UDEMY

DSA

Abdul Bari

**Section 1: Before we Start**

**Introduction**

A program is a set of instructions which performs operations on data. The way we organize the data in the memory during execution of the program is called data structures.

* Physical Data Structures
  + Define how data is stored in memory
  + Arrays
  + Matrices
  + Linked List
* Logical Data Structures
  + Define how data can be utilized
  + Stack
  + Queues
  + Trees
  + Graph
  + Hashing

**Section 2: Essential C and C++ Concepts**

cxzz

**Section 3: Required Setup for Programming**

cxzz

**Section 4: Introduction**

Cxzz

**Section 5: Recursion**

**How Recursion Works**

return\_type fun(parameters) {

if(<base condition>) {

...

fun(parameters);

...

}

}

Recursive functions are traced in form of a tree.

Eg: void fun(int n) {

if(n>0) {

printf(“%d”,n);

fun(n-1);

}

}

fun(3)



Print: 3 fun(2)



Print: 2 fun(1)



Print: 1 fun(0)



return

Output: 3 2 1

Such tree representation is called a tracing tree of a recursive function.

Eg: void fun2(int n) {

if(n>0) {

fun(n-1);

printf(“%d”,n);

}

}

fun2(3)



Print: 3



fun2(2)



Print: 2



fun2(1)

Print: 1



fun2(0)



return

Output: 1 2 3

Recursion has two phases:

* Calling phase
* Returning phase

These phases can be observed by following the arrays in above recursion trees.

**Generalising Recursion**

return\_type fun(parameters) {

if(<base condition>) {

... <executed in calling phase/ ascending phase>

fun(parameters);

... <executed in returning phase/ descending phase>

}

}

Point to note:

Loops only have ascending phase while recursion has both ascending and descending phase.

**How Recursion Uses Stack**

Recursion uses stack to keep track of all the values of variables/ activation records for each call instance of a function.

During ascending phase new records are pushed into the stack and during descending phase the topmost record which is the last previous record is popped or deleted.

Recursive functions are memory consuming functions as they internally implement stack which consumes memory to keep records for each call.

**Recurrence Relation: Time complexity of Recursion**

T(n) = { 1 .. n = 0

{ T(n-1) + 1 .. n > 0

T(n) = T(n-1) + 1

= T(n-2) + 1 + 1

= T(n-3) + 1 + 2

= T(n-k) + k

= T(n-n) + n

= T(0) + n

= 1 + n

= n

So time complexity will be O(n).

**Static variables and Global Variables**

Static variables are created just once and the same copy is used by all the function calls.

int fun(int n) {

static int x = 0;

if(n>0) {

x++;

return fun(n-1)+x;

}

}

Instead of using static variable we could have also used global variable to get the same output.

X = 0 1 2 3 4 5



fun(5) = 25



fun(4) + 5 = 25



fun(3) + 5 = 20



fun(2) + 5 = 15



fun(1) + 5 = 10



fun(0) + 5 = 5



return

**Types of Recursion**

1. Tail Recursion
2. Head Recursion
3. Tree Recursion
4. Indirect Recursion
5. Nested Recursion

**Tail Recursion**

void fun(int n) {

if(n>0) {

...... statements

fun(n-1); // function is called at the end

}

}

In tail recursion there is no operation done in returning/ descending phase.

Tail recursion can be easily converted into iterative solution. From complexity analysis one will find that there is no difference in time complexity while space complexity of recursive solution is more in tail recursion so iterative solution is a better solution.

**Head Recursion**

void fun(int n) {

if(n>0) {

fun(n-1); // function is called in beginging

...... statements

}

}

Head recursion is harder to convert into iterative solution. There is some work left to do during descending phase.

**Tree Recursion**

void fun(int n) {

if(n>0) {

.......

fun(n-1);

.......

fun(n-1);

.......

}

}

fun(3)1



fun(2)2 fun(2)9



fun(1)3 fun(1)6 fun(1)10 fun(1)13



fun(0)4 fun(0)5 fun(0)7 fun(0)8 fun(0)11 fun(0)12 fun(0)14 fun(0)15

Recursion tree for above pseudo code. The subscripts tell the order of function call.

For above function we can see that the time complexity would be O(2n) while space complexity would be O(n) as we just need 4 stack height for n = 3.

**Indirect Recursion**

void funA(int n) {

if(condition) {

...

funB(n-1);

}

}

void funB(int n) {

if(condition) {

...

funA(n-1);

}

}

**Nested Recursion**

void fun(int n) {

if(condition) {

...

fun(fun(n-1));

}

}

**Sum of First N Natural Numbers**

sum(n) = n + sum(n-1) ,if n > 0

= n ,if n = 0

void sum(int n) {

if(n > 0) {

return ( n + sum(n-1));

}

Return n;

}

Also sum(n) = n \* (n+1) / 2;

**Factorial of a Number**

fact(n) = n \* fact(n-1) ,if n > 0

= 1 ,if n = 0

void fact(int n) {

if(n == 0) {

return 1;

}

return n \* fact(n-1);

}

Also, fact(n) = n \* n-1 \* n-2 \* . . . \* 1

**Section 6: Array Representation**

cxzz

**Section 7: Array ADT**

cxzz

**Section 8: Strings**

cxzz

**Section 9: Matrices**

cxzz

**Section 10: Sparse Matrix and Polynomial representation**

cxzz

**Section 11: Linked List**

cxzz

**Section 12: Sparse Matrix and Polynomial Representation using Linked List**

cxzz

**Section 13: Stacks**

cxzz

**Section 14: Queues**

cxzz

**Section 15: Trees**

cxzz

**Section 16: Binary Search Trees**

cxzz

**Section 17: AVL Trees**

cxzz

**Section 18: Search Trees**

cxzz

**Section 19: Heap**

cxzz

**Section 20: Sorting Techniques**

cxzz

**Section 21: Hashing Techniques**

cxzz

**Section 22: Graphs**

cxzz

**Section 23: Asymptotic Notations**

cxzz