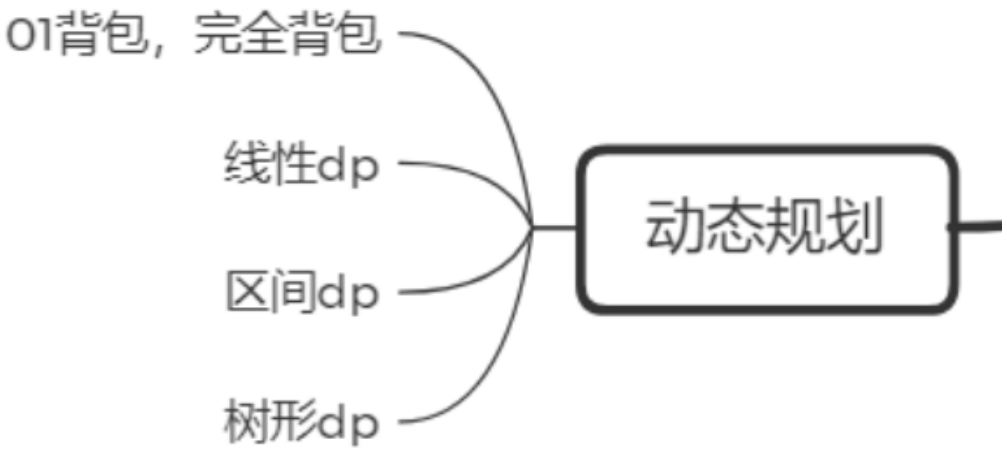
蓝桥杯十天冲刺省一



Day-6 动态规划

前置知识：搜索、贪心、递归以及递推。（感谢 心存侥幸 授权）

在同学们学习动态规划之前，请同学们先掌握基础的搜索以及递归和递推，这对我们学习动态规划有着很好的作用。注意学习动态规划与贪心的区别，所以需要同学们熟悉简单贪心算法（没学过也没事想想超市怎么找零钱的）。

引入：

相信在还没有学习动态规划之前，很多同学已经听说过关于动态规划的传说甚至有的同学还做过一些动态规划的题目，但是都未真正见过动态规划的真面目。动态规划一直都是算法比赛中最常用的思想，也是算法比赛中的重中之重，同时他也是一类非常难的问题。

小学阶段的动态规划题目都是一些非常简单的题目，但是我们需要学习遇到这类题目的时候我们为什么考虑选择用动态规划而不用其他的算法呢？希望同学们怀着这个问题去学习本章内容。

本阶段我们先讲动态规划中两个最简单的问题，他们是背包问题以及线性动态规划问题。

动态规划核心是"状态缓存"，将一个状态的统计结果传递到另外一个状态，这有一点类似递推算法。动态规划是算法竞赛的常客，非常考研同学们问题分析和代码实现能力。

什么是动态规划：

纸币问题

某国有$n(n \leq 1000)$种纸币，每种纸币面额为正整数a_i并且有无限张，试问使用至少多少张纸币可以凑出$w(w \leq 10000)$的金额。

例如，提供面额 1、5、10、20、50、100 元的面额的纸币，要凑足 15 元，至少需要 2 张(1张5元和1张10元)；提供 1、5、11 元的纸币，要凑足15元，则至少需要3张(3张5元)。

分析：第一个样例代表了生活中纸币的面额。按照经验，找钱时尽量先用大面额，超过了再用较小面额的，以此类推。对于第一个样例应用上述贪心思路，得到 $15 = 10 + 5$，两种纸币恰好是最优解。然而这个贪心策略对于第二个例子就行不通了。按照贪心策略我们会得到 $15 = 11 + 1 + 1 + 1 + 1$，使用了 5 张纸币。但是最优策略却不是，贪心做法先入僵局。

贪心策略的最大问题在于鼠目寸光：它过分关注了最大面额，全程没有给5任何机会，那么在正式思考这个问题之前，不妨先来看看如何使用搜索结果第二个样例：

```
#include <bits/stdc++.h>
using namespace std;
int dfs(int w){
    if(w == 0){
        return 0;
    }
    if(w < 0){
        return INT_MAX;
    }
    return min(dfs(w - 1), min(dfs(w - 5), dfs(w - 11))) + 1;
}
int main(){
    cout << dfs(15);
    return 0;
}
```

令 `dfs(w)` 的返回值为凑够w元最少需要的纸币数。在每个阶段，分别尝试使用一张 1 元纸币、一张 5 元纸币、一张 11 元纸币，分别对剩下的金额继续搜索，即 `dfs(w - 1)`、`dfs(w - 5)`和`dfs(w - 11)` 。在这三种情况下，我们找到需要纸币总张数最小的情况，加 1 返回即可(因为每次尝试都是尝试使用一张)。

思考为什么 $w < 0$ 返回一个无穷大的值。因为$w < 0$是不合法的，我们直接让他返回一个很大的值就可以使用 `min` 函数判断掉了，即永远不会选到正无穷这个值。

这段程序又短又好写，但是它的效率好低，因为产生了重复计算。考虑一个问题 `dfs(w)` 的值对于同一个w来说，是不是固定的呢？显然，对于凑过10元钱的最少纸币数只与纸币的面额有关，不会因为调用 `dfs(10)` 的先后顺序而改变。换句话说， `dfs(w)` 的返回值是恒定不变的。调用同一个w它的 `dfs(w)` 值一定相同。所以我们可以看出 `dfs(w)` 的作用更像是数组。

我们在仔细看递归的过程，实际上发现我们一直都是从当前的w一直向下递归到 0，对于上面的例子，我们可以看出 `dfs(w)` 的时候只考虑和 `dfs(w - 1), dfs(w - 5), dfs(w - 11)` 有关。
这是否给了我们启示？如果我们将 `dfs(w)` 看成数组 `dp[w]`，可以得出：

