m; j >= good.v; j--)
                dp[j] = max(dp[j], dp[j - good.v] + good.w);
        else
            for (int j = good.v; j <= m; j++)
                dp[j] = max(dp[j], dp[j - good.v] + good.w);
    }
    cout << dp[m];
    return 0;
}
```

5. 二维费用背包问题

问题：有 N 件物品和一个容量是 V 的背包，背包能承受的最大重量是 M。每件物品只能用一次。体积是 vi，重量是 mi，价值是 wi。求解将哪些物品装入背包，可使物品总体积不超过背包容量，总重量不超过背包可承受的最大重量，且价值总和最大。输出最大价值。

思路：之前的背包问题使用`dp[j]`来表示总量为j时的最大价值，那我们可以使用`dp[i][j]`来表示体积为i，重量为j时的最大价值，每个物品两重循环进行求解。

```
#include<iostream>
#include<algorithm>

using namespace std;

const int V = 110;

int N, C, M;
int dp[V][V];

int main() {
    cin >> N >> C >> M;
    for (int i = 0; i < N; i++) {
        int c, m, w;
        cin >> c >> m >> w;
        for (int j = C; j >= c; j--)
            for (int k = M; k >= m; k--)
                dp[j][k] = max(dp[j][k], dp[j - c][k - m] + w);
    }
    cout << dp[C][M];
    return 0;
}
```

6. 分组背包问题

问题：有 N 组物品和一个容量是 V的背包。每组物品有若干个，同一组内的物品最多只能选一个。每件物品的体积是 vij，价值是 wij，其中 i 是组号，j是组内编号。求解将哪些物品装入背包，可使物品总体积不超过背包容量，且总价值最大。输出最大价值。

思路：现在问题的选择就是每组内的物品选还是不选。

`dp[j] = max(dp[j] ,dp[j - v[0]] + w[0],...,dp[j - v[s - 1]] + w[s - 1])`

```
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 110;

int n, m;
int h[N], e[N], ne[N], index;
int v[N], w[N];
int dp[N][N];
//(x-->y)
void add(int x, int y) {
```

```
    e[++index] = y, ne[index] = h[x], h[x] = index;
}
void dfs(int u) {
    for (int i = h[u]; i; i = ne[i]) {
        int son = e[i];
        dfs(son);
        for (int j = m - v[u]; j >= 0; j--) {
            for (int k = 0; k <= j; k++)//分组选择其中一个
                dp[u][j] = max(dp[u][j], dp[u][j - k] + dp[son][k]);
        }
    }
    for (int j = m; j >= v[u]; j--) dp[u][j] = dp[u][j - v[u]] + w[u];
    for (int j = 0; j < v[u]; j++) dp[u][j] = 0;
}
int main() {
    cin >> n >> m;
    int root, p;
    for (int i = 1; i <= n; i++) {
        cin >> v[i] >> w[i] >> p;
        if (p == -1) root = i;
        else add(p, i);
    }
    dfs(root);
    cout << dp[root][m] << endl;
    return 0;
}
```

7. 有依赖问题的背包

问题：这种背包问题的物品间存在某种“依赖”的关系。也就是说，i依赖于j，表示若选物品i，则必须选物品 j。题目连接：有依赖的背包问题

思路：这种依赖关系可以使用树形结构表示，若要选择子节点，那么必须选择其父节点，所以对于父节点来说，对其子节点进行分组，变成分组背包问题，求出组内的最优解，最后加上父节点。

使用二维数组 `dp[i][j]` 表示节点为 i ，背包限制为 j 时的最优解。每次对 i 节点的子节点进行递归，先算出其子节点不依赖父节点使当限制为 j 的最优解，最后父节点遍历子节点选取最优解。注意父节点在选择的时候要先将自己的位置预留出来，最后能够选择的父节点的 j 才能加上子节点的最优解，不能选择的为零。

```
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 110;

int n, m;
int h[N], e[N], ne[N], index;
int v[N], w[N];
int dp[N][N];
//(x-->y)
void add(int x, int y) {
    e[++index] = y, ne[index] = h[x], h[x] = index;
}
```

