动态规划 A

记忆化搜索 & 动态规划初步

杨乐

blog.csdn.net/yangle61

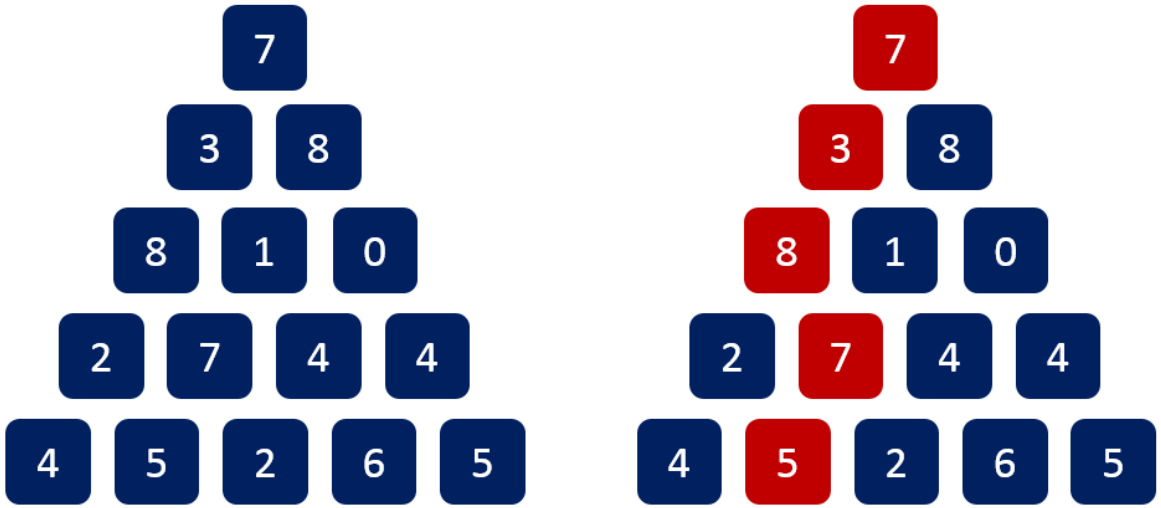
@897982078

yangle61@163.com

2017 年 7 月 2 日

问题引入 - 数字金字塔

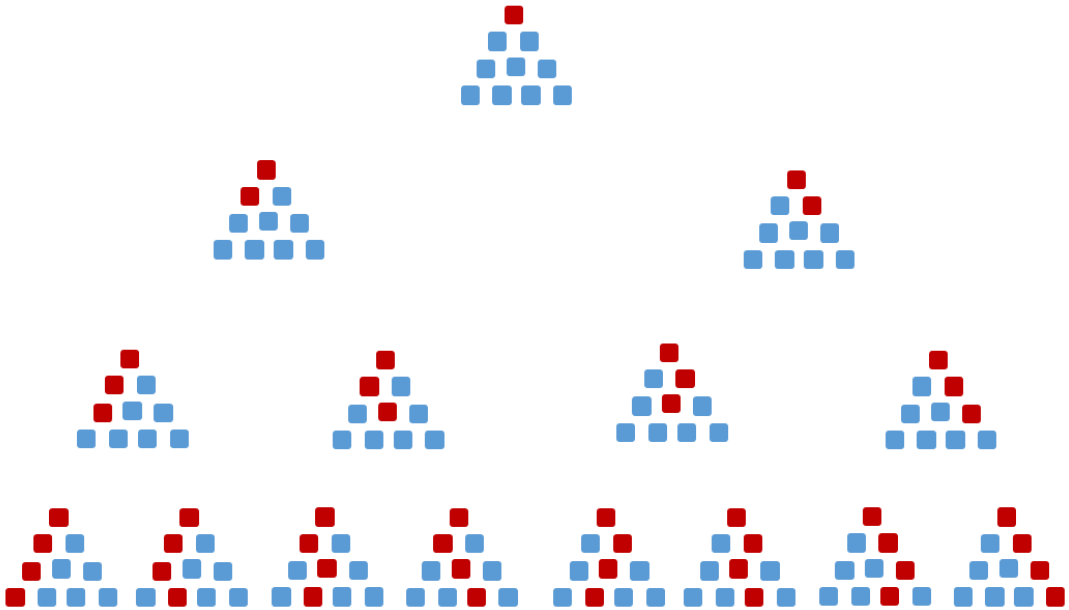
* Luogu 1216
* 观察下面的数字金字塔。
* 写一个程序来查找从最高点到底部任意处结束的路径，使路径经过数字的和最大。每一步可以走到左下方的点也可以到达右下方的点。



图：在上面的样例中，从 7-3-8-7-5 的路径产生了最大值

数字金字塔 - 搜索

* 回忆**搜索**的实现。
- 状态：目前在第 X 行第 Y 列；
- 行动：向左走、向右走；



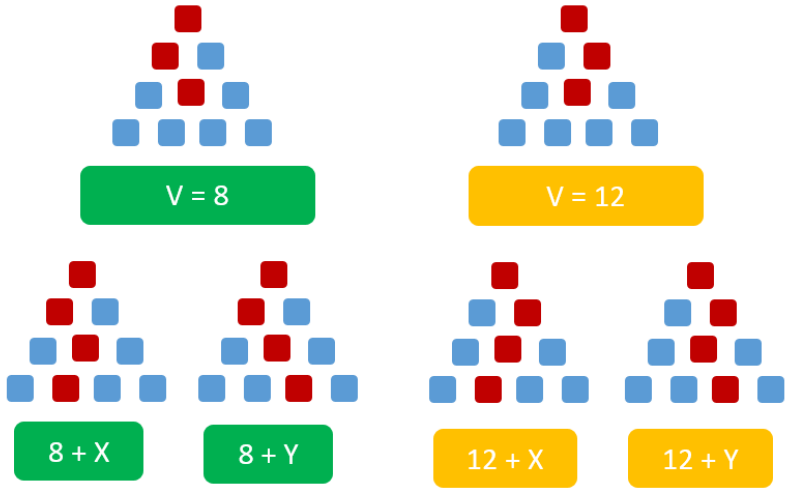
数字金字塔 - 搜索

```
void dfs(int x, int y, int val)
{
    val += a[x][y];

    if(x == n-1)
    {
        if(val > ans) ans = val;
        return;
    }

    dfs(x+1, y, val);
    dfs(x+1, y+1, val);
}
```