- `dfs(w) = min(dfs(w - 1), dfs(w - 5), dfs(w - 11)) + 1` 即三个最小的 + 1即可

等价于：

- `dp[w] = min(dp[w - 1], dp[w - 5], dp[w - 11]) + 1`

所以我们发现，由于每次需要知道更小的值，那我们可以考虑从$1-15$开始，把每一步的 `dp[w]` 算出来即可。

并且由于dfs的特性，我们会一直优先搜索 $w-1$，然后搜索 $w-5$ 最后搜索 $w-11$，等价于我们对于同一种金额开始for循环,设 `a[j]` 为可以选择的金额（1、5、11），从 1 循环到 15 为止（表示一直选择该金额，对应 `dfs(w-a[j])`），即 `dp[w] = dp[w - a[j]]`。

```
#include <bits/stdc++.h>
using namespace std;
int dp[20]; // dp[i]表示凑够 i 的金额最少需要多少纸币
int main(){
    int a[4] = {0, 1, 5, 11};
    memset(dp, 127, sizeof dp); // 因为我们要取最小值，赋值无穷大即可
    dp[0] = 0; // 凑够 0 的金额不需要纸币 或者说需要 0张纸币
    for(int i = 1; i <= 15; i ++ ){ // 计算dp[i] 最小是1 最大是 15 因为dp[15]表示凑够15的金额
        for(int j = 1; j <= 4; j ++ ){ // 看一下我当前在用哪一种纸币找到最小的dp[i - a[j]] + 1
            if(a[j] <= i){ // 如果 i 要是 比 a[j] 小 说明会减成负数，是不合法的 不用考虑
                //理解dp数组含义，dp[i] 表示凑了 i 金额的最小面额，凑够当前的i元可能从 i - 1 i - 5 i - 11来
                // 距离 dp[15] = dp[10] + 1 dp[10] = dp[15 - 5] 为什么减5？因为我们当前选择a[j]是5 考虑放一张5元可不可以
                //  15元可能是从10元的基础上加了1张5元来的
                // 这里的a[j] 就是 1、5、11 面额的纸币
                // 注意要和自己比较一下，因为dp[15]可能是dp[11 + 1 + 1 + 1 +1]也可能是dp[10 + 5]来的
                //要看看是已经算过的小还是新来的小 因为我们面额也是从小开始枚举，可能会先算了dp[1 + 1 + 1....+ 1]的情况
```

```
                dp[i] = min(dp[i], dp[i - a[j]] + 1);
            }
        }
    }
    cout << dp[15]; // 表示凑够15元最小数量
    return 0;
}
```

`dp[0] = 0` 其实就是边界，这里采用了 dp_w 表示凑够 w 的金额最少需要多少张纸币，取代了几乎意义相同的 `dfs(w)` 。在计算 `dfs(w)` 的时候，我们枚举了当前阶段用了哪张纸币，继续搜索，然后选择答案最优的那个。这里同理，在计算 dp_w 的时候也是枚举了这个阶段用了哪张纸币，只不过使用的是之前的答案计算了 dp_w，最终得到目标答案。

分析完上面思路，请同学们好好参考注释理解，并且考虑如果现在让同学们手动输入n张面额和求出w金额的最少纸币数量应该怎么写呢?

这样通过把原问题分解成若干相关子问题，在利用已知的答案依次计算未知问题的答案，最终得到原问题答案的方式，称之为**动态规划（Dynamic Programming，简称DP）**。

动态规划适用基本条件：

一：具有 相同子问题

- 首先，我们必须要保证这个问题能够分解出几个子问题，并且能够通过这些子问题来解决这个问题。并且这些子问题也能被分解成相同的子问题进行求解。例如上方例子：我们可以通过 `dp[w - 1]` `dp[w - 5]` `dp[w - 11]` 最后得到 `dp[w]` ,同样队伍 `dp[w - 1]` 来说也可以通过 `dp[w - 1 - 1]` `dp[w - 1 - 5]` `dp[w - 1 - 11]` 得到。
- 也就是说，假设我们一个问题被分解成了 A、B、C 三个部分，那么这个 A、B、C 三个部分也可以被分解成 A'、B'、C' 三个部分，而不能是 D、E、F 三个部分。

二：满足 最优子结构

- 问题的最优解包含着它的子问题的最优解。即不管前面的策略如何，此后的决策必须是基于当前状态(由上一次决策产生)的最优决策。例如：`dp[w]` 是由 `dp[w - 1]` `dp[w - 5]` `dp[w - 11]` 中最优解来的，那么 `dp[w]` 也应该是最优解。

三：满足 无后效性

- "过去的步骤只能通过当前状态影响未来发展，当前状态是**历史的总结**"。这条特征说明了**动态规划只适用于解决当前决策与过去状态无关的问题。**状态，出现在策略的任何位置，它的地位相同，都可以实施同样的策略，这就是无后效性的内涵。举例：当我们求出 `dp[5]` 的时候，后面的 `dp[10]` 等都可以通过同样的策略来解决，而不会因为 `dp[5]` 的出现，使用了不一样的策略.
- 反过来，现在状态应该与前面状态无关，试想一下，如果第i个状态的改变改变了前面的第j个状态，是不是转移到第j个状态的所有状态都会被改变，这样就不满足无后效性。
- 这是动态规划中极为重要的一点，如果当前问题的具体决策，会对解决其他未来的问题产生影响，如果产生了影响就无法保证决策最优性。

做动态规划的一般步骤

- **First，结合原问题和子问题确定状态**
  - 题目在求什么？要求出这个值我们需要知道什么？什么是影响答案的因素？
  - 状态参数一般有：描述位置、描述数量、描述最大最小等。
- **Second，确定转移方程**
  - **初始化边界是什么**
  - 注意无后效性。
  - 根据状态**最后一次决策**就可以推出状态转移方程。

线性DP

引入：

1.爬楼梯

爬楼梯类型的问题可谓是线性DP的入门题目以及经典中的经典。我们先来看一下题目。

题目分析

首先我们想一下，如果我们要走到第N阶楼梯，前一步应该在那一阶楼梯上?
因为每一次能走两个楼梯或者一个楼梯，所以上一步肯定在 $N-1$ **或者** $N-2$ **楼梯上**。

DP数组的设计

我们令 $DP[N]$ 表示走到第 N 阶楼梯的总方案数

状态转移与边界条件

我们走到第 N 阶楼梯的总方案数应该是所有走到 $N-2$ 阶楼梯的方案数(走两个楼梯) + 所有走到 $N-1$ 阶楼梯的方案数(走一个楼梯)

```
DP[N] = DP[N - 1] + DP[N - 2];
```

参考代码

```
#include<bits/stdc++.h>
using namespace std;
long long a[45];
int main()
{
    int n,i,m;
    a[1]=1;
    a[2]=2;
    for(i=3;i<=40;i++)
    {
        a[i]=a[i-1]+a[i-2];
    }
    cin>>n;
    for(i=1;i<=n;i++)
    {
        cin>>m; //读题  三获已经在第1级 需要走的级数是m-1
        cout<<a[m-1]<<endl;
    }
}
```

