
中国科学技术大学
University of Science and Technology of China
程序设计 II
Programming Design II
搜索
主讲：吴锋

目录
CONTENTS

搜索的思想

例题1：八数码问题

例题2：木棒问题

枚举的思想

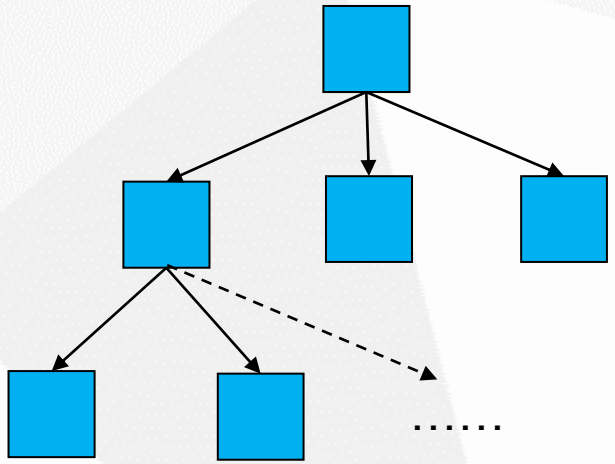
- 枚举：逐一判断所有可能的方案是否是问题的解。
- 例：求出A-I这九个字母对应的数字（1-9），使得下式成立（一一对应）。

$$\begin{array}{r} ABCD \\ \times \quad E \\ \hline FGHI \end{array}$$

- 解法思想：枚举ABCDE的值，计算乘积，判断是否符合要求。

搜索的思想

- 搜索（高级枚举）：有顺序有策略地枚举状态空间中的结点，寻找问题的解。



例题1：八数码问题

- 问题描述

初始状态

2	3	4
1	5	
7	6	8

移动序列 →

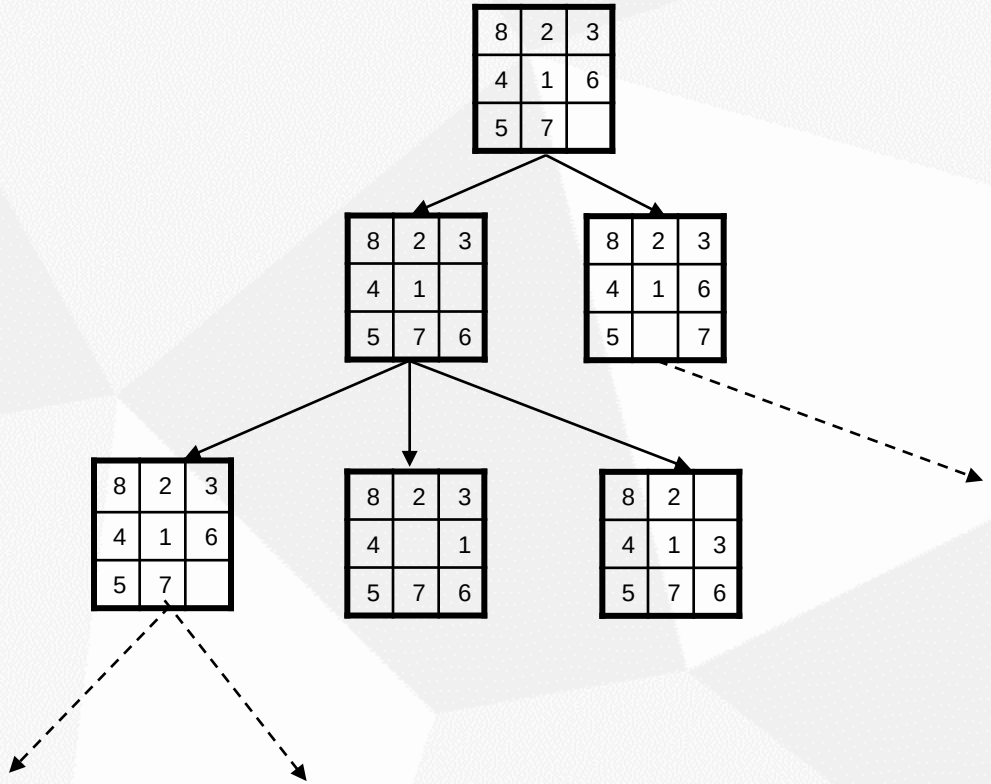
1	2	3
4	5	6
7	8	

目标状态

- 输入数据代表它左边的初始状态：2 3 4 1 5 x 7 6 8
- 输出数据是一个移动序列，使得移动后结果变成右边的目标状态：
  ulldrurdllurdruldr
- 移动序列中
  - u 表示使空格上移，d 表示使空格下移
  - r 表示使空格右移， l 表示使空格左移

