Data Structures

Nahian Salsabil (NNS)

Slides Adapted from Prantik Paul

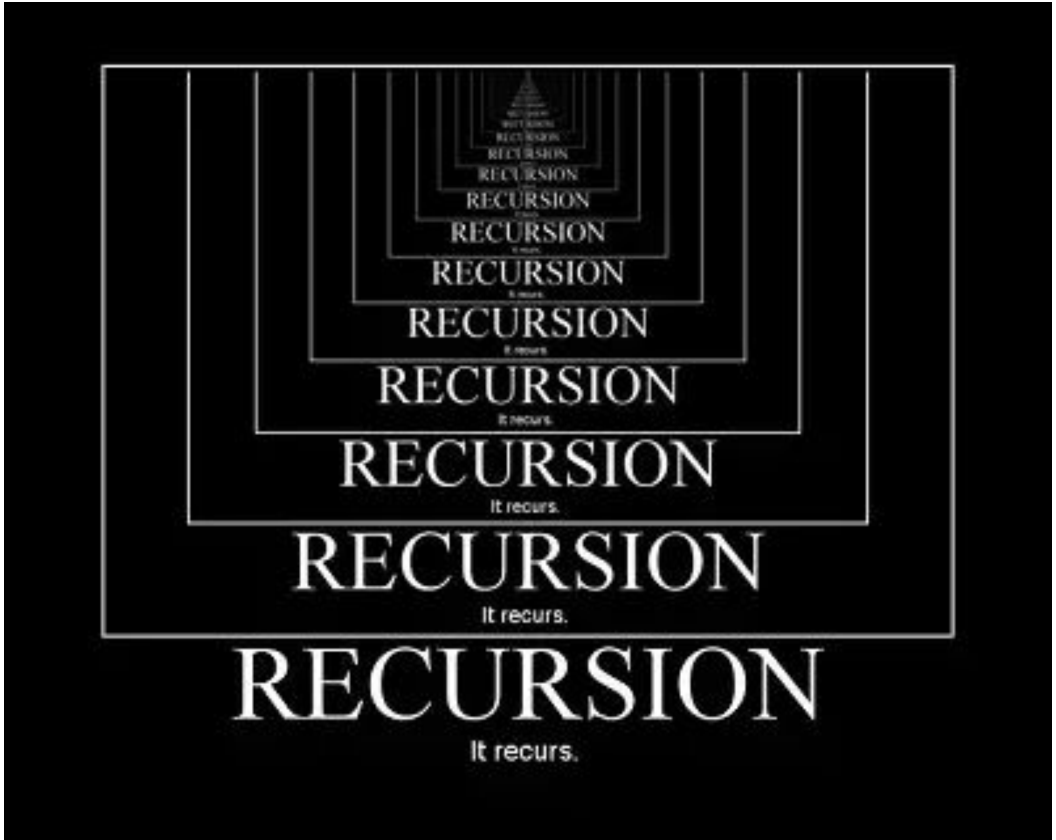
Topics Covered So Far

- Array
- Linked List
- Stack
- Queue

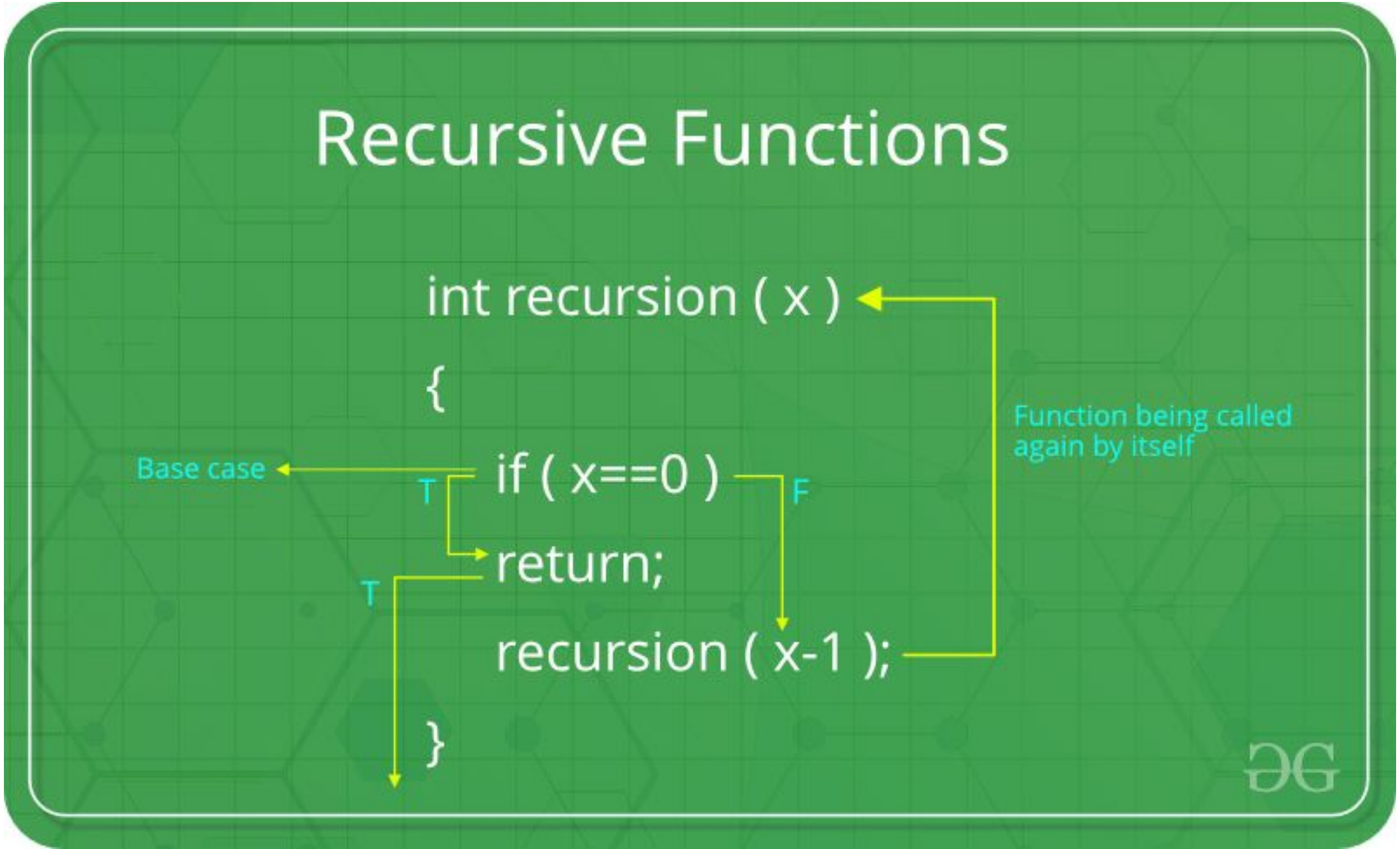
Outline

- Recursion (Basic)

Recursion


RECURSION
RECURSION
RECURSION
RECURSION
RECURSION
It recurs.
RECURSION
It recurs.
RECURSION
It recurs.
RECURSION
It recurs.

Recursive Definition



Recursive Definition (Factorial)

$$1! = 1$$
$$2! = 2 \times 1$$
$$3! = 3 \times 2 \times 1$$
$$4! = 4 \times 3 \times 2 \times 1$$
$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

...

and so on.