2.数字三角形

数字三角形和爬楼梯是类似的，只不过“爬”的范围变成二维平面了，本质上是没有区别的。

题目分析

首先我们将数据重新排列一下
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5
我们可以发现当我们走到第 i,j 个数据的时候，我们上一步一定是从 $i-1,j-1$ 或者 $i-1,j$ 走过来的

DP数组的设计

我们令 $DP[i][j]$ 表示走到 i,j 的最大值

状态转移与边界条件

这个题目和走楼梯的不同的是，**并不是让我们计算方案数，而是计算最大值**，最后一行的 DP 数组为 $dp[N][i]$,我们还要在最后一行中找出其中最大的那个才行

```
DP[i][j] = max(DP[i - 1][j - 1], DP[i - 1][j]) + a[i][j];
```

参考代码

```
#include <bits/stdc++.h>
using namespace std;
long long dp[101][101];
long long maxx = 0;
int a[101][101];
long long max(long long a, long long b)
{
    if (a > b)
        return a;
    else return b;
    return 0;
```

```
}
int main() {
    int n;
    cin >> n;
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= i; j++)
            dp[i][j] = 0;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= i; j++)
        {
            cin >> a[i][j];
        }
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= i; j++)
            dp[i][j] = max(dp[i - 1][j - 1], dp[i - 1][j]) + a[i]
[j];//状态转移
    for (int i = 1; i <= n; i++)
        maxx = max(maxx, dp[n][i]);//还要在最后一行找最大值
    cout << maxx;
    return 0;
}
```

3.充电

有些题目正序不容易看出来状态是怎么转移的，如果倒过来看的话反而会会更加容易求解

题目分析

我们分析这道题目的时候会发现一个问题
这个题目**好像不满足** dp **的无后效性**，选一个的话就要放弃后面几个，明显是当前的决策会影响到后续的决策。
其实对于这样一个线性的 DP 问题，我们既可以从左往右看，也可以从右往左看，这道题如果从右往左看的话无疑会简单很多：
当前第 n 个充电桩我们有两个决策：**选这个充电或者不充电**，如果不充电的话，我们的收益应该不变。如果充电的话，我们当前的收益**受到右边某个位置的收益的影响，而我们是从右往左** dp **的，不会对我们未来的决策产生影响**

dp数组的设计

我们令 $DP[N]$ 表示**从右到左**第N个充电桩为止的最大收益

状态转移与边界条件

我们走到第 N 个充电桩的收益取决于不选(和到 $N+1$ 的方案收益一样)，或者选(右边某个坐标的收益加上自己的收益)，而我们最终的收益为 $dp[1]$(**因为是从右到左 dp 的，倒序最后一个就是正序的第一个**)

```
dp[i] = max(dp[i], a[i] + (c > n ? 0 : dp[c]));
//c为右边某个充电桩，当c大于n时为0（因为该位置越界了，收益肯定为0）
```

参考代码

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e6 + 5;
long long a[N], b[N];
long long dp[N];
int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i] >> b[i];
    }
    dp[n + 1] = 0;//初始收益为0
    for (int i = n; i >= 1; i--) {
        dp[i] = dp[i + 1];//拒绝
        int c = i + b[i] + 1;
        dp[i] = max(dp[i], a[i] + (c > n ? 0 : dp[c]));//接受
    }
    cout << dp[1];
}
```

4.最长公共子序列

LCS 是线性 DP 中的经典题目之一

题目分析

如果用暴力法求解该题目，时间复杂度将会达到一个天文数字（指数级），所以这道题需要用 DP 求解，关键是数组的设计和状态转移。

DP数组的设计

我们令$dp[i][j]$表示 A 序列前 i 个元素和 B 序列前 j 个元素的最长公共子序列

例子：

状态转移与边界条件

当我们判断 `dp[i][j]` 时，正好是以A序列第i个元素结尾和B序列第j个元素结尾。自然而然会有两种情况

- $a[i] == b[j]$
  那么公共序列长度会加 1

```
dp[i][j] = dp[i - 1][j - 1] + 1;
```

- $a[i] != b[j]$
  那么公共序列长度并没有变化，**但是 $dp[i][j]$ 此时不应该等于 $dp[i-1][j-1]$,而是应该会变成两个子问题：A 序列的前 i 个和 B 序列的前 $j-1$ 个以及 A 序列的前 $i-1$ 个和 B 序列的前 j 个**。

```
dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]);
```

最终我们求解的问题也应该是 $dp[i][j]$

参考代码

```
#include <bits/stdc++.h>
using namespace std;
int dp[250][250];
int main() {
    string str1, str2;
    cin >> str1;
    cin >> str2;

    dp[0][0] = 0;
```

```
    for (int i = 1; i <= str1.size(); i++)
        for (int j = 1; j <= str1.size(); j++)
            if (str1[i - 1] == str2[j - 1])
                dp[i][j] = dp[i - 1][j - 1] + 1;
            else
                dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]);

    cout << dp[str1.size()][str2.size()];
}
```

5.最长不下降子序列

LIS 也是一种经典的题型，重点在于 dp 数组的设计

题目分析

这道题麻烦在我们不知道这个最长的数列是以那个元素结尾的（最大的元素不一定是结尾的那个元素）

DP数组的设计

我们令$dp[i]$表示**以序列第i个元素结尾的序列的长度**

例子：

状态转移与边界条件

当我们判断到 $dp[i]$ 的时候，若要令以i结尾的序列最大，肯定要找一个j，**且这个a[j]一定是小于a[i]的(保持递增)**，所以

```
dp[i] = max(dp[j]) + 1;//0 < j < i && a[j] < a[i]
```

我们并不知道这个序列是以那个元素结尾的，所以我们最终所求的结果为

```
for (int i = 1; i <= n; i++)
    maxx = max(maxx, dp[i]);
```

参考代码

```cpp
#include <bits/stdc++.h>
using namespace std;
int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
    return 0;
}
int main() {
    //dp数组设计为以当前数字结尾的最长不下降子序列的长度
    int n, maxx = INT_MIN;
    int a[1001], dp[1001];
    cin >> n;
    for (int i = 1; i <= n; i++)
        cin >> a[i];
    for (int i = 1; i <= n; i++)
        dp[i] = 1;
    for (int i = 2; i <= n; i++)
        for (int j = 1; j < i; j++)
            if (a[i] >= a[j])
            {
                dp[i] = max(dp[i], dp[j] + 1);
                maxx = max(dp[i], maxx);
            }
    cout << maxx;
}
```

6.编辑距离

同样经典的 DP 类型题目，乍一看可能没思路，同样需要仔细分析

题目分析

这道题给了我们三个操作，然后问我们要用最少的操作来改变其中一个串使得这个串和另一个串一模一样，看似复杂，实则分析每个状态后问题将会变得更加清晰

DP数组的设计

我们令 $dp[i][j]$ 表示把 A 序列前 i 个字符转换成 B 序列前 j 个字符所需要的操作步骤

状态转移与边界条件

然后我们不断地判断当前的 $a[i]$ 和 $b[j]$

- 如果该两个字符相等
  说明不用任何操作

