Chapter 5: Recursion

Recursion

- **Recursion:** ***a method to define something in terms of itself***
- **Recursive Function:** ***A function that calls itself***
- One of the most powerful programming tool
- Natural way to solve many problems
- Makes algorithms and its implementation more ***compact*** and ***simple***

The Factorial function

- For a positive integer ***n***, the factorial of ***n*** is defined as the product of all integers between ***n*** and 1
  - For e.g., 5 factorial equals 5 * 4 * 3 * 2 * 1 = 120 and 3 factorial equals 3 * 2 * 1 = 6
  - 0 factorial is defined as 1
- In mathematics, ***n*** factorial is denoted by ***n***!

Factorial Definition

- $n! = 1$, if $n = 0$
  $n! = n * (n - 1) * (n - 2) * \ldots * 1$, if $n > 0$
- Hence,
  0! = 1
  1! = 1
  2! = 2 * 1
  3! = 3 * 2 * 1
  4! = 4 * 3 * 2 * 1

Iterative Algorithm for Function

Algorithm to evaluate the product of all integers between n and 1

```
prod = 1;
for (i = n; i > 0; i--)
     prod *= i;
```

prod is the required result

- This type of algorithm is called ***iterative***
  - Because it calls for the explicit repetition of some process until a certain condition is met

Thinking Recursively

- We know, 4! = 4 * 3 * 2 * 1
- But 3 * 2 * 1 is 3!, so we can write 4! = 4 *3!
- In fact, for any $n > 0$,
  we see that $n!$ equals $n * (n - 1)!$
  - Multiplying n by the product of all integers between from $n - 1$ to 1 yields the product of all integers from n to 1

Recursive Definition of Factorial

- $n! = 1$ if $n = 0$
  $n! = n * (n - 1)!$ if $n > 0$
- Hence,
  0! = 1
  1! = 1 * 0!
  2! = 2 * 1!
  3! = 3 * 2!
  4! = 4 * 3!

A definition that defines an object in terms of simpler case of itself is called a recursive definition

Evaluating Factorials from Recursive Definition

- From definition,
  1. 5! = 5 * 4!
  2. 4! = 4 * 3!
  3. 3! = 3 * 2!
  4. 2! = 2 * 1!
  5. 1! = 1 * 0!
  6. 0! = 1
- Each case is reduced to a simpler case until we reach the case of 0!, which is defined directly as 1
- In line 6, we have evaluated factorial directly, so we backtrack from line 6 to 1, returning the value computed in one line to evaluate the result of the previous line

Properties of Recursion

- Every recursive process consists of two parts:
  - A smallest, base case that is processed without recursion; and
  - A general method that reduces a particular case to one or more of the smaller cases, thereby making progress toward eventually reducing the problem all the way to the base case

Sharad C. Joshi

Fibonacci Numbers

- *fib* (*n*) = *n* if *n* = 0 or *n* = 1
  fib (*n*) = *fib* (*n* – 1) + *fib* (*n* – 2) if *n* >=2
- Evaluate *fib* (5)
  = *fib* (4) + *fib* (3)
  = (*fib* (3) + *fib* (2)) + (*fib* (2) + *fib* (1))
  = ((*fib* (2) + *fib* (1)) + (*fib* (1) + *fib* (0))) +
  ((*fib* (1) + *fib* (0)) + 1)
  = (((*fib* (1) + *fib* (0)) + 1) + (1+0)) + ((1+0) + 1)
  = (((1 + 0) + 1) + (1)) + ((1) + 1)
  = (((1) + 1) + (1)) + ((1) + 1)
  = 5

Sharad C. Joshi

Greatest Common Divisor

- $gcd\ (m, n) = m$ if $n = 0$
  $gcd\ (m, n) = gcd\ (n, m \bmod n)$ otherwise
- Find *gcd* (1440, 408)

Sharad C. Joshi

Conversion to Binary

Algorithm to print the binary representation of N

- Stop if $N = 0$
- Print the binary representation of the integer $N/2$
- Write a ‘1’ if N is odd and a ‘0’ if N is even

Sharad C. Joshi

Recursion in C

- C allows to write functions that call themselves. Such functions are called **recursive**

```
int fact(int n)
{
   int x, y, prod;
   if (n == 0) /* base case */
       prod = 1;
   else {
       x = n-1;
       y = fact(x); /* recursive call */
       prod = n * y;
   }
   return prod;
}
```


How it works

- `printf("%d", fact(4));`
- When `fact` is called for the first time, the parameter `n` is set to `4`
- Since `n` is not `0`, `x` is set equal to `3`
- Now again `fact` is called but with the parameter `n` equal to `3`
- Function `fact` is reentered and the local variables are reallocated, including `n`

