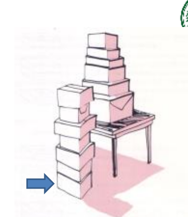
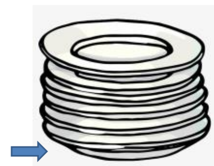
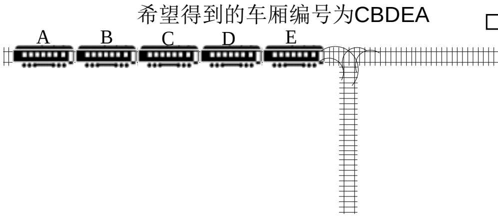
第三章 栈和队列

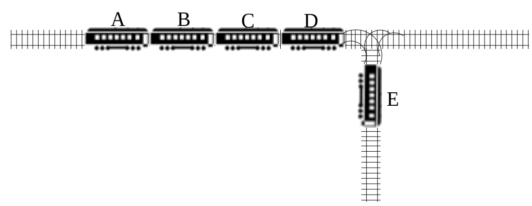





火车车厢的调整

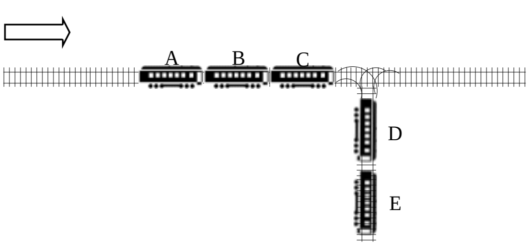
希望得到的车厢编号为CBDEA



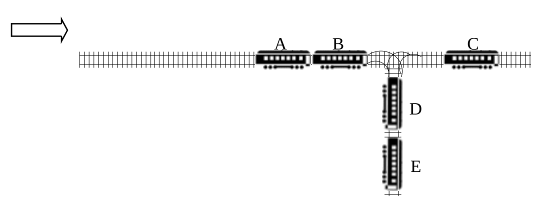
为此需要进行一些入栈和出栈的操作
CBDEA



希望得到的车厢编号为CBDEA



希望得到的车厢编号为CBDEA



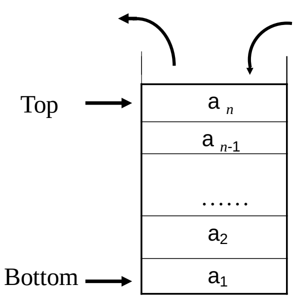
主要内容

- 定义及其基本操作
- 物理结构设计
- 应用实例

3.1 定义

栈(Stack)：是限制在表的一端进行插入和删除操作的线性表。又称为后进先出LIFO (Last In First Out)或先进后出FILO (First In Last Out)线性表。

空栈：不含任何数据元素的栈。
栈顶和栈底：
允许插入（入栈、进栈、压栈）和删除（出栈、弹栈）的一端称为栈顶，另一端称为栈底。

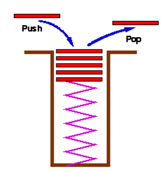


Stacks are sometimes known as last in, first out (LIFO).

基本操作

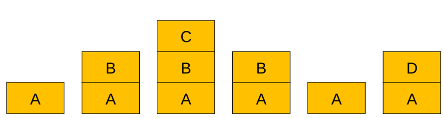
严格按照栈的定义完成的主要操作。

- 置空栈：MakeNull
- 判断栈是否为空：Empty
- 入栈（压栈）：Push
- 出栈：Pop
- 得到栈顶元素：GetTop



Status of Stack with operations

Push(A) Push(B) Push(C) Pop Pop Push(D)



ADT Stack

Data

D={a_i, a_i∈ElementType, i=1,2,…,n, n≥0}

R={<a_{i-1},a_i>|a_{i-1}, a_i∈D, i=2,…,n}

Operation

初始化，构造一个空的栈

MakeNull

操作结果：None.

判栈空

EmptyStack(*b*)

输出：*b*: Boolean.

操作结果：Return True if stack is empty, else return False.

入栈

Push(*e*)

输入：*e*: ElementType.

操作结果：Add element *e* to top of stack.

出栈

Pop(*e*)

输出：Top element *e* of stack.

初始条件：Stack is not empty.

操作结果：Remove top element from stack.

读栈顶元素

Top(*e*)

输出： Top element *e* of stack .

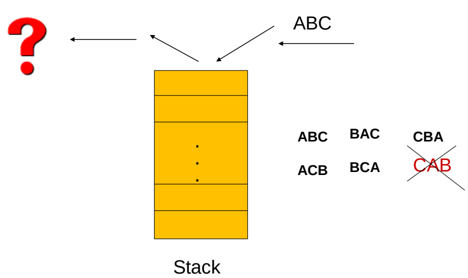
初始条件：Stack is not empty .

操作结果：Return top element from stack.

