计算机导论与程序设计 [CS006001-60]

段江涛

机电工程学院

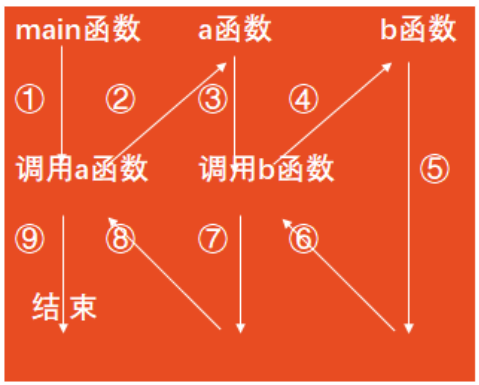
2019 年 11 月

lecture-14 主要内容

用函数实现模块化程序设计 (嵌套与递归)

函数的嵌套调用

```c
#include<stdio.h>
int a(); int b(); // 函数声明
int main()
{
    int c; // 与a()中的c无关
    c=a(); // 函数调用
    return 0;
}
```




```c
int a()
{
    int c; // 与main()中的c无关
    ...;
    c=b();// 函数调用
    ...;
    return c;
}
int b()
{
    ...;
    return 10;
}
```

例: 求 4 个整数中的最大者。

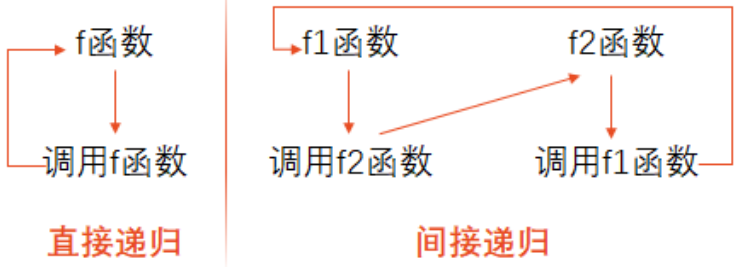
```
#include<stdio.h>
int max2(int x,int y);
int max4(int a,int b,int c,int d);
int main() // 主函数
{
   int a=b,c,d;
   scanf("%d%d%d%d",&a,&b,&c,&d);
   printf("较大者=%d\n", max4(a,b,c,
    d));
   printf("较大者=%d\n", max2(max2(a
    ,b),max2(c,d))); // 等效
   return 0;
}
```

```
// 定义函数
int max2(int x,int y) // 形式
    参数
{
   int z;
   z=x>y ? x : y;
   return z;
}
int max4(int a,int b,int c,
    int d)
{
  return max2(max2(a,b),max2
    (c,d));
}
```

函数的递归调用

在调用一个函数的过程中又出现直接或间接地调用该函数本身，称为函数的递归调用。

```
// 递推公式: f(0)=0,f(1)=1,
//       n>1: f(n)=n+f(n-1)
int f(int n)
{
    int sum;
    if(n==0||n==1) sum=n;
    else sum=n+f(n-1);
    return sum;
}
```

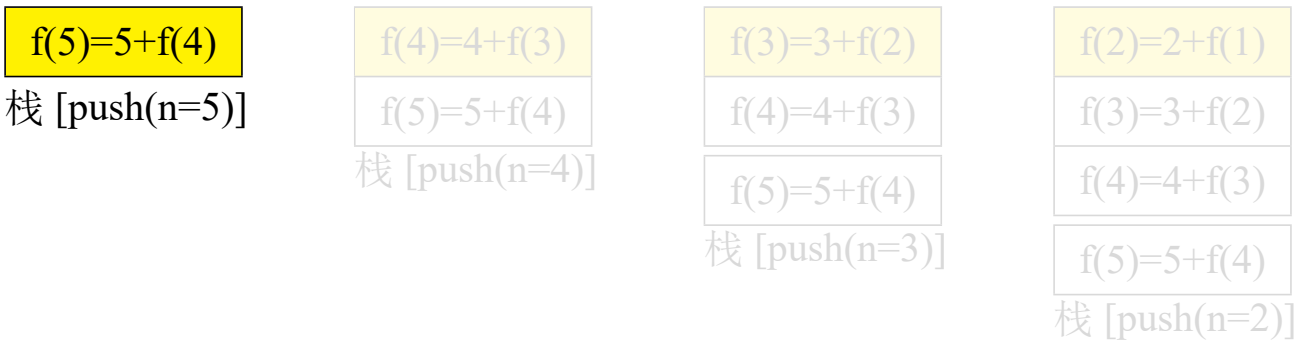


程序中不应出现无终止的递归调用，而只应出现**有限次数的，有终止的递归调用**，这可以用 if 语句来控制，只有在某一条件成立时才继续执行递归调用；否则就不再继续。

递归调用过程分析 (push)

```
// 递推公式: f(0)=0,f(1)=1, n>1: f(n)=n+f(n-1)
int f(int n)
{
  int sum;
  if(n==0||n==1) sum=n;
  else sum=n+f(n-1); // 未完成的计算用"栈"存储起来(push)
  return sum; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

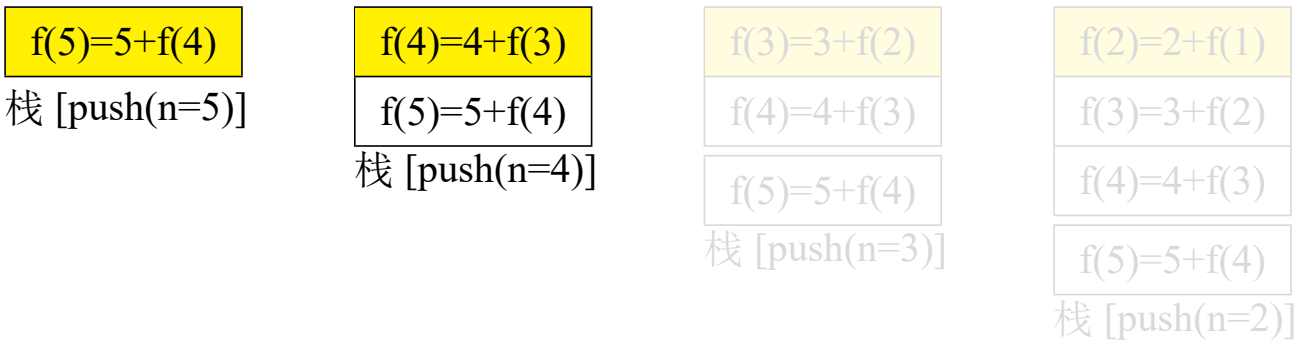
系统内部自动维护一个称作 “栈” 的存储数据的空间,栈是一种 “先进后出 (FILO)” 的数据结构。向栈中存储数据操作称作 push, 取出栈顶数据操作称作 pop。第一个 push 的数据,最后一个被 pop.



递归调用过程分析 (push)

