LECTURE 2

General rules of determining algo run time :-

LOOPS : runtime of steps inside X no. of iterations of the loop.

So, say we have c primitive statements
the run time of a loop with n iteration
$= cn = O(n)$

NESTED LOOPS:

Take the inside out approach.
Total run time = product of sizes of all loops.

Ex:
```
for(i=1; i<=n; i++)
{   for(j=1; j<=n; j++)
    {
        c : primitive ops
    }
}
```
total no. $= c \times n \times n = cn^2 = O(n^2)$

CONSECUTIVE STEPS : Basically the sequential statements executed in a program

```
//constant time
x=x+1;                          -> c0
//executed n-times
for(i=1;i<=n;i++)
//constant time
m=m+2;                          -> c1 n
//outer loop executed n-times
for(i=1;i<=n;i++){
//inner loop executed n-times
for(j=1;j<=n;j++)
//constant time
k= k+1;                         -> c2 n^2
}
```
Total time $= c_0 + c_1n + c_2n^2 = O(n^2)$

Total time =
$c_0 + c_1 n + c_2 n^2$
$= O(n^2)$

IF-THEN-ELSE : For worst case scenario, we assume that the block with the most statements will get executed.

```
If(length()==0){
return false; // then part: constant
}
else {   //else part : (constant +constant)× n
    for(int n =0; n<length();n++){
    //another if: constant + constant (no else part)
        if(! List[n].equals(otherlist.list[n]))
        //constant
        return false;
        }
  }
```

In this example $\because$ else block is larger, we use that to do the calc.

C_0 – the comparison

C_1

$\left.\begin{matrix} C_2 \\ C_3 \end{matrix}\right\} \to (C_2+C_3)n$

Total time $= C_0 + C_1 + (C_2+C_3) \times n = O(n)$

Logarithmic function

Inverse of exponential function

if $a = b^c$

$\Rightarrow \log_b a = c$

Ex. $\log_{10} 100 = 2 \because 10^2 = 100$

$\log_2 8$ - - $\because 2^3 = 8$

Problem 1

```
void function (int n){
//outer loop executes n/2- times
for (i =n/2; i<=n; i++)
//middle loop executes n/2
for (j=1;j+n/2<=n;j=j++)
//loop execute logn times
for (k=1;k<=n;k=k*2)
count++;
}
```

Say $n = 20$

Loop 1 : $i = \frac{20}{2} = 10$; $10 <= 20$ ✓ ; $i{+}{+} \Rightarrow 11$

So this loop runs for value of i
$= 11, 12, 13, 14, 15, 16, 17, 18, 19, 20$

$\therefore$ for 10 times $= \frac{n}{2} = \frac{20}{2} = 10$.

Loop 2: $j=1$; $j + \frac{n}{2} \leq n$; $j{+}{+}$

iteration 1 $\Rightarrow 1 + \frac{20}{2} \leq 20$ now $j=2$
$11 \leq 20$ ✓

iteration 2
$\Rightarrow 2 + \frac{20}{2} = 12 \leq 20$ ✓ ; now $j=3$

⋮

iteration 10 $\Rightarrow 10 + \frac{20}{2} = 20 \leq 20$ ✓ ; now $j = 11$

next iteration will finish the loop
$\therefore$ this loop also ran $\frac{n}{2} = \frac{20}{2} = 10$ times.

Loop 3 : $K=1$; $K <= n$; $K = K*2$

iter 1	$1 <= 20$	; $K = 2$
iter 2	$2 \leq 20$	; $K = 4$
iter 3	$4 \leq 20$	; $K = 8$
iter 4	$8 \leq 20$	; $K = 16$
iter 5	$16 \leq 20$	; $K = 32$
iter 6	$32 \leq 20$ X	→ iteration finished

let's say loop ran 2^x

$20 = 2^x$

$\Rightarrow \log_2 20 = 4.3$ approx 5

$\therefore$ we can say this loop runs $\log n$ times.

Problem 3

```
function (int n){
        //constant time
        if(n==1) return;
        //outer loop execute n times
        for (int i =1;i<=n;i++){
        // inner loop executes only time due to break statement
                for (int j =1;j<=n;j++){
                        printf("*");
                        break;
                }
        }
}
```

Time complexity of the above function is?

$\rightarrow C_0$

$\rightarrow n$

$\rightarrow n$

$\rightarrow C_1 n^2$

Total time = $C_0 + C_1 n^2 = O(n^2)$

- Linear complexity $\Rightarrow O(n)$
- Quadratic complexity $\Rightarrow O(n^2)$
- Cubic complexity $\Rightarrow O(n^3)$
- Logarithmic complexity $\Rightarrow O(\log_2 n)$

Recursive Algo and its performance measurement.

A recursive algo largely looks as below:

→ if problem can be solved by a primitive op.
do it directly

→ else

- break the problem into smaller sub-problems
- solve each of them by recursive calls to your main program
- combine all the solutions of smaller problems to get the final answer.

