- Arrays
- Matrices
- Linked list
- Stack
- Queues
- Tree
- Graph
- Hashing
- Recursion
- Sorting

"Arrangement of data" and how to utilized

- LV.1
- LV.2
- LV.3

Pointer used for :

1. Accessing Heap
2. Accessing resources
3. Parameter passing

Heap Storage

p = new a[10]

Refrence

```
Int main()
{  a=10;
   Int &r= a; }
```

r = 11
a = 11

[a=r]

Pointer to a Structure

```
Struct Rectangle{ Int length, bredth; }

Int main(){ Struct Rectangle *p;
            Struct Rectangle r;
            *p = &r;
            p→length = 50; }
```

Heap Storage allocation dynamic object

```
Int main(){ Struct rectangle *p;
            P = (Struct rectangle*) malloc(sizeof(Struct Rectangle));
            p→length = 200;
```

Function

» Parameter passing

1. Pass by value
2. Call by address
3. Call by reference. [a=&x]

- Structure as parameter
- Array as parameter

Template Class

Generic Class

```
Template <class T>
Class Arithametic
{  Private:
        T a;
        T b;
    Public:
        Arithametic(T a, T b)
            int add();
            int Sub();
};
        Templatc <class T>
Arithoametic :: Arithametic(T a, T b)
            <T>

Templatc<class T>
T Anthametic<T> :: add()
    {     T c;
        c = a+b;
        return c; }
```

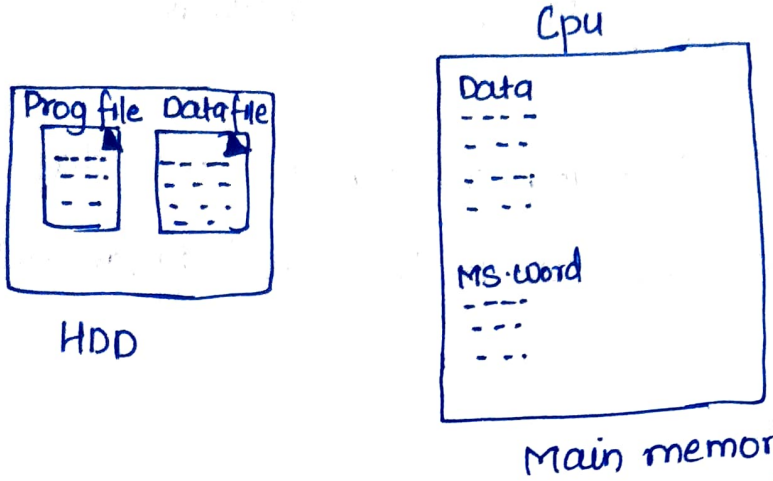
```
Int main()
{   Arithametic <Int> A(10,5);
    Cout << a.add();
```

```
Template <class T>
Class Arithametic
{  Private:
        T a;
        T b;
   Public:
      Arithametic(T a, T b)
        T add()
        T sub() };

template <class T>
Arithametic<T> :: Arithametic(T a, T b)
   {    This -> a = a;
        This -> b = b; }

T Arithmetic<T> :: add()
{       T c;
        return c }
```

Data Structure :

"Arrangement of data"

To utrilised it efficiently.



"Commerical data is stored in the arrangement of table"

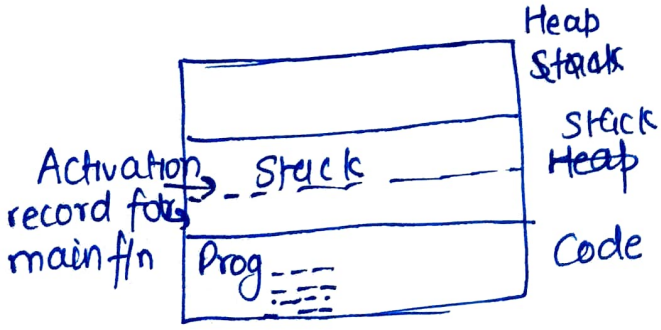
"Data is stored in form of relational model is known as Data base".

