动态规划基础——线性、背包、区间

邓丝雨

牛客竞赛
AC.NOWCODER.COM

引入1：斐波拉契数列

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9	10
F(n)	1	1	2	3	5	8	13	21	34	55	89

递归 vs 递推

递归版本:　求F(n)

```
int f(int n)
{
      if (n==0 || n==1)  return 1;
      else  return f(n-1) + f(n-2);
}
```

太慢!需要优化

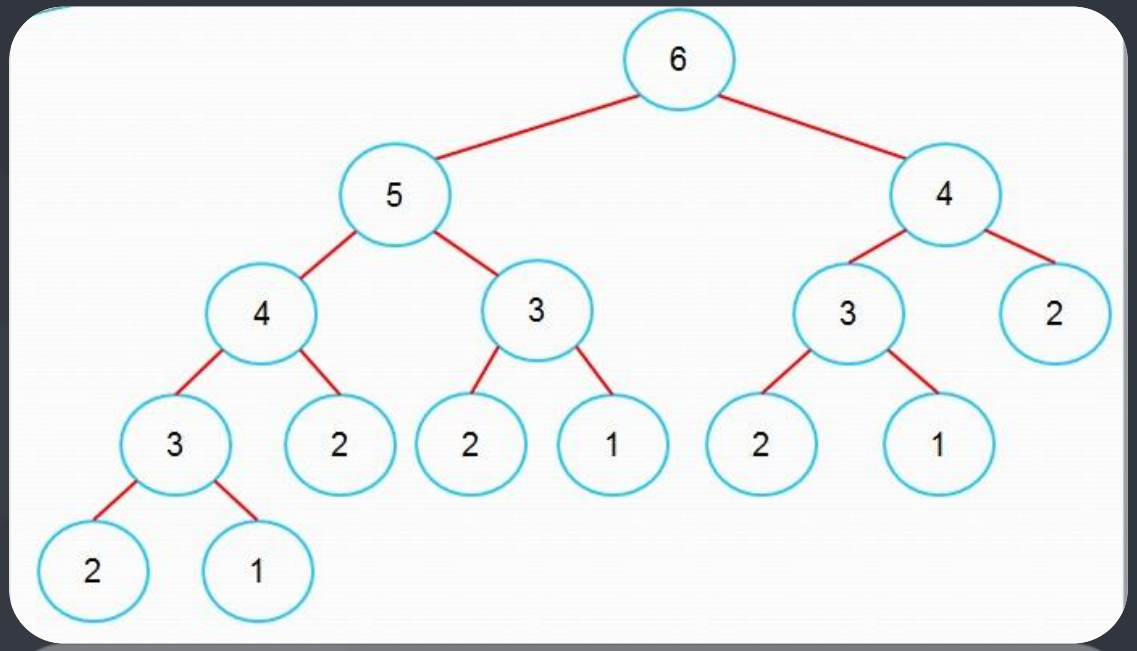
递推: 求F(n)

```
A[0] = A[1] = 1;
for (i =2; i <= n ; i++)
      A[i]=A[i-1] + A[i-2];
return A[n];
```

递归版本：　求F(n)

```
int f(int n)
{
    if (n==0 || n==1)  return 1;
    else  return F(n-1) + F(n-2);
}
```

为什么太慢？？



重复子问题
导致算法效率低下

怎么办?

- 空间换时间
- 已经计算过的记录下来避免重复计算!

记忆化搜索版本: 求F(n)

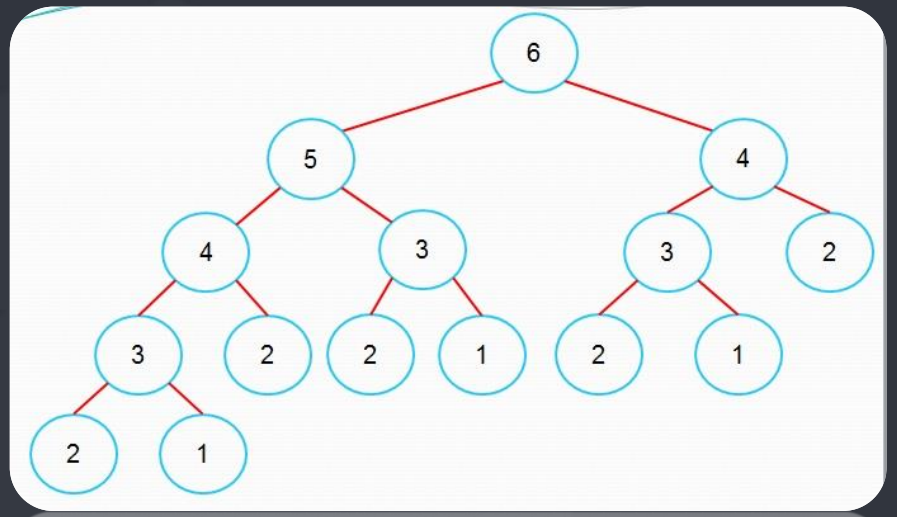
```
1   int  calc  (int n)
2    {
3          if (f([n]!= 0) return f[n];
4         return (f[n] =calc(n-1) + calc(n-2));
5   }
```

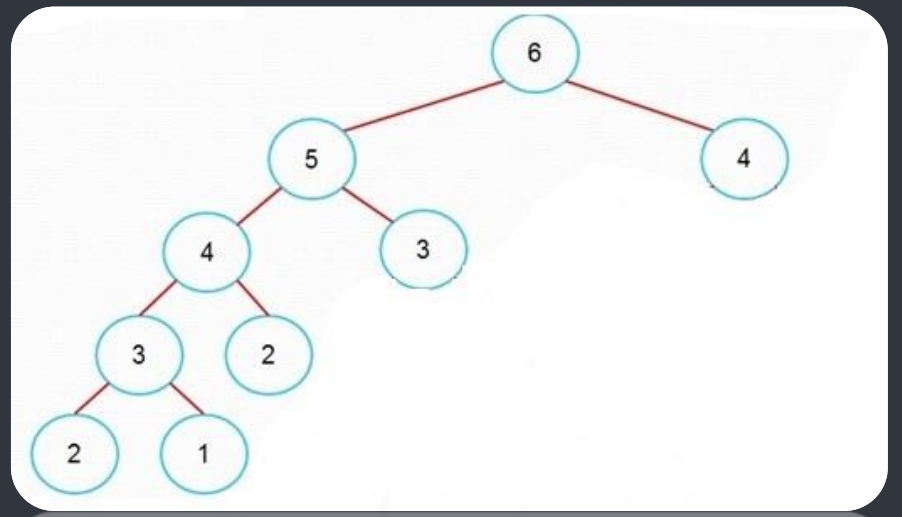
算法复杂度是 O(*n*)

这就是传说中的记忆化搜索了

记忆化搜索

- 记忆化搜索，顾名思义，就是带有记忆化的搜索（这句简直就像废话……）
- 也就是说，用数组等将已经算过的东西记录下来在下一次要使用的直接用已经算出的值，避免重复运算，去掉的重复的搜索树



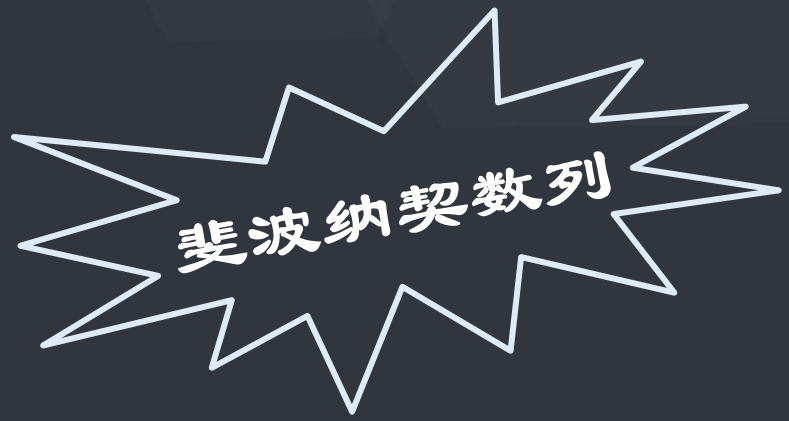


引入2： 走楼梯问题

- 有一人要爬n阶的楼梯，他一次可以爬1阶或2阶，问要爬完这n阶楼梯，共有多少种方法?

