DSA SUPREME

By Ujjwal Maheshwari

https://www.linkedin.com/in/Ujjwal2327

- —> more important note
- —> Heading
- —> Subheading or important point
- —> Data Structure
- —> Complexities

IMPORTANT POINTS

- Focus on work on skill and build networks those work in companies and can talk to HR for you
- Internship- do if you want to learn or convert it as PPO
- Rough sols. and dry run are very important
- Do documentation to promote and mail everything you dismss with HR or team
- Think twice, Code once
- To clarify any code you are confused, use cout statements every where to know what is going on in the code
- Code all approaches you can think of and can find & understand from google
- Revise all incorrect & skipped questions in quizes regularly
- Watch sol. only after attempting the question
- Interviewer will ask Time & Space complexity after every sol. you give

→ 2 websites
  → cppreference
  → cplusplus.com

→ Think on paper → with 5 testcases atleast

→ Write readable codes

→ In interviews, tell approaches of questions using example

→ Signs of beginner → no logic build
  cant solve new ques
  forget the approach
  memorizing the ans

↓

Improve → with more no. of questions
  with time
  with dry run on mind logic, not on code
  with your new approach (even brute force)
  with a lot of practice on syntax
  with consistency

→ Focus on placement, not on feeling

You are not studing to get fun

You are studing to get placed

First placement, then fun and interest.

Also focus on health of you and your parents

Bhaad me gya interest yaar, placement ke baad krna

Jo duniya sunna chahati hai, use sunado

Growth is important

Week 4 [Connect] Class 1:45 - 2:01

→ 20-30 interview experience

↳ to break Google pattern

→ Web Dev and DSA ⟶ both are important

CP ⟶ do if you enjoy it.

→ At least learn one more approach if you are doing questions with map

Many times if you tell map approach, interviewer asks another approach

- Tag all khatarnak questions
  - Otherwise you will forget them after some time
- Revise and code all, every after 2 weeks
- For 6th sem student (like me),
  - At least do potd on leetcode everyday along with course
- Make notes
- In dev, you should have at least 2 major projects
  - 1 → in dot batch
  - 2 → group project if you want
- Focus on Networking,
  After 5 months, you should have atleast 2 friends in every company
- All questions doing in batch, should be on your tips

- Focus on accuracy more than speed
- Resume is made company specific
- Make sure others dont take your credits
- Higher position in career $\propto \frac{\text{Development}}{\text{DSA Complexity}}$
- Interview experience before interview
- Flex se kuch nhi hota
  Comparision se kuch nhi hota
- Dont forcefully try to increase the speed of DSA learning,
  Try to increase the number of hours spent everyday in DSA learning
- if ques seems tough in another datatype but easy in another datatype, then first do that in later data type and after solving the sol. change datatypes

→ In interview, you must have clarity and strong concepts and strong argument behind the things you say,
whether the ans comes right or wrong is another thing

→ Start writing code for smallest problem / test case and then eventually find solution pattern

→ Dont rush to make an optimised code, start from brute force

IMPORTANT C++ NOTES

→ -1 in binary 1 1

-2^{31} in binary 10...0

→ In left shift, vacant bit will be filled by 0

In right shift, vacant bit will be filled by MSB

1 ⟶ in -ve no.

0 ⟶ in +ve no.

(-1) >> n ⟶ (-1) no change

⟶ any no. of times

→ int a = 5;

cout << (++a) * (++a);

a * (++a), a = 6

a * a, a = 7

7 * 7 = 49

cout << (a++) * (++a);

5 * (++a), a = 6

5 * a, a = 7

5 * 7 = 35

cout << (++a) * (a++);

a * (a++), a = 6

a * 6, a = 7

7 * 6 = 42

cout << (a++) * (a++);

5 * (a++), a = 6

5 * 6, a =

30

→ Continue cannot be used in switch case
can only used in loops

→ int main (){

return 0; ⟶ returns to OS
}
0 is used to represent successful execution

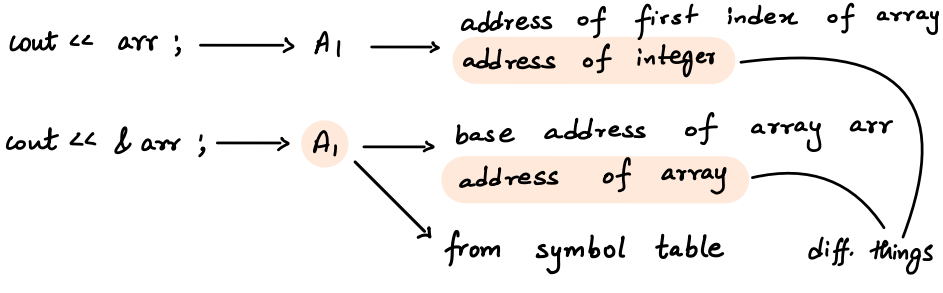
→ A cpp file cant have more than 1 main function

→ main () cant have return type other than int in offline compiler

→ % is heavy operator, so try to use it less

→ array name returns address of first index

Integer Array-



→ To find max., initialize ans with INT_MIN

To find min., initialize ans with INT_MAX

→ xor ⟶ cancels out same element

→ 0 ^ ans = ans
- 0 ^ 1 = 1
- 0 ^ 0 = 0

→ Mapping of 2D array with memory 1D array

linear_index = c * i + j

- c → no. of cols
- i → row index
- j → col index

i = linear_index / c

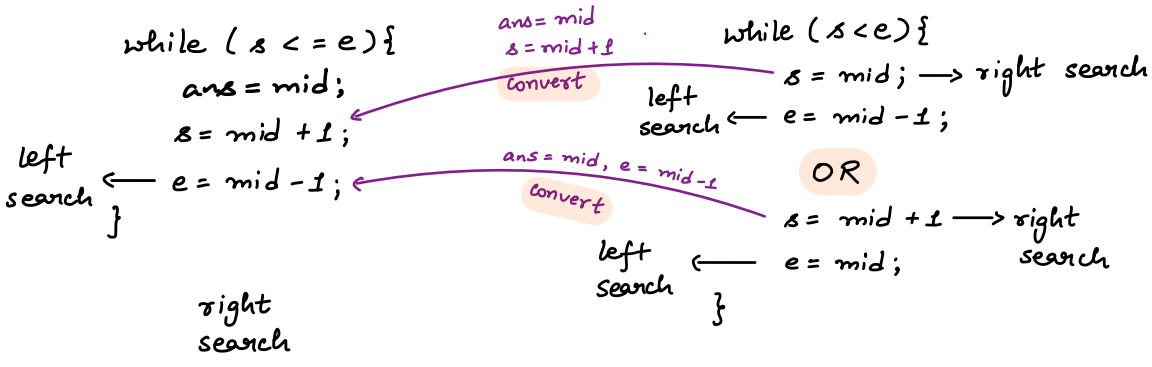
j = linear_index % c

→ Arrays and 2D arrays of any datatype
- ↳ pass by reference

→ Search space
- ↳ range of ans (start and end) in binary search questions

→ store mid in ans if needed in binary search ques

→ In binary search questions



→ cout << (int)(-3.74);

└→ -[int(3.74)]

└→ -(3) → -3

cout << (-22)/7; → -3

cout << 22/(-7); → -3

→ In ASCII → '0' → 48

'A' → 65

'a' → 97

→ In sorted array, try to apply binary search or 2 pointer / 3pointer approach

→ To maximize or minimize
try to use binary search using concept of search space

- → find search space of answer
- → find mid
- → if isPossibleAns(mid) → go to left / right acc. to the ques.

You can use above approach also in other ques like find the duplicate number in an array having elements range 1 to n

→ Week 6 Lecture 2

→ Magical line for recursion
- → Ek case solve krdo baaki recursion smbhal lega
  - → Just believe on it, dont doubt on recursion

→ Week 7 Lecture 2

→ Week 7 Lecture 3

→ If the recursion ques seems tough,
use void function and pass ans by reference in function

→ If in ques. array elements are in range [1, n] or [0, n), try to treat elements as indices of an array
(Week 3 Assignment)

→ Try to make cnts[max element present in array] instead of making cnts[max limit given in question]
Array more than 10^6 ints cant be formed

→
```
int digit = 7;
string s;
s.push_back(digit + '0');
cout << s;  ——→ 7
```

```
s.push_back(digit);
cout << s; ——→ contains
              char with
          ascii value = digit
```

'7' to 7 ——→ '7' - '0'

7 to '7' ——→ 7 + '0'

→ Generally in linked list question, you were asked to play with links / pointers without changing data

→ In linked list, when you have to go from right to left, you have 2 options

1) reverse the linked list

2) use head recursion

→ If we pop a data set after pushing in stack, we get the answer in reverse order

So to reverse anything, you can use stack

→ sets can be used to store only unique / distinct elements

ERROR / BAD PRACTICE

If let's go page

⟶ 13

31, 33, 43, 70, 71, 72, 73, 75, 76, 79, 06, 07, 09, 90, 93, 95

⟶ 14

104

LET'S GO

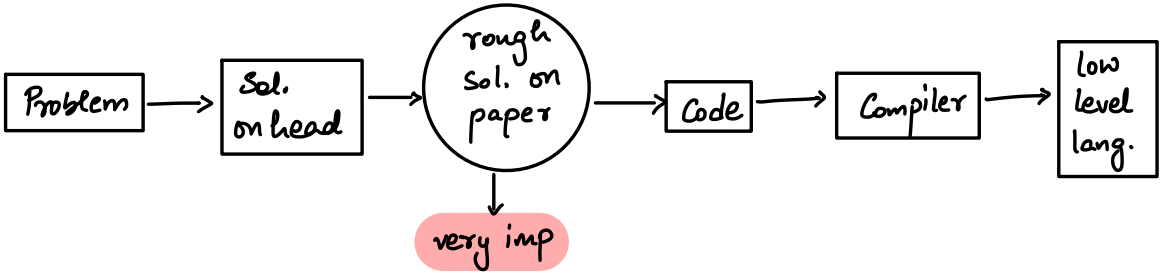
W1-L1

Thought process to solve a problem-

→ Understand a problem

→ input values

→ find approach

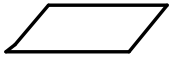


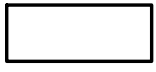
Algorithm- Sequence of steps

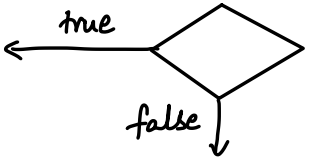
Flowchart- Graphical representation of algo

Components-

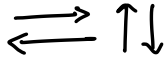

terminator for start /end

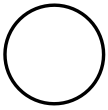

for input /output read /write


computation / process / declaration



decision making block takes condition


flow



Connector takes function

Pseudo Code- Generic way of writing algo

Dry Run → Very important to understand any topic

W1-L2

IDE- Replit, VS-Code

```
# include <iostream>
using namespace std;
int main (){
    cout << "Namaste Bharat";
}
```

preheader file contains implementation of keyword

using standard namespace implementation of cout choosing from multiple types of namespace

region where scope of keyword is defined

to end any statement

string

used to print on console/ standard display

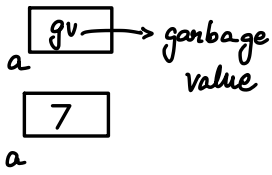
`cout << endl;` → for next line

`cout << '\n';` → for next line

int a ; ⟶ a is an integer

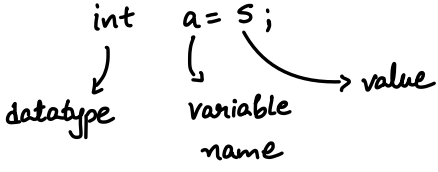
cin >> a ; ⟶ input a from user

ex-7



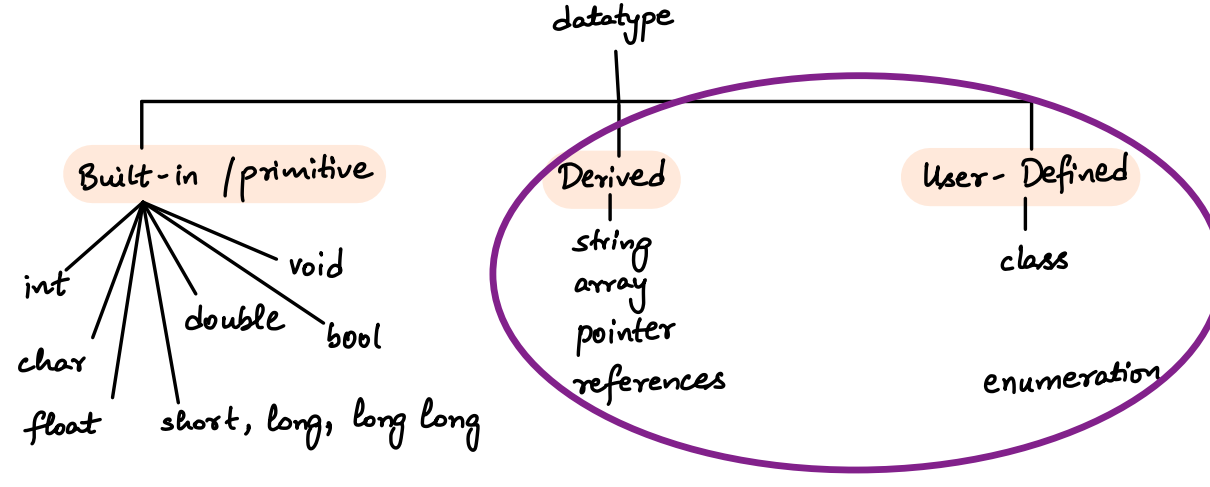
Variables

named memory location



Datatypes

type of data



int - 4 byte - 32 bits in memory

- → -2^{31} to $2^{31}-1$ in signed int
- → 0 to $2^{32}-1$ in unsigned int

char - 1 byte - 8 bits in memory

- → 2^8 different chars.

ASCII
↳ char maps with numerical ASCII value

char ⟷ ASCII value ⟶ store in memory

bool → 1 byte → 8 bits

true - 1
false - 0

↳ because minimum addressable memory is 1 byte
We cannot address 1 bit in memory

float → 4 byte → 32 bits

double → 8 byte → 64 bits

long long → 8 byte → 64 bits

short → 2 byte → 16 bits

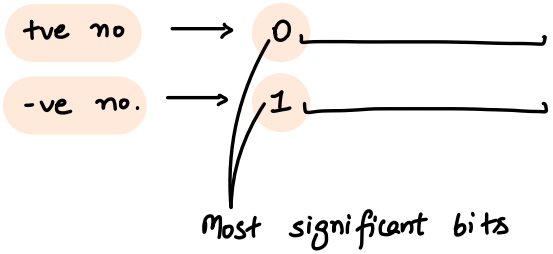
long → 4 byte → 32 bits

How data is stored

```
int a = 5;
```

↳ 32 bits 0... 00101

29 bits



How -ve number is stored in memory

In 2's complement form

→ 1's complement + 1

→ reverse all bits

int a = -7;

7 → 0.....00111 } 32 bits

ignore -ve sign

find binary equivalant

1's (7) → 1.....11000

find 2's complement

2's (7) → 1.....11001

→ this is how -7 will be stored in memory

How to read -ve no. present in memory

→ take 2's complement

1....11001

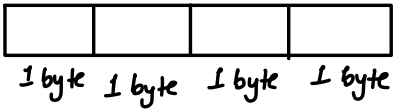
→ 1's complement → 0...00110

2's complement → 0...00111

→ +7

-7

Interesting problem



how computer know these are 4 chars or a single integer

- Using datatype
  - tell 2 things
    - type of data used
    - space used in memory

Signed vs Unsigned

- Unsigned → 0, +ve
- Signed → +ve, -ve, 0
  - by default

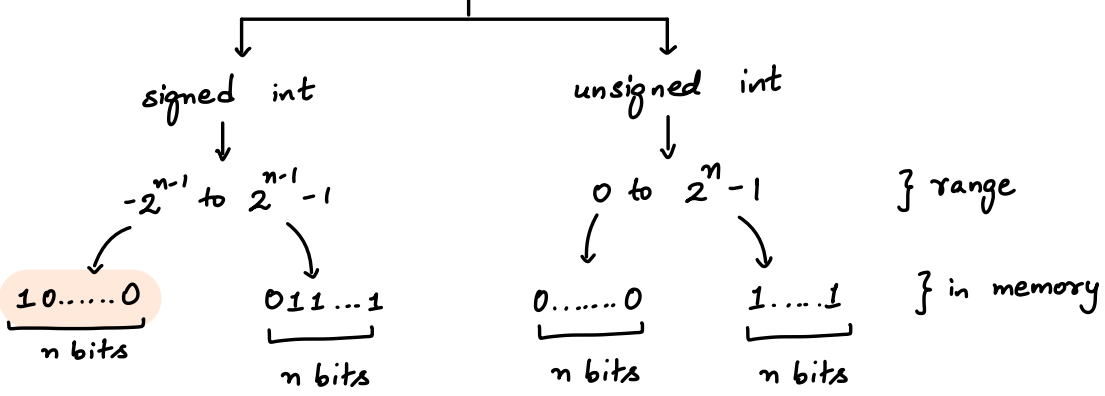
int - 4 byte - 32 bits in memory

- total no. of combinations - 2^{32}
  - signed int
    - -2^{31} to $2^{31}-1$ } range
      - 1 0...0
      - 0 1 1 ... 1 } in memory
      - 2's → 1 0...0 → -2^{31}
  - unsigned int
    - 0 to $2^{32}-1$ } range
      - 0.....0
      - 1....1 } in memory

General Formula

n bits in memory

↳ total no. of combinations = 2^n



-1 → 1...1 (n bits)

Typecasting

↳ convert one type of data to another

implicit typecasting

```
ex- char ch= 97;
    cout << ch;
```
→ a

explicit typecasting

```
ex- char ch = (char) 97;
    cout << ch;
```
→ a

overflow ex-

```
char ch = 9999;
cout<< ch;
```

9999 → (binary conversion) → 100 111 0000 1111

stores only last 8 bits

so ch stores 00001111 in memory

↓

7

↓

acc. to ASCII table

Operators -

Arithmatic Operator

↳ +, -, *, /, %

int op int ⟶ int

float op int
int op float
float op float
} ⟶ float

double op int
int op double
double op double
float op double
double op float
} ⟶ double

bool op bool ⟶ int

bool act as 0 or 1

3 → int
3.0 → float / double

by default ⟶ not int

cout << sizeof(3.0);

↓

8

Relational Operator

(a op b)

>, <, >=, <=, != , ==

these are different things

Output – 0 or 1

false ↙ ↳ true

Assignment Operators

=

Logical Operators

↳ when you have multiple conditions

(a && b) ⟶ and ⟶ true if both are true

(a || b) ⟶ or ⟶ true if any one is true

(! a) ⟶ not ⟶ negate the result

Output - 0 or 1

false ↙ ↘ true

(cond 1 && cond 2 && cond 3)

if cond 1 is false

compiler will not check further as ans will already false

(cond 1 || cond 2 || cond 3)

if cond 1 is true

compiler will not check further as ans will already true

Conditions

W1-L3

```
if (cond.){
    execute
}
```

if

```
if( cond ){
    execute 1
}
else {
    execute 2
}
```

if - else

```
if (cond 1)
    execute 1
else if( cond 2)
    execute 2
```

if - else if

```
if (cond 1)
    execute 1
else if ( cond 2)
    execute 2
else if (...
else
    execute n
```

if - else if - else

```
if (cond 1)
    execute 1
else {
    if ( ) { }
    else ( ) { }
}
```

nested if - else

Loops

↳ to do something repeatedly

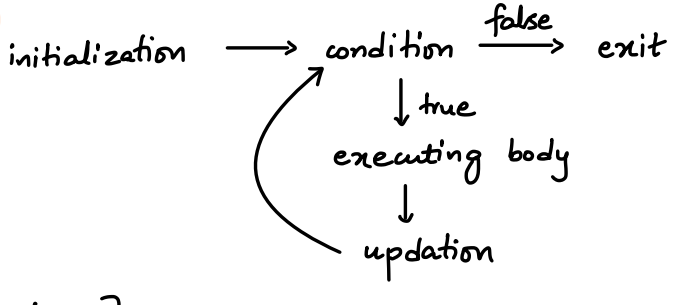
for - loop

initialization, condition, updation

```
for ( int i=0 ; i<5 ; i=i+1 ) {
    cout << "Love" ;
}
```

→ executing body

flow -



initialization
condition
updation

} none is mandatory

one or multiple i, c, u can be added

multiple c → i>5, i<10; → i>5 && i<10

patterns -

Generally 2 loops ⟶

```
outer loop(){          → for rows
    inner loop(){      → for cols
    }
    cout << endl;
}
```

→ a op= b ⟶ a = a op b

op ⟶ +, -, *, %, /, <<, >>

cin in if( )

```
int num;
if(cin >> num){
    cout << "hello";
}
else{
    cout << "hi";
}
```

it will not give error

output -

hello

for all input values of num

↓

0, +ve, -ve

cout in if( )

```
int num = 0;
if(cout << num << endl){
    cout << "hello";
}
else{
    cout << "hi";
}
```

it will not give error

output -

0

hello

for all values of num

↓

0, +ve, -ve

HLL - High level language

↳ human readable and user friendly

W1-L4

C++, C - Middle Level language

namespace ⟶ to avoid collision

↓

multiple definitions of a single keyword

hierarchy

↳ various namespaces

↓

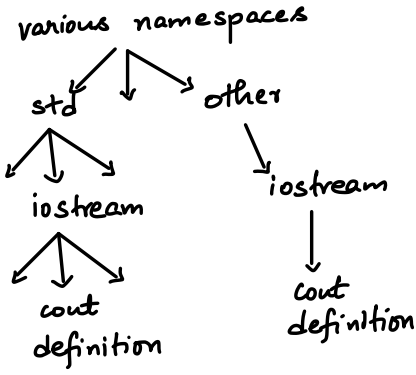
std namespace

↓

iostream preheader file

↓

keyword definition




```
float f = 2.0 + 100;
cout << f;
```

⟶ output - 102 or 102.0

compiler dependent

```
float f = 2.7;
int n = 157;
int diff = n-f;
cout << diff;
```

output -

154

explanation -

n-f = 157 - 2.7 = 154.3

int diff = n-f

diff = 154

W1-HW

ternary operator -

↳ syntax

variable = (condition) ? expression2 : expression3

(condition)? variable = expression2 : variable = expression3

2 and 3 cant be statements, they must be exp.

