#1

1. What is Recursion
2. Example of Recursion
3. Tracing Recursion.
4. Stack used in Recursion
5. Time Complexity
6. Recurrence Relation

Date ______
Page ______

Recursion

```
int
void fun( int n )
{
    1. ____
    2. ____
    3. ____
}
int main( )
{
    1. ____
    2. ____
    3. fun(x) * 2 ;
    4. ____
    5. ____
}
```

(10 written above fun(x); arrows numbered 1, 2, 3, 4, 5 show the flow of control)

We have a main func & a fun called func
How the func call is made (How it works)?

When the prog executes it will
Start from main func execute 1 & 2
then on 3 it will go to fun, execute
all the lines fun ends &
the control come back to
the same line (3rd line) if
any more op present in this [Ex: func(x) * 2]
line they will execute other
wise they will exe 4th & 5th
line.

for ex: the fun is returning something. (Assume fun returns 10)

*2 will be performed once the control comes back to same line.

(general form of recursion)

If a function is calling itself that is called as Recursive func

```
Type fun( param )
{
    if ( < base condition > )
    {
        1. ____
        2. fun( param );
        3. ____
    }
}
```

There must be some base condⁿ that will terminate recursion (There must be some method to terminate otherwise it will go into ∞ calling)

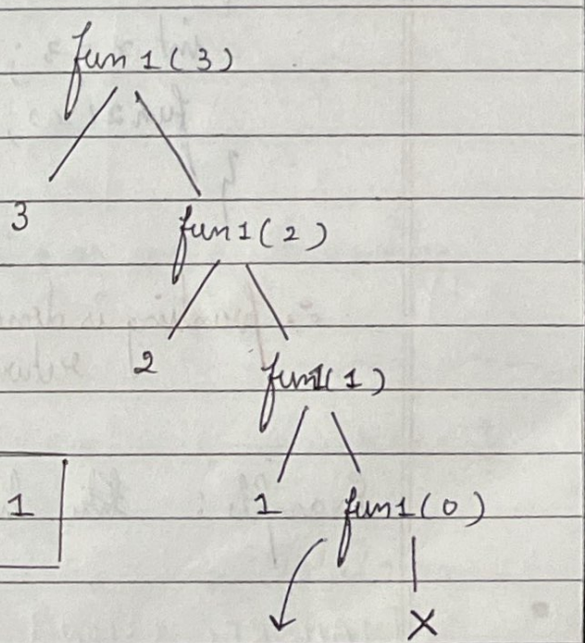
Example:1 (tail recursion) → Ex: printing & then calling itself

```
void fun1(int n)
{
  if(n>0)
  {
1.  printf("%d", n);
2.  fun1(n-1);
  }
}

void main()
{
  int x=3;
  fun1(x);
}
```

Traced in the form of a tree.

↑

Tracing of recursive func.



O/P: 3 2 1

∴ printing was done at calling time

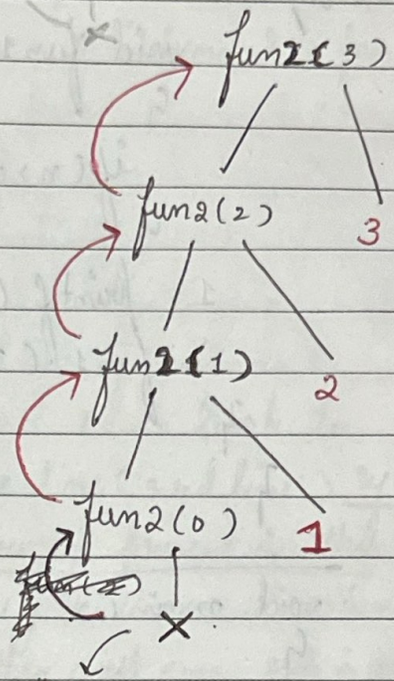
from this call it will go back to previous call & then & come out of the function prev call & so on.

Ex: call is made & then printing is done.
first

(Head recursion)

Example:

```
void fun2(int n)
{
    if(n>0)
    {
        fun2(n-1);
        printf("%d",n);
    }
}

void main()
{
    int x = 3;
    fun2(x);
}
```



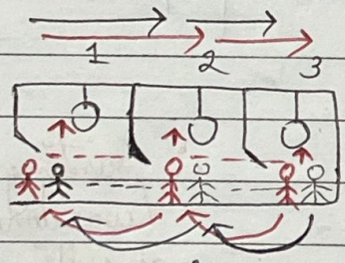
after this control will go back to prev call.

Second statement will be done after 1st sta.. has finished.

∴ printing is done at returning time

O/P: 1 2 3

Example: Bulb door room. [Analogy of Head & tail Rec..]



[TAIL RECURSION]

recursively.

first perform O/P then recursive call.

recursive call → 1. switch on Bulb
2. go to next room.

ascending/
1st stage: calling
2nd stage: returning/ descending.

recursion has two phases.

O/P: 1 2 3

[HEAD RECURSION]

first recursive call then perform operation.

→ 1. go to next room.
2. switch on bulb.

O/P: 3 2 1

recursive function are like a rubber band. → if you stretch it, it will be next, next, next..... & if we release it will come back [pull it]

#2

```
void fun(int n)
{
    if(n>0)
    {
        statement will
Ascending  1. Execute at calling time
           2. fun(n-1)  # 2
descending 3. Returning time
    }
}
```

→ returning time too.

Before func call if anything is there that will be executed at calling time

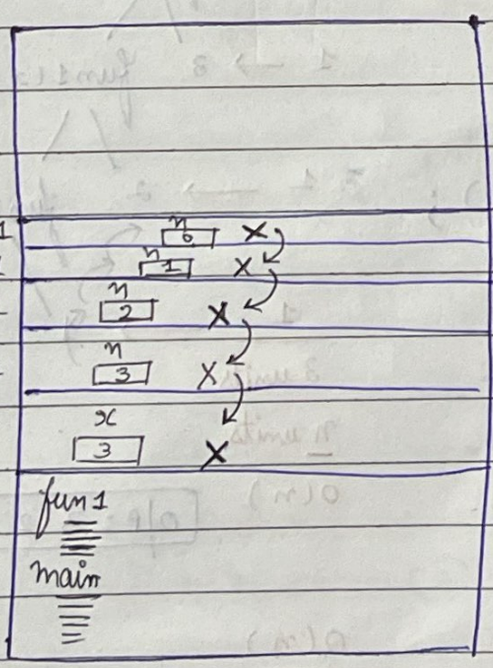
After func call if anything is there that will be executed at returning time.

difference loop vs recursion

- loop → only has ascending phase.
- recursion → ascending as well as descending phase!

#3 How Recursion Uses Stack

Example 1 [see lecture 1]



Size of stack:
4 activation rec.
size = 4.
mem consumed: n = 4
so 4 times.

[Leaving main func, activation rec of the fun 1 size of the mem is 4]

consumed = 4 x ↓ size of var int is.

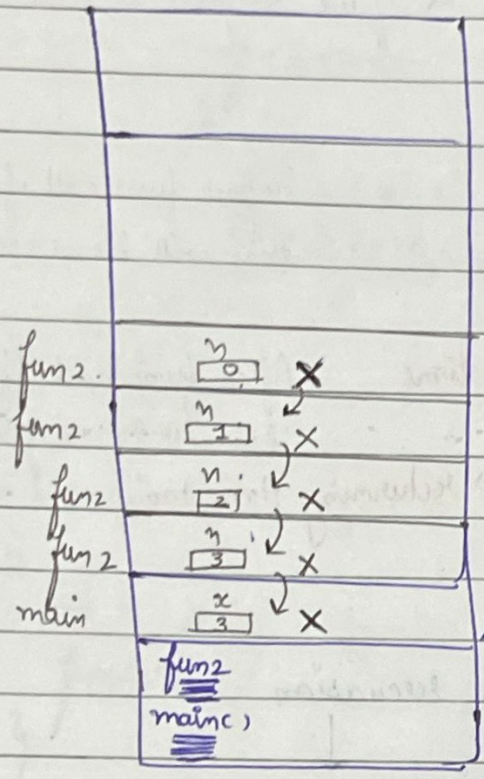
[once a call ends the activation rec for that call is deleted (X)] → control goes back to previous call.