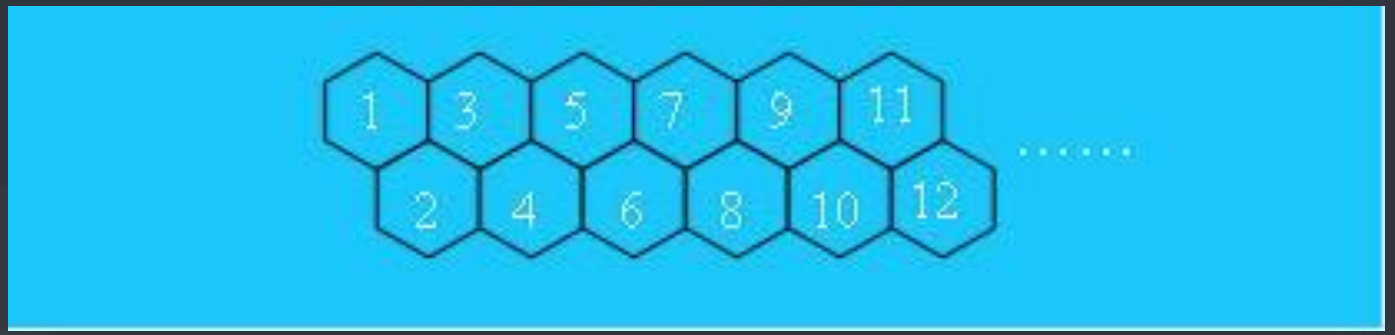
引入2：走楼梯问题

- 假设我们现在在第n阶阶梯上，显然，我们上一步是在n-1阶或者n-2阶，根据分类加法原理，我们可以知道，第n阶的方法=n-1阶的方法+n-2阶的方法
- 同样的，对于n-1阶和n-2阶我们也可以用类似的方法进行求解。
- 而当我们求到1阶和2阶的时候，显然方法种数分别为1、2。
- 所以如果f[i]表示爬到第i阶的方法数，那么
- f[1]=1　f[2] = 2
- f[i] = f[i - 1]+ f[i - 2]



