

# 搜索算法精讲

教师：聂宏轩



## 基础算法

模拟  
枚举  
递推  
贪心  
归并  
分治

搜索  
动态规划

# 1

# 八数码问题

## 洛谷 ----题目描述

在 $3 \times 3$ 的棋盘上，摆有八个棋子，每个棋子上标有1至8的某一数字。棋盘中留有一个空格，空格用0来表示。空格周围的棋子可以移到空格中。

要求解的问题是：给出一种初始布局（初始状态）和目标布局（为了使题目简单，设目标状态为123804765），找到一种最少步骤的移动方法，实现从初始布局到目标布局的转变。

## 输入格式

输入初始状态，一行九个数字，空格用0表示

## 输出格式

只有一行，该行只有一个数字，表示从初始状态到目标状态需要的最少移动次数(测试数据中无特殊无法到达目标状态数据)

## 输入输出样例

### 输入

283164705

### 输出

5

2	8	3
1	6	4
7		5

1	2	3
8		4
7	6	5

八数码问题的初始和目标状态

# 1

# 八数码问题

- 首先，请大家在草稿纸上画出你的思考过程（八数码图的变化过程）
- 例如：

1	2	3
8		4
7	6	5



1	2	3
8	4	
7	6	5

2	8	3
1	6	4
7		5

1	2	3
8		4
7	6	5

八数码问题的初始和目标状态

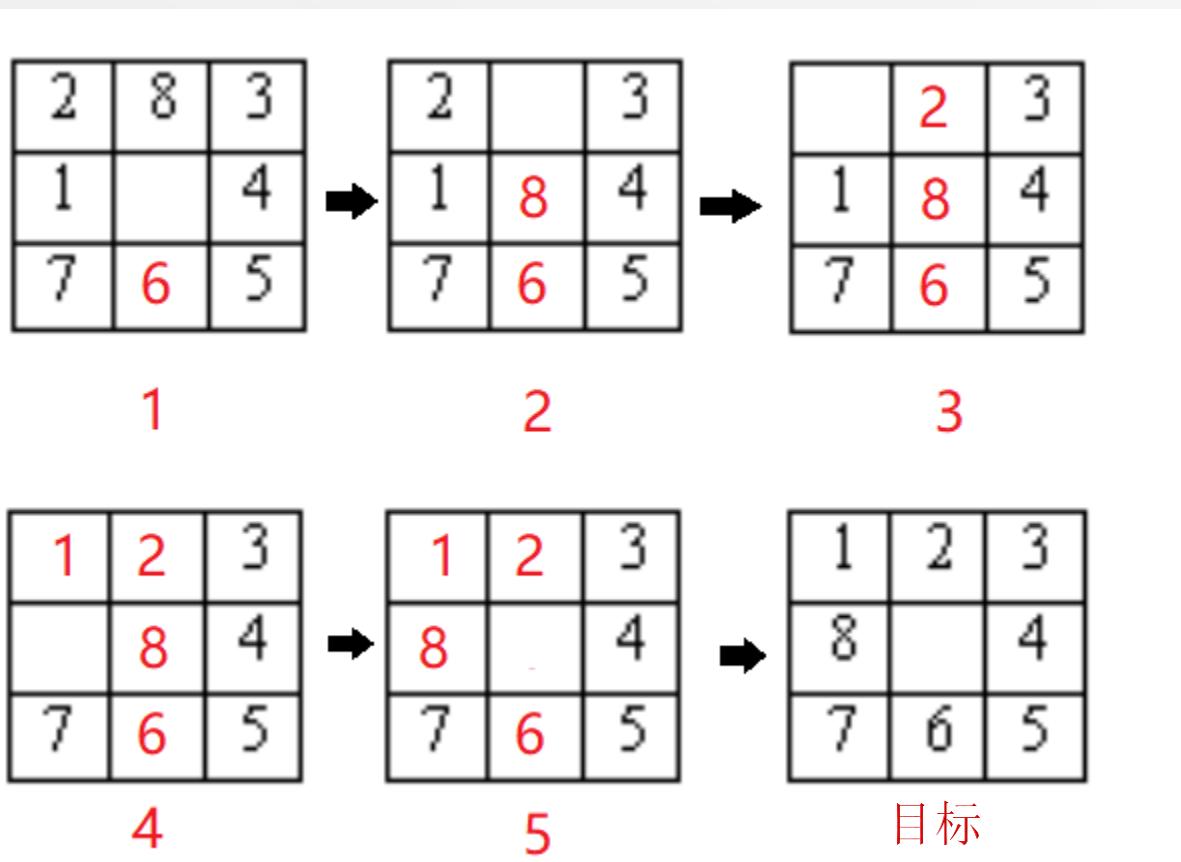
解题思路加油站：

