DATA STRUCTURE USING C

- Way of organising data
- Data should be efficiently used by the program

VARIOUS DATA STRUCTURES

- ARRAY
- MATRIX
- LINKED LIST

] Physical Data Structure (How data is arranged)

- STACK
- QUEUE

] Linear

- TREE
- GRAPH

] Non Linear

- HASHING

] Tabular

] Logical Data Structure (How data is utilized)

- RECURSION
- SORTING TECHNIQUE

Logical Data structures are implemented using physical Data Structures

BASICS

1. Arrays
2. Structure
3. Pointer
4. Reference
5. Parameter Passing
6. Classes
7. Constructor
8. Templates

ARRAYS

→ Collection of similar datatypes

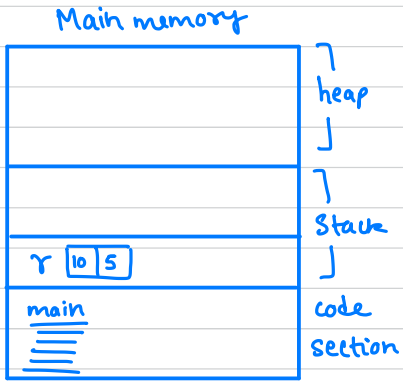
STRUCTURES

→ Collection of different data types

```
Struct rectangle
{
    int length;     — 2
    int breadth;    — 2
};                    4 bytes
```

but till here, it does not occupy any space in memory

```
int main()
{
    Struct rectangle r = {10,5}  → here it occupies memory
    r.length = 10;
    r.breadth = 5;
}
```



```
Struct card
{
    int face;
    int shape;
    int colour;
};

int main()
{
    Struct card deck[52] = {....}
    printf("%d", deck[0].face);
    printf("%d", deck[0].shape);
}
```

POINTERS

↳ address variables

1. Why pointers
2. Decleration
3. Initialization
4. Dereferencing
5. Dynamic allocation

```
int a=10;
int *p; ⟶ decleration
p=&a; ⟶ initialization
printf("%d", a);
printf("%d", *p); ⟶ dereferencing
```

USES OF POINTER

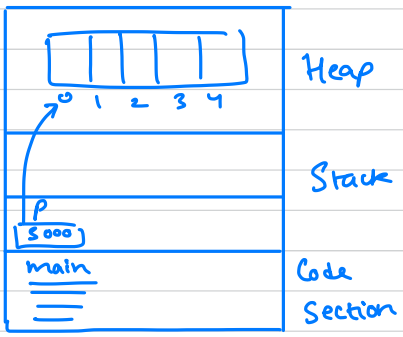
1. Accessing heap
2. Accessing resources like keyboard or mouse.
3. Parameter passing.

HOW TO USE POINTER FOR ALLOCATING HEAP

```
#include<stdlib.h> ⟶ header file for malloc()

int main()
{
    int *p;
    p=(int *)malloc(5*sizeof(int));
}
```

↳ for returning as malloc function returns void pointer (type casting)

↓ Creating space for 5 integer values in heap.



REFERENCE IN C++

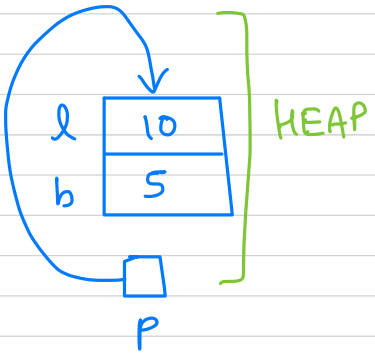
```
int main()
{
    int a=10;
    int &r=a;
    cout<<a;
    cout<<r;
}
```

POINTER TO STRUCTURE

```
Struct rectangle
{
    int length;
    int bredth;
};

int main()
{
    struct rectangle r = {10, 5};
    struct rectangle *p = &r;

    r.length = 15;
    (*p).length = 20;
    p -> length = 20;
}
```

✓ ✓

doesn't occupy 4 bytes of memory (2 bytes)




```
int main()
{
    struct rectangle *p;

    p = (struct rectangle *)malloc(sizeof(struct rectangle));

    p -> length = 10;
    p -> bredth = 5;
}
```

FUNCTIONS

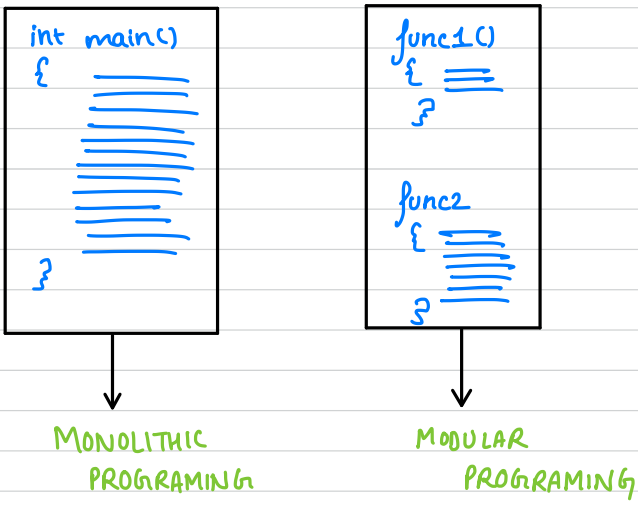
What are functions ⟶ Performs a specific task

Parameters of passing functions

- → Pass by value
- → Pass by address
- → Pass by reference

] only in C

] In C++



FUNCTION EXAMPLE

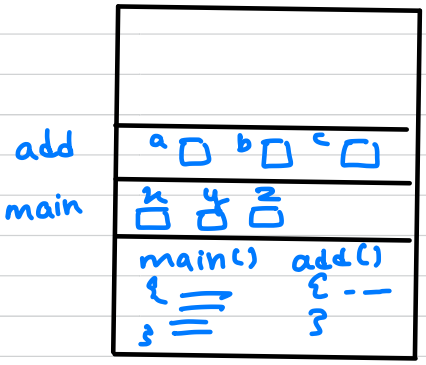
```
int add (int a ,int b)
{
    int c;
    c = a+b;
    return c;
}

int main()
{
    int x, y, z ;
    x = 10;
    y = 5;
    z = add (x,y);
    printf(" %d", z );
}
```

prototype

Formal Parameter

Actual Parameter

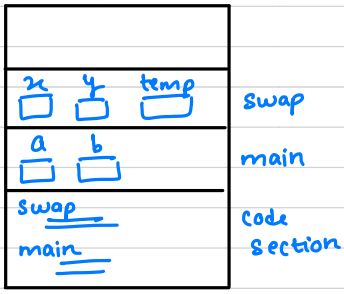
15



CALL BY VALUE

```
Void swap( int x , int y)
{
    int temp;
    x = y;
    y = temp;
}

int main()
{
    int a, b;
    a = 10;
    b = 20;
    swap (a,b);
    printf (" %d %d", a, b);
}
```

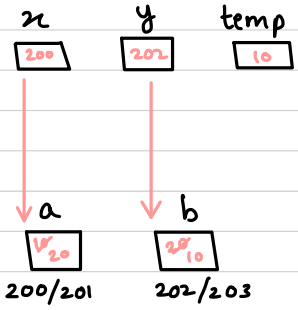
10 20

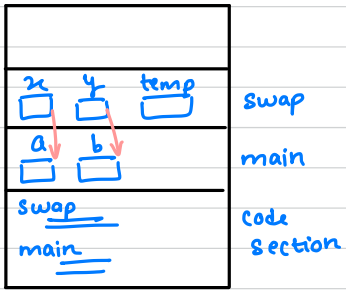


CALL BY ADDRESS

```
Void swap( int *x , int *y)
{
    int temp;
    *x = *y;
    *y = temp;
}

int main()
{
    int a, b;
    a = 10;
    b = 20;
    swap (&a,&b);
    printf (" %d %d", a, b);
}
```