引入3：

- 有一只经过训练的蜜蜂只能爬向右侧相邻的蜂房，不能反向爬行。请编程计算蜜蜂从蜂房a爬到蜂房b的可能路线数。
  其中，蜂房的结构如下所示。



例1：数字三角形

- 设有一个三角形的数塔，顶点结点称为根结点，每个结点有一个整数数值。从顶点出发，可以向左走，也可以向右走。如图所示。

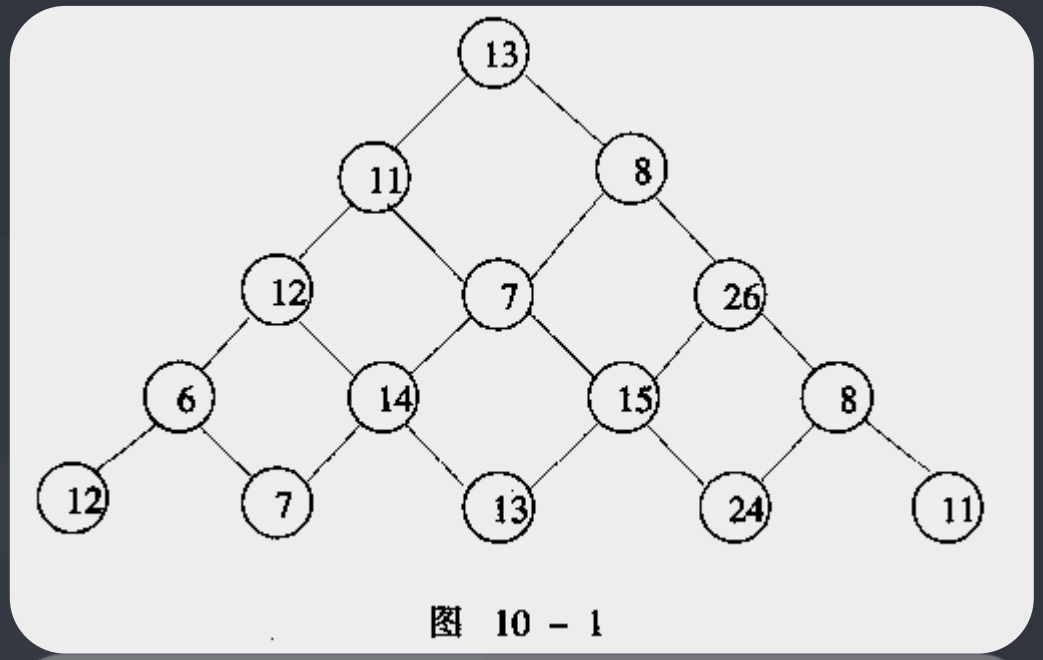


图 10 - 1

问题：
当三角形数塔给出之后，
求从第一层到达底层的
路径最大值。

例1：数字三角形

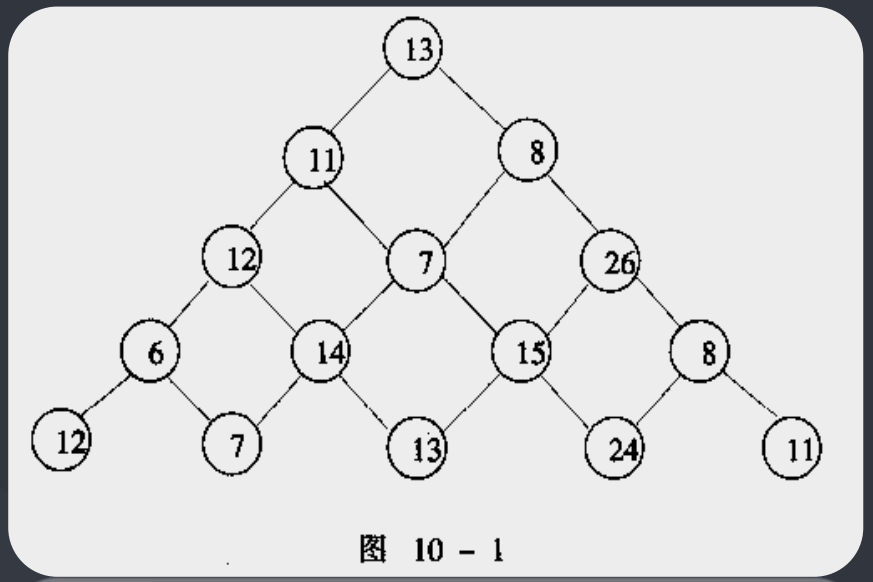


图 10 - 1

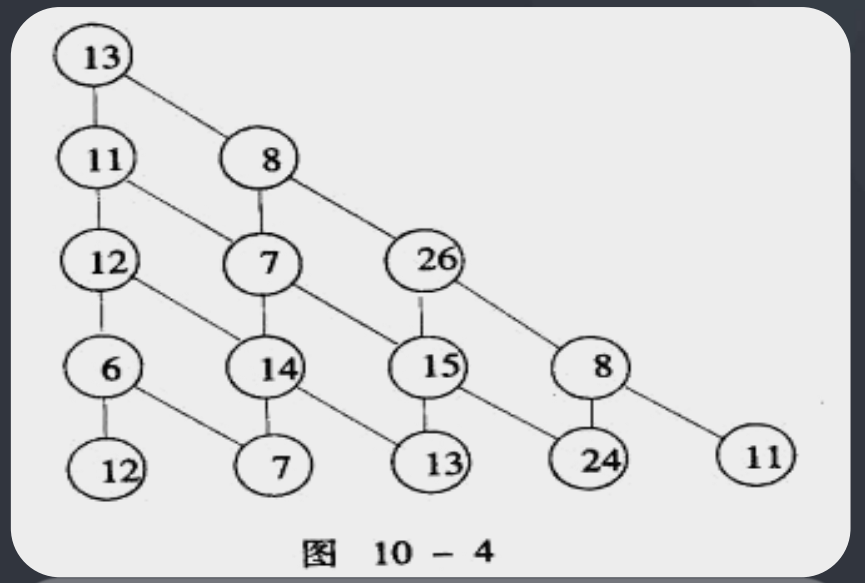
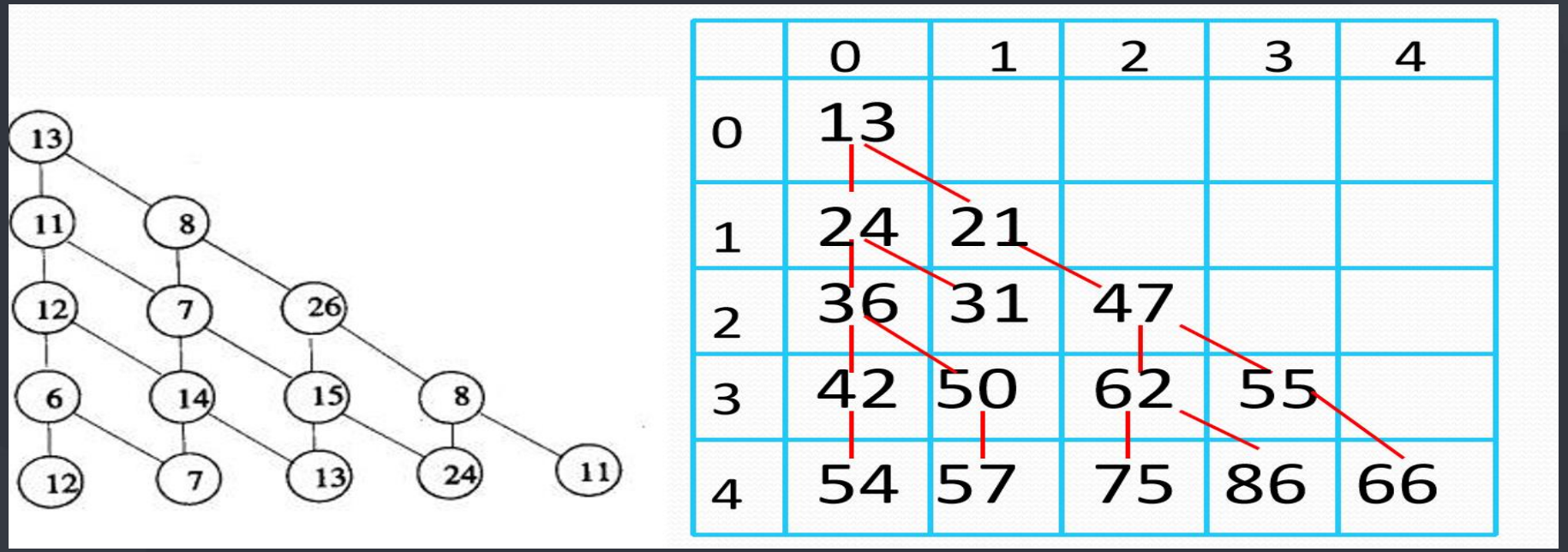


图 10 - 4

例1：数字三角形



	0	1	2	3	4
0	13				
1	24	21			
2	36	31	47		
3	42	50	62	55	
4	54	57	75	86	66

$$f[i,j]=max(f[i-1,j],f[i-1,j-1])+a[i,j]$$

例3：数字三角形

```
for(int i = 0; i < n; i++)
{
    for(int j = 0; j <= i; j++)
    {
        f[i][j] = max(f[i - 1][j], f[i - 1][j - 1]) + a[i][j];
    }
}
```

对最后一行的f值进行扫描，最大的那一个即是结果；
也可以倒着走（从下往上）这样f[1][1]就是结果了。

动态规划原理——加法原理、乘法原理

分类加法原理：

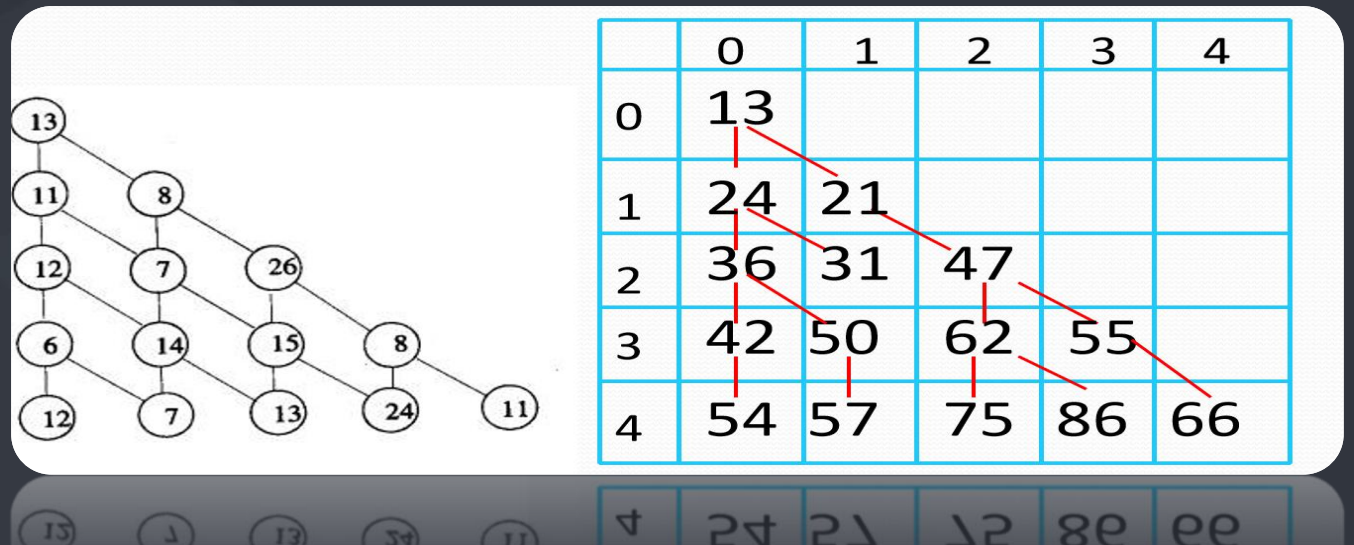
做一件事，完成它可以有n类办法，在第一类办法中有m1种不同的方法，在第二类办法中有m2种不同的方法，……，在第n类办法中有mn种不同的方法，那么完成这件事共有N=m1+m2+m3+…+mn种不同方法。