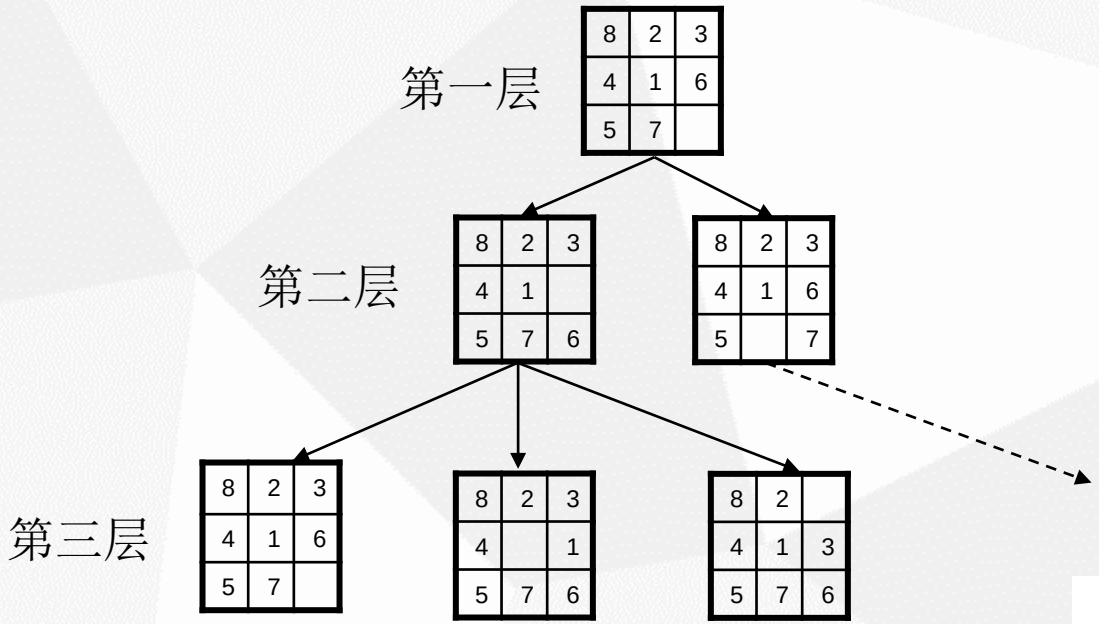
例题1：八数码问题

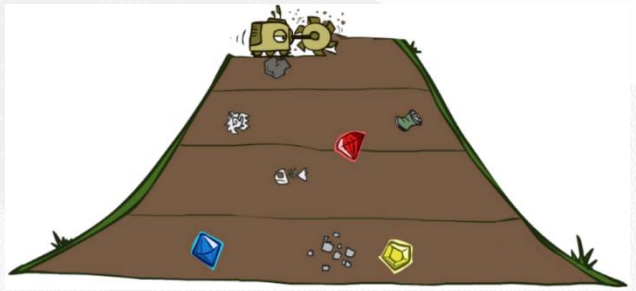
- 状态空间



例题1：八数码问题

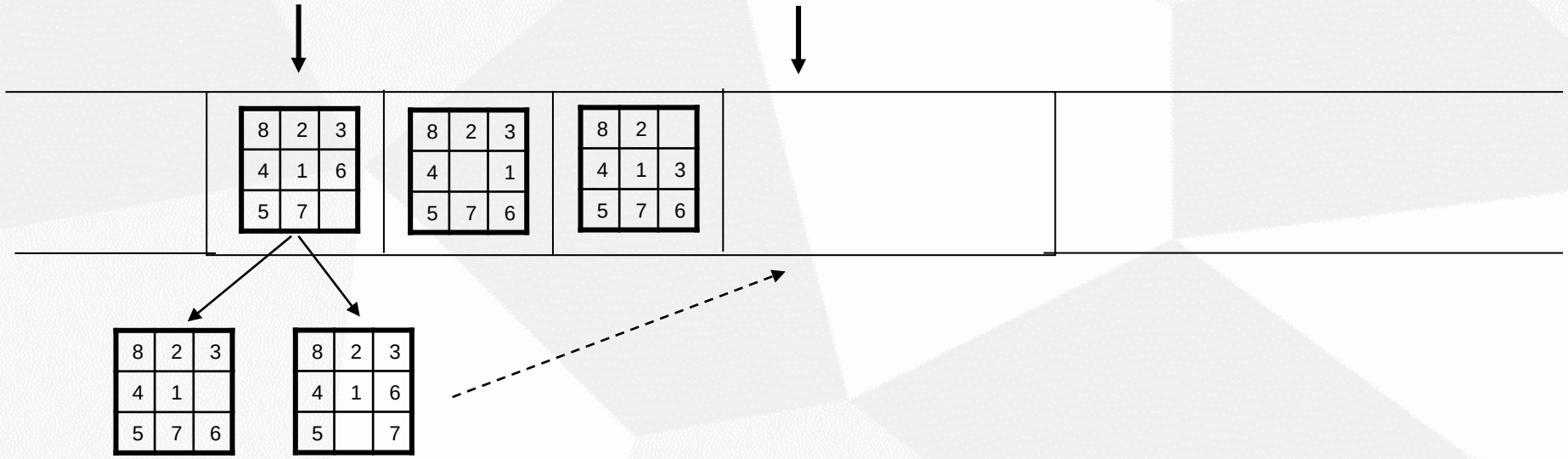
- 解题思路一
  - 广度优先搜索：优先扩展浅层结点，逐渐深入





例题1：八数码问题

- 解题思路一
  - 广度优先搜索：用队列保存待扩展的结点，从队首队取出结点，扩展出的新结点放入队尾，直到找到目标结点（问题的解）



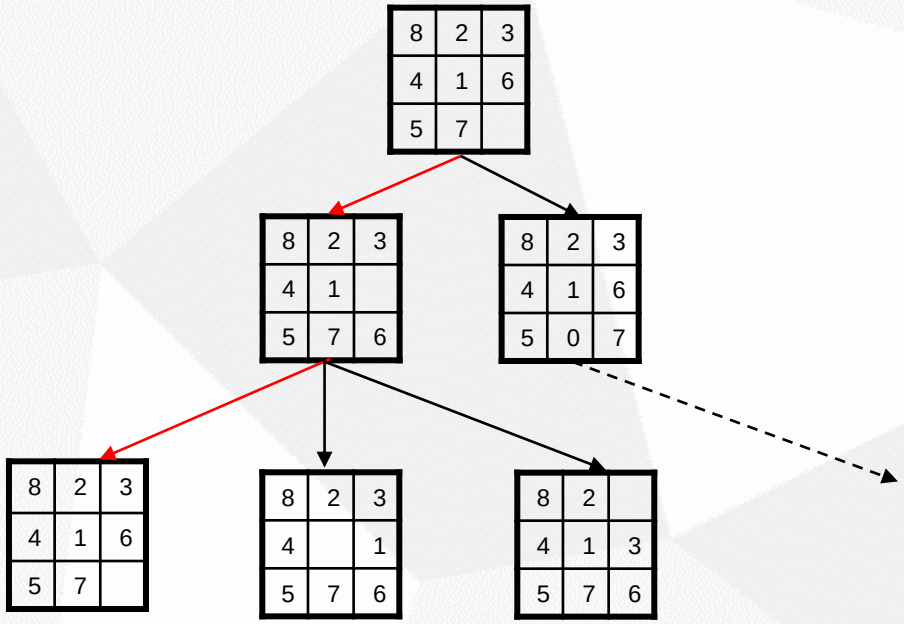
◎ 例题1：八数码问题

- 解题思路一
  - 广度优先搜索的代码框架

```
BFS()
{
        初始化队列;
        while ( 队列不为空 且 未找到目标结点 ) {
                取队首结点扩展，并将扩展出的结点放入队尾;
                必要时要记住每个结点的父结点;
        }
}
```

例题1：八数码问题

- 解题思路二
  - 深度优先搜索：优先深入遍历靠前的结点



例题1：八数码问题