20 10

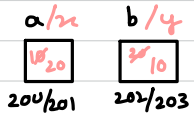


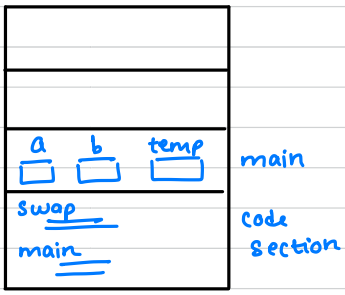


CALL BY REFERENCE (Only in C++)

```
Void swap( int &x , int &y)
{
    int temp;
    x = y;
    y = temp;
}

int main()
{
    int a, b;
    a = 10;
    b = 20;
    swap (a,b);
    printf (" %d %d", a, b);
}
```

20 10





Only 2 bytes of memory is used as in reference, only another name is given to the pre-existing variable

---

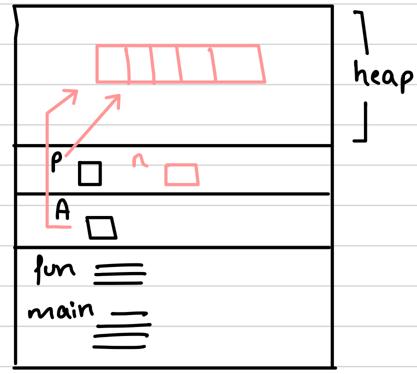
ARRAY AS PARAMETER

```
void fun (int A[], int n)
{
        int i;
        for (i=0 ; i<n; i++)
            printf (" %d", A[i]);
}

int main()
{
    int a[5] = {2,4,6,8,10};
    fun ( A, 5);
}
```

↳ or int *A

as ARRAYS can only be passed as call by address.

```c
int [] fun (int n)     // or  int * fun (int n)
{                      // it returns an array
    int *p;
    p= (int *)malloc (n* sizeof (int));
    return p;
}

int main()
{
    int *A;
    A= fun(5);
}
```



STRUCTURE AS PARAMETER

CALL BY VALUE

```c
int area (struct rectangle r1)
{
    r1.length++;
    return r1.length* r1.bredth;
}

struct rectangle
{
    int length;
    int bredth;
};

int main ()
{
    struct rectangle r = {10,5};
    printf ("%d", area(r));
```

CALL BY REFERENCE

```c
int area(struct rectangle &r1)
```

CALL BY ADDRESS

```c
void change1 (struct rectangle *p, int l)
{
    p->length = l;
}

int main()
{
    struct rectangle r= {10,5};
    change1 (&r, 20);
}
```

PASSING ARRAYS AS CALL BY VALUE USING STRUCTURES

```
Void fun(struct test t1)
{
    t1.A[0] = 10;
}

int main()
{
    struct test t={{2,4,6,8,10}, 5};
    fun(t);
}
```

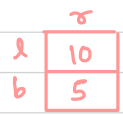
```
struct test
{
    int A[5];
    int n;
}
```

changes made in the array in the function will not be reflected as it is call by value

---

STRUCTURES AND FUNCTIONS

```
struct rectangle
{
    int length;
    int bredth;
}

void initialize (struct rectangle *r, int l, int b)
{
    r -> length = l ;
    r -> bredth = b;
}

int area ( struct rectangle r )
{
    return r.length * r.bredth;
}

void change1 ( struct rectangle *r, int l)
{
    r -> length = l ;
}
```




```
int main()
{
    struct rectangle r ;

    initialize (&r, 10, 5);
    area (r);
    change1 (&r, 20);
}
```

because values need to be changed, so call by address.

CLASS AND CONSTRUCTOR

```
class rectangle
{
    private:
            int length;
            int bredth;
    public:
            rectangle (int l, int b)
            {
                length = l;
                bredth = b;
            }

            int area()
            {
                return length * bredth;
            }

            void changelength (int l)
            {
                length = l;
            }
};
```

```
int main()
{
    rectangle r(10, 5);
    r.area()
    r.changelength(20);
}
```

```
class rectangle
{
    private:
            int length;
            int bredth;
    public:
            rectangle()
            {
                length = bredth = 1;
            }
            rectangle (int l, int b);
            int area();
            int perimeter();
```

constructors (overloaded)

facilitators (They do operations on data members)

```
            int getlength
            {
                return length;
            }
            void setlength (int l)
            {
                length = l;
            }
            ~ Rectangle();
};
```

Accessor

Mutator

Destructor →

```
rectangle :: rectangle (int l, int b)
{
        length = l;
        bredth = b;
}

int rectangle :: area()
{
        return length * bredth;
}

int rectangle :: perimeter()
{
        return 2 * (length + bredth);
}

rectangle :: ~ rectangle()
{
}



int main()
{
        rectangle r(10, 5);
        cout << r.area;
        cout << r.perimeter;
        r.setlength(20);
        cout << r.getlength();
}
```

TEMPLATE CLASS

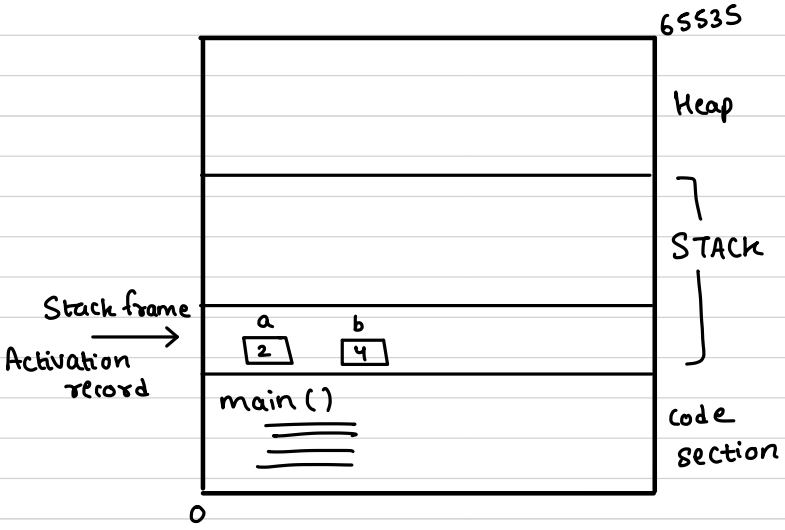
→ For using various datatypes. Using the same class for different datatypes.

```
template < class T >
class arithmetic
{
    private:
        T int a;
        T int b;

    public :
        arithmetic (T int a, T int b)
        T int add ();
        T int sub ();
};

template < class T > ⟶ because effect of previous template has ended
arithmetic <T>:: arithmetic (int a, int b)
                             T     T
{
    this -a = a;
    this -b = b;
}
template <class T>
T int arithmetic <T> :: add()
{
    T int c;
      c= a+b;
      return c;
}

template < class T>
int arithmetic <T> :: sub()
{
    T int c;
      c= a-b;
      return c;
}
```

```
int main()
{
    arithmetic < int> ar(10, 5);
    cout << ar.add();
    arithmetic < float > ar1 (1.5,1.2);
    cout << ar1.add();
}
```