分步乘法原理：

做一件事，完成它需要分成n个步骤，做第一步有m1种不同的方法，做第二步有m2种不同的方法，……，做第n步有mn种不同的方法，那么完成这件事共有N=m1×m2×m3×…×mn种不同的方法。

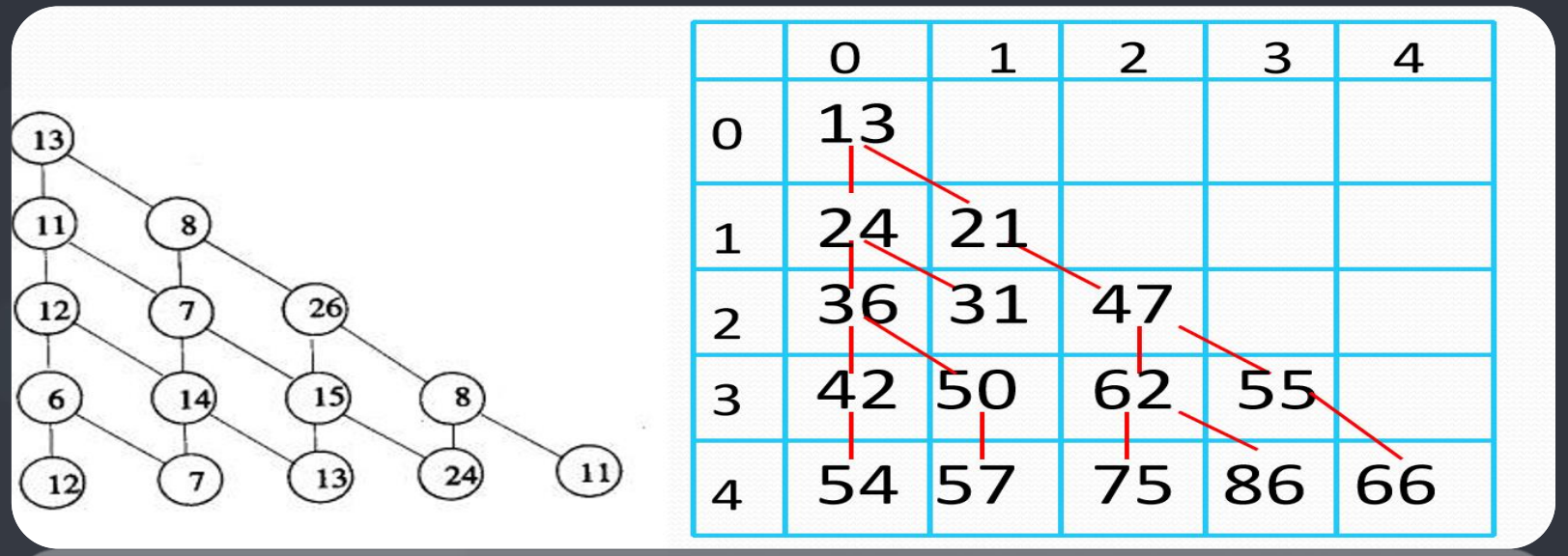
下面给出若干概念

- 动规的定义：
  - 动态规划是解决多阶段决策过程最优化问题的一种方法。
- 阶段：
  - 把问题分成几个相互联系的有顺序的几个环节，这些环节即称为阶段。
- 状态：
  - 某一阶段的出发位置称为状态。通常一个阶段包含若干状态。



下面给出若干概念

- 决策：
  - 从某阶段的一个状态演变到下一个阶段某状态的选择。
- 策略：
  - 由开始到终点的全过程中，由每段决策组成的决策序列称为全过程策略，简称策略。



下面给出若干概念

- 状态转移方程：
  - 前一阶段的终点就是后一阶段的起点，前一阶段的决策选择导出了后一阶段的状态，这种关系描述了由i阶段到i+1阶段状态的演变规律，称为状态转移方程。
  - 形如：

    f[i] = f[i - 1]+ f[i - 2]

    f[i,j]=max(f[i-1,j],f[i-1,j-1])+a[i,j]

    等等

动态规划适用的基本条件

——具有相同子问题

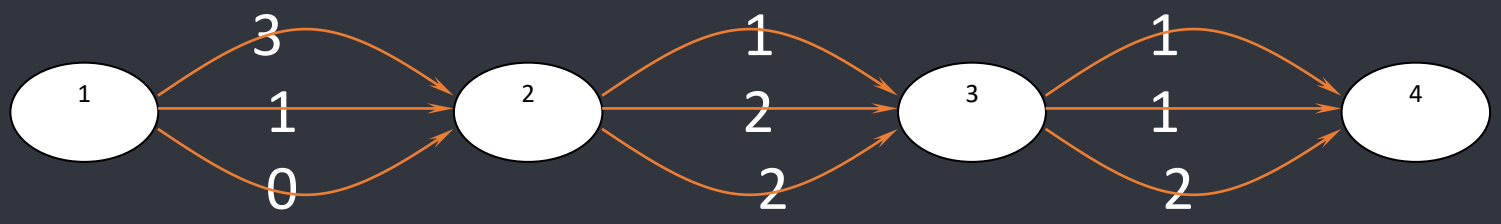
- 首先，我们必须要保证这个问题能够分解出几个子问题，并且能够通过这些子问题来解决这个问题。
- 其次，将这些子问题做为一个新的问题，它也能分解成为相同的子问题进行求解。
- 也就是说，假设我们一个问题被分解为了A,B,C三个部分，那么这A,B,C分别也能被分解为A’,B’,C’三个部分，而不能是D,E,F三个部分。

动态规划适用的基本条件

——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。

反例：



在上图中找出从第1点到第4点的一条路径，要求路径长度mod 4的余数最小。

动态规划适用的基本条件
——满足无后效性

- “过去的步骤只能通过当前状态影响未来的发展，当前的状态是历史的总结”。这条特征说明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，出现在策略任何一个位置，它的地位相同，都可实施同样策略，这就是无后效性的内涵.
- 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，会对解决其它未来的问题产生影响，如果产生影响，就无法保证决策的最优性。

做动态规划的一般步骤

First，结合原问题和子问题确定状态：（我是谁？我在哪？）

- 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?
- （一维描述不完就二维，二维不行就三维四维。）
- 状态的参数一般有
- 1）描述位置的：前(后)i单位，第i到第j单位，坐标为(i,j)，第i个之前（后）且必须取第i个等
- 2）描述数量的：取i个，不超过i个，至少i个等
- 3）描述对后有影响的：状态压缩的，一些特殊的性质

Second, 确定转移方程：（我从哪里来？/我到哪里去？）

- 1）检查参数是否足够；
- 2）分情况：最后一次操作的方式，取不取，怎么样取——前一项是什么
- 3）初始边界是什么。
- 4）注意无后效性。比如说，求A就要求B，求B就要求C，而求C就要求A，这就不符合无后效性了。

