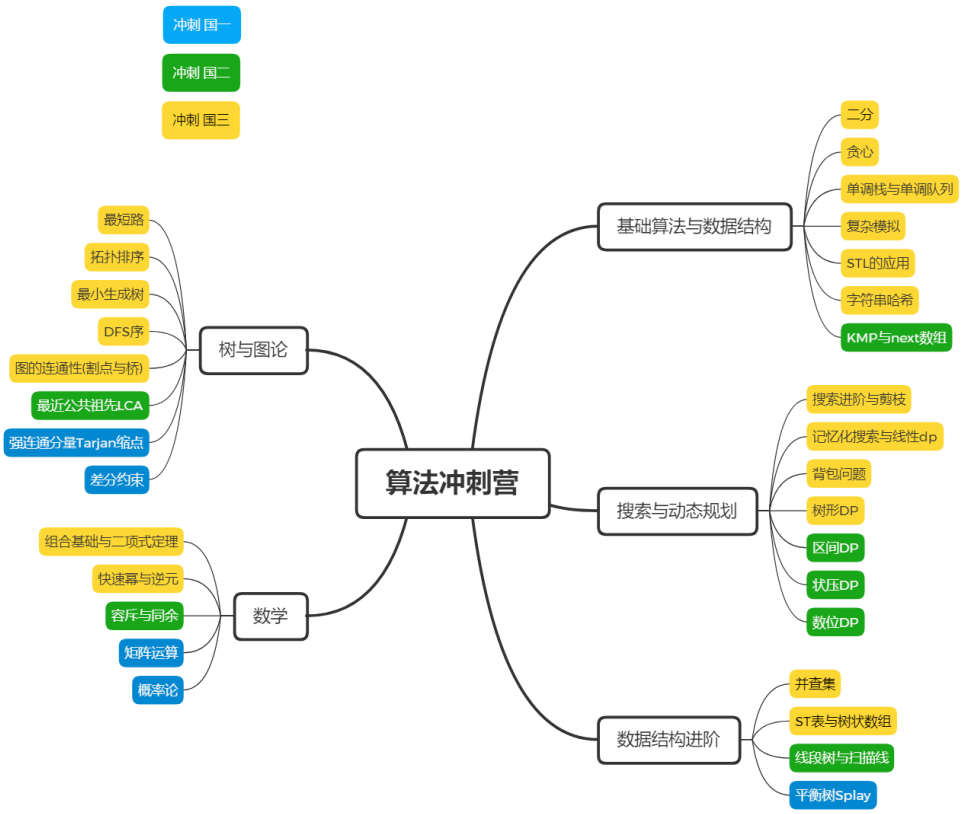
蓝桥杯30天算法冲刺集训

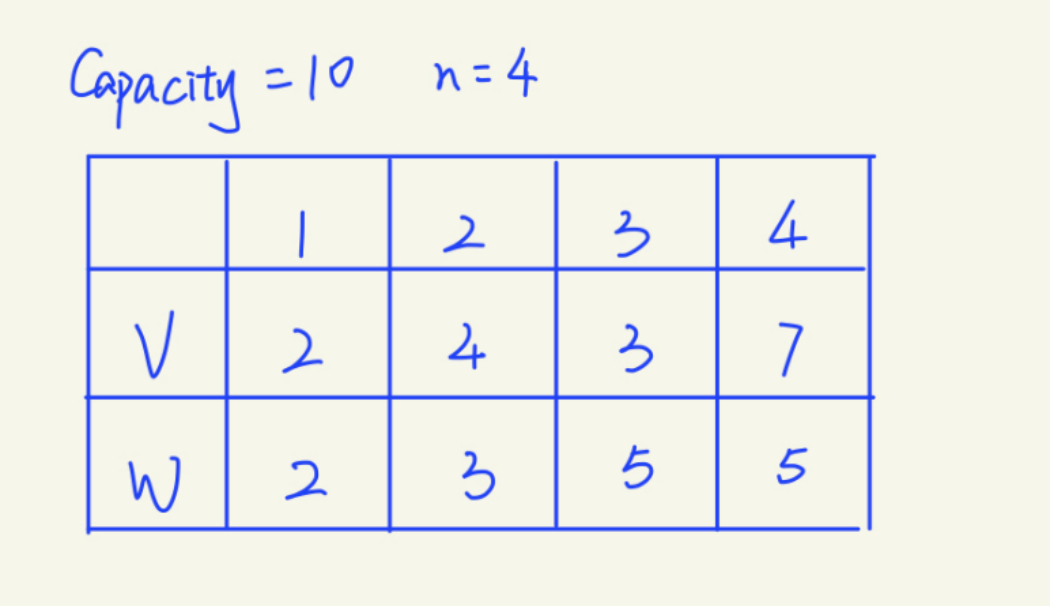


Day-8 背包问题

01背包问题

如果你是一个探险家，有一天跑到了一个地底洞穴，里面有很多很多的金块，当然都是金块了那肯定是不可以拆分的，你现在只有一个承重为 $M(0 \leq M \leq 200)$ 的背包，现在探险家发现了有 $N(1 \leq N \leq 30)$ 种金块，每一个金块的重量分别为 $w_1, w_2, \ldots, w_n$,价值为 $v_1, v_2, \ldots, v_n$，而且每一种金块有且只有一个，求我们可以带回去的最大价值。

假设我们发现了 4 种金块，每一种金块的价值 (v) 和重量 (w) 如下图；其中 *capacity* 表示我的背包容量，n 表示一共有几种金块。

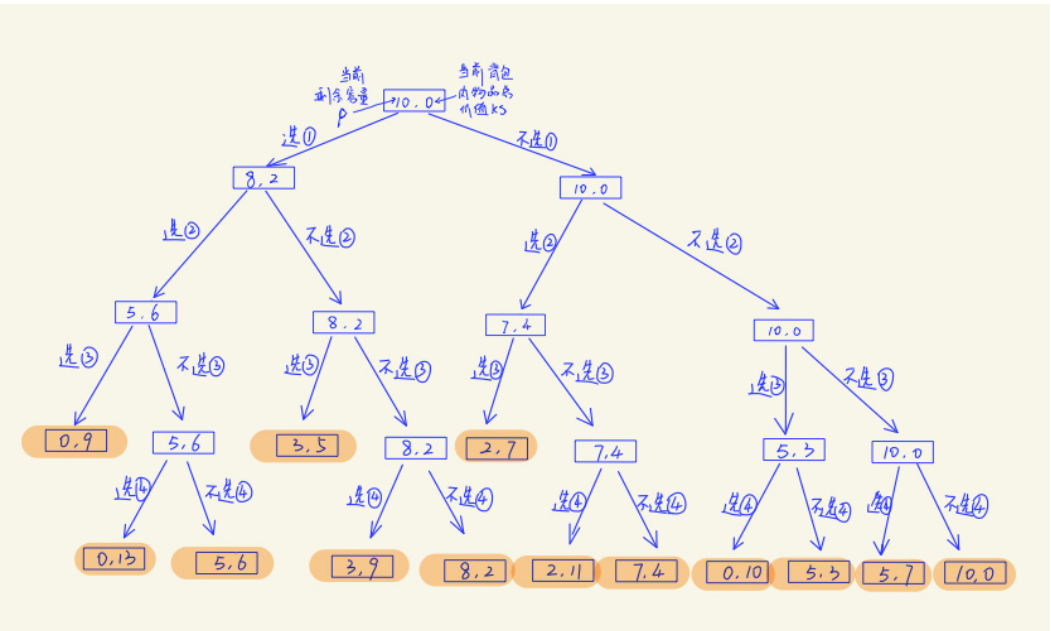


重新思考一个问题，为什么我们把他叫做 $0/1$ 背包呢？

答案很简单，因为我们发现每种只有 1 个，对于每一种金块，我们有哪几种选择呢？很显然答案只有两个，拿还是不拿，如果拿我们记为1，不拿我们记为 0，这是不是是 $0/1$ 背包的含义呢。