- 解题思路二
  - 深度优先搜索
    - 可以用栈实现，在栈中保存从起始结点到当前结点的路径上的所有结点。
    - 实际编程中，一般用递归（系统栈）实现。

◎ 例题1：八数码问题

- 解题思路二
  - 深度优先搜索的非递归框架

```
DFS()
{
        初始化栈;
        while ( 栈不为空 且 未找到目标结点 ) {
                取栈顶结点扩展，扩展出的结点放回栈顶;
        }
        ……
}
```

◎ 例题1：八数码问题

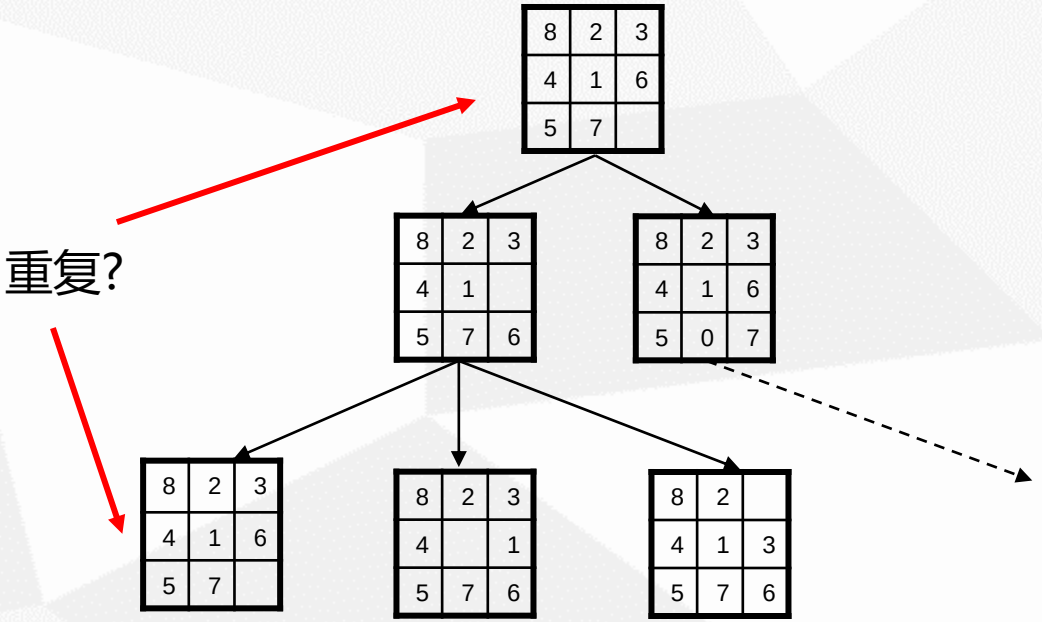
- 解题思路二
  - 深度优先搜索的递归框架：在深度优先搜索中，状态空间的图结构并不一定需要显式的存下来。
    - 此种做法需要一个全局数组array来存放每个走过的node，array[depth]就是进入DFS函数时需要扩展的节点。

```
定义全局变量 node;
DFS(depth) {
    for ( node 的每一个可行变化 ) {
        改变 node;
        DFS(depth + 1);
        恢复 node;
    }
}
```

