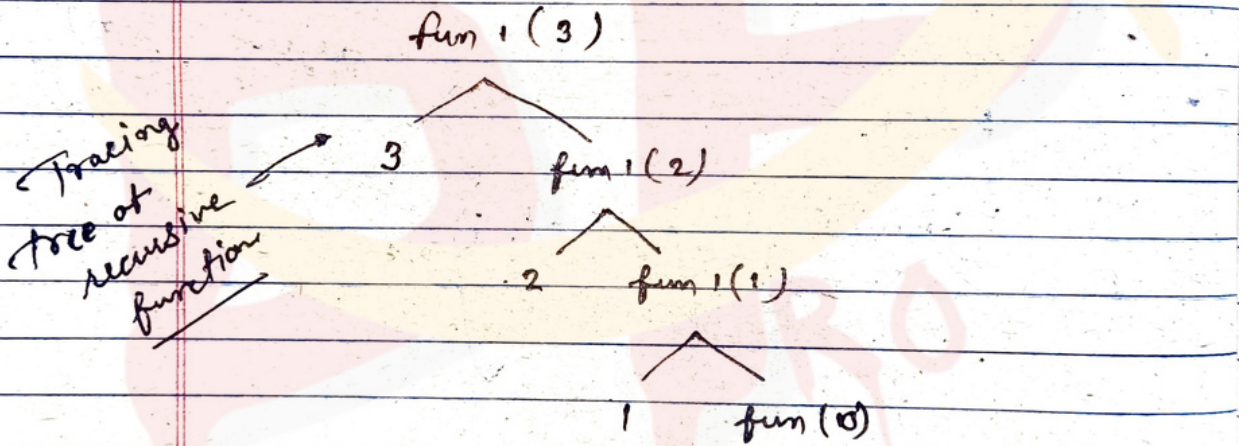
Recursion

infinite

Date
Page

①
```
Void fun1(int n)
{
  if (n > 0)
  {
    printf("%d", n);   → (It is called base condition and it must be written in program for terminating the function, otherwise program runs infinites.)
    fun1(n-1);
  }
}
Void main()
{
  int x = 3;
  fun1(x);
}
```



Tracing tree of recursive function

Here printing is done at calling time.

output - 3, 2, 1

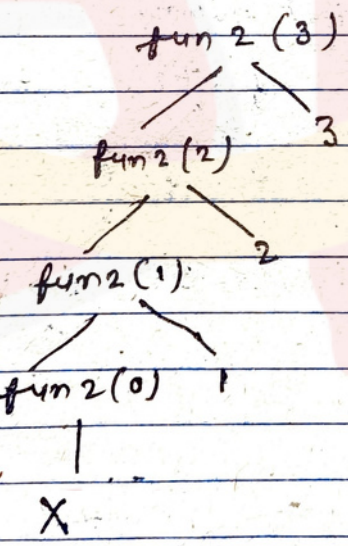
X (Terminated and now go back to the previous call)

→ Once the call has been terminated, the control should go back to the previous call.

* Recursion has two phases-
a) calling phase (b) Returning phase

Date
Page

(2)
```
Void fun2(int n)
{
  if (n>0)
  {
    fun2(n-1);
    printf("%d", n);
  }
}
void main()
{
  int x = 3;
  fun2(x);
}
```



Here printing is done at returning time.

O/P :- 1, 2, 3

Date
Page

* Void fun(int n)
{
if (n>0)
{
Ascending 1. Calling time execute
2. fun(n-1) * 2
Descending 3: Returning time execute