```
void dfs(int u) {
    for (int i = h[u]; i; i = ne[i]) {
        int son = e[i];
        dfs(son);
        for (int j = m - v[u]; j >= 0; j--) {
            for (int k = 0; k <= j; k++)//分组选择其中一个
                dp[u][j] = max(dp[u][j], dp[u][j - k] + dp[son][k]);
        }
    }
    for (int j = m; j >= v[u]; j--) dp[u][j] = dp[u][j - v[u]] + w[u];
    for (int j = 0; j < v[u]; j++) dp[u][j] = 0;
}
int main() {
    cin >> n >> m;
    int root, p;
    for (int i = 1; i <= n; i++) {
        cin >> v[i] >> w[i] >> p;
        if (p == -1) root = i;
        else add(p, i);
    }
    dfs(root);
    cout << dp[root][m] << endl;
    return 0;
}
```

> tips:若是多颗树(即：森林)，那么可以建立一个虚根

8. 背包问题求方案数

问题：在01背包的条件下，求最优选择的方案数。

思路：在进行dp求最优解的时候，其实就已经对方案进行了选择，所以我们只需要在dp的时候另开一个数组 g 来保存当前最优解的方案数，`dp[j] = max(dp[j], dp[j - v] + w)`，若 `dp[j]== dp[j-v] + w` 则当前方案等于 `g[j-v]` 的方案数，否则则等于 `g[j]` 的方案数，但是如果两个值相同，那么方案数就应为 `g[j-v]+g[j]`。似乎有两种方法：

1：将dp数组初始化为 -1，因为若初始化为 0 ，那我们所求的是在不超过 j 的 的最优方案，不能确定此时背包真正容量是多少，就不好统计方案数。最后遍历dp,求出最优值，然后将等于最优值的方案数求和。

2：将g数组初始化为 1(什么都不选是一种方案)，然后对应 `g[m]` 就是方案数。

方法1:

```
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 1010, mod = 1e9 + 7, INF = 1e9;

int n, m;
int dp[N], g[N];

int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) dp[i] = -INF;
    g[0] = 1;
```

```
    for (int i = 0; i < n; i++) {
        int v, w;
        cin >> v >> w;
        for (int j = m; j >= v; j--) {
            int t = max(dp[j], dp[j - v] + w);
            int s = 0;
            if (t == dp[j]) s += g[j];
            if (t == dp[j - v] + w) s += g[j - v];
            if (s >= mod) s -= mod;
            g[j] = s;
            dp[j] = t;
        }
    }
    int maxw = 0;
    for (int i = 0; i <= m; i++) maxw = max(maxw, dp[i]);
    int res = 0;
    for (int i = 0; i <= m; i++) {
        if (dp[i] == maxw) {
            res += g[i];
            if (res >= mod) res -= mod;
        }
    }
    cout << res;
    return 0;
}
```

方法2：

```
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 1010, mod = 1e9 + 7;

int n, m;
int dp[N], g[N];

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

9. 求背包问题方案

问题：在01背包的条件下，求字典序最小的选择方案。

思路：在求出`dp[i][m]`最优解后，在能够购买的情况下，将最优解与`dp[i-1][m-w[i]]+v[i]`进行比较，若相等则是选择了，若不等则是未选，同时`i-1`中的最优值是存储在`m - w[i]`中，在判断`i-1`时`m = m - w[i]`。由于是求字典序最小，所以我们应该将物品倒置循环，将整个最优解存储在`dp[0][m]`中，然后从前往后遍历确定顺序。

```
#include<iostream>
#include<algorithm>

using namespace std;

const int N = 1010;

int n, m;
int dp[N][N], v[N], w[N];

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

> tips:对于背包问题，就是先循环物品，再循环限制，再循环决策

参考资料：

- 背包九讲专题
- ACWing以上的题目均可以在此网站"题库分类:背包九讲"中找到。
- 背包九讲

---

1. 如果是恰好把背包装满，则除dp[0]初始化为0外，其他均初始化为负无穷，因为除dp[0]以外，此时其他容量的背包均没有合法的解 ↩