Total mem consumed by this is O(n)

for n there will be n+1 calls activation record n+1

[act rec depends on no of calls made]

for example 2: [see lecture 1 for example code 2 used to demonstrate this]

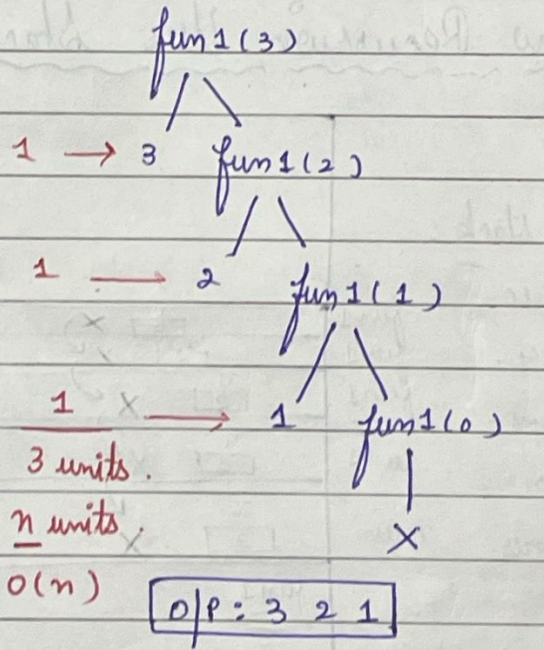


when n = 0 → its not n > 0 ∴ it comes out of the func & the act rec is deleted goes back to previous call → value of n = 1 ∴ print 1 → act rec deleted → go to previous call → and so on....

[The topmost value in act rec will store the value of n that we need to print as we are descending] (during returning) [that means All the values of n are there in stack at diff act recs & these values are utilized at returning time]

#4 We assume that every statement takes one unit of time.

```
void fun1(int n)
{
  if(n>0)
  {
    printf("%d", n);
    fun1(n-1);
  }
}

void main()
{
  int x = 3;
  fun1(x);
}
```



1 / 3 units.

n units.

O(n)

O/P: 3 2 1

O(n)

Time Complexity of recursive func using recurrence relation

assume time taken by that func is.

T(n) ← void func1(int n)
{
1 ← if(n>0)
{
1 ← printf("%d", n);
X 1 ← func1(n-1); → it does not take 1 unit of time its a func call, we should know total time taken by that func call.
T(n-1) ✓ }
}

function call

if time taken by that func1 is T(n) then that total time should be sum of all the times taken by the statements inside.

if func1(int t) → T(n)
then func1(n-1) → T(n-1) time.

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

when n = 0 it will just check the cond^n & not enter inside & it will come out ∴ 1 unit of time

recurrence relation is.

↗ just checking the condition

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

→ if we have any const value here we write it as 1.

Solving through induction method / Successive substitution method.

$T(n) = T(n-1) + 1$ —— (1)

∵ $T(n) = T(n-1) + 1$

∴ $T(n-1) = T(n-2) + 1$.

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

$T(n) = T(n-2) + 2$ —— (2)

$T(n-2) \rightarrow T(n-3)+1$

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

$T(n) = T(n-3) + 3$ —— (3)

$T(n) = T(n-k) + k$ — (4)

Assume $n - k = 0$.

$\therefore n = k$

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

$T(n) = T(0) + n$

$T(n) = 1 + n$. $\longrightarrow O(n)$

```cpp
// # 5 - Let's code Recursion

#include <iostream>
using namespace std;

// tail Recursiom

void tail(int n){

    if(n>0){
        cout<<n;
        tail(n-1);
    }
}

// Head Recursion

void head(int n){

    if(n>0){
        head(n-1);
        cout<<n;

    }
}


int main(){
    int x = 3;
    tail(x);
    cout<<endl;
    head(x);

    return 0;
}
```

$T(n) = T(n-k) + k$ — ④

Assume $n-k = 0$.

$\therefore n = k$

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

$T(n) = T(0) + n$

$T(n) = 1 + n$. ⟶ $O(n)$

#6 Static variables in Recursion

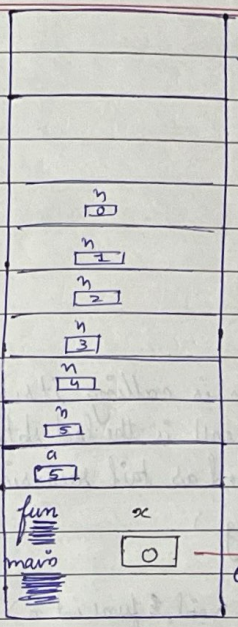
```
int x = 0;                          <- like a global var
int fun(int n)
{
  static int x = 0;                 -> if we take out this var & declare it outside
  if(n>0)
  {
    x++;
    return fun(n-1)+n;              | return fun(n-1)+x
  }
  return 0;
}
main()
{
  int a=5;
  printf("%d", fun(a));
}
```

fun(5) = 15
fun(4) + 5 = 15
fun(3) + 4 = 10
fun(2) + 3 = 6
fun(1) + 2 = 3
fun(0) + 1 = 1
0

result will be same

```
int r;
r = fun(5);
printf("%d\n", r);
r = fun(5);
printf("%d\n", r);
return 0;
```

now if we add this segment twice then the first will make the global/static var x=5 & print result 25 but the second will x=5+10 & print the result 50



~~for static~~ Static var will not be created everytime the func is called, it is created only one time that is at the loading time of a program.

So this x will have a single copy unlike n it won't have multiple copies created.

∴ All the calls of the func will use same copy of x.

static & global variable will be created inside code section will have a single copy unlike normal var.

for static var

for static / int x=0; (global)

global var will also have a single copy used by all func calls & also created inside code section

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
0

```
int fun(int n)
{
  static int x=0;
  if(n>0)
  {
    x++;
    return fun(n-1)+x;
  }
  return 0;
}
```

```cpp
// #7 - Let's code Static & Global in Recursion

#include <iostream>
using namespace std;

// int x = 0; // global variable

int fun(int n){

    if(n>0){
        return fun(n-1)+n;
    }
    return 0;
}

// static var example

int staticfun(int n){

    static int x=0;
    if(n>0){
        x++;
        return staticfun(n-1)+x;
    }
    return 0;
}

int main(){
    int r,s;
    r = fun(5);
    cout<<r<<endl;


    s = staticfun(5);
    cout<<s<<endl;

    s = staticfun(5);
    cout<<s<<endl;

    return 0;
}
```

#8

Date
Page

Types of Recursion

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

Tail Recursion → If a recursive func is calling itself & that recursive call & that call is the last statement in a func then it is called as tail recursion (after that call there is nothing)

```
fun(n)
{
  if(n>0)
  {
    ≡≡≡≡≡
    fun(n-1)+n;
  }
}
```

this will be performed at returning time

hence these types can't be tail recursion.

TAIL RECURSION

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

```
fun(3);
```

↳ This means that all the operations will be performed at calling time only & func will not performing any operation at returning time

Every recursive func can be written as a loop or vice versa.

Date
Page

loop vs tail recursion.

```
void fun(int n)
{
  while(n>0)
  {
    printf("%d", n);
    n--;
  }
}

fun(3);
```

← loop

⟷

Recursion ↓

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

fun(3);
```

Tail recursion can be easily converted in the form of loop.

In terms of time → O(n) → same (tail & loop)

Space → for n → space → O(n) → recursion
(activation record in a stack for 3 i.e. 4 i.e. n+1)

for loop

Activation record created = 1

∴ space — O(1)

	loop	recursion
TC →	O(n)	O(n)
SC →	O(1)	O(n)

In terms of space loop is more efficient than tail recursion.

Some compilers & their code optimisation inside compiler will check for func written as a tail recursion, they they will try to convert it into a loop, it means they will try to reduce the space consumption & they will utilise only O(1) space. So our func will be converted into object code just like loop where space is reduced.

So in conclusion, if we have to write a tail recursion we better convert it into a loop, its more space efficient but it may not be true for every kind of recursion or loop.
In case of tail recursion loop is efficient.

#9

Head Recursion → The first statement inside the func, is recursive call & all the processing it has to do is after wards. (not a single statement before func call)

→ ~~void fun~~ The func doesn't have to perform operation at the time of calling. It has to do everything only at the time of returning

```
void fun(int n)
{
    if(n>0)
    {
        fun(n-1);
    }
}
```

This is head recursion (as nothing is there before the func call)

if something is there before the func call its a recursion we don't have to give any special name for that one

```
void fun(int n)
{
    if(n>0)
    {
        fun(n-1);
    }
}
```

∴ it is not a head recursion

head recursions or a if a recursive func has to do something at returning time it cannot be easily conv. to in the form of loop but it can be converted it doesn't look as it is.

head recursion vs loop

loop

```
void fun(int n)
{
    int i=1;
    while(i<=n)
    {
        printf("%d", i);
        i++;
    }
}
```

recursion

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

won't give the same output as head recursion (red lines says how to convert to head recursion)

o/p: 1 2 3

fun(3);

o/p: 1 2 3

fun(3);

↳ looking at func if you try to convert its not easy!

(if we try to convert as it is like we did with tail recursion (Blue codes) it won't give same output as Head recursion)

#10

Tree recursions

Linear recursion