数字金字塔 - 冗余搜索

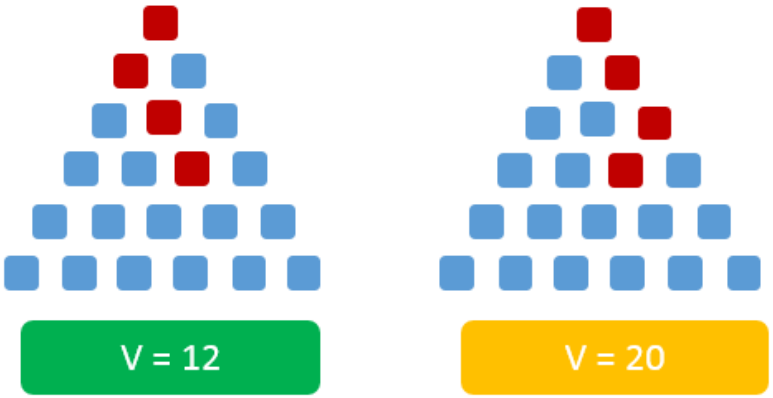
* **冗余搜索**：无用的、不会改变答案的搜索。

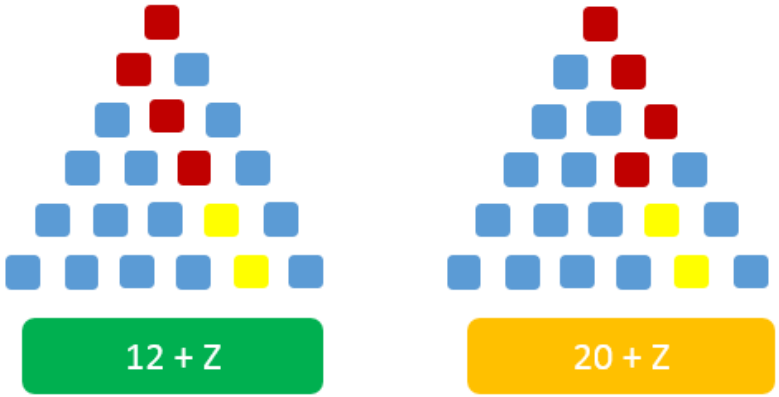
- 例子：观察下面两个例子。用两种方式都能到达第 3 行第 2 列，其中一个总和为 8，一个总和 12。
- 那么可以观察可得，总和为 8 的搜索是**冗余**的(不会改变答案)，即使不继续搜索，**答案也不会改变**。
- 因为 12 往下搜索，无论往左往右，都会比 8 对应的路径大。



数字金字塔 - 冗余搜索

* **冗余搜索**：无用的、不会改变答案的搜索。





数字金字塔 - 冗余搜索

* 如何利用**冗余搜索**，来优化程序？
  - 对于每一个位置记录一个值 F，代表**搜索到此位置时，最大的路径和是多少**。
  - 如果搜索到某个位置，路径和不大于**记录值 F**，则说明这个是冗余搜索，可以直接退出；
  - 否则仍需要继续搜索；
* **记忆化搜索**：我们把这种搜索方式称作**记忆化搜索**。我们会根据之前的**“记忆”**来优化搜索；而在这道题中，每个位置的**“记忆”**是最大的路径和。

数字金字塔 - 记忆化搜索

```
// T1 : 数字金字塔（记忆化搜索）
void dfs(int x, int y, int val) {
    val += a[x][y];

    // 记忆化过程
    if(val <= f[x][y]) return;
    f[x][y] = val;

    if(x == n-1) {
        if(val > ans) ans = val;
        return;
    }

    dfs(x+1, y, val);
    dfs(x+1, y+1, val);
}
```

问题引入 - 背包问题

* Luogu 1048 ： 采药
* 辰辰是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。”

- 总时间：70
- 草药 1 ：时间 71；价值 100
- 草药 2 ：时间 69；价值 1
- 草药 3 ：时间 1 ；价值 2

@ 最优选择：草药 2+3：总时间 70；总价值 3

背包问题 - 搜索

* 回忆**搜索**的实现。

- 状态：目前已经决定到第 x 件物品，当前背包中物品总重量为 w，总价值为 v；
- 行动：这件物品取不取；
- 约束：物品的总重量不超过 W(背包总重量)；
- 目标：物品总价值最大；

背包问题 - 冗余搜索

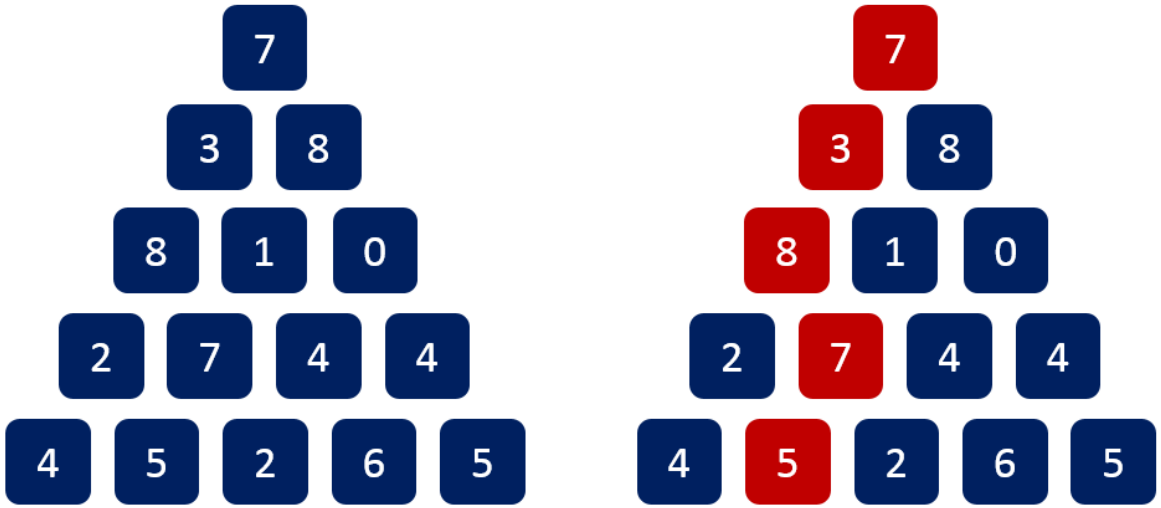
* **冗余搜索**：无用的、不会改变答案的搜索。
- 比较下列两种情况：
- 状态相同：$x_1 = x_2$（当前搜索到同一件物品），$w_1 = w_2$（当前总重量相等）；
- 价值不同：但它们的背包总价值不同，其中 $v_1 < v_2$。
- 则我们可以说**状态 1**为**冗余**的，因为它肯定比**状态 2**要差。
* **记忆化**：对于每个状态（x, w），记录对应的 v 的最大值。

