

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 discuss with HR or team
- Think twice, Code once
- To clarify any code you are confused, use cout statements everywhere 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 quizzes 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 studying to get fun

You are studying 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

To 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 khatar nak 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 don't 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
- Don't 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 datatype 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

→ Don't 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

$1 \longrightarrow$ in -ve no.

$0 \longrightarrow$ in +ve no.

$(-1) \gg n \begin{matrix} \longrightarrow & (-1) & \text{no change} \\ \longmapsto & \text{any} & \text{no. of times} \end{matrix}$

→ `int a = 5;`

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

$$a * (++a), a = 6$$
$$a * a, a = 7$$
$$7 * 7 = 49$$

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

$$5 * (t + a), a = 6$$
$$s^* a, a=7$$
$$5 * 7 = 35$$

```
count << (++a) * (a++);
```

$$a^*(a++) , a = 6$$
$$a^* 6, \quad a = 7$$
$$7 * 6 = 42$$

```
cont << (a++) * (a++);
```

$$5 * (a++) , a = 6$$
$$5^* 6, a =$$

30

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

→ `int main()` {

`return 0;` → returns to OS

}

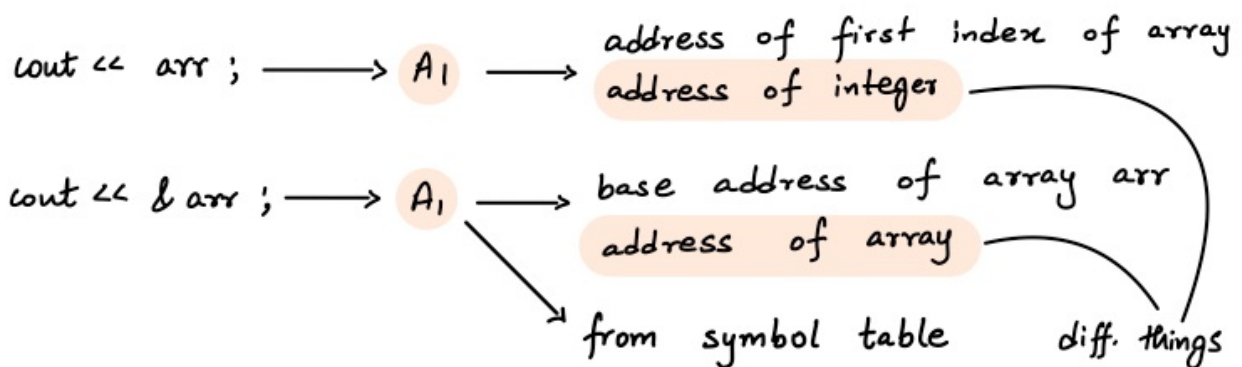
0 is used to represent
successful execution

→ A cpp file can't have more than 1 main function

→ `main()` can't have return type other than `int`
in offline compiler

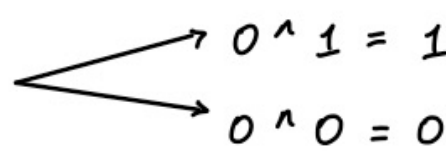
→ `%` is heavy operator, so try to use it less

→ 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

→ $0 \wedge \text{ans} = \text{ans}$ 
 $0 \wedge 1 = 1$
 $0 \wedge 0 = 0$

→ Mapping of 2D array with memory 1D array

$\text{linear_index} = c * i + j$
no. of cols ← c
row index ← i
col index ← j

$i = \text{linear_index} / c$

$j = \text{linear_index} \% c$

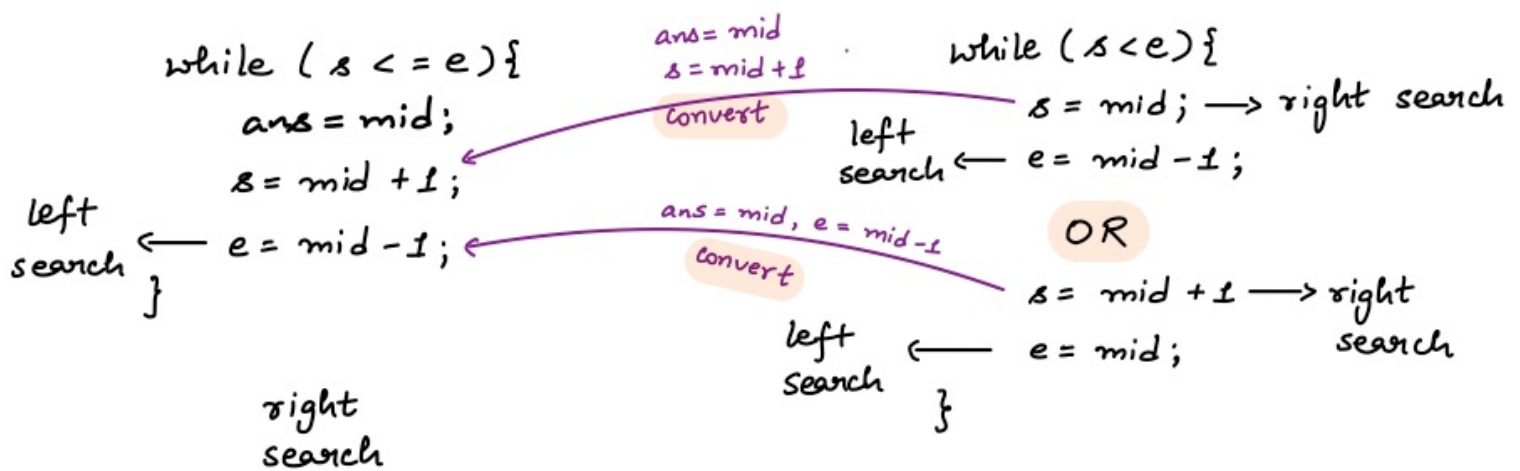
→ Arrays and 2D arrays of any datatype
└→ pass by reference

→ 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 / 3 pointer 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, 86, 87, 89, 90, 93, 95