ex- return (a > b) ? a : b ; ⟶ CORRECT

(a > b) ? return a : return b; ⟶ WRONG ERROR

these are statements, not expression

by default -

cout << sizeof (2.3); ⟶ 8
↳ float

cout << sizeof (a); ⟶ 4 ⟶ int
↳ $-(2^{31}-1)$ to $2^{31}-1$

cout << sizeof (-2^{31}) ⟶ 8
↳ long long

patterns

W2-L1

↳ how to think

→ finding formula for rows and cols

n=5

row	stars
0	0
1	0
2	1
3	2
4	3

→ formula- 0 to < n-1

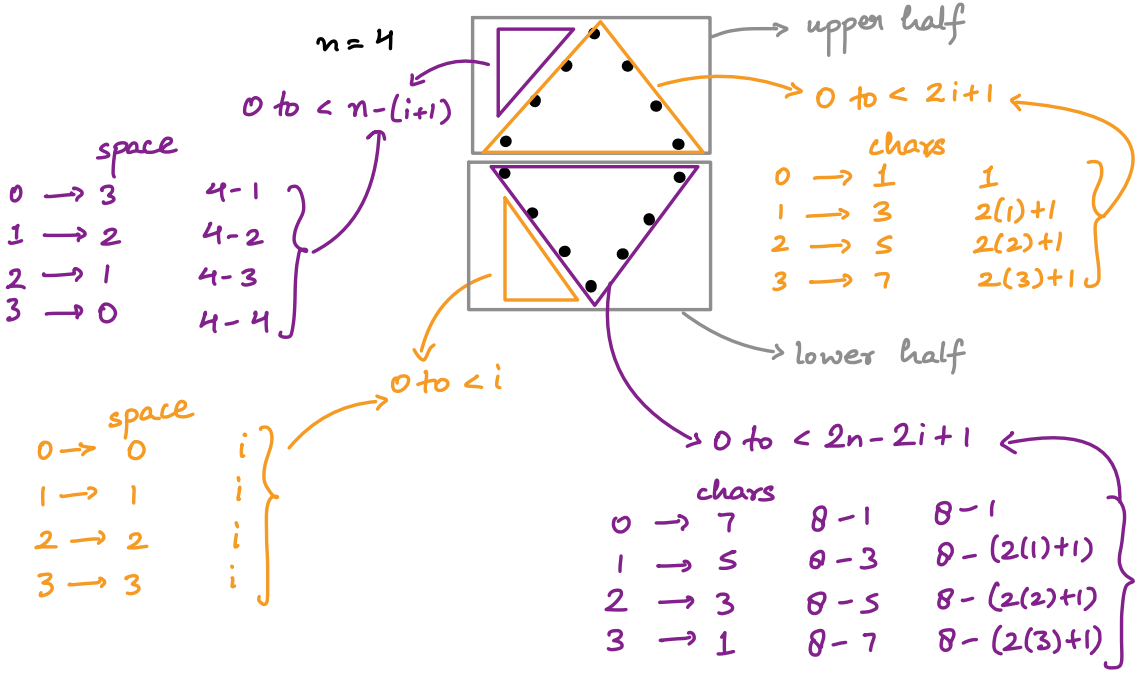
n-1	no. of times loop runs
-1	0
0	0
1	1
2	2
3	3

as condition fails (0<-1)

→ to do anything n times

↳ for(i=0 ; i<n; i++){}

→ break the complex patterns



W2-L2

Bitwise Operators

↳ use on bit level

And	(a & b)	1 if both bits are 1
Or	(a | b)	1 if any or both bits are 1
not	(~ a)	negate the result
xor	(a ^ b)	same values → 0 diff. values → 1

~5

↳ 5 → 0...0101

~5 → 1...1010

↳ how compiler read this

↳ 2's complement

0....0101 → 1's complement

0...0110 → 2's complement

So ~5 = -6

Left and right shift operators

<<

shift all bits to left

* by 2 (not in every case)

↳ if MSB is 1 and 2nd MSB is 0

>>

shift all bits to right

/ by 2 (not in every case)

↳ in -1

a = a << b a left shifts, b times → result → $a \times 2^b$

a = a >> b a right shifts, b times → result → $\frac{a}{2^b}$

b cant be -ve

↳ in case of -ve

↳ gives gv

```
a = 5;
a = a << 1;      a = 10

a = 5;
a = a << 2;      a = 20
```

in left shift → filled with 0

in right shift → filled with 0 or 1

in +ve no. ← 0 or 1 → in -ve no.

right shift in -ve number

-ve no. in memory → 1 .....

↓ right shift

11....

↳ signed bit is used to fill the vacant bit

ex-

5 → 0...0 101

-5 → 1...1 011

-5 >>1 → 1...101 → -3

-1 >> 1 → -1

left shift in number where MSB is 1 and 2nd MSB is 0

no. → 10.. → -ve no.

left shift ↳ 0....

↳ +ve no.

Pre- Post → Increament / Decreament Operator

pre- increament

↳ ++a

↳ first increament by 1, then use

post - increament

↳ a++

↳ first use then increament by 1

pre- decreament

↳ --a

↳ first decreament by 1, then use

post - decreament

↳ a--

↳ first use then decreament by 1

```
int a= 5;
cout << (++a) * (++a);
```

output -

49

↳ due to operator precendence

↳ Links.txt in repo

break and continue

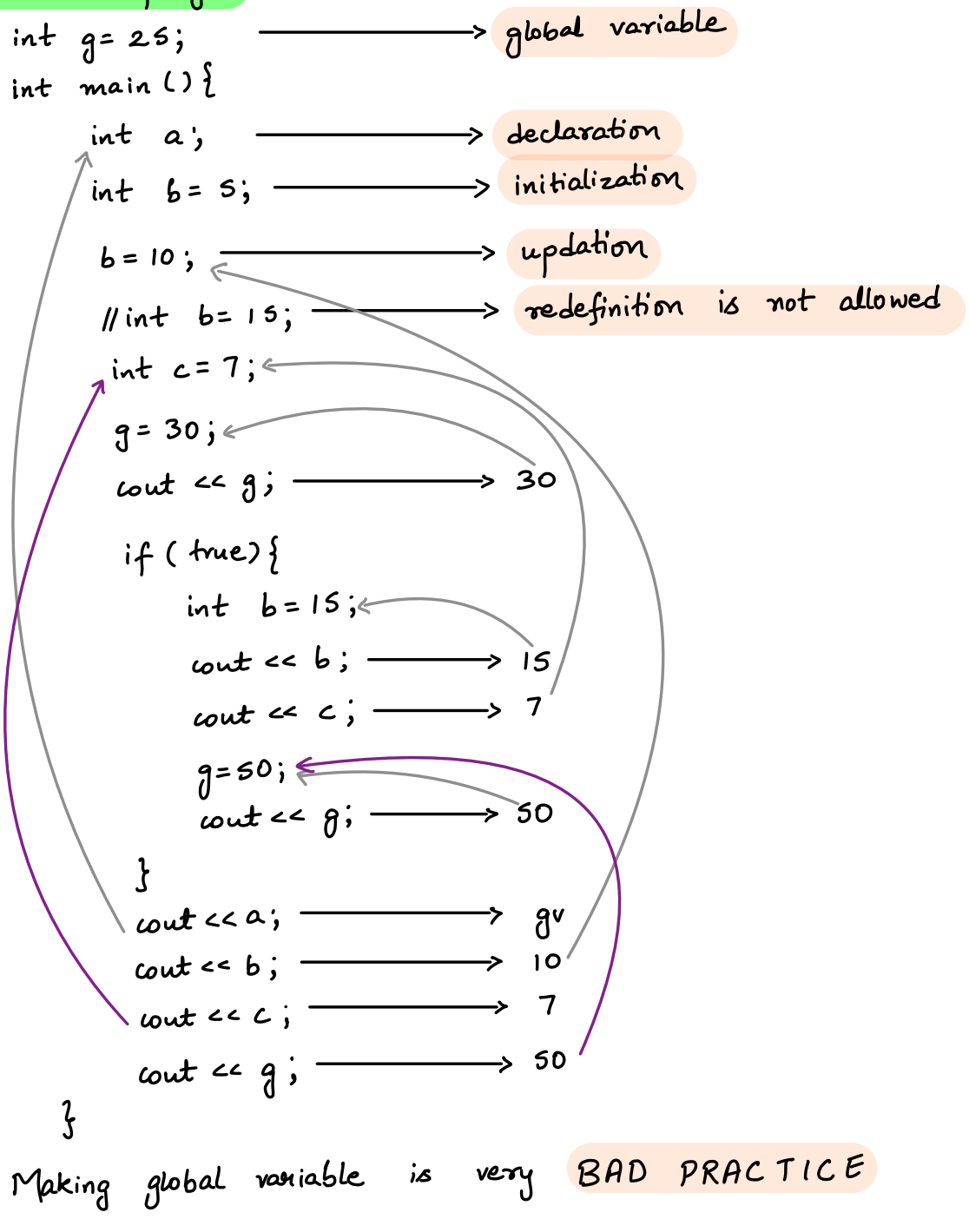
break

↳ exit from that loop

continue

↳ skip that iteration

Variable Scoping -



Making global variable is very BAD PRACTICE

Operator Precedence

- order of priority of operator
- no need to remember
  - use brackets properly

Switch Case

can also have nested switch case

```
switch (expression) {

    case value1 :
        executing body 1
        break;
    case value2 :
        executing body 2
        break;
    :
    case value n :
        executing body n
        break;

    default :
        executing body
}
```

not mandatory

- without break
  - all below executing body will also execute

- continue cannot be used in switch case
  - can only use in loops

W2-L3

Function -

- program linked with well defined task
- why
  - reusable
  - readable
- without
  - bulky
  - lengthy
  - buggy if mistake in any place

syntax -

```
return type functionname ( input parameters) {
    executing body
}
```

void → empty / no value

void x ; ⟶ ERROR

cout << sizeof(int) ⟶ 4

cout << sizeof(void) ⟶ ERROR

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

return 0; ⟶ returns 0 to Operating System

0 is used as means of successful execution

- a cpp file cant have more than 1 main functions
- main cant have return type other than int in offline compiler

Function Call Stack

function call ⟷ function invoke

Stack
- Last In First Out

tells
- what functions are active
- which function calls which
- local variables of function
- return type of function

ex-

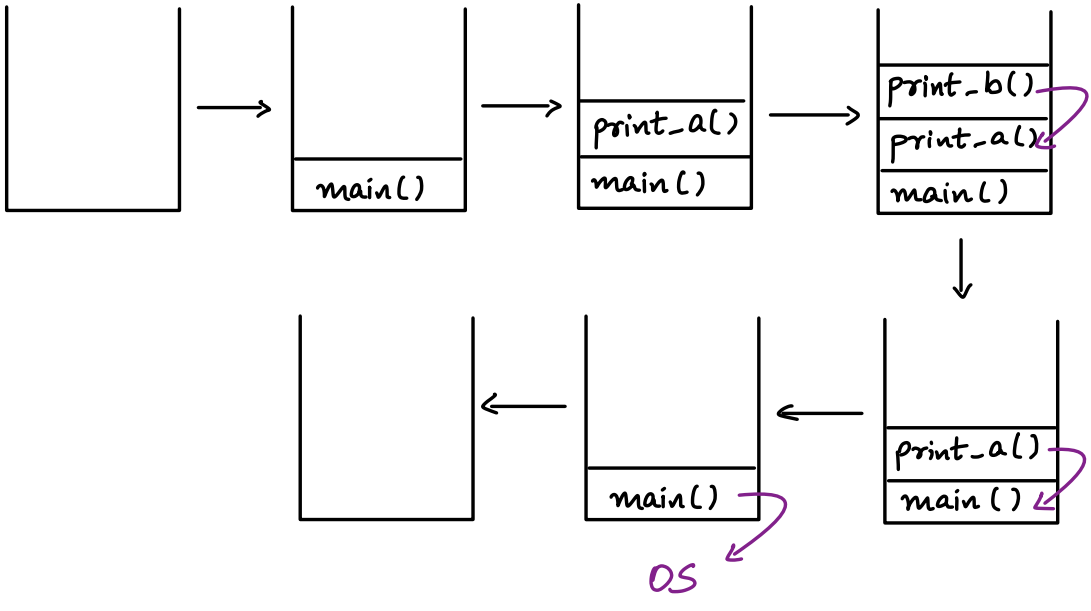
```
int main(){
    int a=5;
    print_a(a)
    return 0;
}
```

```
void print_a(int a){
    cout<< a;
    int b=3;
    print_b(b);
}
```

```
void print_b(int b)
{
    cout<< b;
}
```

return to OS

Output - 5 3



Pass by value ⟶ value is passed to another variable

↳ a copy will be created of variables

```
int main() {
    int a = 5;
    printNumber(a);
    cout << a;
}
```

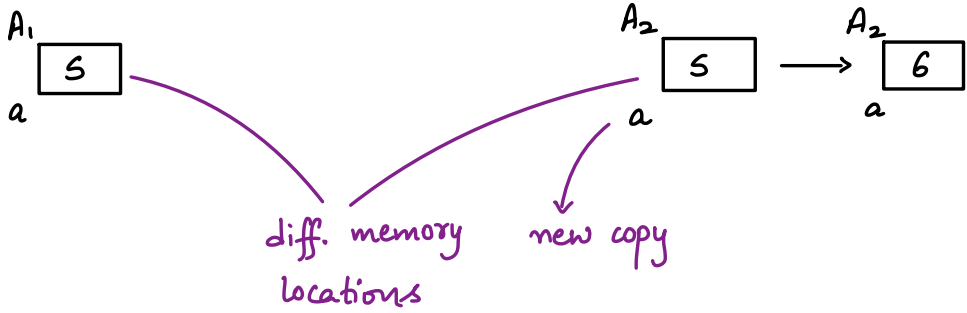
a → argument

cout << a; → 5

```
void printNumber(int a){
    cout << a;
    a++;
    cout << a;
}
```

parameter

cout << a; → 6



Address Of Operator &

```
int n = 5;
cout << &n;
```

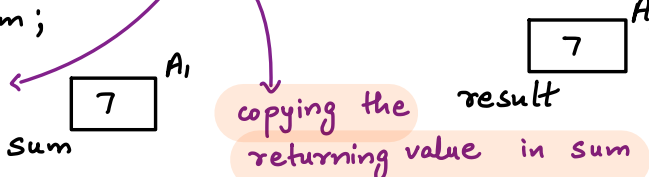
⟶ output - address of n

```
int main(){
    int a = 3;
    int b = 4;
    int sum = add(a, b);
    cout << sum;
    return 0;
}
```

OS

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

returning a value



Function Order

Order 1

```
int add (int a, int b){
    return a+b;
}
int main (){
    int a= 3;
    int b = 5;
    int sum= add (a, b);
    cout << sum;
    return 0;
}
```

function declaration and definition

Order 2

function declaration

```
int add (int a , int b);
int main (){
    int a= 3;
    int b = 5;
    int sum= add (a, b)
    cout << sum;
    return 0;
}
int add (int a, int b){
    return a+b;
}
```

function definition

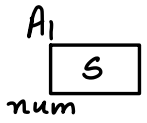
% operator ⟶ heavy operator

↳ so try to use it less

BTS → Behind The Scenes

W2-L4

```
int num=5;
```



Symbol table

int num ⟷ A_1

- datatype
- variable name
- memory location of variable

variable address mapping is stored

```
cout <<num;
```

goto → symbol table → go to → A_1 (address) → print → 5 (value)

```
cout<< sizeof(num);
```

goto → symbol table → int datatype → print → 4

while loop

```
int i=0;
while (i< 5){
    cout << i <<endl;
    i++;
}
```

- `int i=0;` → initialization
- `while (i< 5){` → condition
- `cout << i <<endl; i++;` → body
- updatation

flow

initialization → condition → (false) → exit

condition → (true) → body (containing updatation) → condition

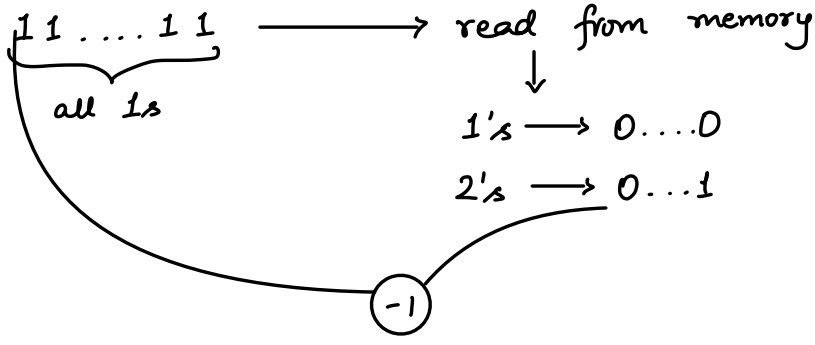
left and right shift operators

```
int a= 2;
a<<1 ;  ------------> no change
cout<< a;  ---------> 2
a= a<<1;  ----------> change ---> left shift by 1
cout <<a;  ---------> 4
```

right shift in -ve no.

↳ link in links.txt in repo

How -1 is stored in memory -



~a = -(a+1) and ~(~a) = a

ex - a = 5; ⟶ 0...0 1 0 1

a= ~a; ⟶ 1...1 0 1 0 ⟶ -6 ⟶ -(5+1)

read ↳ 2's ⟶ 0...0 1 1 0

-6

a= -6;

↳ 1...1 0 1 0

a= ~a; ⟶ 0...0 1 0 1

↳ 5 ⟶ -(-6+1)

W2-R

Number System

- method to represent numeric values using digits

Decimal Number System

- base 10
  - digits → 0 to 9

Binary Number System

- base 2
  - digits → 0, 1
- used in CPU, memory, computer
  - 0 → power off
  - 1 → power on
  - number, images, all files & folder are in binary

Decimal to Binary

- divide no. by 2
- store remainder
- repeat above steps until no. is 0 or 1
- reverse the bits so obtained

Binary to Decimal

- multiply each bit with its place value
  - base ^{i}
    - 2^{i}
- add all products

Time & Space Complexity

W3-R

Time Complexity

- amount of time taken by an algo as a function of length of input
- not actual time
- it defines CPU operations
- use case -
  - to make efficient programs
  - ask by interviewer after every sol. you give

Space Complexity

- amount of space taken by an algo as a function of length of input

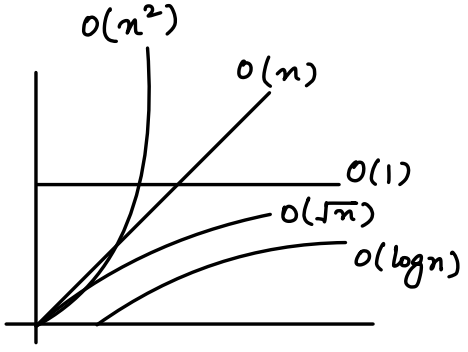
Units to represent Complexity

Big O ⟶ upper bound ⟶ worst case

Theta Θ ⟶ average case

Omega Ω ⟶ lower bound ⟶ best case

Big O Complexities



$O(1)$ ⟶ Constant time

$O(n)$ ⟶ Linear time

$O(\log_2 n)$ ⟶ Logarithmic time

$O(n^2)$ ⟶ Quadratic time

$O(n^3)$ ⟶ Cubic time

nesting ⟶ multiply ⟶ { { } }

in sequence ⟶ add ⟶ { } { }

$$f(n) = 4n^4 + \frac{n^3}{5} + \log n + n\log n \longrightarrow O(n^4)$$

Complexity Order

$$O(1) < O(\log_2 n) < O(\sqrt{n}) < O(n) < O(n\log_2 n) < O(n^2)$$

$$< O(n^3) < O(2^n) < O(n!) < O(n^n)$$

ARRAY

W3-L1

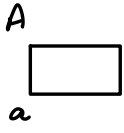
- Data structure to store similar items
  - same datatype
- Continuous memory location
- use case
  - for multiple same kind of data

int a[30000]; ⟶ 30000 variables are ready

continuous memory allocation

- memory wastage
  - if needable memory is present but not in continuous way

int a = 5;



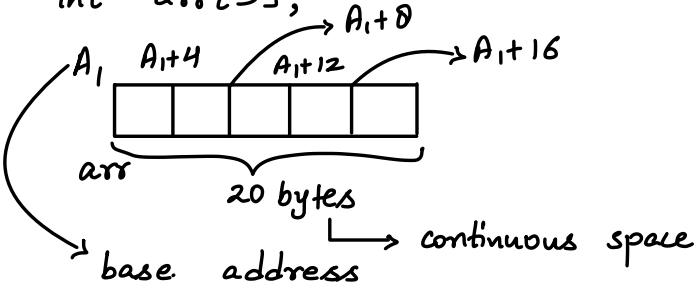
symbol table

int a ⟷ A

int arr ⟷ A_1

Declaration

int arr[5];



`cout << arr ;` ⟶ A_1 ⟶ address of first index of array / address of integer

`cout << &arr ;` ⟶ A_1 ⟶ base address of array arr / address of array

diff. things

Initialization

```
int arr[] = {2,4,6,8,10};
int arr2[5] = {2,4,6,8,10};
int arr3[10] = {2,4,6,8,10};   ⟶ remaining 5 will be 0
//int arr4[4] = {2,4,6,8,10};  ⟶ ERROR
int arr5[10] = {0};            ⟶ initializing all values with 0
```

