Recursion

- Recursion is a function invoking itself, either directly or indirectly.
- It can be used as an alternative to iteration.
- Recursion is an important and powerful tool in problem solving and programming. It is a programming technique that naturally implements the divide-and-conquer problem solving methodology.

In its simplest form the idea of recursion is straightforward:

Example 1:

```
void count_down(int n)
{
   if (n <= 0)
      printf("\nBlast off.\n");
    else{
      printf("%d! ", n);
      count_down(n-1);
   }
}

int main ()
{
   count_down(10);
}
```


Example 2: Multiplication of natural numbers.

a * b = ***a*** added to itself ***b*** times. (iterative definition)

$$\left.\begin{array}{ll} a * b = a & \text{if } b=1 \\ a * b = a * (b-1) + a & \text{if } b > 1 \end{array}\right\} \text{(recursive definition)}$$

e.g.
6 * 3 = 6 * 2 + 6 = 6 * 1 + 6 + 6 = 6 + 6 + 6 = 18

In C:

```
int multiply(int a, int b)
{
   if (b == 1)  /* stopping case */
      return a;
   else         /* recursive step */
      return ( a + multiply(a, b-1));
}
```

Example 3: Factorial Function

n! = n * (n-1) * (n-2) * … * 2 * 1

5! = 5 * 4 * 3* 2 * 1
0! = 1

Mathematical definition:

n! = 1 if *n = 0*
n! = n . (n-1) . (n-2). …. 2 . 1 if *n > 0*

Iteratively:

```
p = 1;
for (x=n; x>=1; x--)
   p = p* x;
```

Recursive definition:

4! = 4 * 3 * 2 * 1 = 4 * 3!

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

Thus,

n! = 1 if n = 0
n! = n(n-1)!* if n > 0

In C:

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

The Nature of Recursion

1)One or more simple cases of the problem (called the *stopping cases*) have a simple non-recursive solution.
2)The other cases of the problem can be reduced (*using recursion*) to problems that are closer to stopping cases.
3)Eventually the problem can be reduced to stopping cases only, which are relatively easy to solve.

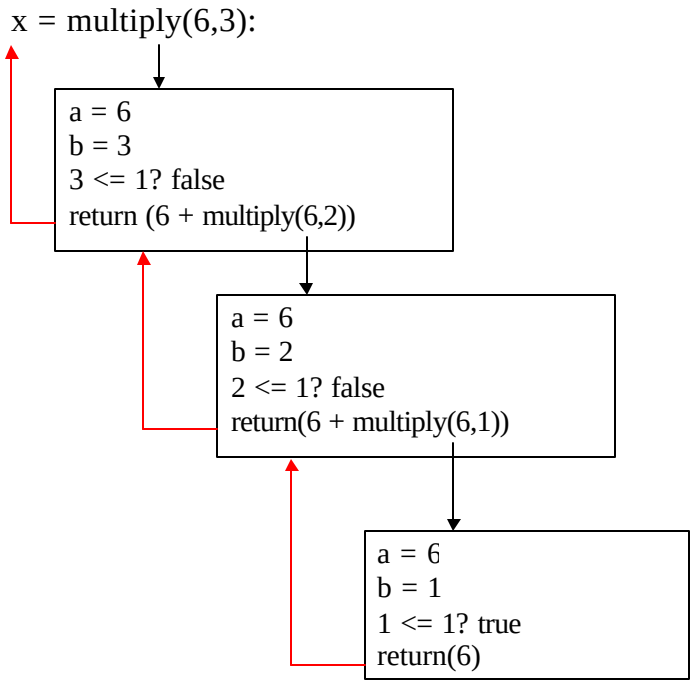
In general:

if (stopping case)
    solve it
else
    reduce the problem using recursion

Tracing a Recursive Function

Computer uses a stack to keep track of function calls. Whenever a new function is called, all its parameters and local variables are pushed onto the stack along with the memory address of the calling statement (this gives the computer the return point after execution of the function)

Tracing the function multiply



Example: Tracing a Recursive Function

Palindrome is a string of characters that reads the same backwards and forwards (e.g. level, deed, mom)

palindrome(5) reads 5 characters and prints them in reverse order.