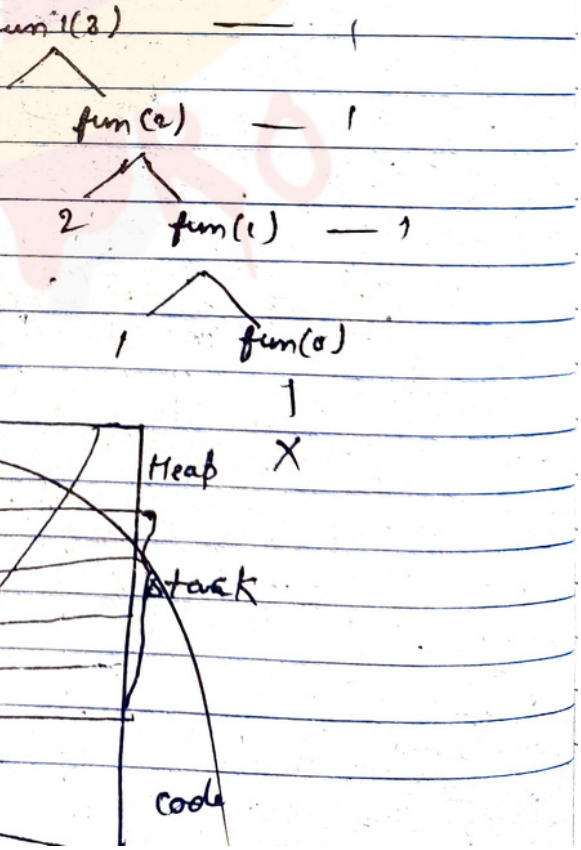
* Loop vs Recursion

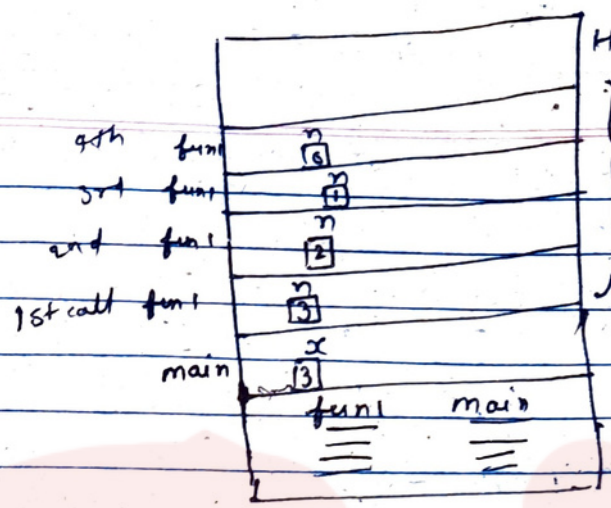
Main difference b/w loop and recursion is that a loop will have only ascending phase, but recursion will have ascending as well as descending.

21/03/22

How recursion uses stack memory

① Void fun(int n)
{
if (n>0)
{
printf("%d",n);
fun(n-1);
}
}
Void main()
{
int x=3;
fun(x);
}

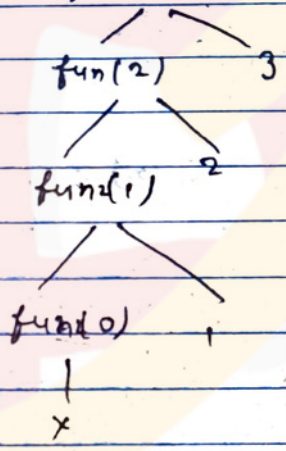




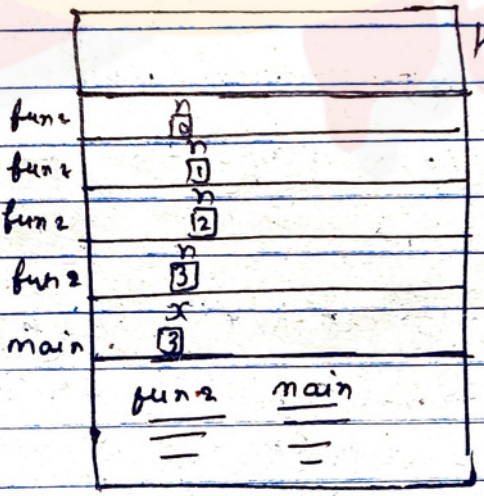
o/p - 3 2 1

for the fun1
Total call n = n + 1
(Here n = 3)
total call = 3 + 1 = 4)

② Void fun2 (int n)
{ if (n > 0)
{ fun2 (n-1);
printf ("%d", n);
}
}
Void main()
int x = 3
fun (x);
}



∴ degree = o(n)



o/p = 1 2 3

Total call = n+1
degree = o(n)

→ Space depends on max^m height of stacks.

Date
Page

Time Complexity of Recursion

Time depends on the value that we are passing from main() fn.