纸和笔才是你思考的好帮手，键盘只是一个打工仔，敲键盘之前先看看你的草稿纸吧！



# 1

# 八数码问题



2	8	3
1	6	4
7		5

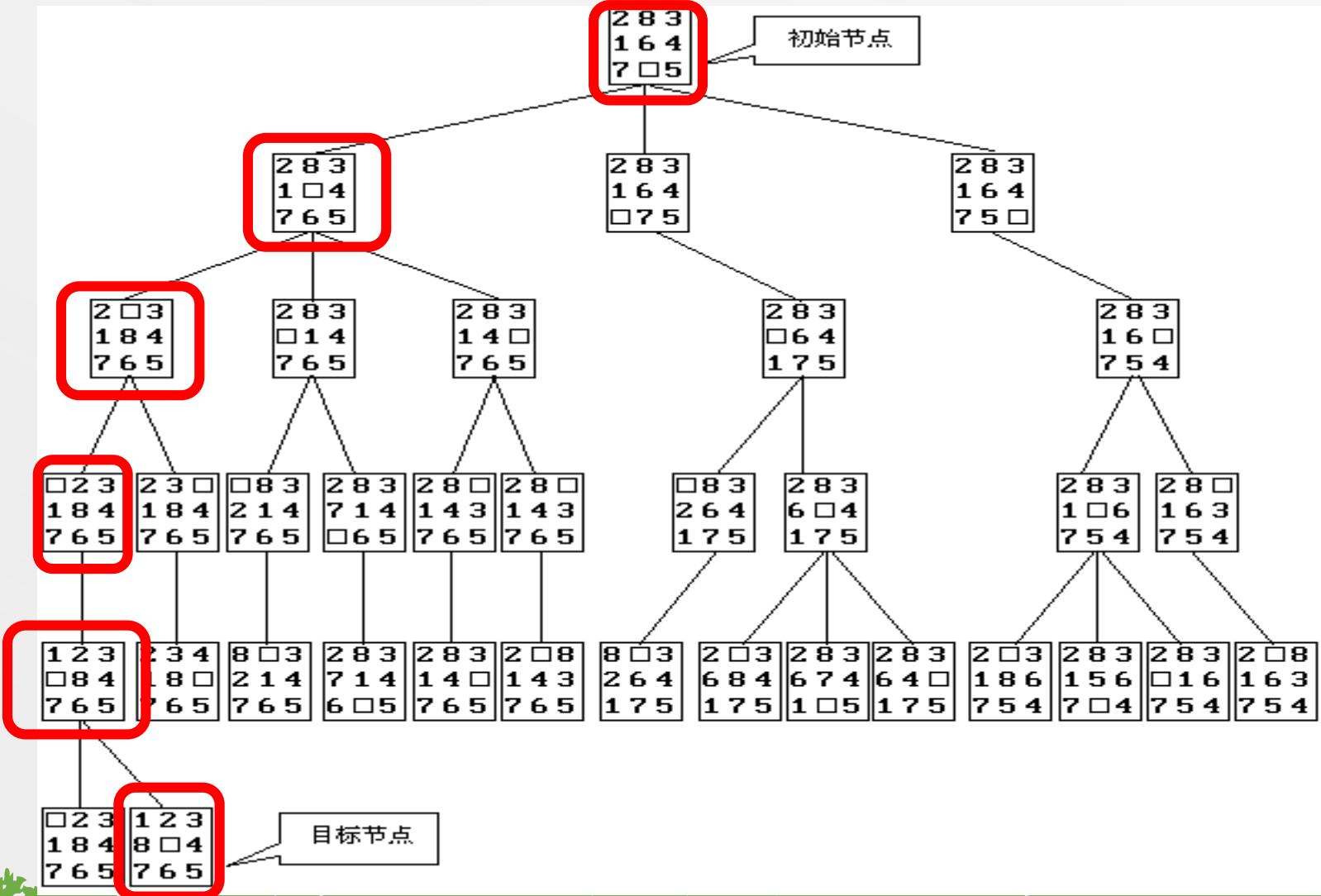
1	2	3
8		4
7	6	5

八数码问题的初始和目标状态



1

# 八数码问题扩展过程



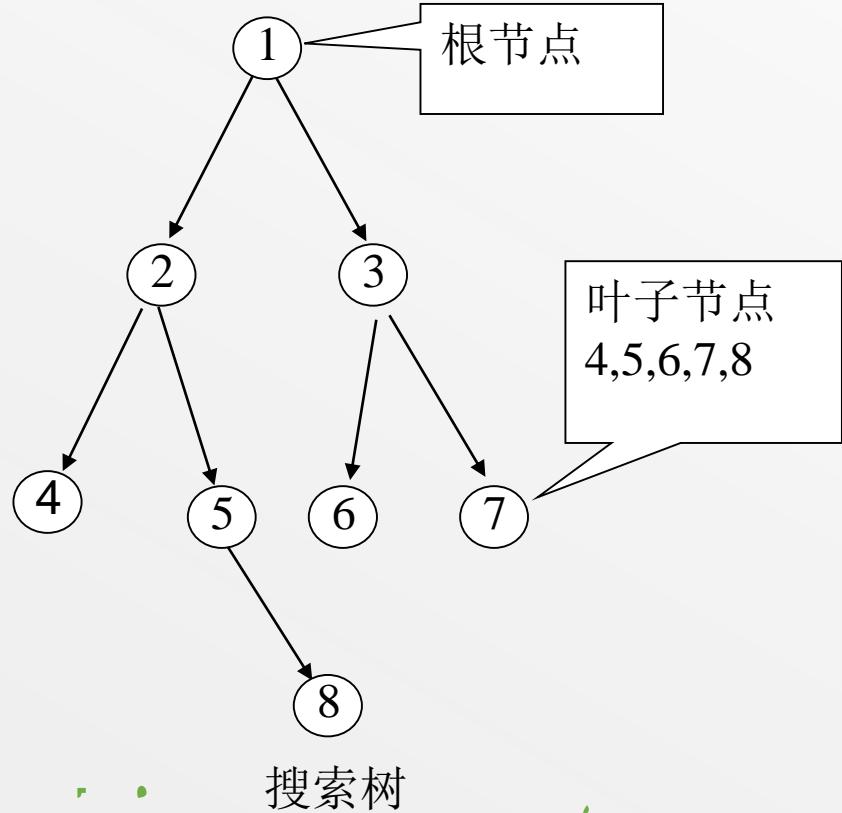
# 1

## 一些基本概念

- 节点：记录扩展的状态。
- 弧/边：记录扩展的路径。
- 搜索树：描述搜索扩展的整个过程。
- 节点的耗散值

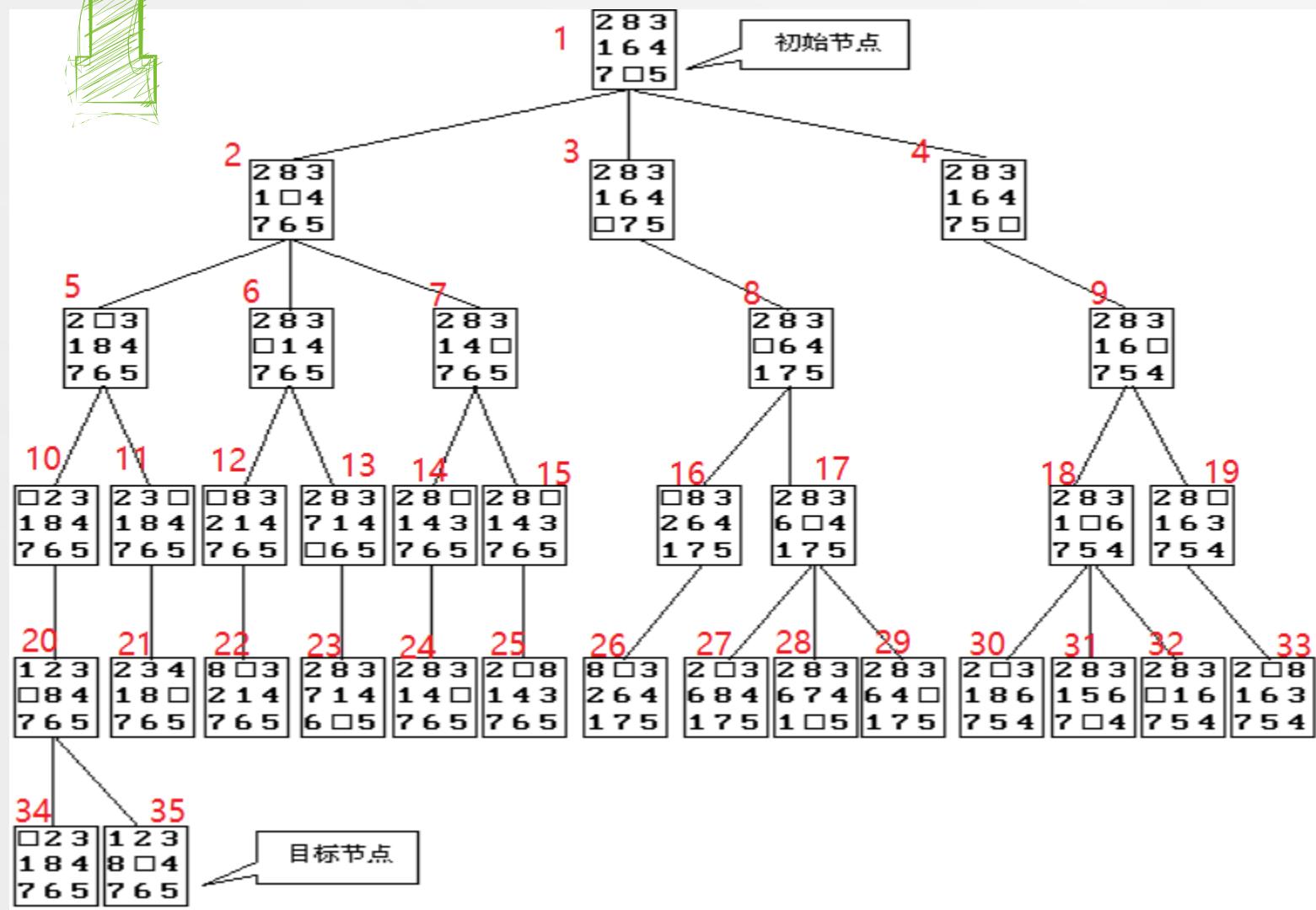
令 $C(i,j)$ 为从节点 $n_i$ 到 $n_j$ 的这段路径（或者弧）的耗散值，一条路径的耗散值就等于连接这条路径各节点间所有弧的耗散值总和。可以用递归公式描述如下：

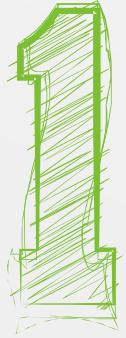
$$C(n_i, t) = C(n_i, n_j) + C(n_j, t)$$



1

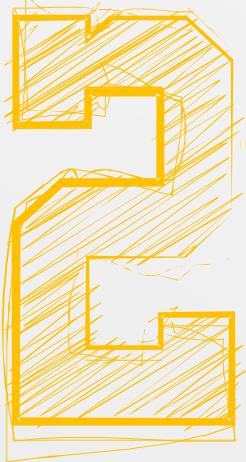
# 八数码问题扩展过程



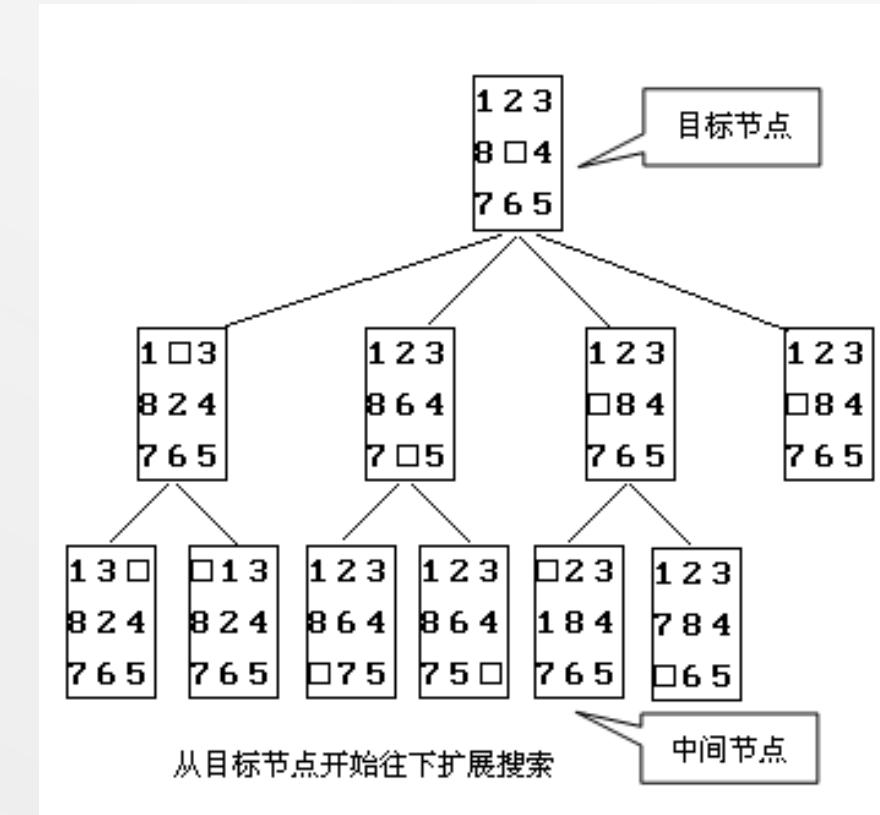
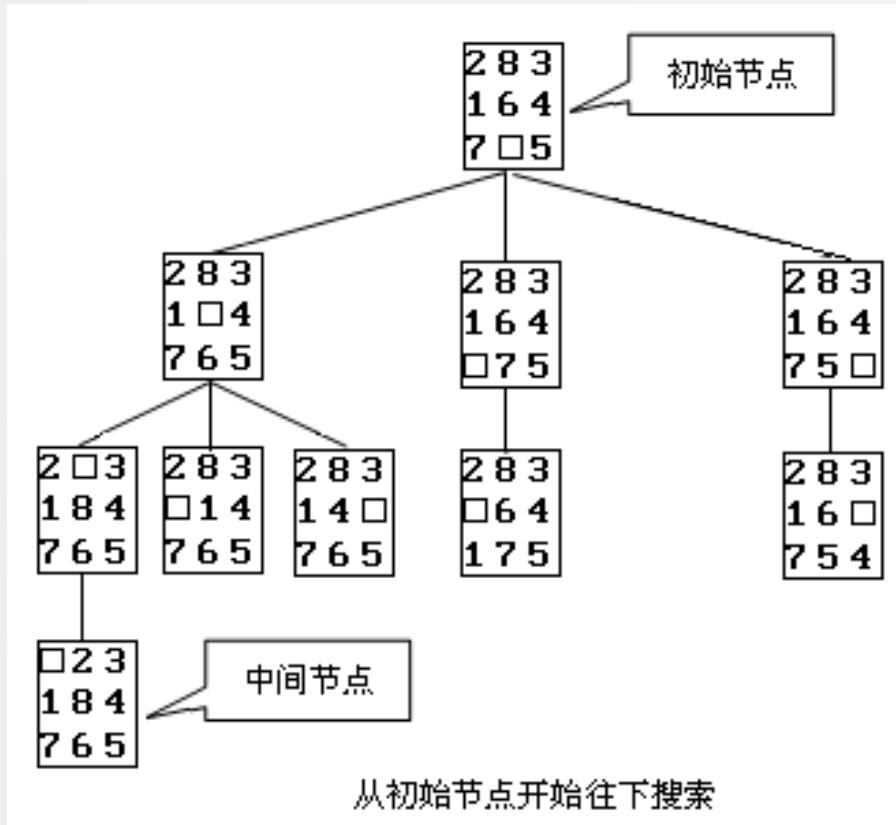