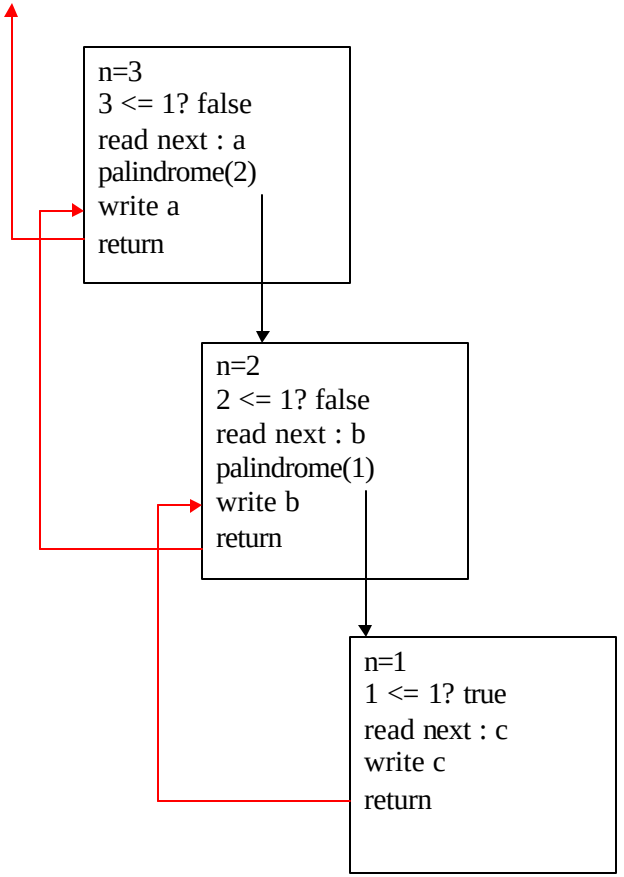
```
void palindrome(int n)
{
   char next;

   if (n == 1) {   /* stopping case */
      scanf("%c",&next);
      printf("%c", next);
    }
   else {
      scanf("%c", &next);
      palindrome(n-1);
      printf("%c",next);
   }
   return;
}

int main()
{
   printf("Enter a string: ");
   palindrome(5);
   printf("\n");
}
```


Trace of palindrome: for input `abc`

palindrome(3);



Example 4: Fibonacci Sequence

It is the sequence of integers:

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
0	1	1	2	3	5	8	13	21	34 …

Each element in this sequence is the sum of the two preceding elements.

The specification of the terms in the Fibonacci sequence:

$$t_n = \begin{cases} n & \text{if } n \text{ is } 0 \text{ or } 1 \text{ (i.e. n < 2)} \\ t_{n-1} + t_{n-2} & \text{otherwise} \end{cases}$$

In C:

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

Calling the function :

```
x = fibonacci(5);
```


Common Errors

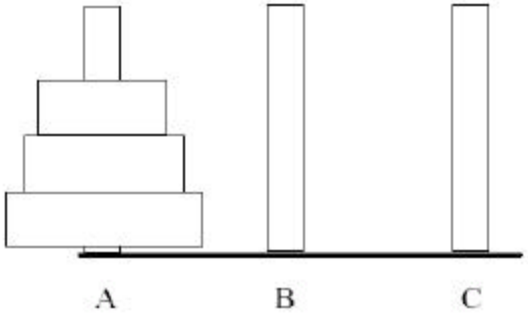
- It may not terminate if the stopping case is not correct or is incomplete (stack overflow: run-time error)
- Make sure that each recursive step leads to a situation that is closer to a stopping case.

Comparison of Iteration and Recursion

- In general, an iterative version of a program will execute more efficiently in terms of time and space than a recursive version. This is because the overhead involved in entering and exiting a function is avoided in iterative version.
- However a recursive solution can be sometimes the most natural and logical way of solving a problem.
  ⇨Conflict: machine efficiency versus programmer efficiency
- It is always true that recursion can be replaced with iteration and a stack.

Problem Solving with Recursion

Towers of Hanoi Problem: involves moving a specified number of disks (N) that are all different sizes from one tower to another.