```
fun(n)
{
  if(n>0)
  {
    fun(n-1);
  }
}
```

↳ calling it self only one time

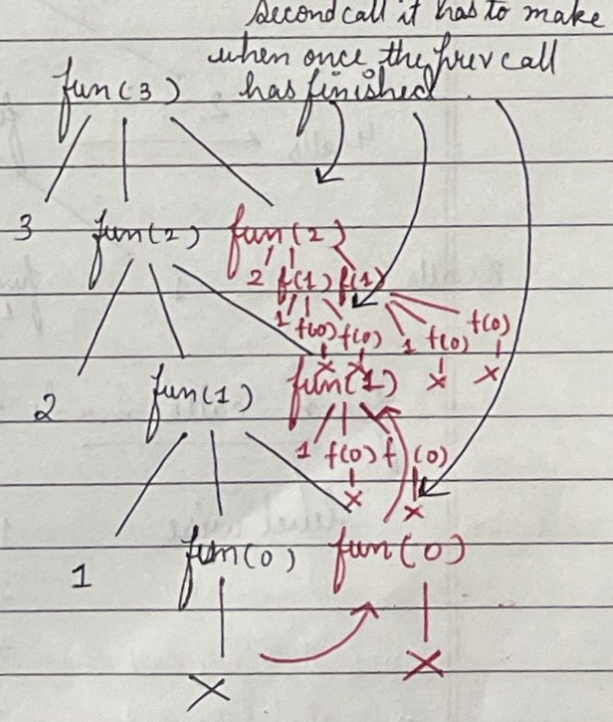
Tree Recursion

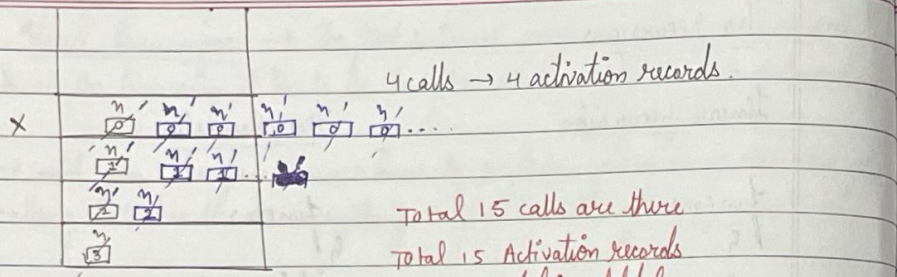
```
fun(n)
{
  if(n>0)
  {
    fun(n-1);
    fun(n-1);
  }
}
```

↳ if a recursive func calling itself more than one time

Tree recursion

```
void fun(int n)
{
  if(n>0)
  {
1.  printf("%d",n);
2.  fun(n-1);
3.  fun(n-1);
  }
}

fun(3);
```



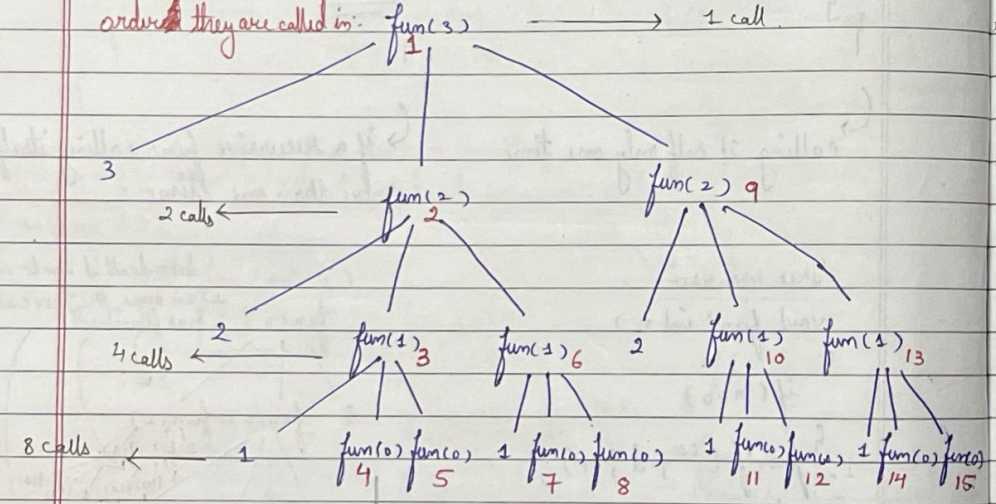


4 calls → 4 activation records.

Total 15 calls are there
Total 15 Activation records are created & deleted.

o/p : 3, 2, 1, 1, 2, 1, 1

drawing the tree (neatly) again



for n calls made:

The red nos denote in what order the calls are made.

level wise
1 → 1 call
2 → 2 calls
3 → 4 calls
4 → 8 calls.

once the func call is finished the act rec for it is deleted & it goes to previous func call

The second call of each function call will be made when first call is finished

Ex: fun(3)

3 fun(2) This call will be made after the first call fun(2) is finished.

for 3 there are 4 levels if n = 4 → 5 levels.

1 + 2 + 4 + 8 = 15

$2^0 + 2^1 + 2^2 + 2^3$

Summation of GP terms.

$= 2^{3+1} - 1$

for n, $2^{n+1} - 1$ calls will be made

$2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$ calls made

$= O(2^n)$

↳ total 2^n calls are made

for any value of n, the no of call will be $O(2^n)$

Time – $O(2^n)$ ✓
Space – $O(n)$

(depends on max height of stack
(space = height of the tree)
if 3 then 4 will be the space
if n then n+1 " " " "
∴ $O(n)$ → SC.

for this ex, TC was $O(2^n)$ it depends on func. what TC we will be getting. Its not that every true recursion will have TC as 2^n for this func its 2^n.

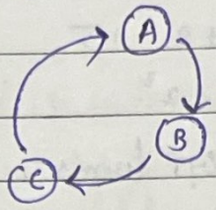
Total Act rec depends on no of calls.
but how much space it was occupying inside the stack?
we have observed that same space was reused inside the stack, one ACT rec was gone and in the same place another ACT rec was created, so we don't require much space just like 2^n but we require space equal to height of the tree we got here.
∴ max stack size we need is n.

```cpp
// #11 - Let's code Tree Recursion

#include <iostream>
using namespace std;

void fun(int n){
    if(n>0){
        cout<<n;
        fun(n-1);
        fun(n-1);
    }
}

int main(){
    fun(3);
    return 0;
}
```

#12

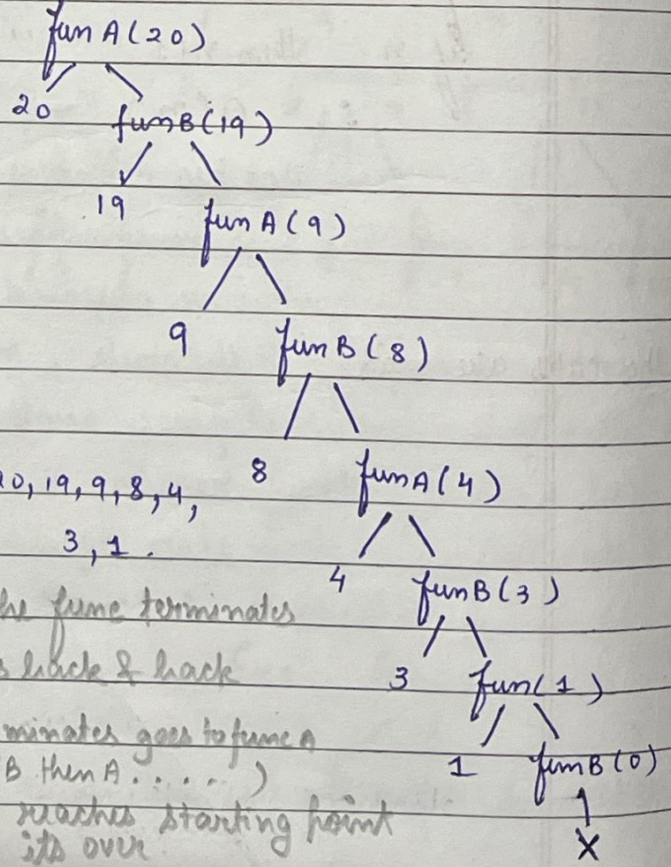
Indirect Recursion → more than one func & they are calling one another in a circular fashion.

```
void A(int n)
{
  if(<——>)
  {
    =
  B(n-1);
  }
}
void B(int n)
{
  if(<——>)
  {
    ≡
  → A(n-3);
  }
}
```



Indirect recursion

```
void funA(int n)
{
  if(n>0)
1. {printf("%d",n);
2. funB(n-1);
}
void funB(int n)
{
  if(n>1)
1. {printf("%d",n);
2. funA(n/2);
}
}
```



O/P: 20, 19, 9, 8, 4, 3, 1.

once the func terminates it goes back & back (B terminates goes to func A then B then A.....) until it reaches starting point & then its over

```cpp
// #13 - Let's code Indirect Recursion

#include <iostream>
using namespace std;

void funB(int n); // before using a function it must be either declared or defined so as we are defining funcB later and using it before in funA therefore we are declaring it before funA to avoid error.

void funA(int n){
    if(n>0){
        cout<<n<<' ';
        funB(n-1);
    }
}

void funB(int n){
    if(n>1){
        cout<<n<<' ';
        funA(n/2);
    }
}


int main(){

    funA(20);
    return 0;
}
```

#14

Nested Recursion → if recursive func will pass parameter as a recursive call.