End ADT

A case

已知输入序列经过栈后到达输出序列，问得到的输出序列有哪些？



Input : ABCD

ABCD ABDC ACBD ACDB ADCB

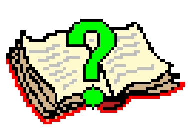
BACD BADC BCAD BCDA BDCA

CBAD CBDA CDBA

DCBA

Number: $$\frac{1}{n+1}C_{2n}^{n}$$

设输入序列是123456AB，经过栈后，可以得到的输出序列有哪些可以作为程序设计中的变量名？



3.2 栈的存储设计

3.2.1 栈的顺序存储表示

栈的顺序存储结构简称为顺序栈，和线性表相类似，用一维数组来存储栈。根据数组是否可以根据需要增大，又可分为静态顺序栈和动态顺序栈。

- 静态顺序栈实现简单，但不能根据需要增大栈的存储空间；
- 动态顺序栈可以根据需要增大栈的存储空间，但实现稍为复杂。

采用静态一维数组来存储栈。

栈底固定不变的，而栈顶则随着进栈和退栈操作变化的，

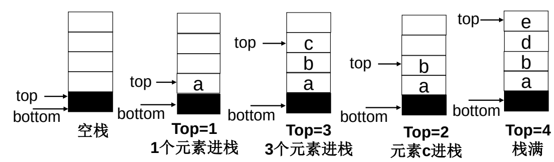
◆ 栈底固定不变的；栈顶则随着进栈和退栈操作而变化，用一个整型变量top（称为栈顶指针）来指示当前栈顶位置。

◆ 用top=0表示栈空的初始状态，每次top指向栈顶在数组中的存储位置。

◆ 结点进栈：首先执行top加1，使top指向新的栈顶位置，然后将数据元素保存到栈顶(top所指的当前位置)。

结点出栈：首先把top指向的栈顶元素取出，然后执行top减1，使top指向新的栈顶位置。

◆ 若栈的数组有Maxsize个元素，则top=Maxsize-1时栈满。下图是一个大小为5的栈的变化示意图。



静态堆栈变化示意图

栈的类型定义

```
# define  MAX_STACK_SIZE  100      /* 栈向量大小 */
# typedef  int  ElemType ;
typedef struct  sqstack
   {  ElemType   stack_array[MAX_STACK_SIZE] ;
       int  top;
   }SqStack ;
```

栈的初始化

```
SqStack Init_Stack(void)
   {    SqStack  S ;
        S.bottom=S.top=0 ; return(S) ;
   }
```


压栈(元素进栈)

```
Status push(SqStack S , ElemType  e)
   /*   使数据元素e进栈成为新的栈顶   */
  {  if  (S.top==MAX_STACK_SIZE-1)
         return  ERROR;      /* 栈满，返回错误标志    */
     S.top++ ;     /*   栈顶指针加1 */
     S.stack_array[S.top]=e  ;  /* e成为新的栈顶    */
     return OK;       /*  压栈成功   */
  }
```


弹栈(元素出栈)

```
Status  pop( SqStack   S, ElemType   *e )
   /*弹出栈顶元素*/
  {  if ( S.top==0 )
        return ERROR ;     /*  栈空，返回错误标志    */
     *e=S.stack_array[S.top] ;
     S.top-- ;
     return OK ;
  }
```

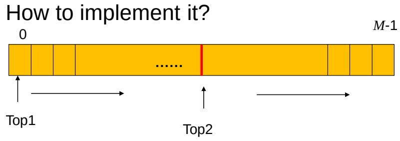

当栈满时做进栈运算必定产生空间溢出，简称“上溢”。上溢是一种出错状态，应设法避免。

当栈空时做退栈运算也将产生溢出，简称“下溢”。下溢则可能是正常现象，因为栈在使用时，其初态或终态都是空栈，所以下溢常用来作为控制转移的条件。

多个栈共享空间问题

Two stacks in a problem.

How to implement it?

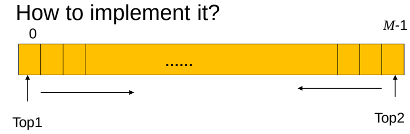


Full of stack ?

存在什么问题?

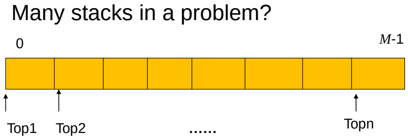
Two stacks in a problem.

How to implement it?



Full of stack：**Top1+1=Top2**

Many stacks in a problem?



存在什么问题?