→ 14

104

LET'S GO

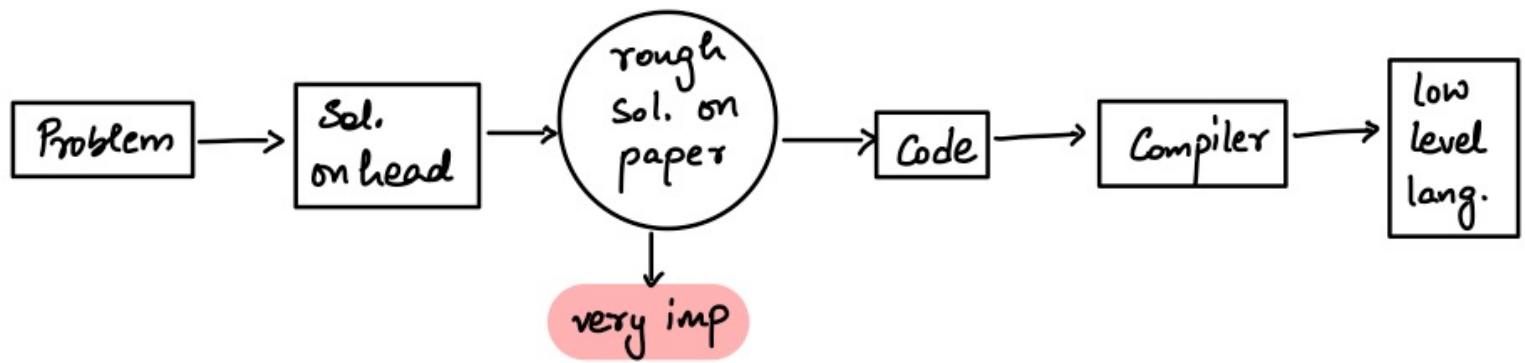
Thought process to solve a problem-

W1-L1

→ Understand a problem

→ input values

→ find approach



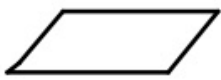
Algorithm- Sequence of steps

Flowchart- Graphical representation of algo

Components-



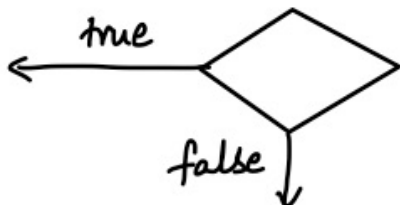
terminator for start / end



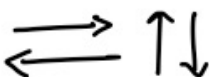
for input / output read / write



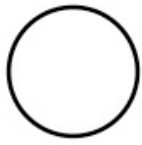
computation / process / declaration



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";
}
```

Annotations for the code above:

- `<iostream>`: preheader file contains implementation of keyword
- `using namespace std;`: using standard namespace implementation of cout choosing from multiple types of namespace
- `int main () {`: region where scope of keyword is defined
- `cout << "Namaste Bharat";`: used to print on console / standard display
- `cout`: to end any statement
- `"Namaste Bharat"`: string

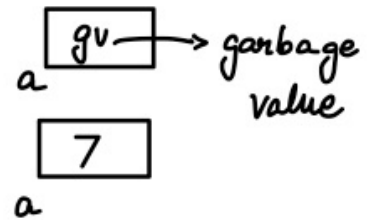
```
cout << endl;
cout << '\n';
```

Annotations for the code above:

- `endl`: for next line
- `'\n'`: for next line

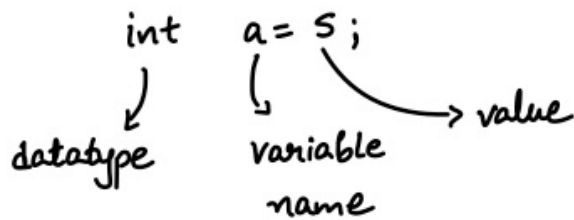
int a; $\longrightarrow$ a is an integer

cin >> a; $\longrightarrow$ input a from user
ex-7



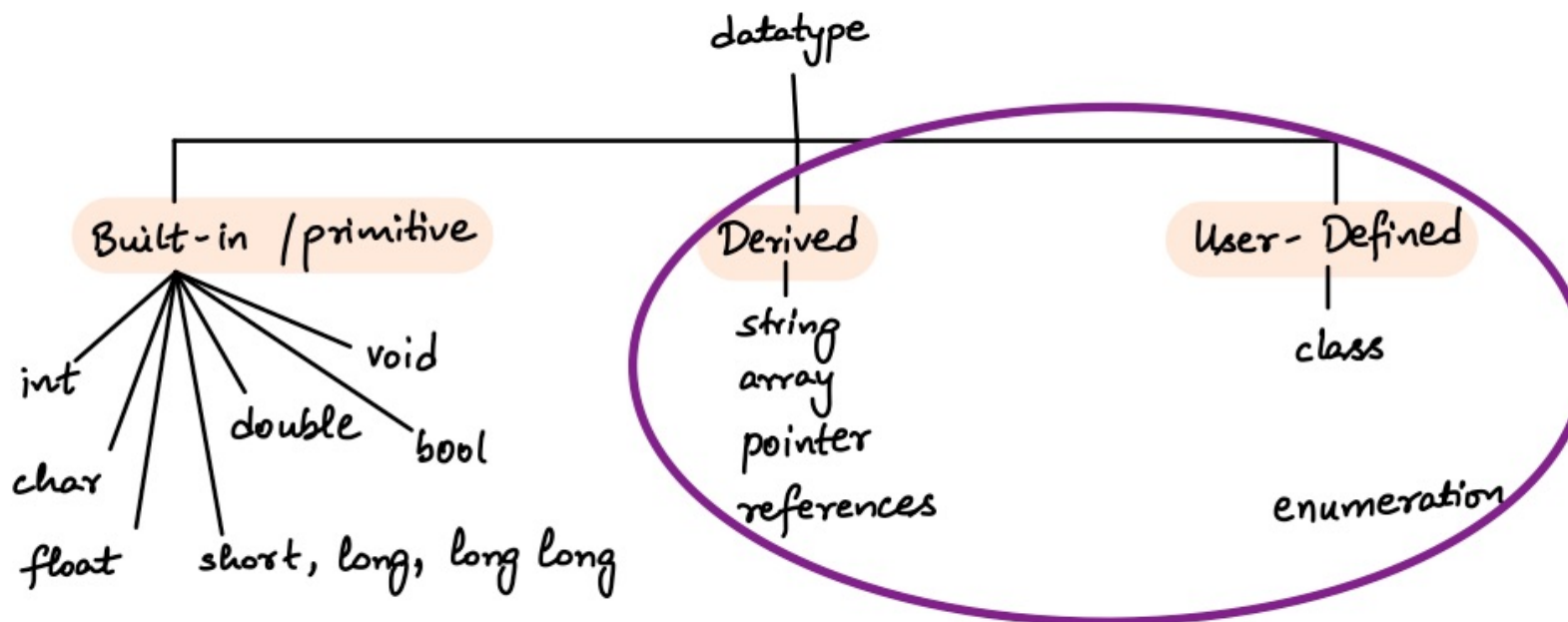
Variables

named memory location



Datatypes

type of data



int - 4 byte - 32 bits in memory

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

char - 1 byte - 8 bits in memory

$\longrightarrow$ 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

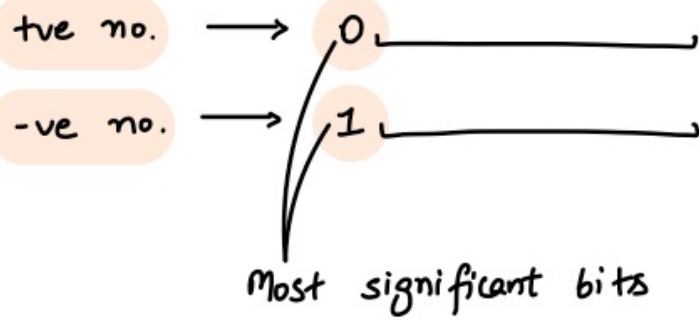
short → 2 byte → 16 bits

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

$\rightarrow$ 1's complement + 1
 $\rightarrow$ reverse all bits

int a = -7;

7 $\rightarrow$ 0...00111 } 32 bits

ignore -ve sign
find binary equivalent

1's (7) $\rightarrow$ 1...11000

find 2's complement

2's (7) $\rightarrow$ 1...11001

$\rightarrow$ this is how -7 will be stored in memory

How to read -ve no. present in memory

$\rightarrow$ take 2's complement

1...11001

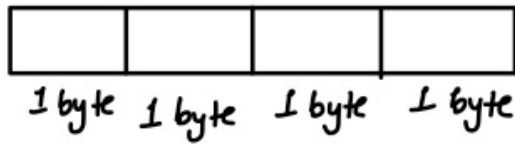
$\rightarrow$ 1's complement $\rightarrow$ 0...00110

2's complement $\rightarrow$ 0...00111

$\rightarrow$ +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

↳ 0, +ve

↓
+ve, -ve, 0

↳ by default

int - 4 byte - 32 bits in memory

↳ total no. of combinations - 2^{32}

signed int

↓
 -2^{31} to $2^{31}-1$

①0...0

011...1

2's → 10...0

→ -2^{31}

unsigned int

↓
0 to $2^{32}-1$

0.....0

1.....1

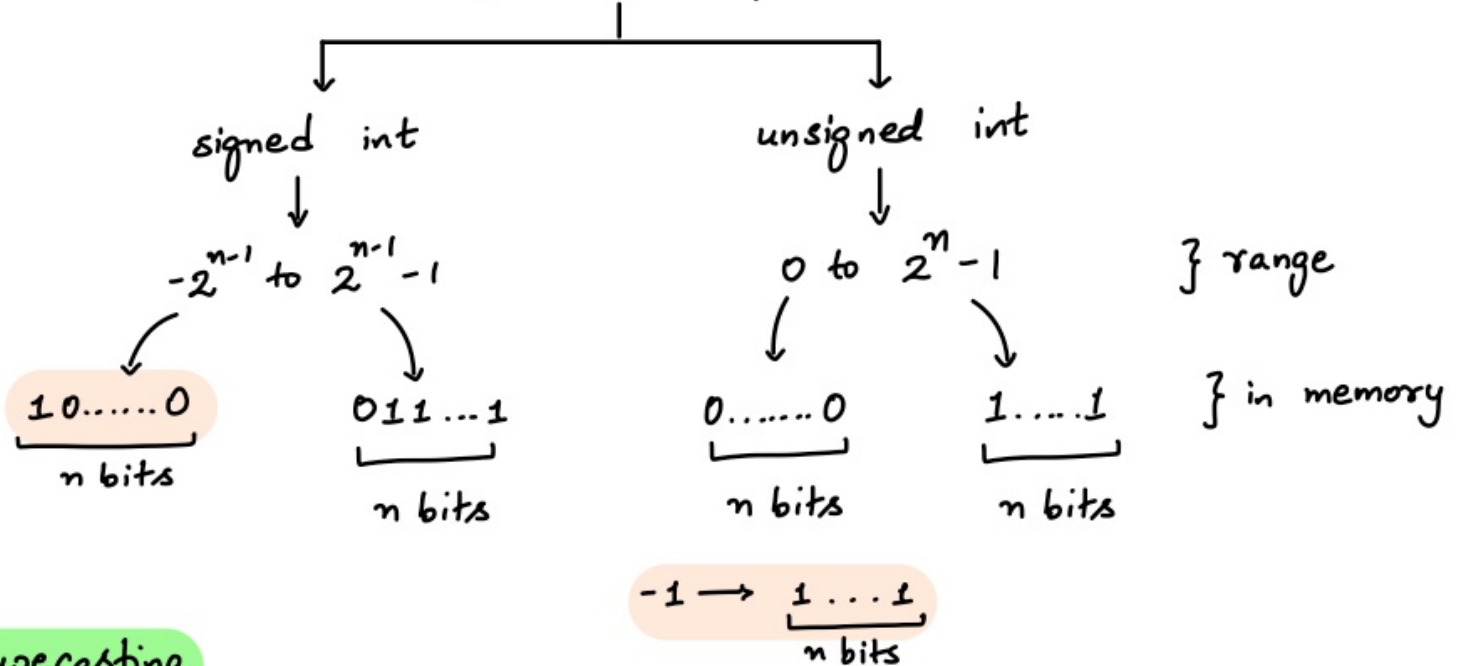
} range

} in memory

General Formula

n bits in memory

↳ total no. of combinations - 2^n



Typecasting

↳ convert one type of data to another

implicit type casting

ex - char ch = 97;

cout << ch; $\longrightarrow$ (a)

explicit typecasting

ex - char ch = (char) 97;

cout << ch; $\longrightarrow$ (a)

overflow ex - char ch = 9999;

cout << ch;

9999 $\xrightarrow{\text{binary conversion}}$ 100 111 0000 1111
stores only last 8 bits

so ch stores 00001111 in memory
↓
7
↓
acc. to ASCII table

Operators -

Arithmetic 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 → cout << sizeof(3.0);
not int ↓
⑥

Relational Operator

(a op b)

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

Output - 0 or 1

false ←

→ true

these are different things

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

$(cond1 \&\& cond2 \&\& cond3)$

if $cond1$ is false

compiler will not check further
as ans will already false

$(cond1 || cond2 || cond3)$

if $cond1$ is true

compiler will not check further
as ans will already true

Conditions

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

if

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

if-else

