Lecture : Recursion

Sidra mudassar

# Recursion

* A function is said to be *recursively defined,* if a function containing either a Call statement to itself or a Call statement to a second function that may eventually result in a Call statement back to the original function.
* A recursive function must have the following properties:
  1. There must be certain criteria, called base criteria for which the function does not call itself.
  2. Each time the function does call itself(directly or indirectly), the argument of the function must be closer to a base value.

# Recursion

* In some problems, it may be natural to define the problem in terms of the problem itself.
* Recursion is useful for problems that can be represented by a simpler version of the same problem.
* Example: the factorial function

**6! = 6 \* 5 \* 4 \* 3 \* 2 \* 1** We could write:

**6! = 6 \* 5!**

# Example 1: factorial function

In general, we can express the factorial function as follows:

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

The factorial function is only defined for *positive* integers. So we should be a bit more precise:

**if n<=1, then n! = 1 if n>1, then n! = n \* (n-1)!**

# factorial function

The C++ equivalent of this definition:

**int fac(int numb){ if(numb<=1)**

**return 1;**

**else**

**return numb \* fac(numb-1);**

**} *recursion* means that a function calls itself**

# factorial function

• Assume the number typed is 3, that is, numb=3.

**fac(3) :**

**3 <= 1 ? No.**

|  |
| --- |
| **int fac(int numb){ if(numb<=1) return 1;**  **else return numb \* fac(numb-1); }** |

**fac(3) = 3 \* fac(2) fac(2) :**

**2 <= 1 ? No. fac(2) = 2 \* fac(1) fac(1) : 1 <= 1 ? Yes. return 1**

**fac(2) = 2 \* 1 = 2 return fac(2)**

**fac(3) = 3 \* 2 = 6 return fac(3) fac(3) has the value 6**

# Factorial Function

For certain problems (such as the factorial function), a recursive solution often leads to short and elegant code. Compare the recursive solution with the iterative solution:

**Iterative solution Recursive solution**

**int fac(int numb){**

**int fac(int numb){int product=1; if(numb<=1)while(numb>1){ return 1;product \*= numb; elsenumb--; return numb\*fac(numb-1);}**

**}return product;**

**}**

# Recursion

To trace recursion, recall that function calls operate as a stack – the new function is put on top of the caller

We have to pay a price for recursion:

* calling a function consumes more time and memory than adjusting a loop counter.
* high performance applications (graphic action games, simulations of nuclear explosions) hardly ever use recursion.

In less demanding applications recursion is an attractive alternative for iteration (for the right problems!)

# Recursion

If we use iteration, we must be careful not to create an infinite loop by accident:

**for(int incr=1; incr!=10;incr+=2)** ...

**int result = 1;**

**while(result >0){ ...**

**result++;**

**}**

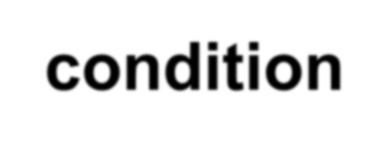
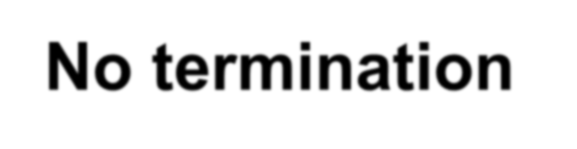
# Recursion

Similarly, if we use recursion we must be careful not to create an infinite chain of function calls:

**int fac(int numb){ return numb \* fac(numb-1);**

## } No termination

Or: **condition**



**int fac(int numb){**

**if (numb<=1) return 1;**

**else return numb \* fac(numb+1);**

**}**

# Recursion

We must always make sure that the recursion *bottoms out*:

* A recursive function must contain at least one non-recursive branch.
* The recursive calls must eventually lead to a non-recursive branch.

# Recursion

* Recursion is one way to decompose a task into smaller subtasks. At least one of the subtasks is a smaller example of the same task.
* The smallest example of the same task has a nonrecursive solution.

Example: The factorial function

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

# Example : Towers of Hanoi

**A**

**B**

**C**

3

2

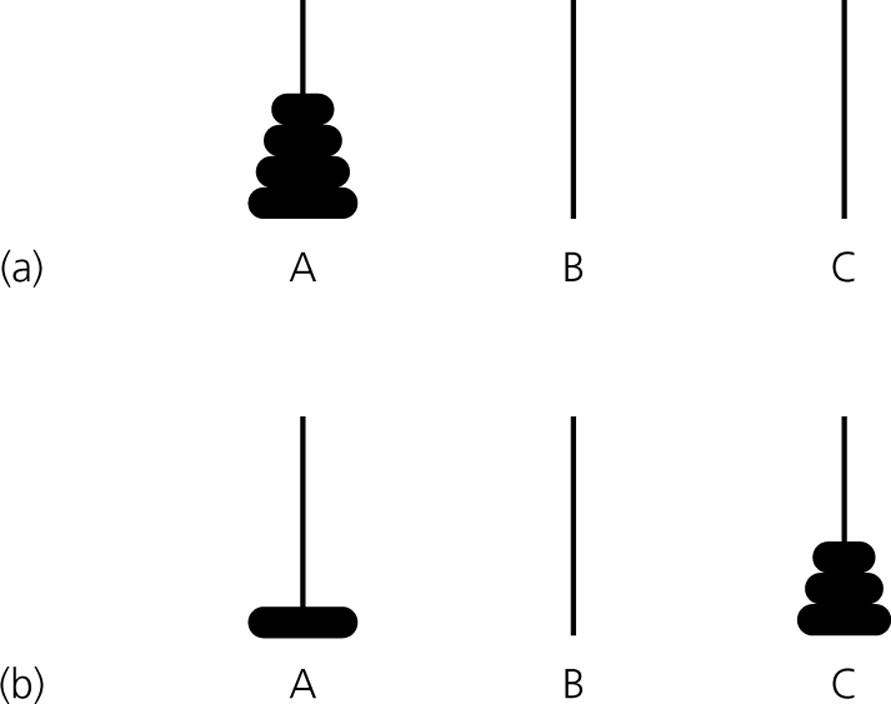
1

* Only one disc could be moved at a time
* A larger disc must never be stacked above a smaller one
* One and only one extra needle could be used for intermediate storage of discs