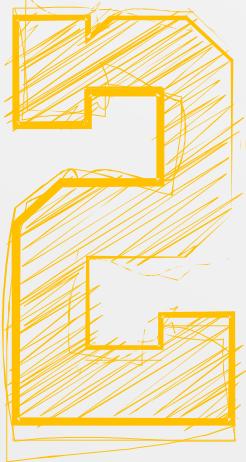
# 广度优先搜索

```
int bfs()
{
    初始化，初始状态存入队列；
    队列首指针head=0; 尾指针tail=1;
    do
    {
        指针head后移一位，指向待扩展结点；
        for (int i=1;i<=max;++i)          //max为产生子结点的规则数
        {
            if (子结点符合条件)
            {
                tail指针增1，把新结点存入列尾；
                if (新结点与原已产生结点重复) 删去该结点（取消入队，tail减1）；
                else
                    if (新结点是目标结点) 输出并退出；
            }
        }
    }while(head<tail);           //队列为空
}
```



# DBFS 双向广度优先遍历





# DBFS 双向广度优先遍历

- 规则必须可逆
- 随时判重
- 交叉扩展

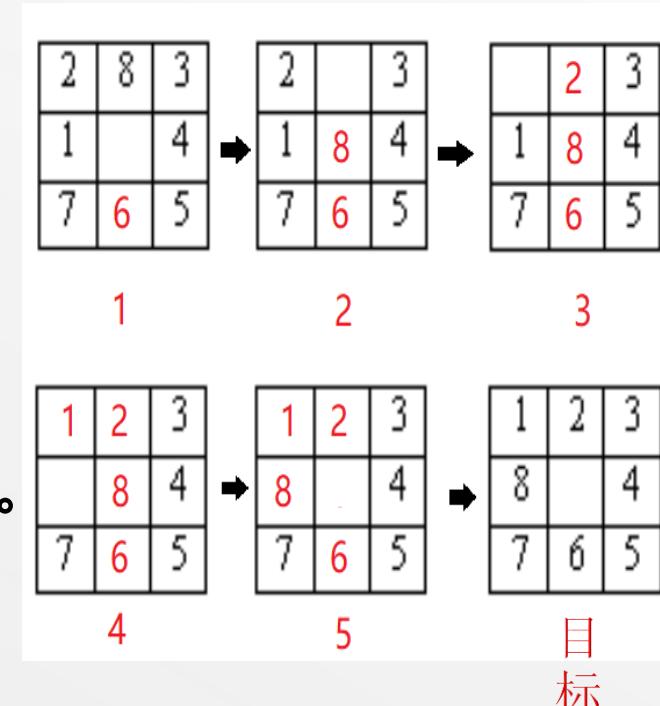
# 9

## A\* (启发式搜索)

- 启发式搜索是利用问题拥有的启发信息来引导搜索，达到减少搜索范围，降低问题复杂度的目的。这种利用启发信息的搜索过程称为启发式搜索方法。

- 评价函数

为了提高搜索效率，引入启发信息来进行搜索，在启发式搜索过程中，要对open 表进行排序，这就需要有一种方法来计算待扩展节点有希望通向目标节点的程度，我们总希望能找到最有希望通向目标节点的待扩展节点优先扩展。一种常用的方法就是定义评价函数  $f$  ( evaluation function) 对各个节点进行计算，**其目的就是估算出“有希望”的节点来。**



## 定义评价函数/估计函数

$$f(n) = g(n) + h(n)$$

n是被评价的节点

$g(n)$ : 表示从初始节点S到节点n的路径的代价;

$h(n)$ : 表示从节点n到目标节点G的路径的代价;