背包问题 - 记忆化搜索

```
// T5 : 采药（记忆化搜索）
void dfs(int t, int x, int val) // t 为剩余时间，val 为总价值
{
    // 记忆化
    if(val <= f[t][x]) return;
    f[t][x] = val;

    if(x == n)
    {
        if(val > ans) ans = val;
        return;
    }

    dfs(t, x+1, val);
    if(t >= w[x]) dfs(t - w[x], x+1, val + v[x]);
}
```

问题引入 - 数字金字塔

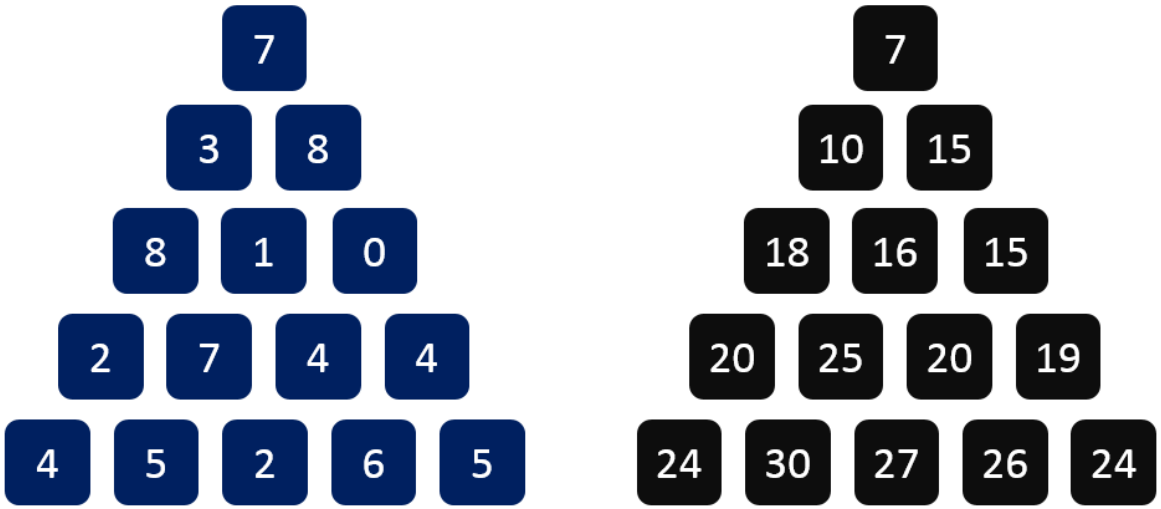
* Luogu 1216
* 观察下面的数字金字塔。
* 写一个程序来查找从最高点到底部任意处结束的路径，使路径经过数字的和最大。每一步可以走到左下方的点也可以到达右下方的点。