```
// 递推公式: f(0)=0,f(1)=1, n>1: f(n)=n+f(n-1)
int f(int n)
{
  int sum;
  if(n==0||n==1) sum=n;
  else sum=n+f(n-1); // 未完成的计算用"栈"存储起来(push)
  return sum; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

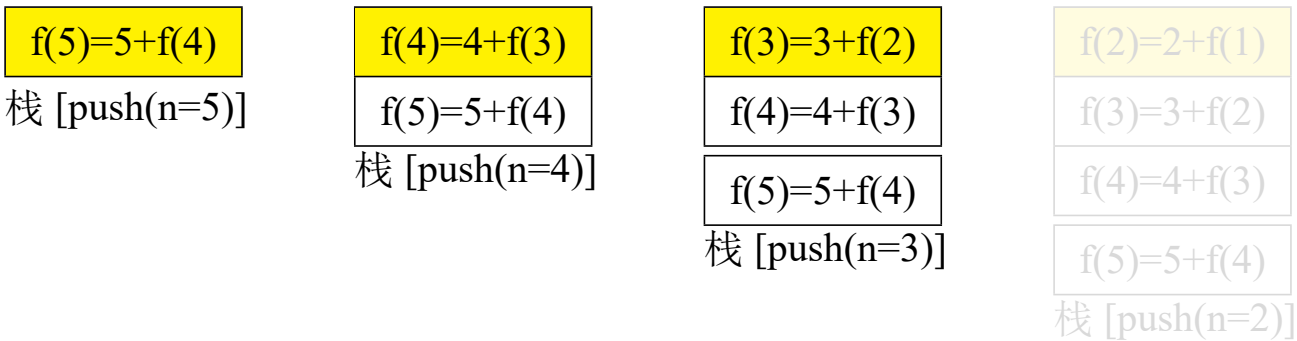
系统内部自动维护一个称作 “栈” 的存储数据的空间,栈是一种 “先进后出 (FILO)” 的数据结构。向栈中存储数据操作称作 push, 取出栈顶数据操作称作 pop。第一个 push 的数据,最后一个被 pop.



递归调用过程分析 (push)

```
// 递推公式: f(0)=0,f(1)=1, n>1: f(n)=n+f(n-1)
int f(int n)
{
  int sum;
  if(n==0||n==1) sum=n;
  else sum=n+f(n-1); // 未完成的计算用"栈"存储起来(push)
  return sum; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

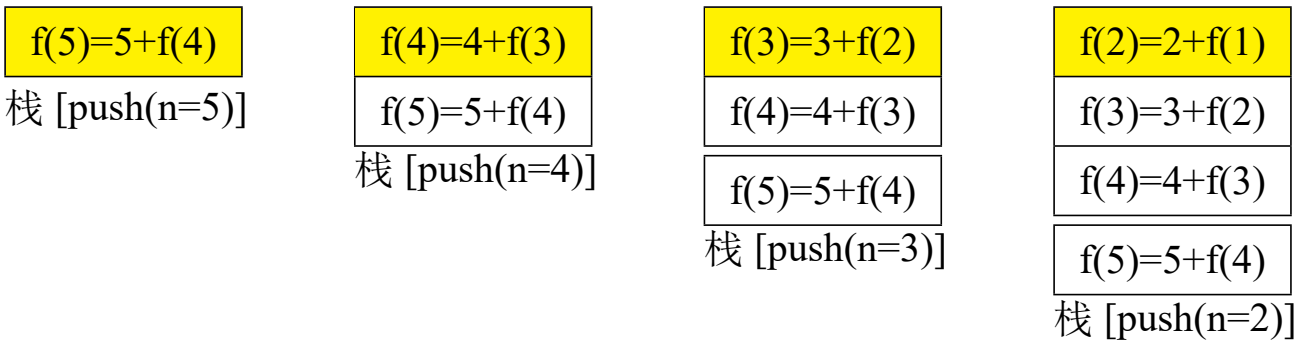
系统内部自动维护一个称作 “栈” 的存储数据的空间,栈是一种 “先进后出 (FILO)” 的数据结构。向栈中存储数据操作称作 push, 取出栈顶数据操作称作 pop。第一个 push 的数据,最后一个被 pop.



递归调用过程分析 (push)

```
// 递推公式: f(0)=0,f(1)=1, n>1: f(n)=n+f(n-1)
int f(int n)
{
  int sum;
  if(n==0||n==1) sum=n;
  else sum=n+f(n-1); // 未完成的计算用"栈"存储起来(push)
  return sum; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

系统内部自动维护一个称作 “栈” 的存储数据的空间,栈是一种 “先进后出 (FILO)” 的数据结构。向栈中存储数据操作称作 push, 取出栈顶数据操作称作 pop。第一个 push 的数据,最后一个被 pop.



递归调用过程分析 (pop)

```
// 递推公式: f(0)=0,f(1)=1, n>1: f(n)=n+f(n-1)
int f(int n)
{
  int sum;
  if(n==0||n==1) sum=n;
  else sum=n+f(n-1); // 未完成的计算用"栈"存储起来(push)
  return sum; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

系统内部自动维护一个称作 “栈” 的存储数据的空间,栈是一种 “先进后出 (FILO)” 的数据结构。向栈中存储数据操作称作 push, 取出数据操作称作 pop。第一个 push 的数据,最后一个被 pop.

f(5)=5+f(4)

栈 [pop(n=5)]

f(5)=5+f(4)=15

f(4)=4+f(3)

f(5)=5+f(4)

栈 [pop(n=4)]

f(4)=4+f(3)=10

f(3)=3+f(2)

f(4)=4+f(3)

f(5)=5+f(4)

栈 [pop(n=3)]

f(3)=3+f(2)=6

f(2)=2+f(1)

f(3)=3+f(2)

f(4)=4+f(3)

f(5)=5+f(4)

栈 [pop(n=2)]

f(2)=2+f(1)=3

递归调用过程分析 (pop)

```
// 递推公式: f(0)=0,f(1)=1, n>1: f(n)=n+f(n-1)
int f(int n)
{
  int sum;
  if(n==0||n==1) sum=n;
  else sum=n+f(n-1); // 未完成的计算用"栈"存储起来(push)
  return sum; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

系统内部自动维护一个称作 “栈” 的存储数据的空间,栈是一种 “先进后出 (FILO)” 的数据结构。向栈中存储数据操作称作 push, 取出数据操作称作 pop。第一个 push 的数据,最后一个被 pop.

f(5)=5+f(4)