◎ 例题1：八数码问题

- 判重（判别重复节点）
  - 新扩展出的结点如果和以前扩展出的结点相同，则则个新节点就不必再考虑。
  - 如何判重？
    - 状态数目巨大，如何存储？
    - 怎样才能较快的找到重复结点？

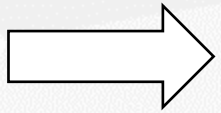
不同的编码方式所需要的存储空间会有较大差别。



◎ 例题1：八数码问题

- 方案一：合理编码，减小存储代价
  - 每个节点对应一个9进制数，即4个字节表示一个节点。
  - 此方案需要编写：字符串形式的9进制数，到其整型值（十进制数）的互相转换函数。
    - int NineToTen(char * s)
    - int TenToNine(int n, char * s)

8	2	3
4	1	6
5	7	



“823416570”

◎ 例题1：八数码问题

- 方案一：合理编码，减小存储代价
  - 判重需要一个标志位序列，每个状态对应于标志位序列中的1位，标志位为0表示该状态尚未扩展，为1则说明已经扩展过了。
  - 标志位序列可以用字符数组存放。数组的每个元素存放8个状态的标志位。位序列最多需要9^9位，因此存放位序列的数组需要：$(9^9/8 + 1)$ 个字节 = 48427562 字节。
    - 数组中一个字符8比特，表示8个状态，如：0001 1000
    - 增加1个字节是为了 $9^9/8$ 余数取整。
  - 如果某个状态对应于一个9进制数a,则其标志位就是标志位序列中的第a位（其所属的数组元素下标是a/8)

◎ 例题1：八数码问题

- 方案二：合理编码，减小存储代价
  - 为结点编号：把每个结点都看一个排列，以此排列在全部排列中的位置作为其编号。
  - 只需要一个整数(4字节)即可存下一个结点。
  - 判别重复节点用的标志数组只需要：排列总数9! =362880 字节即可。
  - 此方案比方案一节省空间，但需要编写给定排列求序号和给定序号求排列的函数。
  - 这些函数的执行速度慢于字符串形式的9进制数到其整型值的互相转换函数。

◎ 例题1：八数码问题

- 判重：时间与空间的权衡
  - 对于状态数较小的问题，可以用最直接的方式编码以空间换时间。
  - 对于状态数太大的问题，需要利用好的编码方法以时间换空间。
  - 具体问题具体分析。

◎ 例题1：八数码问题

- 用广度优先搜索（BFS）解决八数码问题

```
#include <stdio.h>

int nGoalStatus;  //目标状态
unsigned char szFlag[48427562]; //节点是否扩展的标记
char szResult[1000000];
char szMoves[1000000]; //移动步骤
int anFather[1000000]; //父节点指针
int MyQueue[1000000];  //状态队列
int nQHead, nQTail;
char sz4Moves[] = "udrl";//四种动作

int GetBit(unsigned char c, int n) {
     return (c >> n) & 1;
}
void SetBit(unsigned char & c, int n, int v) {
     if (v) c |= (1 << n);
     else c &= ~(1 << n);
}
```

例题1：八数码问题

```
// 九进制字符串转十进制
int NineToTen(char * s) {
     int nResult = 0;
     for (int i = 0; s[i]; i++) {
          nResult *= 9;
          nResult += s[i] - '0';
     }
     return nResult;
}

//十进制数转九进制字符串。
//可能有前导0，返回0的位置
int TenToNine(int n, char * s) {
     int nBase = 1;
     while (nBase <= n) {
          nBase *= 9;
     }
     nBase /= 9;
     int j = 0, nZeroPos;
     do {
          s[j] = n/nBase + '0';
          if (s[j] == '0’)
               nZeroPos = j;
          j++;

          n %= nBase;
          nBase /= 9;
     } while (nBase >= 1);
     s[j] = 0;

     //判是否要加前导0
     if (j < 9) {
          for (int i = j + 1; i > 0; i--)
               s[i] = s[i-1];
          s[0] = '0';
          return 0;
     }
     return nZeroPos;
}
```

◎ 例题1：八数码问题

```
//求从nStatus经过 cMove 移动后得到的新状态
//若移动不可行则返回-1
int NewStatus(int nStatus, char cMove) {
  char szTmp[20];
  int nZeroPos = TenToNine(nStatus, szTmp);

  switch (cMove) {
  case 'u':
    if (nZeroPos - 3 < 0)return -1;
    else {
      szTmp[nZeroPos] = szTmp[nZeroPos - 3];
      szTmp[nZeroPos - 3] = '0';
    }
    break;
  case 'd':
    if (nZeroPos + 3 > 8)return -1;
    else {
      szTmp[nZeroPos] = szTmp[nZeroPos + 3];
      szTmp[nZeroPos + 3] = '0';
    }
    break;
  case 'l':
    if (nZeroPos % 3 == 0)return -1;
    else {
      szTmp[nZeroPos] = szTmp[nZeroPos -1];
      szTmp[nZeroPos -1] = '0';
    }
    break;
   case 'r':
    if (nZeroPos % 3 == 2)return -1;
    else {
      szTmp[nZeroPos] = szTmp[nZeroPos + 1];
      szTmp[nZeroPos + 1] = '0';
    }
    break;
  }

  return NineToTen(szTmp);
}
```