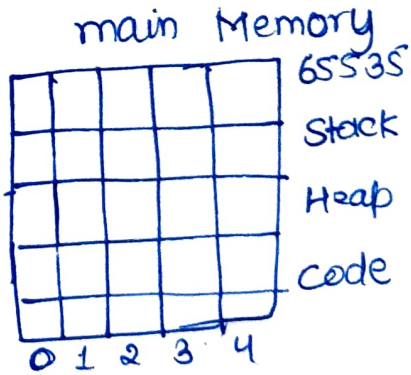
'Data warehouse : Storing data in table disk

algorithum used are Data-mining.

Static Vs Dynamic Memory Allocation

Segment 64kb





C++ P= new int[5];

delocate the memory

delete[] P;

Types of Data Structure.

1. Array
   Linklist } Physical datastructure.

Array: when you know how much data is used.
Linklist: Size can be Increased dynamically.

linear [Stack
         Queues
non-linear [Tree
             Graph
Time saved. Hash Table } Logical data structure.

Types of Data Structure.

1. Array
   Linklist
   > Physical datastructure.

Array: when you know how much data is used.

Linklist: Size can be Increased dynamically.

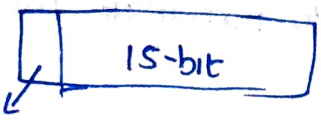
- Linear
  - Stack
  - Queues
- non-linear
  - Tree
  - Graph

> Logical data structure.

- Table based
  - Hash Table

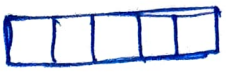
Abstract DataType

1. Representation of data
2. Operation on Data



1 bit for sign — 16 bit

List:



Data:
- Space for storing elem
- Capacity
- Size.

Operation:
- add
- remove
- search
- ⋮

data & operation together known as Abstract Data type.

Time and Space Complexity

$\log_2 n$: dividing all the element by 2.

for loop used for increment by 1.

```
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)
```

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

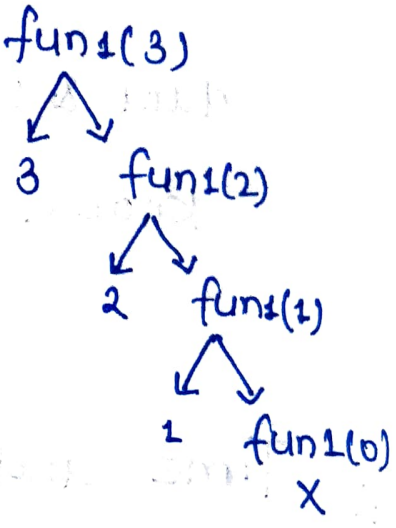
$O(n^2)$

Recursion

"function calling itself known as Recursion"

- Some condition be there to terminate the statement.

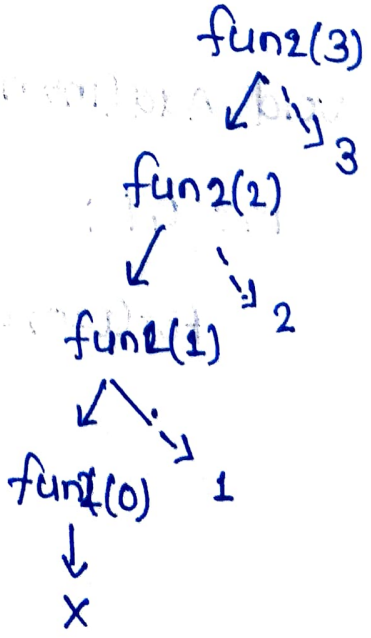
```
void fun1(int n)
{   if(n>0)
    {   Printf("%d",n)
        fun1(n-1) }
}
```



O/P : 3 2 1

```
Void fun2(int n)
{   if(n>0)
    {   fun2(n-1);
        Printf('%d', n);
    }
}
```