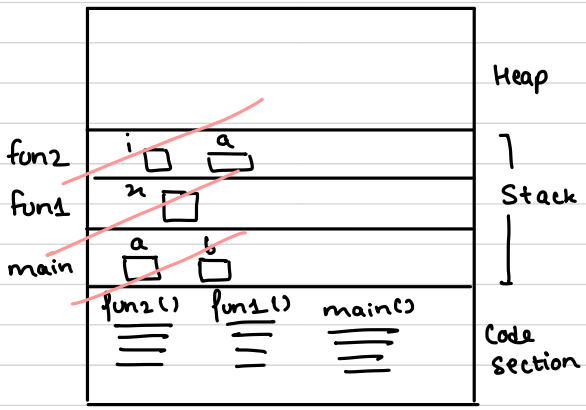
STATIC VS DYNAMIC MEMORY ALLOCATION

```
void main()
{
        int a;
        float b;
}
```



HOW DOES STACK WORK?

```
void fun2(int i)
{
      int a;
}

void fun1()
{
     int x;
     fun2(x);
}

void main()
{
        int a;
        float b;
        fun1();
}
```

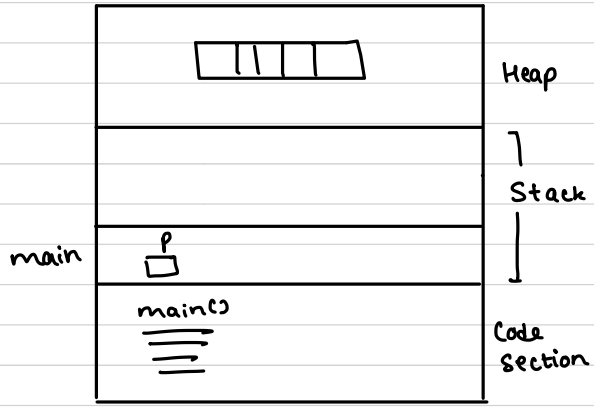


1. Stack is always organised memory.

HOW DOES HEAP MEMORY WORK?

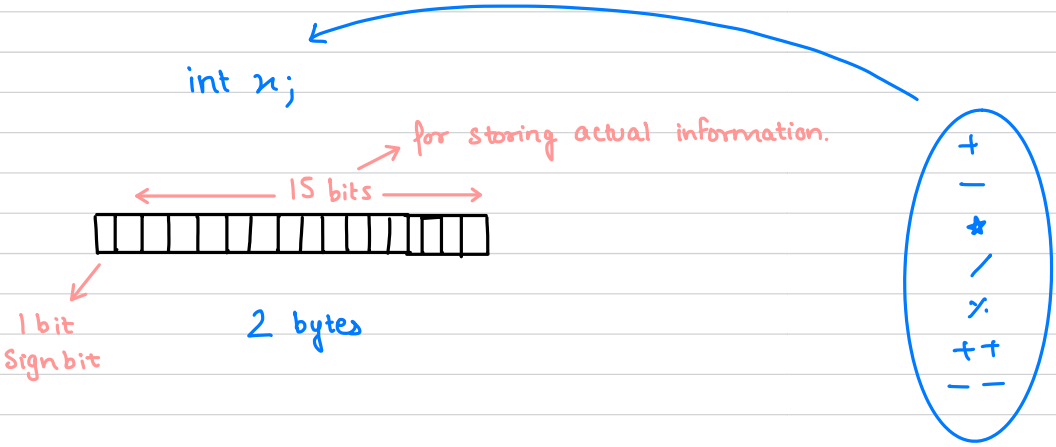
1. Heap may be organised memory or unorganised memory.
2. Heap must be treated as a resource i.e when it is needed, we must use it and when not needed, free it so that it can be used by other applications.
   For eg- printer is a resource.

```
void main()
{
        int *p;

for C++ p= new int[5];
 for C  p= (int*) malloc(2*5);
                  → for array

DEALLOCATION  delete []p;
              p= NULL;
}
```



DATATYPE?

↳ 1. Representation of data.
2. Operation on data.



ABSTRACT DATATYPE

↳ hiding internal details

(part of object oriented programming)

What does log mean?

$\log_2 n$ ⟶ This gets divided by this until it reaches 1.

Time and Space complexity

```
void swap(x,y)
{
      int t;
      t=x;      ——— 1
      x=y;      ——— 1
      y=t;      ——— 1
}                     3
```

O(1)

```
int sum(int A[], int n)
{
      int s,i;
      s=0;                    ——— 1
      for(i=0;i<n;i++)        ——— n+1
      {     1        n
            s=s+A[i];         ——— n
      }
      return s;               ——— 1
}                             2n+3
```

O(n)

because i will be initialized (+1) and i will be incremented for n no of times and 1 time it will fail too (+n)

// because one time it will fail too.

```
Void  Add (int n)
{
      int i,j;
      for(i=0;i<n;i++)                    ——————  n+1
      {
          for(j=0;j<n;j++)                ——————  n*(n+1)
          {
              c[i][j] = A[i][j] + B[i][j]; -  n*n
          }
      }
}
```

because the nested loop will run again and again for (n+1) no of times.

$f(n) = 2n^2+2n+1$

$O(n^2)$

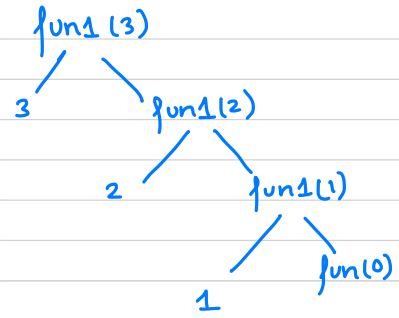
RECURSION

↳ When a function calls itself

EXAMPLE #1

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

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

TRACING TREE



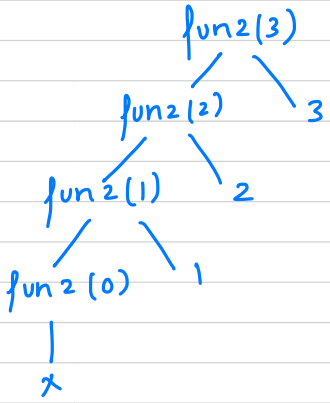
OUTPUT

3 2 1

EXAMPLE #2

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

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

TRACING TREE



OUTPUT

1 2 3

```
void fun (int n)
{
    if (n>0)
    {
ASCENDING 1. calling
        2. fun(n-1)
DESCENDING 3. returning
    }
}
```

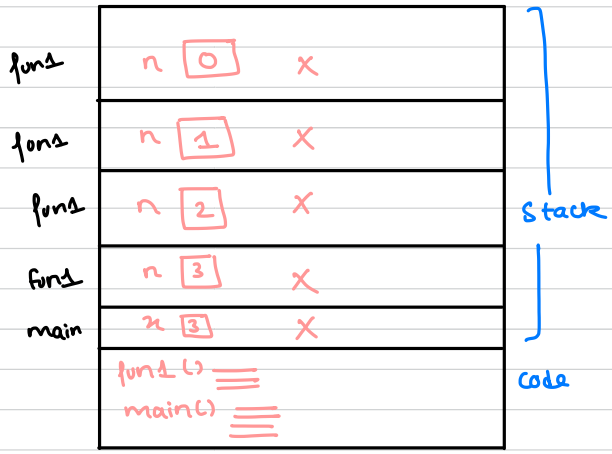
DIFFERENCE BETWEEN LOOP AND RECURSION

The main difference between loop and recursion is that recursion allows two phases i.e ascending and descending while loop allows only ascending

HOW STACK IS UTILISED IN RECURSIVE FUNCTIONS?