$f(n)$ : 表示从 初始节点S经过n到目标节点G的路径的代价



启发策略：

$$f(n) = g(n) + h(n),$$

其中：

$g(n)$ ：为层数。

$h(n)$ ：为当前状态“不在位”（位置不符的数码个数）的数量。



## **第一步基本思想：**

当一个节点扩展后，在它的所有子节点中，选估价函数 $f(n)$ 最优者作为下一个考察的节点。

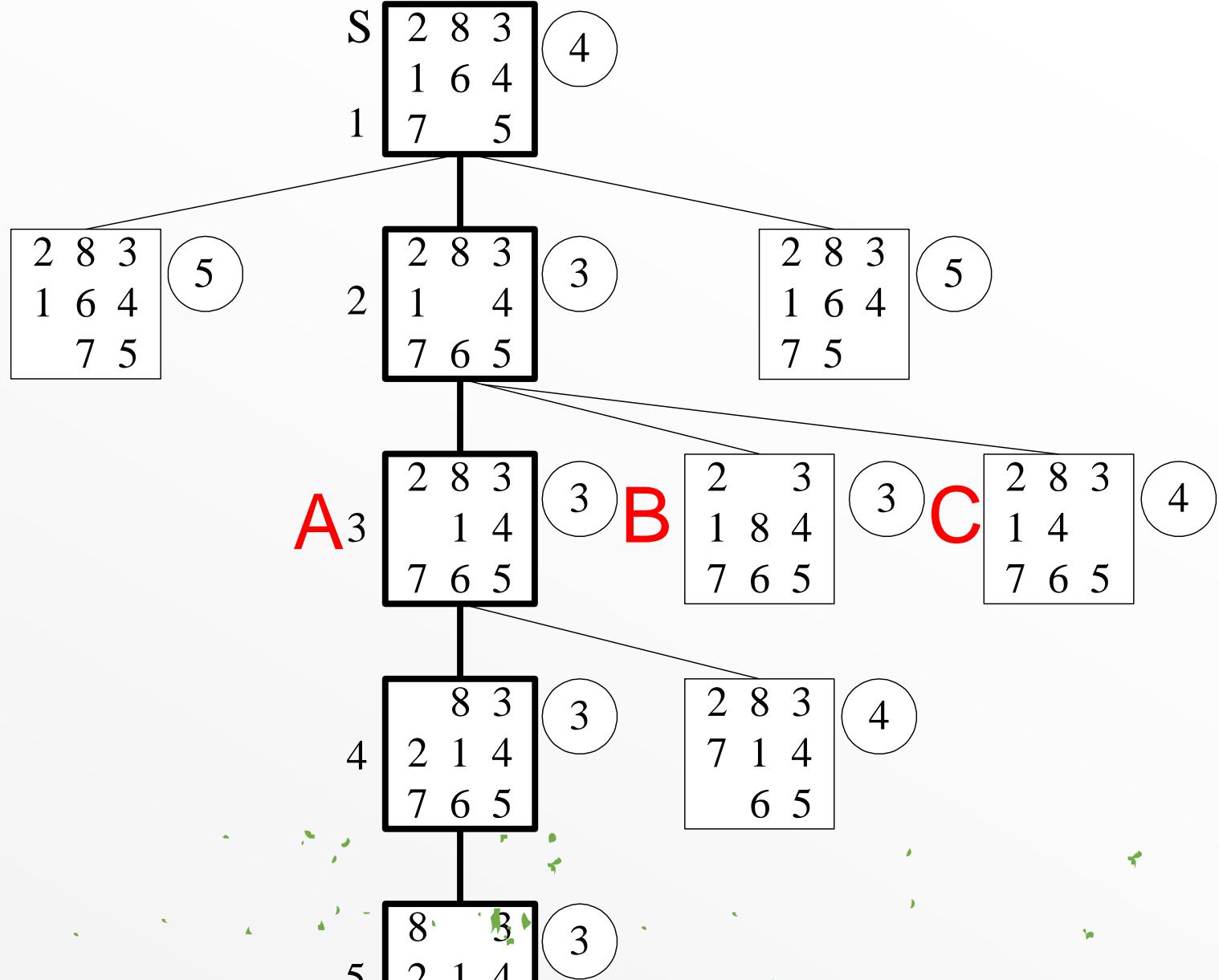


# 第一步：

哪一步能到达最优解？？

**局部择优搜索**

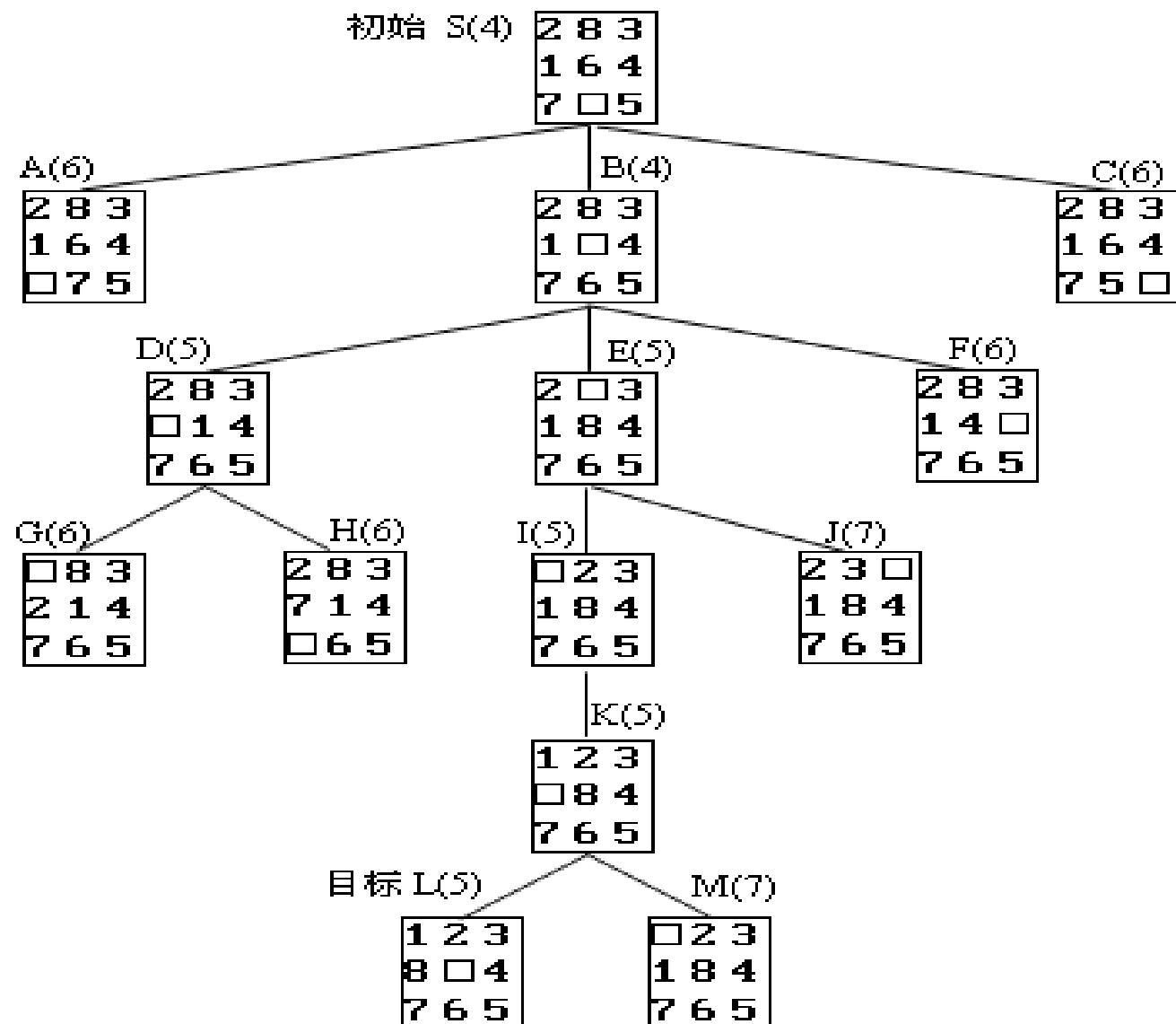
亦称“瞎子爬山法”。  
找不到最佳解。



## **第二步基本思想：**

当一个节点扩展后，从所有已生成而未扩展的节点中，选出最优的节点进行扩展。**全局择优搜索。**

3



八数码问题启发搜索过程

# 3

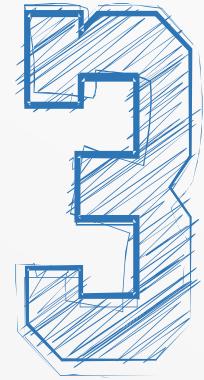
## A\*搜索的性质

**性质1：**若有两个A\*算法A1和A2,若A2比A1有较多的启发信息（即对所有非目标接点均有 $h_2(n) > h_1(n)$ ），则在具有一条从s到t的隐含图上，搜索结束时，由A2所扩展的每一个节点，也必定由A1所扩展，即A1扩展的节点数至少和A2一样多。

- 根据这个性质可知，使用启发函数 $h(n)$ 的A\*算法，比不使用 $h(n)$ （ $h(n)=0$ ）的算法，求得最佳路径时扩展的节点要少，使用 $h(n)$ 启发信息强的A\*算法比使用 $h(n)$ 启发信息弱的A\*算法扩展的节点数要少。因此，我们在考虑问题时，在满足 $h(n) \leq h^*(n)$ 的条件下，要尽量扩大 $h$ 函数的值。

**性质2：**若存在初始节点s到目标节点t的路径，则A\*必能找到最佳路径。如果A\*选作扩展的任一节点n，有 $f(n) \leq f^*(s)$ 。

**性质3：**若 $h(n)$ 满足单调限制条件，也就是说启发函数 $h$ 满足单调性，则A\*扩展了节点n之后就已经找到了到达节点n的最佳路径。即若A\*选n来扩展，在单调限制条件下有 $g(n) = g^*(n)$ ，其由A\*所扩展的节点序列，其f值也是非递减的，即 $f(n_i) \leq f(n_j)$ 。



# A\*搜索的性质

## 1) 基本思想:

定义一个评价函数 $f()$  (进行评估):

$$f(n) = g(n) + h(n)$$

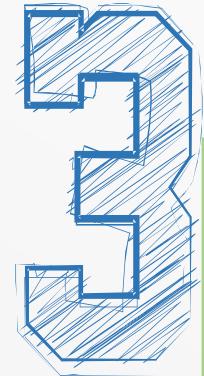
其中n是被评价的节点。

$g^*(n)$ : 表示从**初始节点S**到**节点n**的最短路径的代价; 本题中 $g(n) == g^*(n)$

$h^*(n)$ : 表示从**节点n**到**目标节点G**的最短路径的代价; 本题中显然 $h(n) <= h^*(n)$

$f^*(n)=g^*(n)+h^*(n)$ : 表示从**初始节点S**经过**节点n**到**目标节点G**的最短路径的代价。

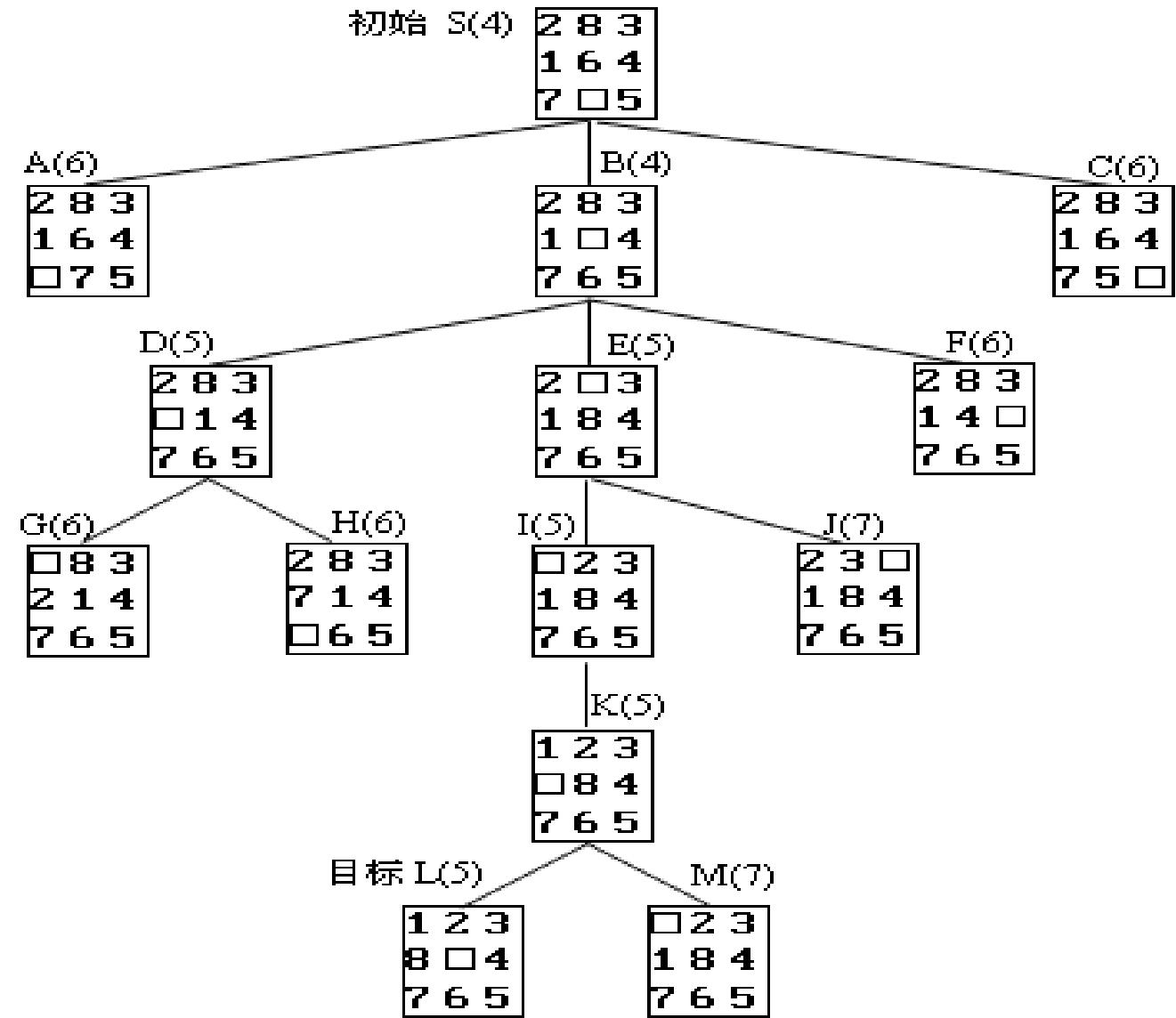
$f(n)$ 、 $g(n)$  和 $h(n)$  则分别表示是对 $f^*(n)$ 、 $g^*(n)$  和 $h^*(n)$  三个函数值的估计值。



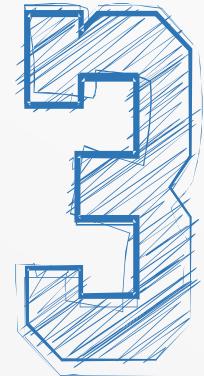
启发函数根据任意节点与目标之间的差异来定义，“**不在位**”将牌个数估计，取  $h(n) = W(n)$ 。