图：在上面的样例中，从 7-3-8-7-5 的路径产生了最大值

数字金字塔 - 记忆化搜索

* 回想**记忆化搜索**的实现步骤。
  - 我们记录了到达每个位置的路径最大值。
  - **考虑**：它们之间是否存在联系？

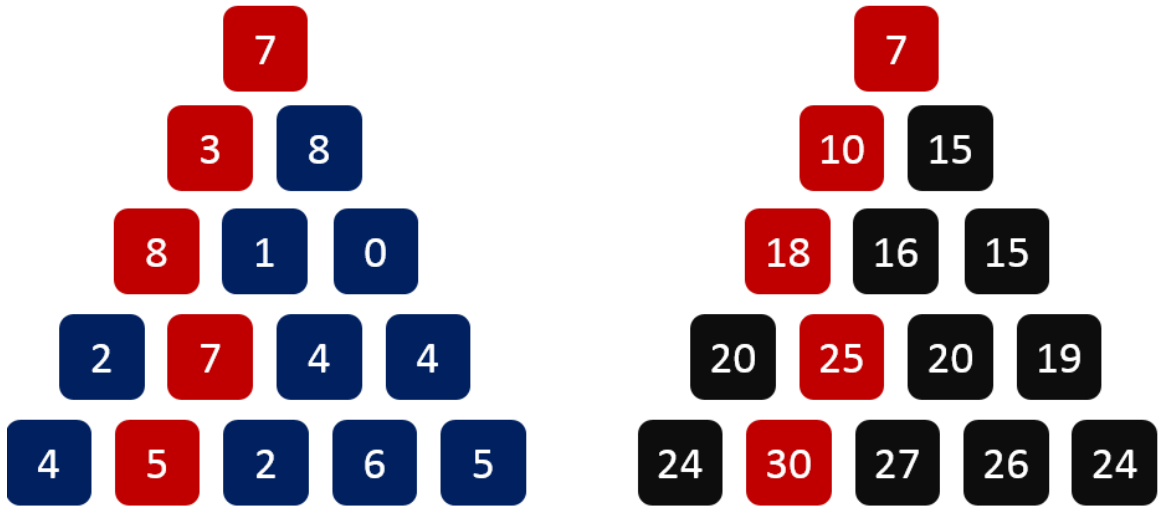


图：右侧三角形：每个位置的路径和最大值

数字金字塔 - 最优性

* **最优性**：设走到某一个位置的时候，它达到了路径最大值，那么在这之前，它走的每一步都是最大值。

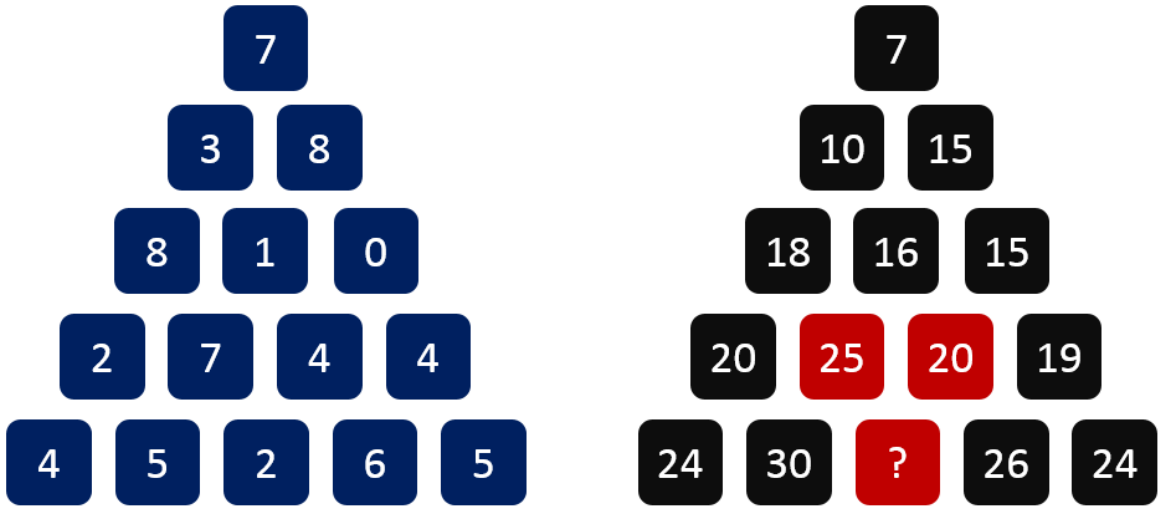
- 考虑这条最优的路径：每一步均达到了最大值



图：所示路径为最优路径，与最优值一一对应

数字金字塔 - 最优性

* **最优性的好处**：要达到一个位置的最优值，它的**前一步**也一定是最优的。
  - 考虑图中位置，如果它要到达最优值，有两个选择，从左上方或者右上方的最优值得到



图：两种可能性

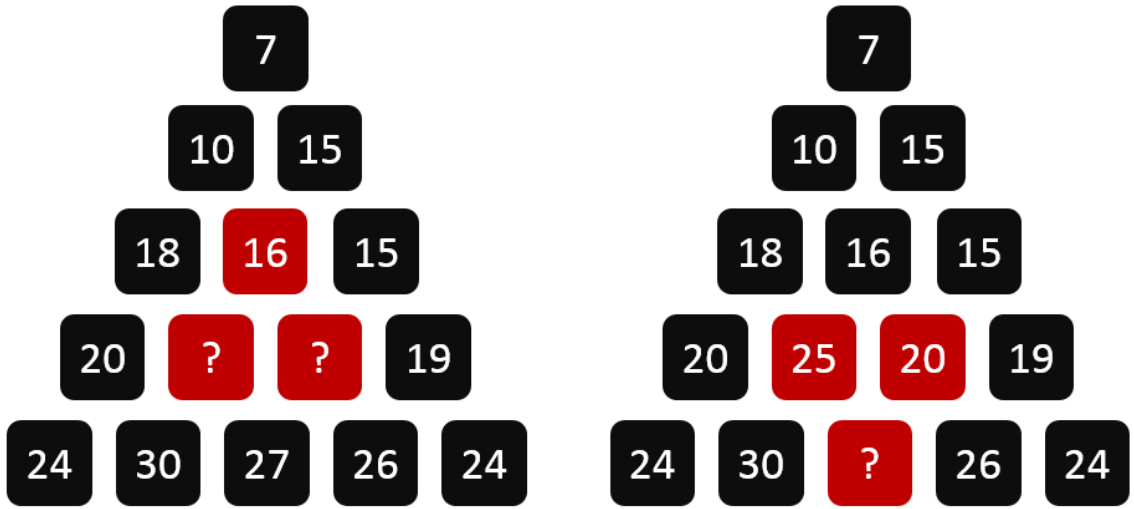
数字金字塔 - 动态规划

* **动态规划（DP）**：只记录状态的最优值，并用**最优值**来推导出其他的**最优值**。

- 记录 `F[i][j]` 为第 `i` 行第 `j` 列的路径最大值。有两种方法可以推导：

@ **顺推**：用 `F[i][j]` 来计算 `F[i+1][j],F[i+1][j+1]`；

@ **逆推**：用 `F[i-1][j],F[i-1][j-1]` 来计算 `F[i][j]`。



图：左图：顺推；右图：逆推

数字金字塔 - 顺推

```
// T2 : 数字金字塔 - 顺推

f[0][0] = a[0][0];
for(int i = 0; i < n - 1; ++ i)
    for(int j = 0; j <= i; ++ j)
    { // 分别用最优值来更新左下方和右下方
        f[i+1][j] = max(f[i+1][j], f[i][j] + a[i+1][j]);
        f[i+1][j+1] = max(f[i+1][j+1], f[i][j] + a[i+1][j+1]);
    }
```

数字金字塔 - 逆推

```
// T4 : 数字金字塔 - 逆推

f[0][0] = a[0][0];
for(int i = 0; i < n; ++ i)
{
    f[i][0] = f[i-1][0] + a[i][0]; // 最左的位置没有左上方
    f[i][i] = f[i-1][i-1] + a[i][i]; // 最右的位置没有右上方

    for(int j = 1; j < i; ++ j) // 在左上方和右上方取较大的
        f[i][j] = max(f[i-1][j-1], f[i-1][j]) + a[i][j];
}

// 答案可能是最第一行的任意一列
ans = 0;
for(int i = 0; i < n; ++ i)
    ans = max(ans, f[n-1][i]);
```

数字金字塔 - 转移方程

* **转移方程**：最优值之间的**推导公式**。

@ **顺推**：

```
F[i+1][j] = MAX (F[i][j] + a[i+1][j]);
F[i+1][j+1] = MAX (F[i][j] + a[i+1][j+1]);
```

@ **逆推**：

```
F[i][j] = MAX (F[i-1][j], F[i-1][j-1]) + a[i][j];
```

(注意！逆推时要注意边界情况！)