EXAMPLE #1

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

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



Here, value of n was 3, so there were 4 calls (Tracing tree).

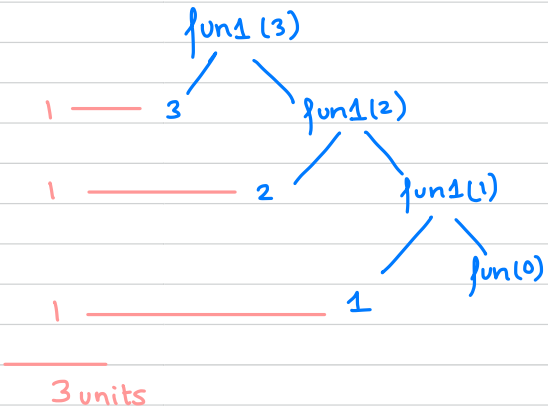
So for value of n, there will be n+1 calls

TIME COMPLEXITY (using Tree)

EXAMPLE #1

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

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

TRACING TREE



Thus, for n calls → n units of time.

$O(n)$

TIME COMPLEXITY (using recurrence relation)

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

(2 = 1 + 1)

This function takes n time.

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

As it is similar to T(n)

Assume it as 1 for constant

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

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

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

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

$T(n-3) + 1$

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

Assume $n-k=0 \therefore n=k$

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

$\vdots$

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

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

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

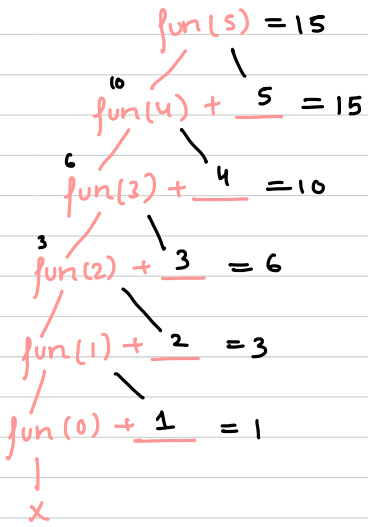
$T(n) = 1 + n$

$O(n)$

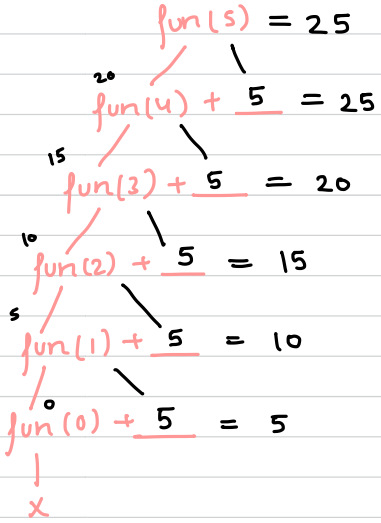
STATIC VARIABLES IN RECURSION

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

main()
{
        int a= 5;
        printf (" %d", fun (a));
}
```

TRACING TREE




```
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));
}
```



TYPES OF RECURSION

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

1. TAIL RECURSION

When the function is calling itself and that call is the last call in the function.

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

1. Every operation is performed at calling time.
2. Tail recursions can easily converted into loops.

TAIL RECURSION AND LOOPS

Some compilers convert your program to loop if you have used tail recursion as they are more effecient

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

fun(3);
```

space O(1)

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

fun (3);
```

O(n)

2. HEAD RECURSION

It means that the the function does not need to perform any operation at the time of calling. It has to do all operations at returning time.

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

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

Head recursions cannot be so easily converted into loops.

3. TREE RECURSION

LINEAR RECURSION

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

When the function calls itself only once.

TREE RECURSION

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

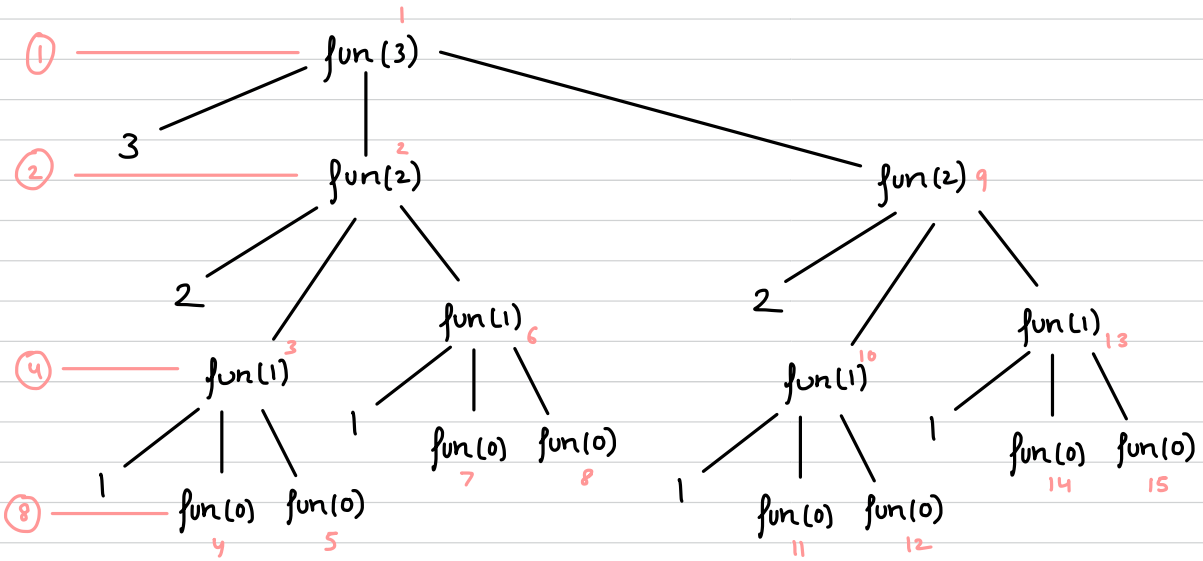
When the function calls itself more than one time.

EXAMPLE OF TREE RECURSION

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