```
dp[i][j] = dp[i - 1][j - 1]
```

- 如果这两个字符不相等的话
  如果当前字符对应不上，会有三种操作，**其中对于插入和删除操作我们要比较 $dp[i-1][j]$和 $dp[i][j-1]$ 这两种之前的状态谁的代价更低**
  插入 `dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + 1`
  删除 `dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + 1`
  替换 `dp[i][j] = dp[i - 1][j - 1] + 1`
- 初始化
  对于任意 `dp[0 ][i]`或者 `dp[i ][0]`来说，无论你要进行什么操作使两个序列相同，最少次数**都是** i ,因为其中一个序列长度为0

参考代码

```
#include <bits/stdc++.h>
using namespace std;
int min(int a, int b)
{
    if (a > b)
        return b;
    else
        return a;
    return 0;
```

```
}
int dp[2005][2005];
int main() {
    //字符编辑距离
    //如果当前字符对应不上
    //插入dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + 1
    //删除dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + 1
    //替换abc//abd  dp[i][j] = dp[i - 1][j - 1] + 1
    string str1, str2;
    cin >> str1 >> str2;
    for (int i = 0; (i <= str1.size() || i <= str2.size()); i++)
    {
        dp[i][0] = i;
        dp[0][i] = i;
    }
    for (int i = 1; i <= str1.size(); i++)
        for (int j = 1; j <= str2.size(); j++)
        {
            if (str1[i - 1] != str2[j - 1])
                dp[i][j] = min((min(dp[i - 1][j], dp[i][j - 1]) +
1), (dp[i - 1][j - 1] + 1));
            else
                dp[i][j] = dp[i - 1][j - 1];
        }
    cout << dp[str1.size()][str2.size()];
}
```

练习题

接龙数列（蓝桥杯C/C++2023B组省赛）

蜗牛（蓝桥杯Java2023B组省赛）

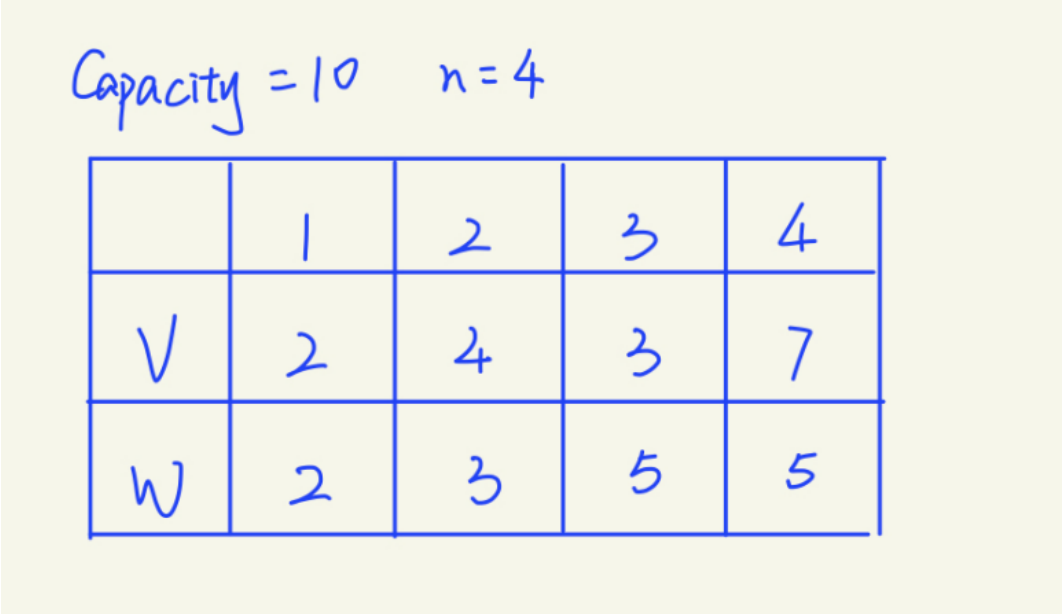
李白打酒加强版（蓝桥杯C/C++2023B组省赛）

背包问题选讲

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。

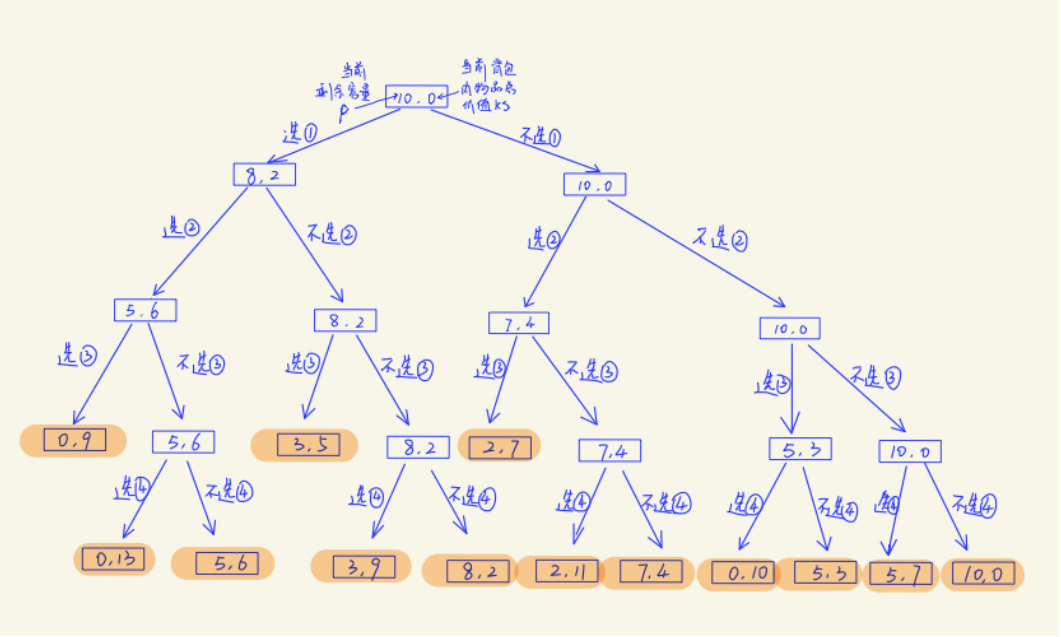


重新思考一个问题，为什么我们把他叫做 $0/1$ 背包呢?

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：

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;
// dp[i][j] 表示从前 i 个物品中选体积不超过 j 的最大价值
int dp[40][210], w[40], v[40];// w和v分别表示物品的重量和价值
int main(){
    int n, V; // n表示有多少种物品，V背包的体积
    cin >> V >> n;
    for(int i = 1; i <= n; i ++ ){ // 输入每种物品的重量和价值
        cin >> w[i] >> v[i];
    }
    dp[0][0] = 0; // 一件物品也不选，他的价值是0 容量也是0 边界值
    for(int i = 1; i <= n; i ++ ){ // 枚举选择第几件物品
        for(int j = 0; j <= V; j ++ ){ // 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]);
            }
        }
    }
    cout << dp[n][V]; // 从前n个物品中选，体积不超过V的最大价值
    return 0;
}
```

