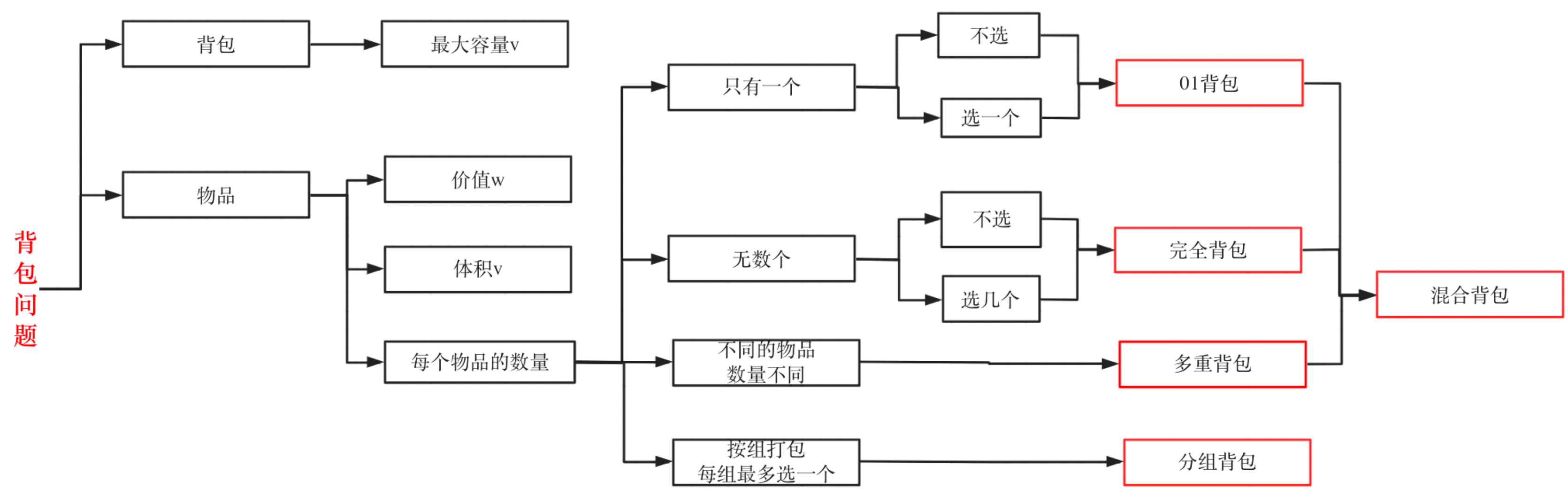
背包问题



前提规范

本文使用使用额外的二维数组 `C[i][j]`，表示在有 `i` 个物品时，背包中的最大容量为 `j`，其中这 `i` 个物品可以全选，也可以不选。

0-1背包

> 有 `N` 件物品和一个容量是 `V` 的背包。每件物品只能使用一次。
>
> 第 `i` 件物品的体积是 `v[i]`，价值是 `w[i]`。
>
> 求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。

二维

假设当前已经处理好了前 `i-1` 个物品的所有状态，那么对于第 `i` 个物品，当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变，故这种情况的最大价值为$C_{i-1,j}$；当其放入背包时，背包的剩余容量会减小 v_i，背包中物品的总价值会增大w_i，故这种情况的最大价值为$C_{i-1,j-v_i}+w_i$。由此得出DP方程。