```
if (cond1)  
    execute 1  
else if (cond2)  
    execute 2
```

if-else if

W1-L3

```

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

```

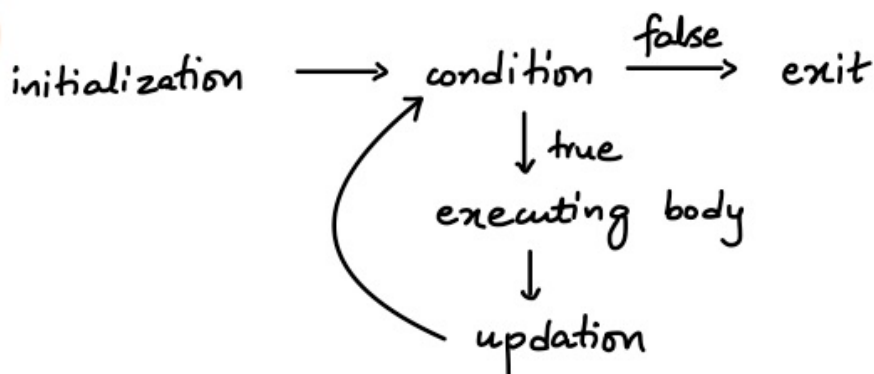
for (int i = 0; i < 5; i = i + 1) {
    cout << "Love";
}

```

Diagram labels for for-loop:

- initialization: `int i = 0;`
- condition: `i < 5;`
- updatation: `i = i + 1;`
- executing body: `cout << "Love";`

flow -



initialization
condition
updatation

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 $\longrightarrow$

```
outer loop() {  
    inner loop() {  
    }  
    cout << endl;  
}
```

$\xrightarrow{\text{for rows}}$
 $\xrightarrow{\text{for cols}}$

$\longrightarrow a \text{ op } = b \longrightarrow a = a \text{ op } b$

$\text{op} \longrightarrow +, -, *, \%, /, <<, >>$

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
 $\downarrow$
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
 $\downarrow$
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

various namespaces

std

other

iostream

cout
definition

iostream

cout
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

ternary operator -

W1-HW

↳ syntax

variable = (condition) ? expression2 : expression3

(condition)? variable = expression 2 : variable = expression 3

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

ex- `return (a > b) ? a : b ;` $\longrightarrow$ CORRECT

$(a > b) ? \text{return } a : \text{return } b;$ $\longrightarrow$ WRONG ERROR

these are statements, not expression

by default -

`wout << sizeof(2.3);` → 8
 |
 → float

`wout << sizeof(a);` —————→ 4 —————→ int
 |
 —————→ $-(2^{31}-1)$ to $2^{31}-1$

$\text{cout} << \text{sizeof}(-2^{31}) \longrightarrow 8$

$\quad\quad\quad \longmapsto \text{long long}$

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 -
 $0 \text{ to } < n-1$

$n-1$
-1
0
1
2
3

no. of times loop runs

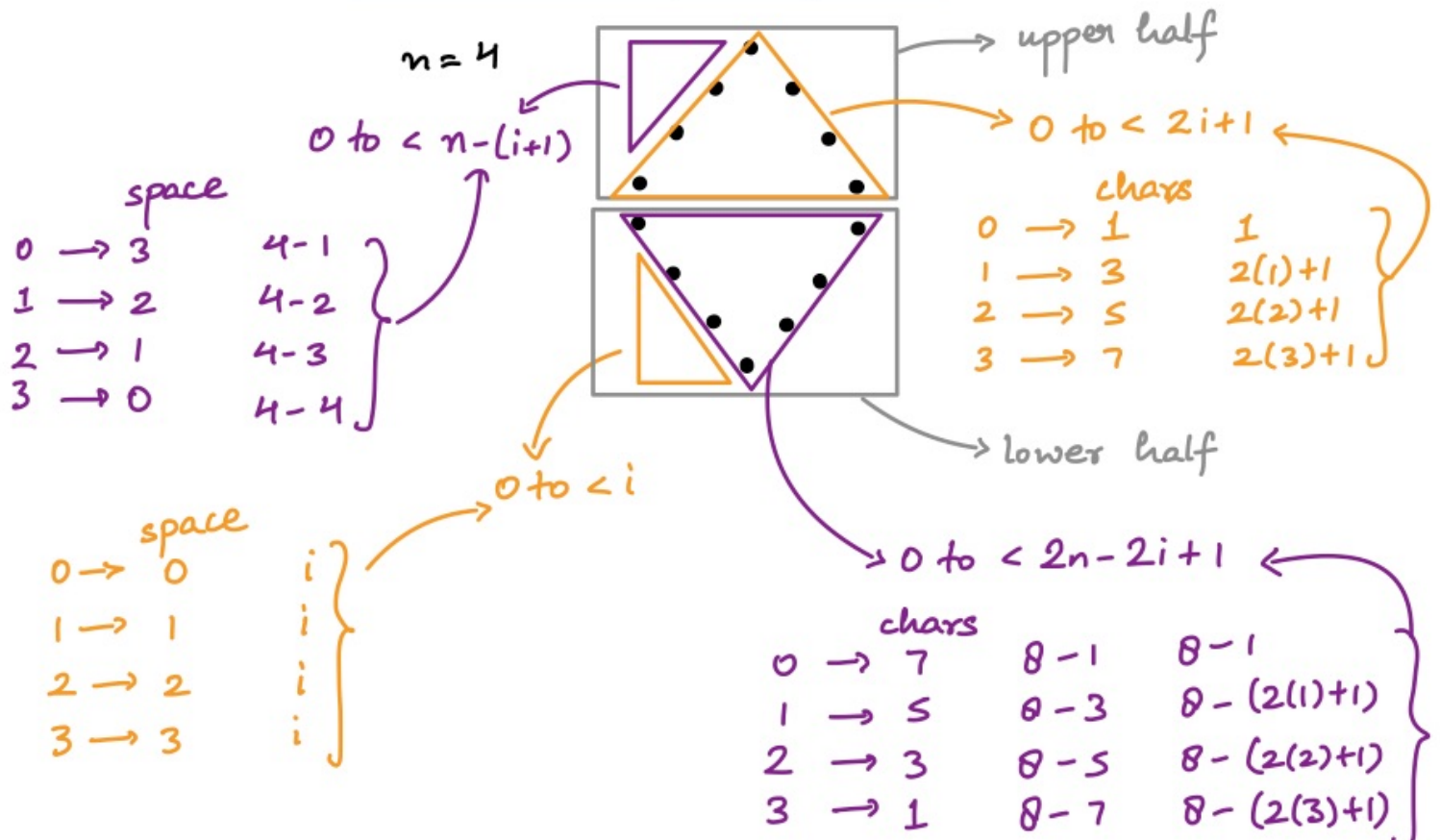
0
0
1
2
3

as condition fails ($0 < -1$)

→ to do anything n times

→ `for(i=0; i<n; i++){ }`

→ break the complex patterns



Bitwise Operators

W2-L2

↳ 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	$(\sim a)$	negate the result
Xor	$(a \wedge b)$	same values $\rightarrow 0$ diff. values $\rightarrow 1$