这个问题怎么做？同学们可以先自行思考，然后在阅读以下内容。

根据前面的纸币问题，我们可以考虑 `dfs` ，对于每一个物品我们选还是不选来进行 `dfs` 。我们可以的到如下的搜索树：



分析完上面的 `dfs` ，我们可以考虑如何用 `dp` 来解决问题。

先来考虑一下状态方程是什么？

`dp[i][j]` 表示前 i 件物品放入一个背包容量为 j 的背包可以获得的最大价值。

根据上方的图，我们可以看出，每一种物品选择就两种，选还是不选，那我考虑如果我们不选择第i件物品呢？我们发现如果我们不选择第i件物品，其实他的价值是等价于只选择 $i-1$ 件物品的，因为我第i件物品并没有装入背包中，所以对背包的容量还是价值都不产生影响。所以我们可以得到不选的时候的状态方程： `dp[i][j] = dp[i - 1][j]` ，现在考虑选择了第 i 件物品，第 i 件物品的价格是 v_i，重量是w_i，如果我们选择了 i 件物品，那么我最后的价值一定会加上v_i，并且我当前的物品是有重量的，我的背包是不是要至少留出w_i的空间，因为只有留出了 w_i 的空间，我们才能装的下第 i 件物品，当前背包是容量是 j，要留下 w_i 的空间，那么背包至少有 $j-w_i$ 的空间，所以我们得到转移方程： `dp[i][j] = dp[i - 1][j - w[i]] + v[i]` ，为什么是 `i-1` 呢？因为我要把第i件物品放进去，前面的 $i-1$ 件物品应该都是放好的，并且背包还应该至少剩余 `j-w[i]` 的空间。

综上所述：状态转移方程是为：

```
dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i])
```

请同学们好好理解 01 背包的方程，他是以后所有背包的基础！

注意边界值：应该是一件都不选，容量为0，价值为0。即 `dp[0][0] = 0`

接下来我们看看代码：

```python
# dp[i][j] 表示从前 i 个物品中选体积不超过 j 的最大价值
V, n = map(int, input().split()) # 背包的体积，物品的种类数
w = [0] * (n + 1) # 物品的重量数组
v = [0] * (n + 1) # 物品的价值数组

for i in range(1, n + 1): # 输入每种物品的重量和价值
    w[i], v[i] = map(int, input().split())

dp = [[0] * (V + 1) for _ in range(n + 1)] # dp数组

dp[0][0] = 0 # 边界值
```

```
for i in range(1, n + 1): # 枚举选择第几件物品
    for j in range(V + 1): # 枚举当前体积
        dp[i][j] = dp[i - 1][j] # 不选
        if w[i] <= j: # 背包装得下才能选
            dp[i][j] = max(dp[i][j], dp[i - 1][j - w[i]] +
v[i])

print(dp[n][V]) # 输出从前n个物品中选，体积不超过V的最大价值
```

同学们现在考虑一个问题：我们数组有没有必要开 $N \times V$ 的数组大小吗？同学们阅读到这里的时候可以自行思考一下。

首先肯定是没有必要的，为什么？我们重新聚焦于我们的状态转移方程：`dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i])`，我们可以发现，对于每一个 i，我们只用到了他的上一层也就是 $i-1$ 这一层，所以实际上我们只需要开一维长度为 2 的数组即可。即 `dp[2][V]` 即可。那考虑我们怎么样进行转移？很简单，我们知道当 `a[2]` 下标是 `0`、`1`，其实就是一个奇数和一个偶数，如果i是奇数那么 $i-1$ 一定是偶数，反之如果 i 是偶数，$i-1$ 一定是奇数，于是我们的状态转移方程就可以写成 `dp[i % 2][j] = max(dp[(i - 1) % 2][j], dp[(i - 1) % 2][j - w[i]] + v[i])`