```
void fun(int n)
{
  if(___)
  {
    ___
    fun(fun(n-1));
  }
}
```

→ unless the result of this recursive call is obtained, this call cannot be made.

A recursive call is taking a recursive call as parameter. basically recursion inside recursion is nested recursion.

```
int fun(int n)
{
  if(n>100)
    return n-10;
  else
    return fun(fun(n+11));
}
```

fun(95)

fun(95) = 91

96 = fun(106) (106-10) (if n>100) return n-10; 106>100 return 106-10 = 96

fun(fun(95+11)) → 106 → 96

fun(96)

97 = fun(107) (if n>100) return n-10; 107>100 return 107-10 = 97

fun(fun(96+11)) → 107 → 97

fun(97)

98 = fun(108) (if n>100) return n-10; 108>100 return 108-10 = 98

fun(fun(108)) → 98

fun(98)

99 = fun(109) (if n>100) return n-10; 109>100 return 109-10 = 99

fun(fun(109)) → 99

fun(99)

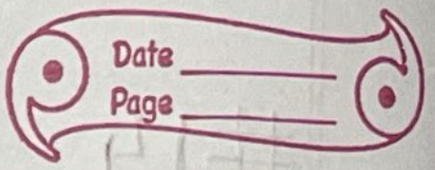
100 = fun(110) (if n>100) return n-10; 110>100 return 110-10 = 100

fun(fun(110)) → 100

fun(100)

100 > 100

∴ enter else part



101 = fun(111)

fun (fun(111)) > 100 if (n > 100) return n - 10

111 > 100 return 111 - 10

101

fun (101) if (n > 100) return n - 10

101 > 100 return 101 - 10

91

~~fun~~ 91

it will now go back & back & finally the result of

fun(95) = 91

X X X

```cpp
// #15 - Let's code Nested Recursion

#include <iostream>
using namespace std;

int fun(int n){
    if(n>100){
        return n-10;
    }
    else{
        return fun(fun(n+11));
    }
}

int main(){

    int r;
    r = fun(95);
    cout<<r;

    return 0;
}
```

#16 Sum of First 'n' Natural Number

1+2+3+4+5+6+7

1+2+3+4+5+.....+n.

$$Sum(n) = \underbrace{1+2+3+4+\cdots+(n-1)}+n$$

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

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

```
int sum( int n )
{
    if (n==0)
    return 0;
else
    return sum(n-1) + n;
}
```

$\frac{n(n+1)}{2}$ → formula for sum of first 'n' natural number.

using formula

```
int Sum(int n)
{
  return n*(n+1)/2;
}
```

O(1)

---

using loop.

```
int Sum(int n)
{
  int i, s=0;          ——— 1
for(i=1; i<=n; i++){   ——— n+1
    s=s+i;             ——— (n).
  return s;            ——— 1
}
```

O(n)

recursive func internally uses stack → so costly to use that method.
takes more space but
time → O(n) → due to act rec generated in stack

```
Sum(5) = 15
  |
Sum(4) + 5 = 10 + 5 = 15
  |
Sum(3) + 4 = 6 + 4 = 10
  |
Sum(2) + 3 = 3 + 3 = 6
  |
Sum(1) + 2 = 1 + 2 = 3
  |
Sum(0) + 1 = 0 + 1
  0
```

n+1 call
time → O(n)
space → O(n)
(n+1)

```
int main()
{
  int r;
  r = Sum(5);
printf("%d", r);
return 0;
}
```

```cpp
// #17 -Let's code Sum of N natural Numbers Using Recursion

#include <iostream>
using namespace std;

// using recursion
int sumR(int n){
    if(n==0){
        return 0;
    }
    else{
        return sumR(n-1) + n;
    }
}

// using formula
int sumF(int n){
    return (n * (n+1))/2;
}

// using loop(iterative)
int sumL(int n){
    int i,s=0;
    for (i = 1; i<=n;i++){
        s=s+i;
    }
    return s;
}


int main(){
    int r;
    r=5;
    cout<<sumR(r)<<' '<<sumF(r)<<' '<<sumL(r);

    return 0;
}
ut)
```

#18

Factorial of a Number

$n! = 1 * 2 * 3 * \cdots * n$

$5! = 1 * 2 * 3 * 4 * 5 = 120$.

$0! = 1$

$1! = 1$

$fact(n) = 1 * 2 * 3 * \cdots * (n-1) * n$

$fact(n) = fact(n-1) * n$

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

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

loop

```
int Ifact (int n)
{
  int f=1;
  int i;
  for(i=1; i<=n; i++)
    f=f*i;
  return f;
}
```

If we use a negative no it will go into infinite calling & terminate (due to stack overflow)

```
int main()
{
  int r;
  r=fact(-1);
  printf("%d", r);
  return 0;
}
```

```cpp
// #19 -Let's code Factorial Using
Recursion

#include <iostream>
using namespace std;

// using recursion
int factR(int n){
    if(n==0){
        return 1;
    }
    else{
        return factR(n-1)*n ;
    }
}

// using loop(iterative)
int FactL(int n){
    int i,f=1;
    for (i=1; i<=n;i++){
        f=f*i;
    }
    return f;
}


int main(){
    int r;
    r=5;
    cout<<factR(r)<<' '<<FactL(r);

    return 0;
}
```

#20

Power using recursion

Exponent (m^n)

$2^5 = 2 * 2 * 2 * 2 * 2$ (4 times, 1 more time)

$m^n = m * m * m * \cdots$ for n times

$pow(m, n) = \underbrace{m * m * m * \cdots * n-1 \text{ times}}_{n-1 \text{ times}} * \underbrace{m}_{1 \text{ more time}}$

$pow(m, n) = pow(m, n-1) * m$

$$pow(m, n) = \begin{cases} 1 & n = 0 \\ pow(m, n-1) * m & n > 0 \end{cases}$$

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

pow(2, 9) = 2^9

pow(2, 8) * 2 (9-1 = 8, n-1)

pow(2, 7) * 2

pow(2, 6) * 2

pow(2, 5) * 2

pow(2, 4) * 2

pow(2, 3) * 2 = 2^4

pow(2, 2) * 2 = 2^3

pow(2, 1) * 2 = 2 * 2 = 2^2

pow(2, 0) * 2 = 1 * 2

As no of act rec created inside stack are dependent upon no of calls (n+1 calls)

for n = 9 it has made 10 calls (9 to 0)

n+1 calls

10 calls. as n+1 calls ∴ TC → O(n)

Size of stack → O(n)

∴ n + 1 calls

Total 9 multiplication performed. is it possible to compute it will less no of multiplication?

Ex: $2^8 = (2^2)^4$ → power is even take half of power.

$= (2 * 2)^4$

$2^9 = 2 * (2^2)^4$

↳ power is odd one multiplication extra then half of power.

with this obs we can write power func faster than prev one.

Date
Page

Lets rewrite the pow func using the reduced mul method (as discussed before)

```
int pow(int m, int n)
{
  if(n==0)
    return(1);

  if(n%2==0)                    --> if power is even
   return pow(m*m, n/2)          --> can reduce the power by %2

  else                          --> if power is odd
   return m* pow(m*m, (n-1)/2);  --> one mul extra & then
}                                    reduce the power by %2
```

Ex: $2^9 \Rightarrow 2*2^8 \Rightarrow 2*(2^4)^2$

{ no mul is there just return → even no. odd no → one mult }

pow(2, 9) = 2^9 (odd)

$2 * pow(2^2, 8/2)$ (9-1) = $2 * pow(2^2, 4)$ (even) = $2 * 2^8 = 2^9$

$pow(2^2 * 2^2, 4/2)$ = $pow(2^4, 2)$ = 2^8

$pow(2^4 * 2^4, 2/2)$ = $pow(2^8, 1)$ = 2^8

$2^8 * pow(2^8 * 2^8, (1-1)/2)$ = $2^8 * pow(2^{16}, 0)$ = $2^8 * 1 = 2^8$

1

6 multiplication → we got answer → this method
9 multiplication → previous method.

sum(n-1)+
fact(n-1)*n
pow(x, n-1)*

```cpp
// #21 - Let's code Power using Recursion

#include <iostream>
using namespace std;

// method 1
int power(int m,int n){
    if(n==0){
        return 1;
    }
    else{
        return power(m,n-1)*m;
    }
}