~ 5

↳ $5 \rightarrow 0 \dots 0101$

$\sim 5 \rightarrow 1 \dots 1010$

↳ how compiler read this

↳ 2's complement

$0 \dots 0101 \rightarrow 1's \text{ complement}$

$0 \dots 0110 \rightarrow 2's \text{ complement}$

-6

So $\sim 5 = -6$

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

$a = a \ll b$ a left shifts, b times $\rightarrow$ result $\rightarrow a \times 2^b$

$a = a \gg b$ a right shifts, b times $\rightarrow$ result $\rightarrow \frac{a}{2^b}$

b can't be -ve

$\hookrightarrow$ in case of -ve

$\hookrightarrow$ gives gv

$a = 5;$

$a = a \ll 1; \quad a = 10$

$a = 5;$

$a = a \ll 2; \quad a = 20$

in left shift $\rightarrow$ filled with 0

in right shift $\rightarrow$ filled with

0 or 1

in +ve no.

in -ve no.

right shift in -ve number

-ve no. in memory $\rightarrow 1 \dots$

$\downarrow$ right shift

$11 \dots$

$\hookrightarrow$ signed bit is used to fill the vacant bit

ex-

$5 \rightarrow 0 \dots 0101$

$-5 \rightarrow 1 \dots 1011$

$-5 \gg 1 \rightarrow 1 \dots 101 \rightarrow -3$

$-1 \gg 1 \rightarrow -1$

left shift in number where MSB is 1

and 2nd MSB is 0

no. $\rightarrow 10 \dots \rightarrow$ -ve no.

left shift $\hookrightarrow 0 \dots$

$\hookrightarrow$ +ve no.

Pre-Post → Increment / Decrement Operator

pre-increment

↳ ++a

↳ first increment by 1, then use

post-increment

↳ a++

↳ first use then increment by 1

pre-decrement

↳ --a

↳ first decrement by 1, then use

post-decrement

↳ a--

↳ first use then decrement by 1

```
int a = 5;
```

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

output -

49

↳ due to operator precedence

→ links.txt in repo

break and continue

break

↳ exit from that loop

continue

↳ skip that iteration

Variable Scoping -

```
int g = 25;           → global variable
int main() {
    int a;             → declaration
    int b = 5;          → initialization
    b = 10;             → updation
    // int b = 15;       → redefinition is not allowed
    int c = 7;
    g = 30;
    cout << g;           → 30
    if (true) {
        int b = 15;
        cout << b;       → 15
        cout << c;       → 7
        g = 50;
        cout << g;       → 50
    }
    cout << a;           → gv
    cout << b;           → 10
    cout << c;           → 7
    cout << g;           → 50
}
```

The diagram illustrates variable scoping and redefinition in C++. It shows a global variable `g` and a local variable `b` inside a `main` function. The code includes declarations, initializations, updates, and a redefinition of `b` inside an `if` block. Arrows indicate the scope of each variable: `g` is global, `a` is local to `main`, and `b` is local to the `if` block. The output of the code is shown next to the corresponding `cout` statements. A purple arrow points from the `if` block back to the `main` function, indicating the return path. A red arrow points from the `if` block to the `cout << g;` statement, indicating the scope of `g`.

Making global variable is very **BAD PRACTICE**

Operator Precedence

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

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

}

can also have
nested switch
case

not
mandatory

without break

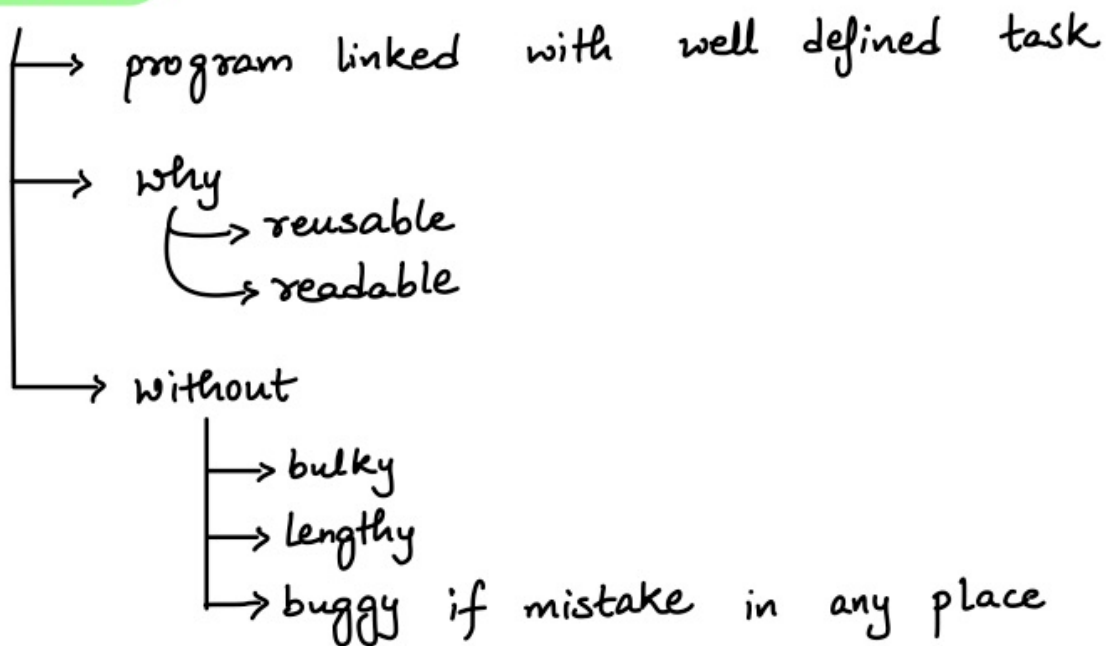
→ all below executing body will also execute

→ continue cannot be used in switch case

→ can only use in loops

Function -

W2-L3



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; → 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 $\leftrightarrow$ function invoke

Stack

↳ Last In First Out

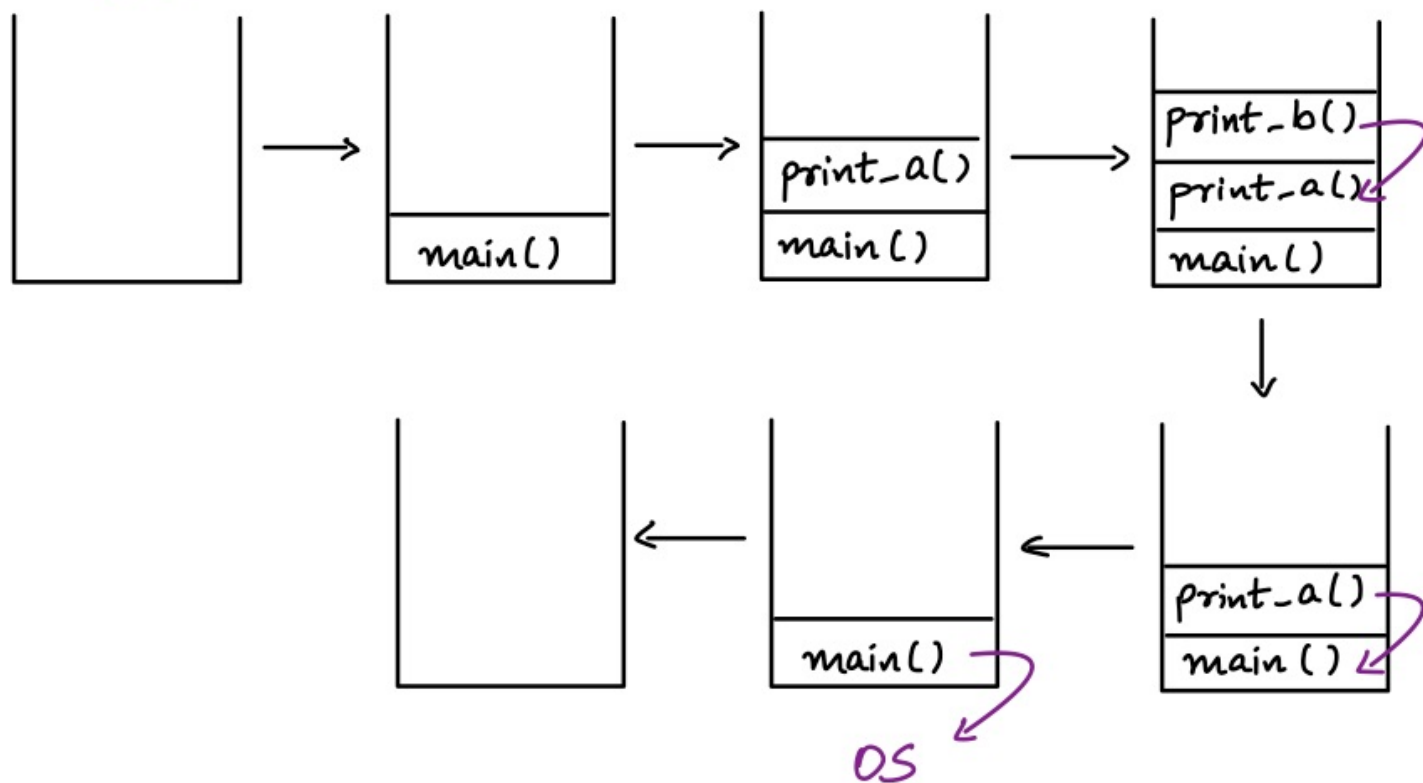
- tells
- what functions are active
 - which function calls which
 - local variables of function
 - return type of function