Ex- recursive algo to find length of string

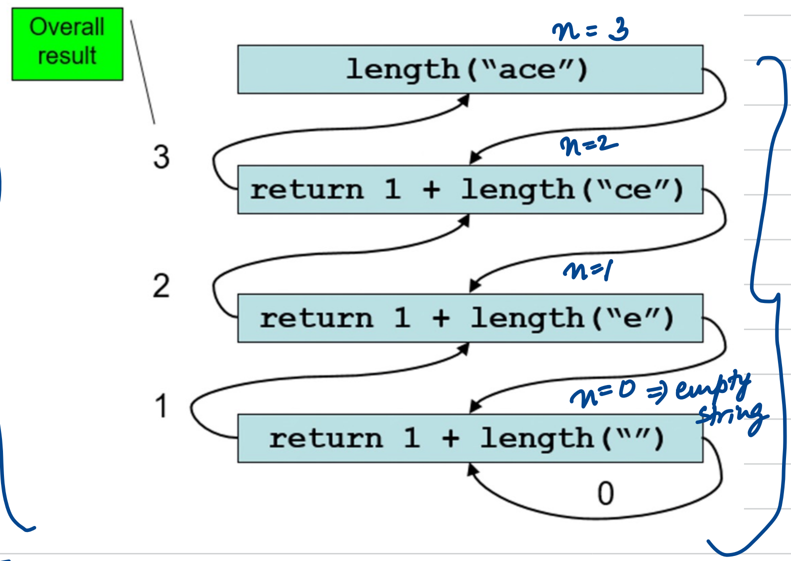
```
Proc    StringLength (str[n])
{
    if str == empty then
        return 0
    else
     {
        n = n-1
        return 1+ StringLength (str[n])
     }
}
```

recursive calls.

Tracing the Recursive Calls :-

return of soln. from the calls made

direction in which soln. are provided.



calls being made to itself

direction of call

Recursive Algo Exercises

1. Print n^{th} Fibonacci No.

```
proc fibonaccin (n)
{
    if n == 0 then
        return Null
    else if n == 1 then
        return 1
    else if n == 2 then
        return 1
    else
        return fibonaccin(n-1) + fibonaccin(n-2)
```

2. Power, given two nos. x^y

```
proc Power (x, y)
{
    if y == 0 then
        return 1
    else if y == 1 then
        return x
    else
        return x * power (x, y-1)
```

3. Recursive array search.

```
proc    arrSearch (arr, x, i)
{
    if arr == empty then
        return null
    else if arr[i] == x then
        return i
    else
        return (arr, x, i-1)
}
```

Recurrence Relations

The way we calculate the runtime function of a recursive Algo is a little bit different. This is called Recurrence

It's an equation/inequality defined in terms of smaller inputs.

Ex:

$$S(n) = \begin{cases} 0 & n = 0 \\ c + S(n-1) & n > 0 \end{cases}$$

Methods to solve the recurrence :-

1. Substitution method
2. Recursion-tree method
3. Master Theorem.

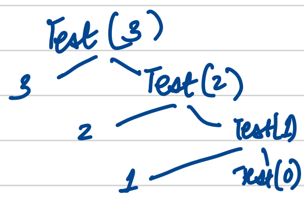
Recursion-tree method

```
void test(int n)
{
    if(n>0)
    {
    printf("%d",n);
    test(n-1);
    }
}
```

→ Sample recursion program

$\because$ branches $= n$

$\therefore O(n)$



But as the program gets complicated, it gets difficult to depict the solution on a tree form. Especially if n is large

Substitution Method

Once you have the recurrence, you substitute the input (n) with the next input in a series, till you reach the size of input (n) where no more recursion is possible and we get a constant run time.

Ex :-

```
void test(int n)
{
    if(n>0)
    {
    printf("%d",n);
    test(n-1);
    }
}
```

Recurrence for given prog:

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$$

→ because if $n=0$, then only the 'if' condition is evaluated

→ because if $n>0$, we do a print $= 1$ & then we call test() again

Now, how to solve them:-

in eqs. $T(n-1)+1$, what is $T(n-1)$?

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

$\Rightarrow T(n-1) = T((n-1)-1)+1$ → substituting n with n-1 wherever n is available in the eqs.

$= T(n-2)+1$

Similarly $T(n-2) = T((n-2)-1)+1$

$= T(n-3)+1$

$T(n-3) = T(n-4)+1$

So now,

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

$T(n) = T(n-2)+2 \quad \because T(n-1) = T(n-2)+1$ → basically $T(n)$ can be defined in terms of $T(n-1)+1$, but also $T(n-2)+2$, $T(n-3)+3$.