DP方程

$$C_{i,j}=\max(C_{i-1,j},C_{i-1,j-v_i}+w_i) \tag{1}$$

假定物品个数为 `n`，背包容量为 `m`，`v[i]` 为 `i` 号物品体积，`w[i]` 为 `i` 号物品的价值

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

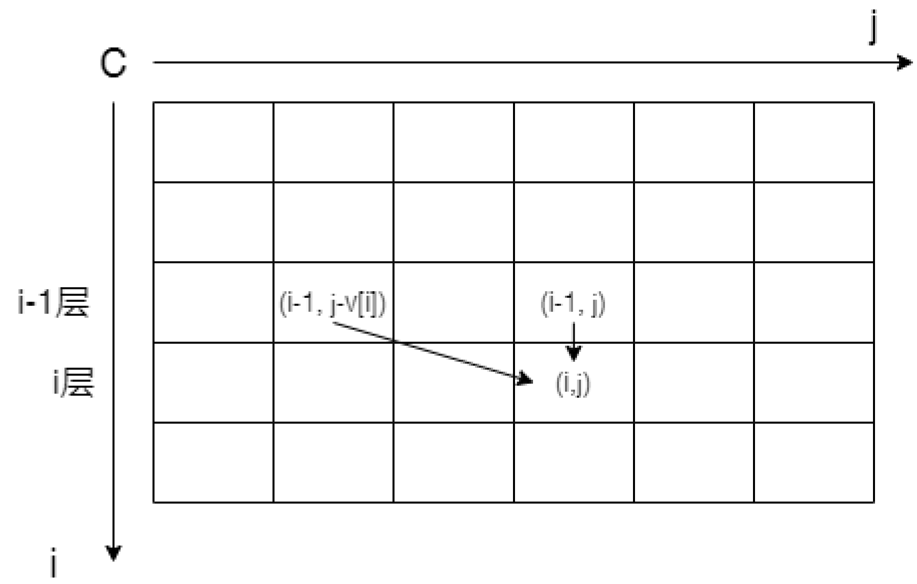
一维

由于$C_{i,j}$的影响因素中都与 `i-1` 相关，因此我们可以略去一维，使用滚动数组来解决这个问题，DP方程如下

$$C_{i,j}=\max(C_j,C_{j-v_i}+w_i) \tag{2}$$

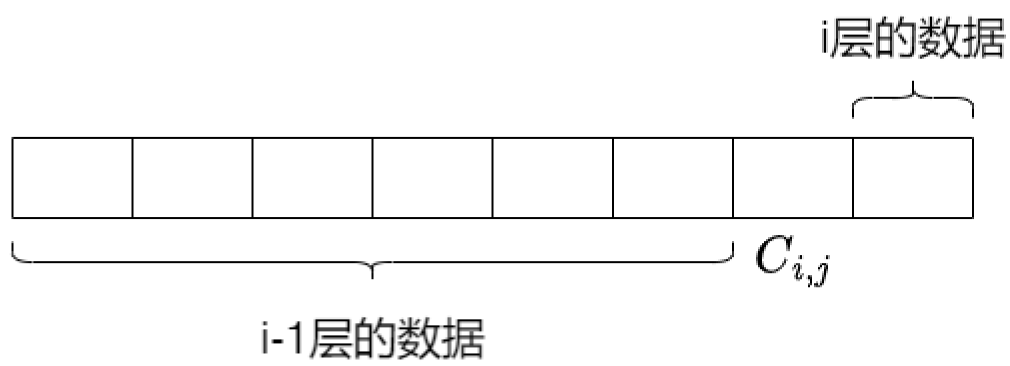
```
for(int i = 1; i<=n; i++){//可选择物品的个数依次增加
    for(int j = V; j>=v[i]; j--){//保证 j - v[i] >= 0恒成立
        c[j] = max(c[j], c[j - v[i]] + w[i]);
    }
}
```

第二层循环逆向原因



二维的时候，`dp[i][j]` 代表的是第 `i` 层第 `j` 个格子的值，它依赖于第 `i-1` 层第 `j` 个格子和第 `i-1` 层第 `j-v[i]` 个格子。如上图。

而当转换到一维的情况时，依赖情况不变，始终依赖 `i-1` 层的数据，如果我们使用正向更新的方式，那么对于某个位置$C_{i,j}$的依赖$C_{i-1,j-v[i]}$已经被更新过了，也就是$C_{i,j}$不是由 `i-1` 层更新得到的，而是由 `i` 层得到的。因此我们需要使用逆向更新的方式。如下图



由图可见，只有当逆向更新时，也就是C_{ij}从右向左更新，$C_{i,j}$仍然由第 `i-1` 层的数据更新，而不会被第 `i` 层的数据更新。因此我们需要使用逆序更新的方式。