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;  
}
```

Output - 5 3

return to OS

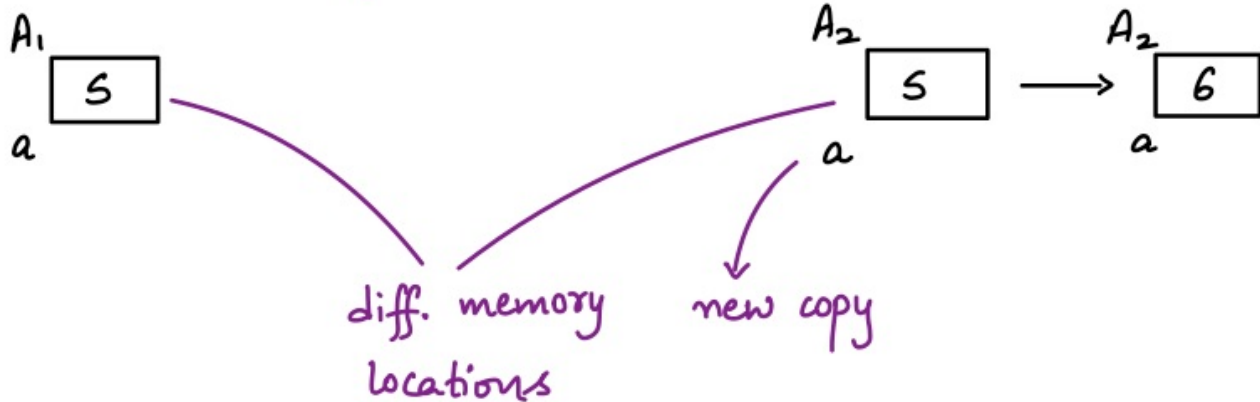


Pass by value → value is passed to another variable

↳ a copy will be created of variables

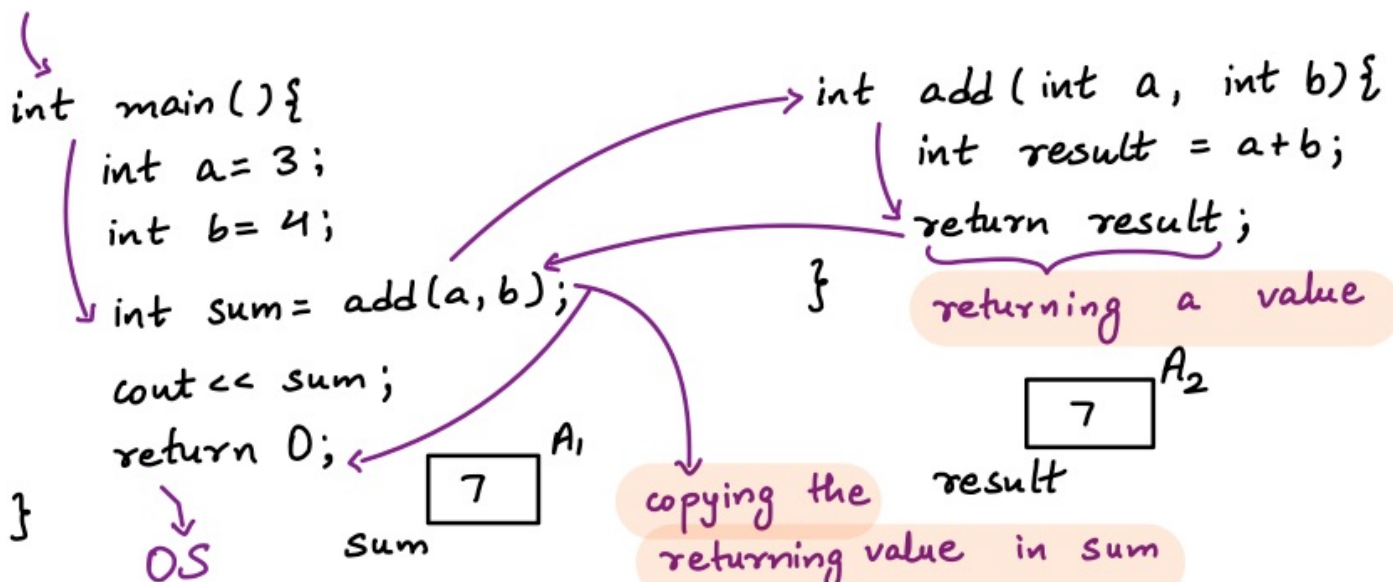
```
int main() {  
    int a = 5;  
    printNumber(a);  
    cout << a;  
}
```

```
void printNumber(int a) {  
    cout << a; → 5  
    a++;  
    cout << a; → 6  
}
```



Address Of Operator &

```
int n = 5;  
cout << &n; → output - address of n
```



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;  
}
```

function
definition

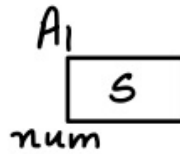
```
{ int add (int a, int b) {  
    return a+b;  
}
```

% operators → heavy operators

↳ so try to use it less

int num=5;

Symbol table



int num ↔ A₁

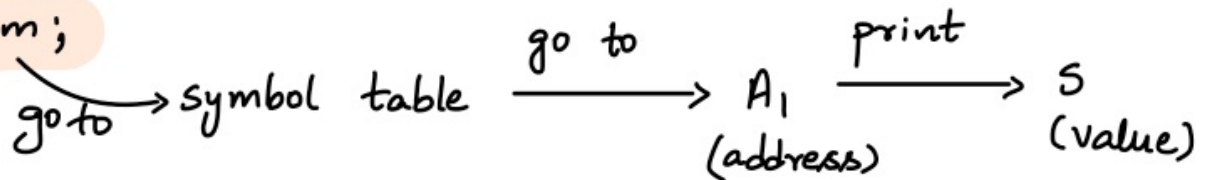
↓
datatype

variable name

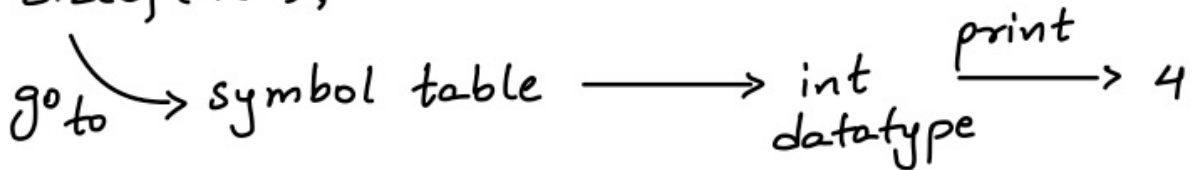
memory location of variable

variable address mapping is stored

cout << num;



cout << sizeof(num);



while loop

int i=0;

initialization

while (i < 5) {

condition

cout << i << endl;

i++

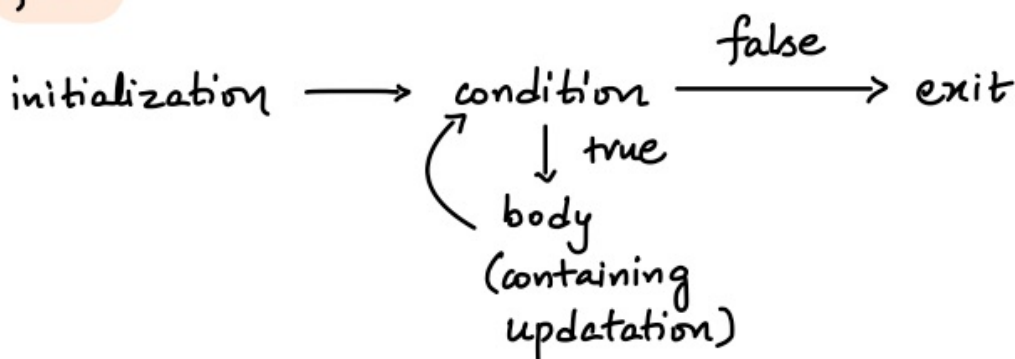
}

body

updatation

}

flow



left and right shift operators

int a = 2;

a << 1; $\longrightarrow$ no change

cout << a; $\longrightarrow$ 2

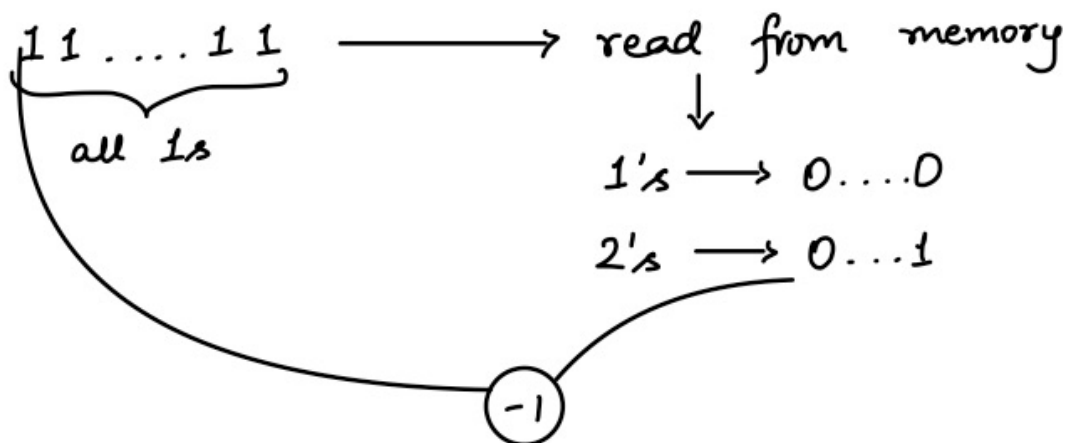
a = a << 1; $\longrightarrow$ change $\longrightarrow$ left shift by 1

cout << a; $\longrightarrow$ 4

right shift in -ve no.