O/P



O/P : 1 2 3

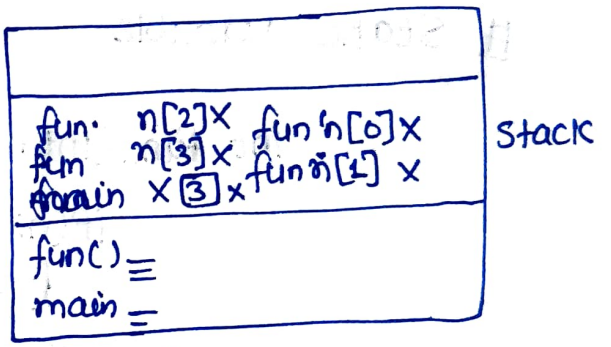
1. Switch on Bulb
2. goto next room

1. goto next room
2. Switch on Bulb.

```
Void fun(int n)
  {
     1. Calling
     2. fun(n-1)*2
     3. Returning;
  }
```

How Recursion is used

```
Void main()
  { int x=3;
    fun(x) }

Void fun1(int n)
1  — if(n>0)
        cout<<n;        cal = n+1;    O(n)
        fun1(n-1);
```



"Recusive fn is memory consuming fn"

```
Void fun(int n)  — T(n)
  { if(n>0)      — 1
    { Printf(n); — 1
      fun1(n+1); — T(n-1)
```

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

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

$T(n) = T(n-1) + 1 \quad -①$

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

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

Assume $n-k=0$

$\therefore n = k$

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

$T(n) = n+1$

$O(n)$

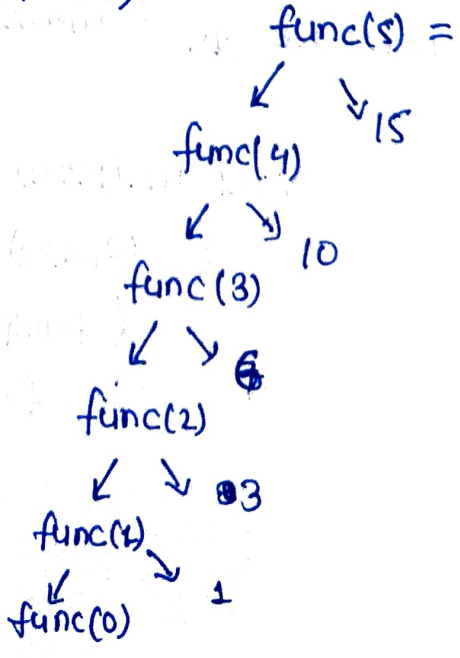
$T(n) = T(n-2) + 1 + 1$

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

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

Static Variable

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

main()
  { int a=5
    Printf("%d", fun(5))
```



```
int func(int n)
{   static int x=0;
    if(n>0)
    { x++;
      return func(n-1)+x;
    } return 0;
    }
int main() { int a=5; cout<< func(a);
```

Since static variable don't copy
So, Value is get changed every time.

~~func(5)~~ x = 1 2 3 4 5

func(4) → 25
func(3) → 20
func(2) → 15
func(1) → 10
func(0) → 5

"Global variable also have single copy".

Types of Recursion

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

Tail Recursion

"If func is called recursively in the last statement then it called Tail Recursion".

```
func(n-1);        func(n-1)+n;
    ✓                  X
```

In tail recursion

loop is better than Tail recursion.

Head Recursion