◎ 例题1：八数码问题

```
int Bfs(int nStatus) {
  int nNewStatus;
  nQHead = 0; nQTail = 1;
  MyQueue[nQHead] = nStatus;
  while (nQHead != nQTail) { //队列不为空
    nStatus = MyQueue[nQHead];
    if (nStatus == nGoalStatus)
      return 1; //找到目标状态

    //尝试4种移动
    for (int i = 0; i < 4; i++) {
      nNewStatus =
         NewStatus(nStatus, sz4Moves[i]);
      if (nNewStatus == -1)
        continue; //不可移，试下一种移法

      int nByteNo = nNewStatus / 8;
      int nBitNo = nNewStatus % 8;
      //如果扩展标记已经存在，则不能入队
      if (GetBit(szFlag[nByteNo], nBitNo))
        continue;
      //设上已扩展标记
      SetBit( szFlag[nByteNo],nBitNo,1);

      //新节点入队列
      MyQueue[nQTail] = nNewStatus;

      //记录父节点
      anFather[nQTail] = nQHead;

      //记录本节点是由父节点经什么动作而来
      szMoves[nQTail] = sz4Moves[i];

      nQTail++;
    }
    nQHead++;
  }
  return 0;
}
```

◎ 例题1：八数码问题

```
int main() {
    nGoalStatus = NineToTen("123456780");
    memset(szFlag,0,sizeof(szFlag));

    char szLine[50], szLine2[20];
    fgets(szLine, 48, stdin);

    int i, j;
    //将输入的原始字符串变为九进制字符串
    for (i = 0, j = 0; szLine[i]; i++) {
      if (szLine[i] != ' ') {
        if(szLine[i] == 'x')
          szLine2[j++] = '0';
        else
          szLine2[j++] = szLine[i];
      }
    }
    szLine2[j] = 0;
```

```
  if (Bfs(NineToTen(szLine2))) {
    int nMoves = 0;
    int nPos = nQHead;
    do {
      szResult[nMoves++] = szMoves[nPos];
      nPos = anFather[nPos];
    } while (nPos);

    for (int i = nMoves -1; i >= 0; i--)
      printf("%c", szResult[i]);
  }
  else
    printf("unsolvable\n");

  return 0;
}
```

◎ 例题1：八数码问题

- 用深度优先搜索（DFS）解决八数码问题
  - 用DFS解决八数码问题不好，如用递归实现，不作特殊处理的话，很容易就导致递归层数太多而栈溢出。
  - 可以不写递归，自己用大数组实现一个栈。这可以避免栈溢出。但是可能导致输出结果的步数太多（几万步）。
  - 如果运气很坏，也可能数组会不够用。

搜索策略总结

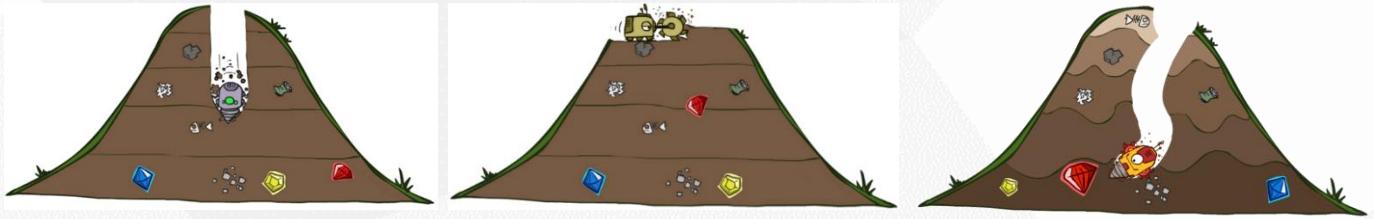
- 广度优先搜索与深度优先搜索的比较
  - BFS一般用于状态表示比较简单、求最优策略的问题。
    - 需要保存所有扩展出的状态，占用的空间大。
    - 每次扩展出结点时所走过的路径均是最短路。
  - DFS几乎可以用于任何问题
    - 只需要保存从起始状态到当前状态路径上的结点。
  - 根据题目要求，以及凭借自己的经验和对两个搜索框架的熟练程度做出选择。

搜索策略总结

- 所有的搜索算法都具有类似的框架，仅仅在节点扩展的策略上存在差异。
  - 从理论上说，所有的节点扩展策略都基于一个优先级队列（Priority Queue），即包含未扩展节点的带优先级的集合。
  - 在实际中，对于DFS和BFS来说，使用栈（Stack）和队列（Queue），可以避免一般的优先级队列带来的log(n)的开销。
  - 在编码上，可以通过使用不同的优先级队列来实现不同搜索策略的切换。