* **顺推**和**逆推**本质上是一样的；顺推和搜索的顺序类似；而逆推则是将顺序反过来；顺推考虑的是“我这个状态的下一步去哪里”，逆推的考虑的是“从什么状态可以到达我这里”。

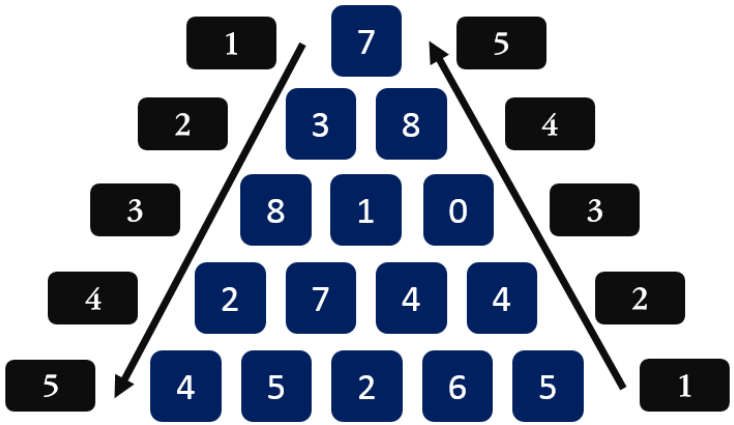
* 转移方程中需要时刻注意边界情况。

数字金字塔 - 逆推/ 改变搜索顺序

```
// T3 : 数字金字塔 - 逆推/ 路径自底向上

// 改变顺序: 记录从底部向上走的路径最优值
for(int i = 0; i < n; ++ i)
    f[n-1][i] = a[n-1][i];

// 逆推过程: 可以从左下方或右下方走过来; 没有边界情况
for(int i = n-2; i >= 0; -- i)
    for(int j = 0; j <= i; ++ j)
        f[i][j] = max(f[i+1][j+1], f[i+1][j]) + a[i][j];

// 答案则是顶端
ans = f[0][0];
```

数字金字塔 - 转移顺序

* **转移顺序**：最优值之间的**推导顺序**。
- 在**数字金字塔**中，为什么能够使用**动态规划**？
- 因为有明确的顺序：自上而下。
- 也就是说，能划分成不同的阶段，这个阶段是**逐步进行**的。
- 这和**搜索顺序**也是类似的。
* 所以，只要划分好阶段，从前往后推，与从后往前推都是可以的。



问题引入 - 背包问题

* Luogu 1048 ： 采药
* 辰辰是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。”

背包问题 - 记忆化搜索

回忆记忆化搜索的方法：

```
// T5 : 采药（记忆化搜索）
void dfs(int t, int x, int val) // t 为剩余时间，val 为总价值
{
    // 记忆化
    if(val <= f[t][x]) return;
    f[t][x] = val;

    if(x == n) {
        if(val > ans) ans = val;
        return;
    }

    dfs(t, x+1, val);
    if(t >= w[x]) dfs(t - w[x], x+1, val + v[x]);
}
```

背包问题 - 动态规划

* **动态规划（DP）**：只记录状态的最优值，并用**最优值**来推导出其他的**最优值**。

- **状态设计**：记录 `F[i][j]` 为，已经决定前 i 件物品的情况，在总重量为 j 的情况下，物品总价值的最大值。同样也是有两种方法可以推导：

@ **顺推**：“我这个状态的下一步去哪里”

@ **逆推**：“从什么状态可以到达我这里”

背包问题 - 状态转移方程

* **转移方程**：状态之间的**推导公式**

- **当前状态**：`F[i][j]` 为，已经决定前 i 件物品的情况，在总重量为 j 的情况下，物品总价值的最大值。
- **顺推**：“我这个状态的下一步去哪里”：我现在要决定下一件物品取还是不取。
  - 如果不取的话，可以达到状态 `F[i+1][j]`；
  - 如果取的话，可以达到状态 `F[i+1][j+w[i+1]]`(需要满足重量约束)；

@ **逆推**：“从什么状态可以到达我这里”：考虑我这件物品取不取。

- 如果是不取的，那可以从 `F[i-1][j]` 推导而来；
- 如果是取的，可以从 `F[i-1][j-w[i]]` 推导而来的(同样需要满足重量约束)。

背包问题 - 顺推

```
// T6 : 采药（DP/顺推）
    for(int i = 0; i < n; ++ i)
        for(int j = 0; j <= t; ++ j)
        {
            // 不取
            f[i+1][j] = max(f[i+1][j], f[i][j]);

            // 取
            if(j + w[i] <= t)
            f[i+1][j+w[i]] = max(f[i+1][j+w[i]], f[i][j]+v[i]);
        }

    // 答案
    ans = 0;
    for(int i = 0; i <= t; ++ i) ans = max(ans, f[n][i]);
```

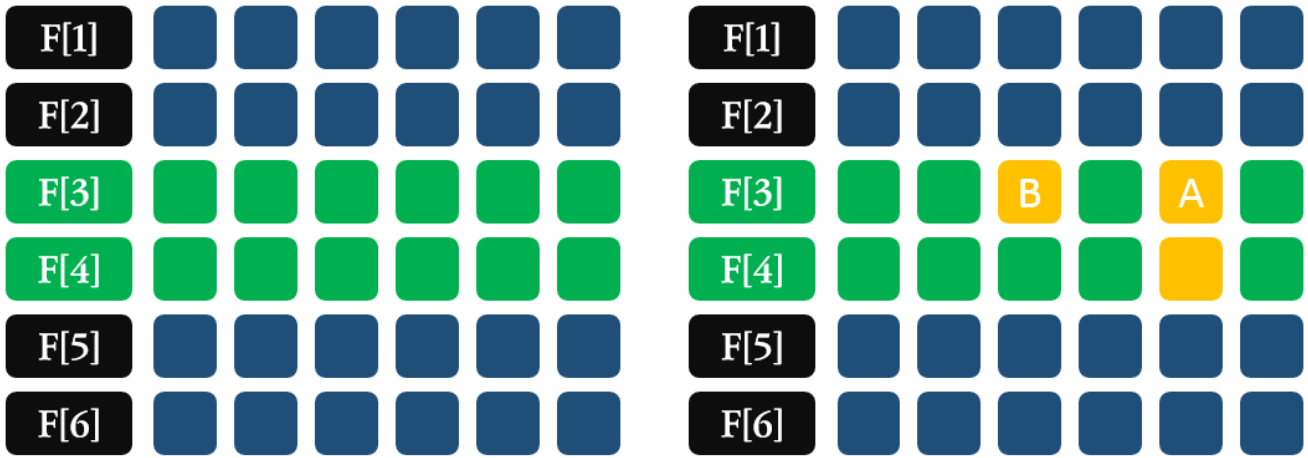
背包问题 - 逆推

```
// T7 : 采药（DP/逆推）
    for(int i = 1; i <= n; ++ i)
        for(int j = 0; j <= t; ++ j)
        {
            // 不取
            f[i][j] = f[i-1][j];

            // 取
            if(j >= w[i]) f[i][j] = max(f[i][j], f[i-1][j-w[i]] + v[i]);
        }

    // 答案
    ans = 0;
    for(int i = 0; i <= t; ++ i) ans = max(ans, f[n][i]);
```