Eg:-
```
Void fun1(int n)          —— T(n)
{
  if(n>0)                 —— 1
  {
    printf("%d",n);       —— 1
    fun1(n-1);            —— T(n-1)
  }
}
```
for (n>0) → T(n-1) + 2 → T(n-1) + 1 (+ constant)

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

$T(n) = T(n-1) + 1$ — (i)

$T(n) = T(n-2) + 1 + 1$ ($\because T(n) = T(n-1)+1$, $\therefore T(n-1) = T(n-2)+1$)

$T(n) = T(n-2) + 2$ — (ii)

Similarly

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

$T(n) = T(n-3) + 3$ — (3)

$T(n) = T(n-k) + k$ — (k)

Assume → $n - k = 0$

$\therefore n = k$

Now

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

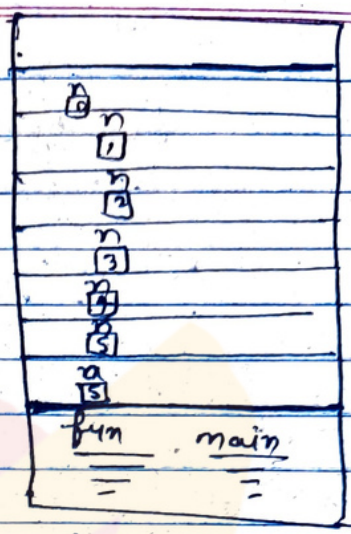
$= T(0) + n$

($\because T(0) = 1$) $\therefore T(n) = 1 + n \rightarrow O(n)$

Local variable in a function (Recursive)

Date
Page

```
int fun (int n)
{
  if (n>0)
  {
    return fun(n-1)+n;
  }
  return 0;
}
main()
{ int a = 5;
  printf("%d", fun(a));
}
```



o/p : 15

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

* Static Variables in Recursion (global variable)

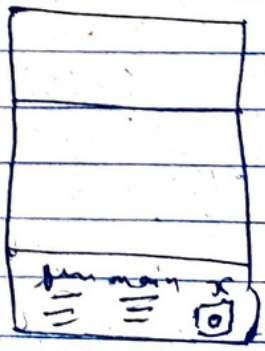
```
int fun (int n)
{ static int x = 0;
  if (n>0)
  {
    x++;
    return fun(n-1)+x;
  }
  return 0;
main()
{ int a = 5;
  printf("%d", fun(a));
}
```

x: 0 → 1 → 3 → 5

fun(5) = 25
fun(4) + 5 = 25
fun(3) + 5 = 20
fun(2) + 5 = 15
fun(1) + 5 = 10
fun(0) + 5 = 6
0

o/p = 25

Here when we declare as global variable then also result also be same.



22/03/22

Types of Recursion

i) Tail recursion -

When all the operation will be performed at calling time only and the function will not be performing any operation at returning time. Means everything is performed at calling time.

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

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

if will performed at returning time so its not a tail recursion.

Comparison of tail recursion with loop-

```
void fun(int n)
{
  while(n>0)
  printf("%d",n);
    n--;
}
main()
fun(3)
```

Time - O(n)

space - O(1)

(If its activation record is only 1)

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

time O(n)

space O(n)

(If its activation record is n+1).

Q. . In the case of tail recursion, loop is more efficient.

Date
Page

1) Head recursion -

When the function doesn't have to perform any operation at the time of calling; it has to do everything only at the time of returning.

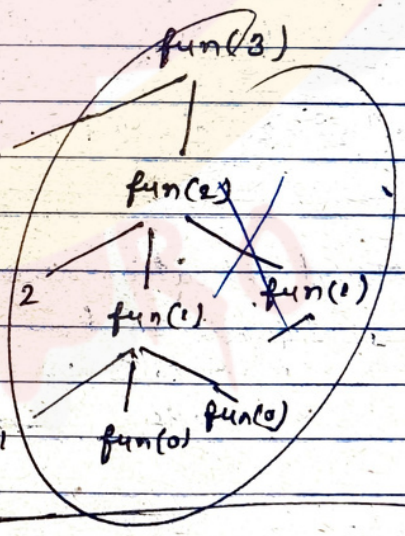
Q:-
```
void fun(int n)
{
  if(n>0)
  {
    fun(n-1);
    printf("%d",n);
  }
}
main()
{
  fun(3);
}
```

```
void fun(int n)
{
  int i=1;
  while(i<=n)
  {
    printf("%d",i);
    i++;
  }
}
main()
{
  fun(3);
}
```