举例：

- 当 $i=3$ 的时候：`dp[i % 2][j] = dp[1][j]` `dp[i % 2][j] = dp[(i - 1) % 2][j]` 等价于 `dp[1][j] = dp[0][j]`
- 当 $i=2$ 的时候：`dp[i % 2][j] = dp[0][j]` `dp[i % 2][j] = dp[(i - 1) % 2][j]` 等价于 `dp[0][j] = dp[1][j]`

于是我们就会发现，当 i 为偶数的时候,$i-1$ 是奇数，反之是偶数，这样我们就可以通过长度为 2 的数组即 0、1 一直进行滚动，我们把这种空间优化称之为**滚动数组优化**。

滚动数组优化

我们现在再回顾一遍 01 背包的过程

关于动态规划类问题我们通常的做题技巧是从最后一个状态从哪里转移来的，从后往前考虑解决问题。背包问题也是采用这种思路。

1.我们首先考虑背包问题最后一个状态，很自然就能想到最后一个状态一定是体积为 v_{last} 时所存放的最大价值最大。

2.确定最后一个状态以后，我们就开始往前考虑，最后一个状态是怎么由前面的状态得到。前一个状态要转移到后一个状态有两种选择

- 选第 i 件物品
- 不选第 i 件物品

同时我们也要遍历$i = 1, 2, \ldots, N$ 个物品保证前一个转移过来的状态也一定是存放体积为 v_{last-1} 时的最大价值。

3.这样就可以设一个二维数组 $dp[i][j]$

设 $dp[i][j]$ 的含义是：在背包体积为 j 的前提下，从前 i 个物品中选能够得到的最大价值。不难发现 $dp[N][V]$ 就是本题的答案。

如何计算 $dp[i][j]$ 呢？我们可以根据上面的思路将它划分为以下两部分：

- 选第 i 件物品：由于第 i 个物品一定会被选择，那么相当于从前 $i-1$ 个物品中选且总体积不超过 $j - v[i]$，对应 $dp[i-1][j-v[i]] + w[i]$ 。
- 不选第 i 件物品：意味着从前 $i-1$ 个物品中选且总体积不超过 j，对应 $dp[i-1][j]$ 。

结合以上两点可得递推公式：

$dp[i][j] = max(dp[i-1][j],\ dp[i-1][j-v[i]] + w[i])$

由于下标不能是负数，所以上述递推公式要求 $j \geq v[i]$。当 $j < v[i]$ 时，意味着第 i 个物品无法装进背包，此时 $dp[i][j] = dp[i-1][j]$。综合以上可得出：

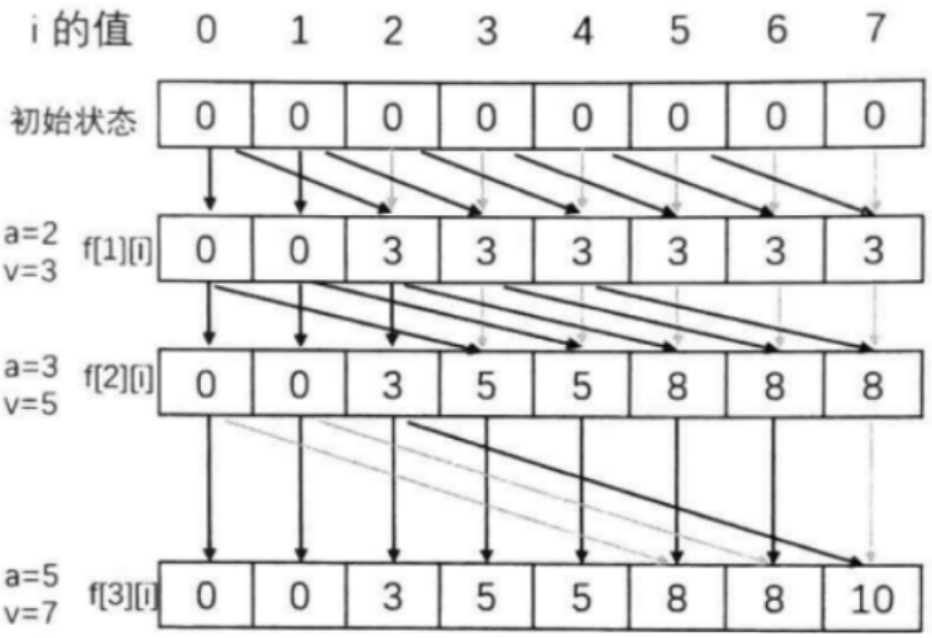
$$dp[i][j] = \begin{cases} dp[i-1][j], & j < v[i] \\ max(d[i-1][j], dp[i-1][j-v[i]] + w[i]), & j \geq v[i] \end{cases}$$