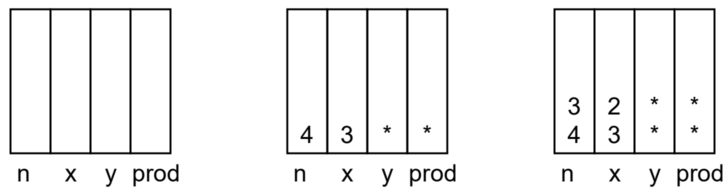
How it works

- Since execution has not left the first call of `fact`, the first allocation of these variables remains
- However, at any time of execution, only the recent copy of the variables can be referenced
- Each time the function `fact` is called recursively, a new set of local variables and parameters is allocated
- When a return from `fact` to a point in a previous call takes place, the most recent allocation of these variables is freed and the previous copy is reactivated

Use of Stacks in Function Call

- The description suggests the use of a stack to keep the successive generations of local variables and parameters
- This stack is maintained by the C system and is invisible to the user (Internal stack)
- Each time a function is called, a new allocation of its variables are pushed on top of the stack
  - Any reference to a local variable and parameters is through the current top of the stack
- When a function returns, the stack is popped, the top allocation is freed and the previous allocation becomes the current stack top

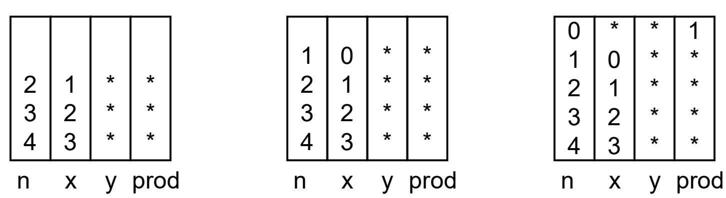
Illustration



Initially   ***fact(4)***   ***fact(3)***

Sharad C. Joshi

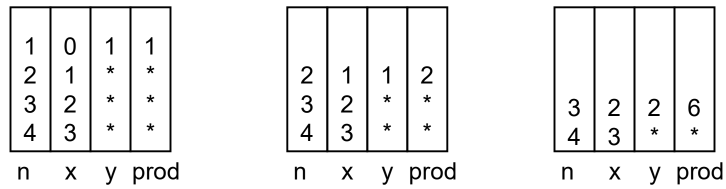
Illustration



fact(2)   ***fact(1)***   ***fact(0)***

Sharad C. Joshi

Illustration



fact(1)

y = fact(0)
prod = n * y

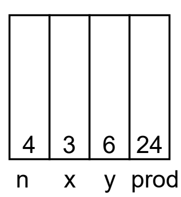
fact(2)

y = fact(1)
prod = n * y

fact(3)

y = fact(2)
prod = n * y

Illustration



fact(4)

y= fact(3)
prod = n * y

The Factorial Function Rewritten

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

Sharad C. Joshi

The PrintBinary Function

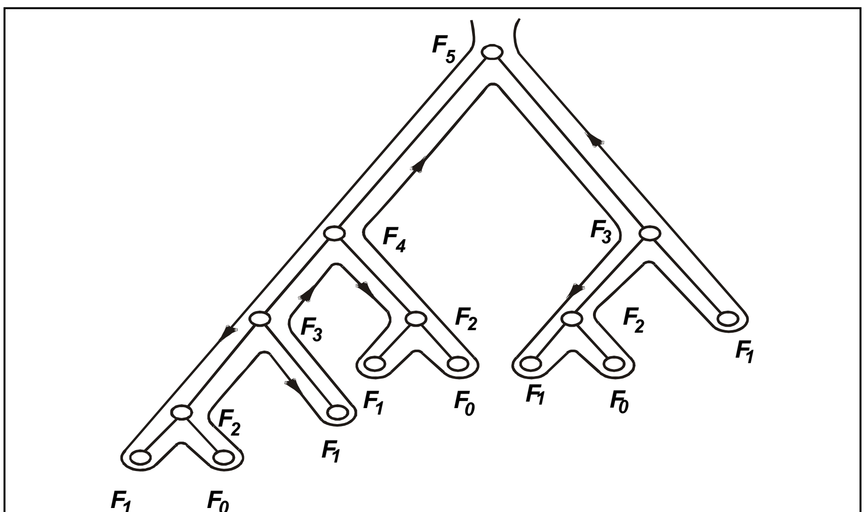
```
void PrintBinary(int n)
{
   if (n == 0)
      return;
   PrintBinary(n/2);
   printf("%d", n%2);
}
```

Sharad C. Joshi

The Fibonacci Function

```
int fib(int n)
{
   int a, b, sum;
   if (n == 0 || n == 1)
      return n;
   else
   {
     a = fib(n - 1);
     b = fib(n - 2);
     sum = a + b;
     return sum;
   }
}
```


Trace of Evaluation of Fib (5)