Comparision of head recursion with loop -

23/03/22 Tree recursion -

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

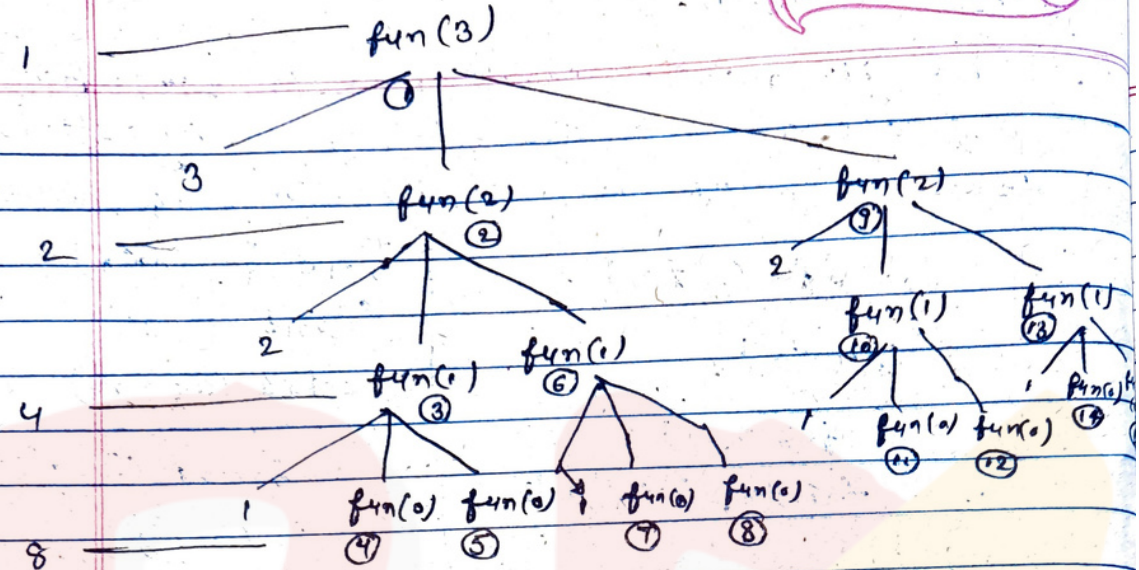


Tree recursion - A function that calling itself more than one time.)

Linear recursion - A function i.e recursive function, if it is calling itself only one time.

Linear recursion (it is not head or tail recursion)

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



o/p = 3 2 1 1 2 1 1

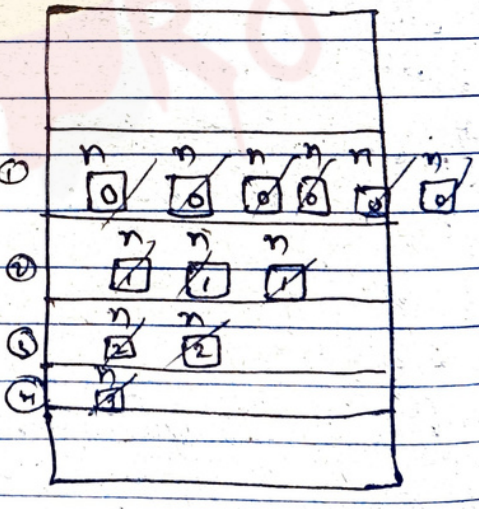
How many calls are made for n value passing -

1 + 2 + 4 + 8 = 15 (For n = 3, calls are 15 times)

$= 2^0 + 2^1 + 2^2 + 2^3 = 2^{3+1} - 1$

$\therefore \boxed{2^0 + 2^1 + 2^2 + 2^3 + \cdots 2^n = 2^{n+1} - 1} \rightarrow O(2^n)$

Here Total activation record = (it depends on how many calls are made) → 15



time → $O(2^n)$

Space - $O(n)$

↓

It's depends on max height of stack means here this 4.