根据状态枚举最后一次决策（即当前状态怎么来的）
就可确定出状态转移方程！

Third, 考虑需不需优化

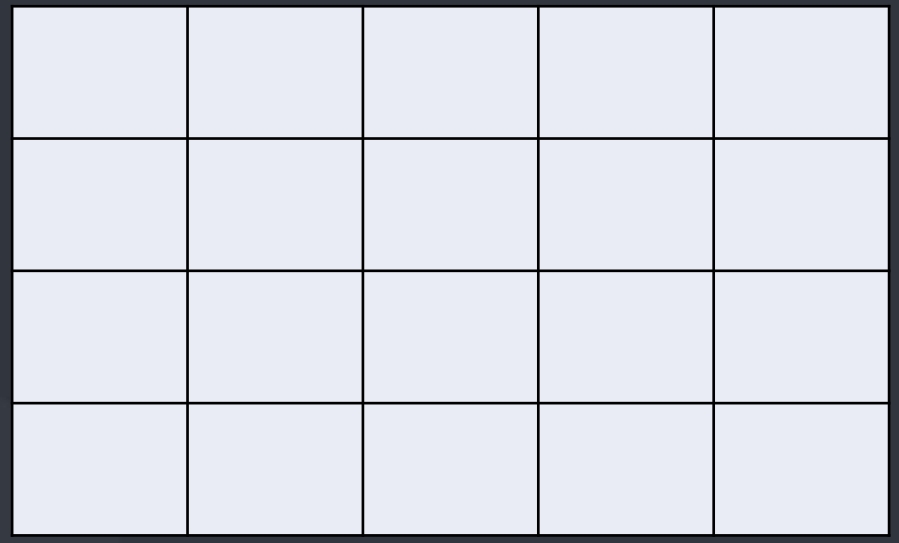
Forth, 确定编程实现方式

- 1）递推
- 2）记忆化搜索

之前的题目都是既可以用递推，
也可以用记忆化搜索，难度相当
而实际上，
有的问题却用记忆化搜索却简单很多解决！

例2：路径条数问题

- N*M的棋盘上左上角有一个过河卒，需要走到右下角。行走的规则：可以向下、或者向右。现在要求你计算出从左上角能够到达右下角的路径的条数。



例2：路径条数问题

- 第一步：确定状态——原问题是什么？子问题是什么？
- 原问题：从(0,0)走到(n,m)的路径数
- 子问题：从(0,0)走到(i,j)的路径数
- f[i][j]表示从左上角走到 (i,j)点的路径条数
- 第二步：确定状态转移方程和边界
- f[i][j] = f[i-1][j] + f[i][j-1]
- f[1][i] =f[i][1]= 1;
- 第三步：考虑是否需要优化
- 第四步：确定实现方法
- （本题其实可以直接用排列组合求）

扩展2.1：过河卒

- 棋盘上A点有一个过河卒，需要走到目标B点。卒行走的规则：可以向下、或者向右。同时在棋盘上C点有一个对方的马，该马所在的点和所有跳跃一步可达的点称为对方马的控制点。因此称之为“马拦过河卒”。
- 棋盘用坐标表示，A点(0, 0)、B点(n, m)(n, m为不超过20的整数)，同样马的位置坐标是需要给出的。
- 现在要求你计算出卒从A点能够到达B点的路径的条数，假设马的位置是固定不动的，并不是卒走一步马走一步。


牛客竞赛
AC.NOWCODER.COM

例3：传球游戏

- n个同学站成一个圆圈，其中的一个同学手里拿着一个球，当老师吹哨子时开始传球，每个同学可以把球传给自己左右的两个同学中的一个（左右任意），当老师再次吹哨子时，传球停止。
- 聪明的小蛮提出一个有趣的问题：有多少种不同的传球方法可以使得从小蛮手里开始传的球，传了m次以后，又回到小蛮手里。两种传球的方法被视作不同的方法，当且仅当这两种方法中，接到球的同学按接球顺序组成的序列是不同的。比如有3个同学1号、2号、3号，并假设小蛮为1号，球传了3次回到小蛮手里的方式有1->2->3->1和1->3->2->1，共2种。

例3：传球游戏

- 第一步：确定状态——原问题是什么？子问题是什么？
- 原问题：从1开始传球第m步球回到1的方法数
- 子问题：从1开始传球第i步球到达j的方法数
- f[i][j]表示第i次传球之后球在第j个人手上的方法数
- 第二步：确定状态转移方程和边界
- f[i][j] = f[i-1][j-1] + f[i-1][j+1]
- f[0][1] = 1;
- 注意由于是一个环，j=1 时 左边（j-1） 为n，j=n 时 右边（j+1）为1
- 第三步：考虑是否需要优化
- 第四步：确定实现方法

例4：最长不下降子序列

- 设有一个正整数的序列：b1,b2,…，bn，对于下标i1<i2<…＜im，若有bi1≤bi2≤…≤bim
- 则称存在一个长度为m的不下降序列。
- 例如，下列数列
- 13 7 9 16 38 24 37 18 44 19 21 22 63 15
- 对于下标i1=1，i2=4，i3=5，i4=9，i5=13，满足13＜16＜38＜44＜63,则存在长度为5的不下降序列。
- 但是，我们看到还存在其他的不下降序列: i1=2，i2=3，i3=4，i4=8，i5＝10，i6=11，i7=12，i8=13，满足：7＜9＜16＜18
- ＜19＜21＜22＜63，则存在长度为8的不下降序列。
- 问题为：当b1,b2,…，bn给出之后，求出最长的不下降序列。

例4：最长不下降子序列

- 第一步：确定状态——原问题？子问题？
- F[i] 前i个数的最长不下降子序列——求不了啊~!为什么求不了?
- 不知道这个序列的最后一个元素是哪个，没法转移
- F[i]以第i个数为结尾的最长不下降子序列
- 第二步：确定状态转移方程
- f[i]=max{f[j]+1}（a[j]<=a[i] 且 j<i）
- f[i] = 1

例5：最长不下降子序列

f[i] = 1
f[i]=max{f[j]+1}（a[j]<=a[i] 且 j<i)

a[i]:

13	7	9	16	38	24	37	18	44	19	21	22

f[i]:

1	1	2	3	4	4	5	4	6	5	6	7

例4：最长不下降子序列

- 第三步：考虑是否需要优化
- O(N^2)
- 可以使用单调队列或者线段树等数据结构优化到O（NlogN） ——留给学有余力的同学
- 第四步：确定实现方法

例4：最长不下降子序列

```
for(i = 0; i < n; i++)
{
   f[i] = 1;
   for(j = 0; j < i; j++)
   {
      if(a[j] <= a[i] && f[j] + 1 > f[i])
         f[i] = f[j] + 1;
   }
}
```

```
int calc(int x)
{
   if (f[x] != 0) return f[x];
   f[x] = 1;
   for (int i = 0; i < x; i++)
   {
      if (a[i] <= a[x])
      {
         t = calc(i);
         if (t + 1 > f[x]) f[x] = t+ 1;
      }
   }
   return f[x];
}
```

例5：滑雪

- Michael喜欢滑雪，这并不奇怪， 因为滑雪的确很刺激。可是为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，你不得不再次走上坡或者等待升降机来载你。Michael想知道载一个区域中最长底滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。下面是一个例子

  1 2 3 4 5

  16 17 18 19 6

  15 24 25 20 7

  14 23 22 21 8

  13 12 11 10 9

  一个人可以从某个点滑向上下左右相邻四个点之一，当且仅当高度减小。在上面的例子中，一条可滑行的滑坡为24-17-16-1。当然25-24-23-...-3-2-1更长。事实上，这是最长的一条。