这样讲还是有些抽象我们来举一个具体的例子

比如我们假定这样一组数据 $v = 7$，$N = 3$

```
7 3
2 3
3 5
5 7
```

我们不难发现状态的转移过程应该如图所示



i 表示 $v=i$ 时所能取得的最大价值

通过这个图我们不难发现每一行的状态只和前一行有关系，即第 0 行元素的更新值放到第 1 行，第 1 行元素的更新值放到第 2 行，以此类推。

这样我们可以把选物品这一维直接省略掉。

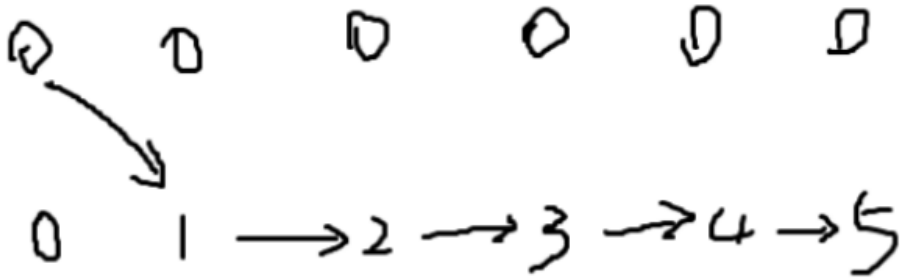
于是我们得到

$dp[j]=max(dp[j],dp[j-v[i]]+w[i]),j\geq v[i]$

但这里不能按下面代码来写：

```
for i in range(1, n + 1):
    for j in range(v[i], m + 1):
        f[j] = max(f[j], f[j - v[i]] + w[i])
```

看起来好像没什么问题，我们用下面这个样例试一下,会发现得到的结果不是 1 而是 5。



```
5 1
1 1
```

如果 j 从小到大遍历，那么会先更新 $dp[j]$ 再更新 $dp[j+v[i]]$ ，这就导致在更新 $dp[j+v[i]]$ 时使用的是第 i 行的 $dp[j]$ 而非第 $i-1$ 行的 $dp[j]$ ，即当 j 从小到大遍历时，二维数组的递推式变成了：

$$dp[i][j]=\begin{cases}dp[i-1][j], & j<v[i]\\ max(d[i-1][j],dp[i][j-v[i]]+w[i]), & j\geq v[i]\end{cases}$$

所以正确的代码应该只需要把第二层循环颠倒一下顺序就可以了

```
for i in range(1, n + 1):
    for j in range(m, v[i] - 1, -1): # 倒着循环
        f[j] = max(f[j], f[j - v[i]] + w[i])
```

优化后的代码模板如下

```
# 定义常量N为1010
N = 1010

# 输入背包容量m和物品数量n
m, n = map(int, input().split())
```

```python
# 定义数组f，用于动态规划的状态转移
f = [0] * N

# 定义两个数组v和w，分别表示物品的体积和价值
v = [0] * N
w = [0] * N

# 输入每个物品的体积和价值
for i in range(1, n + 1):
    v[i], w[i] = map(int, input().split())

# 动态规划求解
for i in range(1, n + 1):  # 遍历每个物品
    for j in range(m, v[i] - 1, -1):  # 从大到小遍历背包容量
        # 转移方程，更新背包容量为j时的最大价值
        f[j] = max(f[j], f[j - v[i]] + w[i])

# 输出结果，即背包容量为m时的最大价值
print(f[m])
```