◎ 搜索策略总结

深度优先 广度优先 启发式



- 一般性的搜索框架：

```
SEARCH() {
    初始化Priority Queue;
    while ( Priority Queue不为空 且 未找到目标结点 ) {
        取Priority Queue中优先级最高的结点扩展;
        if ( 该结点未被扩展过 )
            对该结点进行扩展，并进行标记
            将扩展出的结点放回Priority Queue;
    }
}
```

◦ DFS：Priority Queue = Stack，BFS：Priority Queue = Queue

搜索策略总结

- 影响搜索效率的因素
  - 搜索对象 (枚举什么）
  - 搜索顺序（先枚举什么，后枚举什么）
  - 剪枝 (及早判断出不符合要求的情况）

◎ 例题2：木棒问题

- 问题描述（P185）
  - 乔治拿来一组等长的棍子，将它们随机地裁断（截断后的小段称为木棒），使得每一节木棒的长度都不超过50个长度单位。
  - 然后他又想把这些木棒恢复到为裁截前的状态，但忘记了棍子的初始长度。
  - 设计一个程序，帮助乔治计算棍子的可能最小长度。每一节木棒的长度都用大于零的整数表示。

例题2：木棒问题

- 输入数据
  - 由多个案例组成，每个案例包括两行。第一行是一个不超过64的整数，表示裁截之后共有多少节木棒。第二行是经过裁截后，所得到的各节木棒的长度。在最后一个案例之后，是零。
- 输出要求
  - 为每个案例，分别输出木棒的可能最小长度，每个案例占一行。
- 输入样例

```
9
5 2 1 5 2 1 5 2 1
4
1 2 3 4
0
```

- 输出样例

```
6

5
```

◎ 例题2：木棒问题

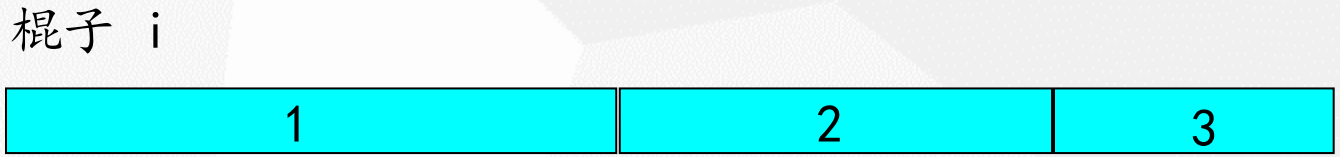
- 解题思路
  - 初始状态：有N节木棒。
  - 最终状态：
    - 这N节木棒恰好被拼接成若干根等长的棍子(裁前的东西称为棍子）。
  - 枚举什么？
    - 枚举所有可能的棍子长度。
    - 从最长的那根木棒的长度一直枚举到木棒长度总和的一半，对每个假设的棍子长度试试看能否拼齐所有棍子。
    - 在拼接过程中，要给用过的木棒做上标记，以免重复使用。
    - 拼好前i根棍子，结果发现第i+1根拼不成了，那么就要推翻第i根的拼法，重拼第i根..... 直至有可能推翻第1根棍子的拼法。

例题2：木棒问题

- 解题思路
  - 首先要解决一个问题：按什么顺序搜索？
  - 把木棒按长度排序。
  - 每次选木棒的时候都尽量先选长的。为什么？
  - 因为短木棒比较容易用来填补空缺。一根长木棒，当然比总和相同的几根短木棒要优先使用。
- 还要解决一个问题：如何剪枝
  - 就本题而言，即尽可能快地发现一根拼好的棍子需要被拆掉，以及尽量少做结果不能成功的尝试。

例题2：木棒问题

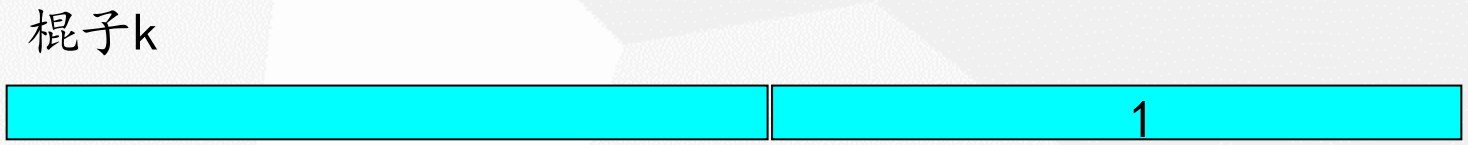
- 解题思路： 剪枝1
  - 每次开始拼第i根棍子的时候，必定选剩下的木棒里最长的一根，作为该棍子的第一根木棒。
  - 就算由于以后的拼接失败，需要重新调整第i根棍子的拼法，也不会考虑替换第i根棍子中的第一根木棒（换了也没用）。
  - 如果在此情况下怎么都无法成功，那么就要推翻第i-1根棍子的拼法。如果不存在第i-1根棍子，那么就推翻本次假设的棍子长度，尝试下一个长度。