```
Void func(int n)
{  if(n>0)
       func(n-1);
   }
```

"no processing during calling time".

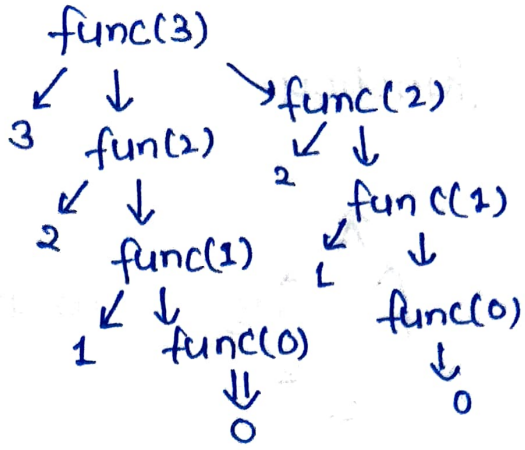
Tree Recursion

"func calling itself more than one time is known as Tree ~~function~~ recursion".

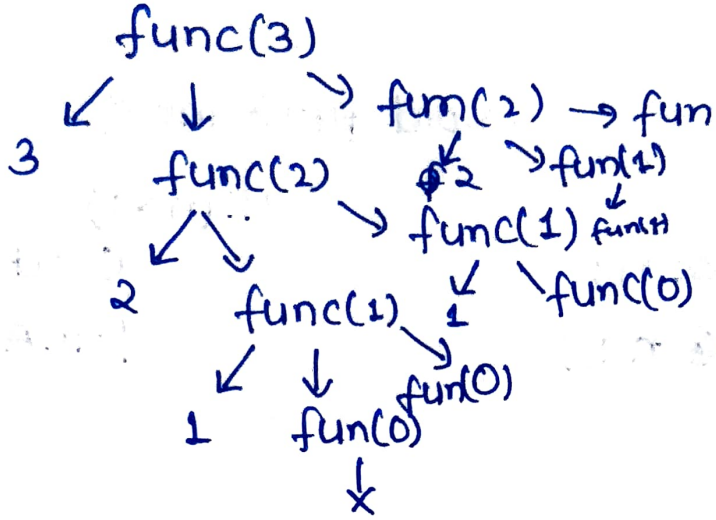
```
func(n)
{ if(n>0)
  {
     func(n-1)
     fun(n-1);
```

```
Void fun(int n)
{  If(n >0)
   {  Printf("%d",n)
      fun(n-1);
      fun(n-1); }
```

n = 3



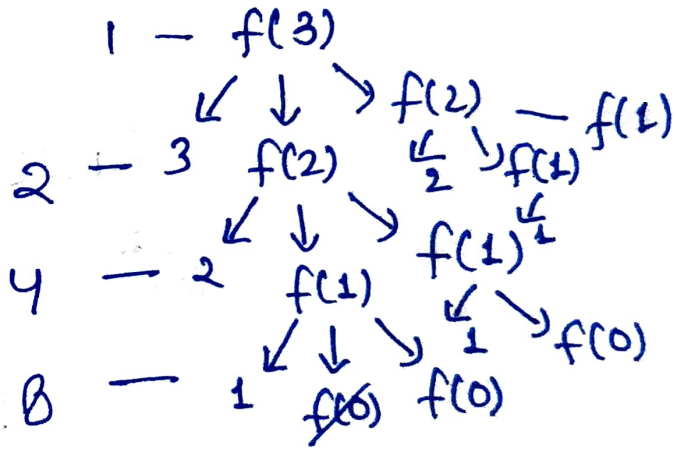
3 2 1 2 1



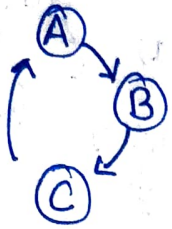
3 2 1, 1, 2, 1, 1

Time : $O(2^n)$

Space :



Indirect function



recursion

```
void A(int n)
{ if(c--->)
  { ==
    B(n-1)
  }
Void B(int n)
  { if(c )
    { c(n-1)
```

```
void funA(int n)
    if(n>0)
        cout<< n;
        funB(n-1);
```

```
Void funB(int n)
    if (n>1)
        Couton
        funA(n/2)
```

func(20)

20 ↙ ↘ funB(19)

19 ↙ ↘ funA(9)

9 ↙ ↘ funB(8)

8 ↙ ↘ funA(4)

4 ↙ ↘ funcB(3)

3 ↙ ↘ funA(1)

1 ↙ ↘ funB(0)

Nested Recursion

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

fun(95)
↓
fun(fun(106))
fun(96)
↓
fun(fun(96))
fun(107)
fun(fun(107))
fun(97)
fun(fun(108))
fun(98)
fun(fun(109))
fun(99)
fun(fun(110))
fun(100)

Sum of n natural number