i.e - if : n = 3 then space = (n+1) = 4

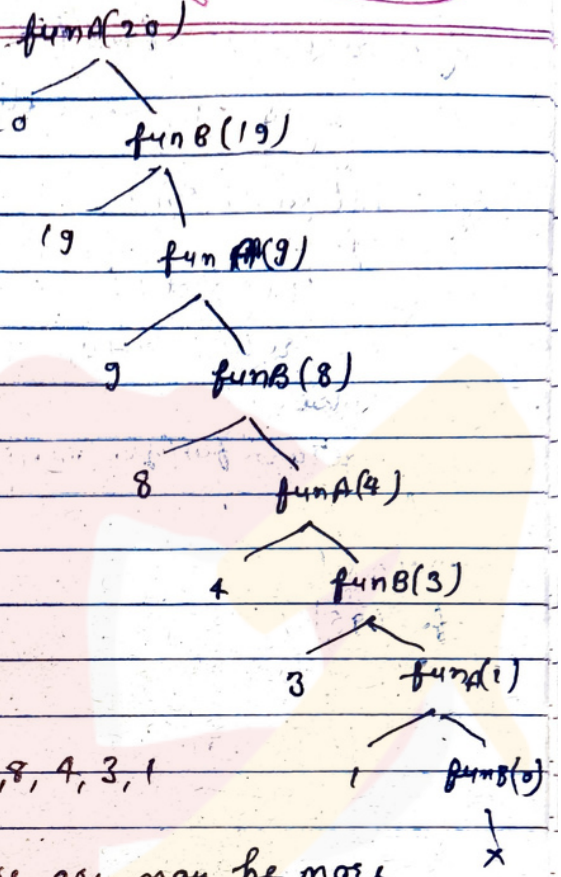
n : 3

stack also - n+1 depends on value of n. : 4

Indirect recursion

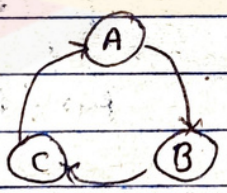
```
Void funA (int n)
  if (n > 0)
  { printf ("%d", n);
    funB (n-1);
  }
}
Void funB (int n)
  { if (n > 1)
    { printf ("%d", n);
      funA (n/2);
    }
}
main()
{ funA(20);
}
```

o/p = 20, 19, 9, 8, 4, 3, 1



In indirect recursion, there are may be more than one fn. and they are calling one-another in a circular fashion.

Eg:-



Nested Recursion (Recursion inside recursion)

(A recursive function that passes parameter as a recursive call.)

Eg:-
```
int fun(int n)
{
  if(n>100)
    return n-10;
  else
    return fun(fun(n+11));
}
main()
{
  fun(95);
}
```

O/P = 91

fun(95)
|
fun(fun(95+11)) 96 = fun(106)
|
fun(96);
|
fun(fun(96+11)) 97 = fun(107)
|
fun(97)
|
fun(fun(97+11)); 98 = fun(108)
|
fun(98)
|
fun(fun(98+11)); 99 = fun(109)
|
fun(99)
|
fun(fun(99+11); 100 = fun(111)
|
fun(100)
|
fun(fun(100+11)) 101 = fun(111)
|
fun(101)
|
91

Sum of first 'n' natural numbers

Date
Page

$1 + 2 + 3 + 4 + \dots + n$

$Sum(n) = 1 + 2 + 3 + \dots + n$

$Sum(n) = 1 + 2 + 3 + \dots + (n-1) + n$

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

Note - Whenever we define any recurrence relation or recursive function then we must have two types of statements -

i) large value of n (~~what is the formula~~)

ii) small value of n (what is the answer)

Q1 ①
```
int sum(int n)
{
  if (n==0);
  return 0;
  else
  return sum(n-1)+n;
}
main()
{
sum(5);
}
```

calls - (n+1)

time - O(n)

space also - O(n)

Sum(5) = 15

Sum(4) + 5 = 15

Sum(3) + 4 = 10

Sum(2) + 3 = 6

Sum(1) + 2 = 3

Sum(0) + 1 = 1

0

Other method

②
```
int sum(int n).
{
return n*(n+1)/2
}
main()
sum(5);
```

time - O(1)

③
```
int sum(int n)
{ int i,s=0;          — 1
for(i=1; i<=n; i++)   — n+1
  s = s+i;            — (n)
return s;             — 1
}
main()
sum(5);
```

time - O(n)