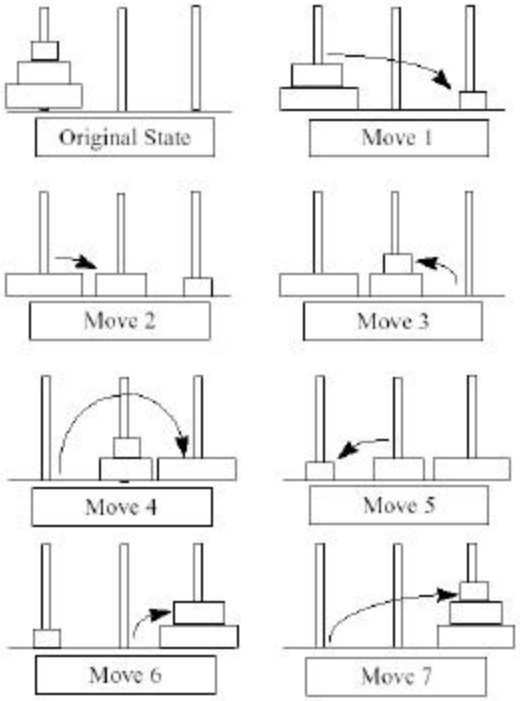


The goal is to move all disks from tower A to C subject to the following rules:

1. Only one disk may be moved at a time and this disk must be the top disk on a tower.
2. A larger disk can never be placed on top of a smaller disk.

The stopping cases of the problem involve moving only one disk.

Towers of Hanoi: Solution



Problem: Solve the Towers of Hanoi for N disks.

Analysis: Solution consists of a printed list of individual disk moves. We need recursion that can be used to move any number of disks from one tower to another, using the third tower as a temporary tower.

Inputs: n: integer,
start:'A', 'B', or 'C',
finish:'A', 'B', or 'C'
temp:'A', 'B', or 'C'

Output: a list of individual disk moves.

Algorithm

if (n == 1) /* *stopping case* */
move a single disk from start *to* finish
else
-*Move* n-1 *disks from* start *to* temp *using* finish
as temporary tower.
- *Move a single disk from start to finish*
- *Move* n-1 *disks from* temp *to*
finish *using* start *as temporary tower*

In C:

```
void tower(int n, char start, char finish, char temp)
{
  if (n == 1)
    printf("Move from %c to %c\n", start, finish);
  else {
    tower(n-1, start, temp, finish);
    printf("Move from %c to %c \n", start, finish);
    tower(n-1, temp, finish, start);
  }
}
```

Test:

```
tower(3,'A','C', 'B');
```

Output:

```
Move from A to C
Move from A to B
Move from C to B
Move from A to C
Move from B to A
Move from B to C
Move from A to C
```

Class Exercises

1) Trace the following recursive function:

```
#include <stdio.h>

int f(char *s)
{
   if (*s == '\0')
     return 0;
   else
     return (1 + f(s+1));
}

int main()
{
   char a[20] = "Computer Science I";

   printf("%d\n",f(a));
}
```


2) Trace the following recursive function:

```
#include <stdio.h>

int f(int c)
{
   if (!(c > 10)) {
     printf("%d\n", c);
     f(c + 1);
   }
}

int main()
{
   f(0);

}
```

3) Trace the following recursive function:

```
#include <stdio.h>
void f(int);
void g(int);

void f(int c)
{
   printf("hello from f()\n");
   if (++c <= 3)
     g(c);
}

void g(int c)
{
   printf("hello from g()\n");
   f(c);
}

int main()
{
  printf("hello from main\n");
  f(1);
  return 0;
}
```


4) Write a recursive function to check if a given item is a member of a set. Function prototype is:

```
/* Inputs: An integer array, the item
   being searched and the index of the
   last element in the array.
   Output: true (1) or false (0)
*/

int isMember(int a[], int item, int n);
```

5) Write a recursive function to check if the contents of an array are in ascending order or not. The function prototype is:

```
/* Inputs: an integer array, the index
   of the last element in the array.
   Output: true or false.
*/

int isAscending(int a[], int n);
```