Making array at runtime

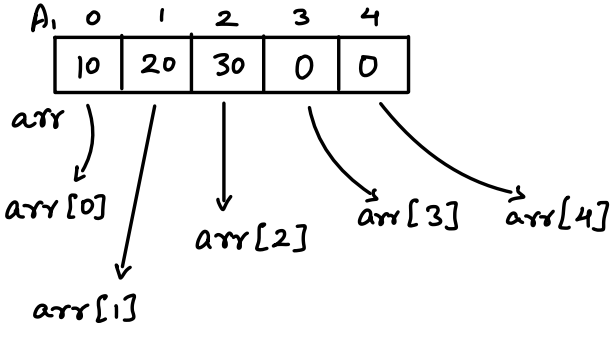
```
int n;
cin >> n;
int arr[n];   ⟶ BAD PRACTICE
```

Index and Access in memory

```
int arr[5] = {10, 20, 30};
```

⟶ 0^{th} based indexing

↳ 0 to n-1



arr[i] ⟶ value at address [A_1 + (i*4)]

BA

index

due to int (datatype size)

thats why 0 based indexing

due to internal working

taking input in array

```
cin >> arr[i];
```

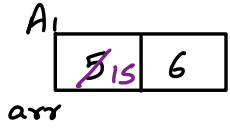
Arrays and Function

```
func ( int arr[], int size) {
}
```

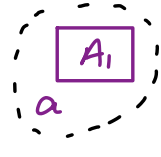
- array is passed by reference because pointer is passed by value
- updation in actual array
- always pass size alongwith arr

arr here is not an array, it is pointer pointing to first index of array (will learn ahead)

```
int main() {
    int arr[] = {5, 6};
    int size = 2;
    func (arr, size);
    return 0;
}
```




```
void func ( int a[], int size){
    a[0] = a[0] +10;
}
```



will learn ahead

sizeof(int); ⟶ 4 ⟶ in bytes

int arr[5];

sizeof(arr); ⟶ 20 ⟶ in bytes

⟶ not 8 (size of pointer)

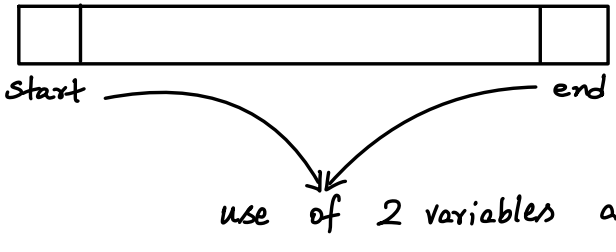
linear search in array

INT_MIN and INT_MAX

→ to find max., start ans with INT_MIN

→ to find min., start ans with INT_MAX

2 pointer approach



use of 2 variables as extreme points

To find size of array

```
int arr[] = {1, 2, 3, 4};
int size = sizeof(arr) / sizeof(int);
```

→ datatype

W3-L2

Vector

- Data structure
- Same as array but dynamic
  - size not fixed
- default size → 0
- if gets full and new items are inserted
  size gets doubled
- pass by value in functions

Initialization

```
vector <int> arr {10,20,30};
```
→ | 10 | 20 | 30 |

```
vector <int> arr (5);
```
→ | 0 | 0 | 0 | 0 | 0 |

```
vector <int> arr (5, -2);
```
(5 → size, -2 → value)
→ | -2 | -2 | -2 | -2 | -2 |

```
int n;
cin >> n;
```
→ let n = 5

```
vector <int> arr(n);
```
→ | 0 | 0 | 0 | 0 | 0 |

```
vector <int> arr (n, 10);
```
→ | 10 | 10 | 10 | 10 | 10 |

Insertion -

```
arr.push_back (5);
```

Remove

```
arr.pop_back ();
```

Size -

```
arr.size ();
```
→ no. of elements it stores

declaration

```
vector <int> arr;
```
- arr.size() → 0
- arr.capacity() → 0

Empty or Not

arr.empty(); ⟶ true if empty

Capacity -

arr.capacity(); ⟶ * by 2 if arr gets fully filled and a new element is inserted

- ⟶ no. of elements it can store
- ⟶ in initialization, capacity = size in all methods of initialization

sizeof(arr); ⟶ compiler dependent

initially

cout << arr ; ⟶ give ERROR

→ Xor ⟶ Cancels out same element

0 ^ ans = ans ⟶ 0^1 = 1, 0^0 = 0

for each loop

```
for (auto val: arr){
    cout << val << ' ';
}
```

2D Arrays

W3-L3

- array of arrays
- use case
  - to work on multiple rows and columns

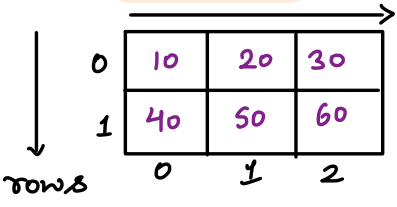
Declaration -

```
int arr[m][n];
```

- m*n elements
- cols → 0 to n-1
- rows → 0 to m-1

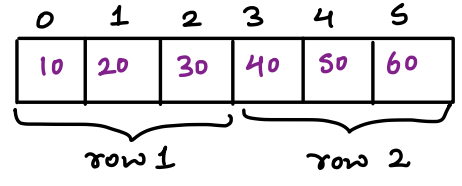
```
int arr[2][3];
```

Visualize -



mapping

in memory -



Access -

```
arr[i][j];
```

- col index → $0 \le j < n$
- row index → $0 \le i < m$

Mapping -

linear_index = c * i + j

- c → no. of cols
- i → row index
- j → col index

$i = linear\_index / c$

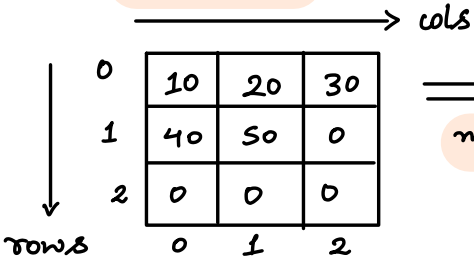
$j = linear\_index \% c$

Initialization -

```
int arr[3][3] = {{10, 20, 30}, {40, 50}};
int arr[3][3] = {10, 20, 30, 40, 50};
```

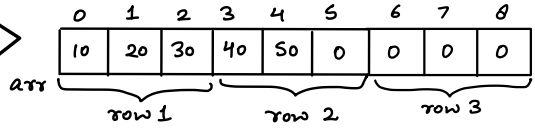
same

Visualize -



mapping

in memory -



2D Arrays and function -

→ pass by reference

```
func ( arr[][500], int rows, int cols)
```

actually arr here is int (*)[500] (array of pointers), not a 2D array

→ cannot leave blank

why → if dont know, put large value

500 tells size of array of pointers

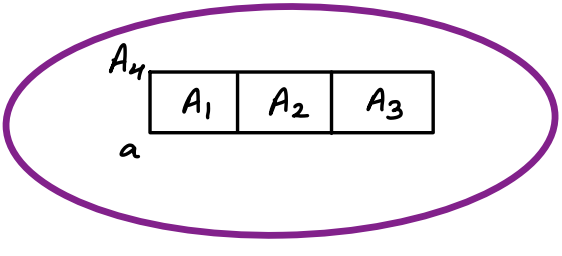
→ this value and no. of cols in array passed in function call should be same

2D Array -

→ not an array of pointers

→ array of arrays

```
func( int a[][3], int rows, int cols) {
    cout << &a;      ---> add. of a ---> A4
    cout << a;       ---> A1
    cout << a[0];    ---> A1
    cout << &a[0];   ---> A1
    cout << &a[0][0]; ---> A1
    cout << a[0][0];  ---> 10
    cout << sizeof(a);  ---> 4
    cout << sizeof(a[0]); ---> 12
}
int main(){
    int arr[2][3] = {10, 20, 30, 40, 50};
    func(arr, 2, 3);
    return 0;
}
```

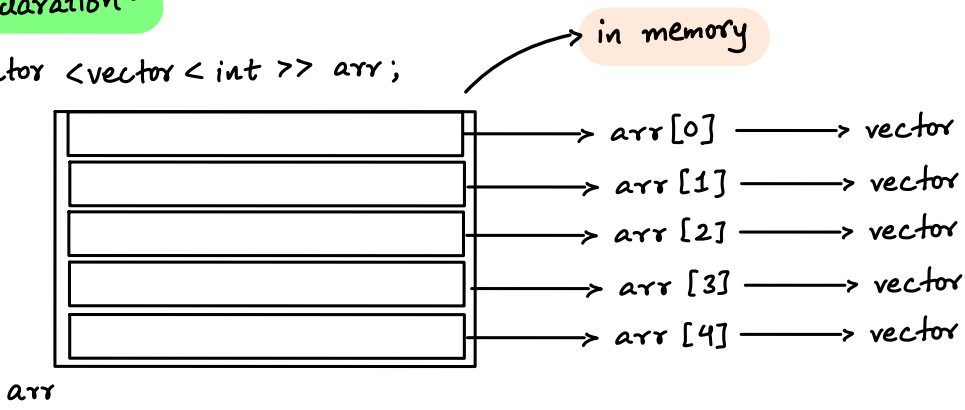


arr (A1)			
A2	40	50	0
A3	0	0	0

2D Vector

Declaration -

```
vector <vector < int >> arr;
```



Declaration -

```
vector < vector <int >> arr;
vector < vector <int >> arr (n);
```

n → no. of rows

Number of rows

```
arr. size ()
```

Number of cols

```
arr[i]. size ()
```

→ in ith row
→ size of ith row

Initialization

```
vector < vector < int > > arr (rows, (vector<int > (cols, value)));
```

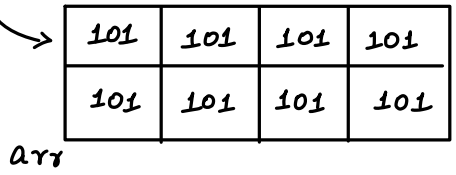
size — initial value in 2D vector

rows → no. of rows in arr
cols → no. of cols in arr
└→ size of 1D vector

initialization of 1D vectors in arr

value → initial value in all elements of all 1D vectors

```
vector < vector < int > > arr ( 2, vector<int>(4, 101));
```



auto keyword -

→ automatically replace with required data type

```
int arr [4]
for ( auto i = 0; i< 4; i++)
    cout << arr [i];
```

: Operator

→ belongs to operator

→ generally used for sequential access

→ map , set , array, vector

ex -

```
vector <int > arr {1 , 2, 3};
for ( int i : arr)
    cout << i;                    ⟶ 1 2 3
for ( auto i : mapping)
    cout <<i;
for ( auto i : sett)
    cout <<i;
```

Searching and Sorting

W4-L1

Searching

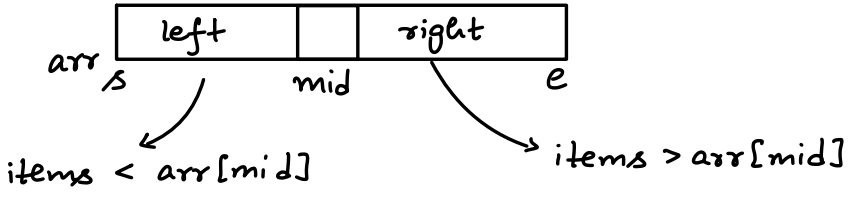
Linear Search

```
int linearSearch ( vector <int> arr, int target ) {
    int n = arr.size();
    for( int i=0; i<n ; i++){
        if ( target == arr[i])
            return i;
    }
    return -1;
}
```

T.C - O(n) S.C. - O(1)

Binary Search -

- condition → sorted order → monotonic function
- binary → 2 → left and right search



```
int binarySearch ( int arr[], int n , int target ){
    int s= 0, e= n-1;
    int mid = s+(e-s)/2;
    while( s<=e) {
        int element = arr[mid];
        if (target == element)
            return mid;
        else if ( target < element)
            e = mid - 1;  ---> search in left subarray
        else
            s = mid +1;  ---> search in right subarray
        mid = s + (e-s)/2;
    }
    return -1;
}
```

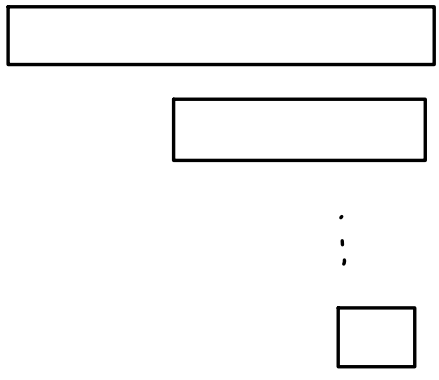
issue in $mid = \frac{s+e}{2}$; ⟶ int overflow if $s+e > INT\_MAX$

⟶ so use $mid = s + \frac{e-s}{2}$;

T.C - $O(\log_2 n)$

S.C. - $O(1)$

T.C. of binary Search



at last $\frac{n}{2^k} = 1$

$k = \log_2 n$

So loop runs $\log_2 n$ times

T.C. = $O(\log_2 n)$

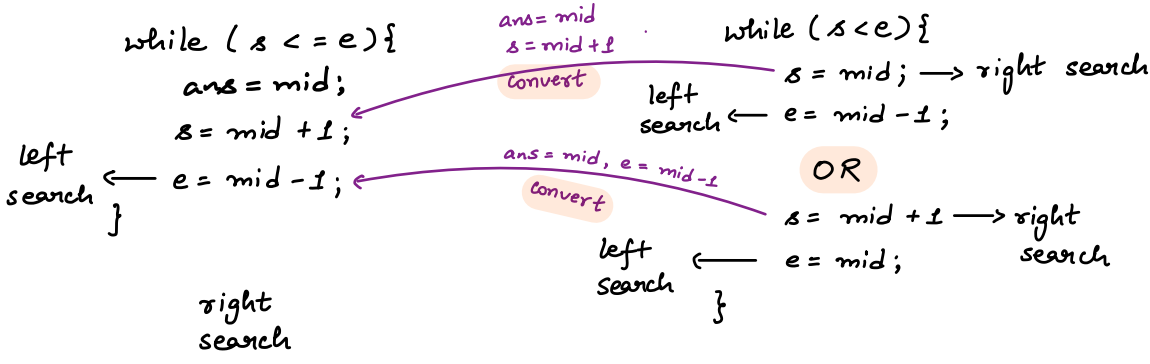
W4-L2

→ Search Space

└→ find range of search space (start & end) in ques of. Binary Search

→ store mid in ans if needed

→ In binary search questions



W4-L3

`cout << (int)(-3.74);`

→ -[int(3.74)]

→ -(3) → -3

`cout << (-22)/7;` → -3

`cout << 22/(-7);` → -3

Types of ques in binary search

→ 1st type → classic questions

- lower bound
- upper bound
- peak in mountain array
- pivot in sorted rotated array
  - can also find array is sorted or not
    - pivot index = n-1
- search in sorted rotated array

→ 2nd type → find in search space (range)

- → predicate function → logic to decide either left or right
- → sqrt of a no.
  divide 2 numbers

Advance Binary Search Problems

- → Book allocation
  Painters Partition
  Aggressive Cows
  Roti / Paratha Spoj
  Eko Spoj

→ 3rd type → observation in index

- → missing element in sorted array
  odd appearing element in array

W4-L5

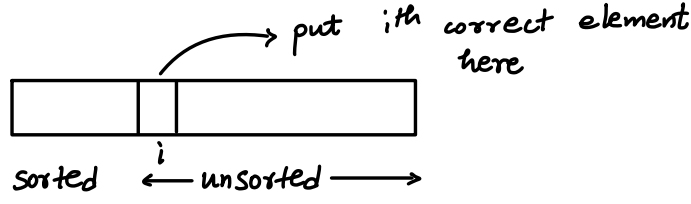
Sorting

- → putting all elements either in increasing order or decreasing order

Selection Sort -

- select minimum element and put it in its right position
- select correct element for i^{th} index
  - i^{th} minimum element

```
void selectionSort ( int arr[], int n) {
    for(int i=0; i<n-1; i++){
        mini = i;
        for ( int j= i+1 ; j<n ; j++) {
            if ( arr[mini] > arr[j])
                mini = j;
        }
        swap (arr[i], arr[mini]);
    }
}
```



T.C. $O(n^2)$ S.C. $O(1)$

Use Case - for small size array
- put i^{th} lowest element to its correct position

Bubble Sort -

→ in i^{th} round put i^{th} largest element to its correct position using adjacent comparisions

```
void bubbleSort ( int arr[], int n){
    for(int i=0 ; i<n-1 ; i++) {
        bool swapping = false;
        for( int j= 0 ; j< n-i-1 ; j++){
            if (arr[j] > arr[j+1]) {
                swap ( arr[j], arr[j+1]);
                swapping = true ;
            }
        }
        if ( swapping = = false)
            break;
    }
}
```

T.C. - $O(n^2)$ → worst and average case (→ reverse sorted)

$O(n)$ → best case → already sorted

S.C. - $O(1)$

Use Case - To put i^{th} largest element to its correct position

Insertion Sort -

↳ take an element and insert it on its correct position by shifting

```
void insertionSort ( int arr [ ], int n ) {

    for ( int i = 1 ; i < n ; i++ ) {
        int curr = arr [ i ];
        int j = i - 1 ;
        for ( ; j >= 0 ; j-- ) {
            if ( arr [ j ] > curr )
                arr [ j+1 ] = arr [ j ] ; // shifting
            else
                break ;
        }
        arr [ j+1 ] = curr ; // inserting
    }
}
```

T.C. - $O(n^2)$ ⟶ worst & average case

$O(n)$ ⟶ best case

S.C. - $O(1)$

Use Case - When array is small or when array is partially sorted

Inbuilt sort function

- sort (arr.begin(), arr.end());
- algo used is Intro sort
  hybrid of quick sort, heap sort, insertion sort
- min. time then any of other sort

T.C. - O(nlogn)

S.C. - Not Defined

Stable and Unstable Algorithm

Stable → order preserve after sorting
→ of same values

Stable Sorting Algo

2 1 2 2 3

↓ after sorting

1 2 2 2 3

if A[i] = A[j] , i < j

then A[i] comes first before A[j] after sorting too

Stable - Bubbe Sort, Insertion Sort, Merge Sort, Count Sort

Other (unstable) sorting algorithms can be made stable by some changes

T.C. and S.C Sorting Algorithm Table

Algo	S.C.	Worst T.C.	Avg. T.C.	Best T.C.
Selection	$O(1)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble	$O(1)$	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion	$O(1)$	$O(n^2)$	$O(n^2)$	$O(n)$
sort func. (intro sort)	not defined	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
merge	$O(n)$	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
quick	$O(n)$	$O(n\log n)$	$O(n\log n)$	$O(n^2)$

CHAR ARRAYS & STRING

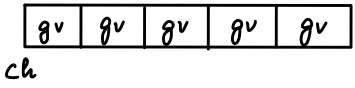
W5-L1

Char Arrays -

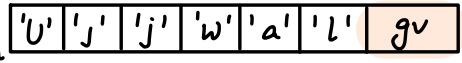
→ Data structure ⟶ used to store data

Not Data type ⟶ tells type of data

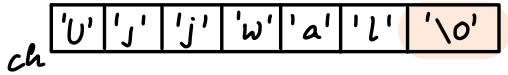
```
char ch[5];
```




```
char ch[] = {'U','j','j','w','a','l'}
```




```
char ch[] = "Ujjwal";
```



Taking input in char array

```
char ch[7];
cin >> ch[i];
```

`cin >> ch;` ⟶ by defaut NULL char will insert at end → '\0'

↓

NULL char refers string termination

→ cin reads until it gets any white space

→ space ' '
tab '\t'
endl '\n'

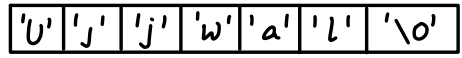
`cout << ch;` ⟶ cout print until it gets delimeter

```
char ch[10];
```

→ allocate as ch[10] = "Ujjwal"

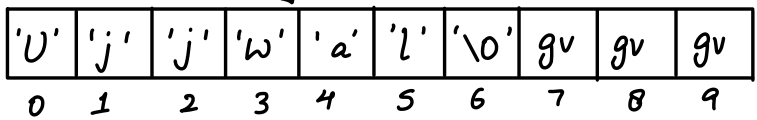
```
cin >> ch;  ⟶  Ujjwal
```

store in buffer memory



buffer

copy in memory of ch




```
cout << ch;  ⟶  Ujjwal  ⟶  stops after 'l'
```

because of '\0' char

Overflow -

```
char ch[4];
cin >> ch;  ⟶  Ujjwal
cout << ch;  ⟶  Compiler Dependent
```

sizeof (char array)

```
char ch[] = "Ujjwal";
cout << sizeof(ch);
```

→ 7

length + 1 → NULL char

```
char ch[] = {'U','j','j','w','a','l'}
cout << sizeof(ch);
```

→ 6 → length

as there is no NULL char at last

get line

cin.getline (char array, max char to write, delimeter);

char where taking input stops

by default '\n' ⟶ enter

ex- cin.getline (ch , 50);

cin.getline (ch , 50, ' ');

Char arrays and function

⟶ pass by reference → actually pass by value but ch is address, so any change acts as pass by reference

```
func ( char ch[]){

}
```

size of char array ⟶ sizeof (ch) / sizeof (char) − 1

sizeof (char) ⟶ 1

Some inbuilt functions of char array

strlen (ch);

strcmp (ch1, ch2);

strcpy (ch1 , ch2);