Recursive Definition (Factorial)

$$n! = \begin{cases} \boxed{1} & \text{if } n = \boxed{0} \\ 1 & \text{if } n = 1 \\ n \times (n-1)! & \text{if } n > 1 \end{cases}$$

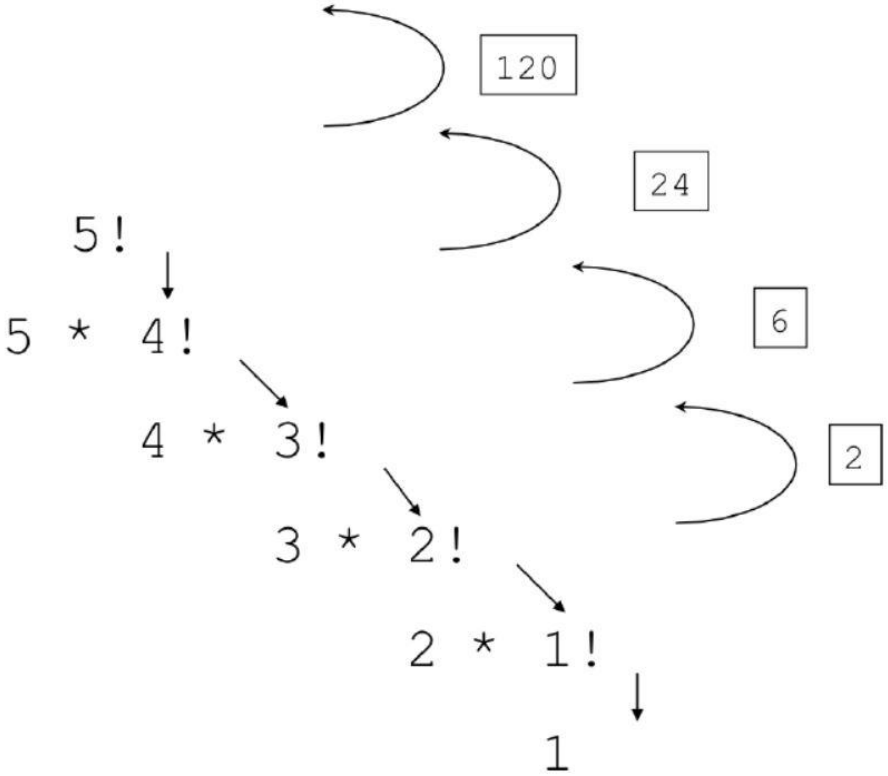
Recursive Definition (Factorial)

$$
\begin{array}{llllllllll}
5! & = & 4 & \times & 4! & & & & & \\
 & & & & 4 & \times & 3! & & & \\
 & & & & & & 3 & \times & 2! & \\
 & & & & & & & & 2 & \times\ 1! \\
 & & & & & & & & & 1
\end{array}
$$

Recursive Programming (Factorial)

```
def fact(n):
  if n==0 or n==1:
    return 1    #Base Case
  else:
    return n * fact(n-1)  #recursive part
```

Recursion Tree (Factorial)



Recursive Definition (Fibonacci)

```
          -
          | 0                                 if n = 0
          | 1                                 if n = 1
fib(n) =  |
          | fib(n-1) + fib(n-2)               n ≥ 2
          -
```

Recursion Tree (Fibonacci)

```
                 fib(5)
                   +
                /     \
               /       \
          fib(4)        fib(3)
            +             +
         /     \        /    \
        /       \      /      \
    fib(3)    fib(2) fib(2)  fib(1)
      +         +      +
```

Sum of numbers

```
def iterativeSum(head):
   sum=0
   temp=head
   while temp!=None:
      sum+=temp.element
      temp=temp.next
   return sum
```

Recursive Definition (Sum of numbers)

```
          -
          | k                  k is the only element
sum   =   |
          | k + sum(n.next)    otherwise
          -
```

Recursive Programming (Sum of numbers)

```
def recursiveSum(head):
  if head.next==None:
    return head.element  #Base case
  else:
    return head.element+recursiveSum(head.next)
```

Recursive Definition (Length of String)

```
             -
             | 0              if string s is empty
len(s)  =    |
             | 1 + len(rest) otherwise
             -
```

Recursive Programming (Length of String)

```
def strLength(str):
  if len(str)==0:
    return 0          #Base case
  else:
    return 1+strLength(str[1: ])
```

Recursive Definition (Length of Linked List)

```
             -
             | 0                 if l is an empty list
len(l)  =    |
             | 1 + len(rest) otherwise
             -
```