例5：滑雪

- 第一步：确定状态——原问题？子问题？
- 原问题：从（1，1）到（n，m）任意一个点滑下的最长路径长度
- 子问题：从（i，j）滑下的最长路径长度
- f[i][j]表示从（i，j）滑下的最长路径长度
- 第二步：确定状态转移方程 ——由于f[i][j]由上下左右四个方向转移过来
- F[i][j] = max {f[i - 1][j] + 1　(a[i - 1][j] <a[i][j])
  {f[i + 1][j] + 1　(a[i + 1][j] <a[i][j])
  {f[i][j - 1] + 1　(a[i][j - 1] <a[i][j])
  {f[i][j + 1] + 1　(a[i][j + 1] <a[i][j])

（初值）：f[i][j] = 1（至少经过自己一个点）

- 第三步：考虑是否需要优化
- 第四步：确定实现方法

例6：最大子串和

- 给你一个有正有负的序列，求一个子串（连续的一段），使其和最大！
- 样例输入： -5 6 -1 5 4 -7
- 样例输出： 14

例6：最大子串和

- 第一步：确定状态——原问题？子问题？
- f[i]前i个数的最大子串和——出现了和前一个题一样的问题：不知道f[i-1]中i-1选没选所以f[i]没法求
- 用f[i]表示一定要选第i个数的情况下前i个数的最大子串和（选第i个数和其左边连续的若干个）
- 第二步：确定状态转移方程
- f[i] =max(f[i - 1]+a[i], a[i])
- 第三步：考虑是否需要优化
- 第四步：确定实现方法

例7：最长公共子序列

- 给定两个序列X和Y，当另一序列Z既是X的子序列又是Y的子序列时，称Z是序列X和Y的公共子序列。
- 最长公共子序列:公共子序列中长度最长的子序列。
- 给定两个序列X={x1,x2,…,xm}和Y={y1,y2,…, yn}，找出X和Y的一个最长公共子序列。
- 样例输入：abcfbc
- abfcab
- 样例输出：4

例7：最长公共子序列

- 第一步：确定状态——原问题？子问题？
- f[i][j]表示前一个字符串的前i位与后一个字符串的前j位的最长公共子序列长度
- 第二步：确定状态转移方程
- 当x[i]==y[j]，f[i][j]=f[i-1][j-1]+1
- 当x[i]!=x[j]，f[i][j]=max(f[i - 1][j] ,f[i][j - 1])
- a[1]==b[1] f[1][1] = 1
- else f[1][1] = 0
- 第三步：考虑是否需要优化
- 第四步：确定实现方法

例子

X = (B, D, C, A, B, A)
Y = (A, B, C, B, D, A,B)

		0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

背包

例1：装箱问题——简化的01背包

- 有一个箱子容量为V（正整数，0<=V<=20000），同时有n个物品（0<n<=30），每个物品有一个体积（正整数）。
- 要求n个物品中，任取若干个装入箱内，使箱子的剩余空间为最小。
- 输入描述：一个整数v，表示箱子容量；一个整数n，表示有n个物品；接下来n个整数，分别表示这n 个物品的各自体积
- 输出描述：一个整数，表示箱子剩余空间。
- 样例输入：24 6
- 8 3 12 7 9 7
- 样例输出：0

例1：装箱问题

- 原问题：i件物品选若干件组成的小于V的最大体积是多少?
- 用可行性描述就可以了
- bool数组f[i][j]表示前i个物品能否放满体积为j的背包
- 枚举最后一次决策——第i个物品放还是不放!
- f[i][j] = f[i-1][j] || f[i-1][j-v[i]]
- 初值 f[i][j] = 0;

例1：装箱问题

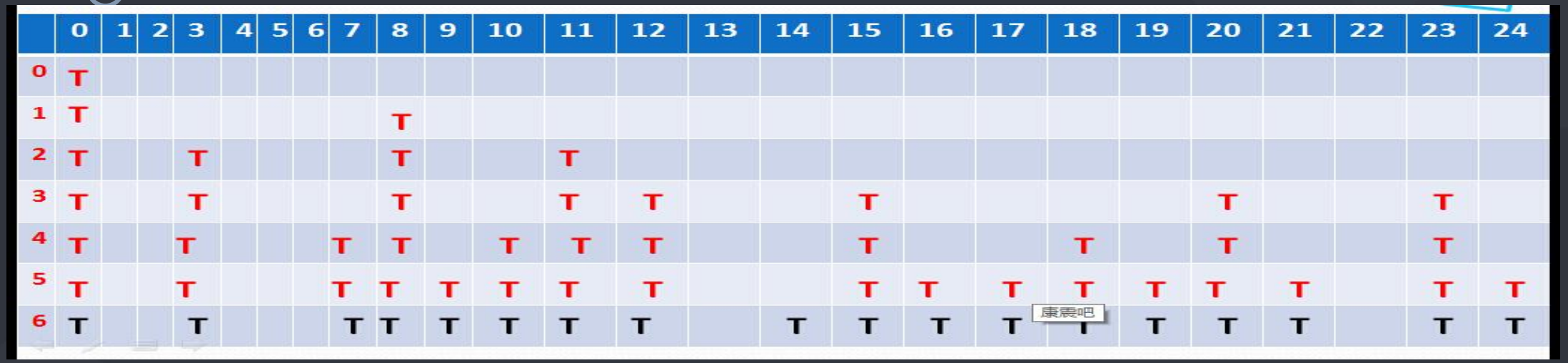
```
f[i][j] = f[i-1][j]
        || f[i-1][j-v[i]]
```

- 样例输入: 24 6

  8 3 12 7 9 7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	T																								
1	T								T																
2	T			T					T			T													
3	T			T					T			T	T			T					T			T	
4	T			T				T	T		T	T	T			T			T		T			T	
5	T			T				T	T	T	T	T	T			T	T	T	T	T	T	T		T	T
6	T			T				T	T	T	T	T	T		T	T	T	T	T	T	T	T		T	T

关于01滚动和就地滚动



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	T																								
1	T								T																
2	T			T					T			T													
3	T			T					T			T	T			T					T			T	
4	T			T				T	T		T	T	T			T			T		T			T	
5	T			T				T	T	T	T	T	T			T	T	T	T	T	T	T		T	T
6	T			T				T	T	T	T	T	T		T	T	T	T	T	T	T	T		T	T

康震吧

我们可以看到每一行的结果实际上只与上一行有关，所以就可以01滚动——f[i][0,1] 一行记录前一行的值，另一行记录当前行的值……

- 不过对于本题更加常用的方法是**就地滚动**！！
- 就地滚动嘛，顾名思义就是用一个一维数组了!之前的状态和当前的状态都记在同一个数组里了!
- 但是简单的变成一维以后有可能会有问题的——
- 如，我们把代码写成这样：

```
for(i=1 ; i<= n ; i++)
 for(j= c[j]; j<v ; j++)
     if(!f[j-c[i]) f[j] = f[j-c[i]] ;
```

- 假设第一个物品体积3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T			T			T			T			T			T			T			T			T

- 这样一个物品就可能被算多次……

- 怎么办？？改变内层循环顺序！
- for(i=1 ; i<= n ; i++)
  for(j=v ; j>=c[i] ; j--)
  　　if(!f[j-c[i]) f[j] = f[j-c[i]] ;
- 假设在判断若干个物品后f数组如下表，我们需要决策的物品体积是5
- 就地滚动相当于把前一阶段的状态和当前阶段的状态放在了同一行。为了确保区分它们，我们需要保证上一行的状态在j的左边，当前行的状态在j的右边。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T		T	T		T		T	T		T		T		T	T		T		T	T		T		

例2:01背包

- 有N件物品和一个容量为V的背包。第i件物品的费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使价值总和最大。
- f[i][j]表示**前i件物品恰放入一个容量为j的背包**可以获得的最大价值
- 两种情况：
- 1.不放当前物品 f[i][j] = f[i-1][j]
- 2.放当前物品 f[i][j] = f[i-1][j-c[i]]+w[i]
- f[i][j]=max{f[i-1][j],f[i-1][j-c[i]]+w[i]}

例2:01背包

- N = 6 V = 12
- 费用c[i]，价值w[i]

5	10
3	7
2	4
4	3
5	17
4	8

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0												
1	0					10							
2	0			7		10			17				
3	0		4	7		11		14	17		21		
4	0		4	7	3	11	10	14	17		21	21	
5	0		4	7	3	17	10	21	24	20	28	27	31
6	0		4	7	8	17	12	21	24	25	28	29	32

$$f[i][j]=\max\{f[i-1][j],f[i-1][j-c[i]]+w[i]\}$$

例2:01背包

- (就地滚动)
- for (i = 1 ; i <= n; i++)
- for (j=m; j>=c[i]; j--)
- 	if (f[j-c[i]] + w[i] > f[j])
- 		f[j] = f[j-c[i]] + w[i];
-

扩展2.1：

- N<= 20, V <= 10^9
- 背包体积太大怎么办?