**$W(n) \leq h^*(n)$**

尽管对具体的  $h^*(n)$  是多少很难确切知道，但能确定得出：至少要移动  $W(n)$  步才能达到目标。



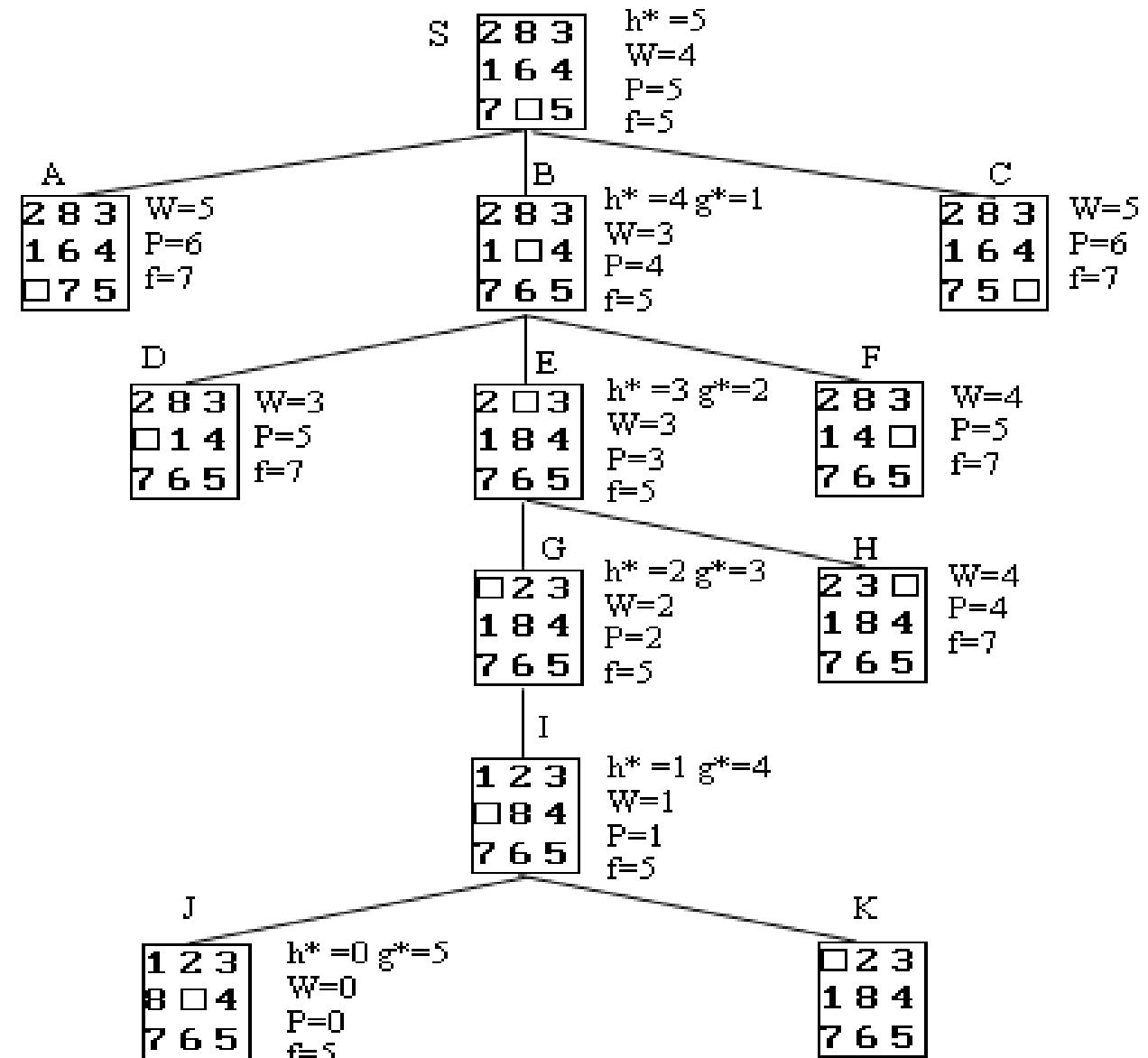
八数码问题启发搜索过程



启发函数根据任意节点与目标之间的距离的信息  
P(n)来定义

P(n)定义为每一个将牌与其目标位置之间距离的总和，取  $h(n) = P(n)$ 。

**P(n)  $\leq$  h<sup>\*</sup>(n)**



八数码问题利用将牌路径和作为启发函数的搜索过程

算法	宽度优先	深度优先	W作为启发函数A*	P作为启发函数 A*	双向搜索
扩展的节点数	35	至少35	14	11	21
稳定性	稳定	不稳定	稳定	稳定	基本稳定

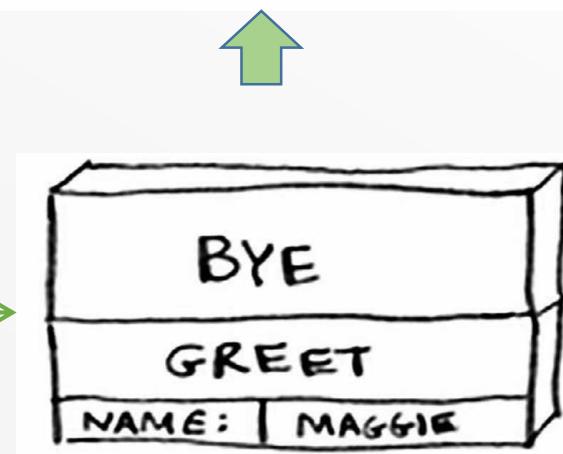
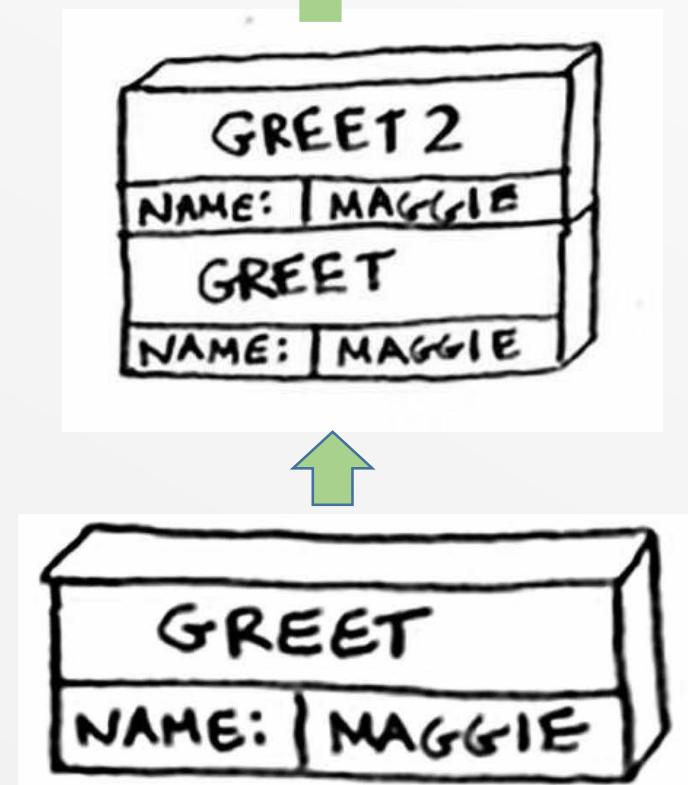
# 什么是递归????



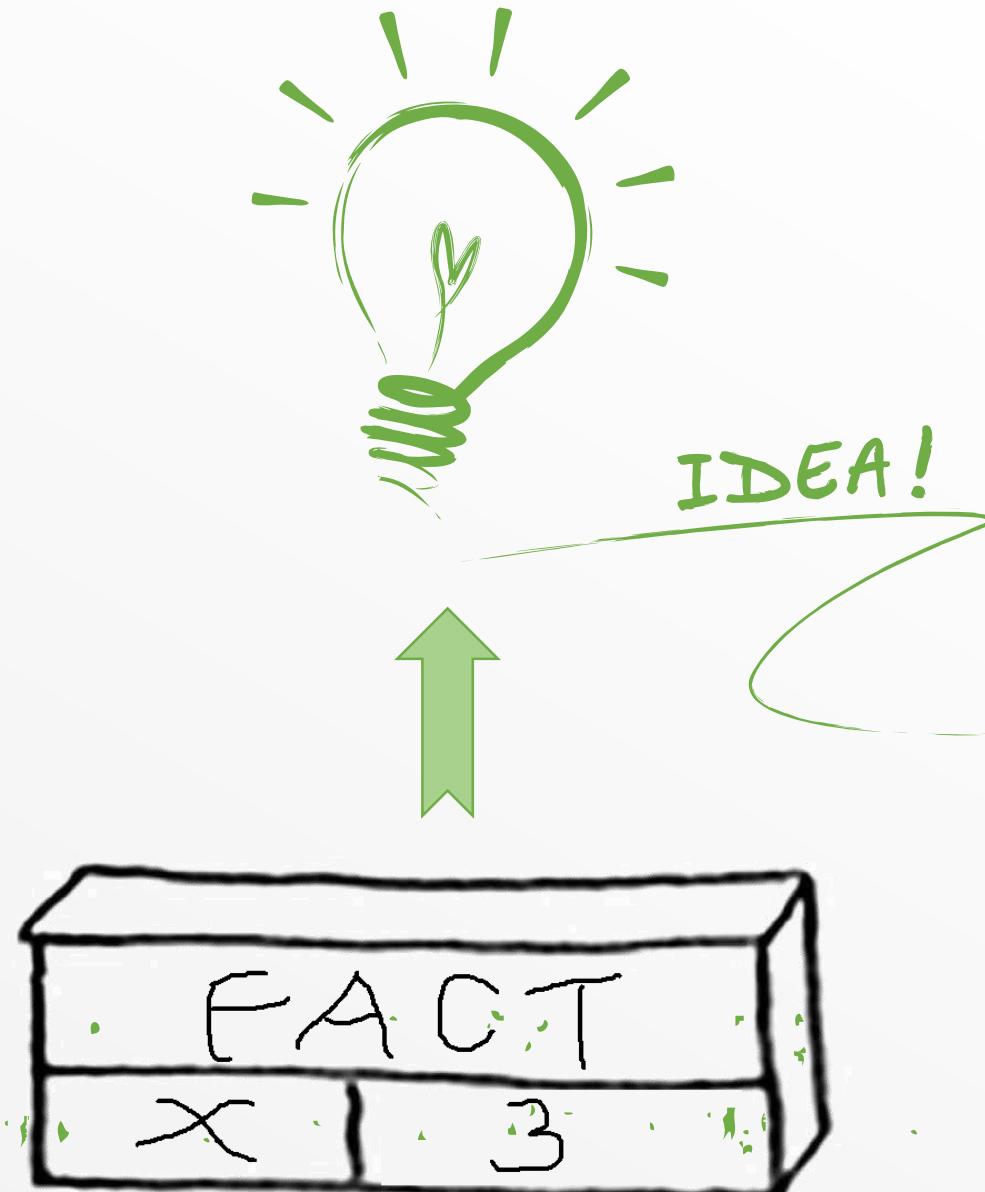
```
def greet(name):  
    print "hello, " + name + "!"  
    greet2(name)  
    print "getting ready to say bye..."  
    bye()
```

```
def greet2(name):  
    print "how are you, " + name + "?"  
def bye():  
    print "ok bye! "
```

```
def greet(name):  
    print "hello, " + name + "!"  
    greet2(name)  
    print "getting ready to say bye..."  
    bye()
```



```
def fact(x):  
    if x == 1:  
        return 1  
    else:  
        return x * fact(x-1)
```