Strings

- Datatype
  Not DataStructure
- Dynamic char array
- NULL char at last of string

string str; ⟶ empty string created

cin >> str; ⟶ Ujjwal

'U'	'j'	'j'	'w'	'a'	'l'	'\0'

str

cout << str; ⟶ Ujjwal

cout << str.length(); ⟶ 6, not 7

getline -

```
string str;
getline (cin, str);
```

char array

```
char ch[] = "B_abba_r";
cout << ch;          ⟶ B_abba_r
cout << arr.size() ⟶ 9

ch[1] = '\0';
ch[6] = '\0';
cout ch;             ⟶ B
```

stops just as it gets NULL char

string

```
string str = "B_abba_r";
cout << str;          ⟶ B_abba_r
cout << str.length() ⟶ 8
cout << str.size()   ⟶ 8

str[1] = '\0';
str[6] = '\0';
cout << str;          ⟶ Babbar
```

Runs till the length of string

Sort function in strings -

W5-L3

```
string str = "babbar";
sort(str.begin(), str.end());  ---> aabbbr
sort(str.begin(), str.end(), greater<char>());
                                  ---> rbbbaa
```

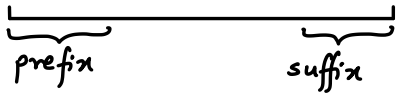
Custom Comparator -

```
bool cmp(char a, char b) {
    return a < b;
}
bool cmp2(char a, char b) {
    return a > b;
}
int main() {
    string str = "babbar";
    sort(str.begin(), str.end(), cmp);
    cout << str;   ---> aabbbr
    string str2 = "babbar";
    sort(str.begin(), str.end(), cmp2);
    cout << str2;  ---> rbbbaa
    return 0;
}
```

a < b : can be any function according to need

a > b : ascending order

Prefix and Suffix



Hash Map → WILL LEARN LATER

- Data structure
- data stored in key-value pair

Initialization

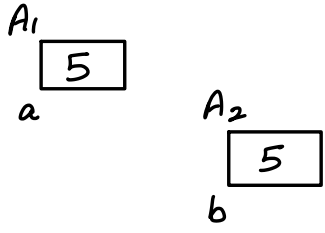
```
map<key datatype, value datatype> map_name;

map<int, char> m; ——> ordered map

m[0] = 'a';

m[1] = 'b';

m[25] = 'z';

cout << m[0]; ——> 'a'

cout << m[25]; ——> 'z'

cout << m[20]; ——> '' ——> NULL char

cout << (int) m[20]; ——> 0
```

SYMBOL TABLE

W6-L1

- Data Structure
- Stores mapping of variable name, datatype & memory location address → Done by OS
- entries in symbol table cant be change

```
int a = 5;
int b = 5;
```



Symbol Table

name	address
a	A_1
b	A_2

→ entry of a in symbol table is made

cout << a; ⟶ 5

& Address Of Operator

cout << &a; ⟶ A_1 ⟶ address of variable a in memory

cout << &b; ⟶ A_2

hexadecimal ↑

Not 100% Sure

$A_2 = A_1 \pm 4$ ⟶ because of consecutive variables in memory

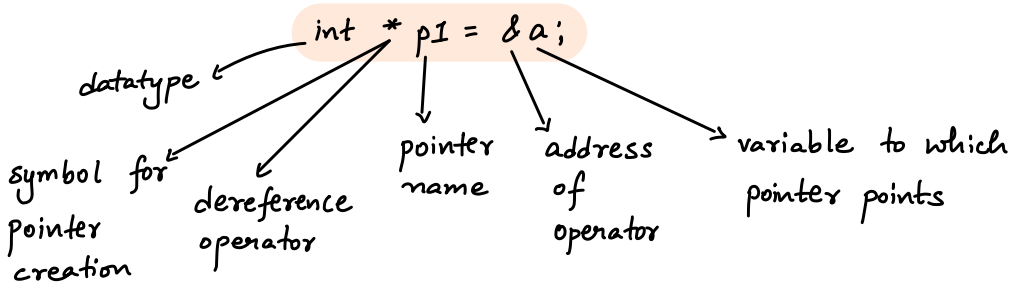
↓ int datatype of a and b

POINTERS

- stores address
- NOT a datatype, just a variable storing address of another variable

```
int a = 5;
int * p1 = &a;
```

p1 is a pointer to integer datatype



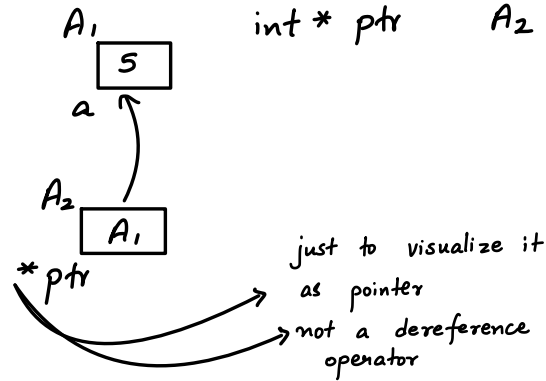

```
string s = "Ujjwal 2327";
string * ptr = &s
```

ptr is a pointer to string datatype

```
int a=5;
int * ptr = &a;
cout << a;      ⟶ 5
cout << *a;     ⟶ ERROR
cout << &a;     ⟶ A1
cout << ptr;    ⟶ A1
cout << *ptr;   ⟶ 5
cout << &ptr;   ⟶ A2
```

symbol table

int	a	A_1
int *	ptr	A_2



*ptr ⟶ value at (address stored in ptr)

↳ dereference operator

&ptr ⟶ address of ptr

size of pointer

sizeof(ptr); ⟶ 8 ⟶ Always

↳ depends on
architecture
compiler implementation
memory organization

↳ 64 bit architecture

↓

it stores address,
datatype does not matters

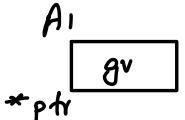
Use case of pointer

- dynamic memory allocation
- memory managment
- pointer arithmetic
  - ↳ go from one location to other
- pass by reference in array
- to create pointer to function
  - ↳ passing a function inside another function as an argument

int * ptr; ⟶ VERY BAD PRACTICE

⟶ Segmentation fault

cout << ptr; ⟶ gv



ptr points to a memory location that may not be of its program

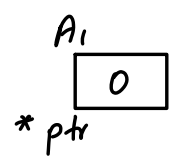
Segmentation fault -

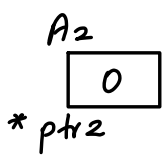
→ using other's memory

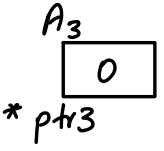
NULL Pointer

```
int * ptr = 0;
int * ptr2 = NULL;
int * ptr3 = nullptr;
```

} same



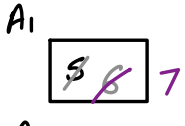


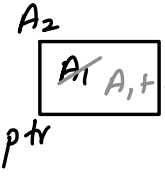



```
cout << ptr ;   ——→ 0
cout << *ptr ;  ——→ ERROR → Segmentation fault
cout << &ptr ;  ——→ A1
```

Arithmetic Operations In Pointers

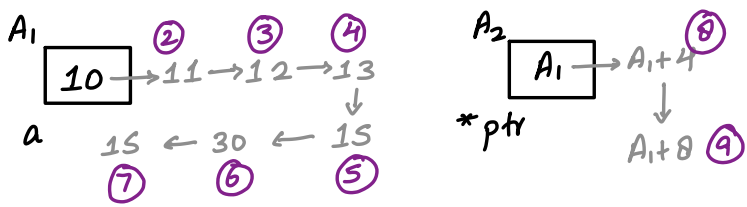
```
int a = 5;
a++;
int * ptr = &a;
*ptr++;
ptr++;
```





A1 to A1 + 3 has already be taken by integer a, so next address will be A1 + 4

```
int  a = 10;
int *ptr = &a;
```



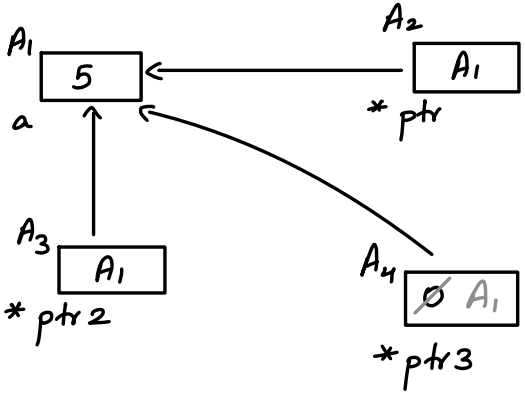
a	10
&a	A_1
ptr	A_1
*ptr	10
&ptr	A_2

1	*ptr * 2	20
2	(*ptr)++	10
3	++(*ptr)	12
4	a = a+1	13
5	*ptr = *ptr + 2	15
6	*ptr = *ptr * 2	30
7	*ptr = $\frac{*ptr}{2}$	15
8	*(ptr++)	15
9	*(++ptr)	gv

```
int  a = 5;
int *ptr = a;   ⟶ ERROR
```

Copying a pointer

```
int  a = 5;
int *ptr = &a;
{ int *ptr2 = ptr;
{ int *ptr3 = 0;
  ptr3 = ptr;
```



ARRAYS & POINTERS

W6-L2

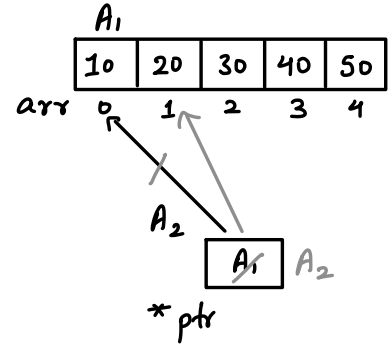
```
int arr[] = {10, 20, 30, 40, 50};
```

`arr` → CONSTANT pointer to the first element of array

- cannot change
- also does not have seperate memory

`arr[i]` ⟶ element at index i

`& arr[i]` ⟶ address of `arr[i]`




```
cout << arr[0];   ⟶ 10
cout << &arr[0];  ⟶ A1
```

```
cout << arr;   ⟶ A1
               ⟶ array name returns add. of 1st index of array

cout << &arr;  ⟶ A1
               ⟶ from Symbol table (add. of array)
```

Same address unlike pointers

```
int *ptr = arr;
cout << *ptr;   ⟶ 10
cout << ptr;    ⟶ A1
cout << &ptr;   ⟶ A2

cout << *arr;       ⟶ 10
cout << *(&arr);    ⟶ 10
cout << *(&arr[0]); ⟶ 10
```

Symbol table

type	name	address
int(*)[5]	arr	A1
int *	ptr	A2

cout << *arr +1; ⟶ 11

cout << *(arr + 1); ⟶ 20

cout << *(arr + 2); ⟶ 30

cout << arr[2]; ⟶ 30

cout << 2[arr]; ⟶ 30

arr[i] ⟷ *(arr + i) ⟷ i[arr]

arr++; ⟶ ERROR ⟶ Entry in symbol table cant be change

ptr++; ⟶ A1+4

→ You can access subpart of an array using pointers

int* ptr = arr;

cout << *(ptr + 2); ⟶ 40

cout << *(ptr +100); ⟶ gv / segmentation fault / out of bound error

Arrays / Array pointers

int arr[] = {10, 20, 30};

① cout << arr; ⟶ A1
cout << &arr; ⟶ A1
} Same

② arr++; ⟶ ERROR

③ sizeof(arr); ⟶ 3*4=12

Pointers / Normal pointers

int * ptr = arr;

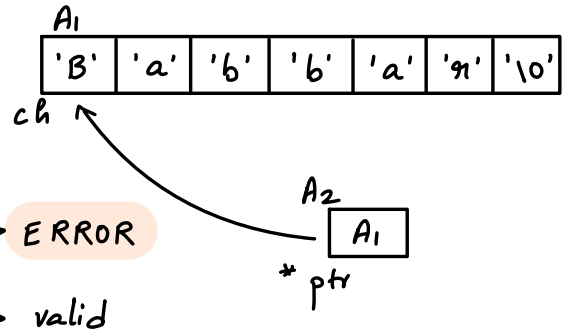
① cout << ptr; ⟶ A1
cout << &ptr ⟶ A4
} diff.

② ptr++; ⟶ VALID

③ sizeof(ptr) ⟶ 8
↓
size of address

```
int * p1 = arr;
int * p2 = &arr;   ──────→ ERROR
int * p3 = &arr[0];
```

CHAR ARRAYS & POINTERS




```
char ch[] = "Babbar";
char * ptr = ch;
char * p = &ch;    ──────→ ERROR
char * p2 = &ch[0]; ──────→ valid
```

```
cout << ch;  ──→ Babbar
             ──→ not an address

cout << ptr; ──→ Babbar
             ──→ not an address
```

Whole string from that location until NULL char

cout implementation is diff. in char pointers and char arrays

```
cout << &ch;    ──────→ A1
cout << ch[0];  ──────→ B
cout << &ch[0]; ──────→ Babbar
cout << *ptr;   ──────→ B
cout << &ptr;   ──────→ A2
```

ch[i] ⟷ *(ch + i) ⟷ i[ch]

→ ch, &ch[0], and ptr values are addresses but due to diff. cout implementation in char pointers and char arrays,
cout << ch , cout << &ch[0] and cout << ptr will give Babbar

```
char ch[] = "Sherbano";
char *ptr = ch;
```

cout << ch ; ⟶ Sherbano

cout << *ch ; ⟶ S

cout << &ch ; ⟶ A_1

cout << *(ch + 3); ⟶ n

cout << ptr ; ⟶ Sherbano

cout << &ptr ; ⟶ A_2

cout << *(ptr + 3); ⟶ n

cout << ptr + 2; ⟶ erbano

cout << *ptr ; ⟶ S

cout << ptr + 8; ⟶ " ⟶ NULL char

cout << ptr + 9; ⟶ gv

```
cout << ch[0];   ——————→ S

cout << &ch[0];  ——————→ Sherbano

cout << &(*ch);  ——————→ Sherbano
```

CHAR AND POINTER

```
char ch = 'k';
char * ptr = &ch;

cout << ch;   ——————→ k

cout << &ch;  ——————→ k.....
                        → gv
                        → print until it gets '\0'

cout << ptr;  ——————→ k.....
                        → gv
                        → print until it gets '\0'

cout << &ptr; ——————→ A2

cout << *ptr; ——————→ k
```

Behind The Scenes

```
char ch[10] = "Babbar";
```

→ 2 step process

① temp memory

'B'	'a'	'b'	'b'	'a'	'r'	'\0'

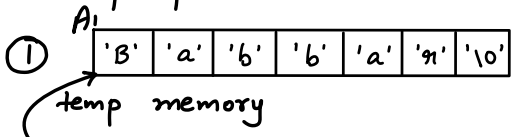
copy ↓

② ch

'B'	'a'	'b'	'b'	'a'	'r'	'\0'	gv	gv	gv

```
char * ch = "Babbar"
```

→ 2 step process



② A1 → pointer points to temporary memory

* ch

↓

BAD PRACTICE

POINTERS WITH FUNCTIONS

→ pass by value

a copy of pointer is made

→ but simulates pass by reference

↳ as in case of arrays & functions

pointer is passed

```
int func ( int arr[]){
    cout << arr;      ——→ A1
    cout << *arr;     ——→ 10
    cout << &arr;     ——→ A2
    cout << sizeof(arr); ——→ 8
}
```

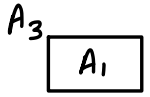
A2 | A1

* arr

copy of pointer is created

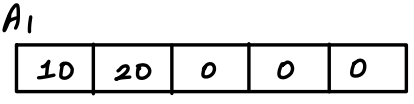
```
int func2 ( int * arr ){
    cout << arr ;  ------------------> A1
    cout << *arr ;  ----------------> 10
    cout << &arr ;  ----------------> A3
    cout << sizeof(arr) ;  ---------> 8
}
```

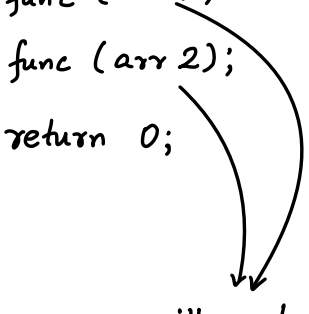
pointer is passed



copy of pointer is created

```
int main(){
    int arr[5] = {10, 20};
    cout << arr ;  ------------------> A1
    cout << *arr ;  ----------------> 10
    cout << &arr ;  ----------------> A1
    cout << sizeof(arr) ;  ---------> 5*4 = 20
    func (arr);
    func (arr2);
    return 0;
}
```

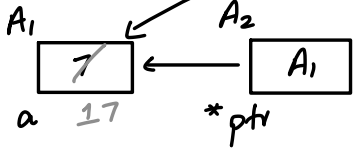


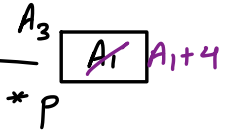


- → whole array will not pass
- → only array pointer / base address will pass
- → pass by reference for array
- → copy of pointer pointing to base address will be created

```
int main(){
    int a = 7;
    int * ptr = &a;
    update(ptr);
    return 0;
}
```

```
void update(int * p){
    (1) *p = *p + 10;
    (2) p++;
}
```





*(A1) = *(A1) + 10

*(A1) = 7 + 10 = 17

a = 17

Function to Pointers in HW

Basic Mathematics for DSA

W6-L3 R

Sieve of Eratosthenes Theorem

→ to find no. of prime numbers between 1 & n

Steps- → seive

- make an array of size n and mark them all as primes from 2 to n
- Start from 2 till end, mark all no. > 2 comes in the table of 2 as non prime
- Do above step for numbers 2 to <n if they are marked prime
- Count all remaing marked prime numbers

```
int countPrimes(int n) {
    if (n <= 1)
        return 0;

    vector<int> isPrime(n, true);
    isPrime[0] = isPrime[1] = 0;   // making both 0

    int ans = 0;
    for(int i=2; i<n; i++){
        if (isPrime[i]){
            ans++;
            for(int j= 2*i; j<n; j+=i)
                isPrime[j] = false;
        }
    }
    return ans;
}
```

`i*i < n` → to optimize T.C. as 2i to (i-1)*i are already marked when i= 2 to (i-1)

TC - $O(n * \log(\log n))$ SC - $O(n)$

$$n\left[\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \frac{n}{11} + \cdots\right]$$

$$n^2\left[\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{11} + \cdots\right]$$

$$n \log(\log n)$$

Same with **segmented sieve** → having high & low

GCD / HCF and LCM

- Greatest Common Divisor
- $gcd(a, b) = gcd(a-b, b)$ $\quad a > b$

  $gcd(a, b) = gcd(b-a, a)$ $\quad b > a$

  $gcd(a, b) = gcd(a \% b, b)$ $\quad a > b$

  $gcd(a, b) = gcd(b \% a, a)$ $\quad b > a$

  } takes more time bcz of % op.

  Use first method as % is heavy operator and computer takes more time

- Euclid Algorithm

  Apply above formula till one of the parameter becomes 0
  And other one will be GCD

```
int gcd( int a, int b) {
    if (a==0)
        return b;
    else if (b==0)
        return a;
    while (a>0 && b>0){
        if (a>b)
            a=a-b;
        else
            b=b-a;
    }
```

```
    return (a == 0) ? b : a;
}
```

TC - O(min(a, b))

LCM

$$lcm * gcd = a * b$$

$$lcm = \frac{a * b}{gcd}$$

Modulo Arithmetic

- $a \% n \longrightarrow [0, n)$
- $(a + b) \% n = (a \% n) + (b \% n)$

$$(a - b) \% n = (a \% n) - (b \% n)$$

$$(a * b) \% n = (a \% n) * (b \% n)$$

$$(\ldots((a \% n) \% n) \ldots \% n) = a \% n$$

Fast Exponentiation

- $a^b = a^{b/2} * a^{b/2}$, b is even
- $a^b = [a^{b/2} * a^{b/2}] * a$, b is odd

```
int fastExponentiation ( int a, int b){
    int ans = 1;
    while (b){
        if (b & 1)
            ans = ans * a;
        a = a* a ;
        b >>= 1 ;   // b = b >>1  or  b = b/2
    }
    return ans
}
```

dry run code on 2^5, if confusion

T.C. - O(log n)

Learn wild, void and dangling pointers from dashboard after learning dynamic allocation

MULTI LEVEL POINTER

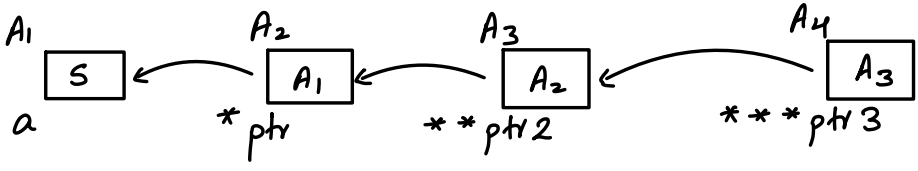
W6-L4

```
int a = 5;
int *ptr = &a;
int **ptr2 = &ptr ⟶ double pointer
```