// method 2 (reduced multiplication method)
int power2(int m, int n){
    if(n==0){
        return 1;
    }
    if(n%2==0){
        return power2(m*m,n/2);
    }
    else{
        return m * power2(m*m, (n-1)/2);
    }
}


int main(){
    int base=2;
    int exp=8;

    cout<<power(base,exp)<<' '<<power2(base,exp);


    return 0;
}
```

#22

Taylor Series Using Recursion

Taylor series e^x

If we observe there are 3 operation being performed → 1) Summation of terms
2) Power of x
3) denominator → factorial

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots + n \text{ terms}$$

multiplication

Sum(n-1)+n Sum(n) = 1 + 2 + 3 + - - - + n
fact(n-1)*n fact(n) = 1 * 2 * 3 * - - - * n
pow(x,n-1)*x pow(x,n) = x * x * x * - - - n times

→ addition / done at returning time (start from 1)

for taylor series we have to combine these 3 values and write a recursive func.

Where you have to involve multiple values in a recursion we can use static variables

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{2!} + \frac{x^4}{4!} + \dots + n \text{ terms}$$

en no.

e(x,4)
e(x,3)
e(x,2) = $1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!}$
e(x,1) = $1 + \frac{x}{1} + \frac{x^2}{2!}$
e(x,0) = 1 → $1 + \frac{x}{1}$
1

so in e(x,3) func call it should be x^3 for making it x^3 we need x^2 so x can be mul to obtain x^3 similarly, we need 2! so can be mul by 3 to make it 3!

A func for taylor series must perform 3 operations but the func can return only one result.
∴ when we have to involve multiple values in a recursion we can use static var.

$2^8 * 1 = 2^8$

we need x & 1 so x is multiplied by num to become x^2 & 2 is multiplied by den to "2!

So for handling num & den we will take 2 static var.

P → Power → num
f → fact → den.

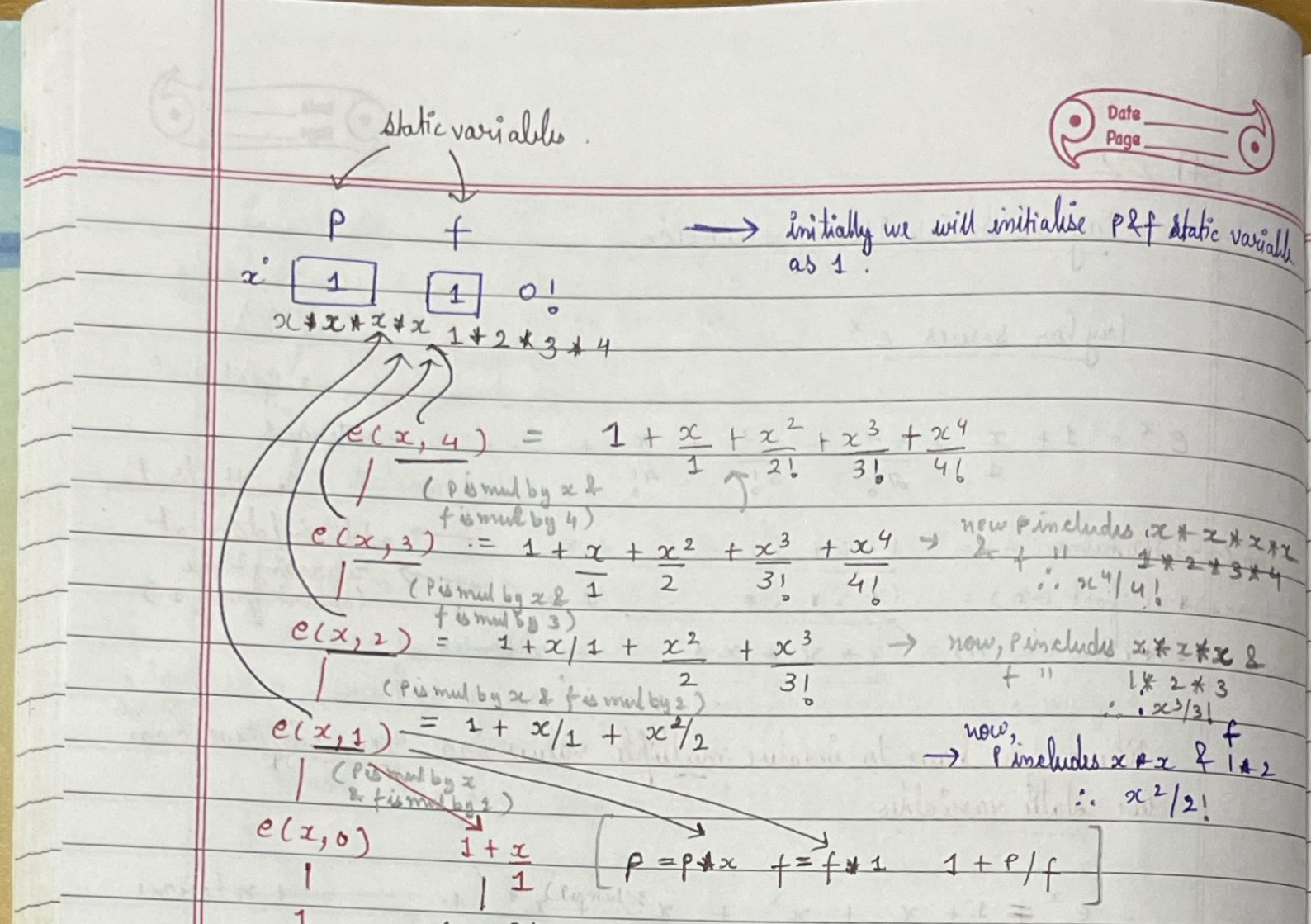
observation:

Ex. Sum(n) = 1 + 2 + 3 + . . . + n

The recursive call will start from n & go on reducing upto 1 & then it will start adding ∴ addition of values is done at returning time. (actual addition starts from 1 onwards) Ex:

Sum(4)
Sum(3) + 4
Sum(2) + 3
Sum(1) + 2
Sum(0) + 1
0

(Same for factorial too → product is done at returning time) (Similarly in power also)



→ Initially we will initialise P & f static variables as 1.

new P includes x*x*x*x & f " 1*2*3*4 ∴ x⁴/4!

→ now, P includes x*x*x & f " 1*2*3 ∴ x³/3!

now, → P includes x*x & f " 1*2 ∴ x²/2!

→ (P includes x & f includes 1 ∴ x/1!.)

[Recursive function for taylor series]

```
int e ( int x, int n )
{
  static int p = 1, f = 1;
  int r;          ← to store result of the func
  if (n==0)
    return(1);
  else
  {
    r = e(x, n-1);
    p = p * x;
    f = f * n;
    return r + p/f
  }
}
```

→ the result of the func is added by p/f Ex: 1 + p/f
∴ we need to take a var to store result of the func.

Ex: e(x,4)
e(x,3)
⋮

1. call itself
2. P is mul by x
3. f is mul by value of n

→ happening at returning time.

→ result of func + p/f
this is what we have to go on returning
(look at black lines of above Ex)

```
#include <stdio.h>
double e(int x, int n)
{
static double p=1, f=1;
double r;

if(n==0)
    return 1;
r = e(x,n-1);
p = p*x;
f = f*n;
return r+p/f;
}

int main()
{
printf("%.lf \n", e(1,10));
return 0;
}
```

→ e^1 & no of terms are 10;

(3,10)

(4,10)

(4,15)

output

2.718282

20.079665

54.443104

54.597883

```cpp
// #23 - Let's code Taylor Series using Recursion

#include <iostream>
using namespace std;

double e(int x, int n){
    static double p=1,f=1;
    double r;

    if(n==0){
        return 1;
    }
    else{
        r = e(x,n-1);
        p=p*x;
        f=f*n;
    }
    return r + p/f;
}

int main(){

    cout<<e(4,15);
    return 0;
}
```

#24

Taylor Series using Horner's Rule

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots \text{---} + n \text{ terms}$$

Total Mul req: 0 0 $\frac{x*x}{1*2}$ $\frac{x*x*x}{1\times2\times3}$ $\frac{3}{3}$

$0+0+1+4+6+8+10$ → How many multiplication required.

$$2\left[1+2+3+4\cdots\right]$$

$$2\ \frac{n(n+1)}{2}$$

If we are evaluating taylor series for n, terms then this many total mult will be req. $= n(n+1) = O(n^2)$

reducing Mul. (HOW TO REDUCE NO OF MULT) earlier $O(n^2)$ Quadratic

[By taking commons we can reduce the no of multipli..]

$$1 + \frac{x}{1} + \frac{x^2}{1*2} + \frac{x^3}{1*2*3} + \frac{x^4}{1*2*3*4}$$

$$1 + \frac{x}{1}\left[1 + \frac{x}{2} + \frac{x^2}{2*3} + \frac{x^3}{2*3*4}\right]$$

reduction

$$1 + \frac{x}{1}\left[1 + \frac{x}{2}\left[1 + \frac{x}{3} + \frac{x^2}{3*4}\right]\right]$$

$$1 + \frac{x}{1}\left[1 + \frac{x}{2}\left[1 + \frac{x}{3}\left[1 + \frac{x}{4}*1\right]\right]\right]$$

only 4 mul required.

reduced to $O(n)$ linear

here at calling time operations are performed

[the recursive func we wrote previously were performing operations mostly at returning time but here at calling time itself we need to find out this term (Ex: $\left[1+\frac{x}{4}\right]$ & multiply it by $x/3$ & add 1.

Then $\left[1+\frac{x}{3}\left[1+\frac{x}{4}\right]\right]$ should be multiplied by $x/2$ & add 1

then $\left[1+\frac{x}{2}\left[1+\frac{x}{3}\left[1+\frac{x}{4}\right]\right]\right]$ " " " " $x/1$ " " 1

so for $x/4$ we should multiply by 1 & add 1 ... This way we get our result]

So the operations are happening at calling time ∴ it can be easily returned using loop.

~~e(x, n)~~ can be done by loop

[TAYLOR SERIES HORNER'S RULE]

~~here at each~~