```
fun(3);
```

● ACTIVATION RECORDS



$1 + 2 + 4 + 8 = 15$

$2^0 + 2^1 + 2^2 + 2^3 = 2^{3+1} - 1$ $\quad 2^{n+1} - 1$

GP Series

Time = $O(2^n)$

Space = $O(n)$

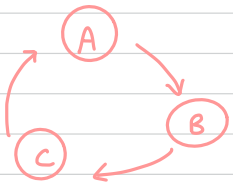
[No of levels of activation records $n+1$
3 – parameter
4 – levels]

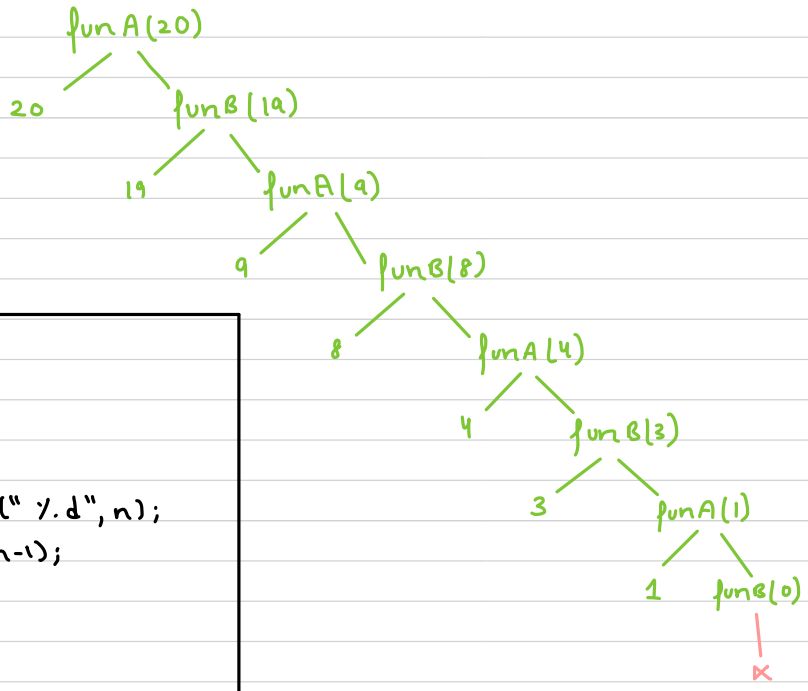
OUTPUT

3 2 1 1 2 1 1

4. INDIRECT RECURSION

In indirect recursion, there are more than one function and they are calling one another in a circular manner.



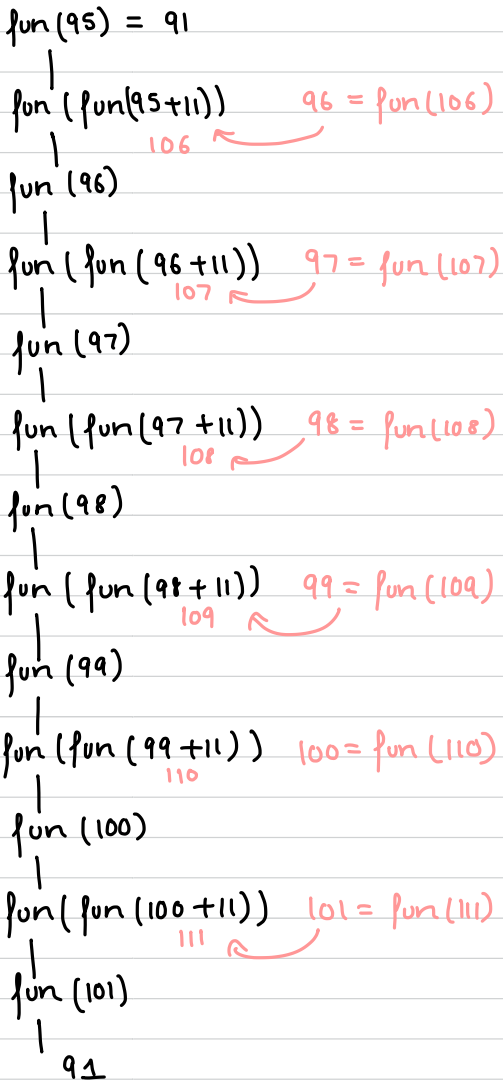



```
void funA(int n)
{
        if (n>0)
        {
              printf(" %d", n);
              funB(n-1);
         }
}

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

5. NESTED RECURSION

In a nested recursion, the recursive function will pass parameter as a recursive call.

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

fun (95);
```



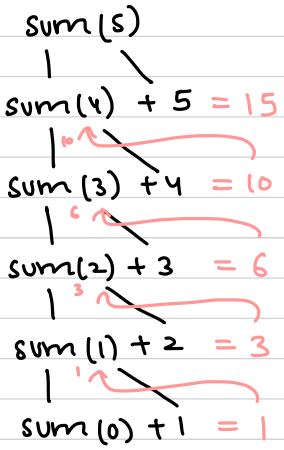
SUM OF FIRST N NATURAL NUMBERS

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

$Sum(n) = 1+2+3+4+\ldots\ldots+(n-1)+n$

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

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




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

Time = O(n)
Space = O(n)

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

O(n)

```
Int sum (int n)
{
    int i,s=0;          ——— 1
    for (i=1;i<=n;i++)  ——— n+1
        s=s+i;          ——— n
    return s;           ——— 1
}                           2n+3
```

O(n)

FACTORIAL USING RECURSION

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

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

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

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

POWER USING RECURSION

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

$pow(m,n) = m * m * m \ldots\ldots * (n-1)$ times $* m$

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

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

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

Here 10 calls are being performed

But this way is taking longer

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

$= (2*2)^4$

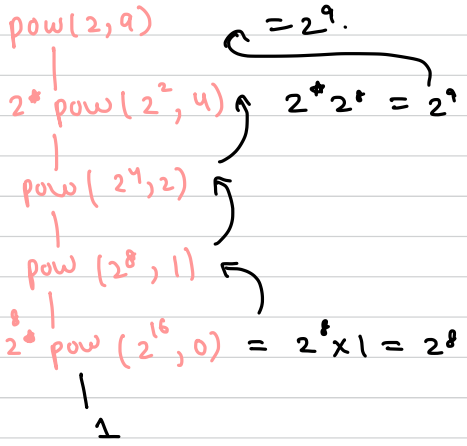
$2^9 = 2 * (2^2)^4$

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

REWRITING POWER FUNCTION

FASTER METHOD

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

    else
        if (n%2==0)
            return pow(m*m, n/2);
        else
            return m*pow(m*m, (n-1)/2);
}
```



TAYLOR'S SERIES

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

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

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

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

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

$e(x,0)$

$= 1$

$p = p * x$ $\quad f = f * 1$ $\quad 1 + p/f$

```
int  e(int x, int n)
{
      Static  int p=1, f=1;
      int r;

      if ( n == 0)
            return 1;

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

p	f
1	1
$x*x*x*x$	$1 \times 2 \times 3 \times 4$

TAYLOR'S SERIES USING HORNER'S RULE

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

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

No of multiplications ↓

0 0 1+1 2+2 6 8 10

2 4

$2(1+2+3+\ldots n)$

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

$O(n^2)$ Quadratic

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

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

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

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

↑ ↑ ↑ ↑

$O(n)$ Linear

```
int e(int x, int n)
{
    int s=1;

    for(;n>0;n--)
        s=1+x/n*s;

    return s;
}
```

USING FOR LOOP

```
int e(int x, int n)
{
    static int s=1;

    if(n==0)
        return s;
    else
        s=1+x/n*s;

    return e(x,n-1);
}
```

USING RECURSION

FIBONACCI SERIES

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

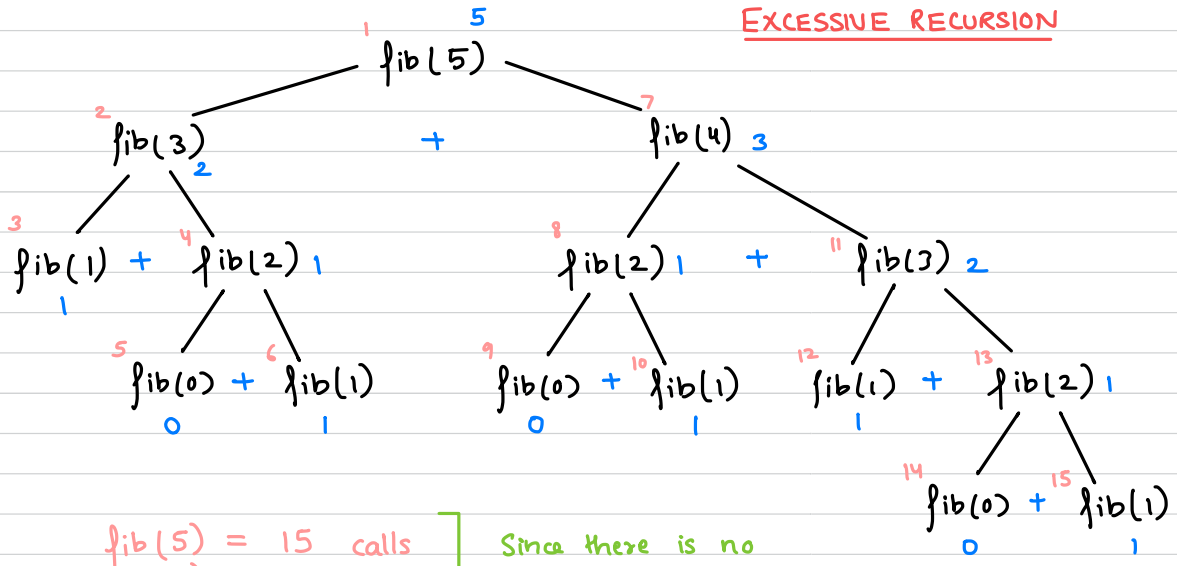
$$fib(n) = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ fib(n-2) + fib(n-1) & n>1 \end{cases}$$

PROGRAM USING ITERATION