↳ ptr2 is a pointer to int * data

```
int ***ptr3 = &ptr2;
```

↳ ptr3 is a pointer to int ** data



a ⟶ 5

&a ⟶ A1

&&a ⟶ ERROR

ptr ⟶ A1

&ptr ⟶ A2

*ptr ⟶ 5

ptr2 ⟶ A2

&ptr2 ⟶ A3

*ptr2 ⟶ A1

**ptr2 ⟶ 5

apply dereference operator 2 times

ptr3 ⟶ A3

&ptr3 ⟶ A4

*ptr3 ⟶ A2

**ptr3 ⟶ A1

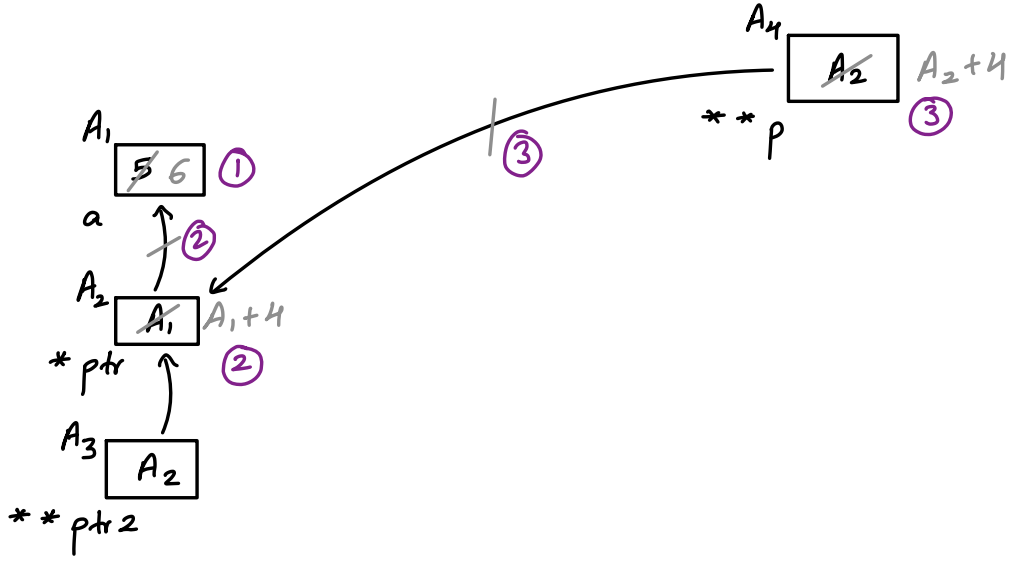
***ptr3 ⟶ 5

```
int main(){
    int a = 5;
    int *ptr = &a;
    int **ptr2 = &ptr;
    func(ptr2);
    return 0;
}
```

both are same

```
func(&ptr);
```

```
void func(int **p){
① (**p)++;
② (*p)++;
③ p++;
}
```



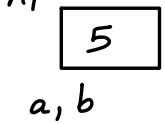

```
int a = 5;
int *p = &a;
int **q = p;
```

ERROR

REFERENCE VARIABLE

- alternate of pointers
  - a, *ptr, **ptr → a
    - diff. names of same memory location
  - very confusing
- diff. names of same variable
  - same memory location
- only new entry in symbol table,
  no other memory will allocate
- can access a variable by diff. names

```
int  a = 5;
int  &b = a;
```



Symbol Table

a	A_1
b	A_1

Use Case

- reference variable cant be set to NULL
  pointers can be set to NULL
  So more safety in reference variable
  Always points to valid object / variable
- pointers are difficult to understand,
  more readability in reference variable
- generally used to implement PASS BY REFERENCE concept

PASS BY REFERENCE

- reference variable passes in function
- does not create copy

```
int main () {
    int a = 5;
    update (a);
    update2 (&a);
    int *ptr = &a;
    //update3 (&a);
    update3 (ptr);
    return 0;
}
```

ERROR constant add. in pass by reference

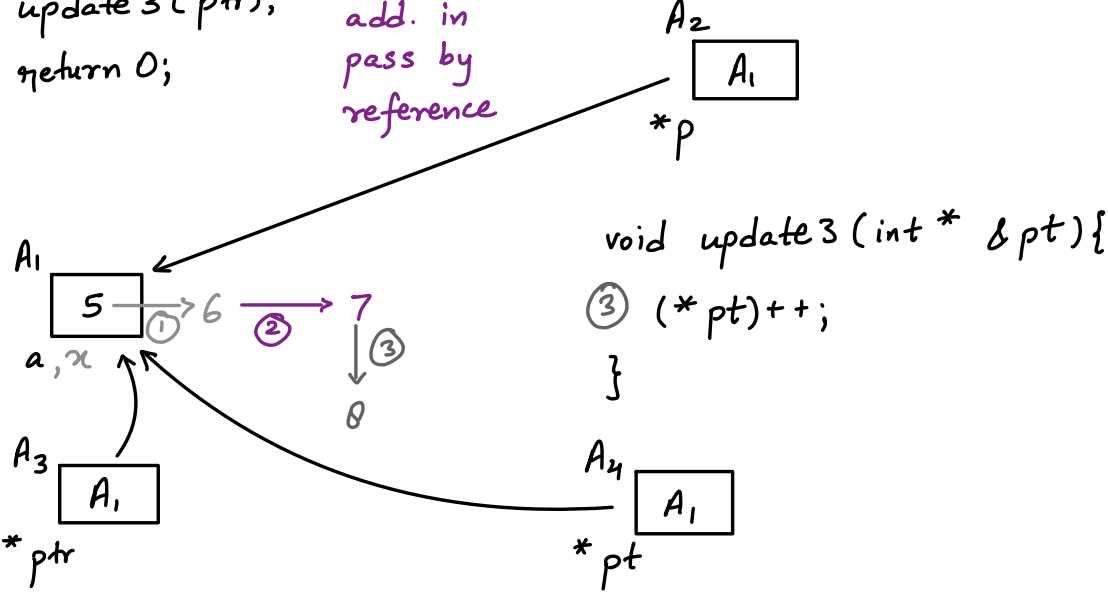
```
void update( int &x){
① x++;
}
```

PASS BY REFERENCE

```
void update2( int * p) {
② (*p)++;
}
```

PASS BY VALUE

```
void update3 (int * &pt){
③ (*pt)++;
}
```



```
vector<int> func(vector<int> &arr){
    for(int i=0; i<arr.size(); i++)
        arr[i]++;
    return arr;
}

int solve(int &arr){
    cout<<arr;
}

int main(){
    vector<int> arr{1,2,3,4};
    vector<int> ans = func(arr);
    for(auto i: ans)
        cout<<i;
    solve(arr);
    int arr2[] = {1,2,3};
    solve(arr2);
    return 0;
}
```

this is reference variable of type int

int solve(int &arr[])

ERROR

array of reference is not allowed

2 3 4 5

ERROR

vector != int

ERROR

int[3] != int

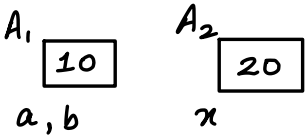
RETURN BY REFERENCE

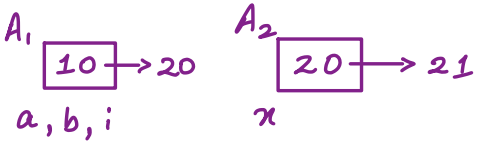
- return a variable, not a value
- To implement atleast 1 of 2 conditions must be true.
  1. passing a reference variable
  2. passing a global variable

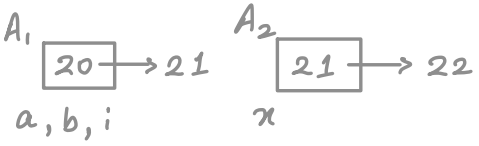
passing a reference variable

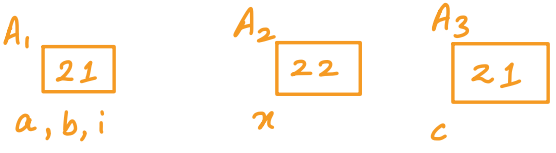
```
int main (){
    int a = 10;
    int &b = a;
    int x = 20;
    ref(a) = x;
    x++;
    ref (b) = x;
    x++;
    int c = ref(a);
    int &d = ref(a);
    return 0;
}
```

```
int & ref( int &i) {
    return i;
}
```

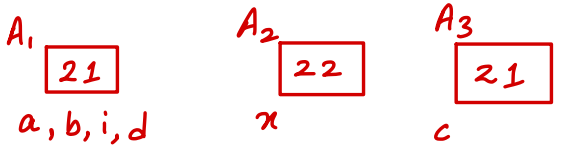




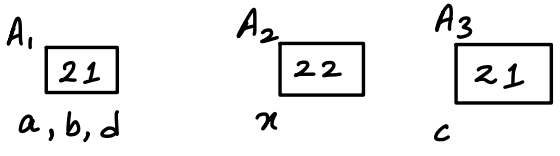





```
int c = ref(a) = i;
int c = 21;
```




```
int &d = ref(a) = i;
int &d = i;
```



passing a global variable

```
int x = 5;

int main () {
    ref () = 10;
    ref 2 () = 20;
    return 0;
}
```

```
int & ref () {
    return x;
}
```

x = 10;

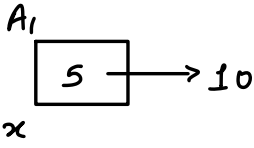
```
int ref 2 () {
    return x;
}
```

ref 2 () = 20; ⟶ ERROR

returning a value, not variable

10 = 20;

⟶ that's why ERROR



pass by value but return by reference

```
int main () {
    int a = 10;
    int & b = a;
    int x = 20;
    ref (a) = x;
    x ++;
    ref (b) = x;
    x ++;
```

```
int & ref ( int temp) {
    return temp ;
}
```

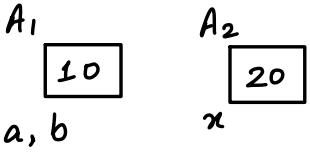
BAD PRACTICE

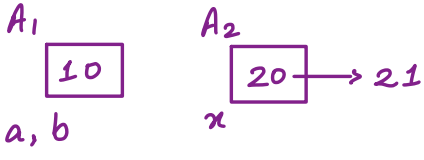
ERROR

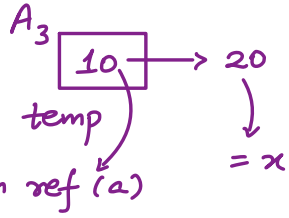
returning a variable stored in temporary memory

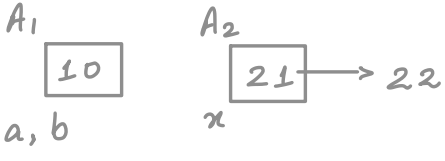
```
    int c = ref(a);
    int &d = ref(a);
    return 0;
}
```

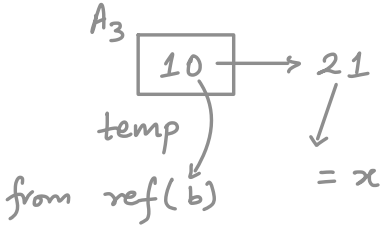
lets assume it will run, then this would happen

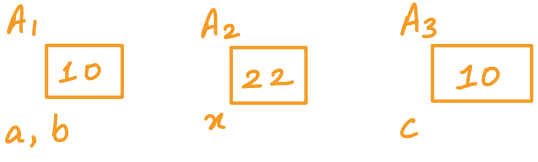


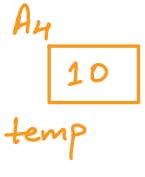




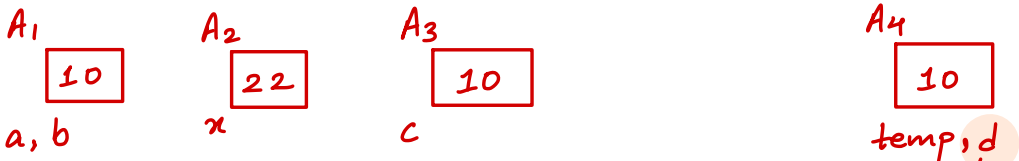









```
int c = ref(a) = temp;
int c = temp;
int c = 10
```



d (variable of main function refers to a temporary memory

→ BAD PRACTICE

Have similarity

DON'T DO THIS AT HOME

IMP QUESTION

```
int main () {
    int * ptr = solve ();
    return 0;
}
```

```
int * solve () {
    int a = 5;
    int * p = &a;
    return p;
}
```

pointing to a temp memory, 'a' variable will finished outside solve function so value of 'a' can be change with time → BAD PRACTICE

Some Important Questions

W6-A

→
```
int *ptr = 0;
int a = 10;
*ptr = a;
cout << *ptr ;
```
→ runtime ERROR

→
```
char * ch = 'a';
char * ptr = &a;
ch++;
cout << *ptr ;
```
→ 'b'

→
```
int a[] = {1, 2, 3, 4, 5}
int * p = a++;
cout << * p ;
```
→ ERROR

- pointer of static variables are constant pointers
- pointer of dynamic memory allocation variables can be change

name of array refers to base address

→
```
char ch = "hello";
cout << ch ;
ch++;
cout << ch;
```
→ hello

→ ERROR

→ `double arr[] = {2.5, 7.9, 50.25};`

`double * ptr = arr;`

`ptr = ptr + 0.25;` ⟶ ERROR

↳ pointer arithmetic op. can only be done using integers

`cout << ptr;`

→ `int arr[] = {1, 2, 3, 4, 5};`

`int * ptr1 = arr;`

`int * ptr2 = arr + 3;`

`cout << ptr2 - ptr1;` ⟶ 3, NOT 12

→ `int a = 5;`

`int * ptr = &a;`

`int ** ptr2 = &ptr;`

`ptr2 = &a;` ⟶ ERROR ⟶ int ** cant convert to int *

`cout << *ptr2;`

→ `int a = 5;`

`int * p = &a;`

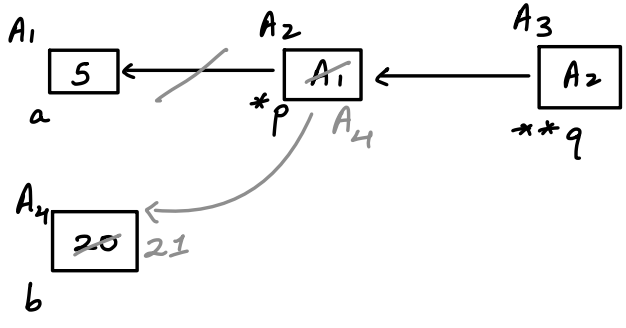
`int ** q = &p;`

`int b = 20;`

`* q = &b;`

`(*p)++;`

`cout << a << ' ' << b;` ⟶ 5 21



```
int a = 5;
int * ptr = &a;
int ** ptr2 = &ptr;
int ** ptr3 = &(&a);
cout << ** ptr3;
```

→ ERROR

address of (address in memory location)

(doesn't make any sense)

POITER IMP DOUBT

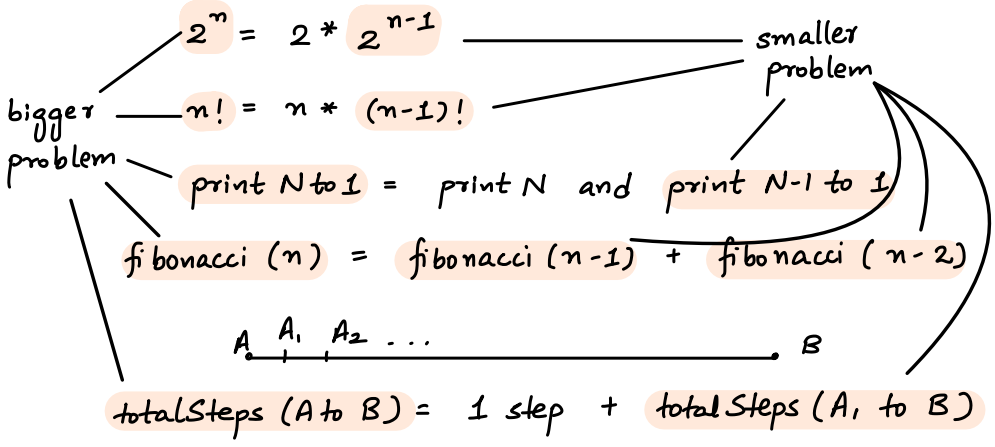
(IN DASH BOARD)

W6-L5

RECURSION

W7-L1

- When a function calls itself
- When sol. of bigger problem depends on similar smaller problem(s)



Components of Recursion

- Base Case / Stopping cond. → return statement
- Recurrence Relation (recursive call)
- Processing → optional

Head Recursion

- recurrence relation before processing

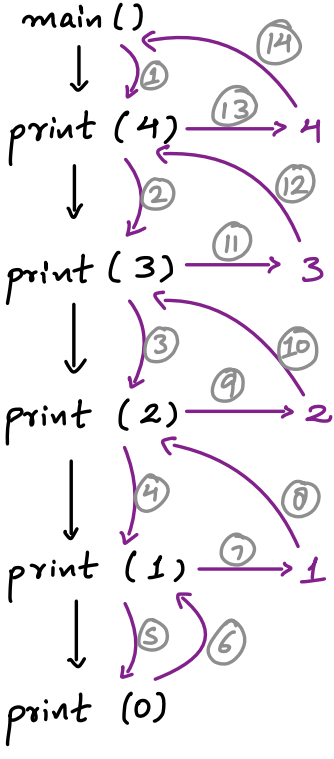
Tail Recursion

- recurrence relation after processing

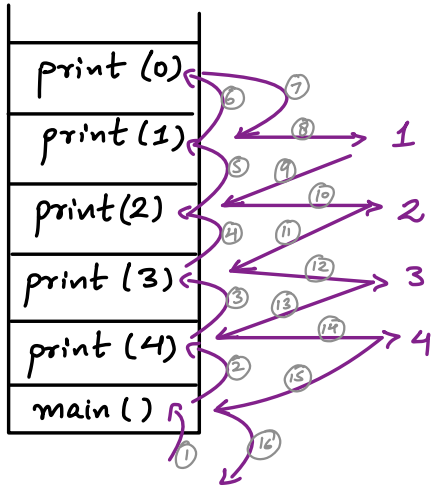
```
int main () {
    int n = 4;
    print (n);
    return 0;
}
```

```
void print ( int n){
    // base condition
    if ( n==0)
        return;
    // recurrence relation
    print (n-1);
    // processing
    cout << n;
}
```

Recursion Tree



Call Stack

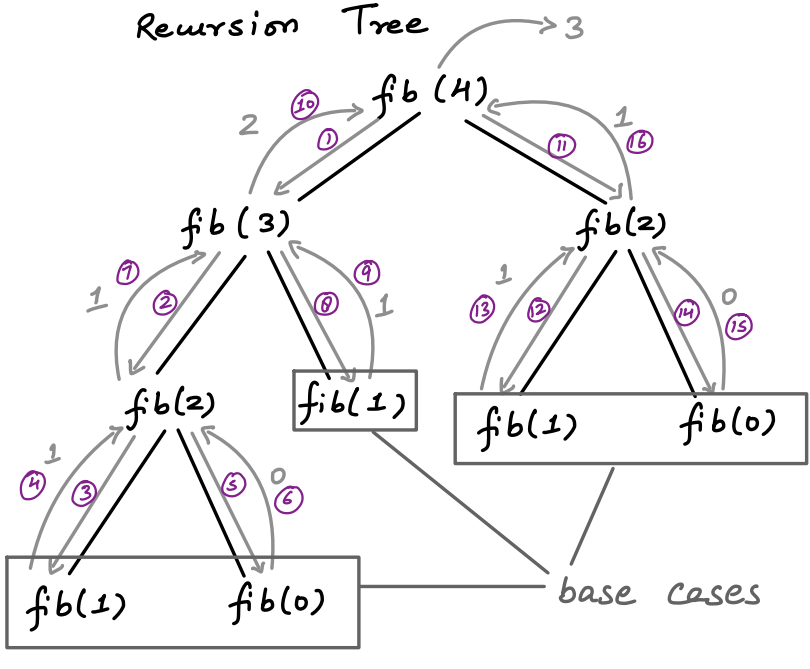


Output - 1 2 3 4

```
int main(){
    int n = 4,
    int ans = fib(n);
    cout << ans;
    return 0;
}
```

```
int fib( int n){
    if ( n==0 || n==1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

n	0	1	2	3	4
fib(n)	0	1	1	2	3



Magical line for Recursion

Ek case solve krdo, baaki recursion sambhal lega

→ Just believe on it, dont doubt on recursion

→ func (vector <int > arr , int i, int & ans)

pass by reference

W7-L2

if we want to retain value of ans after function call

→ Try to pass everything by reference IF POSSIBE

↳ SC ↓es

TC ↓es (no copying of variables again & again)

→ to reverse answer, you can use recursion

Integer literal with a leading 0

n= 0100 ; ⟶ interpretted as an octal number

cout << n; ⟶ 64 ⟶ convert 0100 octal to decimal

cin >> n ; ⟶ (0 1 00)

cout << n; ⟶ 100 ⟶ no conversion as cin reads 0100 as 100

Entry in function call stack

W7-L3

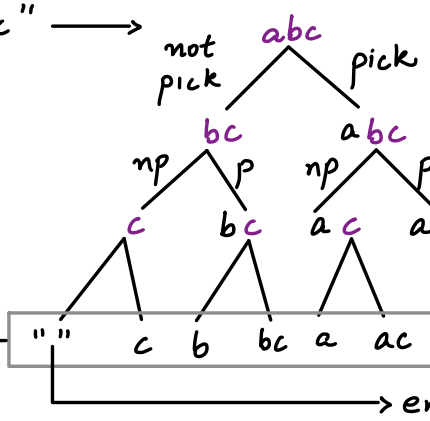
↳ Computer has track of variables of that function about their memory location and all in symbol table

→ Expression can't be pass by reference

Subsequences

→ Made by either including or excluding every element, but order remains same

→ ex- "abc" →



subsequences of string "abc"

→ 2^n, n → length of string

Arithmetic Operations on Strings

```
s1 = "Ujjwal";
s2 = "2327";
cout << "Ujjwal" + "2327"; ——→ ERROR
s1 = s1 + s2;
cout << s1; ——→ "Ujjwal2327" → Concatinate
```

-, *, /, % ——→ NOT VALID in strings

Coin change problem

W7-L4

↳ find min. coins equivalant to given amount