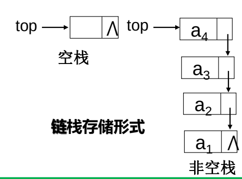
3.2.2 栈的链式存储表示

栈的链式表示

栈的链式存储结构称为链栈，是运算受限的单链表。其插入和删除操作只能在表头位置上进行。因此，链栈没有必要像单链表那样附加头结点，栈顶指针top就是链表的头指针。

链栈的结点类型说明如下：

```
typedef  struct  Stack_Node
  {  ElemType   data ;
     struct Stack_Node  *next ;
  } Stack_Node ;
```



链栈基本操作的实现

栈的初始化

```
Stack_Node  *Init_Link_Stack(void)
  {   Stack_Node  *top ;
      top=(Stack_Node  *)malloc(sizeof(Stack_Node )) ;
      top->next=NULL ;
      return(top) ;
  }
```


压栈(元素进栈)

```
Status push(Stack_Node *top , ElemType  e)
   {  Stack_Node  *p ;
      p=(Stack_Node  *)malloc(sizeof(Stack_Node)) ;
      if (!p)  return  ERROR;
        /*  申请新结点失败，返回错误标志 */
      p->data=e ;
      p->next=top->next  ;
      top->next=p ;   /*  钩链  */
      return OK;
   }
```

弹栈(元素出栈)

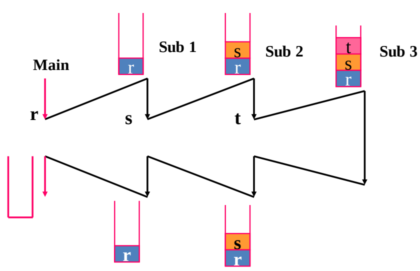
```
Status pop(Stack_Node  *top , ElemType *e)
  /*  将栈顶元素出栈  */
  {  Stack_Node  *p ;
     ElemType  e ;
     if  (top->next==NULL )
        return ERROR ;   /* 栈空，返回错误标志  */
     p=top->next ; e=p->data ;   /*  取栈顶元素  */
     top->next=p->next ;   /*  修改栈顶指针  */
     free(p) ;
     return OK ;
  }
```


顺序栈和链式栈的比较

- 时间效率
  - 顺序栈和链栈所有操作都是常数时间
- 空间效率
  - 顺序栈须说明一个固定的长度、空间浪费
  - 链栈没有栈满的问题，只有当内存没有可用空间时才会出现栈满，但是每个元素都需要一个指针域，从而产生了结构性开销

3.3 栈的应用

- 函数调用



例1. 数制转换

十进制整数*N*向其它进制数*d*(二、八、十六)的转换是计算机实现计算的基本问题。

$28_{10}=3*8+4=34_8$

$72_{10}=1*64+0*16+2*4+0=1024_4$

$53_{10}=1*32+1*16+0*8+1*4+0*2+1=110101_2$

转换法则：该转换法则对应于一个简单算法原理：

$n=(n \text{ div } d)*d+n \bmod d$

其中：div为整除运算, mod为求余运算

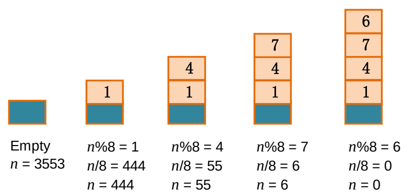
例如 $(1348)_{10}=(2504)_8$，运算过程如下：

n	*n* div 8	*n* mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

采用静态顺序栈方式实现

```
void conversion(int  n , int  d)
  /*将十进制整数N转换为d(2或8)进制数*/
  {  SqStack  S ;   int k, *e ;
     S=Init_Stack();
     while  (n>0)  {  k=n%d ; push(S , k) ; n=n/d ;  }
      /*  求出所有的余数，进栈  */
     while  (S.top!=0)    /*  栈不空时出栈，输出  */
       {   pop(S, e) ;
           printf("%1d" , *e) ;
       }
  }
```

例：十进制转换成八进制　$(3553)_{10} = (6741)_8$



例2. 括号匹配问题

在文字处理软件或编译程序设计时，常常需要检查一个字符串或一个表达式中的括号是否相匹配?

匹配思想：从左至右扫描一个字符串(或表达式)，则每个右括号将与最近遇到的那个左括号相匹配。则可以在从左至右扫描过程中把所遇到的左括号存放到堆栈中。每当遇到一个右括号时，就将它与栈顶的左括号(如果存在)相匹配，同时从栈顶删除该左括号。

示例:

2+（3*（5+9）-44/（6-3））

2+（3*（5+9-44/（6-3））

算法描述

```
#define TRUE  0
#define FLASE  -1
SqStack  S ;
S=Init_Stack() ; /*堆栈初始化*/
int Match_Brackets( )
   {   char ch , x ;
       scanf(“%c” , &ch) ;
        while (asc(ch)!=13)
```



```
int Match_Brackets( )
   {   char ch , x ;
        scanf(“%c” , &ch) ;
         while (asc(ch)!=13)
        {   if  ((ch==‘(’)||(ch==‘[’))  push(S , ch) ;
                 else if  (ch==‘]’)
                    {  x=pop(S) ;
                       if (x!=‘[’)
                           {  printf(“’[’括号不匹配” ) ;
                              return FLASE  ;  }  }
                 else if  (ch==‘)’)
                     {  x=pop(S) ;
                        if (x!=‘(’)
                          { printf(“’(’括号不匹配” ) ;
                            return FLASE  ;}
                     }
            }
            if  (S.top!=0)
                {   printf(“括号数量不匹配！ ” ) ;
                    return FLASE  ;
                }
            else  return TRUE  ;
   }
```