栈 [pop(n=5)]

f(5)=5+f(4)=15

f(4)=4+f(3)

f(5)=5+f(4)

栈 [pop(n=4)]

f(4)=4+f(3)=10

f(3)=3+f(2)

f(4)=4+f(3)

f(5)=5+f(4)

栈 [pop(n=3)]

f(3)=3+f(2)=6

f(2)=2+f(1)

f(3)=3+f(2)

f(4)=4+f(3)

f(5)=5+f(4)

栈 [pop(n=2)]

f(2)=2+f(1)=3

递归调用过程分析 (pop)

```
// 递推公式: f(0)=0,f(1)=1, n>1: f(n)=n+f(n-1)
int f(int n)
{
  int sum;
  if(n==0||n==1) sum=n;
  else sum=n+f(n-1); // 未完成的计算用"栈"存储起来(push)
  return sum; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

系统内部自动维护一个称作 “栈” 的存储数据的空间,栈是一种 “先进后出 (FILO)” 的数据结构。向栈中存储数据操作称作 push, 取出数据操作称作 pop。第一个 push 的数据,最后一个被 pop.

f(5)=5+f(4)

栈 [pop(n=5)]

f(5)=5+f(4)=15

f(4)=4+f(3)
f(5)=5+f(4)

栈 [pop(n=4)]

f(4)=4+f(3)=10

f(3)=3+f(2)
f(4)=4+f(3)
f(5)=5+f(4)

栈 [pop(n=3)]

f(3)=3+f(2)=6

f(2)=2+f(1)
f(3)=3+f(2)
f(4)=4+f(3)
f(5)=5+f(4)

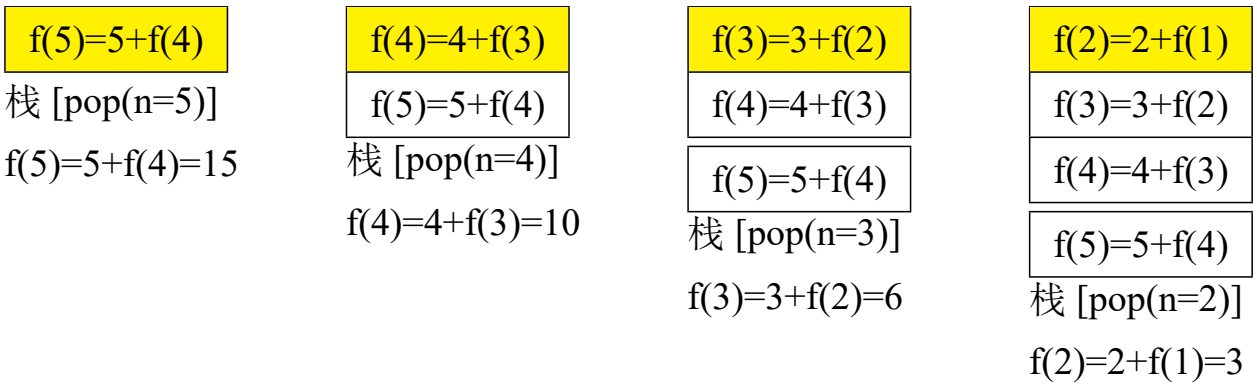
栈 [pop(n=2)]

f(2)=2+f(1)=3

递归调用过程分析 (pop)

```
// 递推公式: f(0)=0,f(1)=1, n>1: f(n)=n+f(n-1)
int f(int n)
{
  int sum;
  if(n==0||n==1) sum=n;
  else sum=n+f(n-1); // 未完成的计算用"栈"存储起来(push)
  return sum; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

系统内部自动维护一个称作 “栈” 的存储数据的空间,栈是一种 “先进后出 (FILO)” 的数据结构。向栈中存储数据操作称作 push, 取出数据操作称作 pop。第一个 push 的数据,最后一个被 pop.



例: age(n)

有 5 个学生坐在一起,问第 5 个学生多少岁,他说比第 4 个学生大 2 岁。问第 4 个学生岁数,他说比第 3 个学生大 2 岁。问第 3 个学生,又说比第 2 个学生大 2 岁。问第 2 个学生,说比第 1 个学生大 2 岁。最后问第 1 个学生,他说是 10 岁。请问第 5 个学生多大。

$$\text{第 n 个学生年龄}\begin{cases} age(n) = 10 & (n = 1) \\ age(n) = age(n-1) + 2 & (n > 1) \end{cases}$$

例: age(n)

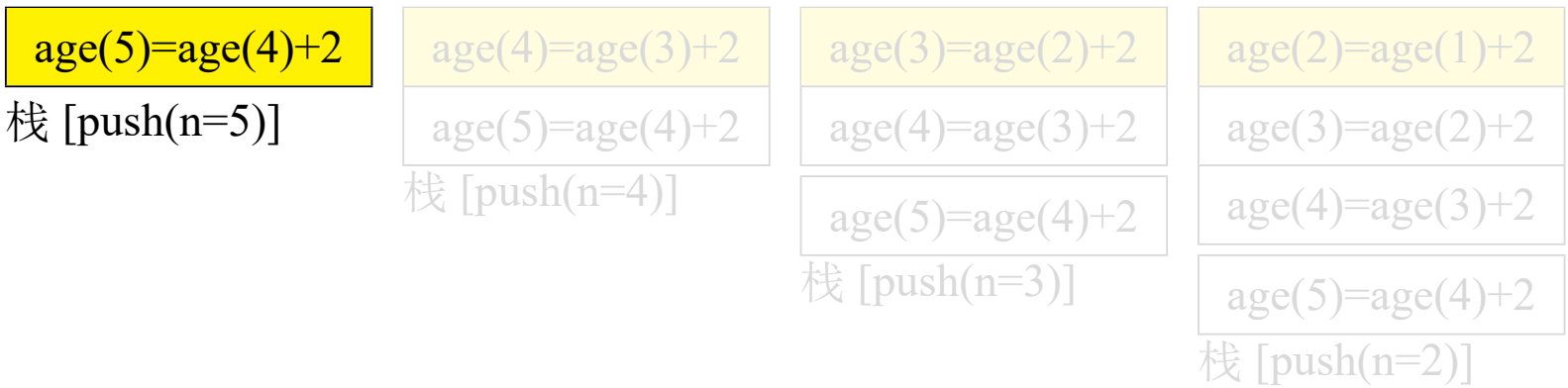
有 5 个学生坐在一起,问第 5 个学生多少岁,他说比第 4 个学生大 2 岁。问第 4 个学生岁数,他说比第 3 个学生大 2 岁。问第 3 个学生,又说比第 2 个学生大 2 岁。问第 2 个学生,说比第 1 个学生大 2 岁。最后问第 1 个学生,他说是 10 岁。请问第 5 个学生多大。

$$\text{第 n 个学生年龄}\begin{cases} age(n) = 10 & (n = 1) \\ age(n) = age(n-1) + 2 & (n > 1) \end{cases}$$

