背包问题

01背包

每个物品1件

```
const int N = xx, V = xx;
int n, m; // 物品数量，背包体积
int dp[V]; // dp[N][V] => dp[V]
int main() {
    scanf("%d%d", &n, &m);
    int v, w;
    for (int i = 1; i <= n; ++i) {
        scanf("%d%d", &v, &w);
        for (int j = m; j >= v; --j) {
            dp[j] = max(dp[j], dp[j-v] + w);
        }
    }
    printf("%d\n", dp[m]);
}
```

完全背包

每个物品无数件

- 状态计算： `dp[i][j]` 表示的集合可以划分为 { 选0/1/2/../k/../最满 个物品i }
  - 选0个物品i： = `dp[i-1][j]`
  - 选k个物品i： = `dp[i-1][j-k*volume[i]] + k * weight[i]` 且要合法
- 状态转移公式： `dp[i][j] = max( dp[i-1][j], dp[i-1][j-volume[i]] + weight[i], ..., dp[i-1][j-k*volume[i]] + k*weight[i], ... ),`
  - 观察 `dp[i][j-weight[i]]` 可优化公式为 `dp[i][j] = max( dp[i-1][j], dp[i][j-weight[i]] + weight[i] )`
- 代码优化 `dp[i]` ，观察其与01背包的区别，唯一不同仅在 `dp[i][j-weight[i]]` 的第一维，此处直接用到了[i]当前状态，而不是[i-1]上一轮状态，所以无需向01背包那样内循环逆序**以避免覆盖上一个状态的值**，可直接正向循环，因为用到的i状态就在此前更新

```
const int N = xx, V = xx;
int n, m;
int dp[V];
int main() {
    scanf("%d%d", &n, &m);

    int v, w;
```

```
    for (int i = 1; i <= n; ++i) {
        scanf("%d%d", &v, &w);
        for (int j = v; j <= m; ++j) {
            dp[j] = max(dp[j], dp[j-v] + w);
        }
    }
    printf("%d\n", dp[m]);
}
```

多重背包

每个物品有指定数量p
物品种数n，背包容量m，物品数量p

【普通做法】$O(nmp)$

```
const int MAXN = 102, MAXV = 102;
int v[MAXN], w[MAXN], s[MAXN];
int N, V;
#ifdef OPT
    int dp[MAXV];
#else
    int dp[MAXN][MAXV];
#endif

int main() {
    scanf("%d%d", &N, &V);
    for (int i = 1; i <= N; ++i) scanf("%d%d%d", &v[i], &w[i], &s[i]);
    // can even be placed in the dp calculation

#ifdef OPT
    for (int i = 1; i < N; ++i) {
        for (int j = V; j >= v[i]; --j) {
            for (int k = 0; k <= s[i]; k++) {
                if (j - k*v[i] >= 0) dp[j] = max(dp[j], dp[j - k*v[i]] +
k*w[i]);
            }
        }
    }
    for (int k = 0; k <= s[N]; k++) // i = N, j = V
        if (V - k*v[N] >= 0) dp[V] = max(dp[V], dp[V - k*v[N]] + k*w[N]);
    printf("%d\n", dp[V]);
#else
    for (int i = 1; i <= N; ++i) {
        for (int j = 1; j <= V; ++j) {
            for (int k = 0; k <= s[i]; k++) {
                if (j - k*v[i] >= 0) dp[i][j] = max(dp[i][j], dp[i-1][j -
k*v[i]] + k*w[i]);
            }
```

```
        }
    }
    printf("%d\n", dp[N][V]);
#endif
    return 0;
}
```

【二进制优化】 $O(n \log p \cdot m)$
基本思路：将数量拆分，构成01背包

- 有s个就拆分为s件物品，就可以变为01背包，但这样拆依旧会超时：2000 -> 2000个， 最多可生成 2000 * 1000 个物品，再n^2算法就超了， 具体即 $O((np) \cdot m)$和最基本的算法一样
- 根据二进制表示拆分，7 -> 1 2 4, 8-> 1 2 4 8，这样就是$O(n \log p \cdot m)$，但这样也不对，因为可以表示出比原来还多的数量 $\rightarrow$ 拆分的最后一个仅为余出来的数 8 -> 1 2 4 1, 10 -> 1 2 4 3

代码优化：分开的vector不如和起来的vector，使用现成的pair似乎不如自己简单定义的结构体（不过要使用`emplace_back`的话要把用到的构造函数定义一下）
从混合背包回来：重写时意识到了更简洁的写法，时间从1800ms压到300ms；从cin/cout改printf/scanf，时间从387ms变为351ms

```
const int MAXV = 2002;
int dp[MAXV];
int N, V;

int main() {
    cin >> N >> V;
    int v, w, s;
    for (int i = 1; i <= N; ++i) {
        cin >> v >> w >> s;

        int k = 1, kv, kw;
        while (s) {
            kv = k * v, kw = k * w;
            for (int j = V; j >= kv; --j)
                dp[j] = max(dp[j], dp[j-kv] + kw);
            s -= k;
            k = min(s, k << 1);
        }
    }
    cout << dp[V] << endl;
}
```