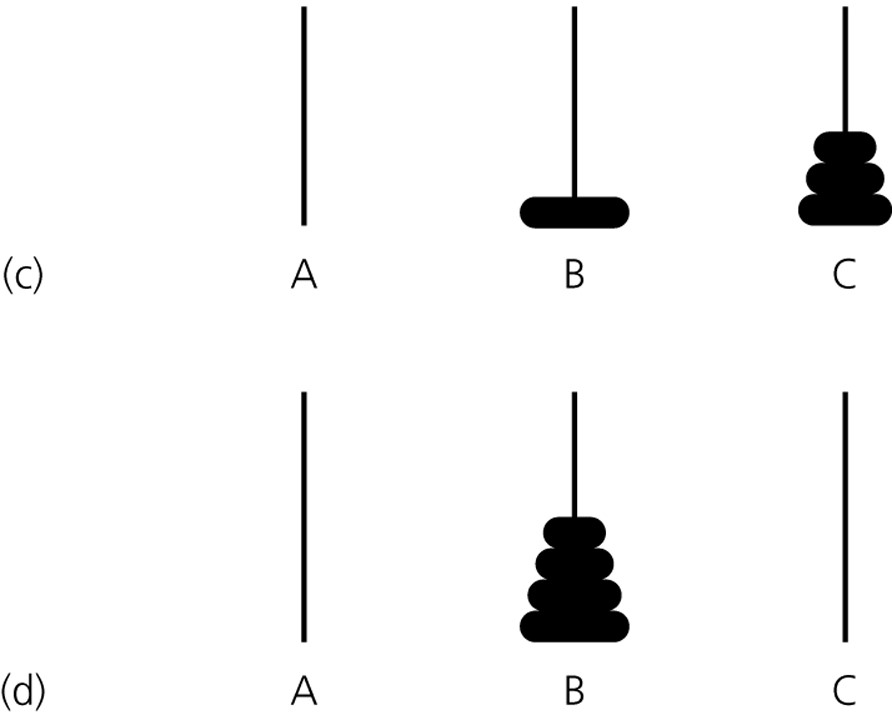
# Towers of Hanoi

* From the moves necessary to transfer one, two, and three disks, we can find a *recursive pattern* - a pattern that uses information from one step to find the next step.
* If we want to know how many moves it will take to transfer 64 disks from post A to post C, we will first have to find the moves it takes to transfer 63 disks, 62 disks, and so on.

a) The initial state; b) move *n* - 1 disks from *A* to *C*



c) move one disk from *A* to *B*; d) move *n* - 1 disks from *C* to *B*



# Towers of Hanoi

* The recursive pattern *can* help us generate more numbers to find an *explicit* (non-recursive) pattern. Here's how to find the number of moves needed to transfer larger numbers of disks from post A to post C, when M = the number of moves needed to transfer n-1 disks from post A to post C:
* for **1 disk** it takes 1 move to transfer 1 disk from post A to post C;
* for **2 disks**, it will take 3 moves: 2M + 1 = 2(**1**) + 1 = **3**
* for **3 disks**, it will take 7 moves: 2M + 1 = 2(**3**) + 1 = **7**
* for **4 disks**, it will take 15 moves: 2M + 1 = 2(**7**) + 1 = **15** • for **5 disks**, it will take 31 moves: 2M + 1 = 2(**15**) + 1 = **31**
* for **6 disks**... ?

# Towers of Hanoi

**Number of Disks (n) Number of Moves**

1. 21 - 1 = 2 - 1 = 1
2. 22 - 1 = 4 - 1 = 3 3 23 - 1 = 8 - 1 = 7
3. 24 - 1 = 16 - 1 = 15
4. 25 - 1 = 32 - 1 = 31
5. 26 - 1 = 64 - 1 = 63

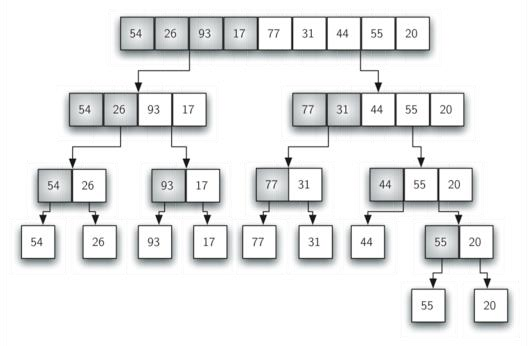
• So the formula for finding the number of steps it takes to transfer n disks from post A to post C is:

**2 n- 1**

# Merge Sort

• Merge sort is a divide and conquer strategy to improve the performance of sorting algorithms. Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item. If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a **merge**, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list.

Split Array



# Merge Array