例 3. 表达式处理

表达式是由操作数（operand）、运算符（operator）和界限符（delimiter）组成的。

常用算法是算符优先法，就是根据运算符和界限符的优先次序的规定来实现表达式求值的。

算术四则运算规则——运算符的优先次序规定：

1)先乘除，后加减；

2)从左算到右；

3)先括号内，后括号外。

算术表达式的三种表示

名称	别名	格式	例子
中缀(infix)表示		<操作数> <操作符> <操作数>	A+B
前缀(prefix)表示	波兰式表示	<操作符> <操作数> <操作数>	+AB
后缀(postfix)表示	逆波兰式表示	<操作数> <操作数> <操作符>	AB+

例如：

$$(a+b)*(a-b)\begin{cases} *+ab-ab \\ (a+b)*(a-b) \\ ab+ab-* \end{cases}$$

1) 中缀表达式求值

4+2*3-10/5

51*(24-15/3)+6

建立一个算符优先表

根据算术四则运算的三条规则，表中给出了任意两个相继出现的算符θ_1和θ_2之间的优先关系。

前面的 θ_1 和后面的 θ_2

θ_1 \ θ_2	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

中缀表达式求值算法

- 1：操作数栈置空，操作符栈压入算符“#”
- 2：依次读入表达式的每个单词
- 3：如果是操作数，压入操作数栈
- 4：如果是操作符，将操作符栈顶元素 θ_1与读入的操作符 θ_2进行优先级比较
  - 4.1如果栈顶元素优先级低，将 θ_2压入操作符栈
  - 4.2如果相等，弹操作符栈
  - 4.3如果栈顶元素优先级高，弹出两个操作数，一个运算符，进行计算，并将计算结果压入操作数栈，重复第4步的判断
- 5:直至整个表达式处理完毕

- 3*(7-2)#

步骤	操作符栈	操作数栈	输入字符	操作
1	#		3*(7-2)#	压入“3”
2	#	3	*(7-2)#	压入“*”
3	#*	3	(7-2)#	压入“(”
4	#*(	3	7-2)#	压入“7”
5	#*(	37	-2)#	压入“-”
6	#*(-	37	2)#	压入“2”
7	#*(-	372	)#	弹出“-”压入7-2
8	#*(	35	)#	弹出“(”
9	#*	35	#	计算3*5
10	#	15	#	操作符栈空，结束


```
{  SOPTR.Push(”#”);
   cin >>c;
   while (c!’#’ ||SOPTR.top()!=‘#’)
   {     if (!In(c,OP)) { SOPND.Push(c);
                            cin >>c; }
         else
             switch (Precede(SOPTR.top(),c))
             {    case ‘<‘:    SOPTR.Push(,c);
                               cin >>c;
                               break;
                  case ‘=‘:    SOPTR.Pop(x);
                               cin >>c;
                               Break;
                  case ‘>’:    SOPTR.Pop(theta);
                               SOPND.Pop(b);
                               SOPND.Pop(a);
                               SOPND.Push(Operate(a,theta,b));
                               break;
             }
   }
   return SOPND.top();    }
```

```
d=c;
cin >>c;
While ((c!=‘#’)&&(In(c,OP)))
 {
   d=d*10+c;
   cin >>c;
 }
SOPND.Push(d);
```


2) 后缀表达式求值

4+3*5	4, 3, 5*+
2*(5+9*4/2)+6*5	2, 5, 9, 4*2/+*6, 5*+

求解算法：

设定一个操作数栈OPND；从左向右依次读入，当读到的是运算数，将其加入到运算数栈中；若读入的是运算符，从运算数栈取出两个元素，与读入的运算符进行运算，将运算结果加入到运算数栈。直到表达式的最后一个运算符处理完毕。

3) 中缀表达式转换成后缀表达式

中缀	后缀
4+3*5	4, 3, 5*+
2*(5+9*4/2)+6*5	2, 5, 9, 4*2/+*6, 5*+

中缀表达式中，运算符的出现次序与计算顺序不一致；

后缀表达式中，运算符的出现次序就是计算次序。

A+B*C-D#　　ABC*+D-

读到的符号	运算符栈	输出序列
A	#	A
+	#+	A
B	#+	AB
*	#+*	AB
C	#+*	ABC
-	#-	ABC*+
D	#-	ABC*+D
#	#	ABC*+D-

算　法

- 1：操作符栈压入算符“#”
- 2：依次读入表达式的每个单词
- 3：如果是操作数，则输出
- 4：如果是操作符，将操作符栈顶元素 θ_1 与读入的操作符 θ_2 进行优先级比较
  - 4.1如果栈顶元素优先级低，将 θ_2 压入操作符栈
  - 4.2如果相等，弹操作符栈
  - 4.3如果栈顶元素优先级高，弹出栈定元素并输出，重复第4步的判断
- 5:直至整个表达式处理完毕