$\longrightarrow$ link in links.txt in repo

How -1 is stored in memory -



$\sim a = -(a+1)$ and $\sim(\sim a) = a$

ex - a = 5; $\longrightarrow$ 0...0101

a = $\sim a$; $\longrightarrow$ 1...1010 $\longrightarrow$ -6 $\longrightarrow$ -(5+1)

$\xrightarrow{\text{read}} 2's \longrightarrow 0 \dots 0110$

a = -6;

$\xrightarrow{\text{read}} 1 \dots 1010$

a = $\sim a$; $\longrightarrow$ 0...0101

$\xrightarrow{\text{read}} 5 \longrightarrow -(-6+1)$

-6

Number System

W2-R

↳ 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 $\longrightarrow$ upper bound $\longrightarrow$ worst case

Theta $\Theta \longrightarrow$ average case

Omega $\Omega \longrightarrow$ lower bound $\longrightarrow$ best case

Big O Complexities

$O(1) \longrightarrow$ Constant time

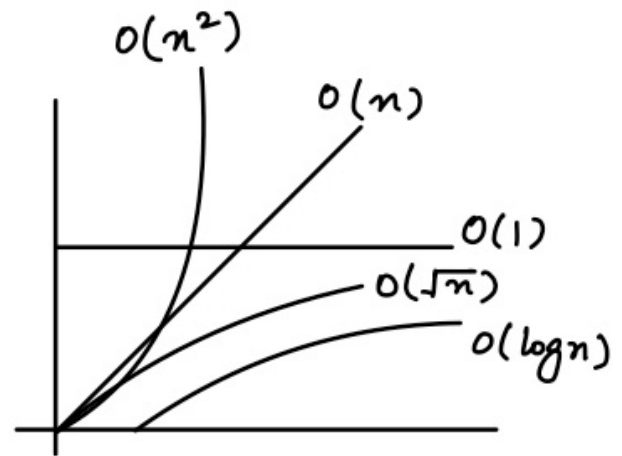
$O(n) \longrightarrow$ Linear time

$O(\log_2 n) \longrightarrow$ Logarithmic time

$O(n^2) \longrightarrow$ Quadratic time

$O(n^3) \longrightarrow$ Cubic time

nesting $\longrightarrow$ multiply $\longrightarrow$ $\{ \{ \}$
in sequence $\longrightarrow$ add $\}$
 $\longrightarrow$ $\{ \}$
 $\{ \}$



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

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

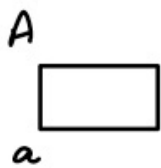
W3-L1

- 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;`



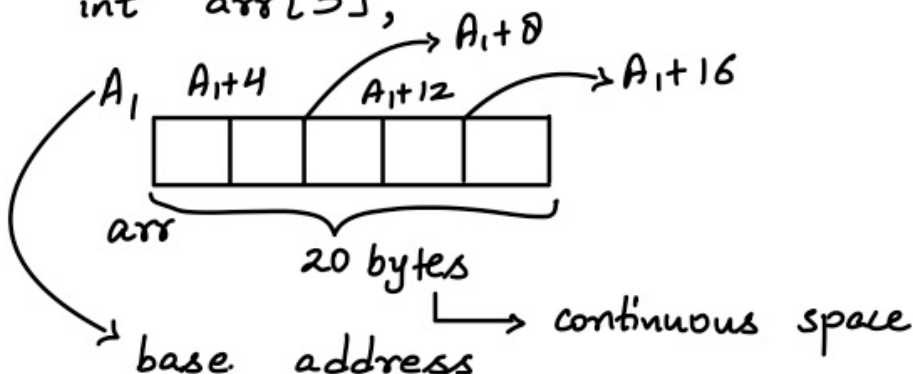
symbol table

`int a` ↔ A

`int arr` ↔ A₁

Declaration

`int arr[5];`



`cout << arr ;` $\longrightarrow$ A_1 $\longrightarrow$ address of first index of array
address of integer

`cout << &arr ;` $\longrightarrow$ A_1 $\longrightarrow$ 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 } ;` $\longrightarrow$ remaining 5 will be 0

`//int arr4 [4] = { 2, 4, 6, 8, 10 } ;` $\longrightarrow$ ERROR

`int arr5 [10] = { 0 } ;` $\longrightarrow$ initializing all values with 0

Making array at runtime

`int n ;`

`cin >> n ;`

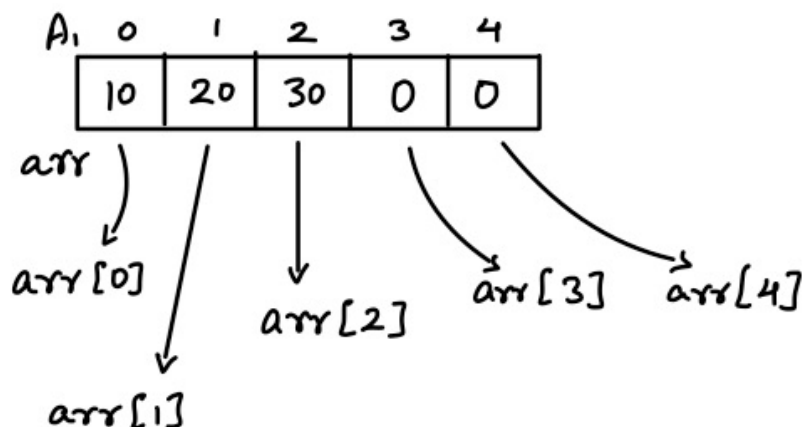
`int arr [n] ;` $\longrightarrow$ BAD PRACTICE

Index and Access in memory

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

0^{th} based indexing

$\hookrightarrow 0$ to $n-1$



$arr[i] \longrightarrow \text{value at address } [A_1 + (i * 4)]$

that's why 0 based indexing

BA

index

due to int
(datatype size)

taking input in array

due to internal working

$\hookrightarrow cin \gg arr[i];$

Arrays and Function

$\hookrightarrow func (int arr[], int size) \{$
 $\}$

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

array is passed by reference
because pointer is passed by
value

updatation in actual array

always pass size alongwith arr

```
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;  
}
```

A_1 will learn ahead

A_1
arr

5	6
---	---

`sizeof(int);` $\longrightarrow$ 4 $\longrightarrow$ in bytes

`int arr[5];`

`sizeof(arr);` $\longrightarrow$ 20 $\longrightarrow$ in bytes
 $\longrightarrow$ not 0 (size of pointer)

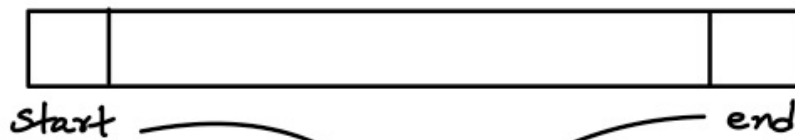
linear search in array

INT_MIN and INT_MAX

$\longrightarrow$ to find max., start ans with INT_MIN

$\longrightarrow$ 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);`
 $\longrightarrow$ datatype

Vector

W3-L2

- 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);`
 size 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();` $\longrightarrow$ true if empty

Capacity -

`arr.capacity();` $\longrightarrow$ * by 2 if arr gets fully filled and a new element is inserted

$\longrightarrow$ no. of elements it can store

$\longrightarrow$ in initialization, capacity = size in all methods of initialization

`sizeof(arr);` $\longrightarrow$ compiler dependent initially

`cout << arr;` $\longrightarrow$ give ERROR

$\rightarrow$ Xor $\longrightarrow$ Cancels out same element

$$0 \wedge \text{ans} = \text{ans} \begin{cases} \rightarrow 0 \wedge 1 = 1 \\ \rightarrow 0 \wedge 0 = 0 \end{cases}$$