扩展

解的输出

```
int j = V;
for(int i = n; i>0; i--){
    if(c[i][j] != c[i-1][j]){//说明在第i个物品拿了，才导致在同等背包大小的情况下，其装入的体积不同
        x[i] = 1;
        j -= v[i];//减去当前背包中的物品
    }
    else
        x[i] = 0;
}

for(int i = 1; i<=n; i++)
    cout << x[i] << ' ';
```

完全背包

> 完全背包问题就是在0-1背包上的扩充，完全背包允许一个物品多次装入，只要保证结果价值最大即可。我们假定每个物品最大可放入k_i个

DP方程

$$C_{i,j} = \max_{k=0}^{+\infty}(C_{i-1,j-k_i\times v_i} + w_i \times k_i) \tag{3}$$

三维

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

二维

推导，对三维形式展开：得到如下公式

$$C_{i,j} = \max_{k=0}^{+\infty}(C_{i-1,j}, C_{i-1,j-v_i} + w_i, C_{i-1,j-2\times v_i} + w_i \times 2, \cdots, C_{i-1,j-k\times v_i} + w_i \times k) \tag{4}$$

那么同理，`C[i][j-v[i]]` 等于

$$C_{i,j-v_i} = \max_{k=0}^{+\infty}(C_{i-1,j-v_i}, C_{i-1,j-2\times v_i} + w_i, \cdots, C_{i-1,j-(k+1)\times v_i} + w_i \times k) \tag{5}$$

如果对 `C[i][j-w[i]]` 的左右同加 `w[i]`，那么刚好与 `(4)` 中max中除 `C[i][j]和C[i-1][j]` 项外的完全相同（注意，当k趋向于无穷大时，`k+1=k`，原因自己 `google`），因此可以 `C[i][j]` 可以优化为

$$C_{i,j} = \max(C_{i-1,j}, C_{i,j-v_i} + w_i) \tag{6}$$

注意：由于 `C[i-1][j]` 是已知项，因此我们可以设置 `C[i][j]` 初始值为 `C[i-1][j]`

```
for(int i = 1; i≤n; i++){
    for(int j = 1; j≤m; j++){
        c[i][j] = c[i-1][j];//先设定默认值
        if(j ≥ v[i])
            c[i][j] = max(c[i][j], c[i][j-v[i]] + w[i]);
    }
}
```

一维

可以看出和01背包非常相似，如何做出进一步优化，针对01背包，使用滚动数组，由第 `i-1` 层的数据进行更新，而对于完全背包，如果使用二维（写法二）的方式，和01背包进行对比，可以发现和01背包的区别在于，完全背包使用第 `i` 层的数据进行更新，而01背包逆序的原因在于不能使用 `i` 层数据，而是使用 `i-1` 层数据，完全背包正好反过来，由此得到完全背包的一维形式

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

降维为一维后，为什么不需要二维时候 `c[i][j] = c[i-1][j]`，考虑这个问题我们首先需要知道，在当前 `i` 层时，未进行迭代时，滚动数组还是 `i-1` 层的数据，因此在一维的情况时，我们不必考虑在像二维那样在将 `i-1` 层的值赋给当前层。

多重背包

> 我们知道完全背包的任何物品可以拿无限个，而如果对每个物品的个数有一个最大限制，那么就是另一种背包问题，由此引出多重背包，即每个物品的个数最大为 `k[i]` 个。

在完全背包的基础上添加限制条件 `0 ≤ k ≤ k[i]` 即可，DP方程如下：

$$C_{i,j} = \max_{k=0}^{K[i]}(C_{i-1,j-k\times v_i} + w_i \times k) \tag{7}$$

三维

```
for(int i = 1; i≤n; i++){
    for(int j = 1; j≤m; j++){
        for(int k = 0; k≤K[i] && k * v[i] ≤ j; k++){
            c[i][j] = max(c[i][j], c[i-1][j - k*v[i]] + k * w[i]);
        }
    }
}
```