```
recursion(int n)
{
    if(n>0)
        recursion(n-1);
        sum+=n;
```

```
int Sum(int n)
    if(n==0)
        return 0;
    else
        return Sum(n-1)+n;
```

$(m)^n$

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

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

Taylor Series

$$e^x = 1 + x + \frac{x^2}{\angle 2} + \frac{x^3}{\angle 3} \quad \text{.....} \quad (3)$$

$\frac{(n)^n}{fac(n)}$

$\frac{Pow(4,4)}{fac(4)}$

$\frac{Pow(3)}{fac(3)}$ –

[taylor(n-1) + n]

taylor(n-1) + n

$1 + 3 + \frac{9}{2} + \frac{27}{6}$

4 + 4 5 +

$1 + 3 + 4.5 + \frac{27}{6}$

if(n>0)

$\left[\frac{Pow(3)}{fac(3)}\right]$ * taylor(3-2)

else if(n==0)

return 1;

f = f*f

Tylor Series

$$e = 1 + x + \frac{x^2}{\lfloor 2} + \frac{x^3}{\lfloor 3} + \frac{x^4}{\lfloor 4} \text{ ----}$$

e(x,4)
↙
e(x,3) → $1 + \frac{x^3}{1} + \frac{x^2}{\lfloor 2} + \frac{x^3}{\lfloor 3}$
↙
e(x,2) → $1 + \frac{x}{1} + \frac{x^2}{\lfloor 2}$
↙
e(x,1) → $1 + \frac{x}{1}$
↙
1

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

$$1 + x\left[1 + \frac{x}{2}\left[1 + \frac{x}{3}\left[1 + \frac{x}{4}\right]\right]\right] \quad e(x,4)$$

$r = e(x, n-1);$

e(x,4) f(4) p = p*x; f = f*n; return = r + p/f

e(x,4)
↙
e(x,3)
↙
e(x,2) P=x f=1
↙
e(x,1)
↙
e(x,0) → 1
↓
[n=0] 0

$0 + \frac{1}{1} = 1$

f(n-1)

$f\{2\{f([f(4)])\}\}$ $[1 + \frac{x}{4}]$

e(x,4)
{ 1 + e(x,n-1) }

$1 + \frac{x}{1} + \frac{x^2}{2} +$

2,4

function(4)
{
e(x,4)
r = 1 + e(x,n-1);
e(x,n-1);
return r + e(x,

e(x,4) → [1 + x/4]
↙
e(x,3)
↙
e(x,2) → $1 + x\left[1 + \frac{x}{\lfloor 2}\right]$
↙
e(x,1) → 1 + x
↙
e(x,0) → 1 + 0

$1 + 2 + \frac{4}{2} + \frac{8}{6} + \frac{16}{24}$ (2, 4/3, 2/3)

$5 + \frac{4}{3}$

$1 + 2 + \frac{4}{2} + \frac{8}{6} + \frac{16}{24}\ \frac{2}{3}$

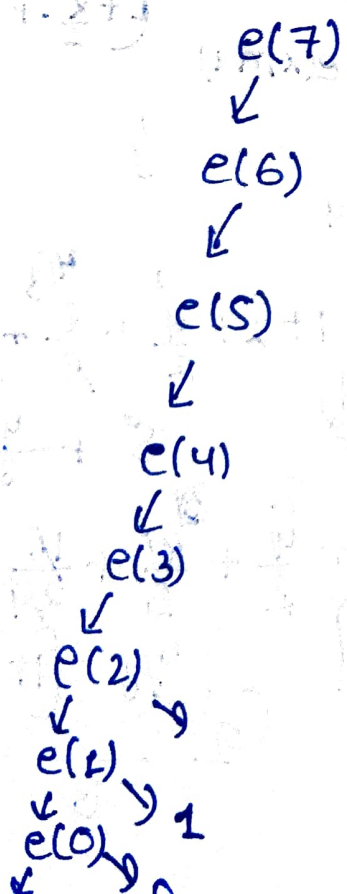
$\frac{4}{3}\left[1 + x/4\right]$

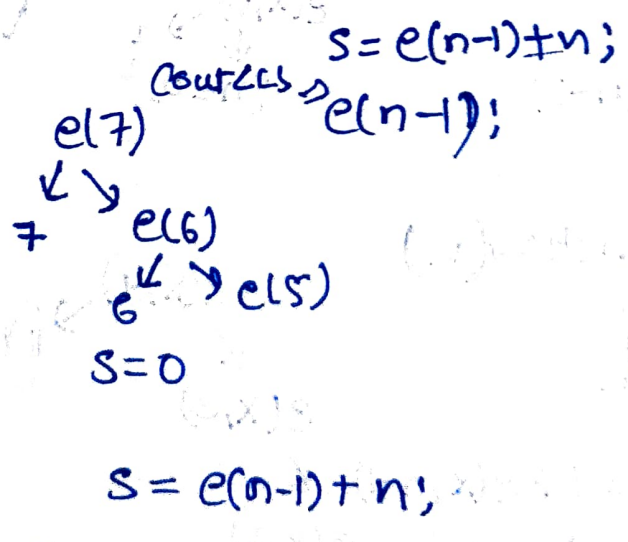
$\frac{2}{3}$

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

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

Sin, Cos series

Fabbinacci Series.





s = e(n-1) + n;

f(n) 0 1 1 2 3 5 8 12

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

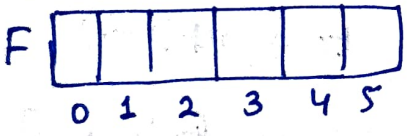
```
fib(int n)
    if(n<=1)
        return n;
    return fib(n-2)+fib(n-1);
```

: Excessive Recursion

$O(2^n)$

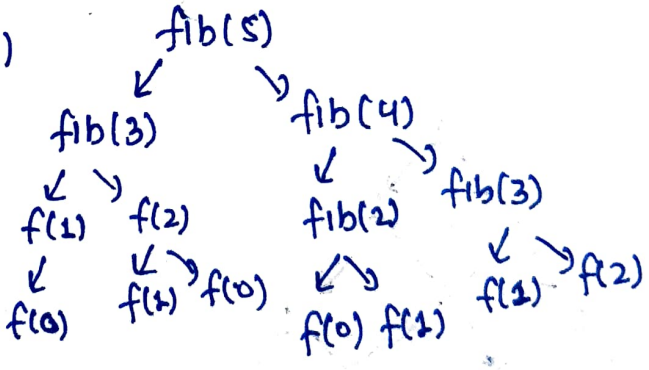
```
int fib(int n)
    int t=0, t1=1, s=1
    if(n<=1) return n;
    for(i=2; i<n; i++)
        s = t0+t1;
        t0=t1;
        t1=s;
    return s;
```

memoization



O(n)

"storing the result".