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：

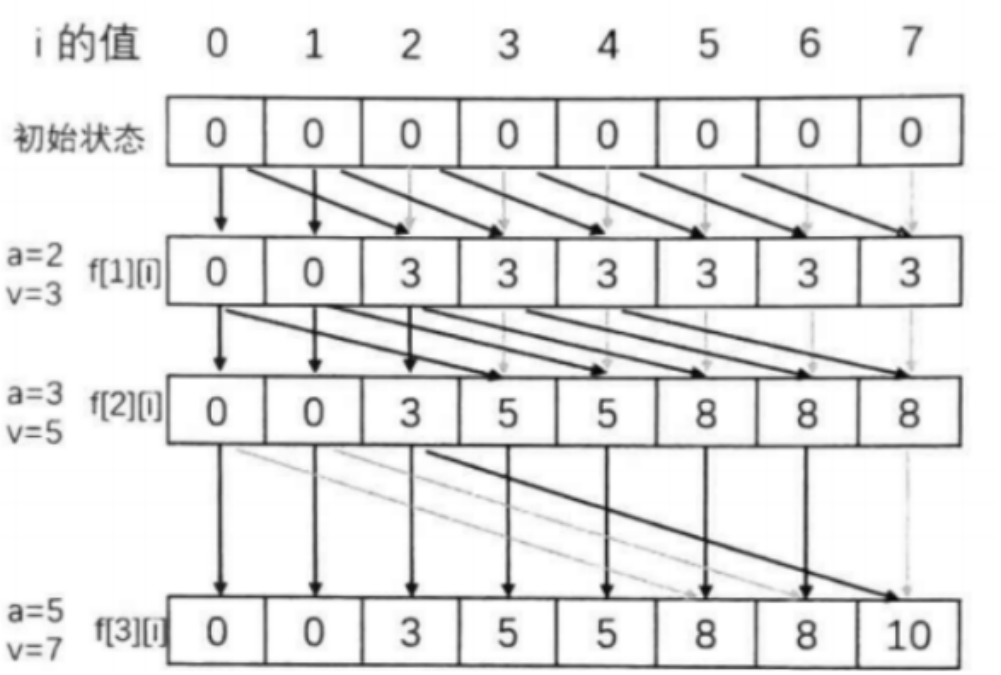
$$dp[i][j]=\begin{cases}dp[i-1][j], & j<v[i]\\ max(d[i-1][j],dp[i-1][j-v[i]]+w[i]), & j\geq v[i]\end{cases}$$

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。

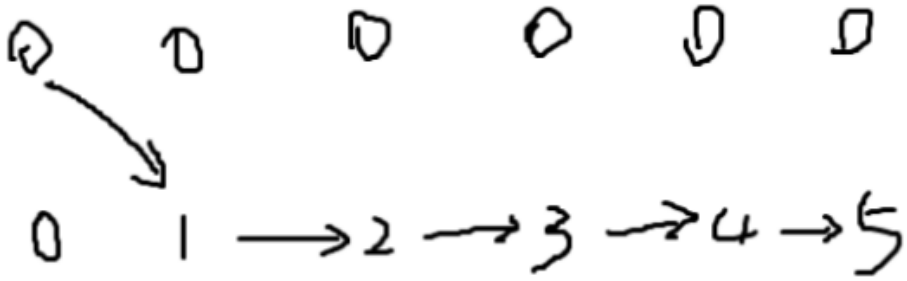
于是我们得到

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

但这里不能按下面代码来写：

```
for(int i = 1;i <= n;i++) {
    for(int j = v[i];j <= m;j++) {
        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(int i = 1;i <= n;i++) {
    for(int j = m;j >= v[i];j--) {//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

```cpp
#include<bits/stdc++.h> // 包含C++标准库中的所有头文件

using namespace std; // 使用标准命名空间

const int N = 1010; // 定义常量N为1010

int n,m; // 定义整型变量n和m，分别表示物品的数量和背包的容量
int f[N]; // 定义数组f，用于动态规划的状态转移
int v[N],w[N]; // 定义两个数组v和w，分别表示物品的体积和价值

int main() { // 主函数入口

    // 输入背包容量m和物品数量n
    cin >> m >> n;

    // 输入每个物品的体积和价值
    for(int i = 1;i <= n;i++) {
        cin >> v[i] >> w[i];
    }

    // 动态规划求解
    for(int i = 1;i <= n;i++) { // 遍历每个物品
        for(int j = m;j >= v[i];j--) { // 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
    cout << f[m];

    // 返回0，表示程序执行成功
    return 0;
}
```

拓拓的套餐

熟悉 01 背包和二维费用背包问题的同学应该可以看出来，这是二维费用的 01 背包问题。

用 $f[i][j][k]$ 表示前 i 个物品第一个约束条件取 j 个第二个约束条件取 k 个的最小方案

然后可以仿照背包问题将 i 那一维给删掉，本题数据范围很小，不删除空间也足够。

注意这里是至少，因此转移方程为

$$f[j][k]=\min(f[j][k],f[\max(0,j-a[i].x)][\max(0,k-a[i].y)]+1)$$

参考代码：