例: age(n), 递归调用过程分析 (push)

```
// 递推公式: age(1)=10; n>1: age(n)=age(n-1)
int age(int n)
{
  int y;
  if(n==1) y=10;
  else y=age(n-1)+2; // 未完成的计算用"栈"存储起来(push)
  return y; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

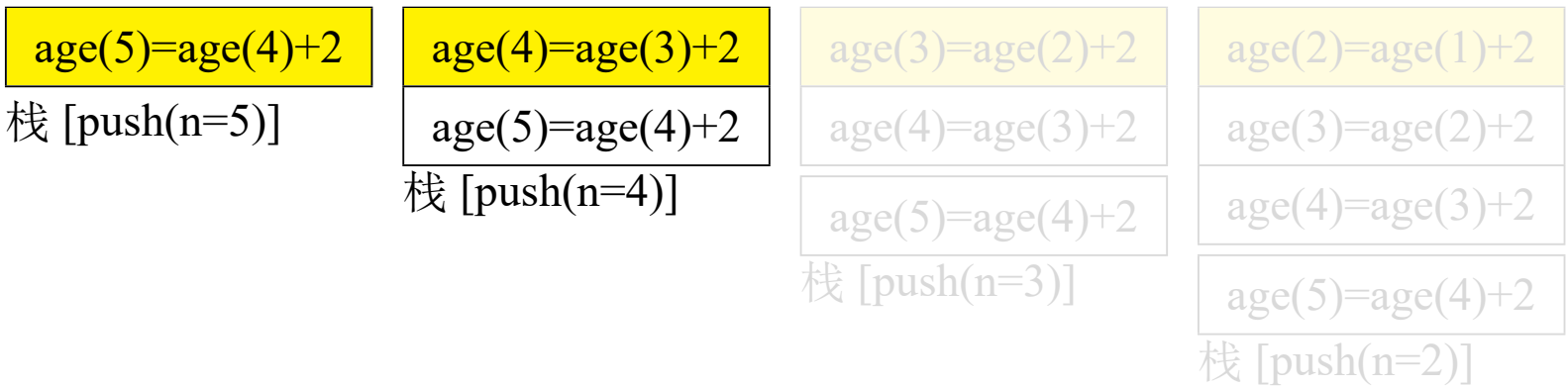
压栈: push



例: age(n), 递归调用过程分析 (push)

```
// 递推公式: age(1)=10; n>1: age(n)=age(n-1)
int age(int n)
{
  int y;
  if(n==1) y=10;
  else y=age(n-1)+2; // 未完成的计算用"栈"存储起来(push)
  return y; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

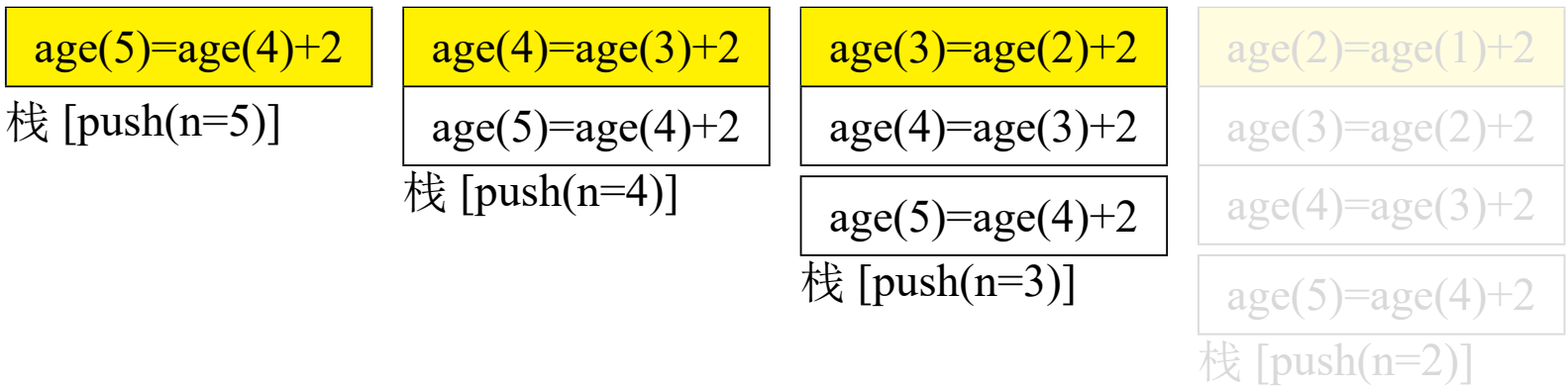
压栈: push



例: age(n), 递归调用过程分析 (push)

```
// 递推公式: age(1)=10; n>1: age(n)=age(n-1)
int age(int n)
{
  int y;
  if(n==1) y=10;
  else y=age(n-1)+2; // 未完成的计算用"栈"存储起来(push)
  return y; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

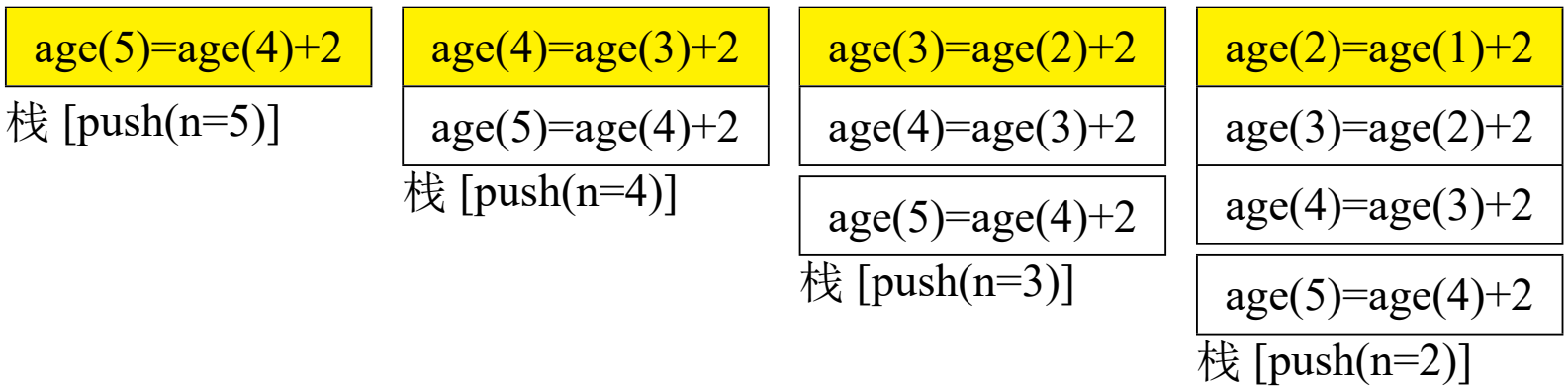
压栈: push



例: age(n), 递归调用过程分析 (push)