```
int e(int x, int n)
{
    int s = 1;  → for holding result
    for(; n>0; n--)
        s = 1 + x/n * s;
    return s;
}
```

→ Ex: 1+x * 1 returned to repeat it until n > 0.

[HORNER'S RULE TAYLOR SERIES]

recursive version (with help of stack var)

```
int e(int x, int n)
{
    static int s = 1;  → we have to perform mult at calling time ∴ taking help of static variable.
    if (n==0)
        return s;

    s = 1 + x/n * s;
    return e(x, n-1);
}
```

→ we previously discussed that recursions have ascending & descending but this procedure can be performed only in ascending.

```
double e(int x, int n)
{
    static double s = 1;
    if(n==0)
        return s;
    s = 1 + x * s / n;
    return e(x, n-1);
}
```

```
int main()
{
    printf("%.1f\n", e(1, 10));
    return 0;
}
```

iterative version of taylor series (normal one)

```
double e(int x, int n)
{
    double s = 1;
    int i;
    double num = 1;
    double den = 1;
    for(i=1; i<=n; i++)
    {
        num *= x;
        den *= i;
        s += num/den;
    }
    return s;
}
int main()
{
    printf("%.1f\n", e(1, 10));
    return 0;
}
```

```cpp
// #25 - Let's code Taylor Series Using Horner's Rule

#include <iostream>
using namespace std;

// using loop
double e(int x, int n){
    double s=1;
    for(;n>0;n--){
        s = 1 + ((double)x/n) * s;
    }
    return s;
}

// using recursion
double e2(int x, int n){
    static double s=1;
    if(n==0){
        return s;
    }
    else{
    s = 1 + ((double)x/n) * s;
    return e2(x,n-1);
    }

}

int main(){
    cout<<e(2,10)<<' '<<e2(2,10);
    return 0;

}
```

```cpp
// #26 - Let's code Taylor Series Using Iterative Method

#include <iostream>
using namespace std;

double e(int x, int n)
{
 double s=1;
 int i;
 double num=1;
 double den=1;

 for(i=1;i<=n;i++)
 {
 num*=x;
 den*=i;
 s+=num/den;
 }
 return s;
}
int main()
{
 cout<<e(1,10);
 return 0;
}
```

#27

Fibonacci Series Using Recursion

fib(n) →	0	1	1	2	3	5	8	13
terms →	0	1	2	3	4	5	6	7

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

Recursive

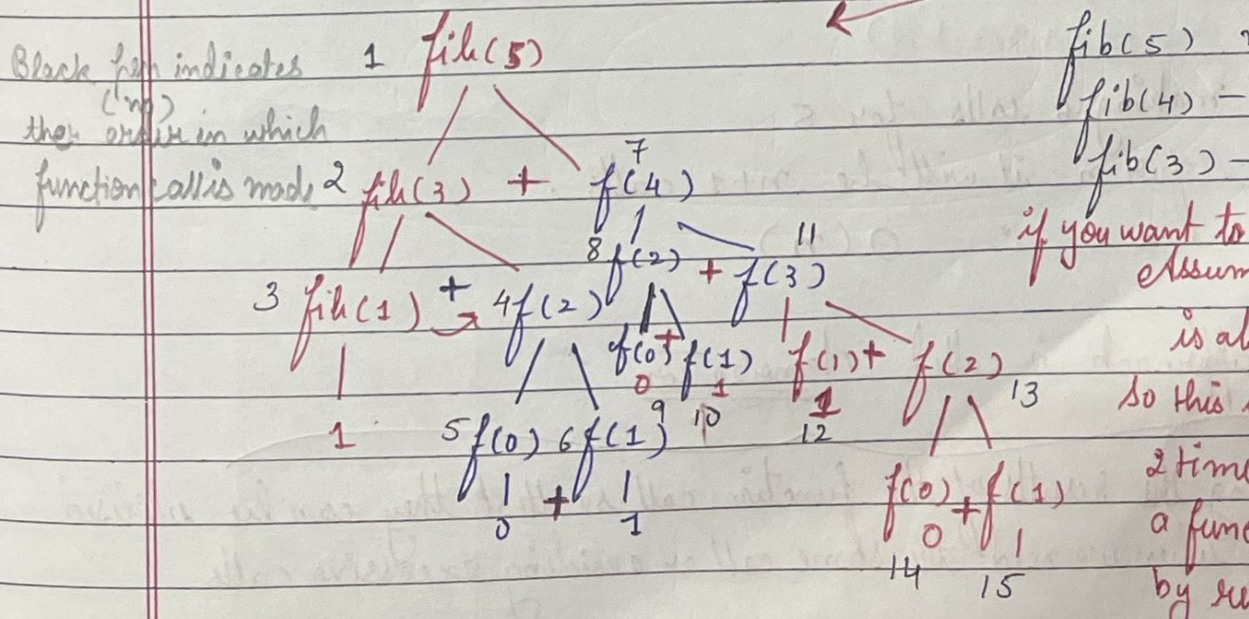
```
int fib(int n)
{
  if(n<=1)
    return n;
  return fib(n-2) +
         fib(n-1);
}
```

iterative

```
int fib(int n)
{
  int t0=0, t1=1, s=0, i;   // =0 → to avoid warning from compiler (in case)   — 1
  if(n<=1) return n;                                                          — 1
  for(i=2; i<=n; i++)                                                         — n
  {
    s = t0+t1;   — n-1
    t0 = t1;     — n-1
    t1 = s;      — n-1
  }
  return s;      — 1
}
```

= 4n.

O(n)



fib(5) no of calls 15
fib(4) — 9
fib(3) — 5

if you want to know time taken assume that fib(n-2) is also fib(n-1)
So this is calling itself 2 times → So when a func is calling 2 times by reduced value of 1 (n-1) then time is $O(2^n)$ → approx

if we observe iterative version takes $O(n)$ time
& recursive version takes $O(2^n)$ time

HOW TO REDUCE IT?

If you see the recursion tree, the value of $f(3)$, $f(2)$, $f(1)$, $f(0)$ are called multiple times. et recursive func is calling itself multiple times for the same value, such a recursive func is called as <u>Excessive recursion</u> (∴ fibonacci recursion is excessive recursion)

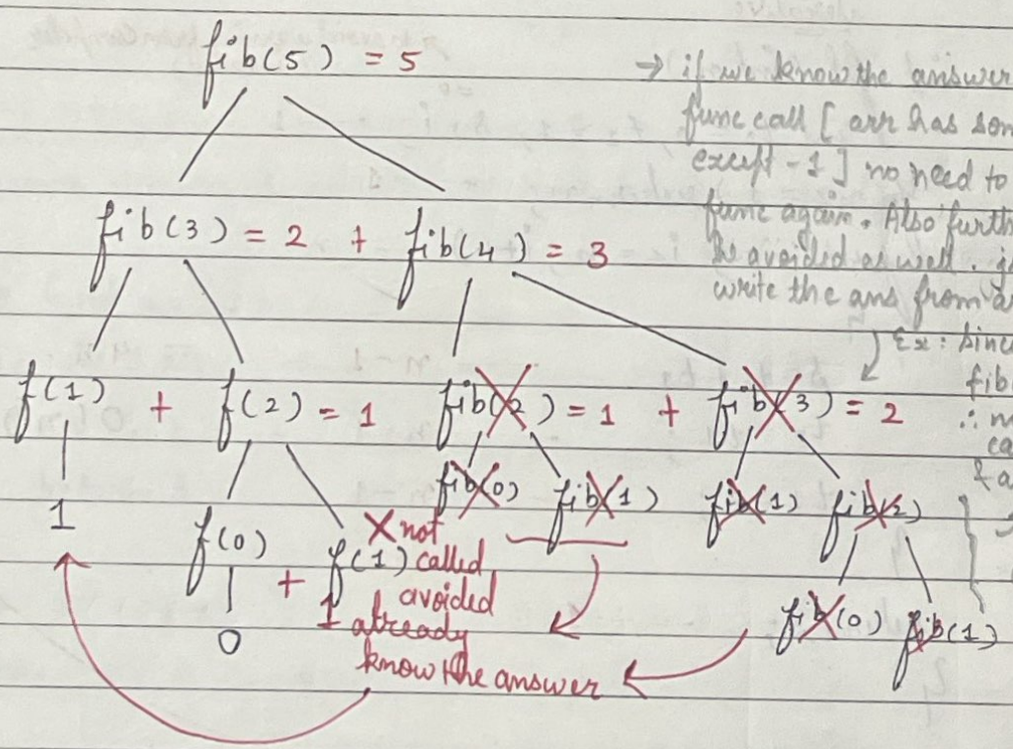
<u>how to avoid excessive calls?</u>

⇒ take an array

F [-1 | -1 | -1 | -1 | -1 | -1 | -1] → <u>initialise it with -1</u>

0 1 2 3 4 5 6

0 1 1 2 3 5



→ if we know the answer of the func call [arr has some value except -1] no need to call the func again. Also further calls will be avoided as well. just simply write the ans from arr

Ex: since we know fib(3) = 3 ∴ no need to call again & also, further calls are avoided as well

(for 5, 4, 3, 2, 1, 0)

making 6 calls for 5

for n it will be n+1 calls.

∴ $O(n)$

This approach is called <u>Memoization</u>

Storing the results of the function call so that they can be utilised again when we need the same call or avoiding excessive calls.