可以考虑把2, 3换掉重拼棍子i，但是把1换掉是没有意义的

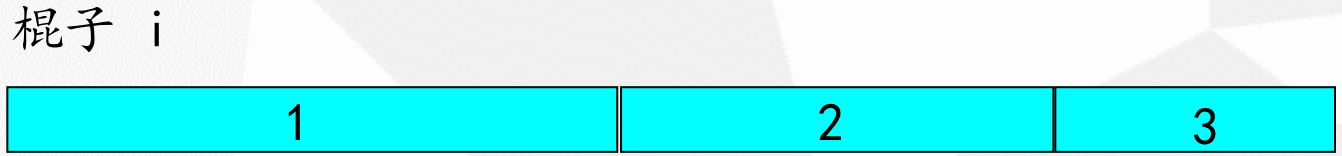
例题2：木棒问题

- 解题思路： 剪枝1
  - 为什么替换第i根棍子的第一根木棒是没用的？
  - 因为假设替换后能全部拼成功，那么这被换下来的第一根木棒，必然会出现在以后拼好的某根棍子k中。
  - 那么我们原先拼第i根棍子时，就可以用和棍子k同样的构成法来拼，照这种构成法拼好第i根棍子，继续下去最终也应该能够全部拼成功。



◎ 例题2：木棒问题

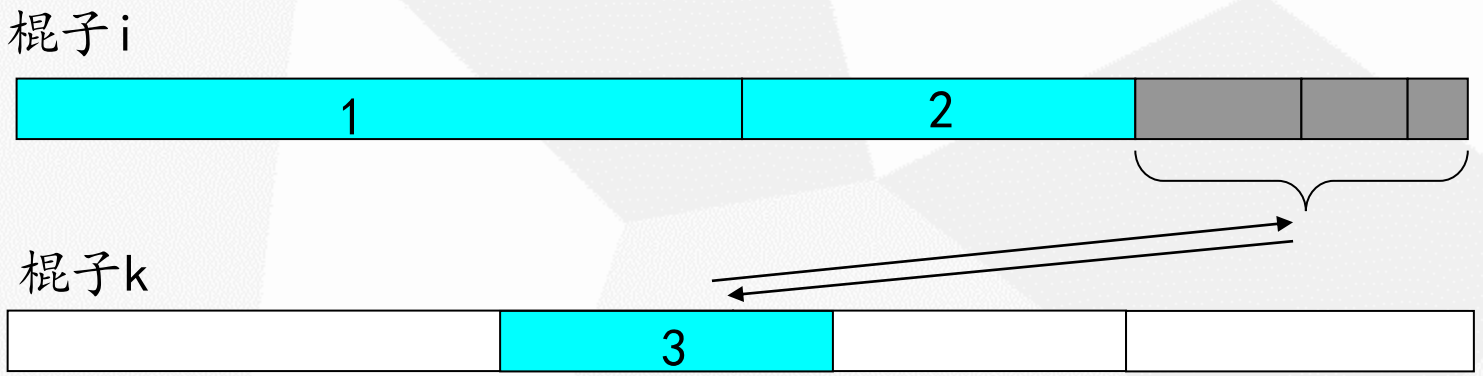
- 解题思路： 剪枝2
  - 不要希望通过仅仅替换已拼好棍子的最后一根木棒就能够改变失败的局面。
  - 假设由于后续拼接无法成功，导致准备拆除的某根棍子如下：



将 3 拆掉，留下的空用其他短木棒来填，是徒劳的

例题2：木棒问题

- 解题思路：剪枝2
  - 不要希望通过仅仅替换已拼好棍子的最后一根木棒就能够改变失败的局面。
  - 假设替换3后最终能够成功，那么3必然出现在后面的某个棍子k里。将棍子k中的3和棍子i中用来替换3的几根木棒对调，结果当然一样是成功的。这就和i原来的拼法会导致不成功矛盾。



例题2：木棒问题

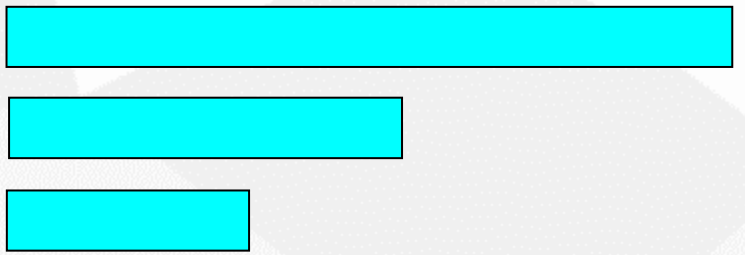
- 解题思路： 剪枝3
  - 如果某次拼接选择长度为L1的木棒，导致最终失败，则在同一位置尝试下一根木棒时，要跳过所有长度为L1的木棒。

已拼接部分　　未拼接部分

长度为L1的未拼接木棒，共有N1根

长度为L2的未拼接木棒，共有N2根

长度为L3的未拼接木棒，共有N3根



◎ 例题2：木棒问题

- 具体实现：基于深度优先搜索

```
// 函数表示: 当前有nUnusedSticks根未用木棒，而且当前正在拼的那根棍子比假定的棍子长度短了nLeft, 那么在这种情况下能全部否拼成功
int DFS(int nUnusedSticks, int nLeft ) {
    // 终止条件之一
    if (nUnusedSticks == 0 &&  nLeft == 0) return 1;
   // 基本递推关系：找一根长度不超过nLeft的木棒(假设长为len)，
   // 拼在当前棍子上，然后
    return DFS(nUnusedSticks – 1, nLeft – len);
}
```