【单调队列优化】

- 从I朴素的dp思路开始，**参考完全背包问题的归纳**

```
完全背包
dp[i][j  ] = max( dp[i-1][j], dp[i-1][j-v]+w, dp[i-1][j-2v]+2w, ... , dp[i-1][j-kv]+    kw, ... )
dp[i][j-v] = max(             dp[i-1][j-v]  , dp[i-1][j-2v]+ w, ... , dp[i-1][j-kv]+(k-1)w, ... )
所以 dp[i][j] = max( dp[i][j], dp[i][j-v]+w )

多重背包
dp[i][j   ] = max( dp[i-1][j], dp[i-1][j-v]+w, dp[i-1][j-2v]+2w, ... , dp[i-1][j-sv]+    sw )  # s项
dp[i][j- v] = max(             dp[i-1][j-v]  , dp[i-1][j-2v]+ w, ... , dp[i-1][j-sv]+(s-1)w, dp[i-1][j-(s+1)v]+sw )  # s项
dp[i][j-2v] =
dp[i][j-3v] =
...

记 r = j % v，则要求的每个点是 r, r+v, r+2v, .., j-v, j
dp[i][j   ]:  算从j开始向前的s个点（窗口大小 = s+1）
dp[i][j- v]:  算从j-v开始向前的s个点
...
对于每一项，都是求了长度为s的窗口内所有点的max值(当然还有偏移值w)；(特殊情况，前面的数不足填满窗口了，那就不算)
此即滑动窗口问题：
    154.滑动窗口：在线性时间求出所有滑动窗口的最大值，记滑动窗口大小为k，数据量为n，则单调队列优化将复杂度从O(nk)优化到O(n)
如此，便可将计算dp[i][]的复杂度从O(vs)优化到O(v)
```

从窗口的视角看

- 完全背包问题：`dp[i][xx]` 每项求的都是**前缀**的最值
- 多重背包问题：`dp[i][xx]` 每项求的是**滑动窗口**的最值

```
const int MAXN = 1002;
const int MAXV = 20002, MAXS = 20002;

int q[MAXV]; // mono queue
int N, V;
int dp[MAXV];
#define offset_value(idx, base) (dp[idx] + ((base)-idx)/v * w)

int main() {
    scanf("%d%d", &N, &V);
```

```
    int v, w, s;

    for (int i = 1; i <= N; ++i) {
        scanf("%d%d%d", &v, &w, &s);
        for (int j = V; j > V-v; --j) {
            int head = 0, tail = -1;

            // Preload
            int k, cnt;
            for (k = j, cnt = 0; k >= 0 && cnt <= s; k -= v, cnt++) {
                while (head <= tail && offset_value(q[tail], j) <=
offset_value(k, j)) tail--;
                q[++tail] = k;
            }
            k -= (s + 1 - cnt) * v;

            // Begin to Calculate dp[j], dp[j-v], ..., dp[j%v]
            for (int tar = j; tar > 0; tar -= v, k -= v) {
                dp[tar] = max(dp[tar], offset_value(q[head], tar));

                if (head <= tail && q[head] > k + s*v) head++;
                if (k >= 0) {
                    while (head <= tail && offset_value(q[tail], j) <=
offset_value(k, j)) tail--;
                    q[++tail] = k;
                }
            }
        }
    }
    printf("%d\n", dp[V]);
}
```

分组背包

给物品分组，同组物品只能选一个

```
const int MAXN = 102, MAXV = 102;
const int MAXS = 102;
int N,V;
int dp[MAXV];

int main() {
    cin >> N >> V;
    int s;
    int v[MAXS] = {0}, w[MAXS] = {0};
    // Remember to Initialize [0] = 0, It's Local Variables!
    for (int i = 1; i <= N; ++i) {
        cin >> s;
        for (int k = 1; k <= s; ++k) cin >> v[k] >> w[k];
```

```
        for (int j = V; j > 0; --j) {
            for (int k = 0; k <= s; ++k) {
                if (j >= v[k]) dp[j] = max(dp[j], dp[j-v[k]] + w[k]);
            }
        }
    }
    cout << dp[V] << endl;
}
```

二维费用背包

多了一维费用；本身很简单，但变形想不清楚的话也做不来，见1020潜水员

```
int N, V, M;
int dp[102][102];
int main() {
    cin >> N >> V >> M;
    int v, m, w;
    for (int i = 0; i < N; ++i) {
        cin >> v >> m >> w;
        for (int j = V; j >= v; --j) {
            for (int k = M; k >= m; --k) {
                dp[j][k] = max(dp[j][k], dp[j-v][k-m] + w);
            }
        }
    }
    cout << dp[V][M] << endl;
}
```