```
int coinChange( vector<int> arr , int amount) {
    if (amount == 0)
        return 0;
    if (amount < 0)
        return INT_MAX;     // means INVALID or
                               cant be changed
    int mini = INT_MAX;

    for( int i=0; i< arr.size(); i++){
       mini = min ( mini, coinChange (arr, target - arr[i]);
    }
    // now min. coins req. has been found
    if ( mini == INT_MAX)
        return INT_MAX; // means INVALID or
                           cant be changed
    else
        return mini +1 ;
}
```

Type 1 - inclusion exclusion pattern ⟶ for using either this or that subsequence questions

Type 2 - for loop in recursion

→ If the recursion ques seems tough, use void function and pass ans by reference in function

TC & SC of Recursive Solutions

W7-R

T.C. → Can be found by recursive relation or recursive tree

S.C. → found using max depth of function call stack
↳ max space used in any particular time instant in function call stack

Linear Search

$f(n) = k + f(n-1)$ Base case → for $n = 0$

Recursive Relation Method -

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

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

.
.

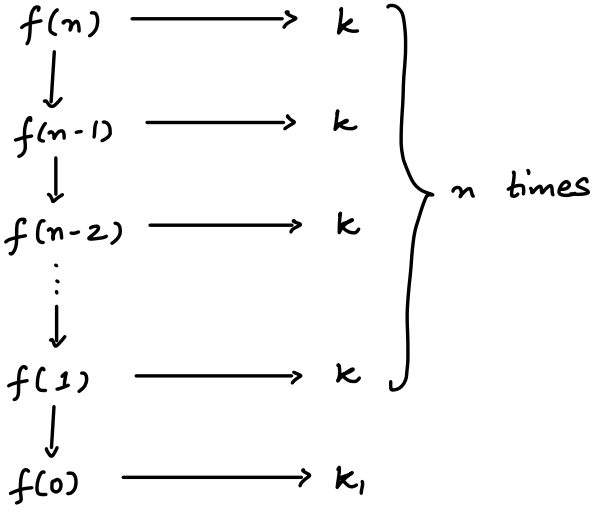
$T(0) = k_1$

$T(n) = nk + k_1$

$T(n) = O(n)$

T.C - $O(n)$

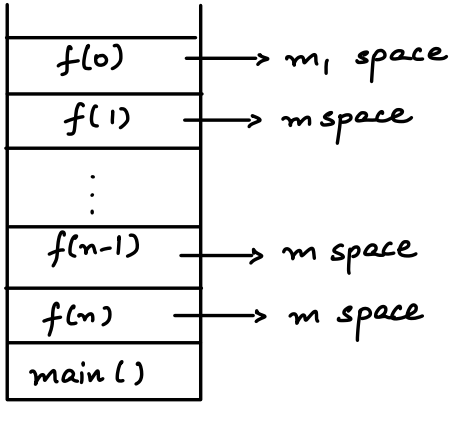
Recursive Tree Method - Best method



$T(n) = nk + k_1$

$T(n) = O(n)$

Space Complexity -



m and m_1 are constants

$SC - mn + m_1$

$SC - O(n)$

→ max depth

Binary Search

$f(n) = k + f(n/2)$; base case $\longrightarrow$ for $n = 0$

Recursive Relation Method

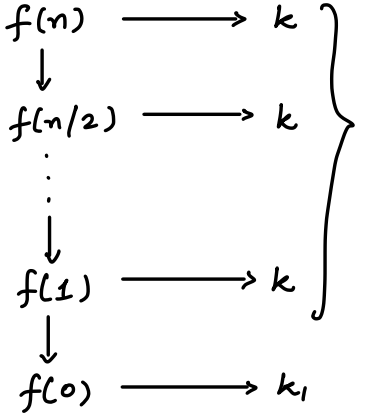
$$T(n) = k + T(n/2)$$
$$T(n/2) = k + T(n/4)$$
$$\vdots$$
$$T(1) = k + T(0)$$
$$T(0) = k_1$$

a times

$a = \log_2 n$

$$T(n) = ak + k_1$$
$$T(n) = k \log_2 n + k_1$$
$$T(n) = O(\log_2 n)$$

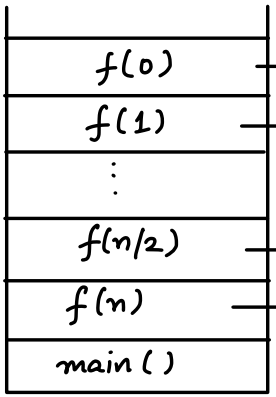
Recursive Tree Method



$a = \log_2 n$

$$TC - k \log_2 n + k_1$$
$$TC - O(\log_2 n)$$

Space Complexity -



m_1 space

m space

m space

m space

a times

m and m_1 are constants

$a = \log_2 n$

$SC - m \log_2 n + m_1$

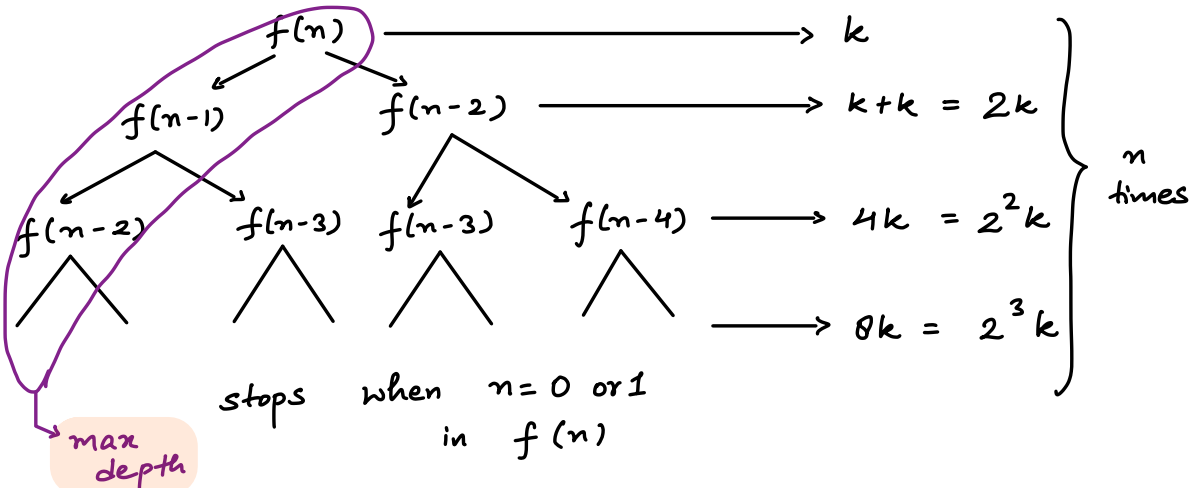
$SC - O(\log_2 n)$ → max depth

Fibonacci Series

$f(n) = k + f(n-1) + f(n-2)$

base case → for $n = 0$ or 1

Recursive Tree Method



$T.C \leq [k + 2k + 2^2k + \dots + 2^{n-1}k]$

$T.C \leq k * [1 + 2 + 2^2 + \dots + 2^{n-1}]$ → assume for simplicity and for large numbers

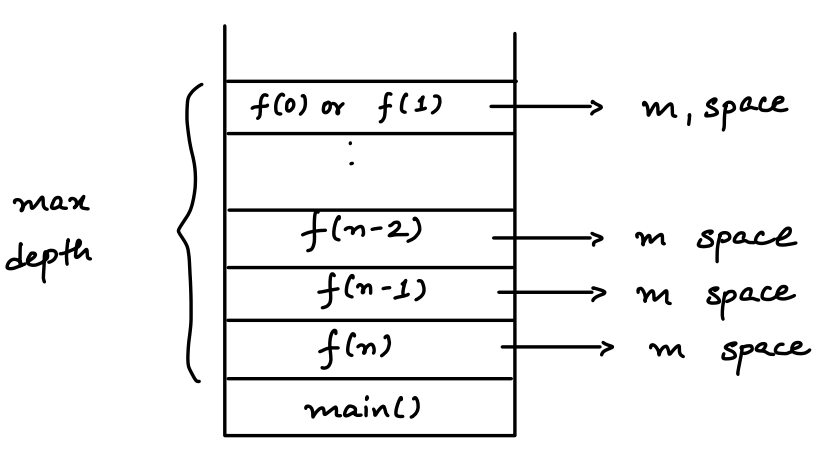
$T.C \leq k * (2^n - 1)$

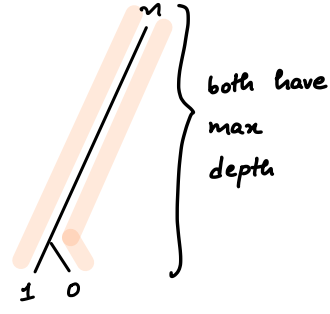
T.C - O(2^n)

→ exponential TC

→ very bad

Space Complexity



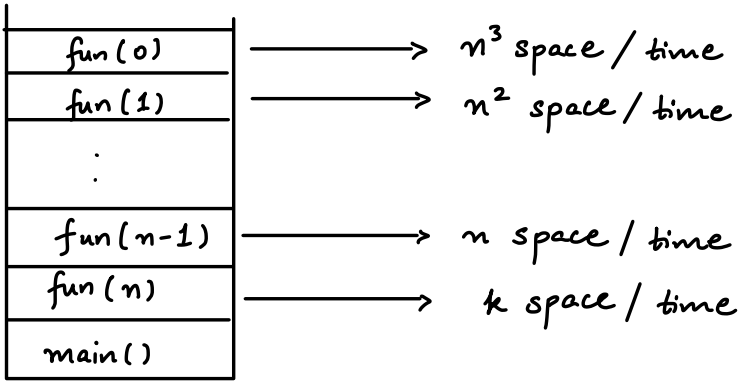


$SC - m(n-1) + m_1$

SC - O(n)

→ max depth

if this is such a case



S.C. - $n * n^3$

↳ max SC in any call

T.C. - $n * n^3$

↳ max T.C. in any call

Divide & Conquer

→ using recursion

W8-L1

Merge Sort

→ Divide the array in 2 halves

→ Sort them

→ Merge those 2 sorted subarrays

Standand Merge Sort

```
void merge( vector <int> & arr, int s, int e) {
    int mid = e - (s + e)/2;
    int len1 = s - mid + 1;
    int len2 = e - mid;
    vector <int> left (len1);
    vector <int> right (len2);
    int i=0, j=0, k=s;
    while (i< len1)
        left [i++] = arr[k++];
    while (j< len2)
        right [j++] = arr[k++];

    i=0, j=0, k=s;
```

```
    while( i< len1  && j< len2){
        if( left[i] <= right[j])
            arr[k++] = left[i++];
        else
            arr[k++] = right[j++];
    }
    while( i< len1)
        arr[k++] = left[i++];
    while (j< len2)
        arr[k++] = right[j++];
}
void mergeSort( vector<int> &arr, int s, int e){
    if( s>=e)
        return;
    int mid = s + (e-s)/2;
    mergeSort (arr, s, mid);
    mergeSort ( arr, mid+1, e);
    merge (arr, s, e);
}
```

due to copy in every level

T.C. - O(nlogn) → in best case too → number of levels

S.C. - O(n) → $(n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots \text{ till } 1)$

IMPLACE MERGE SORT - GFG

Quick Sort

W8-L2

→ Partition

Choose one element named pivot and put it on its correct position

Put elements less than pivot in left subarray and otherwise in right subarray

→ Sort left & right subarrays recursively

```
int partition( vector<int> & arr, int s, int e){
    int pivotIndex = s;
    int pivot = arr[s];

    int cnt = 0;
    for( int i=s+1; i<=e ; i++){
        if( arr[i] > pivot)
            cnt ++;
    }
    int correctPosition = s + cnt;
    swap( arr[pivotIndex], arr[correctPosition]);
    pivotIndex = correctPosition
```

→ This is one way

Other ways - choose last
- choose random
↓
randomised quick sort

```
    int i=s, j=e;
    while( i< correctPosition && j> correctPosition){
        while( arr[i] < pivot)
            i++;
        while( arr[j] >= pivot)
            j--;
        if( i< correctIndex && j> correctIndex)
            swap (arr[i++], arr[j--]);
    }
    return correctPosition;
}
void quickSort( vector<int> &arr, int s, int e){
    if( s>=e)
        return;
    int p = partition( arr, s, e);
    quickSort( arr, s, p-1);
    quickSort (arr, p+1, e);
}
```

T.C. - $O(n \log n)$ ⟶ $\log n$ number of function calls

$O(n^2)$ ⟶ for both ascending & descending sort

S.C. - $O(n)$ ⟶ n number of function calls

Backtracking

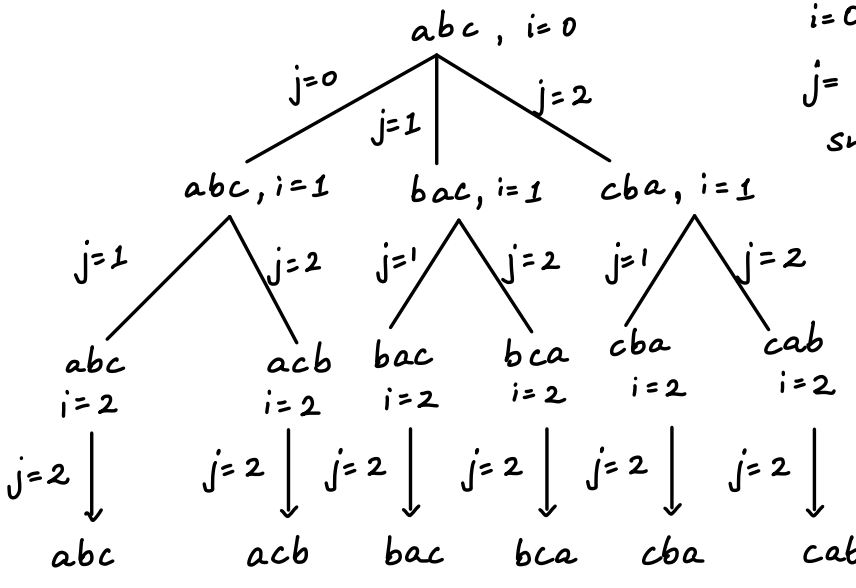
- specific form of recursion
- explore all possible solutions and
- dont check the solution you have already discard
- brute force
- used for reference variables passed in functions

Permutation in a string

s = "abc"

permutations - abc, acb, bac, bca, cab, cba

- total - n!



i=0 to < str.length()

j= i to < str.length()

swap(str[i], str[j])

T.C:- n!

S.C. - s.length()

W8-L7

Time Complexity

- max time required by an algorithm
  - as a function of input
  - for worst case
- number of calls * T.C. - time in 1 call
- T.C. - total time in 1 level * number of levels

Space Complexity -

- max space taken by an algorithm at any particular instant
- S.C. - space in 1 call * max depth

→ Remember min space and time can be O(1) not O(0)

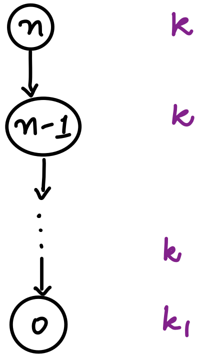
Type 1 Linear Tree

```
fun (n){
    Base Case
    Processing
    fun (n-1)
}
```

k_1, k_2 } k

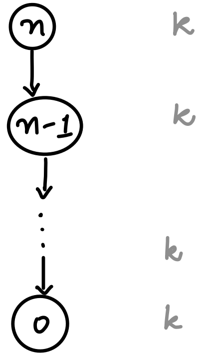
} k → pass by value
1 variable n is created

→ pass by reference
min space is constant



T.C. - $nk + k_1$

T.C. - $O(n)$

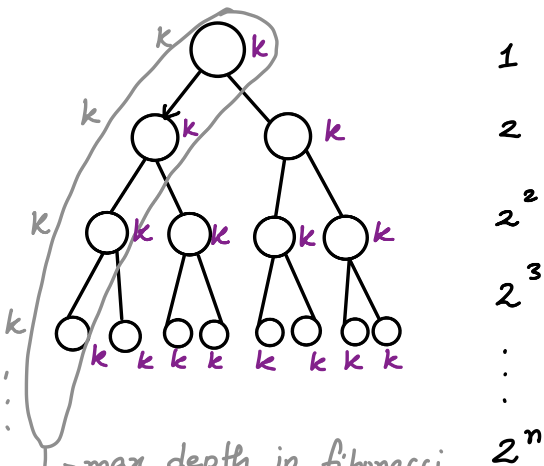


S.C. - $(n+1)k$

S.C. - $O(n)$

- Counting
- factorial
- power of 2
- check sorted
- max in array
- min in array

Type 2 Non Linear Tree



T.C. - $k + 2k + 4k + \cdots + 2^n k$

T.C. - $O(2^n)$

S.C. - nk

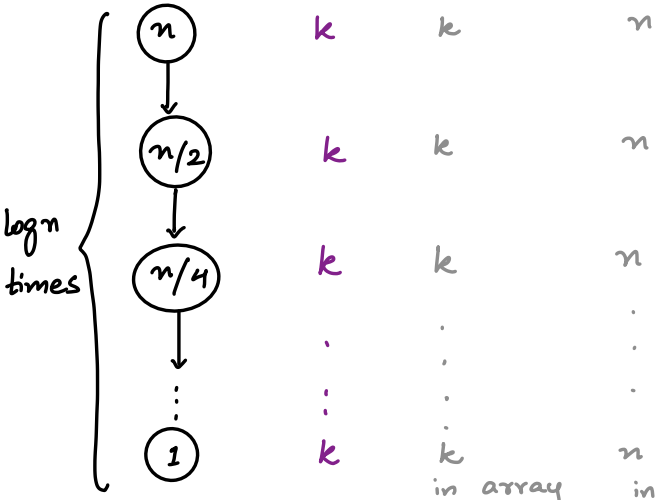
S.C. - $O(n)$

fibonacci

jump stairs

subsequences

Type 3 Logarithmic



T.C. - $\log n * k$ S.C. - $\log n * k$ S.C. - $\log n * n$

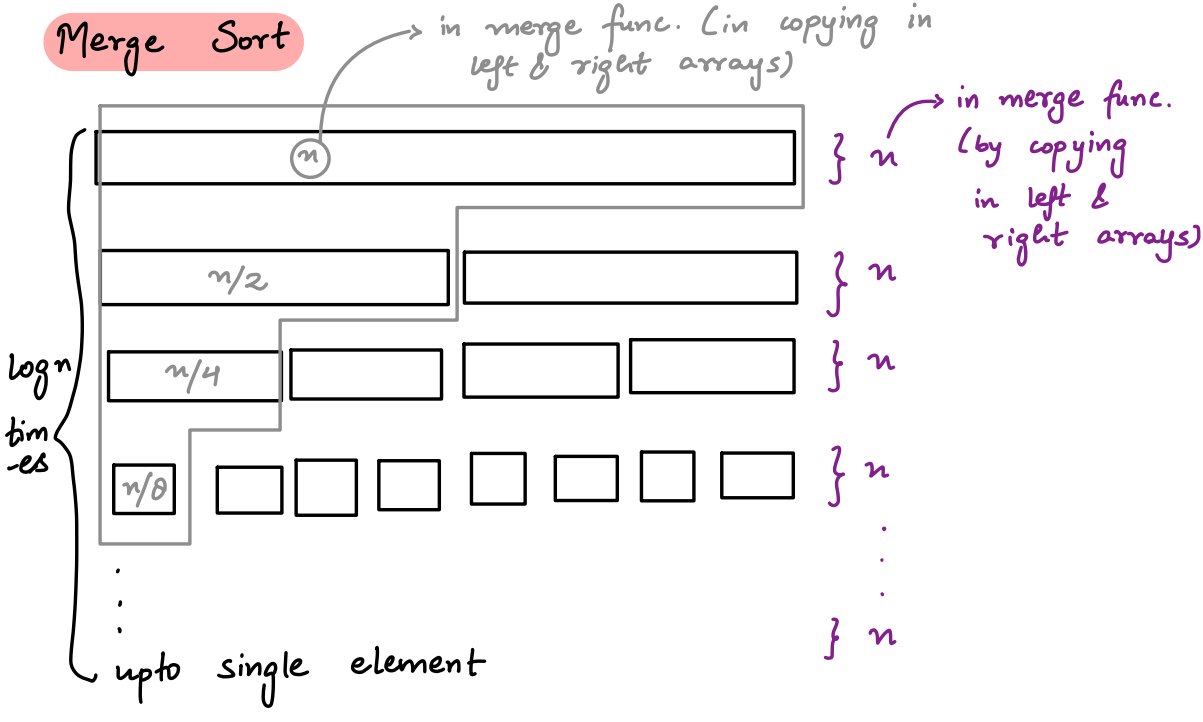
T.C. - $O(\log n)$ **S.C. - $O(\log n)$** **S.C. - $n \log n$**

binary search

array / vector pass by reference

vector pass by value

Merge Sort



T.C - time in 1 level * number of levels

T.C. - $n * \log n$

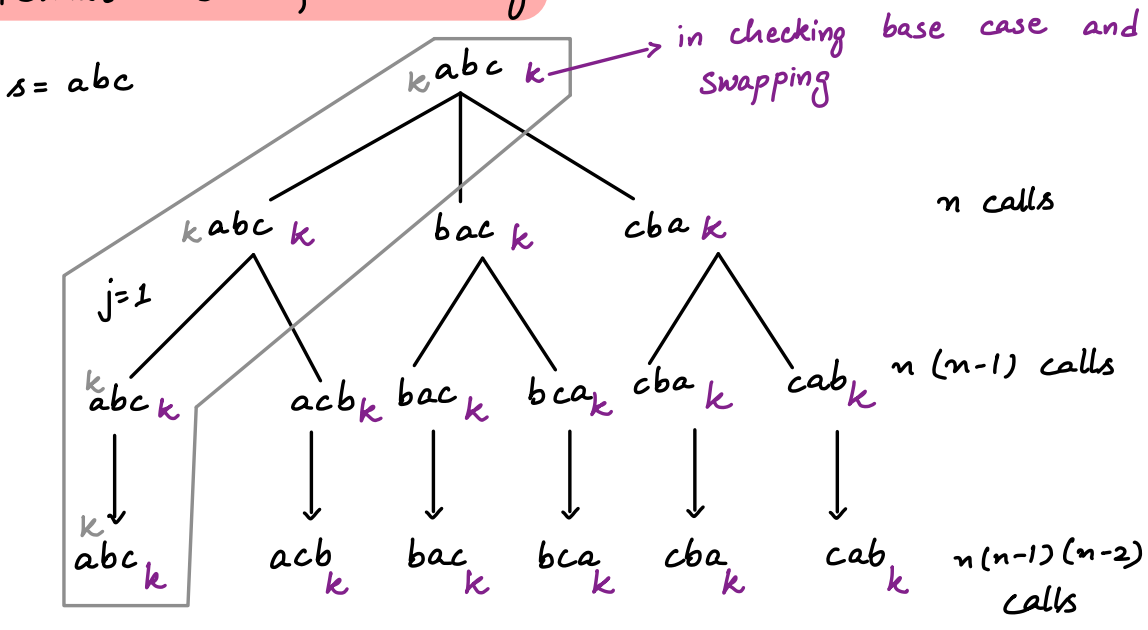
T.C. - $O(n \log n)$

S.C. - $n + \frac{n}{2} + \frac{n}{4} + \cdots + 2$

S.C. - $O(n)$

Backtracking Questions

Permutations of a string



T.C. - $k + nk + n(n-1)k + n(n-1)(n-2)k + \cdots + n!\,k$

T.C. - $O(n!)$

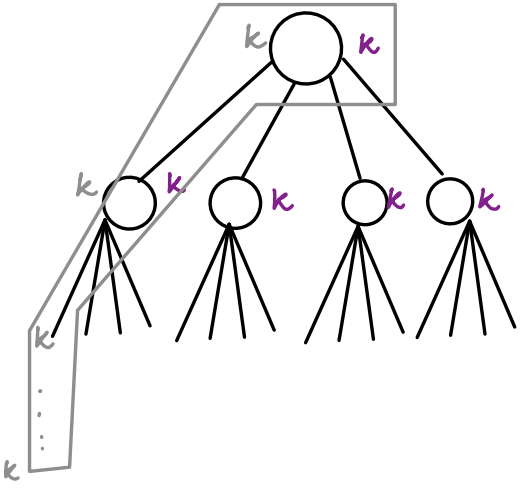
S.C. - $n * k$

S.C. - $O(n)$

Rat in a maze

m rows
n cols