◎ 例题2：木棒问题

```
#include <stdio.h>
#include <stdlib.h>
int T, S, L;
int anLength[65], anUsed[65];
int Dfs(int nUnusedSticks, int nLeft);
int Cmp(const void *e1, const void *e2) {
  return *((int *) e1) - *((int *) e2);
}

int main() {
  while (1) {
    scanf("%d", &S);
    if (S == 0) break;

    int nTotalLen = 0;
    for (int i = 0; i < S; i++) {
      scanf("%d", &anLength[i]);
      nTotalLen += anLength[i];
    }
    qsort(anLength, S, sizeof(int), Cmp);
    for (L = anLength[0]; L <= nTotalLen/2; L++)
    {
        if (nTotalLen % L) {
            continue;
        }

        memset(anUsed, 0, sizeof(anUsed));
        if (Dfs(S, L)) {
            printf("%d\n", L);
            break;
        }
    }

    if (L > nTotalLen / 2) {
        printf("%d\n", nTotalLen);
    }
  }
  return 0;
}
```

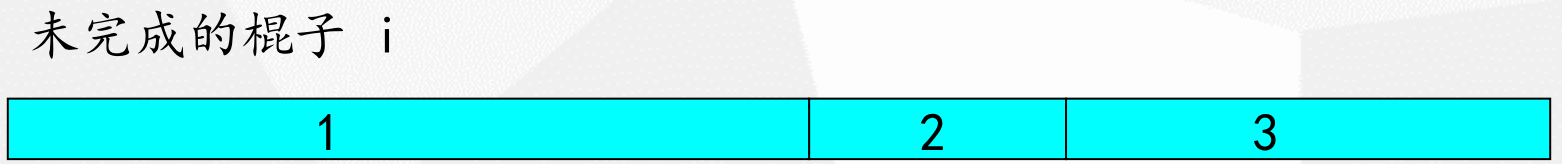
例题2：木棒问题

```
// nLeft表示当前正在拼的棍子和 L 比还缺的长度
int Dfs(int nUnusedSticks, int nLeft) {
     if (nUnusedSticks == 0 && nLeft == 0)return 1;
     if (nLeft == 0) nLeft = L;  //一根刚刚拼完，开始拼新的一根

     for (int i = 0; i < S; i++) {
          if (!anUsed[i] && anLength[i] <= nLeft) {
               if(i > 0 && anUsed[i-1] == false && anLength[i] == anLength[i-1]) {
                    continue; //剪枝3
               }
               anUsed[i] = 1;
               if (Dfs(nUnusedSticks - 1, nLeft - anLength[i]))return 1;
               else {
                    anUsed[i] = 0;//说明本次不能用第i根，第i根以后还有用
                    if (anLength[i] == nLeft || nLeft == L)return 0; //剪枝2、1
               }
          }
     }
     return 0;
}
```

例题2：木棒问题

- 解题思路： 剪枝4
  - 拼每一根棍子的时候，应该确保已经拼好的部分，长度是从长到短排列的，即拼的过程中要排除类似下面这种情况：



木棒3 比 木棒2长，这种情况的出现是一种浪费

  - 因为要是这样往下能成功，那么2, 3 对调的拼法肯定也能成功。由于取木棒是从长到短的，所以能走到这一步，就意味着当初将3放在2的位置时，是不成功的。

例题2：木棒问题

- 解题思路： 剪枝4
  - 排除办法：每次找一根木棒的时候，只要这不是一根棍子的第一条木棒，就不应该从下标为0的木棒开始找，而应该从刚刚(最近）接上去的那条木棒的下一条开始找。这样，就不会往2后面接更长的3了

  - 为此，要设置一个全局变量 nLastStickNo ，记住最近拼上去的那条木棒的下标。

例题2：木棒问题

```
// nLeft表示当前正在拼的棍子和 L 比还缺的长度
int Dfs(int nUnusedSticks, int nLeft) {
     if (nUnusedSticks == 0 && nLeft == 0) return 1;
     if (nLeft == 0) nLeft = L;  //一根刚刚拼完，开始拼新的一根

     int nStartNo = 0;
     if (nLeft != L) nStartNo = nLastStickNo + 1; //剪枝4

      for (int i = nStartNo;i < S;i ++) {
          if (!anUsed[i] && anLength[i] <= nLeft) {
               if (i > 0 && anUsed[i-1] == false && anLength[i] == anLength[i-1])
                    continue; //剪枝3
               anUsed[i] = 1;  nLastStickNo = i;
               if (Dfs(nUnusedSticks - 1, nLeft - anLength[i]))return 1;
               else { anUsed[i] = 0;//说明本次不能用第i根，第i根以后还有用
                    if (anLength[i] == nLeft || nLeft == L) return 0;//剪枝2、1
               }
          }
     }
     return 0;
}
```