时间复杂度：`O(nmk)`

降维

二进制优化法

简单来说，就是把每一个物品的s个物品按照二的倍数来划分，但是至于为什么这样划分，我们可以从二进制的方向来思考，例如

我们将255划分为：1，2，4，8，···，64，128，他们对应的二进制分别为：

```
1:   0000 0001
2:   0000 0010
4:   0000 0100
8:   0000 1000
      ···
64:  0100 0000
128: 1000 0000
```

你会发现，如果把1，2，4，···，64，128加起来恰好是255，而八位二进制可以表示0~255之间的任何数，也就是二进制 `0000 0000` ~ `1111 1111`，我们单独的把每一个二进制位拿出，于是就得到了1，2，4，···，64，128这样的序列，也就是说，通过1，2，4，···，64，128这个序列任意的排列组合，我们可以得到0~255的任意一个数，这点可以从二进制相加上看出来。

问题一：那如果某个物品的个数恰好是254，又该如何划分呢？

很简单：我们划分成1，2，4，8，···，64之后，我们在额外补充一个127，得到序列1，2，4，8，···，64，127；观察可以发现，序列1，2，4，8，···，64任意排列组合可以得到0~127的任意一个数，那如果在加上补充的127，那这个序列就可以表示0~254的任意数。

现在原先我们需要从0~255个中任意选择的问题，划分到了从1，2，4，8，···，64，128这几个数选择的问题，时间复杂度也由O(255)降低到了O(7)，也就是O(N)优化到了O(logN)。

```
const int N = 1e5, M = 1000;
int v[N], w[N], c[M], cnt = 0;
for(int i = 0; i<n; i++){
    int a, b, s;
    cin >> a >> b >> s;
    int k = 1;
    while(k <= s){//以2为倍数开始拆分
        cnt++;
        v[cnt] = a * k;//每个二进制拆分后的体积
        w[cnt] = b * k;//每个二进制拆分后的价值
        s -= k;
        k *= 2;
    }
    if(s > 0){//拆分后剩余的情况
        cnt++;
        v[cnt] = a * s;
        w[cnt] = b * s;
    }
}

n = cnt;

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

时间复杂度：`O(nm*log(s))`

单调队列优化法

尚未学到，暂时不更新

分组背包

> 给定N组物品，和一个容量为V的背包。
>
> 每组物品有若干个，但同一组物品中只能选择一个。每件物品的体积是 V_{ij}，价值是W_{ij}，其中 `i` 是组号， `j` 是组内编号。
>
> 请问拿那些物品时，使得背包内的物品价值最大。

朴素解法

这个题实际上就是01背包的变种，我们寻找每个物品中使得背包中价值最大的

$$C_{i,j} = \max_{k=0}^{S[i]}(C_{i-1,j}, C_{i-1,j-v[i][k]} + w[i][k]) \tag{8}$$

注意：我们每次比较的时候都是在该组中先拿出一个，然后去寻找该组中比拿到的这个价值大的，然后放下刚刚拿出的，选择价值大的。

```
for(int i = 1; i<=n; i++){//枚举n个组
    for(int j  = 0; j<=m; j++){
        f[i][j] = f[i-1][j];//默认不放入该物品
        for(int k = 1; k<=s[i]; k++){//枚举当前组的每个物品，寻找价值最大的
            if(j >= v[i][k])
                f[i][j] = max(f[i][j], f[i-1][j - v[i][k]] + w[i][k]);
        }
    }
}
```

优化

这里同01背包一样，我们可以使用滚动数组来降低**空间复杂度**，但是需要注意，即使使用滚动数组也无法降低时间复杂度。因为这个题就没法降低时间复杂度。

```
for(int i = 1; i<=n; i++){
    for(int j = m; j>=0; j--){//注意，我们使用的是第i-1层数据，我们不希望第i层被更新之前i-1层数据被篡改，因此我们使用逆序
        c[j] = c[j-1];
        for(int k = 0; k<s[i]; k++){
            if(j >= v[i][k])
               c[j] = max(c[j], c[j - v[i][k]] + w[i][k]);
        }
    }
}
```

混合背包

尚未学到，暂时不更新