```
func ( )
    base case

    4 recursive calls
        └> down , left, right, up
}
```



1
4
16
.
.
.

m * n levels

$T.C. - k + 4k + 4^2k + \cdots + 4^{mn}k$

$T.C. - O(4^{mn})$

$S.C. - k * mn + mn$ (depth; mn → of visited 2D array & pass by reference)

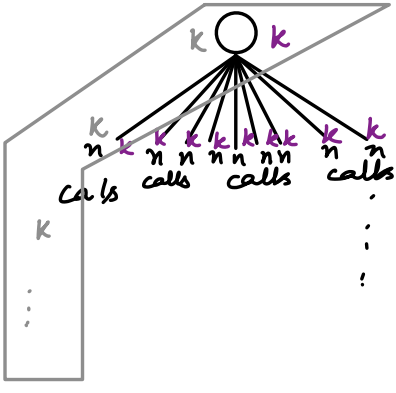
$S.C. - O(m * n)$

Sudoku Solve

$n * n$ matrix

Standard
↳ $n = 9$

```
func() {
    base case
    n recursive calls + isSafe func.
}
```



$T.C. - k + kn + kn^2 + \dots kn^{n*n}$

$T.C. - O(n^{n^2})$

$S.C. - k * (n*n)$ → depth

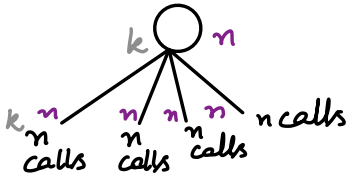
$S.C. - O(n^2)$

if n is already given 9, $T.C - O(1)$ & $S.C - O(1)$

N Queens

```
func ( )
    base case ──────────> k
    n is Safe function ──> O(n)      } O(n)
    n recursive calls
}
```

k { k_1, k_2



1

n

n^2

$\vdots$

n^n

k
.
.
.

T.C. - $n(1) + n(n) + n(n^2) + \cdots + n(n^n)$

T.C. - $O(n^n)$ ⟶ worst case

T.C. - $O(n!)$ ⟶ average case

S.C. - $k * n$

S.C. - $O(n)$

S.C. - $k * n + n$ ⟶ in optimized sol. for creating 3 maps

S.C. - $O(n)$

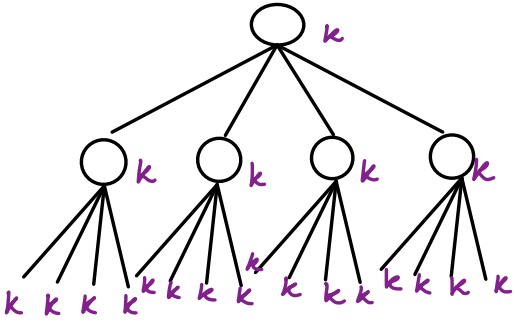
→ $n * n!$ can be written as $n!$

→ $n * 2^n$ can be written as 2^n

Phone Keypad Problem

1	2 abc	3 def
4 ghi	5 jkl	6 mno
7 pqrs	8 tuv	9 wxyz

```
fun ( ) {
    base case      k
      └→ store output in ans array
    at max 4 recursive calls
}
```



1

4

4^2

.

.

.

4^n

$T.C. - k + 4k + 4^2k + \ldots + 4^nk$

$T.C. - O(4^n)$

in max depth ↑ ; length of 1 output string ↑

$S.C. - (k + 2k + \cdots + nk) + (4^n * n)$

- $(k + 2k + \cdots + nk)$: store output in base case
- 4^n : no. of strings stored in ans array
- n : length of 1 output string

$S.C. - O(4^n)$

W9-L1

OPP

- Object Oriented Programming
- programming technique in which everything revolves around objects
  - creation
  - interaction
  - access
  - changes
- 2 things
  - property / state → variables
  - behaviour / method → functions
- why ?
  - readable
  - reusable
  - easily mantain
  - easily to understand

Class

- to make user defined / custom datatype
- has structure definition

class - design / blueprint / idea

object - actual entity

- instance of class
  - has existence

→ Class and Object both take memory

Creating a class

```
class className {
    // states
    string name;

    // method
    void sleep(){
    }
};
```

sizeof (class) -

```
class animal {

};
cout << sizeof(animal); ⟶ 1
```

to give existence for identifying this class, so min. possible memory is allocated

```
class animal {
    int age;
};
cout << sizeof(animal); ⟶ 4
```

```
class animal {
    string name;
};
cout << sizeof(animal);
```

⟶ 24 Why?

```
class animal {
    int age;
    char ch;
};
cout << sizeof(animal);
```

⟶ 8

↳ due to padding and greedy alignment

```
class animal {
    string name;
    string age;
    void sleep(){
        cout << name << " is sleeping";
    }
};
```

states / properties / data members

method / behaviour / member function

Static Object Creation

```
animal ramesh ;
```

Accessing state and method of object

(not class)

```
objectName. state ;
objectName. method(parameters);
ex- ramesh. age ;
    ramesh . sleep();
```

Access Modifiers

→ defines access scope of state / method

3 types

public

can access that state / behaviour inside as well as outside the class

private (by default)

can access that state / behaviour only inside the class, not outside the class

protected

same as private but can be access inside class child

```
class animal {
    public :
    string name ;
    string age ;
    void sleep () {
        cout << name ;
    }
    private :
    void eat () {
    }
};
```

→ after marking, all below state / behaviour will be of that type

To access private member outside class

↳ use public methods getter and setter

```
class animal {
    string name ;
    public :
    string getName () {
        return name ;
    }
```

getter

```
    void setName (string newName) {
        name = newName;
    }
};
int main() {
    animal ramesh;
    cout << ramesh.name;
    cout << ramesh.getName();
    ramesh.setName("Lion");
    cout << ramesh.getName();
```

setter

ERROR

gv

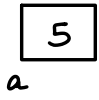
Lion

Dynamic Memory Allocation

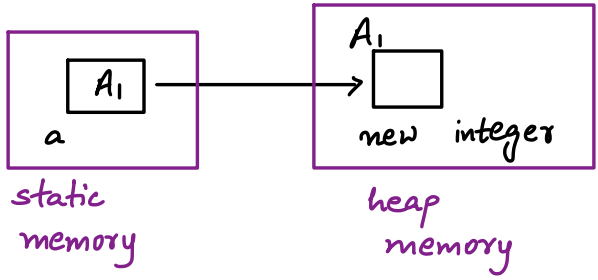
2 types of memory

- stack
  - small
  - static memory allocation
- heap
  - large
  - dynamic memory allocation
  - new keyword
    - allocate memory in heap
    - and returns it memory location address

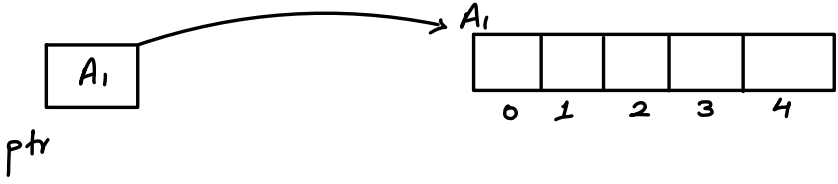
```
int a = 5;
```




```
int * a = new int
```




```
int * ptr = new int [5];
```



→ space allocated in heap memory, cannot be cleaned automatically even after finishing of func.

→ there is no garbage collector in C++

Allocation

```
int * a = new int;

int * arr = new int [5];
```

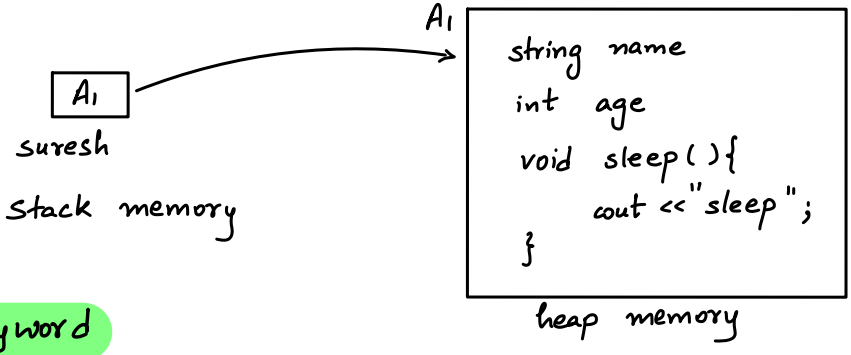
De allocation

```
delete a;

delete arr [];
```

Object Creation using dynamic memory allocation

```
animal * suresh = new animal;
suresh.name;  ⟶ ERROR as suresh is a pointer, * suresh is object

(*suresh).name;
(*suresh).sleep();
```
} 1st way of access

```
suresh → name;
suresh → sleep();
```
} 2nd way of access



this keyword

↳ pointer to current object

```
class animal {
    int age;
    public:
    void setAge (int age){
        this → age = age;
        // (*this).age = age;
    }
};
```
} both are correct

Object Creation

→ **Constructor** Call (in both static and dynamic allocation)

- → initialize the object
- → function with no return type
- → its name is same as class
- → created by default

```
class animal {
    int age;
    int weight;

    // default constructor
    animal() {
        cout << "Constructor Called";
        this->age = 10;
        this->weight = 20;
    }
```

now built-in constructor does not called, it will be called

→ accessing members and methods using this key word is GOOD PRACTICE

```
    // parameterised constructor
    animal(int age) {
        this->age = age;
        cout << "Parameterised constructor called";
    }
```

```
// parameterised constructor 2
animal (int age, int weight){
    this->age = age;
    this->weight = weight;
    cout << "Parameterised constructor 2 called";
}

// copy constructor (wrong way)
animal (animal obj) {
    this -> age = obj. age;
    this -> weight = obj. weight;
    cout << " Copy Constructor Called";
}

// copy constructor (right way)
animal (animal & obj) {
    this -> age = obj. age;
    this -> weight = obj. weight;
    cout << " Copy Constructor Called";
}
}
```

```
animal a;
animal b = new animal;
animal c = a;
animal d (a);
animal e = * b;
animal f (* b);
```

↳ copy constructor call
↳ pass by value
⇓
infinite loop

Destructor

- free memory
- for static object creation, destructor gets called automatically where object's scope ends
- for dynamic object creation, you have to do it manually using `delete objName;`
- no return type
- no input parameter

```
class animal {
    int age;
    int weight;

    // destructor
    ~animal(){
        cout << "destructor called";
    }
}
```

W9-L2

4 Pillers of OOPs

- Encapsulation
- Inheritence
- Polymorphism
- Abstraction (Pitaji of all OOPs pillers)

Encapsulation

- Wrap data members and member functions in one parent entity (class)
- Objective → Data Hiding
  - security
  - privacy
  - can make readonly
  - can decide what to hide and what to show
  - work with parent entity is simple

Objective of class → Encapsulation

Perfect / Full / 100% Encapsulation

- When all data members are marked private and access them using getter and setter

Inheritence

- inherit some properties (data members and data functions) from parent / super / base class to sub / child / derived class
- Objective → Reusability
- is a relationship
- syntax
  - class childName : mode1 parentName1, ..., modeN parentNameN {
    //additional states and methods
    }

childName is a parentName

```
class Animal {
    int age;
    public:
    int weight;
    protected:
    void eat() {
        cout << "eating";
    }
};
```

```
class Dog : public Animal {
};
```

Dog is a Animal

```
class cat : public Animal,
    protected Dog {
}
```

Access Modifier of Base class	Mode of inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	NA	NA	NA

NA → Not Accessible
└→ cant be inherited

Protected

└→ Same as private access modifier but can be inherited

↓

can be accessed inside child class

Private

→ can access inside class but not outside

→ cannot be inherited

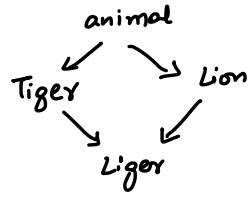
↓

cannot be accessed inside child class

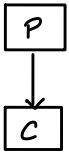
P- Parent C- Child

Type of Inheritence

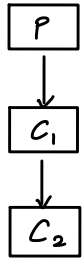
- Single
- Multilevel
- Multiple
- Heirarchical
- Hybrid → mixture of all above 4



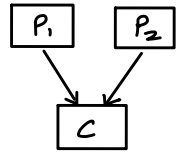
Single



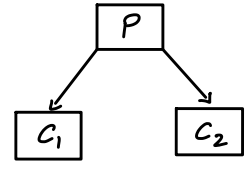
Multilevel



Multiple



Heirarchical

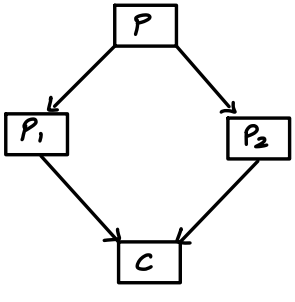


Diamond Problem / Inheritence Ambiguity Problem

→ If multiple parents have same property then how to access them

→ objectName . ParentClassName :: property ;

scope resolution operator



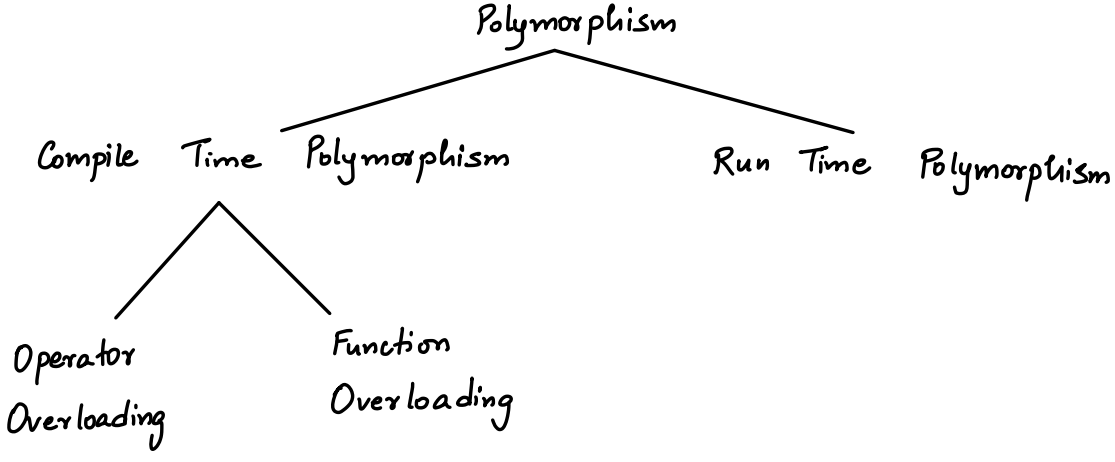
→ If parent and child have same property, then parent property inside child class will get overwrite by child property

→ Multiple inheritence is possible in C++, not in Java

Polymorphism

→ form

→ existed in many forms



Function Overloading

→ one function existing in multiple forms / signatures

→ Multiple Signatures

→ diff. in either no. of parameter or type of parameters, not diff. in return type

→ 5.12 ⟶ double

5.12f ⟶ float

```
class Maths {
    public:
    int sum (int a, int b) {
        return a+b;
    }
    float sum ( int a, double ){
        return a+b;
    }
    int sum( int a, int b, int c){
        return a+b+c;
    }
};

int main () {
    Maths m;
    cout << m. sum ( 5, 2);  ------------> 7
    cout << m. sum( 5, 2.5); ------------> 7.5
    cout << m. sum( 5, 2, 7); ------------> 14
    return 0;
}
```

Operator Overloading -

→ to make new implementation of an operator

object 1 op variable 2

- → same as
  object 1 . op (variable2);
- → object 1 → current object
- → op → method
- → variable 2 → input parameter

```
class Para {
    public :
    int val ;

    Para ( int val ) {
        this -> val = val ;
    }

    int operator + ( Para obj2 ) {
        return this -> val - obj2. val ;
    }

    int operator + ( char ch ) {
        cout << this -> val + 10 ;
    }

};
```

```
int main (){
    Para a(5);
    Para b(2);
    cout << a + b;
    a + 'c';
    a + '7';
    return 0;
}
```

→ "operator +" function must have exactly 1 parameter as + is binary operator

↓

2 operands

current object — input parameter

Run Time Polymorphism -

→ polymorphism after execution

Function Overriding -

Function / Method Over-riding

- when function of parent class is defined seperately inside child class
- reusable + custom behaviour where ever you want

```
class Animal {
    public:
    void speak (){
        cout << "speak";
    }
};

class Dog : public Animal {
    public:
    void speak (){
        cout << "bark";
    }
};

int main (){
    Animal a;
    cout << a.speak();

    Dog d;
    cout << d.speak();
```

```
class Animal {
    public:
```

// upcasting

```
Animal * b = new Dog ;
```

```
cout << b. speak();
```

——————> speak ——> function of parent class called

```
Animal * b = new Dog;
```

```
cout << b. speak ();
```

——————> bark

// down casting

```
Dog * c = new Animal ;
```

——————> Compiler dependent (ERROR or not)

```
cout << c. speak ;
```

——————> bark

```
Dog * c = new Animal ;
```

OOPS Week is not completed from my side will complete in future

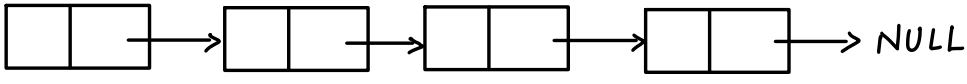
Abstraction

- Objective → Implementation Hiding
- Talk about in a larger scale
  not focus on detail

LINKED LIST

W10-L1

- Dynamic linear data structure to store similar data
- non-continuous memory allocation
- insertion and deletion is easy
  - O(1) if you are standing on that node / pointer
- no concept of indexing,
  it has concept of addresses
- collection of nodes
  - 

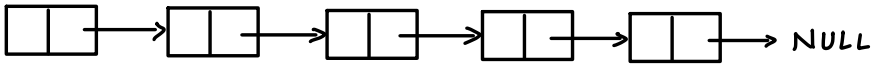


node creation (simplest)

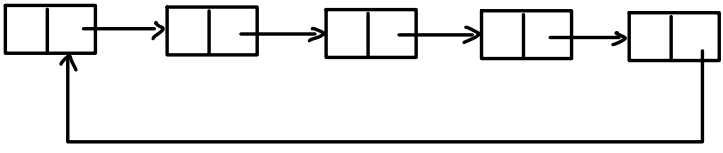
```
class node {
    public:
    int data;
    node * next;
}
```

Types of linked list

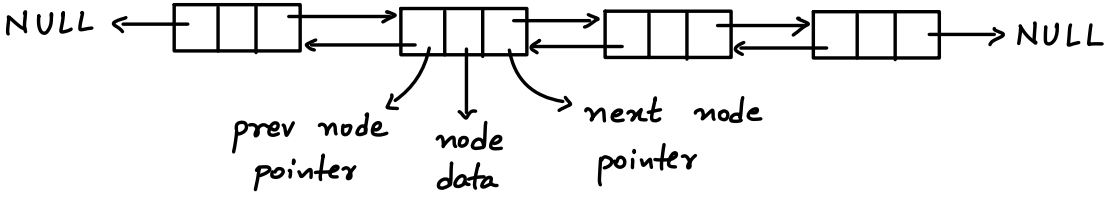
- Singly LL



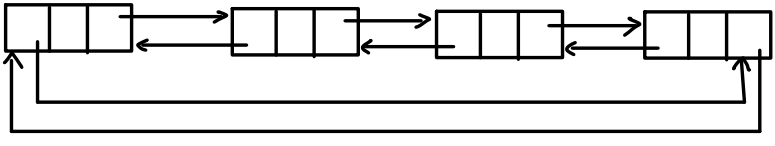
- Circular LL



- Doubly LL



- Circular Doubly LL



→ LL is nothing but Hindi (any other language)

→ Linear Data Structure

↳ single descendent

→ In LL ques, there are high chances of error, so check edge cases (head, tail, size=1, not found, and more) before submit.

W10-L2

Doubly Linked list

→ node creation