```
int fib (int n)
{
        int t0 = 0, t1 = 1, s, i;        — 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)
```

PROGRAM USING RECURSION

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



fib(5) = 15 calls
fib(4) = 9 calls
fib(3) = 5 calls

Since there is no definite pattern, than we have to assume that the function is calling itself

fib(n-2) + fib(n-1);
2 fib(n-1)

↓

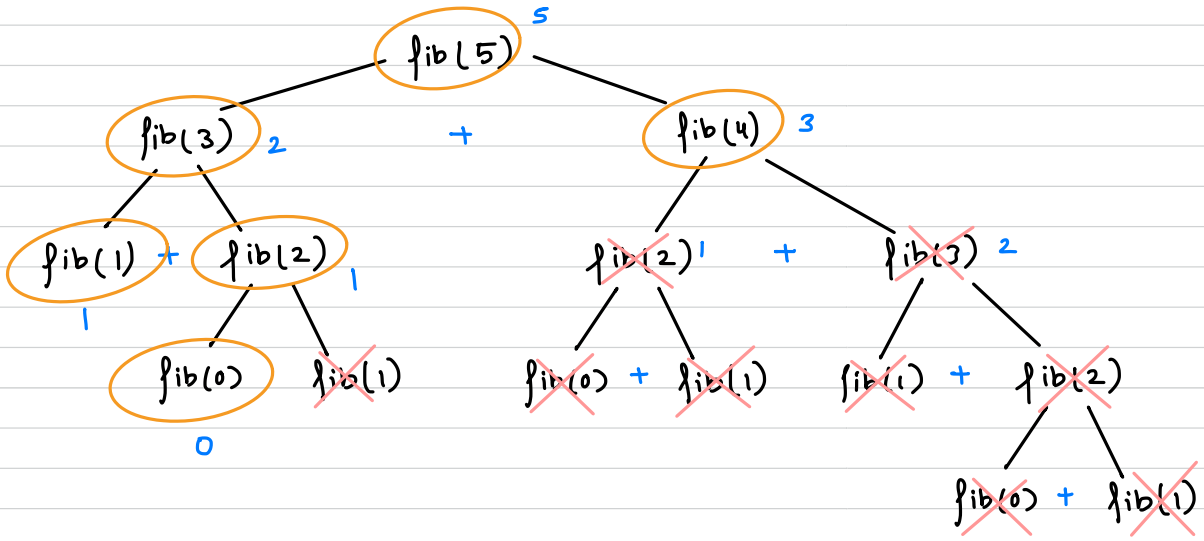
order of (2^n)

This tree lies in the category of excessive recursion because it calls itself multiple times for the same parameters.

To reduce the order of this function, we will write another program using static array and initialize all its values with -1.

F

-1	-1	-1	-1	-1	-1	-1
0	1	1	2	3	5	
0	1	2	3	4	5	6



So, for 5 as n, 6 calls are made.
∴ for n, n+1 calls are made

O(n)

This approach of storing result in an array is called MEMOIZATION.

↳ Storing the result of function calls, so they can be utilized again for avoiding excessive calls

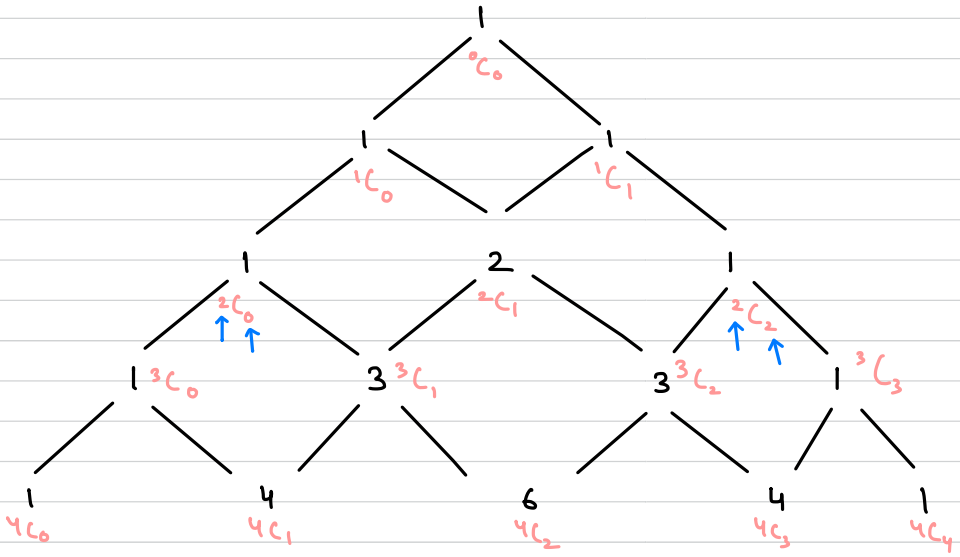
```
int F[10];

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);

        return F[n-2]+F[n-1];
    }
}
```

COMBINATION FORMULA

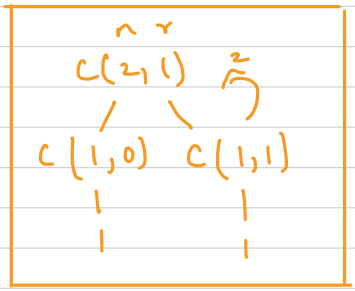
$${}^nC_r = \frac{n!}{(n-r)!\,r!}$$