完全背包

有 N 件物品和一个容量是 V 的背包，每件物品可以使用无限次。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。 输出最大价值。

暴力思路

我们和 01 背包一样先推转移式

设 $dp[i][j]$的含义是：在背包体积为 j 的前提下，从前 i 种物品中选能够得到的最大价值。

如何计算 $dp[i][j]$ 呢？我们可以将它划分为以下若干部分：

选 0 个第 i 种物品：相当于不选第 i 种物品，对应 $dp[i-1][j]$。
选 1 个第 i 种物品：对应 $dp[i-1][j-v[i]]+w[i]$。
选 2 个第 i 种物品：对应 $dp[i-1][j-2\cdot v[i]]+2\cdot w[i]$。
...

上述过程并不会无限进行下去，因为背包体积是有限的。设第 i 种物品最多能选 t 个，于是可知 $t = \lfloor j/w[i] \rfloor$从而得到递推式：

$dp[i][j] = max_{0 \le k \le t} dp[i-1][j - k \cdot v[i]] + k \cdot w[i]$

代码如下

```
# 定义常量N为1010
N = 1010

n, m = map(int, input().split()) # 输入物品数量n和背包容量m

# 定义二维数组f，用于动态规划的状态转移
f = [[0] * (m + 1) for _ in range(N)]

# 定义数组v和w，分别表示物品的体积和价值
v = [0] * N
w = [0] * N

# 输入每个物品的体积和价值
for i in range(1, n + 1):
    v[i], w[i] = map(int, input().split())

# 动态规划求解
for i in range(1, n + 1): # 遍历每个物品
    for j in range(m + 1): # 遍历背包容量
        for k in range(j // v[i] + 1): # 遍历当前物品的体积倍数，不超过背包容量
            f[i][j] = max(f[i][j], f[i - 1][j - k * v[i]] + k * w[i]) # 更新状态转移方程

print(f[n][m]) # 输出背包容量为m时的最大价值
```

我们这里用了三重循环，考虑一下能否优化？（以下图片来源自之前看过的一篇csdn博客，但不记得作者了不好意思QAQ）

考虑 $dp[i][j]$，此时第 i 种物品最多能选 $t_1 = \lfloor \frac{j}{w[i]} \rfloor$ 个，将递推式展开：

$$\begin{aligned} dp[i][j] = \max(&dp[i-1][j], dp[i-1][j-w[i]] + v[i], \\ &dp[i-1][j-2\cdot w[i]] + 2\cdot v[i], \\ &\vdots \\ &dp[i-1][j-t_1\cdot w[i]] + t_1\cdot v[i]) \end{aligned}$$

下面考虑 $dp[i][j-w[i]]$，此时第 i 种物品最多能选 $t_2 = \lfloor \frac{j-w[i]}{w[i]} \rfloor = \lfloor \frac{j}{w[i]} - 1 \rfloor = t_1 - 1$ 个，相应的递推式为

$$\begin{aligned} dp[i][j-w[i]] = \max(&dp[i-1][j-w[i]], dp[i-1][j-w[i]-w[i]] + v[i], \\ &dp[i-1][j-w[i]-2\cdot w[i]] + 2\cdot v[i], \\ &\vdots \\ &dp[i-1][j-w[i]-t_2\cdot w[i]] + t_2\cdot v[i]) \end{aligned}$$

我们再把 $t1 = t2 + 1$ 代入上式可化简为

$$\begin{aligned} dp[i][j-w[i]] = \max(&dp[i-1][j-w[i]], dp[i-1][j-2\cdot w[i]] + v[i], \\ &dp[i-1][j-3\cdot w[i]] + 2\cdot v[i], \\ &\vdots \\ &dp[i-1][j-t_1\cdot w[i]] + (t_1-1)\cdot v[i]) \end{aligned}$$