3.4 递归

递归是算法设计中一种重要的方法，可以使许多程序结构简化，易理解，易证明。

递归定义的算法有两个部分：

递归基：直接定义最简单情况下的函数值；

递归步：通过较为简单情况下的函数值定义一般情况下的函数值。

递归算法设计

递归算法既是一种有效的算法设计方法，也是一种有效的分析问题的方法。递归算法求解问题的基本思想是：对于一个较为复杂的问题，把原问题分解成若干个相对简单且类同的子问题,这样，原问题就可递推得到解。

适宜于用递归算法求解的问题的充分必要条件是：

(1) 问题具有某种可借用的类同自身的子问题描述的性质；

(2) 某一有限步的子问题(也称作本原问题)有直接的解存在。

当一个问题存在上述两个基本要素时，该问题的递归算法的设计方法是：

(1) 把对原问题的求解设计成包含有对子问题求解的形式。

(2) 设计递归出口。

递归算法分类

- 单路递归（一个递归过程中只有一个递归入口）
- 多路递归（一个递归过程中有多个入口）
- 间接递归（函数可通过其他函数间接调用自己）
- 迭代递归（每次递归调用都包含一次循环递归）

3.4.1递归算法举例

例1. 阶乘函数

$$n! = n*(n-1)*(n-2)*......*1$$

$$n! = n*(n-1)!$$

$$f(n)=\begin{cases}1 & n=0\\ n*f(n-1) & n>0\end{cases}$$

递归基

递归步

算法描述

```
Long factorial (long n)
 {
   int temp;
   if (n==0) return 1;
    else
      {
         temp=n*factorial(n-1);

          return temp;
       }
  }
```


例2. Fibonacci数列

无穷数列1，1，2，3，5，8，13，21，34，55，……，称为Fibonacci数列。

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

递归基

递归步

第n个Fibonacci数可递归地计算如下：

```
int fibonacci(int n)
  {
     if (n <= 1) return 1;
     return fibonacci(n-1)+fibonacci(n-2);
  }
```


例3. Ackerman函数

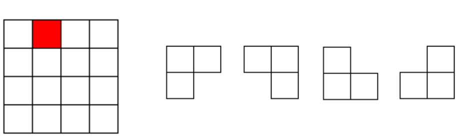
当一个函数及它的一个变量是由函数自身定义时，称这个函数是**双递归函数**。

Ackerman函数$A(n, m)$定义如下：

$$\begin{cases}A(1,0)=2 & \\ A(0,m)=1 & m\ge 0\\ A(n,0)=n+2 & n\ge 2\\ A(n,m)=A(A(n-1,m),m-1) & n,m\ge 1\end{cases}$$

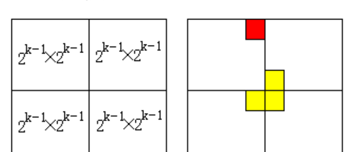
例 4 棋盘覆盖

在一个$2^k\times 2^k$ 个方格组成的棋盘中，恰有一个方格与其它方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。



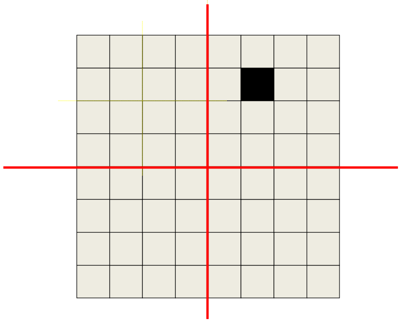
划分：

当k>0时，将$2^k\times 2^k$棋盘分割为4个$2^{k-1}\times 2^{k-1}$ 子棋盘(a)所示。特殊方格必位于4个较小子棋盘之一中，其余3个子棋盘中无特殊方格。为了将这3个无特殊方格的子棋盘转化为特殊棋盘，可以用一个L型骨牌覆盖这3个较小棋盘的会合处，如 (b)所示，从而将原问题转化为4个较小规模的棋盘覆盖问题。递归地使用这种分割，直至棋盘简化为棋盘1×1。

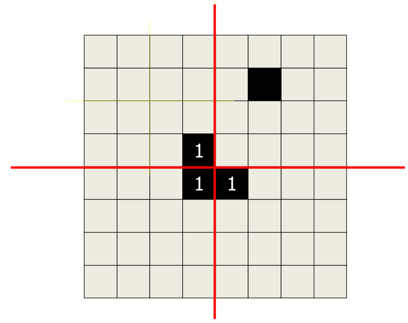


(a) (b)