24/03/2022

Date
Page

* Factorial using recursion

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

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

$fact(n) = 1 * 2 * 3 * 4 * \dots * (n-1) * n$

$\boxed{fact(n) = fact(n-1) * n}$

$0! = 1$

$1! = 1$

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

Using recursion

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

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

Using loop (Iteration)

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

Exponent $[(m)^n]$

Date
Page

$2^5 = 2 * 2 * 2 * 2 * 2$

$m^n = m * m * m * m * m$ ... for n times

pow(m, n) = (m × m × m * ... n-1 times) * m

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;
}
main()
int a = 2, b = 9
int r = pow(a,b);
printf("%d", r);
}
```

→ ~~time~~ = n+1 → (calls)

time = O(n)

→ space = O(n)

pow(2, 9)
↓
pow(2, 8) * 2
|
pow(2, 7) * 2
|
pow(2, 6) * 2
|
pow(2, 5) * 2
|
pow(2, 4) * 2
|
2^4 = pow(2, 3) * 2
|
2^3 pow(2, 2) * 2
|
2^2 = pow(2, 1) * 2
|
2 = pow(2, 0) * 2
|

Date
Page

2nd method m^n

$2^8 = (2*2)^4 = (2^2)^4$

$2^6 = (2^2)^3 = (2*2)^3$

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

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

$2^{11} = 2*(2^2)^5$

(this method works faster than previous method)

when n is even -

$Pow(m,n) = Pow(m*m, n/2)$

when n is odd -

$Pow(m,n) = m * Pow(m*m, (n-1)/2)$

```
int Pow(int m, int n)
{
  if (n==0)
    return 1;
  if (n%2==0)
    return Pow(m*m, n/2);
  else
    return m*Pow(m*m, (n-1)/2);
}
main()
{ int a=2, b=9
  int r = Pow(a,b);
  printf("%d", r);
}
```

$2^9 = Pow(2,9)$

$2*2^8 = 2*Pow(2*2, 9/2)$ (4)

$2^4 = Pow(2^2*2^2, 4/2)$ (2)

$2^2 = Pow(2^2*2^2, 2/2)$ (1)

$2^8*1 = 2^8*Pow(2^8*2^8, 1/2)$ (0)

25/08/2

* When you have to involve multiple values in the recursion, then you can use static variables

* Taylor series e^x -

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

```
#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("%.1f\n", e(4,4));
    return 0;
}
```

P: 1 → 1 → 1+1*x → 1+x+x*x

F: 1 → 1 → 1*2 → 1*2*3 → 1*2*3*4

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

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

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

$e(x,1) = 1 + \frac{x}{1} + \frac{x^2}{2}$

$e(x,0)$ → 1

$p = p*x$; $f = f*1$; $1 + p/f$

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

How many time multiplication is required in taylor series -

0 0 $\frac{x \times x}{1 \times 2}$ $\frac{x \times x \times x}{1 \times 2 \times 3}$ $\frac{3}{3}$

$2 + 4 + 6 + 8 + 10 \cdots$

$= 2[1 + 2 + 3 + 4 + \cdots]$

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

$= n(n+1)$ → Multiplication requir

time - $O(n^2)$ - quadratic

other way to reduce multiplication (By taking common)

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

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

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

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

Taylor series using Horner's rule →

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

Here multiplication required only 4 times -

time - $O(n)$ - linear

Date
Page

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

Using loop

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

Using recursion

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

```
#include <stdio.h>
double e(int x, int n)
{ static double s;
  if (n == 0)
    return s;
  s = 1 + (double)(x/n) * s;   // s = 1 + x*s/n;
  return e(x, n-1);
}
int main()
{
  printf("%.1f \n", e(2, 10));
  return 0;
}
```

28/03/2022

Using loop

```
#include <stdio.h>
double e(int x, int n)
{
    double s = 1;
    int i;
    double num = 1;
    double den = 1;      // denominator
    for(i=1; i<=n; i++)
    {
        num *= x;
        den *= i;
        s += num/den
    }
    return s;
int main()
{
    printf("%lf\n", e(1, 10));      → 2.718282
    return 0;
}
```

* Excessive recursion -

That recursion which calls itself multiple times for the same parameter

* Memorisation - (storing the results) -

Storing the results of the function call so that they can be utilized again when we need the same call or avoiding excessive calls. This approach is called as a memorisation.