```
#include<bits/stdc++.h> // 包含C++标准库中的所有头文件

const int N = 1e4 + 5; // 定义常量N为10005

using namespace std; // 使用标准命名空间

pair<int, int> a[N]; // 定义pair数组a，用于存储食品的量
int f[N][N]; // 定义二维数组f，用于动态规划的状态转移

int main() { // 主函数入口

    int n, ft, sd, sum1 = 0, sum2 = 0; // 定义整型变量n、ft、sd和sum1、sum2，分别表示食品的种类、第一种食品的量、第二种食品的量以及两种食品的总量
    cin >> n; // 输入食品种类数
    cin >> ft >> sd; // 输入第一种食品和第二种食品的量
    for(int i = 1; i <= n; i++) { // 遍历每种食品
        cin >> a[i].first >> a[i].second; // 输入每种食品的量
        sum1 += a[i].first; // 统计第一种食品的总量
        sum2 += a[i].second; // 统计第二种食品的总量
    }

    if(sum1 < ft || sum2 < sd)  // 如果第一种或第二种食品的总量小于所需量
        cout << "-1" << endl; // 输出-1，表示无解
    else {
        memset(f, 0x3f, sizeof f); // 将数组f初始化为最大值
        f[0][0] = 0; // 初始化f[0][0]为0
        for(int i = 1; i <= n; i++) // 遍历每种食品
            for(int j = 300; j >= 0; j--) // 遍历第一种食品的量
                for(int k = 300; k >= 0; k--) // 遍历第二种食品的量
                    f[j][k] = min(f[j][k], f[max(0, j - a[i].first)][max(0, k - a[i].second)] + 1); // 更新状态转移方程
        int ans = 0x3f3f3f3f; // 定义ans为最大值
```

```
        for(int i = ft; i <= 300; i++) // 遍历第一种食品的量
            for(int j = sd; j <= 300; j++) // 遍历第二种食品的量
                ans = min(ans, f[i][j]); // 更新最小值
        cout << ans << endl; // 输出最小解
    }
    return 0; // 程序结束
}
```

练习题

USACO 2.2.2 Subset Sums 集合

完全背包

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 $dp[i-1][j-2\cdot v[i]]+2\cdot w[i]$。
$\cdots$

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

```
#include <iostream> // 包含输入输出流头文件

using namespace std; // 使用标准命名空间
```

```cpp
const int N = 1010; // 定义常量N为1010

int f[N][N]; // 定义二维数组f，用于动态规划的状态转移
int v[N], w[N]; // 定义数组v和w，分别表示物品的体积和价值

int main() { // 主函数入口

    int n, m; // 定义整型变量n和m，分别表示物品的数量和背包的容量
    cin >> n >> m; // 输入物品数量n和背包容量m

    // 输入每个物品的体积和价值
    for (int i = 1; i <= n; i++) {
        cin >> v[i] >> w[i];
    }

    // 动态规划求解
    for (int i = 1; i <= n; i++) { // 遍历每个物品
        for (int j = 0; j <= m; j++) { // 遍历背包容量
            for (int k = 0; k * v[i] <= j; k++) { // 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
            }
        }
    }

    cout << f[n][m] << endl; // 输出背包容量为m时的最大价值

    return 0; // 程序结束
}
```

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：

$$dp[i][j] = \max(dp[i-1][j],\ dp[i-1][j-w[i]] + v[i],\\ dp[i-1][j-2\cdot w[i]] + 2\cdot v[i],\\ \vdots\\ dp[i-1][j-t_1\cdot w[i]] + t_1\cdot v[i])$$

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

$$dp[i][j-w[i]] = \max(dp[i-1][j-w[i]],\ dp[i-1][j-w[i]-w[i]] + v[i],\\ dp[i-1][j-w[i]-2\cdot w[i]] + 2\cdot v[i],\\ \vdots\\ dp[i-1][j-w[i]-t_2\cdot w[i]] + t_2\cdot v[i])$$

我们再把 $t1 = t2 + 1$ 代入上式可化简为

$$dp[i][j-w[i]] = \max(dp[i-1][j-w[i]],\ dp[i-1][j-2\cdot w[i]] + v[i],\\ dp[i-1][j-3\cdot w[i]] + 2\cdot v[i],\\ \vdots\\ dp[i-1][j-t_1\cdot w[i]] + (t_1-1)\cdot v[i])$$

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(int i = 1 ; i <=n ;i++) {
    for(int j = 0 ; j <=m ;j++)
    {
        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

```
#include <bits/stdc++.h> // 包含C++标准库中的所有头文件

using namespace std; // 使用标准命名空间

const int N = 1010; // 定义常量N为1010

int f[N]; // 定义数组f，用于动态规划的状态转移
```

```
int v[N], w[N]; // 定义数组v和w，分别表示物品的体积和价值

int main() { // 主函数入口

    int n, m; // 定义整型变量n和m，分别表示物品的数量和背包的容量
    cin >> n >> m; // 输入物品数量n和背包容量m

    // 输入每个物品的体积和价值
    for (int i = 1; i <= n; i++) {
        cin >> v[i] >> w[i];
    }

    // 动态规划求解
    for (int i = 1; i <= n; i++) { // 遍历每个物品
        for (int j = v[i]; j <= m; j++) { // 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
        }
    }

    cout << f[m] << endl; // 输出背包容量为m时的最大价值

    return 0; // 程序结束
}
```

练习题：

Money Systems 货币系统

```
#include <bits/stdc++.h> // 包含C++标准库中的所有头文件

using namespace std; // 使用标准命名空间

long long a[30]; // 定义长整型数组a，用于存储每个物品的体积
unsigned long long dp[30][10000 + 10]; // 定义无符号长整型二维数组dp，用于动态规划的状态转移

int main() { // 主函数入口

    int v, n; // 定义整型变量v和n，分别表示物品的种类和背包的容量
    cin >> v >> n; // 输入物品种类v和背包容量n
```

```
    // 输入每个物品的体积
    for (int i = 1; i <= v; i++) {
        cin >> a[i];
    }

    // 初始化dp数组，当背包容量为0时，方案数均为1
    for (int i = 0; i <= v; i++) {
        dp[i][0] = 1;
    }

    // 动态规划求解
    for (int i = 1; i <= v; i++) { // 遍历每个物品
        for (int j = 1; j <= n; j++) { // 遍历背包容量
            if (j >= a[i]) { // 如果当前背包容量大于等于当前物品的体积
                dp[i][j] = dp[i - 1][j] + dp[i][j - a[i]]; // 更新状
态转移方程
            } else {
                dp[i][j] = dp[i - 1][j]; // 否则，方案数不变
            }
        }
    }

    cout << dp[v][n]; // 输出背包容量为n时的方案数

    return 0; // 程序结束
}
```

布尔型背包

限制条件：在 01 背包或完全背包的要求的基础上，还要求能 **恰好装满背包**。这里以完全背包代码为例：