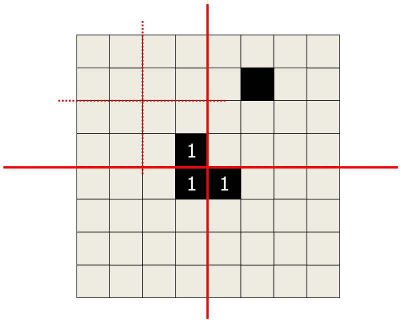
- $2^3 \times 2^3$棋盘的部分结果



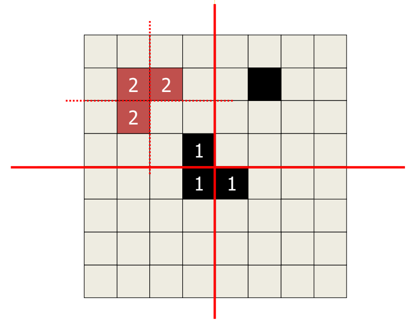
- $2^3 \times 2^3$棋盘的部分结果



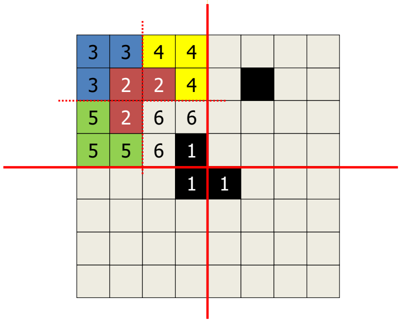
- $2^3 \times 2^3$棋盘的部分结果



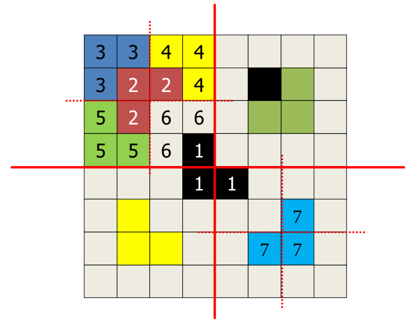
- $2^3 \times 2^3$棋盘的部分结果



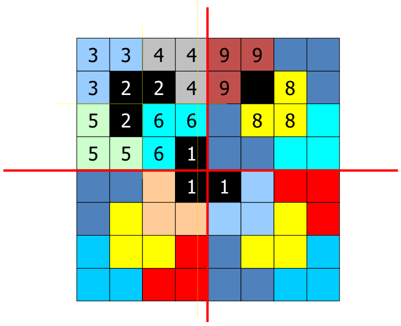
- $2^3 \times 2^3$棋盘的部分结果



- $2^3 \times 2^3$棋盘的部分结果

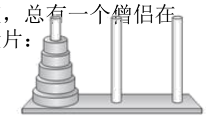


- $2^3 \times 2^3$棋盘的部分结果



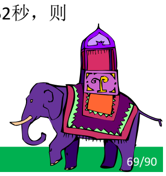
相传...

- 印度北部贝拿勒斯圣庙里
  - 一块黄铜板上插着三根宝石针，在其中一根针上从下到上地穿好了由大到小的64片金片，这就是汉诺塔。不论白天黑夜，总有一个僧侣在按照下面的法则移动这些金片：
    - 一次只移动一片
    - 不管在哪根针上
    - 小片必须在大片上面
  - 僧侣们预言，当所有的金片都从梵天穿好的那根针上移到另外一根针上时，世界就将在一声霹雳中消灭，而梵塔、庙宇和众生也都将同归于尽。



假如这个传说是真的！

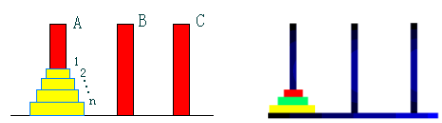
- 多少年？
  - 假设有n片，移动次数是$f(n)$。
    - 显然f(1)=1，f(2)=3，f(3)=7，且f(k+1)=2*f(k)+1。
    - 即f(n)=2^n-1。
  - n=64时
    - f(64)= 2^64-1=18446744073709551615
    - 假如每秒钟一次，平均每年31556952秒，则 18446744073709551615/31556952 =584554049253.855年 =**5845亿年以上**



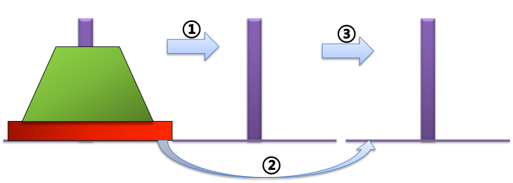
例5. Hanoi塔问题

设A,B,C是3个塔座。开始时，在塔座A上有一叠共n个圆盘，这些圆盘自下而上，由大到小地叠在一起。各圆盘从小到大编号为1,2,…,n,现要求将塔座A上的这一叠圆盘移到塔座B上，并仍按同样顺序叠置。在移动圆盘时应遵守以下移动规则：

- 规则1：每次只能移动1个圆盘；
- 规则2：任何时刻都不允许将较大的圆盘压在较小的圆盘之上；
- 规则3：在满足移动规则1和2的前提下，可将圆盘移至A,B,C中任一塔座上。



- 解法
  - 为了把n>1个盘子从木桩1移到木桩3，需要把n-1个盘子递归的移到木桩2（借助3）；
  - 然后把最大的盘子（第n个）移到木桩3；
  - 再把n-1个盘子由木桩2移动到木桩3（借助1）。