```
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)
```

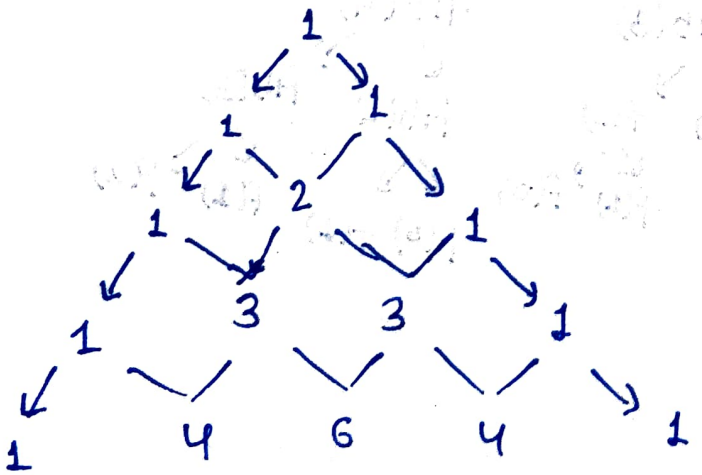
```
Int F[10];
Int fib(int n)
    if (n<=1)
        F[n]=n;
        return n;
    else if (F[n-2]==-1)
            F(n-2)= fib(n-2)
        if (F[n-1]==-1
            F(n-1) = fib(n-1)
        return F(n-2)+F(n-1);
```

Combination Formula

$${}^{n}C_{r} = \frac{\lfloor n}{\lfloor r \lfloor n-r}$$

$${}^{5}C_{2} \quad \frac{\lfloor 5}{\lfloor 2 \lfloor 3}$$

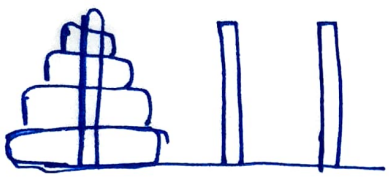
```
Comb(int n, int r)
{ int t1,t2,t3
  t1 = fact(n)
  t2 = fac(r)
  t3 = fact(n-r)
  return (t1/t2*t3); }
```



$${}^{n-1}C_{r-1} + {}^{n-1}C_{r} = {}^{n}C_{r}$$

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

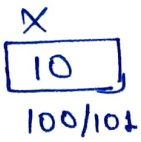
» Tower of Hanoi

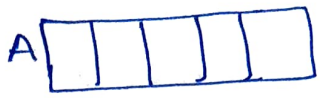


Arrays

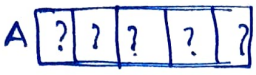
"collection of similar element".

Scalar → int x = 10;





int A[5]



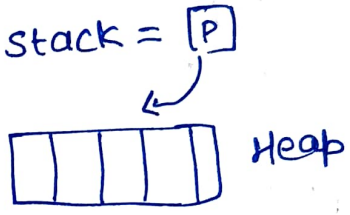
garbage

In C++, array can be created at run-time

C, at compile time.

int *P

C++ P = new int [5];



P = (int *) malloc(5 * sizeof(int))

C++ delete [] P;

C free(P)

* Stack array, size can't be increased or decreas

But heap array can be.

int *p = new int [5]

int *q = new int [8]

for(i=0; i<5; i++)

q[i] = p[i];

delete[]P;

P = q;

q = NULL;

memcopy

int *A[3];

int* A[3];