背包问题 - 数组压缩

* **数组压缩**：用一个一维数组来代替二维数组

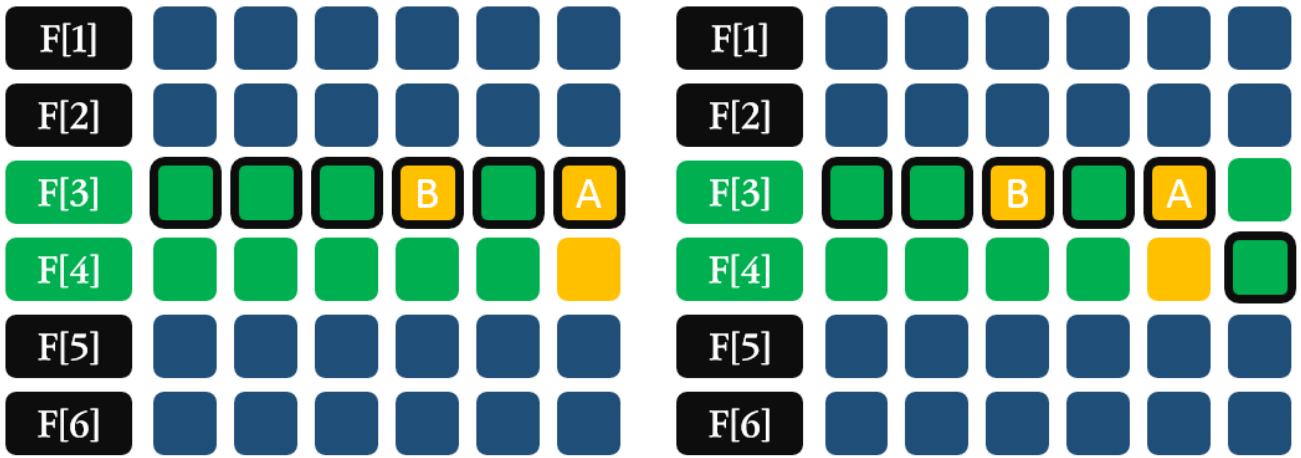
- 观察**状态转移方程**，可以发现 `F[i]` 只由 `F[i-1]` 决定。也就是说，前面的很多状态对后面都是没有影响的。所以我们可以只保留前一行的状态。



图：例如，*F[4]* 中的状态只和 *F[3]* 中的两个状态 *A*、*B* 相关

背包问题 - 数组压缩

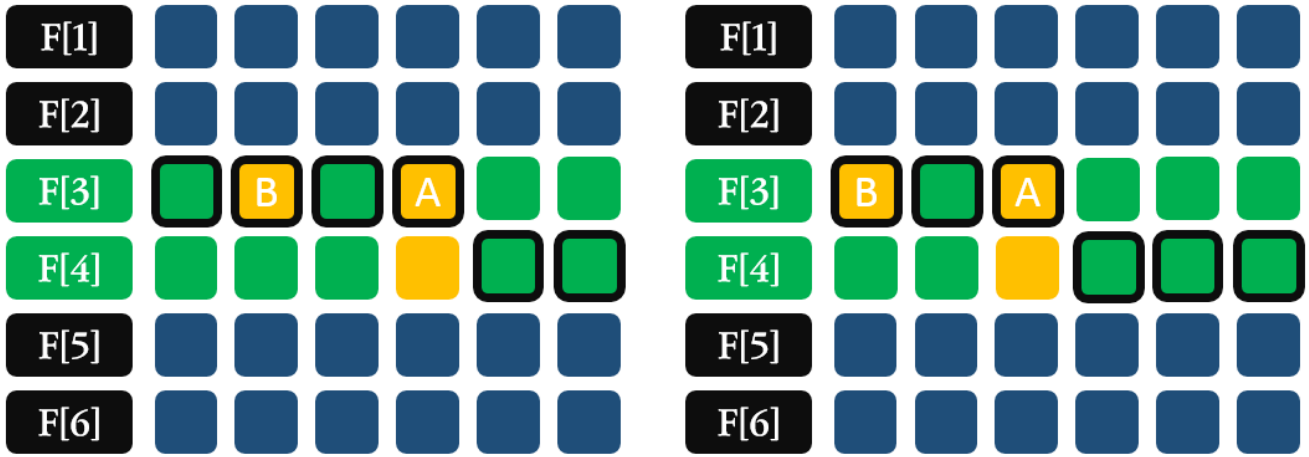
* 所以一个直观的做法是，记录两个数组，分别记录 `F[i-1]` 与 `F[i]` 的值。
* 但更进一步，我们可以甚至不记录两行，只记录一行的状态。
- 我们倒着枚举，在 `F[i-1]` 数组中逐步更新，让它逐步变为 `F[i]`。