如何实现移动圆盘的操作呢？

（1）递归中止条件：当n=1时，问题比较简单，只要将编号为1的圆盘从塔座A直接移动到塔座C上即可；

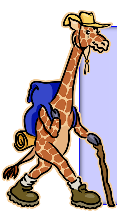
（2）问题分解：当n>1时， 需利用塔座B作辅助塔座，若能设法将压在编号为n的圆盘上的n-1个圆盘从塔座A(依照上述原则)移至塔座B上， 则可先将编号为n的圆盘从塔座A 移至塔座C上，然后再将塔座B上的n-1个圆盘(依照上述原则)移至塔座C上。而如何将n-1个圆盘从一个塔座移至另一个塔座问题是一个和原问题具有相同特征属性的问题，只是问题的规模小个1，因此可以用同样方法求解。 由此可得如下算法所示的求解n阶Hanoi塔问题的函数。

```
void hanoi(int n,char x,char y,char z) /*将塔座X上按直径由小到大且至上而
    下编号为1至n的n个圆盘按规则搬到塔座Z上， Y可用作辅助塔座 */
{
  if(n==1)
     move(x,1,z); /* 将编号为1的圆盘从X移动Z */
  else {
    hanoi(n-1,x,z,y); /* 将X上编号为1至n-1的圆盘移到Y,Z作辅助塔 */
    move(x,n,z); /* 将编号为n的圆盘从X移到Z */
    hanoi(n-1,y,x,z); /* 将Y上编号为1至n-1的圆盘移动到Z， X作辅助塔 */
     }
}
```


下面给出三个盘子搬动时hanoi(3, A, B, C) 递归调用过程.

```
hanoi(2,A,C,B):
        hanoi(1,A,B,C)  move(A->C)     1号搬到C
        move(A->B)                     2号搬到B
        hanoi(1,C,A,B)  move(C->B)     1号搬到B
move(A->C)                             3号搬到C
hanoi(2,B,A,C):
        hanoi(1,B,C,A) move(B->A)      1号搬到A
        move(B->c)   2号搬到C
        hanoi(1,A,B,C) move(A->C)      1号搬到C
```

```
hanoi(n, x, y, z)
{
  if(n==1)
      move(x,1,z);
   else {
      hanoi(n-
   1,x,z,y);
      move(x,n,z);
      hanoi(n-1,y,x,z);
   }
}
```

总之，

递归式一种优雅的算法。

但我们应该谨慎使用递归算法，因为它们的简洁可能会掩盖它们的低效率。

采用一个用户定义的栈来模拟系统的递归调用工作栈。该方法通用性强，但本质上还是递归，只不过人工做了本来由编译器做的事情，优化效果不明显。

3.4.2 栈与递归

对于非递归函数，调用函数在调用被调用函数前，系统要保存以下两类信息：

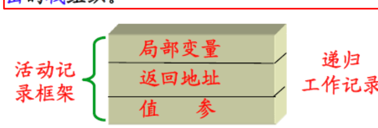
（1）调用函数的返回地址；

（2）调用函数的局部变量值。

当执行完被调用函数，返回调用函数前，系统首先要恢复调用函数的局部变量值，然后返回调用函数的返回地址。

递归函数被调用时，系统要作的工作和非递归函数被调用时系统要作的工作在形式上类同，但保存信息的方法不同。

递归函数被调用时，系统需要一个运行时栈。系统的运行时栈也要保存上述两类信息。每一层递归调用所需保存的信息构成运行时栈的一个工作记录，在每进入下一层递归调用时，系统就建立一个新的工作记录，并把这个工作记录进栈成为运行时栈新的栈顶；每返回一层递归调用，就退栈一个工作记录。因为栈顶的工作记录必定是当前正在运行的递归函数的工作记录，所以栈顶的工作记录也称为活动记录。

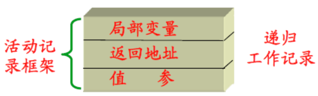
- 每一次递归调用时，需要为过程中使用的参数、局部变量和返回地址等另外分配存储空间。
- 每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织。
- 

◆ 递归函数的内部执行过程

■(1) 运行开始时，首先为递归调用建立一个工作栈，其结构包括值参、局部变量和返回地址；

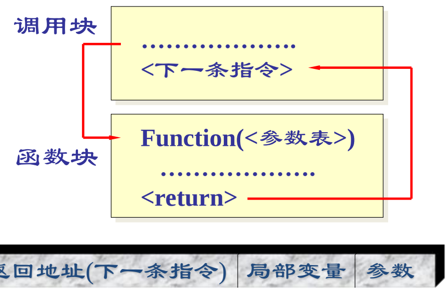
■(2) 每次执行递归调用之前，把递归函数的值参和局部变量的当前值以及调用后的返回地址压栈；

■(3) 每次递归调用结束后，将栈顶元素出栈，使相应的值参和局部变量恢复为调用前的值，然后转向返回地址指定的位置继续执行。



工作记录的含义？意义？

函数递归时的活动记录



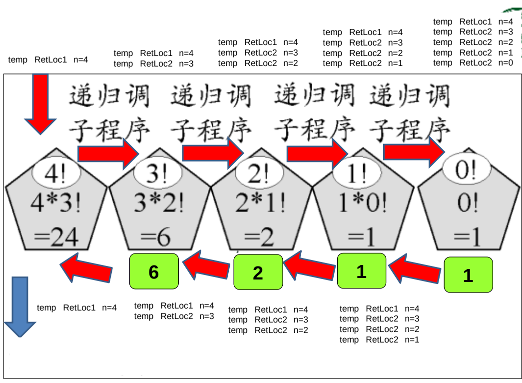

```
     long Factorial ( long n ) {
        int temp;
        if ( n == 0 ) return 1;
        else temp = n * Factorial (n-1);
RetLoc2
        return temp;
     }

     void main ( ) {
        int n;
        n = Factorial (4);
RetLoc1
     }
```