```
int F[10];
int fib(int n)
{
  if (n<=1)
  {
    F[n] = n;
    return n;
  }
  else
  {
    if (F[n-2] == -1)
      F[n-2] = fib(n-2);
    if (F[n-1] == -1)
      F[n-1] = fib(n-1);
    F[n] = F[n-2] + F[n-1];
    return F[n-2] + F[n-1];
  }
}

int main()
{
  int i = 0;
  for (i=0; i<10; i++)
    F[i] = -1;

  printf("%d\n", mfib(5));
  return 0;
}
```

```cpp
// #28 - Let's code Fibonacci

#include <iostream>
using namespace std;

// iterative
int fibI(int n)
{
    int t0 = 0, t1 = 1, s;

    if (n <= 1)
    {
        return n;
    }

    for (int i = 2; i <= n; i++)
    {
        s = t0 + t1;
        t0 = t1;
        t1 = s;
    }
    return s;
}

// recursive
int fibR(int n)
{
    if (n <= 1)
    {
        return n;
    }
    else
    {
        return fibR(n - 2) + fibR(n - 1);
    }
}

// using memoization
int f[10];

int fibM(int n)
{

    if (n <= 1)
    {
        f[n] = n;
        return n;
    }

    else
    {
        if (f[n - 2] == -1)
        {
            f[n - 2] = fibM(n - 2);
        }
        if (f[n - 1] == -1)
        {
            f[n - 1] = fibM(n - 1);
        }

        f[n] = f[n - 2] + f[n - 1];
        return f[n - 2] + f[n - 1];
    }
}

int main()
{

    for (int i = 0; i < 10; i++)
    {
        f[i] = -1;
    }
    cout << fibI(7) << ' ' << fibR(7) << ' ' << fibM(7);
    return 0;
}
```

#29

COMBINATION FORMULA

$$nC_r = \frac{n!}{r!(n-r)!}$$

```
int c (int n, int r)
{
  int t1, t2, t3;
  t1 = fact(n);      n
  t2 = fact(r);      r
  t3 = fact(n-r);    n
  return t1/(t2*t3); 1
}
```

time taken → $3n$ $O(n)$

Ex:

A B C D E F G

Select

ABC

ABD

✗ ACB

↳ interchanging of pos will not give a new selection its called permutation

∴ ABC & ACB will not be counted as diff

7 objs given

Select only 3

How many ways?

^5C_r

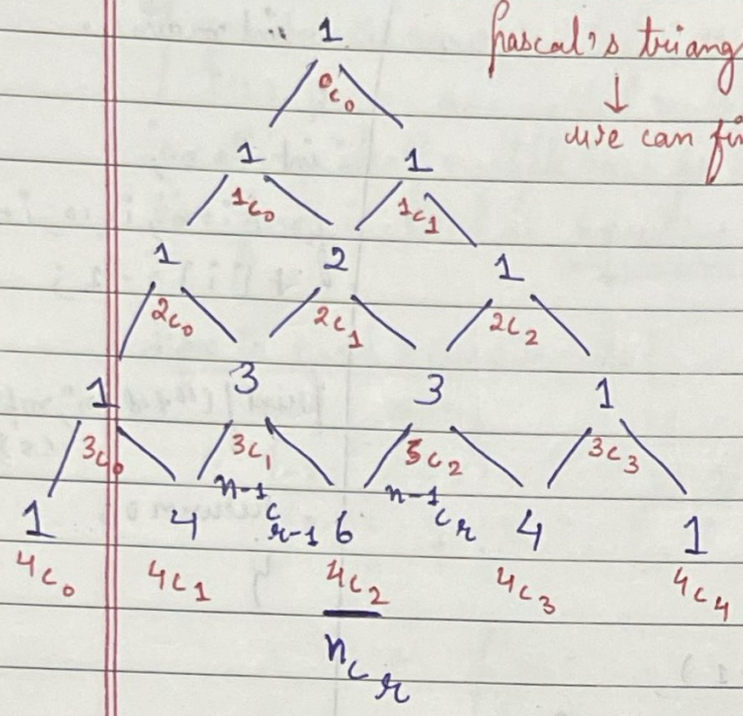
r ↓

$0 - 5$

$^5C_0$ $^5C_1$ $^5C_2$ ... $^5C_5$

→ also called as selection formula, if the set of objects are given then in how many ways we can select subset of those objects.

previously we saw a simple func to obtain combination
Now let's see recursive method,
to get an idea of recursive func first we will look at pascal's triangle



pascal's triangle
↓
use can find the value of n_{C_r}

n_{C_r} values can obtained adding $^{n-1}C_{r-1}$ & $^{n-1}C_r$.

```
int c(int n, int r)
{
if (r==0 || n==r)  ⟶ extreme points
    return 1;
else
    return c(n-1, r-1) + c(n-1, r);
}
```

ex: $^2C_0 = 1$, $^2C_2 = 1$, $^3C_0 = 1$, $^3C_3 = 1$ ⟹ r=0 or r=n

— x —— x

so it will make the call from bottom towards top. (bottom up calls)
whenever it reaches 1 it terminates.

meaning r==0 || n==r.

Date
Page

Recursive

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

int nCr(int n, int r)
{
  int num, den;
  num = fact(n)
  den = fact(r) * fact(n-r);

  return num/den;
}

int main()
{
printf("%d\n", nCr(5,1));
 return 0;
}
```

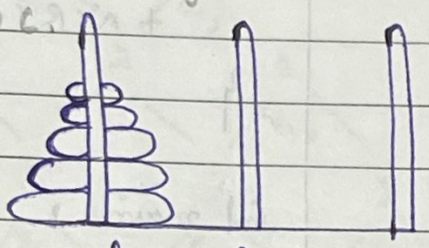
```
int NCR(int n, int r)
  if(n==r || r==0)
  return 1;
  return NCR(n-1, r-1)
    + NCR(n-1, r);
}

int main()
{
printf("%d\n", NCR(4,
                   2));
 return 0;
}
```

```cpp
// #30 - Let's Code nCr using Recursion

#include<iostream>
using namespace std;

// factorial func
int fact(int n){
    if(n==0) return 1;
    return fact(n-1)*n;
}

int nCr(int n, int r){
    int num,den;
    num = fact(n);
    den = fact(r)* fact(n-r);

    return num/den;
}

// using pascal triangle
int NCR(int n, int r){
    if(n==r || r==0) return 1;
    return NCR(n-1,r-1)+NCR(n-1,r);
}

int main(){
    cout<<nCr(4,2)<<endl<<NCR(4,2);
    return 0;
}
```

#31

TOWER OF HANOI

Three towers given, disks on tower A, our task is to move disk from A to C.



from using to

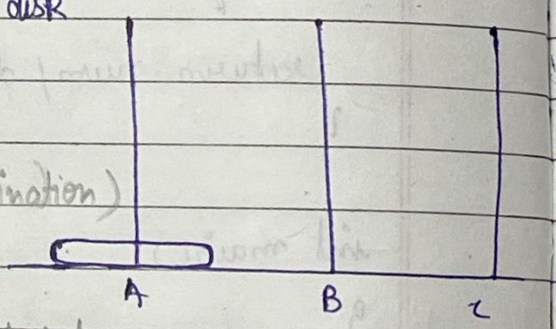
[But the condition is 1. we have to move one disk at a time.
2. we have to transfer like that no larger disk should be kept over smaller disk]

By following these 2 cond^n we have to move disks A to C by using auxillary tower B. [for helping]

problem 1.