扩展2.2：

- N<= 40, V <= 10^9

扩展2.3：

- N<= 100, V <= 10^9

例3：完全背包

- 有 N 种物品和一个容量为 V 的背包，每种物品都有无限件可用。放入第 i 种物品的费用是 Ci，价值是 Wi 。求解：将哪些物品装入背包，可使这些物品的耗费的费用总和不超过背包容量，且价值总和最大。
- 第一步：确定状态
- f[i, j] 依然表示前 i 种物品恰放入一个容量为 v的背包的最大权值。
- 第二步：确定状态转移方程
- f [i,j] = max（f [i -1,j-k*c[i]] + k * w[i])

  (0 <= k*c[i] <= j)

时间复杂度O(V*Σ(V/c[i]))，这样子的话
绝大部分题都是过不去的，该怎么办呢?

例3：完全背包

- 还记不记得之前讲01背包的就地滚动的有一段错误代码：
- for(i=1;<=n;i++)
  for(j=c[j];j<v;j++)
  if(!f[j-c[i]) f[j]=f[j-c[i]] ;
- 导致所有的物品都被算了多次……
- 这不正是完全背包所需要的么？
- （假设第一个物品体积为3，价值是5）

```
for ( i = 1 ; i <= n; i++ )
  {
     for (j=c[i]; j<=m; j++)
        if (f[j-c[i]] + w[i] > f[j])
           f[j] = f[j-c[i]] + w[i];

  }
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0			5			10			15			20			25			30			35			40

例4：多重背包

- 有N种物品和一个容量为V的背包。第i种物品最多有n[i]件可用，每件费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。
- 最朴素额状态转移方程和完全背包一样
- f[i][v]=max{f[i-1][v-k*c[i]]+k*w[i]|0<=k<=n[i]}
- 复杂度是O(V*Σn[i])。
- 比较大，需要优化……

例4：多重背包

- 转化为01背包求解：把第i种物品换成n[i]件01背包中的物品，则得到了物品数为Σn[i]的01背包问题
- 当然这样直接求解的复杂度仍然是O(V*Σn[i])。
- 我们考虑把第i种物品换成若干件物品，使得原问题中第i种物品可取的每种策略——取0..n[i]件——均能等价于取若干件代换以后的物品。另外，取超过n[i]件的策略必不能出现。
- 方法是：将第i种物品分成若干件物品，其中每件物品有一个系数，这件物品的费用和价值均是原来的费用和价值乘以这个系数。使这些系数分别为1,2,4,...,2^(k-1),n[i]-2^k+1，且k是满足n[i]-2^k+1>0的最大整数。例如，如果n[i]为13，就将这种物品分成系数分别为1,2,4,6的四件物品。

例4：多重背包

复杂度为O(V*Σlog n[i])
很圆满的解决了问题！

- 将n[i]拆成1,2,4,...,2^(k-1),n[i]-2^k+1，（k是满足n[i]-2^k+1>0的最大整数）道理何在?
- 1） 1+2+4+...+2^(k-1)+n[i]-2^k+1 = n[i] 这就保证了最多为n[i]个物品
- 2）1,2,4,.....,2^(k-1),可以凑出1到2^k – 1的所有整数（联系一个数的二进制拆分即可证明）
- 3） 2^k.....n[i]的所有整数可以用若干个上述元素凑出（可以理解为凑n[i]-t，而n[i]为上面所有数的和，t则是一个小于2^k 的数，那么在所有的数中去掉组成2^k 的那些数剩下的就可以组成n[i]-t了）

例5：二维费用的背包问题

- 二维费用的背包问题是指：对于每件物品，具有两种不同的费用；选择这件物品必须同时付出这两种代价；对于每种代价都有一个可付出的最大值（背包容量）。问怎样选择物品可以得到最大的价值。设这两种代价分别为代价1和代价2，第i件物品所需的两种代价分别为a[i]和b[i]。两种代价可付出的最大值（两种背包容量）分别为V和U。物品的价值为w[i]。

例5：二维费用的背包问题

- 设f[i][v][u]表示前i件物品付出两种代价分别为v和u时可获得的最大价值。状态转移方程就是：
- f[i][v][u]=max{f[i-1][v][u],f[i-1][v-a[i]][u-b[i]]+w[i]}

当发现由熟悉的动态规划题目
添加限制条件变形得来的题目时，
可以尝试在原来的状态中
加一维以满足新的限制条件。

例6：分组背包

- 有N件物品和一个容量为V的背包。第i件物品的费用是c[i]，价值是w[i]。这些物品被划分为若干组，每组中的物品互相冲突，最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。
- 本题和前面的题最大的不同是——每组物品有若干种策略：选择本组的某一件，或者一件都不选。

例6：分组背包

- f[k][v]表示前k组物品花费费用v能取得的最大权值
- f[k][v]=max{f[k-1][v],f[k-1][v-c[i]]+w[i]|物品i属于组k}
- 使用一维数组的伪代码如下：
- for 所有的组k

```
    for v=V..0
      for 所有的i属于组k
        f[v]=max{f[v],f[v-c[i]]+w[i]}
```

- “for v=V..0”这一层循环必须在“for 所有的i属于组k”之外。这样才能保证每一组内的物品最多只有一个会被添加到背包中。

区间dp

例1：括号匹配

- 题目大意：给出一个括号序列，求出其中匹配的括号数
- 例子：
- (((()))) 6
- ()()() 6
- ([]]) 4
-)[)(0
- ([][][) 6

例1：括号匹配

- f[i][j] 表示ai……aj的串中，有多少个已经匹配的括号
- 如果ai与ak是匹配的
- f[i][j] = max(f[i][j], f[i + 1][k - 1] + f[k + 1][j] + 2)
- （相当于是将i到j分成[xxxxx]xxxxx两部分）
- 否则f[i][j] = f[i + 1][j]
- （将第一个元素去掉——因为它肯定不能算）
- 边界 f[i][i] = 0

例1：括号匹配

- 推荐使用记忆化搜索
- 如果用递推的话，应该是区间大小由小到大递增作为最外层循环