计算Fact时活动记录的内容

递归调用序列

参数	返回地址	返回时的指令
4	RetLoc1	RetLoc1 return 4*6 //返回24
3	RetLoc2	RetLoc2 return 3*2 //返回6
2	RetLoc2	RetLoc2 return 2*1 //返回2
1	RetLoc2	RetLoc2 return 1*1 //返回1
0	RetLoc2	RetLoc2 return 1 //返回1



规范化的递归转换成非递归的方法

栈的定义：

调用参数区

$p_1, p_2, \ldots, p_m$

局部参数区

$m_1, m_2, \ldots, m_s$

断点地址*RT*

```
Long factorial(long n)
 {
   int temp;
   if (n==0)
       return 1;
    else
      { temp=factorial(n-1)*n;

         return temp;
      }
 }
```

断点地址的描述

按语句标号来记录，含有*t*个调用递归过程本身的语句，则设定*t*+2个语句标号，L0设在第一个可执行的语句上，

Li(*i*=1..*t*)设在*t*个递归调用的语句的返回处，

Lj(j=*t*+1)设在过程体结束的语句上。

```
Long factorial(long n)
 {
   int temp;
  L0: if (n==0)
       return 1;
    else
     {L1: temp=factorial(n-1)*n;

         return temp;
      }
 L2:
}
```


断点地址的标注

按语句标号来记录，含有*t*个调用递归过程本身的语句，则设定*t*+2个语句标号，L0设在第一个可执行的语句上，

Li(*i*=1..*t*)设在*t*个递归调用的语句的返回处，

Lj(j=*t*+1)设在过程体结束的语句上。

```
void hanoi(int n, char X, char Y, char Z)
{
 L0: if (n <= 1)
     move(X,Z);
 else
 {
// 最大的盘在X上不动，把X上的n－1个盘移到Y
 hanoi(n-1,X,Z,Y);
 L1: move(X,Z); //移动最大盘到Z，放好
 hanoi(n-1,Y,X,Z);//把 Y上的n－1个盘移到Z
   L2: }
L3:
}
```


规则一

在标号L0前增加语句：

S.Push(L(t+1), p_1, p_2, …, p_m, q_1, q_2, …, q_n, m_1, m_2, …, m_s)

此后，所用的变量均由栈定元素指示。

```
void hanoi(int n, char X, char Y, char Z)
{
 L0: if (n <= 1)
     move(X,Z);
 else
 {
// 最大的盘在X上不动，把X上的n－1个环移到Y
 hanoi(n-1,X,Z,Y);
 L1: move(X,Z); //移动最大盘到Z，放好
 hanoi(n-1,Y,X,Z);//把 Y上的n－1个盘移到Z
 L2:
}
L3: }
```

S.Push(L3,n,x,y,z)

If (S.top(n)<=1)
Move(S.top(X),S.top(Z))

规则二

将调用递归的语句Pi(a_1,a_2,…$a_{(m+n)}$) 改为：

S.Push(Li, a_1,a_2,…$a_{(m+n)}$)；

goto L0;

Li: (v_1,v_2,…,$v_{(m+n)}$)=S.Pop();

```
void hanoi(int n, char X, char Y, char Z)
{
 L0: if (n <= 1)
     move(X,Z);
 else
 {
// 最大的盘在X上不动，把X上的n－1个盘移到Y
 hanoi(n-1,X,Z,Y);
 L1: move(X,Z); //移动最大盘到Z，放好
 hanoi(n-1,Y,X,Z);//把 Y上的n－1个盘移到Z
   L2: }
L3:
}
```

S.Push(L1,n-1,x,z,y);
Goto L0;
L1: (v1,v2,v3,v4)=S.Pop();

规则三

在所有的递归出口处，增加语句 goto RT=S.Top(RT)

在最后一个标号处，是最终的函数值出栈语句：
(v_1,v_2,…,$v_{(m+n)}$)=S.Pop();

规则三

```
void hanoi(int n, char X, char Y, char Z)
{
 L0: if (n <= 1)
     move(X,Z);
 else
 {
// 最大的盘在X上不动，把X上的n－1个盘移到Y
 hanoi(n-1,X,Z,Y);
 L1: move(X,Z); //移动最大盘到Z，放好
 hanoi(n-1,Y,X,Z);//把 Y上的n－1个盘移到Z
   L2: }
L3:
}
```

goto RT=S.Top(RT)

goto RT=S.Top(RT)

(v_1,v_2,…,$v_{(m+n)}$)=S.Pop();