TOH (1, A, B, C)

Move disk from A to C using B.



[one disk given]

TOH (no of disk, Source, Auxillary, distination)

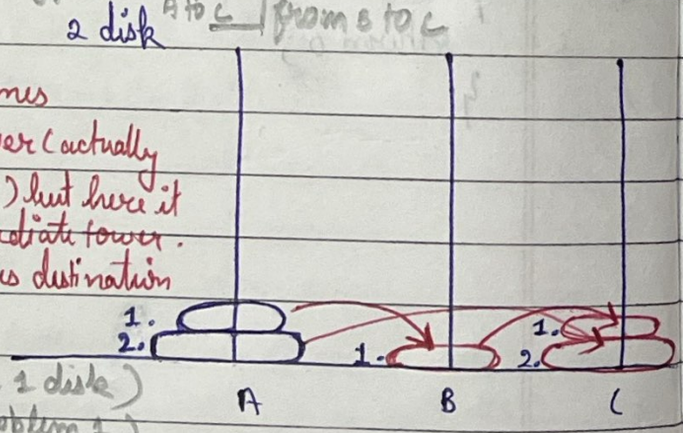
problem 2. [two disk given]

TOH (2, A, B, C) → 1. move smaller disk from A to B.
2. move bigger disk from A to C. then move smaller disk from B to C

Smaller disk ← 1. TOH (1, A, C, B) → in this step C becomes auxillary tower (actually it a destination) but here it is an intermediate tower. & B becomes distination

larger tower ← 2. Move disk from A to C using B.

← 3. TOH (1, B, A, C)

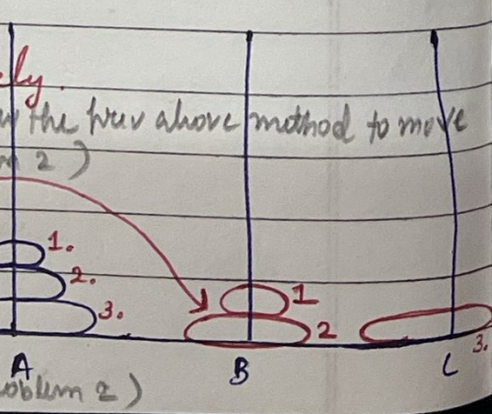


Step 1 & 3 are recursive.
(will follow the prev above method to move 1 disk) (problem 1)

problem 3.

source intermediator dest

TOH (3, A, B, C)



Move two disks to B using the above method recursively (Not two disk at a single time) → This step will follow the prev above method to move two disk (problem 2)

1. TOH (2^{n-1}, A, C, B) ←
2. Move disk from A to C using B.
3. TOH (2^{n-1}, B, A, C) → move 2 disk from B to C following prev above method (problem 2)

using 3 disk problem we can define a recursive problem for n disk.

Date
Page

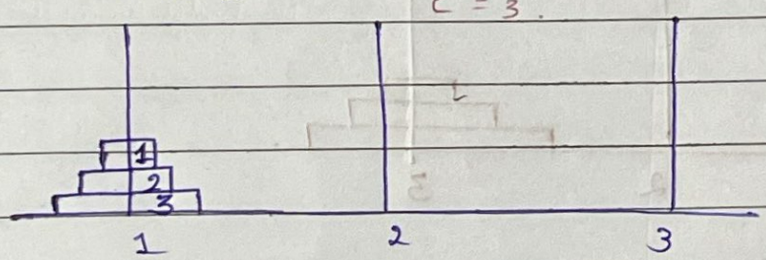
no of disks → towers ← variables.

```
void TOH (int n, int A, int B, int C)
{                    from using to.
  if (n>0)
  {
    TOH(n-1, A, C, B);
    printf("from %d to %d", A, C);
    TOH(n-1, B, A, C);
  }
}
```

TOH(3, 1, 2, 3);

n A B C

n = 3
A = 1
B = 2
C = 3.



calls: ① 3, 1, 2, 3

② 2, 1, 3, 2 — 1 to 3

③ 1, 1, 2, 3 — 1 to 2

④ 0, ✗ — 1 to 3

⑤ 0, ✗

⑥ 1, 3, 1, 2 — 3 to 2

⑦ 0, ✗

⑧ 0, ✗

⑨ 2, 2, 1, 3 — 2 to 3

⑩ 1, 2, 3, 1 — 2 to 1

⑪ 0, ✗

⑫ 0, ✗

⑬ 1, 1, 2, 3 — 1 to 3

⑭ 0, ✗

⑮ 0, ✗

printing is done like Inorder traversal.

for n disks Exponential time taking func.

$1+2+2^2+\ldots+2^n = O(2^n)$

Time

no of disks n = 3 calls = 15 $(1+2+2^2+2^3)$

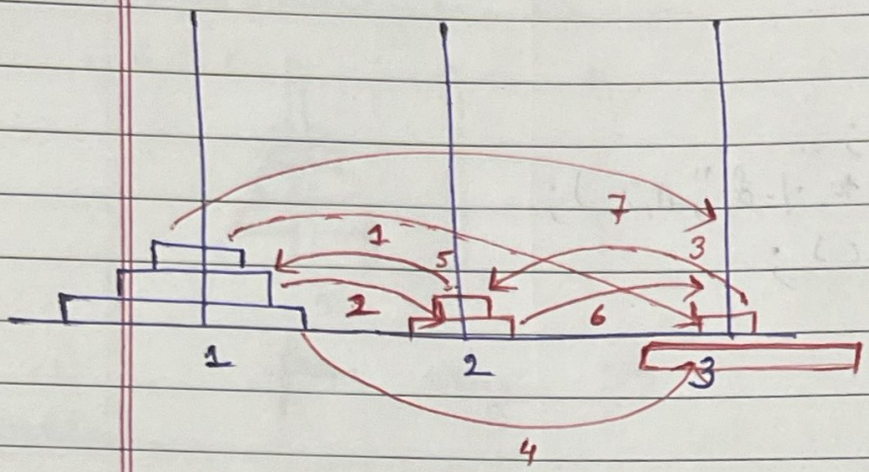
$2^{n+1}-1$ n = 2 calls = 7. 2^4-1.

The order of way it will be displayed → (1,3), (1,2), (3,2), (1,3), (2,1), (2,3), (1,3).

	calls	
$n = 3$	15	$1 + 2 + 2^2 + 2^3 = 2^4 - 1$
$n = 2$	7	

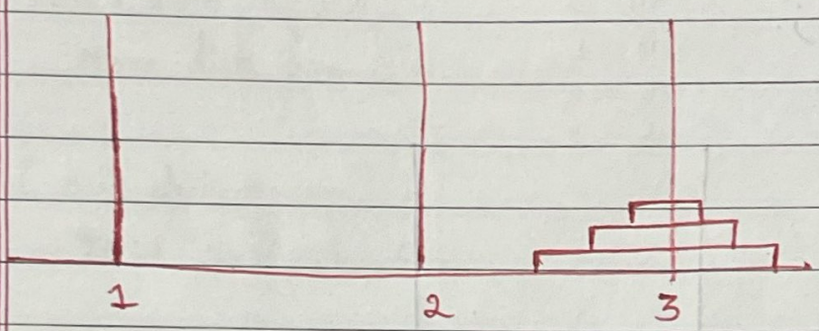
$1 + 2 + 2^2 + \dots 2^n = 2^{n+1} - 1$

$= O(2^n)$ → Exponential time problem.



moves

1. (1, 3)
2. (1, 2)
3. (3, 2)
4. (1, 3)
5. (2, 1)
6. (2, 3)
7. (1, 3)



—— x ——

```
void TOH(int n, int A, int B, int C)
{
  if(n>0)
  {
   TOH(n-1, A, C, B);
   printf("(%d, %d)\n", A, C);
   TOH(n-1, B, A, C);
  }
}
int main()
{
  TOH(3, 1, 2, 3);          (2, 1, 2, 3)
  return 0;
}
```

(1, 2)
(1, 3)
(2, 3).

```cpp
// #32 - Let's Code Tower of Hanoi

#include<iostream>
using namespace std;

void TOH(int n, int A,int B,int C){
    if(n>0){
        TOH(n-1,A,C,B);
        cout<<"from "<<A<<" to "
<<C<<endl;
        TOH(n-1,B,A,C);
    }
}

int main(){
    TOH(2,1,2,3);
    return 0;
}
```