```
const int N = 1e4+5;
int n, t, m, v[N], w[N];
int dp[N];
int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++)
        cin >> w[i] >> v[i];
    dp[0] = 1;    // 装满 0 体积的最大价值是 0
```

```
    for(int i = 1; i <= n; i++)
        for(int j = w[i]; j <= m; j++)   // for(int j = m; j >= w[i]; j--)   ---- 01 背包
        {
            if(dp[j-w[i]])  // 布尔判断: j-w[i] 能装满才可以进行更新 ---- dp[j-w[i]] != 0: 能装满 j-w[i]
                dp[j] = max(dp[j], dp[j-w[i]]+v[i]);
        }
    if(dp[m] == 0)
        cout << "NO";
    else
        cout<<dp[m]-1<< endl;
    return 0;
}
```

多重背包（了解）

同学们主要理解完全背包和 01 背包即可，多重背包大家可以自行理解一下。

有 N 种物品和一个容量是 V 的背包。

第 i 种物品最多有 s_i 件，每件体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。输出最大价值。

同学们可以看一下，多重背包和01背包和完全背包的区别在哪里?

细心的同学已经发现了，我们分组背包它每组物品是有限制的，不是无限个，那我们对于某一个物品，我们可以怎么样选择呢?

是不是对于某一个物品我们可以选择 0个、1个、2个、3个、....、s_i个，于是我们发现，状态转移方程和完全背包没有进行优化的时候一模一样。

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

代码：

```
for(int i = 1; i <= n; i ++ ){
    for(int k = 0; k <= s[i]; k ++ ){
        for(int j = 0; j <= m; j ++ ){
            // 最多可以选择 s[i] 个 k 就是 0 ~ s[i]
            if(k * v[i] <= j){
                dp[i][j] = max(dp[i][j], dp[i -1][j - k * v[i]] + k
* w[i]);
            }
        }
    }
}
```

对于多重背包问题的优化，我们这里不在阐述，感兴趣的同学在我们以后的学习中，会和大家讲到。

对于多重背包的习题，大家先将例题做好即可。

分组背包（了解）

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 v_{ij}，价值是 w_{ij}，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。

大家再来看看，分组背包和其他背包的区别是什么？有细心的同学已经注意到了，分组背包就是有很多组物品，然后每一组物品只能取一个。只能选一个？大家有没有恍然大悟!这不就是 01 背包吗？所以分组背包很简单，就是对于每一个组，我们组内做01背包即可，问题是如何分组，我们可以用 `vector<int> a` 也可以用 `w[i][j] 表示第i组第j个物品的重量`，根据大家的喜好可以自行选择。

状态转移方程：

不选第 i 组物品的第 j 个，如果选的话要留出 `w[i][j]` 的空间。

`dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i][j]] + v[i][j]` 其中 `w[i][j]` 表示第 i 组第 j 个物品的重量， `v[i][j]` 表示第 i 组物品第 j 个物品的价值

代码：

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

分组背包我们目前阶段也不需要掌握，大家做会练习题即可。

到此我们的背包基本问题都已经给同学们分析完毕了，希望同学们好好理解 01背包 和 完全背包 ， 01背包 是所有背包的入门。

区间DP（了解）

区间DP的做法较为固定，即枚举区间长度，再枚举左端点，之后枚举区间的断点进行转移。

区间类型动态规划是线性动态规划的拓展，它在分阶段划分问题时，与阶段中元素出现的顺序和由前一阶段的哪些元素合并而来有很大的关系。（例：$f[i][j]=f[i][k]+f[k+1][j]$）

区间类动态规划的特点：

- 合并：即将两个或多个部分进行整合。
- 特征：能将问题分解成为两两合并的形式。
- 求解：对整个问题设最优值，枚举合并点，将问题分解成为左右两个部分，最后将左右两个部分的最优值进行合并得到原问题的最优值。

```
// 区间动态规划模板:
for(int len=1;len<=n;len++)    // 枚举区间长度
{
    for(int l=1,r;(r=l+len)<=n;l++)   // 枚举左端点和对应右端点
    {
        //do something
        for(int k=l;k<r;k++)
        {
            //更新 DP 数组，例如: dp[l][r] = min(dp[l][r],dp[l]
[i]+dp[i+1][r])
        }
    }
}
```

合并石子

```
/******author:ssxyz******/
#include <bits/stdc++.h>
using namespace std;

const int N = 410;

int n;
int a[N], s[N];
int f[N][N]; //f[i][j]表示合并[i,j]区间的最小总分
int g[N][N]; //g[i][j]表示合并[i,j]区间的最大总分

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++)
    {
        cin >> a[i];
        a[i + n] = a[i];
    }
    for(int i = 1; i <= 2*n; i++)
    {
        s[i] = s[i - 1] + a[i];
    }