# 从深搜开始！！

## 数字金字塔

观察下面的数字金字塔。写一个程序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。每一步可以从当前点走到左下方的点也可以到达右下方的点。

在上面的样例中,从13到8到26到15到24的路径产生了最大的和86。

### 输入:

第一个行包含 $R(1 \leq R \leq 1000)$ , 表示行的数目。

后面每行为这个数字金字塔特定行包含的整数。

所有的被供应的整数是非负的且不大于100。

### 输出:

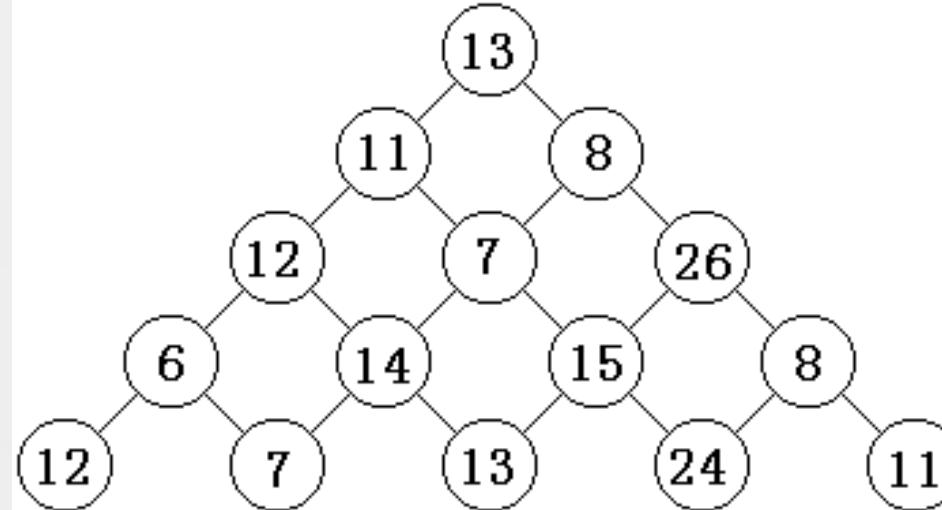
单独的一行, 包含那个可能得到的最大的和。

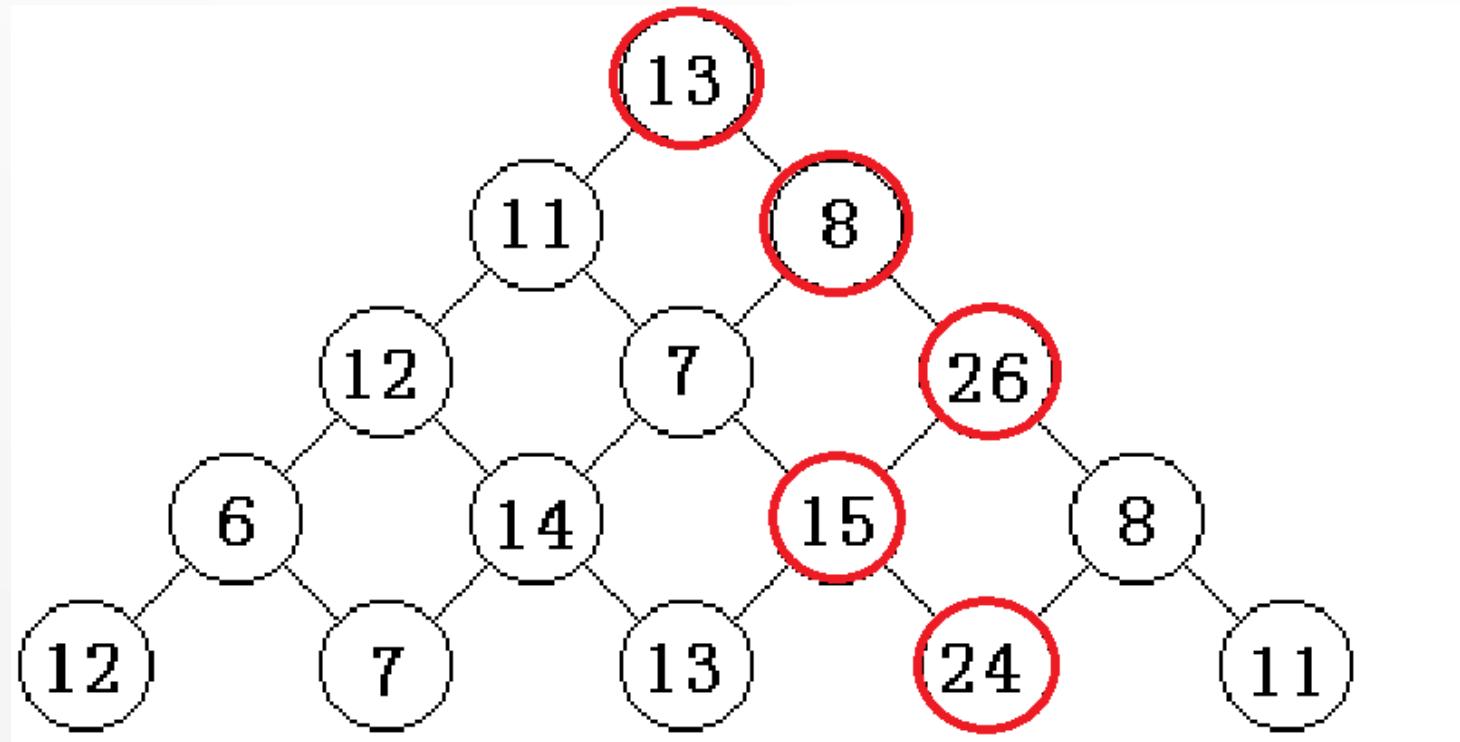
### 样例输入:

```
5          //数塔层数
13
11 8
12 7 26
6 14 15 8
12 7 13 24 11
```

### 样例输出:

```
86
```





1.  $\text{min}(x) = \text{leftmost}(x)$   
2.  $\text{max}(x) = \text{rightmost}(x)$   
3.  $\text{parent}(x) = \text{parent}(x)$   
4.  $\text{left}(x) = \text{left}(x)$   
5.  $\text{right}(x) = \text{right}(x)$

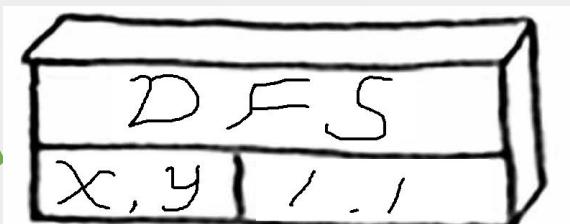
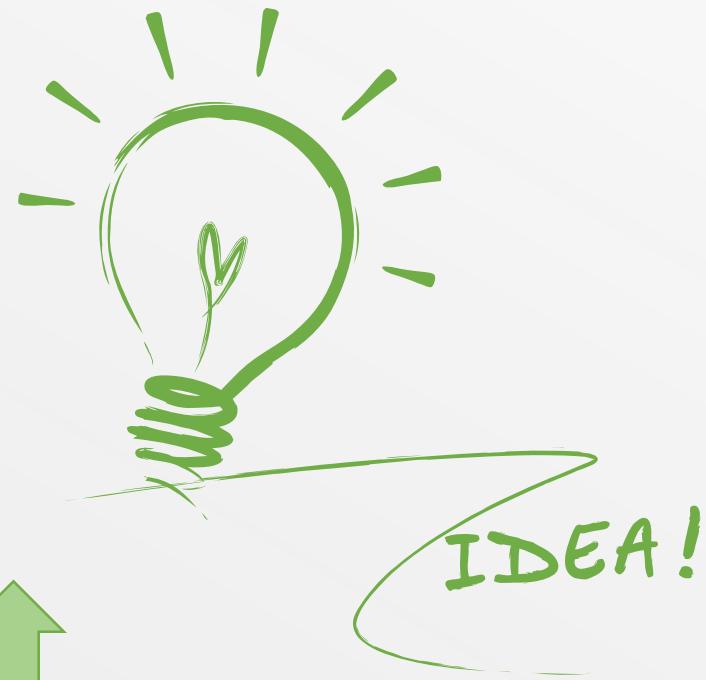
## 递归回溯法算法框架[一]

```
int Search(int k)
{
    for (i=1;i<=算符种数;i++)
        if (满足条件)
    {
        保存结果
        if (到目的地) 输出解;
        else Search(k+1);
        恢复: {回溯一步}
    }
}
```

## 递归回溯法算法框架[二]

```
int Search(int k)
{
    if (到目的地) 输出解;
    else
        for (i=1;i<=算符种数;i++)
            if (满足条件)
            {
                保存结果;
                Search(k+1);
                恢复: {回溯一步}
            }
}
```

```
void Dfs(int x,int y,int Curr)
{
    if (x==N)
    {
        if (Curr>Ans)Ans=Curr;
        return;
    }
    Dfs(x+1,y,Curr+A[x+1][y]);
    Dfs(x+1,y+1,Curr+A[x+1][y+1]);
}
```