```
int *A[3];

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

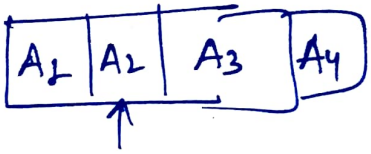
```
int **A                    A[ ]

  A = new int*[3]

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

[10][15]

A[i][j]

100

A[i][j] [n*i+j] 2



a[2] = a[2];

An

[A4 = A3]

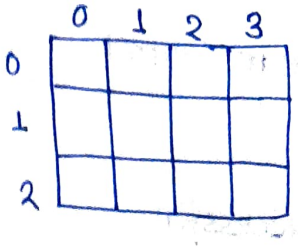
[A3 = A2]

a[2] = a[2]

Arrays in compiler

Int A[3][4]

1. Row-major mapping
2. Column-major mapping



A[1][2]

$Add_r([i][j]) = lo + [i*n + j]*w$

Add + [1x4 + j]*2

A[1][2] ≡ 200 + [2*3 + 1]*2

1. Array ADT

Operations:

```
Struct Array {
        Int A[10];
        Int Size;
        Int length; };
```

```
Class Array {
    Int *A;
    Int Size;
    Int length; }
```

Cin >> &arr.Size

```
Void display ( Struct Array A)
  { Int i;   for(i=0; i<arr.length; i++)
                 cout<< arr.A[i]; }
```

```
Void Append(Struct Array *arr, Int x)
        Int(arr→length < arr→size)
                arr→ A[arr→length++] = x
```

```
Void Insert( Struct Array *arr, Int Index, Int x)
    { Int i;
      if(Index>0 && Index<=arr→length)
            for(i= arr→length; i>Index ; i--
                  arr→A[i] = arr→A[i-1];
      arr→A[index]=x;
      arr→length ++ }
```