背包问题求方案数

求最优选法（价值最大的）的方案数量
`cnt[i][j]` ：前i个物品，小于等于体积j的情况下最大价值的方案数

- 对于 `dp[i][j] = max(dp[i-1][j], dp[i-1][j-v] + w)` ：选择了哪一边就继承哪一边的方案数， `cnt[i-1][j]` 或 `cnt[i-1][j-v]` ；若两边相同，则同时继承两边，即两者加和 `cnt[i-1][j] + cnt[i-1][j-v]`

```
const int MAXN = 1002, MAXV = 1002;
const int MOD = 1e9 + 7;
int dp[MAXV];
int cnt[MAXV];
int N, V;
int main() {
    cin >> N >> V;
    for (int j = 0; j <= V; ++j) cnt[j] = 1;
    int v, w;
```

```
    for (int i = 1; i <= N; ++i) {
        cin >> v >> w;
        for (int j = V; j >= v; --j) {
            int use = dp[j-v] + w;
            if (dp[j] < use) {
                dp[j] = use;
                cnt[j] = cnt[j-v];
            } else if (dp[j] == use) {
                cnt[j] = (cnt[j] + cnt[j-v]) % MOD;
            }
        }
    }
    cout << cnt[V] << endl;
}
```

背包问题求具体方案

变为求具体的方案，此处还需满足最小字典序（输出最小字典序的方案）

- 路径要记录，不再能压缩数组了 => **其实要是在计算中即时记录信息，也可以继续压缩**(正常求段dp)，见 SIMPLE宏中的写法
- 最小字典序，要从前往后选（从前往后，能选就选）：所以不妨背包第一维 `i` 加物品时，从后往前看（这样最后得到的 `dp[1][V]` 就是最终方案，即可从1开始向后判断是否选择物品），每次尽可能选（和字典序的需求保持一致）

```
const int MAXN = 1002, MAXV = 1002;
int dp[MAXV]; // 1d is ok
bool select_q[MAXN][MAXV];
int v[MAXN], w[MAXN];
int N, V;
int main() {
    cin >> N >> V;
    for (int i = 1; i <= N; ++i) cin >> v[i] >> w[i];
    for (int i = N; i >= 1; --i) {
        for (int j = V; j >= 1; --j) {
            if (j >= v[i]) {
                int nvalue = dp[j-v[i]] + w[i]; // use
                if (nvalue >= dp[j]) {
                    dp[j] = nvalue;
                    select_q[i][j] = 1;
                    continue;
                }
            }
            select_q[i][j] = 0;
        }
    }

    int j = V;
```

```
    for (int i = 1; i <= N; i++) {
        if (select_q[i][j]) {
            cout << i << " ";
            j -= v[i];
        }
    }
    cout << endl;

    return 0;
}
```

混合背包问题

01、完全、多重背包混合，即每个可选物品可能有1个，s个或无限个

- 那其实每轮 `[i]` 循环时，按当下物品的类型进行迭代的dp计算即可；这样应该就不能压缩dp了

```
const int MAXN = 1002, MAXV = 1002;
int dp[MAXV];
int N, V;
int main() {
    cin >> N >> V;
    int v, w, s;
    for (int i = 1; i <= N; ++i) {
        cin >> v >> w >> s;
        if (s == -1) {
            for (int j = V; j >= v; --j)
                dp[j] = max(dp[j], dp[j-v] + w);
        } else if (s == 0) {
            for (int j = v; j <= V; ++j)
                dp[j] = max(dp[j], dp[j-v] + w);
        } else {
            // binary optimization
            int remain = s, k = 1;
            while (remain > 0) {
                int kv = k * v, kw = k * w;
                for (int j = V; j >= kv; --j)
                    dp[j] = max(dp[j], dp[j-kv] + kw);
                remain -= k;
                k = min(remain, k << 1);
            }
        }
    }
    cout << dp[V] << endl;
}
```

有依赖的背包

树型dp + 分组背包

- 按方案划分
- 按体积划分

```
const int N = 100 + 2, V = N;
int n, m;
int v[N], w[N];
int dp[N][V];

int h[N], e[N], ne[N], idx;
int root;

void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

void dfs(int u) {
    for (int i = h[u]; ~i; i = ne[i]) {
        int son = e[i];
        dfs(son);

        // like group bp
        for (int j = m - v[u]; j > 0; --j) { // volume: u volume
            for (int k = 0; k <= j; ++k) { // select: give son volume k
                dp[u][j] = max(dp[u][j], dp[u][j-k] + dp[son][k]);
            }
        }
    }
    for (int i = m; i >= v[u]; --i) dp[u][i] = dp[u][i-v[u]] + w[u];
    for (int i = 0; i < v[u]; ++i) dp[u][i] = 0;
}

int main() {
    memset(h, -1, sizeof h);

    scanf("%d%d", &n, &m);
    int p;
    for (int i = 1; i <= n; ++i) {
        scanf("%d%d%d", &v[i], &w[i], &p);
        if (p == -1) root = i;
        else add(p, i);
    }
    dfs(root);
    printf("%d\n", dp[root][m]);
}
```