该方法实际上是**自上而下**把所有路径都走了一遍，由于每一条路径都是由N-1步组成，每一步（每层每个节点）有“左”、“右”两种选择，因此路径总数为 $2^{N-1}$ ，所以该方法的时间复杂度为 $O(2^{N-1})$ ，数据量上升一定超时。

二叉树的遍历  
前序遍历  
中序遍历  
后序遍历

## 方法二：记忆化搜索

记忆化搜索需要对方法一中的搜索进行改装。由于需要记录从一个点开始到终点的路径的最大权值和，因此我们重新定义递归函数Dfs。

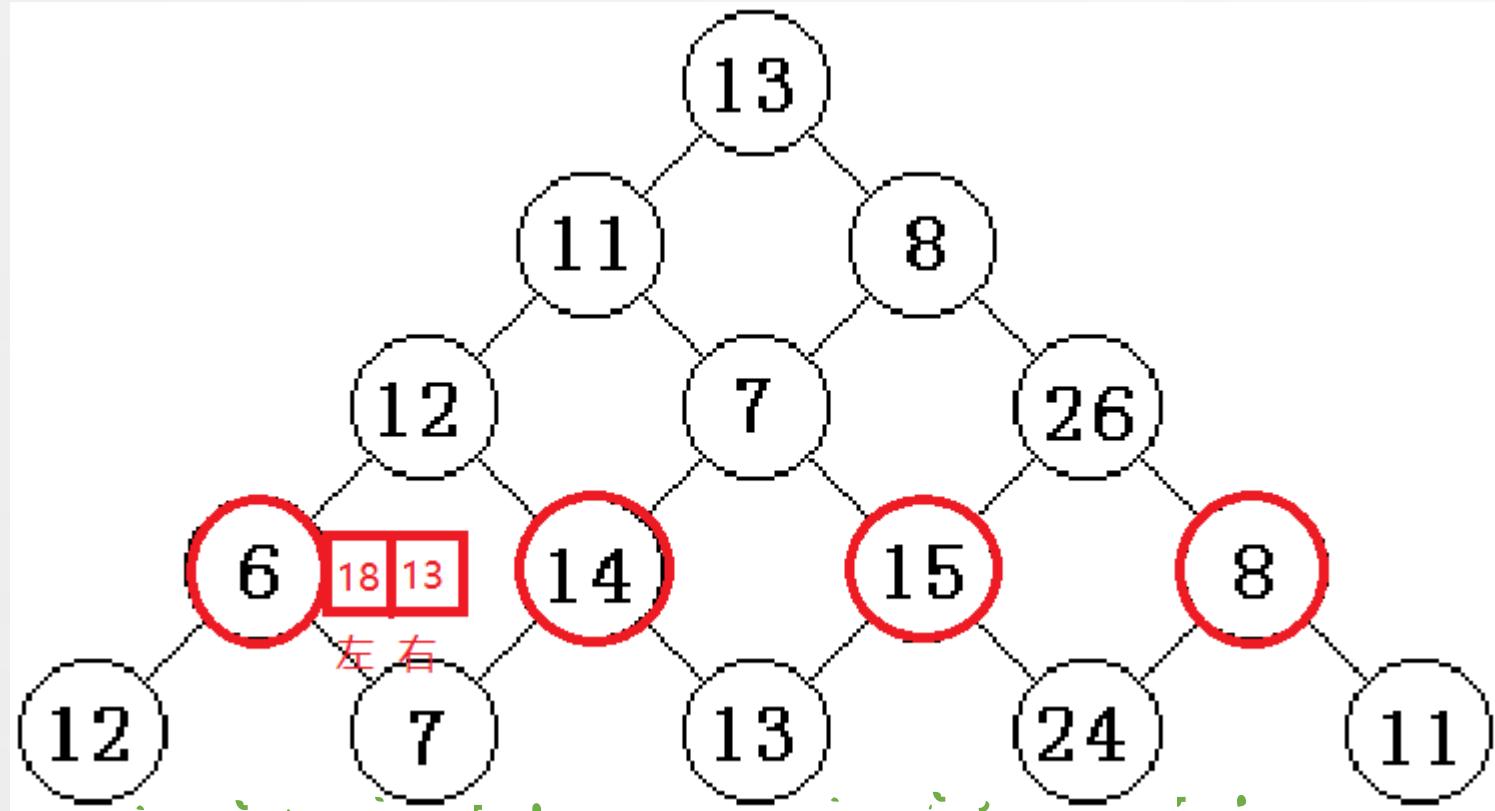
定义Dfs( $x, y$ )表示从 $(x, y)$ 出发到终点的路径的最大权值和，答案就是Dfs(1, 1)。计算Dfs( $x, y$ )时考虑第一步是向左还是向右，我们把所有路径分成两大类：

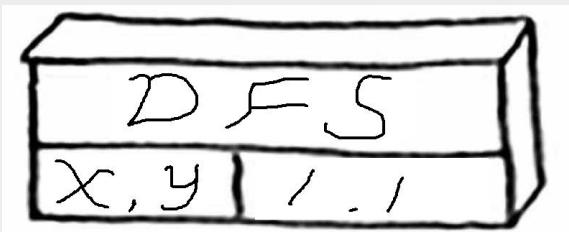
①第一步向左：那么从 $(x, y)$ 出发到终点的这类路径就被分成两个部分，先从 $(x, y)$ 到 $(x+1, y)$ 再从 $(x+1, y)$ 到终点，第一部分固定权值就是 $A[x][y]$ ，要使得这种情况的路径权值和最大，那么第二部分从 $(x+1, y)$ 到终点的路径的权值和也要最大，这一部分与前面的Dfs( $x, y$ )的定义十分相似，仅仅是参数不同，因此这一部分可以表示成Dfs( $x+1, y$ )。综上，第一步向左的路径最大权值和为 $A[x][y] + Dfs(x+1, y)$ ；

②第一步向右：这类路径要求先从 $(x, y)$ 到 $(x+1, y+1)$ 再从 $(x+1, y+1)$ 到终点，分析方法与上面一样，这类路径最大权值和为 $A[x][y] + Dfs(x+1, y+1)$ ；

为了避免重复搜索，我们开设全局数组F[x][y]记录从 $(x, y)$ 出发到终点路径的最大权值和，一开始全部初始化为-1表示未被计算过。在计算Dfs( $x, y$ )时，首先查询F[x][y]，如果F[x][y]不等于-1，说明Dfs( $x, y$ )之前已经被计算过，直接返回 F[x][y]即可，否则计算出Dfs( $x, y$ )的值并存储在F[x][y]中。

## 方法二：记忆化搜索--自上而下拓展 --自下而上回归





```
int Dfs(int x,int y)
{
    if (F[x][y]==-1){
        if (x==N)F[x][y]=A[x][y];
        else F[x][y]=A[x][y]+max(Dfs(x+1,y),Dfs(x+1,y+1));
    }
    return F[x][y];
}
```

由于 $F[x][y]$ 对于每个合法的 $(x,y)$ 都只计算过一次，而且计算是在 $O(1)$ 内完成的，因此时间复杂度为 $O(N^2)$ 。



## 方法三：动态规划(顺推法)

方法二通过分析搜索的状态重复调用自然过渡到记忆化搜索，而记忆化搜索本质上已经是动态规划了。下面我们完全从动态规划的算法出发换一个角度给大家展示一下动态规划的解题过程，并提供动态规划的迭代实现法。

### ①确定状态：

题目要求从(1,1)出发到最底层路径最大权值和，路径中是各个点串联而成，路径起点固定，终点和中间点相对不固定。因此定义 $F[x][y]$ 表示从(1,1)出发到达(x,y)的路径最大权值和。最终答案 $Ans=\max\{F[N][1], F[N][2], \dots, F[N][N]\}$ 。

### ②确定状态转移方程和边界条件：

不去考虑(1,1)到(x,y)的每一步是如何走的，只考虑最后一步是如何走，根据最后一步是向左还是向右分成以下两种情况：

向左：最后一步是从(x-1,y)走到(x,y),此类路径被分割成两部分，第一部分是从(1,1)走到(x-1,y),第二部分是从(x-1,y)走到(x,y)，要计算此类路径的最大权值和，必须用到第一部分的最大权值和，此部分问题的性质与 $F[x][y]$ 的定义一样，就是 $F[x-1][y]$ ，第二部分就是 $A[x][y]$ ，两部分相加即得到此类路径的最大权值和为 $F[x-1][y]+A[x][y]$ ；

向右：最后一步是从(x-1,y-1)走到(x,y),此类路径被分割成两部分，第一部分是从(1,1)走到(x-1,y),第二部分是从(x-1,y)走到(x,y)，分析方法如上。此类路径的最大权值和为 $F[x-1][y-1]+A[x][y]$ ；  
 $F[x][y]$ 的计算需要求出上面两种情况的最大值。综上，得到状态转移方程如下：

$$F[x][y]=\max\{F[x-1][y-1], F[x-1][y]\}+A[x][y]$$

与递归关系式还需要递归终止条件一样，这里我们需要对边界进行处理以防无限递归下去。  
观察发现计算 $F[x][y]$ 时需要用到 $F[x-1][y-1]$ 和 $F[x-1][y]$ ，是上一行的元素，随着递归的深入，最终都要用到第一行的元素 $F[1][1]$ ,  
 $F[1][1]$ 的计算不能再使用状态转移方程来求，而是应该直接赋予一个特值 $A[1][1]$ 。这就是边界条件。