```
// 递推公式: age(1)=10; n>1: age(n)=age(n-1)
int age(int n)
{
  int y;
  if(n==1) y=10;
  else y=age(n-1)+2; // 未完成的计算用"栈"存储起来(push)
  return y; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

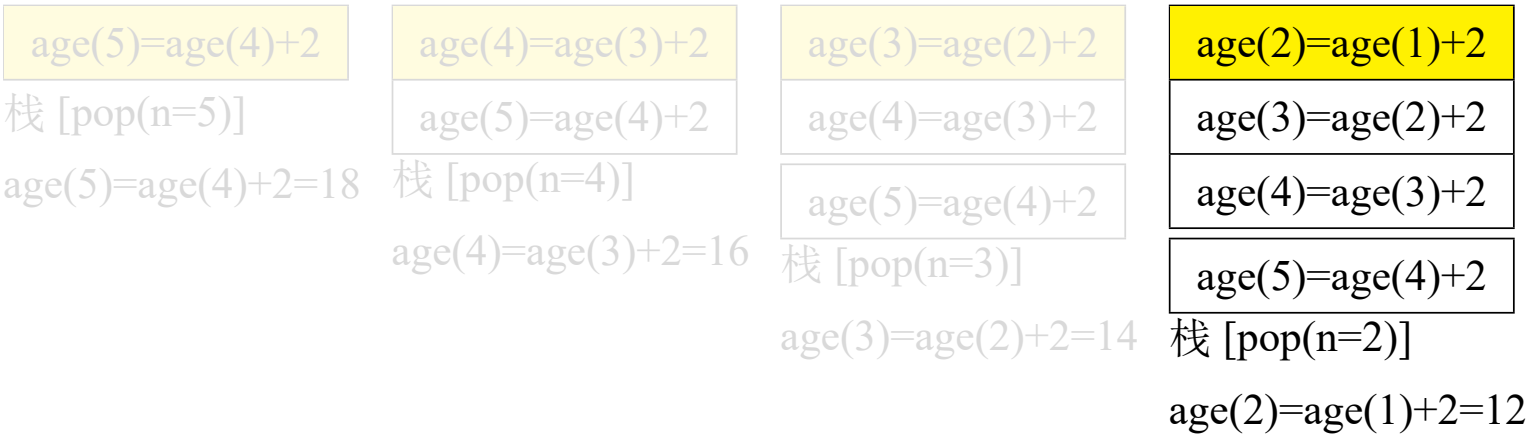
压栈: push



例: age(n), 递归调用过程分析 (pop)

```
// 递推公式: age(1)=10; n>1: age(n)=age(n-1)
int age(int n)
{
  int y;
  if(n==1) y=10;
  else y=age(n-1)+2; // 未完成的计算用"栈"存储起来(push)
  return y; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

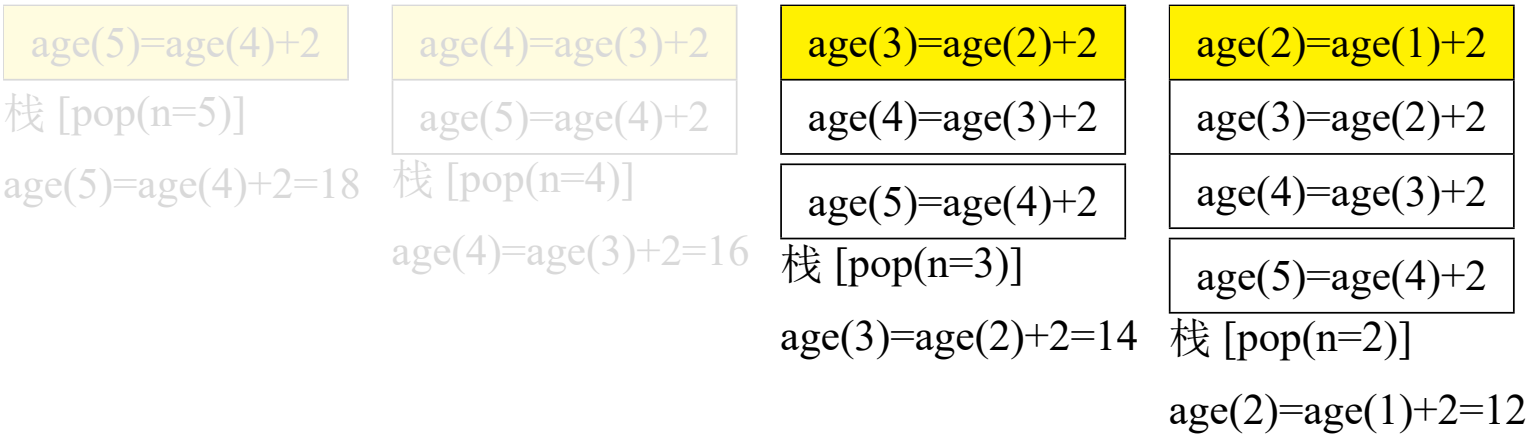
弹出: pop



例: age(n), 递归调用过程分析 (pop)

```
// 递推公式: age(1)=10; n>1: age(n)=age(n-1)
int age(int n)
{
  int y;
  if(n==1) y=10;
  else y=age(n-1)+2; // 未完成的计算用"栈"存储起来(push)
  return y; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