Recursive Programming (Length of Linked List)

```
def listLength(head):
  if head==None:   #empty linked list
    return 0       #base case
  else:
    return 1+listLength(head.next)  #Recursive part
```

Recursive Definition (Sequential Search LL)

```
                  -
                  | false               if list l is empty
                  | true                if l.item = k
contains(l,k)  =  |
                  | contains(l.next,k)  otherwise
                  -
```

Recursive Programming (Sequential Search LL)

```
def contains(head,key):
  if head==None:
    return False  #base case
  elif head.element==key:
    return True   #base case
  else:
    return contains(head.next,key)  #recursive part
```

Recursive Programming (Sequential Search Array)

```
def contains(arr,key):
  if len(arr)==0:
    return False    #base case
  elif arr[0]==key:
    return True     #base case
  else:
    return contains(arr[1: ],key)   #recursive part
```

Recursive Programming (Sequential Search Array)

- Array Slicing
- Inefficient
- To be Avoided

Solution

- Left Index
- Left = 0 => array begins at expected index 0
- Left = len(Array) - 1 => the array is the last element
- Left = len(Array) => 0-sized array

Recursive Definition (Seq Search : Left Index)

```
                  -
                  | false                          if l ≥ a.length
                  | true                           if a[l] = k
contains(a,l,k) = |
                  | return contains(a,l+1,k) otherwise
                  -
```

Recursive Programming (Seq Search : Left Index)

```
def contains(arr,left,key):
  if left>=len(arr):
    return False      #base case
  elif arr[left]==key:
    return True       #base case
  else:
    return contains(arr,left+1,key)   #recursive part
```

Recursive Programming (Find Maximum - Linear Linked List)

```
def maximum(a,b):
  return a if a>=b else b

def findMax(head):
  if head.next==None:
    return head.element    #base case
  else:
    maxRest=findMax(head.next)      #recursive part
    return maximum(head.element,maxRest)
```

Recursive Programming (Find Maximum - Linear Array)

```
def maximum(a,b):
  return a if a>=b else b

def findMax(arr, left):
  if left == len(arr)-1:
    return arr[left]     #base case
  else:
    maxRest=findMax(arr, left+1)     #recursive part
    return maximum(arr[left], maxRest)
```

Recursive Definition (Exponentiation)

$$a^n = \begin{cases} 1 & n = 0 \\ a \times a^{n-1} & n > 0 \end{cases}$$

Recursive Programming (Exponentiation)

```
def exp(a, n):
  if n==0:
    return 1     #base case
  else:
    return a * exp(a, n-1)     #recursive part
```

Recursive Definition (Binary Search in sorted array)

```
if the array is empty (if l > r that is):
  return false
else:
  Find the position of the middle element: mid = (l + r)/2
  If key == data[mid], then return true
  If key > data[mid], the search the right half data[mid+1..r]
  If key < data[mid], the search the left half data[l..mid-1]
```

Recursive Definition (Binary Search)

```
                 -
                 | false                if l > r
                 | true                 if k = a[mid]
contains(a,l,r,k) = |
                 | contains(a,mid+1,r,k) if k > a[mid]
                 | contains(a,l,mid-1,k) if k < a[mid]
                 -
```

Recursive Programming (Binary Search)

```
def contains(arr, left, right, key):
  if left > right:
    return False     #base case
  else:
    mid=(left+right)//2
    if key==arr[mid]:
      return True    #base case
    elif key > arr[mid]:
      return contains(arr, mid+1, right, key) #recursive part
    else:
      return contains(arr, left, mid-1, key)  #recursive part
```

Recursive Programming (Find Maximum - Binary)

```
def maximum(a,b):
  return a if a>=b else b

def findMax(arr, left, right):
  if left == right:
    return arr[left]     #base case
  else:
    mid = (left+right)//2
    maxLeftHalf=findMax(arr, left, mid)     #recursive part
    maxRightHalf=findMax(arr, mid+1, right) #recursive part
    return maximum(maxLeftHalf, maxRightHalf)
```