```
for (int l = 2; l <= n; l++)                    //枚举区间长度
{
    for (int i =0; i + l - 1 < n; i++) //枚举区间起点
    {
        int j = i + l - 1;                      //计算区间终点
        ............
        ............
    }
```

例2:最长回文子序列长度

- 给你一个长度为n 的序列，求最长回文子序列长度

例2：最长回文子序列长度

- f[i][j] 表示ai……aj的串中，最长回文子序列长度
- 如果ai与aj是一样的
- f[i][j]= f[i+1][j-1]+2
- 否则：f[i][j] = max(f[i+1][j], f[i][j-1])

扩展2.1：最长回文子串长度

- 给你一个长度为n 的序列，求最长回文子串长度

例3：石子合并

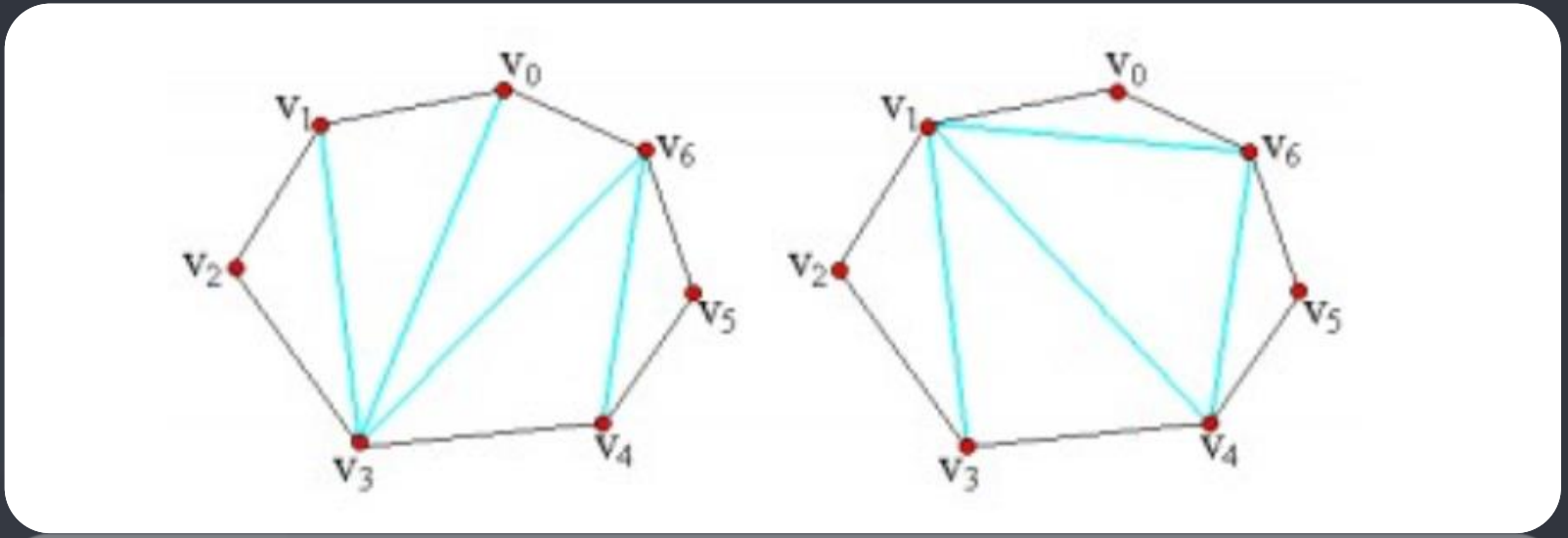
- 在一个园形操场的四周摆放N堆石子,现要将石子有次序地合并成一堆.规定每次只能选相邻的2堆合并成新的一堆，并将新的一堆的石子数，记为该次合并的得分。
- 试设计出1个算法,计算出将N堆石子合并成1堆的最小得分和最大得分.

例3：石子合并

- 先考虑没有环的情况：
- f[i,j]表示合并i到j的所有石子的得分
- f[i,j] = max(f[i][j]， f[i][k] + f[k + 1][j] + sum[i][j])
- sum[i][j] 表示i到j的石子的价值和！（也可以前缀和实现 sum[i]表示前i个石子的价值，那么我们需要的就是sum[j] – sum[i -1]）
- 但是现在有环！——可以通过取模的方法把它变成循环的！
- 也可以将序列加倍：变成‘12341234’，就可以完全用链的方法解决了！
- 边界：f[i][i] = 0

例4：凸多边形的三角拆分

- 给定一个具有 N(N <=50)个顶点(从 1 到 N 编号)的凸多边形，每个顶点的权均已知。问如何把 这个凸多边形划分成 N-2 个互不相交的三角形，使得这些三角形顶点的权的乘积之和最小。



- 设f[i][j](i <j)表示从顶点I 到顶点J 的凸多边形三角剖分后所得到的最大乘积，我们可以得到下面的动态转移方程：
  f[i][j]=Minf[i][k]+f[k][j]+s[i]*s[j]*s[k](i<k<j)
- 显然，目标状态为：f[1][n]

例5:田忌赛马

- 我国历史上有个著名的故事： 那是在2300年以前。齐国的大将军田忌喜欢赛马。他经常和齐王赛马。他和齐王都有三匹马：常规马，上级马，超级马。一共赛三局，每局的胜者可以从负者这里取得200银币。每匹马只能用一次。齐王的马好，同等级的马，齐王的总是比田忌的要好一点。于是每次和齐王赛马，田忌总会输600银币。
- 田忌很沮丧，直到他遇到了著名的军师——孙膑。田忌采用了孙膑的计策之后，三场比赛下来，轻松而优雅地赢了齐王200银币。这实在是个很简单的计策。由于齐王总是先出最好的马，再出次好的，所以田忌用常规马对齐王的超级马，用自己的超级马对齐王的上级马，用自己的上级马对齐王的常规马，以两胜一负的战绩赢得200银币。实在很简单。

例5:田忌赛马

- 如果不止三匹马怎么办？这个问题很显然可以转化成一个二分图最佳匹配的问题。把田忌的马放左边，把齐王的马放右边。田忌的马A和齐王的B之间，如果田忌的马胜，则连一条权为200的边；如果平局，则连一条权为0的边；如果输，则连一条权为 - 200的边……如果你不会求最佳匹配，用最小费用最大流也可以啊。
- 然而，赛马问题是一种特殊的二分图最佳匹配的问题，上面的算法过于先进了，简直是杀鸡用牛刀。现在，就请你设计一个简单的算法解决这个问题。

- 似乎可以贪心?
- 最强的马战平时，单一的贪心策略存在反例
- 例子1:
- 田忌： 1 2 3
- 齐王： 1 2 3

- 例子2:
- 田忌: 2 3
- 齐王: 1 3

例5:田忌赛马

- 总结起来就是：田忌能赢要赢得最少，如果一定输就要输的最惨，如果能打平就出现打平和输两种情况。
- 所以：
- **田忌出马不是出最强的，就是出最弱的**
- 先将田忌和齐王的马按从大到小顺序排序且默认齐王按从大到小出马（齐王出马的顺序不影响田忌的策略和得分）

例5:田忌赛马

- F[i][j]表示田忌区间[i，j]的马比下去的最优得分
- f[i][j]:=max(f[i+1][j]+cost(i,k),f[i][j-1]+cost(j,k));
- k表示齐王当前出的马,cost(i,k)是田忌第i匹马与齐王第k匹马相比的结果。
- （本题还有若干其他的状态转移方程，大家可以自己思考）

例5:田忌赛马

- 然后要考虑边界，边界是什么？边界一定是i=j时的值，f[i][i]又等于什么呢?
- 由于齐王是从大到小派出马的，当区间为【1，n】的时候齐王出的马是第一匹，随着区间长度的减少，齐王的出的马变成了后面的，于是——f[i][i] 是齐王出最弱那匹马的时候的与田忌第i匹马相比的收益。