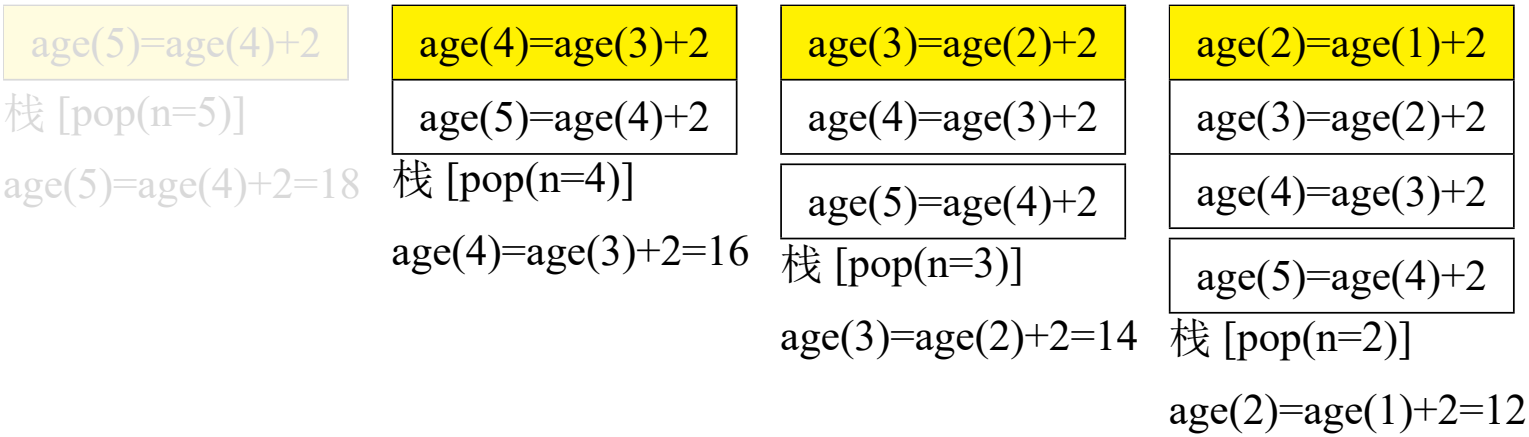
弹出: pop



例: age(n), 递归调用过程分析 (pop)

```
// 递推公式: age(1)=10; n>1: age(n)=age(n-1)
int age(int n)
{
  int y;
  if(n==1) y=10;
  else y=age(n-1)+2; // 未完成的计算用"栈"存储起来(push)
  return y; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

弹出: pop



例: age(n), 递归调用过程分析 (pop)

```
// 递推公式: age(1)=10; n>1: age(n)=age(n-1)
int age(int n)
{
  int y;
  if(n==1) y=10;
  else y=age(n-1)+2; // 未完成的计算用"栈"存储起来(push)
  return y; // 函数return前, 从"栈"顶取数据,计算,直到"栈"空
}
```

弹出: pop

age(5)=age(4)+2

栈 [pop(n=5)]

age(5)=age(4)+2=18

age(4)=age(3)+2
age(5)=age(4)+2

栈 [pop(n=4)]

age(4)=age(3)+2=16

age(3)=age(2)+2
age(4)=age(3)+2
age(5)=age(4)+2

栈 [pop(n=3)]

age(3)=age(2)+2=14

age(2)=age(1)+2
age(3)=age(2)+2
age(4)=age(3)+2
age(5)=age(4)+2

栈 [pop(n=2)]

age(2)=age(1)+2=12

例: $n!$

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

```
double fac(int n) // n!较大, 因此函数返回类型设置为double
{
   if(n==0 || n==1) return 1;
   else return n*fac(n-1);
}
int main()
{
   printf("%.0lf\n",fac(2)); // 2
   printf("%.0lf\n",fac(3)); // 6
   printf("%.0lf\n",fac(4)); // 24
   printf("%.0lf\n",fac(20));// 2432902008176640000
   return 0;
}
```

例: $n!$

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

```
double fac(int n) // n!较大，因此函数返回类型设置为double
{
   if(n==0 || n==1) return 1;
   else return n*fac(n-1);
}
int main()
{
   printf("%.0lf\n",fac(2)); // 2
   printf("%.0lf\n",fac(3)); // 6
   printf("%.0lf\n",fac(4)); // 24
   printf("%.0lf\n",fac(20));// 2432902008176640000
   return 0;
}
```

例: 斐波那契数列

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

```
double F(int n) // Fn较大, 因此函数返回类型设置为double
{
   if(n==0 || n==1) return 1;
   else return F(n-1)+F(n-2);
}
int main()
{
   int i;
   for(i=0;i<20;i++)
   {
      printf("%d\t",(int)F(i));
      if((i+1)%4==0) printf("\n");^^I
   }
   return 0;
}
```

例: 斐波那契数列

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

```
double F(int n) // Fn较大, 因此函数返回类型设置为double
{
   if(n==0 || n==1) return 1;
   else return F(n-1)+F(n-2);
}
int main()
{
   int i;
   for(i=0;i<20;i++)
   {
      printf("%d\t",(int)F(i));
      if((i+1)%4==0) printf("\n");^^I
   }
   return 0;
}
```

求整数 a, b 的最大公约数, 欧几里得算法

古希腊数学家欧几里德在其著作 «The Elements» 中最早描述了这种算法。

定理: 两个整数的最大公约数等于其中较小的那个数和两数相除余数的最大公约数。

伪代码分析

```
a,b的最大公约数, [则: a=mb+r, m=a/b; r=a%b]
循环, b作被除数, 分母是余数r,
[则: n=b/r; r=nb; a=mb+nb=(m+n)b; 如果一个整数能整除b, 必整除a]
直到r=0, 本轮循环的a(上轮循环的b)就是最大公约数。
while(1)
{
  r = a%b; // 注意b为0时, 不能计算余数, a就是最大公约数
  if(r==0) { gcd=a; break; } // 本轮循环的a(上轮循环的b)就是最大公约数
  a=b; b=r; // 准备下一轮迭代
}
```

求整数 a, b 的最大公约数, 非递归实现

```
int a,b,r,t;
scanf("%d%d",&a,&b); // 机试系统不要想当然给提示语句, 除非题目要求
if(a<b) { t=a; a=b; b=t; } // 交换a,b,使a是较大者
while(1)
{
    if(b==0) { t=a; break; } // 分母为0时, a就是最大公约数
    r = a%b;
    if(r==0) {t=b; break;} // 本轮循环的a(上轮循环的b)就是最大公约数
    a=b; b=r; // 准备下一轮迭代
}
printf("%d\n",t);// 输出最大公约数
```

求整数 a, b 的最大公约数, 递归实现

```
// 求a,b的最大公约数(约定a是分子,b是分母)
int gcd(int a, int b)
{
   int result;
   if(b==0) result=a;
   else result=gcd(b,a%b); // b是分子, a%b成为分母
   return result;
}

int main()
{
  int a,b,t;
  scanf("%d%d",&a,&b);
  if(a<b) // 确保 a > b
  {
    t=a; a=b; b=t;
  }
  printf("%d\n",gcd(a,b));
  return 0;
}
```

例: 数字处理

编写一个程序,从键盘输入一个非零整数 n(0 < n <= 1000000000),对整数 n 进行如下处理:

将整数的各位数字取出来相加,如果结果是一位数则输出该数,否则重复上述过程,直到得到的结果为一位数,并输出该结果。

例如:n=456,变换过程如下

4+5+6=15

1+5=6

输出结果为 6

例: 数字处理—非递归实现