```
int C (int n, int r)
{
        int t1, t2, t3;
        t1 = fact (n);      ------------ n
        t2 = fact (r);      ------------ n
        t3 = fact (n-r);    ------------ n

        return  t1 /(t2* t3);  ---------- 1
}                                         ---
                                          3n
                                          O(n)
```

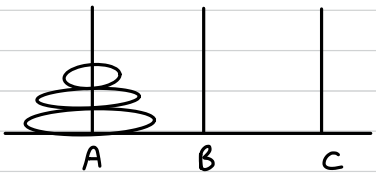
PASCAL'S TRIANGLE




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

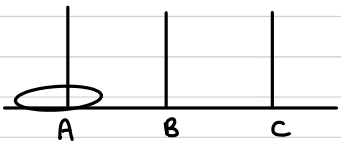


TOWER OF HANOI

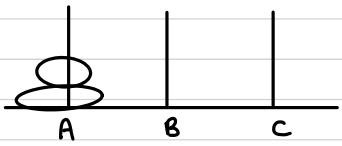


1. Move one disk at a time.
2. No bigger disk can be there above a smaller one.

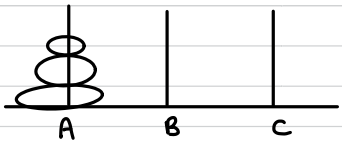
TOH (1, A, B, C)
Move disk A to C using B.



TOH (2, A, B, C)
1. TOH (1, A, C, B)
2. Move disk A to C using B.
3. TOH (1, B, A, C)



TOH (3, A, B, C)
1. TOH (2, A, C, B)
2. Move disk from A to C using B
3. TOH (2, B, A, C).



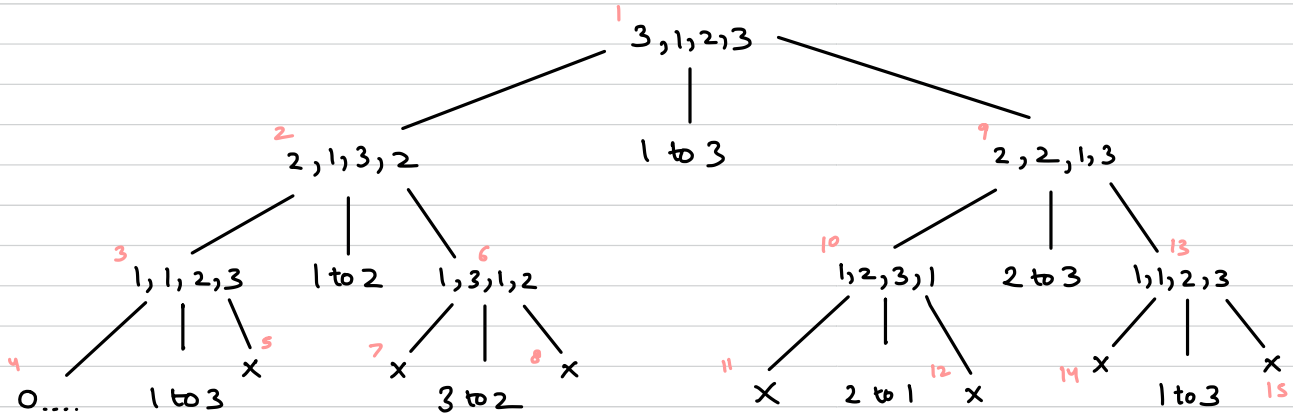
FOR n number of disk.

TOH ($\not{3}^{n}$, A, B, C)
1. TOH ($\not{2}^{n-1}$, A, C, B)
2. Move disk from A to C using B
3. TOH ($\not{2}^{n-1}$, B, A, C).

```
void TOH(int n, int A, int B, int C)
{
    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)

A → 1
B → 2
C → 3



OUTPUT

(1 to 3), (1 to 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	$1+2+2^2 = 2^3-1$

$2^{n+1}-1$

$O(2^n)$

ARRAYS

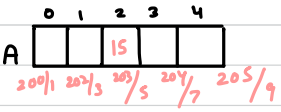
Scalar

```
int x = 10;
```

2 bytes

vector

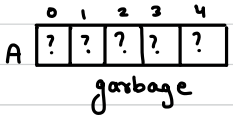
```
int A[5];
A[2] = 15;
```



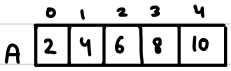
10 bytes

DECLARATION OF ARRAYS

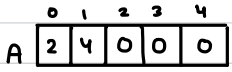
① `int A[5];`



② `int A[5] = {2,4,6,8,10};`

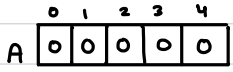


③ `int A[5] = {2,4};`

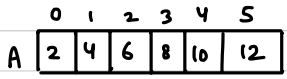


→ Rest of the elements get automatically initialized by 0.

④ `int A[5] = {0};`

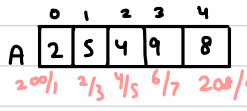


⑤ `int A[] = {2,4,6,8,10,12}`



→ Depending upon the number of elements, size of the array is automatically allocated.

`int A[5] = {2,5,4,9,8};`



→ Array addresses are contagious.

```
printf("%d", A[2]);
printf("%d", 2[A]);
printf("%d", *(A+2));
```

→ OUTPUT
4

```
for(i = 0; i < 5; i++)
    printf("%u", &A[i]);
```

↓ ↓
for printing address.

STATIC VS DYNAMIC ARRAY

Size of the array is static

Size of the array is dynamic

→ Once an array is created, it's size cannot be modified.

In C

→ Size of the array is decided at compile time

In C++

→ The size of the array can be decided at run time also.

```
int n;
cin>>n;
int A[n];
```

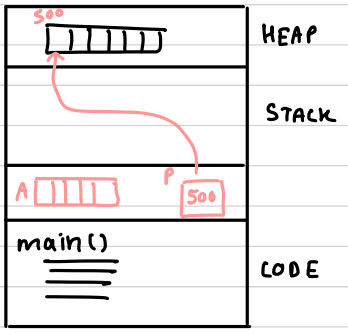
ACCESSING HEAP

```
void main()
{
    int A[5];
    int *p;

C++ p = new int[5];

C   p = (int *)malloc(5 * sizeof(int));
                <stdlib.h>
    :
    :
C++ delete []p;
  C free(p);
}
```

p[0] = 5

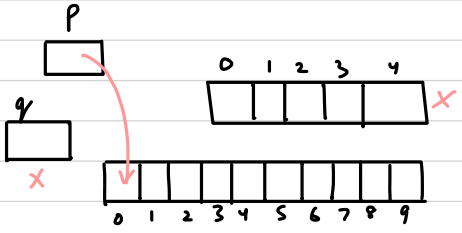
Otherwise MEMORY LEAK

Shortage of memory.



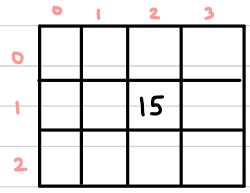
ONE WAY OF INCREASING SIZE OF ARRAY

```
int *p = new int[5];
int *q = new int[10];

for(i=0; i<5; i++)
    q[i] = p[i]

delete []p;
p = q;
q = NULL;
```



2D - ARRAY

① `int A[3][4] = { {1,2,3,4}, {2,4,6,8}, {3,5,7,9} };`



↓ Array will be stored inside stack.

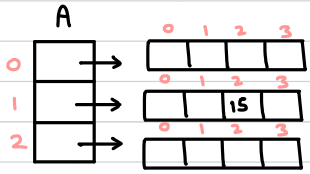
`A[1][2] = 15`

② → Array of pointers

`int *A[3];`

```
A[0] = new int[4];
A[1] = new int[4];
A[2] = new int[4];
```

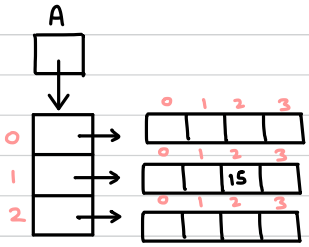
Memory is created inside heap.



`A[1][2] = 15;`

We can access array of pointers in the same way.