这样我们再把这个化简的式子代回 $d[i][j]$ 的式子中，我们会惊人的发现 $dp[i][j] = max(dp[i-1][j], dp[i][j-w[i]] + v[i])$，核心代码如下

```
for i in range(1, n + 1): # 遍历每个物品
    for j in range(m + 1): # 遍历背包容量
        f[i][j] = f[i - 1][j]
        if j - v[i] >= 0:
            f[i][j] = max(f[i - 1][j], f[i][j - v[i]] + w[i])
```

但这个式子好眼熟，难道说。。。没错是不是我们之前错误的那个滚动优化01背包的式子

```
# 定义常量N为1010
N = 1010

n, m = map(int, input().split()) # 输入物品数量n和背包容量m

# 定义数组f，用于动态规划的状态转移
f = [0] * N

# 定义数组v和w，分别表示物品的体积和价值
```

```
v = [0] * N
w = [0] * N

# 输入每个物品的体积和价值
for i in range(1, n + 1):
    v[i], w[i] = map(int, input().split())

# 动态规划求解
for i in range(1, n + 1): # 遍历每个物品
    for j in range(v[i], m + 1): # 遍历背包容量
        f[j] = max(f[j], f[j - v[i]] + w[i]) # 更新状态转移方程

print(f[m]) # 输出背包容量为m时的最大价值
```

蓝桥杯真题

费用报销（蓝桥杯C/C++2022B组国赛）

这种可以放又可以不放，每次只能放一个的题目，都可以想到经典的01背包问题。但是有一点变形的地方在于需要考虑日期的问题，也就是两个日期之间的差值是不能小于K天的。

那么我们是不是可以先对日期进行一下排序，日期靠前的在前面，那么我们就套用经典的背包转移就好了，也就是说我们设$f[i][j]$代表第$1\sim i$个物品占用体积为j的时候的最大价值。

注意这个时候的$1\sim i$是对于已经经过日期排序后的数组的，然后我们还需要找到距离当前日期之前的最近的差值大于等于K天的日期的下标假设这个下标是idx

那么我们每次转移的式子就是：

$$f[i][j]\leftarrow f[idx][j-v[i]]+w[i]$$

这样子直接转移就好了，最终的复杂度是$O(NM)$

```
class Ren:
    def __init__(self, m, d, v, t):
        self.m = m
```

```
        self.d = d
        self.v = v
        self.t = t

n, m, k1 = map(int, input().split())
dx = [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
s = [0] * 13
for i in range(2, 13):
    s[i] = s[i - 1] + dx[i - 1]

d = [None] * 1005
f = [[0] * 5005 for _ in range(1005)]
lst = [0] * 5005

for i in range(1, n + 1):
    month, day, value = map(int, input().split())
    time = s[month] + day
    d[i] = Ren(month, day, value, time)
d[0] = Ren(0, 0, 0, 0)

# Remove None values before sorting
d = [x for x in d if x is not None]
d = sorted(d, key=lambda x: x.t)

for i in range(1, n + 1):
    for j in range(i):
        if d[i].t - d[j].t >= k1:
            lst[i] = j

for i in range(1, n + 1):
    for j in range(m, -1, -1):
        f[i][j] = f[i - 1][j]
        if j >= d[i].v:
            f[i][j] = max(f[i][j], f[lst[i]][j - d[i].v] +
d[i].v)

print(f[n][m])
```

搬砖（蓝桥杯C/C++2022B组国赛）

这种选或者不选的问题可以往经典的01背包上面去想，首先需要注意有个条件，也就是说 并且对于塔中的每一块砖来说，它上面所有砖的重量和不能超过它自身的价值 ，我们用w_i表示每块砖的重量，用v_i表示每块砖的价值，然后用W_i来表示第i块砖块上面的砖块的重量之和。那么我们这个限制条件就可以公式化的表示为$W_{i-1} \leq v_i$。

接着我们就需要想办法在满足这个条件的情况下，我们怎么去**贪心**的放是最好的了。

首先我需要说明的一点是，为什么需要贪心，不是背包DP就满足所有的情况都考虑了吗?