```
#include<stdio.h>
int bitsSum(int a);
int main()
{
   int n,sum;
   scanf("%d",&n);
   while(1)
   {
      sum=bitsSum(n);
      if(sum<=9) break;//1位数字
      else n=sum; // 继续下一轮迭代
    }
    printf("%d\n",sum);
    return 0;
}
```

```
// 整数a的各位数字之和
int bitsSum(int a)
{
   int sum=0;
   while(a)
   {
      sum += a%10;
      a /= 10;
   }
   return sum;
}
```

例: 数字处理—递归实现

```c
#include<stdio.h>
int bitsSum(int a);
int bits1(int n);

int main()
{
    int n,sum=0;
    scanf("%d",&n);
    printf("%d\n",bits1(n));
    return 0;
}
```

```c
// 整数a的各位数字之和
int bitsSum(int a)
{
    int sum;
    if(a==0) sum=0;
    else sum=bitsSum(a/10)+a%10;
    return sum;
}
// 确保最后是1位数字
int bits1(int n)
{
    int result;
    result=bitsSum(n);
    if(result<=9) return result;//1位
    else result=bits1(result);//递归
}
```

例: 二进制输出 (正序)

```
void to_binary(unsigned long n);
int main()
{
  // 无符号长整型
  unsigned long x=0XD2; //11010010
  scanf("%x",&x); //十六进制输入
  to_binary(x);
  putchar('\n');
  return 0;
}

void to_binary(unsigned long n)
{
  int r;
  r=n%2;//01001011
  // push(先计算的二进制位r)
  if(n>=2) to_binary(n/2);
  // pop(后进先出)
  putchar('0'+r);//11010010
}
```

以x=XD2为例,分析递归过程。

`r=n%2;`

计算 $2^0, 2^1, \ldots, 2^7$

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
0	1	0	0	1	0	1	1

`to_binary(n/2);`

push(先计算的 r), 栈顶 (top, 最左端), 栈底 (bottom, 最右端)

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	0	1	0

`putchar('0'+r);`

pop(后进先出), 即栈顶元素先出, 栈底元素最后出

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	0	1	0

例: 二进制输出 (倒序)

```
void to_binary(unsigned long n);
int main()
{
  // 无符号长整型
  unsigned long x=0XD2; //11010010
  scanf("%x",&x); //十六进制输入
  to_binary(x);
  putchar('\n');
  return 0;
}

void to_binary(unsigned long n)
{
  int r;
  r=n%2;//01001011
  putchar('0'+r);//01001011
  // push(先计算的二进制位r)
  if(n>=2) to_binary(n/2);
}
```

以x=XD2为例,分析递归过程。

`r=n%2;`

计算 $2^0, 2^1, \ldots, 2^7$

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
0	1	0	0	1	0	1	1

`to_binary(n/2);`

push(先计算的 r), 栈顶 (top, 最左端), 栈底 (bottom, 最右端)

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	0	1	0

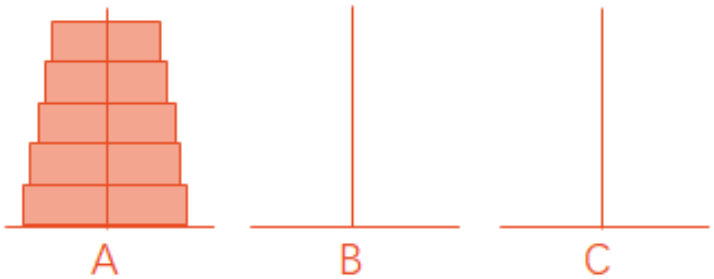
pop(后进先出), 即栈顶元素先出, 栈底元素最后出

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	0	1	0

例: Hanoi(汉诺) 塔问题

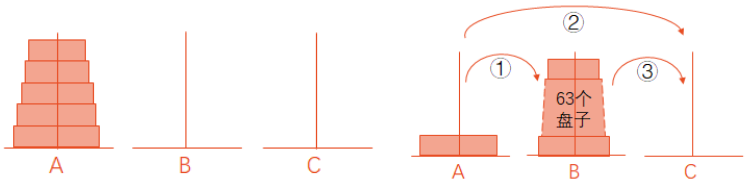
古代有一个梵塔,塔内有 3 个座 A,B,C。开始时 A 座上有 64 个盘子,盘子大小不等,大的在下,小的在上。有一个老和尚想把这 64 个盘子从 A 座移到 C 座,但规定每次只允许移动一个盘,且在移动过程中在 3 个座上都始终保持大盘在下,小盘在上。在移动过程中可以利用 B 座。

要求编程序输出移动盘子的步骤。



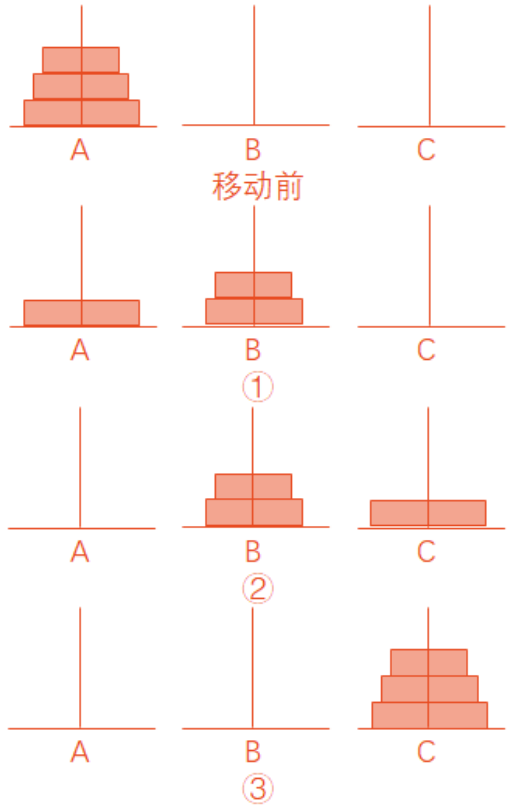
例: Hanoi(汉诺) 塔问题—解题思路

- 老和尚想: 假如有第 2 个和尚将上面 63 个盘子从 A 座移到 B 座。我就能将 64 个盘子从 A 座移到 C 座。。
  1. 命令第 2 个和尚将 63 个盘子从 A 座移到 B 座;
  2. 自己将 1 个盘子 (最底下的、最大的盘子) 从 A 座移到 C 座;
  3. 再命令第 2 个和尚将 63 个盘子从 B 座移到 C 座。
- 第 2 个和尚想: 假如有第 3 个和尚能将上面 62 个盘子从 A 座移到 C 座,我就能将 63 个盘子从 A 座移到 B 座。
  1. 命令第 3 个和尚将 62 个盘子从 A 座移到 C 座;
  2. 自己将 1 个盘子从 A 座移到 B 座;
  3. 再命令第 3 个和尚将 62 个盘子从 C 座移到 B 座。
- ...



例: Hanoi(汉诺) 塔问题—解题思路 (3 层)

- 将 A 座的 3 个盘子移到 C 座, 分 3 步:
  1. 将 A 上 2 个盘子移到 B**(借助 C 座)**。
  2. 将 A 上 1 个盘子移到 C**(直接实现)**。
  3. 将 B 上 2 个盘子移到 C**(借助 A 座)**。
- 分解第 1 步: 将 A 上的 2 个盘子移到 B:
  1. 将 A 上 1 个盘子移到 C**(借助 C 座)**。
  2. 将 A 上 1 个盘子移到 B。
  3. 将 C 上 1 个盘子移到 B。
- 分解第 3 步: 将 B 座的 2 个盘子移到 C 座:
  1. 将 B 上 1 个盘子移到 A**(借助 A 座)**。
  2. 将 B 上 1 个盘子移到 C。
  3. 将 A 上 1 个盘子移到 C。



例: Hanoi(汉诺) 塔问题—解