图：粗框中记录的是 *F* 数组对应的值

背包问题 - 数组压缩

- 因为是倒着枚举的，先枚举的位置都已经无用了，可以直接用 `F[i]` 的元素来替换。



图：粗框中记录的是 F 数组对应的值

背包问题 - 逆推/ 数组压缩

```
// T10 : 采药（DP/逆推/数组压缩）

    // 用一个一维数组来代替二维数组
    for(int i = 1; i <= n; ++ i)
        for(int j = t; j >= 0; -- j) // 重量：倒着枚举
        {
            // 不取：对数组没有影响
            // f[i][j] = f[i-1][j];

            // 取
            //if(j >= w[i]) f[i][j] = max(f[i][j], f[i-1][j-w[i]] + v[i]);
            if(j >= w[i]) f[j] = max(f[j], f[j - w[i]] + v[i]);
        }
    // 在枚举过程中，大于 j 的位置等于 f[i][j]，小于 j 的位置等于 f[i-1][j]
```

问题引入 - 完全背包问题

* POJ 1384 : **Piggy-Bank**
* 现在有 *n* 种硬币，每种硬币有特定的重量 *cost[i]* 和它对应的价值 *val[i]*. 每种硬币可以无限使用。已知现在一个储蓄罐中所有硬币的总重量正好为 *m* 克，问你这个储蓄罐中最少有多少价值的硬币？如果不可能存在 *m* 克的情况，那么就输出“*This is impossible.*”
* 我们也把这类问题归入到背包问题当中：有物品，重量限制，价值最大。
* 但与**采药问题**不同的是，每件物品可以无限使用。

完全背包问题 - 状态转移方程 - 顺推

* **转移方程**：状态之间的**推导公式**

- **当前状态**：`F[i][j]` 为，已经决定前 `i` 件物品的情况，在总重量为 `j` 的情况下，物品总价值的最大值。

@ **顺推**：“我这个状态的下一步去哪里”：考虑我这件物品**取多少件**。

> 如果是不取的，那可以推导到 `F[i+1][j]`;
> 如果是取一件，那可以推导到 `F[i+1][j+w[i]]`;
> 如果是取 `k` 件，那可以推导到 `F[i+1][j+w[i]*k]`.

背包问题 - 顺推

```
// T8 : Piggy-Bank（DP/顺推）

// 初值处理：由于问题求的是最小值，所以先把所有状态赋值为最大值
for(int i = 1; i <= n + 1; ++ i)
    for(int j = 0; j <= m; ++ j) f[i][j] = INF;

// 第一件还没取，重量为 0
f[1][0] = 0;

for(int i = 1; i <= n; ++ i) // i：已经决定的物品
    for(int j = 0; j <= m; ++ j) // j：总重量
        for(int k = 0; j + w[i] * k <= m; ++ k) // k：这件物品取多少件
            f[ i+1 ][ j+w[i]*k ] =
            min( f[ i+1 ][ j+w[i]*k ], f[i][j] + p[i]*k );
            // w 重量；p 价值
```

完全背包问题 - 状态转移方程 - 逆推

* **转移方程**：状态之间的**推导公式**

- **当前状态**：`F[i][j]` 为，已经决定前 i 件物品的情况，在总重量为 j 的情况下，物品总价值的最大值。

@ **逆推**：“从什么状态可以到达我这里”：考虑我这件物品**取多少件**。

  > 如果是不取的，那可以从 `F[i-1][j]` 处推导得到；
  > 如果是取一件，那可以从 `F[i-1][j-w[i]]` 处推导得到；
  > 如果是取 `k` 件，那可以从 `F[i-1][j-w[i]*k]` 处推导得到；

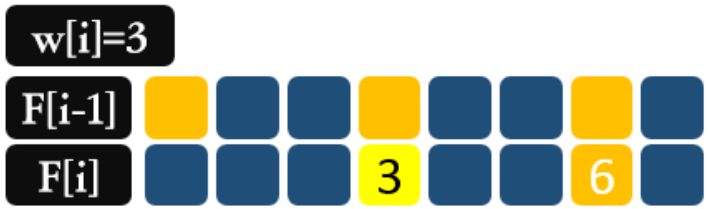
背包问题 - 逆推

```
// T9 / work1 : Piggy-Bank（DP/逆推）

  for(int i = 0; i <= n; ++ i)
     for(int j = 0; j <= m; ++ j) g[i][j] = INF;
  g[0][0] = 0;

  for(int i = 1; i <= n; ++ i)
     for(int j = 0; j <= m; ++ j)
        for(int k = 0; j >= w[i] * k; ++ k)
           g[i][j] = min( g[i][j], g[i-1][j-w[i]*k] + p[i]*k );
```

完全背包问题 - 状态转移方程 - 逆推优化

* **逆推**：观察和逆推相关的状态。

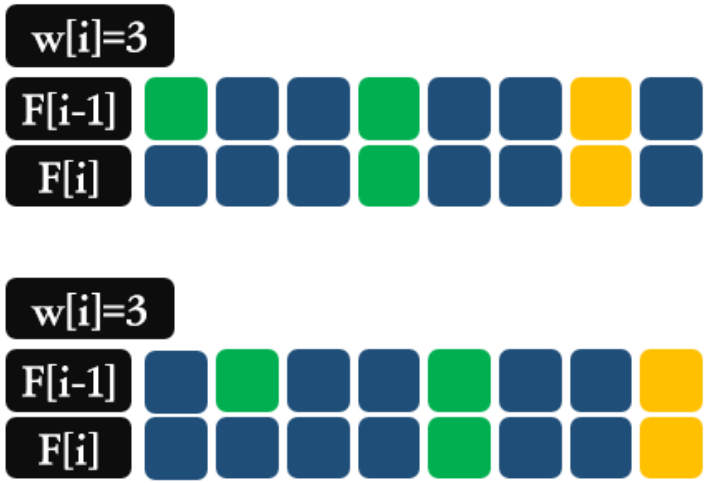
  假设 *w[i]=3*，则 *F[i][6]* 与 *F[i-1][0,3,6]* 相关；*F[i][7]* 与 *F[i-1][1,4,7]* 相关；与此同时，*F[i][3]* 与 *F[i-1][0,3]* 相关；*F[i][4]* 与 *F[i-1][1,4]* 相关。



* 则可以得到，实际上与 *F[i][j]* 相关的状态只比 *F[i][j-w[i]]* 多一个。

完全背包问题 - 状态转移方程 - 逆推优化

* 则所以我们可以这样推导：
  - 如果是不取的，那可以从 `F[i-1][j]` 处推导得到；
  - 如果是取一件或更多，那可以从 `F[i][j-w[i]]` 处推导得到；*(因为是可以取任意件，所以从 F[i] 中取最优而不是从 F[i-1] 中取)*