$T(n) = T(n-3)+3$

$T(n) = T(n-4)+4$

$\vdots$

$T(n) = T(n-k)+k$ where $n-k=0$ or $n=k$

in such case $T(n-k) = T(0)$ & we can say,

$T(n) = T(n-n)+n$

$= T(0)+n$

$= 1+n = O(n)$ its' a linear case

Problem 1

$$T(n) = \begin{cases} O(1) & n=1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

$O(1) \Rightarrow$ constant time

$O(n) \Rightarrow$ linear time, so we can write the recurrence as

$$T(n) = \begin{cases} c & n=1;\ c \text{ is a constant} \\ 2T(n/2)+n & n>1 \end{cases}$$

Now in this recurrence, n isn't getting reduced by 1, but its getting divided by 2, So our series =

$n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \frac{n}{16} \cdots \frac{n}{n}=1$

$T(n/2) = 2T\left(\frac{n/2}{2}\right) + \frac{n}{2} = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$

$T\left(\frac{n}{4}\right) = T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$

$\therefore \quad T(n) = 2T\left(\frac{n}{2}\right) + n$

$= 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n \quad = \underbrace{2^2 T\left(\frac{n}{2^2}\right)} + 2n$

$= 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$

$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$

$\vdots$

$= 2^k T\left(\frac{n}{2^k}\right) + kn$

$= 2^{\log_2 n} T(1) + (\log_2 n \times n)$

$= (n \times 1) + n \log_2 n$

$= n \log_2 n + n$

$\therefore O(n \log_2 n)$

Say we have reached the smallest sub problem with the last expr.

$\Rightarrow T\left(\frac{n}{2^k}\right) = T(1)$

$\Rightarrow \frac{n}{2^k} = 1$

$\Rightarrow n = 2^k$

$\Rightarrow k = \log_2 n$

Problem 2

$$T(n) = \begin{cases} 0 & n = 0 \\ T(n-1) + 2n - 1 & n > 0 \end{cases}$$

$T(n) = T(n-1) + 2n - 1 \quad \Longrightarrow \quad n \to n-1 \quad = T((n-1)-1) + 2(n-1) - 1$

$= T(n-2) + 2n - 2 - 1$

$= T(n-2) + 2n - 3$

$T(n) = [T(n-2) + 2n - 3] + 2n - 1$

$= T(n-2) + 4n - 4$

$T(n) = [T(n-3) + 2n - 5] + 4n - 4 \quad\quad n \to n-2 \quad = T(n-2-1) + 2(n-2) - 1$

$= T(n-3) + 6n - 9 \quad\quad = T(n-3) + 2n - 5$

$T(n) = [T(n-4) + 2n - 7] + 6n - 9 \quad\quad n \to n-3 \quad = T(n-4) + 2n - 7$

$= T(n-4) + 8n - 16 \quad\quad n \to n-4 \quad = T(n-5) + 2n - 9$

$\vdots$

$T(n) = T(n-k) + 2kn - k^2$ Lets' say we reached $n-k=0$ or $n=k$

$= T(0) + 2n \cdot n - n^2$

$= 0 + 2n^2 - n^2$

$= n^2 \quad\quad \Rightarrow O(n^2)$

Master Theorum

we determine running time of algo (divide and conquer algo) based on asymptotic notations.

Say, $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$

n = size of problem
n/b = size of each sub-problem
$f(n)$ = cost of work done outside of recursive call

Then it can be written as

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + \theta\left(n^k \log^p n\right)}$$

where $a \geq 1$, $b > 1$, $k \geq 0$ & p is a real number.

We compare a to b^k, to solve recurrences using Master Theorum.

Case-1

If $a > b^k$, then $T(n) = \theta(n^{\log_b a})$

Case-2

If $a = b^k$ and

- If $p < -1$, then $T(n) = \theta(n^{\log_b a})$
- If $p = -1$, then $T(n) = \theta(n^{\log_b a}.\log^2 n)$
- If $p > -1$, then $T(n) = \theta(n^{\log_b a}.\log^{p+1} n)$

Case-3

If $a < b^k$ and

- If $p < 0$, then $T(n) = O(n^k)$
- If $p >= 0$, then $T(n) = \theta(n^k \log^p n)$

Ex 1: $T(n) = 3T(n/2) + n^2$ $\Rightarrow$ $a = 3$ $b = 2$ $k = 2$ $p = 0$

$a \ngtr b^k \Rightarrow 3 < 2^2$

$\therefore$ case 3. & $p = 0$

$$T(n) = \theta(n^2 \log^0 n)$$
$$= \theta(n^2)$$

Ex 2: $T(n) = 2T(n/2) + n$ $\Rightarrow$ $a = 2$ $b = 2$ $k = 1$ $p = 0$

$2 = 2^1$ $\therefore a = b^k$ So, case 2, $p > -1$

$$\therefore T(n) = \theta(n^{\log_b a} . \log^{p+1} n)$$
$$= \theta(n \log n)$$

— END