The Towers of Hanoi Problem

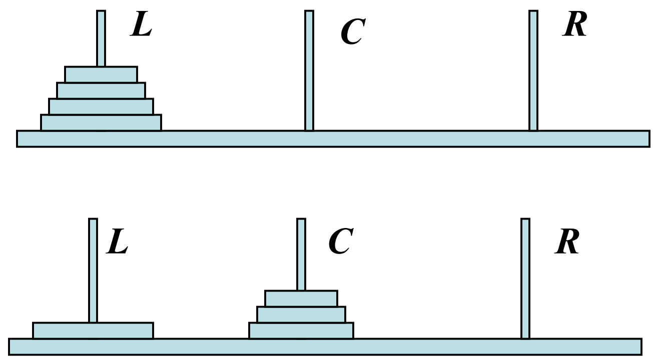
- Three pegs, L, C, and R, exists
- Disks of different diameters are placed on peg L so that a larger disk is always below a smaller disk
- The objective is to move the disks to peg R, using C as auxiliary
  - Only the top disk on any peg may be moved to any other peg
  - A larger disk may never rest on a smaller one

The Idea

- The idea that gives a solution is to concentrate our attention not on the first step (which must be to move the top disk somewhere), but rather on the hardest step: moving the bottom disk
- There is no way to reach the bottom disk until all the disks above the bottom have been moved, and, furthermore they must all be on peg C so that we can move the bottom disk from peg L to ***R***

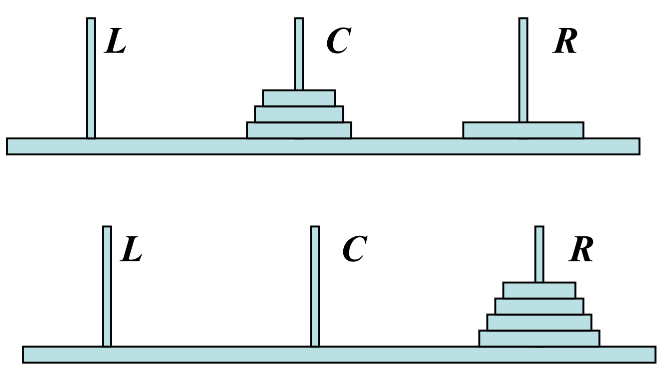
Sharad C. Joshi

Demonstration



Sharad C. Joshi

Demonstration



Sharad C. Joshi

Algorithm for Towers of Hanoi

- **Algorithm, move *n* disks from *L* to R using C as auxiliary**

1. If ***n*** == **1**, move the single disk from **L** to **R** and stop
2. Move the top ***n*** – **1** disks recursively from ***L*** to ***C***, using ***R*** as auxiliary
3. Move the nth disk from ***L*** to ***R***
4. Move the ***n*** – **1** disks recursively from ***C*** to ***R***, using ***L*** as auxiliary

https://www.youtube.com/watch?reload=9&v=YstLjLCGmgg

Sharad C. Joshi

The Move Function

```
void Transfer(int n, char from, char to, char aux)
{
   if (n == 1) {
      printf("Move disk 1 from peg %c to peg %c\n",
              from, to);
      return;
   }

   Transfer(n-1, from, aux, to);
   printf("Move disk %d from peg %c to peg %c",
          n,from, to);
   Transfer(n-1, aux, to, from);
}
```

Sharad C. Joshi

Recursion tree for 3 disks

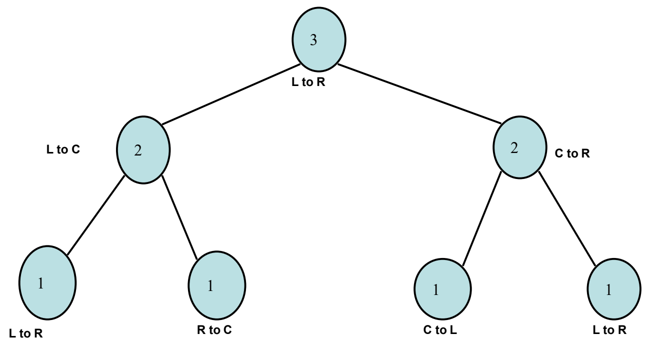


Fig: Working of TOH disk transfer as a binary tree

Sharad C. Joshi

Constructing Recursive Algorithms

- Find a way to divide the whole task, so that it becomes manageable
- ***Identify the base case:*** What is the trivial solution step and what is its associated condition?
- ***Identify the recursion step:*** How can the problem be made (slightly) smaller?
- Make sure that the problem reduction eventually leads to the trivial case

Sharad C. Joshi

Iteration and Recursion

- ***Iteration***
  - as long as the condition is true the loop body is executed
  - when the loop body has been executed for the last time, the loop completely terminates
- ***Recursion***
  - as long as the recursion condition is true the method is called again
  - when the base case has been reached, no further recursion occurs
  - however, all recursive calls then unfold backwards, possibly leading to the execution of further code

Recursion and Efficiency

- Some recursive solutions are so inefficient that they should not be used
- Factors that contribute to the inefficiency of some recursive solutions
  - Overhead associated with method calls
  - It consumes more storage space, all the automatic (local) variables are stored on the stack
  - If the condition is not checked during recursion, computer may run out of memory.
  - If proper care is not taken, recursion may result in non-terminating iterations.

The End