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];` → $m \times n$ elements
→ cols → 0 to $n-1$
→ rows → 0 to $m-1$

`int arr[2][3];`

visualize -

			cols
0	10	20	30
1	40	50	60
rows	0	1	2

mapping

in memory -

0	1	2	3	4	5
10	20	30	40	50	60
row 1			row 2		

Access -

`arr[i][j];`
→ col index $0 \leq j < n$
→ row index $0 \leq i < m$

Mapping -

$$\text{linear_index} = c * i + j$$

no. of cols

row index

col index

$$i = \text{linear_index} / c$$

$$j = \text{linear_index} \% c$$

Initialization-

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

same

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

visualize -

		→ cols		
	0	10	20	30
	1	40	50	0
	2	0	0	0
rows		0	1	2

mapping

in memory -

0	1	2	3	4	5	6	7	8
10	20	30	40	50	0	0	0	0
row 1			row 2			row 3		

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 don't 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
}

```

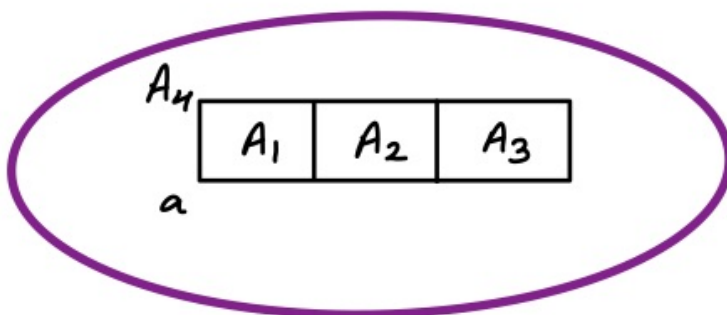
```

int main() {
    int arr[2][3] = {10, 20, 30, 40, 50};
    func(arr, 2, 3);
    return 0;
}

```

arr

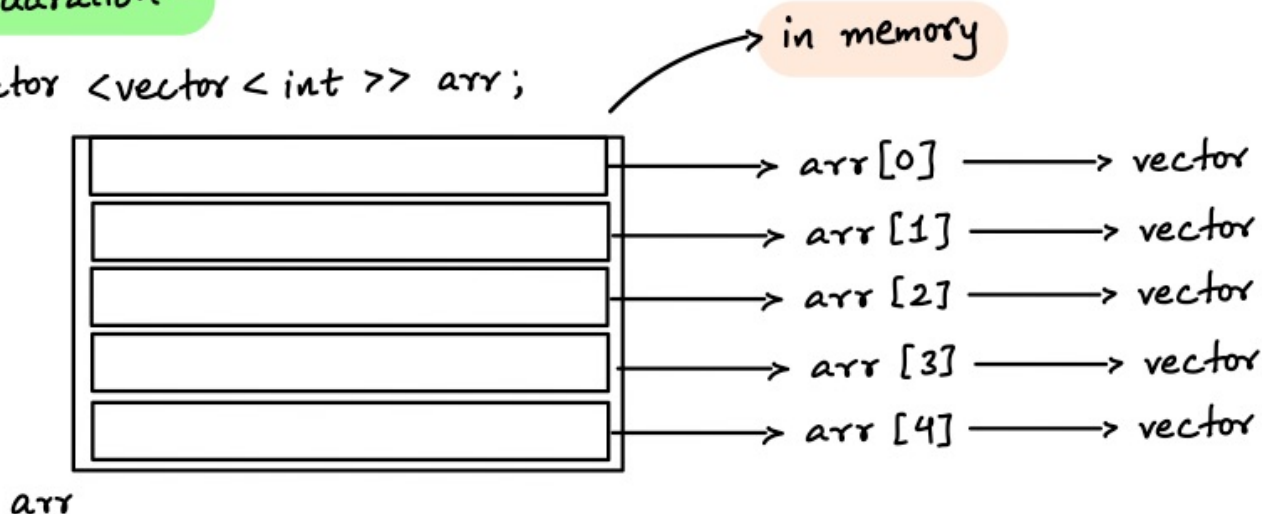
A ₁	10	20	30
A ₂	40	50	0
A ₃	0	0	0



2D Vector

Dedaration -

`vector <vector <int >> arr;`



Dedaration -

`vector <vector <int >> arr;`

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

→ no. of rows

Number of rows

`arr.size()`

Number of cols

`arr[i].size()`

└──→ in *i*th row
└──→ size of *i*th row

Initialization

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

rows → no. of rows in arr

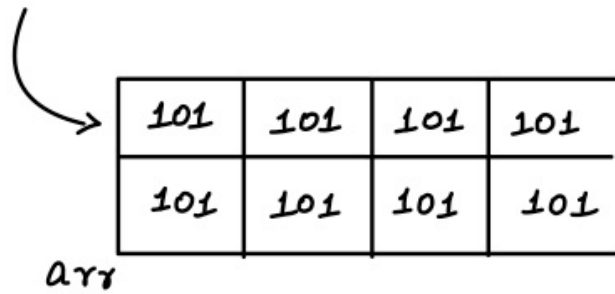
cols → no. of cols in arr
└──→ size of 1D vector

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

initialization of 1D
vectors in arr

size initial value in 2D
 vector

```
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

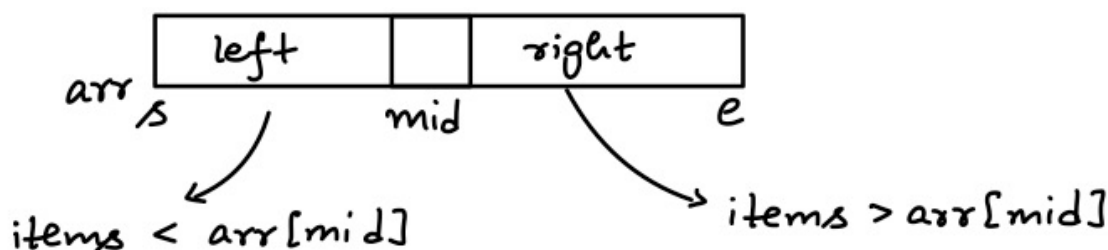
```
int linearSearch ( vector <int> arr, int target ) {  
    int n = arr.size();  
    for ( int i=0; i<n ; i++) {  
        if ( target == arr[i] )  
            return i;  
    }  
    return -1;  
}
```

T.C - $O(n)$

S.C. - $O(1)$

Binary Search -

condition $\rightarrow$ sorted order $\rightarrow$ monotonic function
binary $\rightarrow$ 2 $\rightarrow$ 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;  $\longrightarrow$  search in left subarray
        else
            s = mid + 1;  $\longrightarrow$  search in right subarray
        mid = s + (e-s)/2;
    }
    return -1;
}