    for(int len = 2; len <= n; len++) //枚举区间长度
```

```
    {
        for(int i = 1; i + len - 1 < 2*n; i++) //枚举左端点
        {
            int j = i + len - 1; //右端点
            f[i][j] = 1e9;
            for(int k = i; k < j; k++)
            {
                f[i][j] = min(f[i][j], f[i][k] + f[k+1][j] + s[j] -
s[i-1]);
                g[i][j] = max(g[i][j], g[i][k] + g[k+1][j] + s[j] -
s[i-1]);
            }
        }
    }
    int maxv = 0, minv = 1e9;
    for(int i = 1; i <= n; i++) //枚举长度为n的所有合并答案
    {
        minv = min(minv, f[i][i+n-1]);
        maxv = max(maxv, g[i][i+n-1]);
    }
    cout << minv << endl << maxv;
}
```

练习题

合并石子（蓝桥杯JavaB组2023年省赛）

树形DP（了解）

p>之前我们学过的DP，绝大多数是解决线型问题、区间问题或者网格问题的。在解决线性或者区间问题的时候，我们常常以前 i 个位置作为一个子问题，或者以区间 $[i,j]$ 作为一个子问题。现在我们要学习一种全新的动态规划，它是以树或者图为模型的动态规划。它一般以子树作为子问题，或者以拓扑序上靠后的那部分作为子问题，以每个结点作为状态。

在树上设计动态规划算法时，一般就以节点从深到浅（子树从小到大）的顺序作为 DP 的“阶段”DP 的状态表示中，第一维通常是节点编号（代表以该节点为根的子树）。大多数时候，我们采用递归的方式实现树形动态规划。对于每个节点 x，先递归在它的每个子节点上进行 DP，在回溯时，从子节点向节点 x 进行状

态转移去推父结点的状态信息，边界条件则是直接由叶子节点确定。整体上采用**后序遍历**的方法求解状态。

没有上司的舞会

因为题目要求选出的点中不能有两个点具有父子关系，也就是选了父亲，那么所有儿子都不能选；没选父亲，那么儿子才可以选。

如果我们确定父亲不选，那各个儿子和它的子树就构成了一个子问题。如果父亲确定选，那就构成了确定儿子不选的一个子问题。

所以设计状态的时候，需要将当前的结点选还是不选给设计进去。我们设 $f[i][0/1]$ 表示以 i 号点为根的子树在 i 号点不选/选的情况下，能选出的最大权值和。如果 i 号点我们没选，那对于它的每个儿子我们都有选或不选两种选择，对应的 dp 值也就是 $f[son(i)][0]$ 和 $f[son(i)][1]$ 取两者中较大的那一个。如果 i 号点我们选了，那么对于它的每个儿子都只有不选一个选择，对应的dp值就是 $f[son(i)][0]$。自然，转移的时候就可以写出

$$\begin{cases} f[i][0] = \sum_{j=son(i)} \max(f[j][0], f[j][1]) \\ f[i][1] = \sum_{j=son(i)} f[j][0] \end{cases}$$

我们发现，在所有可能的 4 种转移中唯独缺少了 $f[j][1] \to f[i][1]$，这其实也就对应题目要求的父子结点不能同时被选的情况。

由于每个结点只会被遍历一次，因此时间复杂度为 $O(n)$。

参考代码：

```
#include<bits/stdc++.h>
const int N = 6005;
int n, u, v, r[N], in[N], f[N][2];
// f[i][0]: 不选 i 时，i 子树上的最大欢乐值
// f[i][1]: 选 i 时，i 子树上的最大欢乐值
vector<int> g[N];
int dfs(int u) {
    f[u][0] = 0;
    f[u][1] = r[u];  // 初始化当前结点状态
    for(auto v : g[u]) {
        f[u][0] += dfs(v);
        f[u][1] += f[v][0];
    }
    return max(f[u][0], f[u][1]);
}
```

```
int main(){
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> r[i];
    while(cin >> u >> v) {
        if(!u&&!v) break;
        g[v].push_back(u);
        in[u]++;
    }
    for(int i =1; i <= n; i++)
        if(in[i] == 0) {
            cout << dfs(i);
            break;
        }
    return 0;
}
```

二叉苹果树

因为要求剩下 q 条边，所以剩下多少条边是需要写进状态里面的。根据树形 DP 的一般特点，可以设出：$f[i][j]$ 表示以 i 为根节点的子树上可以选 j 条边时，边上苹果之和的最大值，易得 $f[i][0]=0$。转移时，每次枚举左子树选择了 k 条边，右边剩 $j-k-?$ 条边(注意，选择左/右子树时，必选父子之间的树边，因此这里用 $?$ 来表示不确定)。

记左孩子为 l_i，左边树枝上苹果数为 w_l，右孩子为 r_i，右边树枝上苹果数为 w_r，可以得到转移：

$$f[i][j]=\max_{1\le k\le j-2}\{f[l_i][k]+f[r_i][j-k-2]+w_l+w_r, f[l_i][j-1]+w_l, f[r_i][j-1]+w_r\}$$

上面的转移分别对应：把边分配到左右子树、不选右子树、不选左子树。

由于在枚举每个结点时，对每个状态都花了 $O(j)$ 的时间枚举怎么分配边，所以时间复杂度为 $O(nq^2)$。

```
#include <bits/stdc++.h> // 包含C++标准库中的所有头文件

using namespace std; // 使用标准命名空间

const int N = 1e2 + 5; // 定义常量N为105

int n, q, f[N][N]; // 定义整型变量n、q，以及二维数组f，用于动态规划的状态转移
struct Node { // 定义结构体Node，表示每个节点的信息
```

```
    int v, w; // 节点的编号和权值
};
vector<Node> g[N]; // 定义向量数组g，存储每个节点的子节点信息

void dfs(int u, int fa) { // 深度优先搜索函数，参数u表示当前节点，fa表示父节点
    f[u][0] = 0; // 初始化当前节点的状态，背包容量为0时，最大价值为0
    int l = -1, r = -1; // 记录左右子树节点的下标
    for(int i = 0; i < g[u].size(); i++) { // 遍历当前节点的所有子节点
        int v = g[u][i].v; // 获取子节点的编号
        if(v == fa) continue; // 如果子节点是父节点，则跳过
        if(l == -1) l = i; // 记录左子树节点的下标
        else r = i; // 记录右子树节点的下标
        dfs(v, u); // 递归处理子节点
    }
    if(l == -1) return; // 如果没有子节点，则直接返回
    for(int j = 1; j <= q; j++) { // 遍历背包容量
        // 分配任务时考虑左右子树都有边的情况
        for(int k = 0; q >= 2 && k + 2 <= j; k++) { // 遍历分配给左右子树的任务量
            f[u][j] = max(f[u][j], f[g[u][l].v][k] + f[g[u][r].v][j - k - 2] + g[u][l].w + g[u][r].w); // 更新状态转移方程
        }
        f[u][j] = max(f[u][j], f[g[u][l].v][j - 1] + g[u][l].w); // 只分配给左孩子
        f[u][j] = max(f[u][j], f[g[u][r].v][j - 1] + g[u][r].w); // 只分配给右孩子
    }
}

int main() { // 主函数入口
    cin >> n >> q; // 输入节点数量n和背包容量q
    for(int u, v, w, i = 1; i < n; i++) { // 输入每条边的起点、终点和权值
        cin >> u >> v >> w; // 输入边的信息
        g[u].push_back({v, w}); // 将边的信息存储到起点的子节点列表中
        g[v].push_back({u, w}); // 将边的信息存储到终点的子节点列表中
    }
    dfs(1, 0); // 从根节点开始深度优先搜索
    cout << f[1][q]; // 输出根节点的背包容量为q时的最大价值
    return 0; // 程序结束
```

```
}
```

练习题

左孩子右兄弟（蓝桥杯Python2021年省赛）