* 而从这种逆推也可以方便的写出数组压缩。

背包问题 - 逆推优化

```
// T9 / work2 : Piggy-Bank（DP/逆推优化）

  for(int i = 0; i <= n; ++ i)
     for(int j = 0; j <= m; ++ j) g[i][j] = INF;
  g[0][0] = 0;

  for(int i = 1; i <= n; ++ i)
     for(int j = 0; j <= m; ++ j)
     {
        g[i][j] = g[i-1][j];
        if(j >= w[i]) g[i][j] = min(g[i][j], g[i][j-w[i]] + p[i]);
     }
```

背包问题 - 逆推优化/数组压缩

```
// T9 / work3 : Piggy-Bank（DP/逆推优化/数组压缩）

  for(int j = 0; j <= m; ++ j) f[j] = INF;

  f[0] = 0;

  for(int i = 1; i <= n; ++ i)
     for(int j = 0; j <= m; ++ j)
           if(j >= w[i]) f[j] = min( f[j], f[j-w[i]] + p[i] );
```

问题引入 - 背包计数问题

* Luogu 1466 : 集合
* 对于从 *1* 到 *N (1 <= N <= 39)* 的连续整数集合，能划分成两个子集合，且保证每个集合的数字和是相等的。举个例子，如果 *N=3*，对于 *[1，2，3]* 能划分成两个子集合，每个子集合的所有数字和是相等的：
  - [3] 和 [1,2]
* 这是唯一一种分法（交换集合位置被认为是同一种划分方案，因此不会增加划分方案总数）如果 *N=7*，有四种方法能划分集合 *[1，2，3，4，5，6，7]*，每一种分法的子集合各数字和是相等的：
  - [1,6,7] 和 [2,3,4,5] *(注： 1+6+7=2+3+4+5)*
  - [2,5,7] 和 [1,3,4,6]
  - [3,4,7] 和 [1,2,5,6]
  - [1,2,4,7] 和 [3,5,6]
* 给出 *N*，你的程序应该输出划分方案总数，如果不存在这样的划分方案，则输出 *0*。程序不能预存结果直接输出（不能打表）。

问题引入 - 背包模型

* **物品**：可以把所有的数字看作物品。对于数字 `i`，其对应的重量为 `i`，则我们需要求出**装满载重为 `M` 的背包的方案数**(*其中 `M` 为所有数总和的一半*)。
* **状态**：（仿照之前的方法）设 `F[i][j]` 为已经考虑完数字 `1-i` 了，当前数字总和为 `j` 的总方案数。
* **状态转移方程 - 顺推**：考虑有没有取数字 `i`。

  没取：`F[i+1][j] += F[i][j]`

  取了：`F[i+1][j+i] += F[i][j] (j+i<=M)`
* **状态转移方程 - 逆推**：考虑有没有取数字 `i`。

  `F[i][j] = F[i-1][j] + F[i-1][j-i] (j>=i)`

背包问题 - 顺推

```
// T14 / work1 : 集合（DP/顺推）

  // 初值:（什么都不取）和 =0, 有一种方案
  f[1][0] = 1;

  for(int i = 1; i <= n; ++ i)
     for(int j = 0; j <= m; ++ j)
     {
        f[i+1][j] += f[i][j];
        if(i + j <= m) f[i+1][i+j] += f[i][j];
     }
```

背包问题 - 逆推

```
// T14 / work2 : 集合（DP/逆推）

  f[0][0] = 1;

  for(int i = 1; i <= n; ++ i)
     for(int j = 0; j <= m; ++ j)
     {
        f[i][j] = f[i-1][j];
        if(j >= i) f[i][j] += f[i-1][j-i];
     }
// T14 / work3 : 集合（DP/逆推/数组压缩）

  g[0] = 1;
  for(int i = 1; i <= n; ++ i)
     for(int j = m; j >= i; -- j) // 注意要倒着枚举
        g[j] += g[j-i];
```

问题引入 - 完全背包计数问题

* Luogu 1474 ： 货币系统
* 母牛们不但创建了它们自己的政府而且选择了建立了自己的货币系统。由于它们特殊的思考方式，它们对货币的数值感到好奇。
* 传统地，一个货币系统是由 *1,5,10,20* 或 *25,50,* 和 *100* 的单位面值组成的。
* 母牛想知道有多少种不同的方法来用货币系统中的货币来构造一个确定的数值。
* 举例来说，使用一个货币系统 *[1,2,5,10,...]* 产生 *18* 单位面值的一些可能的方法是:*18x1*，*9x2*，*8x2+2x1*，*3x5+2+1*，等等其它。写一个程序来计算有多少种方法用给定的货币系统来构造一定数量的面值。

问题引入 - 背包模型

* **物品**：可以把所有的货币看作物品。对于每种货币，其对应的重量为它的面值(注意到与前一道题目相比，每个物品是可以取任意件的)。
* **状态**：（仿照之前的方法）设 `F[i][j]` 为已经考虑完前 `i` 种货币了，当前钱的总和为 `j` 的总方案数。
* **状态转移方程 - 逆推**：考虑货币 `i` 取了多少件。

$$F[i][j] = \sum F[i-1][j - w[i]*k]$$

背包问题 - 逆推

```
// T15 / work1 : 货币系统（DP/逆推）

  f[0][0] = 1;
  for(int i = 1; i <= v; ++ i)
     for(int j = 0; j <= n; ++ j)
        for(int k = 0; k <= j / a[i]; ++ k)
           f[i][j] += f[i-1][j - a[i]*k];

// T15 / work2 : 货币系统（DP/逆推/数组压缩）

  g[0] = 1;
  for(int i = 1; i <= v; ++ i)
     for(int j = a[i]; j <= n; ++ j)
        g[j] += g[j - a[i]];
```

动态规划 - 总结

* 解决**动态规划**问题，需要注意以下几点：
- **顺序/状态**：与搜索类似，依然需要考虑问题的**先后顺序**，以及状态的表示
- **转移方程**：有**顺推**与**逆推**两种；需要注意初值、边界情况。
* 同时在实现**背包问题**中，可以使用数组压缩来优化空间。