```
class node {
    public:
    int data;
    node * next;
    node * prev;
}
```

Circular Linked List

→ no Head and Tail

→ insertion & deletion is only on the basis of value

→ Converting loop code to recursion code

→ Add parameters that are changing in loop

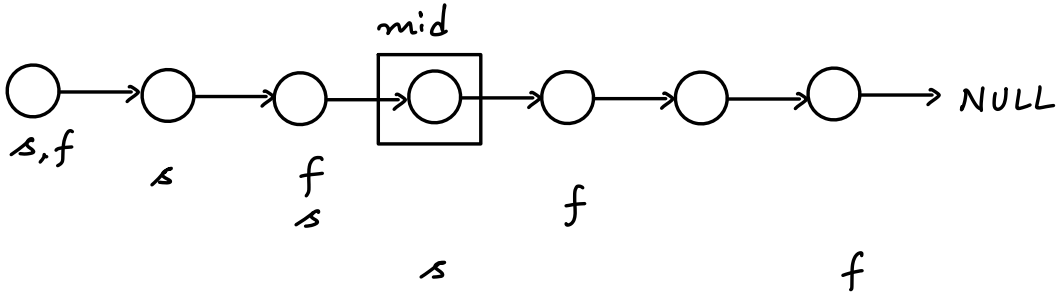
→ Stopping condition in loop acts as base case

Tortoise Algorithm

W10-L3

- fast and slow pointer approach
- get middle

```
int getMiddle (node * head) {
    if (head == NULL)
        return -1;

    node * fast = head;
    node* slow = head;

    while (fast != NULL){
        fast = fast -> next;
        if (fast != NULL){
            fast = fast -> next;
            slow = slow -> next;
        }
    }
    return slow -> data;
}
```

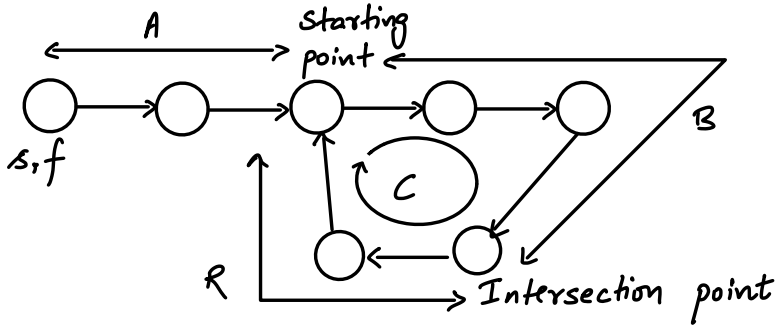


Floyd's Cyclic Detection Algorithm

- to detect and delete loop in linked list
- detect the loop in linked list

  find starting point of loop in linked list

  delete the loop from linked list



distance (f) = 2 * distance (slow)

$A + B + xC = 2(A + B + yC)$

$A + B = (x - y)C$

A + B made some cycles

$B = (x - y)C - A$

So after B steps from intersection point, some no. of cycles starting from starting point get completed

So intersection point + A ⇒ reach starting point

Also head + A ⇒ reach starting point

$$R + zC = A$$

```
162  // also works if loop starts from head
163  void removeLoop(node* &head){
164      if(head==NULL)
165          return;
166
167      node* fast = head;
168      node* slow = head;
169      bool loopFound = false;
170
171      while(fast!=NULL){
172          fast = fast->next;
173          if(fast != NULL){
174              fast = fast->next;
175              slow = slow->next;
176          }
177
178          if(fast==slow){
179              loopFound = true;
180              break;
181          }
182      }
183
184      if(loopFound){
185          slow = head;
186          while(fast!=slow){
187              fast = fast->next;
188              slow = slow->next;
189          }
190
191          cout << "loop started from " << slow->data << '\n';
192          node*prev = slow;
193          while(prev->next!=slow){
194              prev = prev->next;
195          }
196          prev->next = NULL;
197
198      }
199      else{
200          cout << "no loop was present\n";
201      }
202  }
```

node *&head

also works if loop starts from head

STACKS

W11-L1

- Data Structure
- LIFO (Last In First Out) Order
- no index based accessing, only can access top

initialization

```
stack <int> s;
```

insertion

```
s.push(20);
```

deletion

```
s.pop();
```

top element

```
s.top();
```

can only be use if we are sure that stack is non empty

empty or not

```
s.empty();
```

size

```
s.size();
```

→ If we pop a data set after pushing in stack, we get the answer in reverse order

So to reverse anything or to use LIFO order you can use stack

→ Stack can be implement either by array (static or dynamic) or linked list

{ in implementation dont use vector as it is stl class }

Copying a stack

W11-L2

```
stack <int> st1;
stack <int> st2 = st1;
```

→ if we have to do something for every element of stack, generally we need to write 2 functions,

first -> for removing top, storing it and use recursion

second -> for doing that something on a single element (top) of stack

ex - sort a stack

↳ insert top of stack in sorted stack

reverse a stack

↳ insert top of stack at the bottom

→ General format to solve standard stack question

```
void func (st)
    if (st.empty()) {
        processing
        return;
    }
    int top = st.top();
    st.pop();

    func(st);
    processing
}
```

base case

save top

pop

recursion call

processing

Pair -

```
pair<int, char> p;
p = make_pair(27, 'x');
p.first = 27;   p.second = 'x';
p = {27, 'x'};
```

} initialize

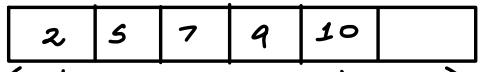
} insert

W11-L3

QUEUE

W12-L1

→ Data Structure in which data is stored in the form FIFO (first in first out) order



front

deletion

first element of queue

q.front()

q.back()

rear

insertion

place to insert next element

rear always shows empty block

creation

```
queue <int> q;
```

insertion -

```
q.push(10);
```

deletion -

```
q.pop();
```

front element

```
q.front();
```

back element

```
q.back();
```

size

```
q.size();
```

empty or not

```
q.empty();
```

→ queue can be implemented using either by dynamic array or by linked list

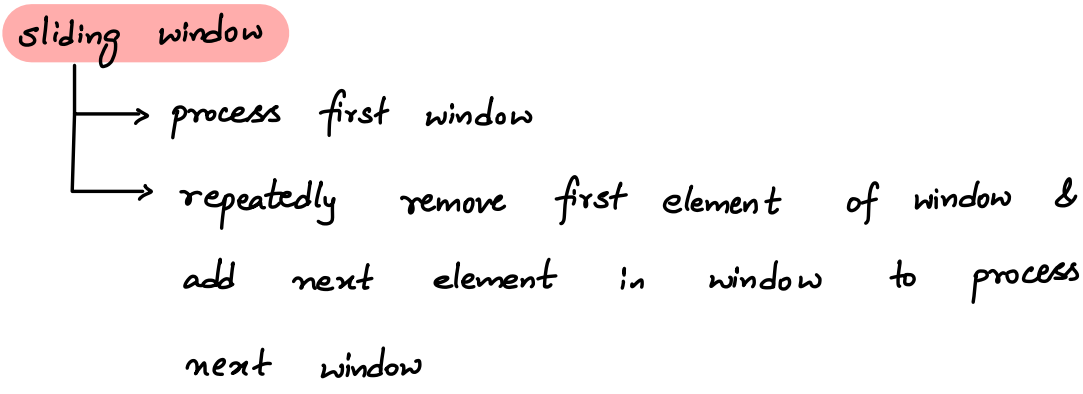
→ if front == rear ⟶ queue is empty

→ element access through index is not possible

→ if iterative and recursive, both have same complexities, then iterative sol. takes less time generally

→ **queue patterns**

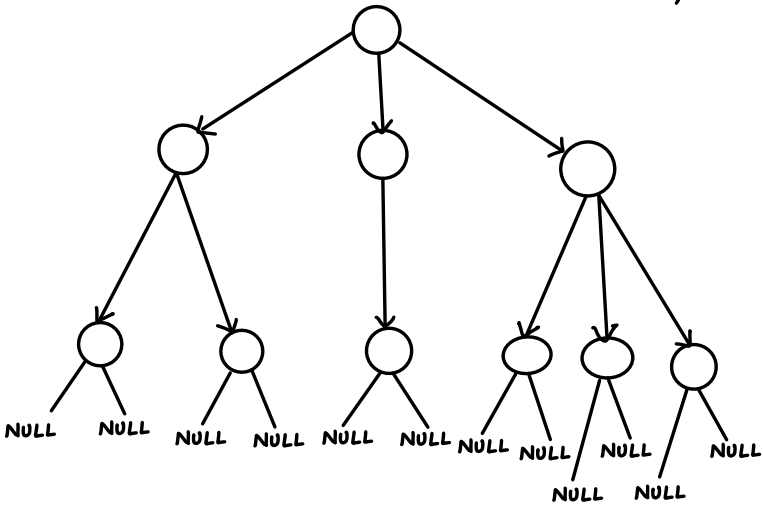
- → queue / deque in sliding window
- → traversals (trees and graph)
- → use after sorting vector
- → circular tour ques.


sliding window
process first window
repeatedly remove first element of window & add next element in window to process next window

TREE

W13-L1

→ non linear data structure in which data is stored in heirarchical form



Binary Tree

→ Tree having no. of children for each node is less than or equal to 2

n-ary tree

→ Tree having no. of children for each node is less than or equal to n

→ generic tree

node

→ smallest unit of tree

for binary tree

```
class node {
    int data;
    node * left;
    node * right;
}
```

for n-ary tree

```
class node {
    int data;
    vector <node*> child;
}
```

root node

- → top most / starting node of the tree
- → only one in whole tree

leaf node

→ node with no child

non leaf node

→ node other than root and leaf nodes

parent ⟶ child

→ parent of root node is either NULL or itself or anything else according to the ques

ancestor

→ all nodes in the path from that node to root node

descendent

→ all nodes in the path from that node to any no. of leaf nodes

sibling

→ nodes that are children of same parent

level

→ start from root as 0 and increase as we go below towards leaf by 1

height of node → no. of nodes/edges (acc. to ques) from leaf to node in longest path

→ start from bottom most node as 1 and increase as we go upwards towards the root by 1

height of tree

→ height of root node

→ trees questions

- → online assessment
  - → mcq → simple or formula based (total no of nodes/edges)
  - → coding
- → interview

→ mostly classical ques and sometimes hard ques using multiple data structures

→ tree is aasaan if you know recursion

Tree traversal algorithms

- → level order traversal (BFS)
- → spiral traversal (BFS)
- → preorder, inorder, postorder traversal (DFS)
  - preorder → CLR
  - inorder → LCR
  - postorder → LRC

  C - process curr node L - goto left R - goto right
- → moris traversal

→ there is no official algo to create tree from data

Skew tree

↳ tree in which each node has only left child

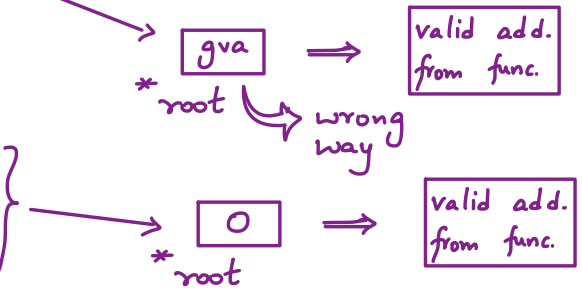
or

tree in which each node has only right child

→

```
int main (){
    // wrong way
    node * root = func();

    // right way
    node * root = NULL;
     root = func();
}
```

```
node * func (){
    return node;
}
```



→ if we have to store nodes, then store their pointers (node *) as we know that node size can either be larger but node pointer will always of same size i.e. either 4 or 8 acc. to compiler Another reason is that multiple nodes can have same data but with diff. addresses

W13-L2

Balanced Binary Tree

- at every node the absolute diff. between heights of left and right subtree must be ≤ 1

Perfect / Full Binary Tree

- every node has either 0 or 2 children and only last level has leaf nodes

Complete Binary Tree

- tree in which nodes are created levelwise and from left to right
- all levels are completely filled except last one

Sum Tree

- replace every node value with sum of values of that node, left subtree sum, and right subtree su

→ if you dont find logic to solve tree question, just start running recursion in examples and you fill find solution eventually

W13-L4

→ There are only 2 approaches to solve any tree question

- → use recursion in both left and right subtree

  OR

- → find path from root to that node / leaf

→ build tree from inorder and preorder (CLR)

→ build tree from inorder and postorder (LRC)

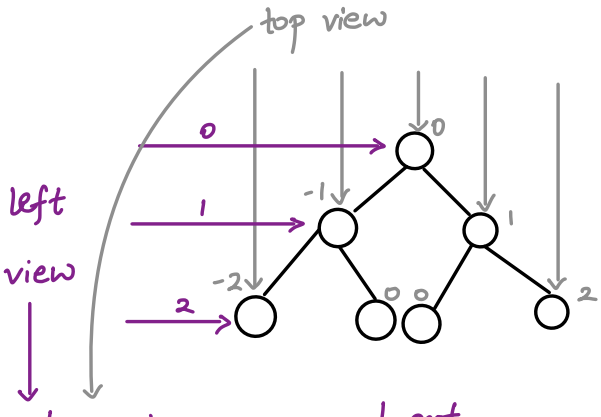
- → right recursion call before left recursion call

→ In ques. where we find views

W13-L5

(like top / left / bottom / right view of the tree)

we use 3 concepts → map
→ level order traversal
→ horizontal distance from root



left view ↓

store in map and not update if another found on the same level / horizontal distance

for bottom and right view store in map and update if another found on the same level / horizontal distance

left view & right view ⟶ 4 approaches
↓
in github repo code

BINARY SEARCH TREE

W14-L1

→ for every node, every data in left subtree should be smaller than node data and every data in right subtree should be greater than node data

→ inorder traversal of BST is always sorted
  → store or use acc. to ques

→ in average case, height of tree is $\log_2 n$
  ↓
  simple & BST both

→ in interviews, there is very high chances of asking variation of deletion in BST

→ given a leaf node, find its successor and predecessor leaf node of that node

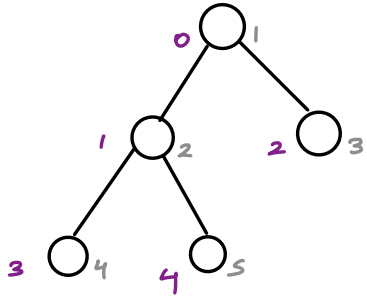
HEAPS

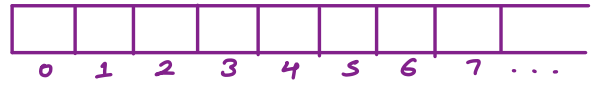
W15-L1

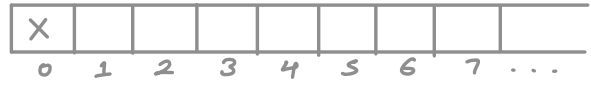
→ Data Structure that exists in the form of Complete Binary Tree and holds heap property

max heap - every parent value is greater than its children value

min heap - every parent value is less than its children value







i^{th} node → left child - $2i+1$, right child - $2i+2$
→ parent - $(i-1)/2$

i^{th} node → left child - $2i$, right child - $2i+1$
→ parent - $i/2$

→ We visualize heap as tree but create as array

insertion and deletion

insertion

- insert at bottom
- rearrange the nodes if heap property is violated
  - compare node with its parent
- T C. - O(n)

deletion

- deletion is only done of top / root node
- replace last value with top node
- delete last node (reduce size by 1)
- rearrange the nodes if heap property is violated
  - compare node with both of its children
- T. C. - O(n)

Use Case of Heap

- to find maximum & minimum element in O(1) time

Heapify

- process of rearranging a node to its correct position starting from that node as tree
- same as deletion code
  - rearrange the nodes if heap property is violated
    - compare node with both of its children

→ if there are n nodes in heap (CBT), then from $\left(\frac{n}{2}+1\right)^{th}$ to n^{th} nodes (in 1 based indexing) are leaf nodes and so are valid heap and there is no need to heapify those leaf nodes

Build heap

- convert a given array to a heap
- use heapify function
- T.C. - O(n)

Heap Sort

- swap first node and last node
- reduce size by 1
- run heapify algo
- repeat these steps till size becomes 1
- T.C. - O(nlogn)

DYNAMIC PROGRAMMING

W17-L1

- technique used to optimize sol.

- **key point**
  - those who forget the past are condemned to repeat it

    ⇩

    dont compute the things, you have already completed

    ⇩

    store all the distinct answers you are getting within the process

Phases of question solving

- Recursion
- Top Down Approach (Recursion + Memoisation)
- Bottom Up Approach (Iterative + Tabulation)
- Space Optimization

2 Conditions needed to apply DP

- overlapping subproblems
  - solving same problem again & again
- optimal substructure
  - optimal sol. of bigger problem can be calculated using optimal sol. of smaller problem

Top Down (Recursion + Memoisation)

- create and initialize the dp array or map
- store all the new answers in dp array and return that
- check whether the ans is already exists in dp array

write condition after base case

```
if (dp array contains ans for given input)
    return ans;
```

Bottom Up (Iterative + Tabulation)

- create and initialize dp array / map
- store base case answers in dp array
- find ans. from bottom to up (reverse of recursion) iteratively (using loops) and store them in dp array.
- add extra base case(s) if necessary

S.C. (bottom up) is better than S.C. (top down)

↓ iterative

↓ recursive

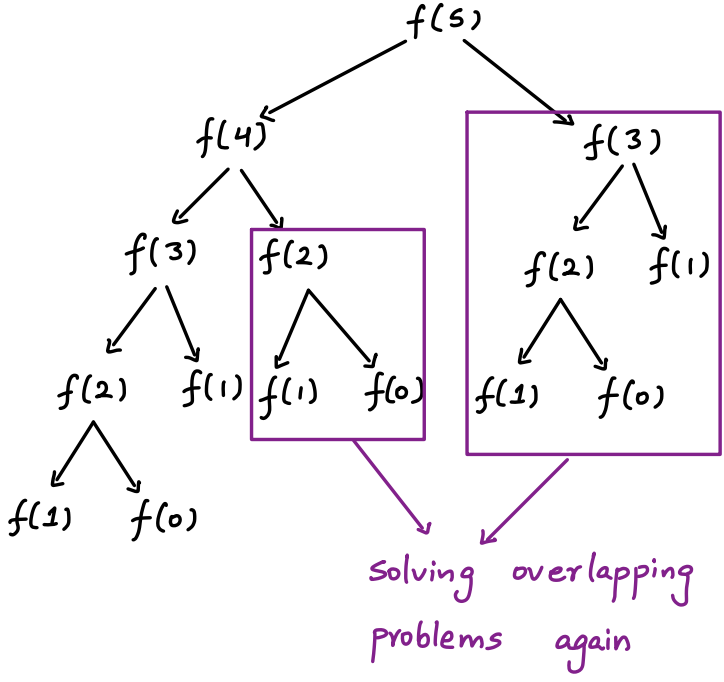
→ In majority cases size of dp array in bottom up is same as that of top down, but in some we have to change it

→ No. of dimensions of dp array (1D, 2D, etc) depends upon the no. of changing parameters in the ques

Space Optimization Sol.

- check where the useless space is used and try to reduce it.
- Analyze the dp expression and findout to what parameters curr value is dependent and find out the pattern in those parameters

ex- fibonacci



recursion
T.C. - $O(2^n)$
S.C. - $O(n)$

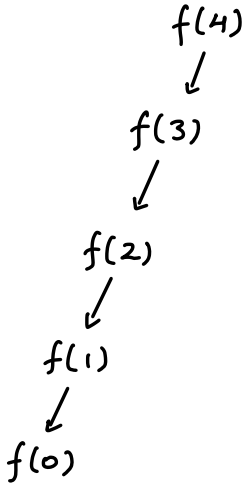
top down
T.C. - $O(n)$
S.C. - $O(n+n)$

bottom up
T.C. - $O(n)$
S.C. - $O(n)$

space optimization
T.C. - $O(n)$
S.C. - $O(1)$

DP can be used

factorial



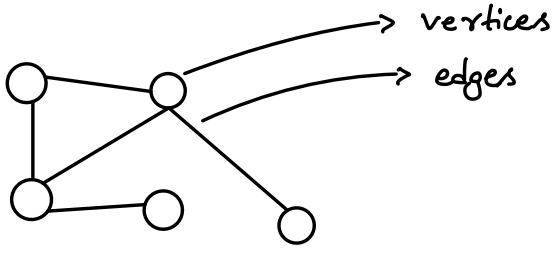
no overlapping problems

only have optimal substructure

DP cannot be used

GRAPHS

→ Data Structure made up of nodes and edges (vertices)



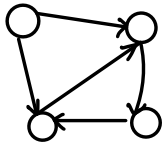
→ use case

↳ google map, facebook

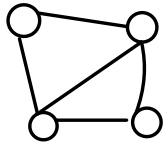
Types of graphs

→ Directed Graph

→ Undirected Graph

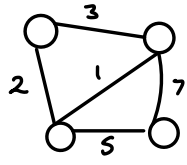


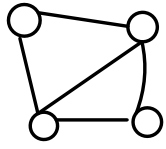
Directed



Undirected

↳ Weighted Graph





if weight is not given and we need it, assume it as 1

Degree

↳ no. of edges attach to a node in undirected graph

↳ types of degree

↳ indegree

↳ no. of incoming edges to a node

↳ out degree

↳ no. of outgoing edges from a node

Path

↳ sequence of nodes you traversed such that a node visits atmost once

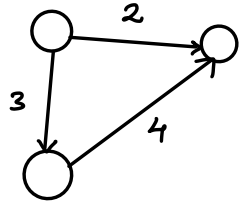
Cyclic Graph

- Graph in which at least one cycle is formed
- All undirected graphs are cyclic

Acyclic Graph

- Directed Graph that has no edges

Weighted Acyclic Directed Graph



Graph Implementation -

- Adjacency Matrix → 2D array → element - 0/1 or weight
- Adjacency List → Unordered map of vector / list of int / pair
  - ↳ {node, weight}
  - → majorly used