③



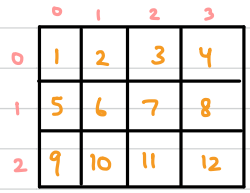

```
int **A;
A = new int*[3];
A[0] = new int[4];
A[1] = new int[4];
A[2] = new int[4];
```

All the memory is allocated in heap.

for column

```
for (i=0; i<3; i++)
{
    for (j=0; j<4; j++)
    {
        A[i][j] = __;
    }
}
```

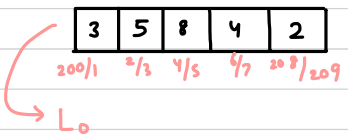
for row



- Elements accessed in order

HOW COMPILER GENERATES FORMULA FOR ADDRESS OF AN ARRAY

int A[5] = {3, 5, 8, 4, 2};



ADD (A[3]) = 200 + 3 * 2 = 206

ADD (A[3]) = L_0 + 3 * 2

ADD (A[i]) = L_0 + i * w → FORMULA

No of operations = 2

Base Address — Index — Size of datatype.

ADD (A[i]) = L_0 + (i-1) * w → Formula, when indices are starting from one onwards

No of operations = 3

More time consuming.

That's why C and C++ do not start array index with 1. Only for one extra operation, time taken by the program will increase and this will make the program slower.

ROW MAJOR

int A[3][4];
m n

Add (A[1][2]) = 200 + [4 + 2] * 2 = 212

Add (A[2][3]) = 200 + [2 * 4 + 3] * 2 = 222

Add (A[i][j]) = L_0 + [i * n + j] * w

4 operations

Add([i][j]) = L_0 + [(i-1) * n + (j-1)] * w

6 operations

COLUMN MAJOR

$ADD(A[1][2]) = 200 + [2*3+1]*2 = 214$

$ADD(A[1][3]) = 200 + [3*3+1]*2 = 220$

$ADD(A[i][j]) = L_0 + (j*m+i)*\omega$

FORMULAS FOR nD ARRAYS

$A[d1][d2][d3][d4];$ 4D ARRAY

Row Major

$$ADD(A[i_1][i_2][i_3][i_4]) = L_0 + [i_1*d_2*d_3*d_4 + i_2*d_3*d_4 + i_3*d_4 + i_4]*\omega$$

Column Major

$$ADD(A[i_1][i_2][i_3][i_4]) = L_0 + [i_4*d_3*d_2*d_1 + i_3*d_2*d_1 + i_2*d_1 + i_1]*\omega$$

ROW MAJOR FOR nD

$$L_0 + \sum_{p=1}^{n}\left[i_p * \prod_{q=p+1}^{n} d_q\right]*\omega$$

COLUMN MAJOR for nD

$$L_0 + \sum_{p=n}^{1}\left[i_p * \prod_{q=p-1}^{1} d_q\right]*\omega$$

(self tried)
(please check)

A[d1][d2][d3][d4]; 4D ARRAY HORNER'S RULE

Row Major

$$Add(A[i_1][i_2][i_3][i_4]) = L_0 + [\underbrace{i_1 * d_2 * d_3 * d_4}_{3} + \underbrace{i_2 * d_3 * d_4}_{2} + \underbrace{i_3 * d_4}_{1} + i_4] * \omega$$

4D → 3 + 2 + 1
5D → 4 + 3 + 2 + 1
nD → n-1 + n-2 + n-3 .... + 1 = $\frac{n(n-1)}{2}$

$O(n^2)$

$$i_4 + i_3 * d_4 + i_2 * d_3 * d_4 + i_1 * d_2 * d_3 * d_4$$

$$i_4 + d_4[i_3 + i_2 * d_3 + i_1 * d_2 * d_3]$$

$$i_4 + d_4[i_3 + d_3[i_2 + i_1 * d_2]]$$

$O(n)$

FORMULA FOR 3D ARRAYS

int A[l][m][n];

ROW MAJOR

$$Add(A[i][j][k]) = L_0 + [i * m * n + j * n + k] * \omega$$

COLUMN MAJOR

$$Add(A[i][j][k]) = L_0 + [k * m * l + j * l + i] * \omega$$

ARRAY ADT

↳ Abstract Datatype

↓

1. Representation of data
2. Operations on data

DATA

1. Array Space
2. Size
3. Length (No of elements)

① `int A[10];`

② `int *A;`
`A = new int[size];`

OPERATIONS

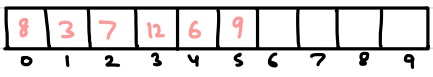
1. Display()
2. Add(x) / Append(x)
3. Insert(index, x)
4. Delete(index)
5. Search(x)
6. Get(index)
7. Set(index, x)
8. Max(), Min()
9. Reverse()
10. Shift() / Rotate()

1. DISPLAY

pseudo code

```
for(i = 0; i < length; i++)
{
        print(A[i])
}
```

Array Size = 10
Length = 6



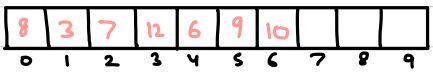
2. Add(x) / Append(x)

```
A[Length] = x;      ——— 1
Length++;           ——— 1
```

f(x) = 2
f(n) = 2

O(n°) = O(1)

Array Size = 10
Length = 7

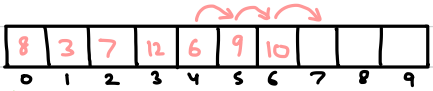


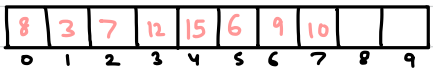
3. Insert (4, 15) (index, x)

```
for (i = length ; i > index ; i--)
{
    A[i] = A[i-1] ;        — 0-n
}
A[index] = x ;             — 1
Length++ ;
```

Inserting at index 8 ↵ ↓ Inserting at 0

Array size = 10
Length = 8





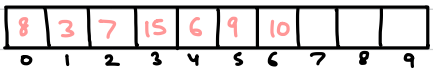
O(1) min O(n) max

4. Delete (3) (index)

```
x = A[index] ;                          — 1
for ( i = index ; i <= Length-1 ; i++)
    A[i] = A[i+1] ;                     — 0-n
Length-- ;                              — 1
```

min = 2
max = n+2

Array size = 10
Length = 7

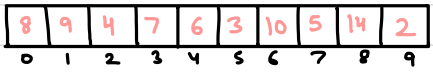


Best O(1) Worst O(n)

Index should be in range of length
we cannot leave empty space between two elements in an array.

LINEAR SEARCH → Searching each element and incrementing

Array size = 10
Length = 10



Key = 5 ← Succesful
Key = 12 ← Unsuccesful

```
for (i=0 ; i < length ; i++)
    if (key == A[i])
        return i ;
return -1 ;
```

Best — O(1)
Worst — O(n)
Average — O(n)

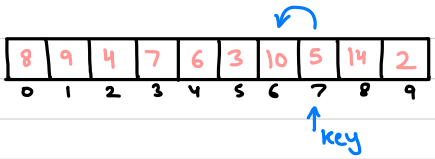
Average

found at location 1
found at 2

$$\frac{1+2+3+\dots n}{n} = \frac{(n+1)n}{2} \cdot \frac{1}{n} = \frac{n+1}{2}$$

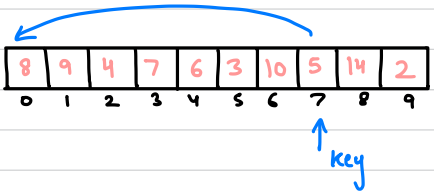
FASTER WAY OF LINEAR SEARCH

1. Transposition : Moving the searched element one step back




```
for (i=0;i<length; i++)
 {
        if ( key == A[i])
        {
            swap( A[i], A[i-1]);
            return i-1;
        }
}
```

2. Move to front / head : The searched element is brought to first position in array.




```
for (i=0;i<length; i++)
 {
        if ( key == A[i])
        {
            swap( A[i], A[0 ]);
            return 0;
        }
}
```