```
void hanoi(int n,char one,
    char two,char three);
void move(char one,char two)
    ;
int main()
{
   int n=3;
   scanf("%d",&n);
   //n个盘由'A'移到'C',借助'B'
   hanoi(n,'A','B','C');
   return 0;
}
void move(char one,char two)
{
 printf("%c->%c\n",one,two);
}
```

```
//将n个盘从one移到three,借助two
void hanoi(int n,char one,char two,char
    three)
{
   if(n==1) move(one,three);
   else
   {
      //n-1盘从one移到two,借three
      hanoi(n-1,one,three,two);
      //将1个盘从one移到three
      move(one,three);
      //n-1盘从two移到three,借one
      hanoi(n-1,two,one,three);
   }
}
```

- 将 A 座的 3 个盘子移到 C 座, 分 3 步:
  1. 将 A 上 2 个盘子移到 B**(借 C)**。
  2. 将 A 上 1 个盘子移到 C**(直接)**。
  3. 将 B 上 2 个盘子移到 C**(借 A)**。
- 分解第 1 步: 将 A 上的 2 个盘子移到 B:
  1. 将 A 上 1 个盘子移到 C**(借 C)**。
  2. 将 A 上 1 个盘子移到 B。
  3. 将 C 上 1 个盘子移到 B。
- 分解第 3 步: 将 B 座的 2 个盘子移到 C 座:
  1. 将 B 上 1 个盘子移到 A**(借 A)**。
  2. 将 B 上 1 个盘子移到 C。
  3. 将 A 上 1 个盘子移到 C。

```
//将n个盘从one移到three,借助two
void hanoi(int n,char one,char two,char
    three)
{
  if(n==1) move(one,three); // 递归终止条件,
    出栈pop(2,...,n-1,n), 回归, 回溯
  else
  { // 压栈push(n,n-1,n-2,...2)
    //n-1盘从one移到two,借three
    hanoi(n-1,one,three,two);
    //将1个盘从one移到three
    move(one,three);
    //n-1盘从two移到three,借one
    hanoi(n-1,two,one,three);
  }
}
```

$A \to C, A \to B, C \to B, A \to C, B \to A, B \to C, A \to C$

递归小结

- 递归函数必须满足两个条:
  1. 在每一次调用自己时,必须是 (在某种意义上) 更接近于解;
  2. 必须有一个终止处理或计算的准则。
- 递归能够解决的问题:
  1. 数据的定义是按递归定义的。如 Fibonacci 函数。
  2. 问题解法按递归算法实现。如 Hanoi 问题。
  3. 数据的结构形式是按递归定义的。如二叉树、广义表等。
- 使用递归的关键在于将问题分解为小部分,递归不能永远进行下去,因为它总是以最小可能性问题结束,
- 递归函数的优点是定义简单,逻辑清晰。
- 理论上,所有的递归函数都可以写成循环的方式,但循环的逻辑不如递归清晰。
- 层次越深, 调用栈 (内存) 越大, 效率越低, 并且可能会溢出, 。因此, 能用循环解决的问题, 尽量不要用递归.

局部变量及其作用域

定义在函数内部或复合语句中的变量称为**局部变量**。

局部变量的**作用域**是从定义语句开始至函数或复合语句结束,不会影响作用域以外的同类型的同名变量值。

```
// 各函数中的局部变量，不会相互影响。
float f1(folat a)
{
   float c; // 本函数局部变量
   ....;
}
float f2(float a)
{
   float c; // 本函数局部变量
   ....;
}
int main()
{
   float c=10,d; // 本函数局部变量
   d=f1(c);
   d=f2(d);
}
```

```
// 各复合语句中的局部量，不会相互影响。
float f1(folat a)
{
   float c; // 本函数局部变量
   for(i=0;i<10;i++)
   {
     int t; // 复合语句中的局部变量
     ....
   }
   if(c>10)
   {
     int t; // 复合语句中的局部变量
     ....
   }
   ...
}
```

形式参数被当作本函数的局部变量

形式参数被当作本函数的局部变量,因此它不会影响实际参数的值。

```
float f1(folat a)
{
  float c; // 本函数内部的局部变量不得与
    形式参数同名
  a=30; // 改变形式参数的值不会影响实际
    参数的值
  ....;
}
int main()
{
   float c=10,d; // 本函数局部变量
   d=f1(c); // f1函数对形参的改变不会影
    响实参c的值
   float x[2]={0.1,0.2}; // 函数内部
    可以改变数组元素的值
   printf("%f,%f,%f\n",c,x[0],x[1])
    ; // 10,10.1,10.2
}
```

```
// 函数内部可以改变数组元素的值,但是对地址
    的改变不会影响实参的地址
float f2(folat a[])
{
  a[0]=10.1; // 对数组元素的改变, 就是
    对实参数组元素的改变
  a[1]=10.2;
  a=a+1; // 对数组名(地址)的改变, 不会
    影响实参的地址。
  ...
}
```

全局变量及其作用域

定义在函数外部的变量称为**全局变量**。与模块化函数封装思想冲突,不推荐使用。全局变量的作用域是从定义语句开始至该文件结束,会影响作用域内的同类型的同名变量值。

```
int g=0; // 全局变量
float f1(folat a)
{
  g=10; // 改变全局变量的值
  ....;
}
float f2(float a)
{
  g=20; // 改变全局变量的值
  ....;
}
int main()
{
   g=30; // 改变全局变量的值
   ....;
}
```

欢迎批评指正！