综上得：

$$\text{状态转移方程: } F[x][y]=\max\{F[x-1][y-1], F[x-1][y]\}+A[x][y]$$

$$\text{边界条件: } F[1][1]=A[1][1]$$

现在让我们来分析一下该动态规划的正确性，分析该解法是否满足使用动态规划的两个前提：

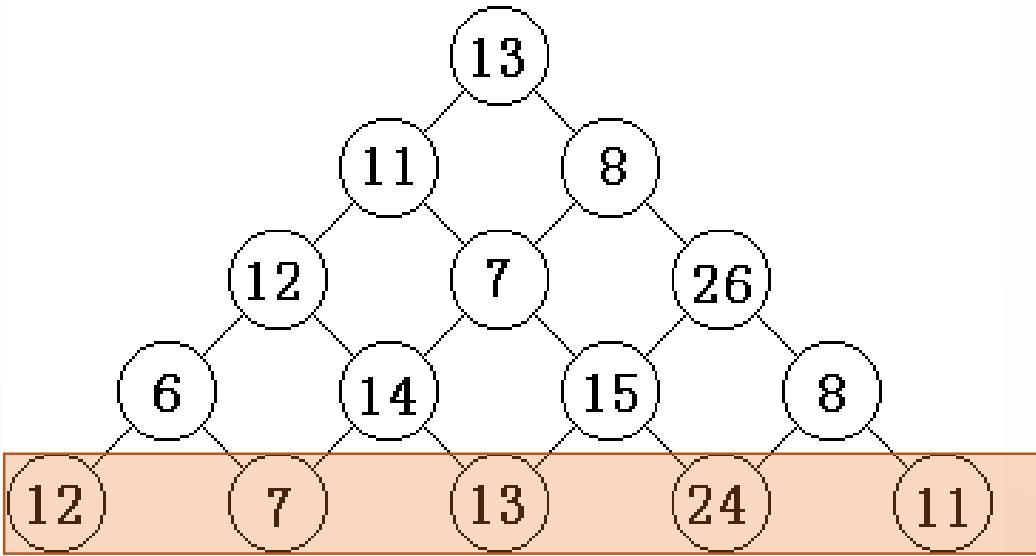
**最优化原理**：这个在分析状态转移方程时已经分析得比较透彻，明显是符合最优化原理的；

**无后效性**：状态转移方程中，我们只关心 $F[x-1][y-1]$ 与 $F[x-1][y]$ 的值，计算 $F[x-1][y-1]$ 时可能有多种不同的决策对应着最优值，选哪种决策对计算 $F[x][y]$ 的决策没有影响， $F[x-1][y-1]$ 也是一样。这就是无后效性。

### ③程序实现：

由于状态转移方程就是递归关系式，边界条件就是递归终止条件，所以可以用递归来完成，递归存在重复调用，利用记忆化可以解决重复调用的问题，方法二已经讲过。记忆化实现比较简单，而且不会计算无用状态，但递归也会受到“栈的大小”和“递推+回归执行方式”的约束，另外记忆化实现调用状态的顺序是按照实际需求而展开，没有大局规划，不利于进一步优化。

这里介绍一种迭代法。与分析边界条件方法相似，计算 $F[x][y]$ 用到状态 $F[x-1][y-1]$ 与 $F[x-1][y]$ ，这些元素在 $F[x][y]$ 的上一行，也就是说要计算第 $x$ 行的状态的值，必须要先把第 $x-1$ 行元素的值计算出来，因此我们可以先把第一行元素 $F[1][1]$ 赋为 $A[1][1]$ ，再从第二行开始按照行递增的顺序计算出每一行的有效状态即可。时间复杂度为 $O(N^2)$ 。



$A(x,y)$ 表示各个节点的值  
本题的目标是)出发到最底层路径最大权值和  
因此 定义 $F[x][y]$ 表示到达 $(x,y)$ 节点时权值的最大值!  
就可以知道 $Ans = \max\{F[N][1], F[N][2], \dots, F[N][N]\}$

$A(x,y)$  表示各个节点的值

$F(x,y)$  表示从(1,1)出发到达(x,y)的路径最大权值和

变化量	1, 1	2, 1	2, 2	3, 1	3, 2	3, 3	.....
$A(x,y)$							
$F(x,y)$ 产生过程							
$F(x,y)$							

状态转移方程： $F[x][y] = \max\{F[x-1][y-1], F[x-1][y]\} + A[x,y]$

边界条件： $F[1][1] = A[1][1]$

解题思路加油站：

纸和笔才是你思考的好帮手，键盘只是一个打工仔，敲键盘之前先看看你的草稿纸吧！

```
for(int i = 2;i <= N;i ++)
    for(int j = 1;j <= i;j++)
        F[i][j]=max(F[i-1][j-1],F[i-1][j])+A[i][j];
```

# 练习指南



宽搜	深搜	记忆化搜索	动态规划
反恐精英	poj1321	poj1088	单位矩阵
迷宫	<b>P1731生日蛋糕</b>	余数最大	逃离地牢
逃离机场	hdoj2437	hdoj1078	维和部队
<b>剪指甲</b>	zoj1204	poj2111	勇士传说
<b>HDOJ3085</b>	hdoj1501		首都选址
<b>P1949聪明的打字员</b>	hdoj1044		

# 附录一： 搜索与回溯算法



搜索与回溯是计算机解题中常用的算法，很多问题无法根据某种确定的计算法则来求解，可以利用搜索与回溯的技术求解。回溯是搜索算法中的一种控制策略。它的基本思想是：为了求得问题的解，先选择某一种可能情况向前探索，在探索过程中，一旦发现原来的选择是错误的，就退回一步重新选择，继续向前探索，如此反复进行，直至得到解或证明无解。

如迷宫问题：进入迷宫后，先随意选择一个前进方向，一步步向前试探前进，如果碰到死胡同，说明前进方向已无路可走，这时，首先看其它方向是否还有路可走，如果有路可走，则沿该方向再向前试探；如果已无路可走，则返回一步，再看其它方向是否还有路可走；如果有路可走，则沿该方向再向前试探。按此原则不断搜索回溯再搜索，直到找到新的出路或从原路返回入口处无解为止。

## 递归回溯法算法框架[一]

```
int Search(int k)
```

```
{
```

```
for (i=1;i<=算符种数;i++)
```

```
if (满足条件)
```

```
{
```

    保存结果

    if (到目的地) 输出解;

        else Search(k+1);

    恢复：保存结果之前的状态{回溯一步}

```
}
```

```
}
```

## 递归回溯法算法框架[二]

```
int Search(int k)
```

```
{
```

```
    if (到目的地) 输出解;
```

```
    else
```

```
        for (i=1;i<=算符种数;i++)
```

```
            if (满足条件)
```

```
{
```

```
                保存结果;
```

```
                Search(k+1);
```

```
                恢复：保存结果之前
```

```
                的状态{回溯一步}
```

```
}
```

八皇后问题：要在国际象棋棋盘中放八个皇后，使任意两个皇后都不能互相吃。（提示：皇后能吃同一行、同一列、同一对角线的任意棋子。）

放置第 i 个(行)皇后的算法为：

```
int search(i);
{
    int j;
    for (第i个皇后的位置j=1;j<=8;j++) //在本行的8列中去试
        if (本行本列允许放置皇后)
    {
        放置第i个皇后;
        对放置皇后的位置进行标记;
        if (i==8) 输出 //已经放完个皇后
        else search(i+1); //放置第i+1个皇后
        对放置皇后的位置释放标记，尝试下一个位置是否可行;
    }
}
```

## 附录二： FIFO队列和优先队列

在实际应用中，我们将会大量使用到FIFO队列和优先队列，本节将介绍这两个C++自带算法。

当然，使用他们必须包含 queue 头文件。

### 定义FIFO队列

queue<类型名> 变量名

queue<int> que //定义que为一个  
int类型的FIFO队列

queue<char> a //定义 a 为一个  
char 类型的FIFO队列

queue<data> c //定义c为一个  
data类型的FIFO队列

//其中data为自定义的数据类型，可以为结构体

### 定义简单的优先队列

priority\_queue <int> heap; //定义heap为一  
个int类型的优先队列

priority\_queue <double> k; //定义k为一个  
double类型的优先队列

这两种定义方式都是大根堆，想要使其变成小根堆，可以将每个数  
据都乘以-1

## 定义结构体的优先队列

```
struct data{  
    int x;  
    bool operator < (const data & a) const { //const一定要写  
        return a.x<x;           //等于小根堆  
    }  
};  
priority_queue <data>q; //q 优先队列的优先规则由data重载小于号决定
```

# FIFO队列和优先队列的操作

q. empty() ( )	如果队列为空，则返回true,否则返回false
q. size()	返回队列中元素的个数
q. pop()	删除队首元素，但不返回其值
q. front() ( )	返回队首元素的值，但不删除该元素（仅适用于FIFO队列）
q. back()	返回队尾元素的值，但不删除该元素（仅适用于FIFO队列）
q. top()	返回具有最高优先级的元素的值，但不删除该元素（仅适用于优先队列）
q. push() ( )	对queue，在队尾压入一个新元素 对于priority_queue,在基于优先级的适当位置插入新元素

题目描述:

在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。

每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过 $n-1$ 次合并之后，就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。

因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。假定已知每个果子重量都为1，并且已知果子的种类数和每种果子的数目，你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。

例如有三种果子，数目依次为1, 2, 9。可以先将1、2堆合并，新堆数目为3，耗费体力为3。接着，将新堆与原先的第三堆合并，又得到新的堆，数目为12，耗费体力为12。所以多多总共耗费体力为 $3+12=15$ 。可以证明15为最小的体力耗费值。

输入格式:

输入文件包括两行，第一行是一个整数  $n$  ( $1 \leq n \leq 10000$ )，表示果子的种类数。第二行包含  $n$  个整数，用空格分隔，第  $i$  个整数  $a_i$  ( $1 \leq a_i \leq 20000$ ) 是第  $i$  种果子的数目。

样例数据:

输入

3

1 2 9

输出

15

备注:

**【数据规模】**

对于 30% 的数据，保证有  $n \leq 1000$ ;

对于 50% 的数据，保证有  $n \leq 5000$ ;

对于全部的数据，保证有  $n \leq 10000$ 。



利用C++自带优先队列的合并果子代码：本质是数据结构中的堆！！

```
#include<cstdio>
#include<iostream>
#include<queue>
using namespace std;
priority_queue <int> que;
int main() {
    int n;      scanf("%d",&n);
    for (int i=0,x;i<n;++i) {
        scanf("%d",&x); que.push(-x);
    }
    int ans=0;
    for (int i=1,tmp;i<n;++i) {
        tmp=que.top();  ans-=que.top();
        que.pop();       tmp+=que.top();
        ans-=que.top(); que.pop();
        que.push(tmp);  }
    cout<<ans<<endl;   return 0;
}
```