没有限制条件的背包DP确实能考虑所有摆放的条件，但是因为我们加了一个限制条件$W_{i-1} \leq v_i$，所以我们套用之前的背包DP的方法，或者说我们两次for循环枚举，都是要从上到下去摆放砖块的，也就是说我前面的$f[i][j]$算出来的结果也是对后面的算出来的结果有影响的。实际上可以想到我们选或者不选这种01背包DP，也是一种求子集的问题，但是这是一种条件子集。我们可以举一个例子。

```
10 1
8 2
1 10
```

可以看到我们有三块砖，我们如果按照当前的顺序进行01背包在限制条件的情况下，第一块砖的w_i等于10，那么后面的砖都无法进行摆放了，因为都不满足限制的条件。

那么我们怎么进行排序呢?

显然，$v_i - W_{i-1}$ 越小，第 i 块砖利用得越充分。

如何使这个值变小呢?

设现有满足条件的一些砖，考虑将第 i 块与第 $i+1$ 块砖互换。

	交换前	交换后
第 i 块砖	$v_i - W_{i-1}$	$v_i - W_{i-1} - w_{i+1}$
第 $i+1$ 块砖	$v_{i+1} - W_i$	$v_{i+1} - W_{i-1}$

因此，最值得的构造是使 $\forall 1 \leq i \leq n-1, w_i + v_i < w_{i+1} + v_{i+1}$。在这种情况下，$v_i - w_{i+1} < v_{i+1} - w_i$，$v_i - W_{i-1} - w_{i+1} < v_{i+1} - W_i$。

即：按 $w_i + v_i$ 从小到大排序。

剩余部分为 0-1 背包，我们设置$f[i][j]$表示的是在当前排序下，第$1 \sim i$块砖头占用体积为j所能贡献出来的最大价值。

我们发现如果我们要求满足$W_{i-1} \leq v_i$这个条件，那么我们需要在第二个for循环里面枚举体积j的时候设置上$j - a[i].w \geq a[i].v$而我们的j又是从$a[i].w$开始枚举的，所以可以确定第二维度的枚举范围$j \in [a[i].w, a[i].w + a[i].v]$

直接转移即可

```
N = 1005
M = 20005

def cmp(a, b):
    return a['w'] + a['v'] < b['w'] + b['v']

n = int(input())
a = []
for _ in range(n):
    w, v = map(int, input().split())
    a.append({'w': w, 'v': v})

a.sort(key=lambda x: x['w'] + x['v'])

f = [[0] * M for _ in range(n + 1)]
ans = 0

for i in range(1, n + 1):
    for j in range(1, M):
        f[i][j] = f[i - 1][j]
    for j in range(a[i - 1]['w'], a[i - 1]['w'] + a[i - 1]['v']
+ 1):
        f[i][j] = max(f[i][j], f[i - 1][j - a[i - 1]['w']] +
a[i - 1]['v'])
        ans = max(ans, f[i][j])

print(ans)
```

这样已经可以满分通过此题，但是我们又发现实际上本题的$w_i \leq 20$非常的小，所以这样才没有MLE，考虑能否滚动数组优化，发现能不能取跟到底前几个没有关系，那么我们直接滚动数组优化一下即可。

最终时间复杂度为$O(NM)$空间复杂度为$O(M)$

```
N = 1005
M = 20005

class Node:
    def __init__(self, w, v):
        self.w = w
        self.v = v

def cmp(a, b):
    return a.w + a.v < b.w + b.v

n = int(input())
a = []
for _ in range(n):
    w, v = map(int, input().split())
    a.append(Node(w, v))

a.sort(key=lambda x: x.w + x.v)

f = [0] * M
ans = 0

for i in range(1, n + 1):
    for j in range(a[i - 1].v + a[i - 1].w, a[i - 1].w - 1,
-1):
        f[j] = max(f[j], f[j - a[i - 1].w] + a[i - 1].v)
        ans = max(ans, f[j])

print(ans)
```

习题

[小蓝与抽奖]

[小蓝与小红]