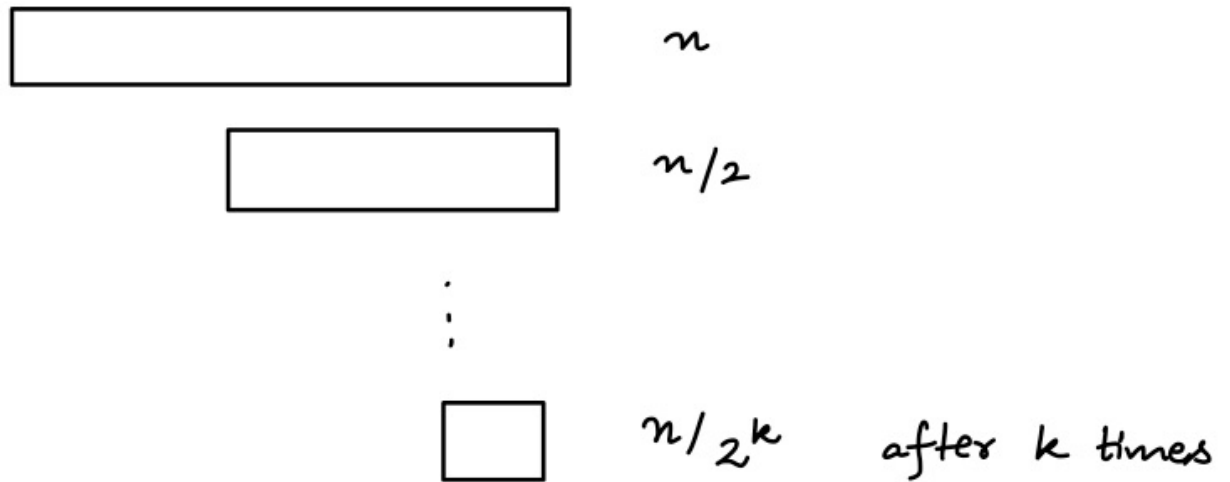
```

issue in $mid = \frac{s+e}{2}$; $\longrightarrow$ int overflow
 if $s+e > INT_MAX$
 $\longrightarrow$ 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

$$\text{T.C.} = O(\log_2 n)$$

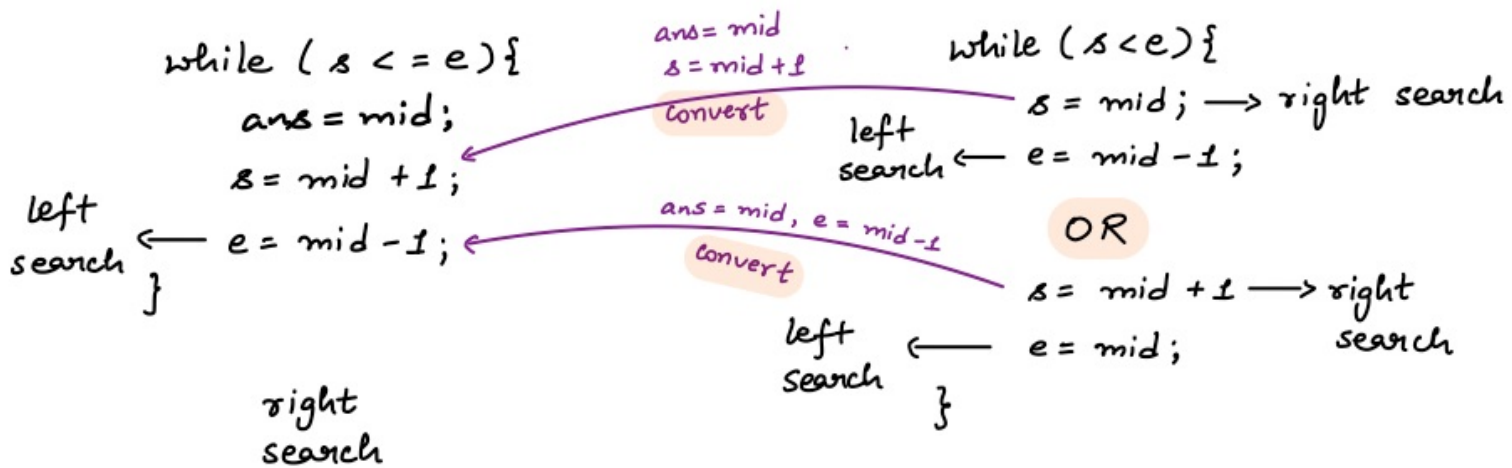
W4-L2

→ Search Space

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

→ store mid in ans if needed

→ In binary search questions



`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

can also find array is sorted or not

← pivot in sorted rotated array

search in sorted rotated array

└─> pivot index = $n-1$

W4-L3

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

└→ predicate function

└→ sqrt of a no.

divide 2 numbers

logic to decide
either left or
right

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
add appearing element in array

Sorting

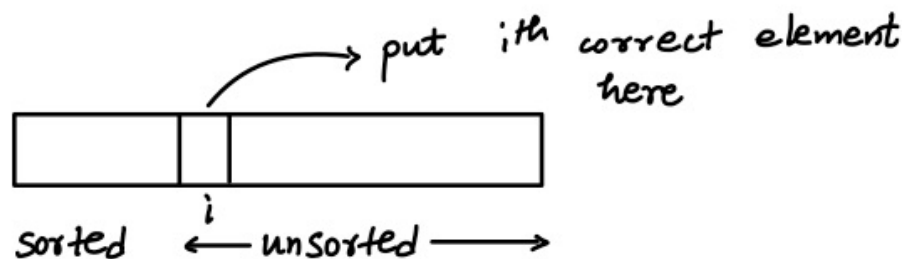
└→ putting all elements either in
increasing order or decreasing order

W4-L5

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 comparisons

```
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. - $O(n \log n)$

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 - Bubble 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];

gv	gv	gv	gv	gv
----	----	----	----	----

 ch

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

'U'	'j'	'j'	'w'	'a'	'l'	gv
-----	-----	-----	-----	-----	-----	----

 ch

char ch[] = "Ujjwal";

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

 ch

Taking input in char array

char ch[7];

cin >> ch[i];

cin >> ch; → by default 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 delimiter

char ch[10];

allocate as ch[10] = "Ujjwal"

cin >> ch; → Ujjwal

store in buffer memory

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

buffer

copy in memory of ch

'U'	'j'	'j'	'w'	'a'	'l'	'\0'	gv	gv	gv	
ch	0	1	2	3	4	5	6	7	8	9

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, delimiter);`

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

```
func (char ch[]) {  
  
}
```

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

Size of char array → $\frac{\text{sizeof}(ch)}{\text{sizeof}(\text{char})} - 1$

↳ 1

Some inbuilt functions of char array

`strlen(ch);`

`strcmp(ch1, ch2);`

`strcpy(ch1, ch2);`

Strings

- Datatype
Not Data Structure
- Dynamic char array
- NULL char at last of string

`string str;` → empty string created

`cin >> str;` → Ujjwal

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

`cout << str;` → Ujjwal str

`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
```

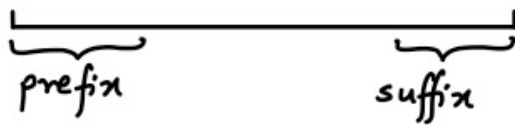
```
    return 0;
```

```
}
```

can be any function
according to need

ascending order

Prefix and Suffix



Hash Map $\longrightarrow$ WILL LEARN LATER

$\longrightarrow$ Data structure
 $\longrightarrow$ data stored in key-value pair

Initialization

`map < key datatype, value datatype > map-name;`

`map < int, char > m;` $\longrightarrow$ ordered map

`m[0] = 'a';`

`m[1] = 'b';`

`m[25] = 'z';`

`cout << m[0];` $\longrightarrow$ 'a'

`cout << m[25];` $\longrightarrow$ 'z'

`cout << m[20];` $\longrightarrow$ " $\longrightarrow$ NULL char

`cout << (int) m[20];` $\longrightarrow$ 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; A_1 5
 a

int b = 5; A_2 5
 b

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
 ↑
 hexadecimal

cout << &b; → A_2

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

int * p1 = &a;

Annotations for `int * p1 = &a;`:

- `int`: datatype
- `*`: dereference operator
- `p1`: pointer name
- `=`: address of operator
- `&a`: variable to which pointer points
- `int *`: symbol for pointer creation

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

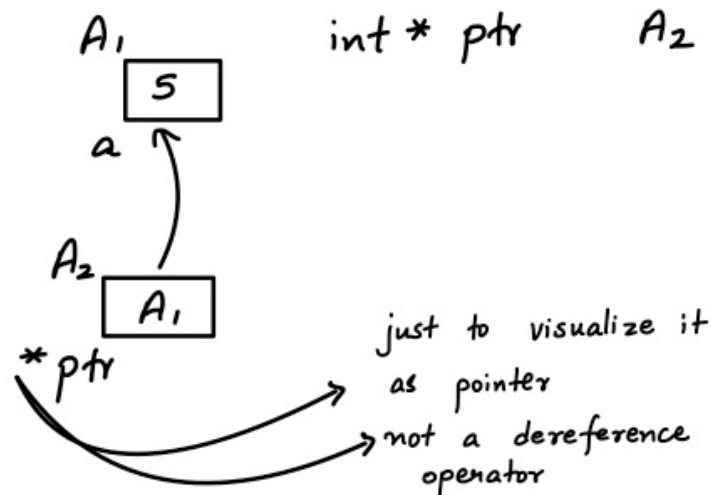
cout << *ptr; → 5

cout << &ptr; → A₂

symbol table

int a A₁

int * ptr A₂



*ptr → value at (address stored in ptr)
→ dereference operator

&ptr → address of ptr

size of pointer

sizeof(ptr); → 8
64 bit architecture

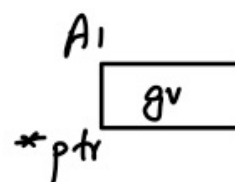
depends on
architecture
compiler implementation
memory organization

Always
↓
it stores address,
datatype does not matters

Use case of pointer

- dynamic memory allocation
- memory management
- 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
cout << ptr; → gv
→ Segmentation fault



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

A_1 0 $*ptr$

A_2 0 $*ptr2$

A_3 0 $*ptr3$

cout << ptr; → 0

cout << *ptr; → ERROR

cout << &ptr; → A_1

Segmentation fault

Arithmetic Operations In Pointers

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

A_1 ~~5~~ 6 7

a

A_2 ~~A₁~~ A₁+4 $*ptr$

A_1 to A_1+3 has already been taken by integer a, so next address will be A_1+4

```
int a = 10;
```

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

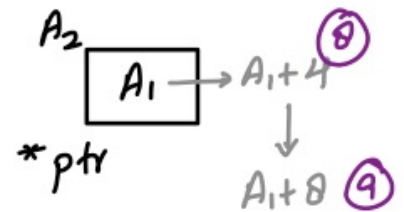
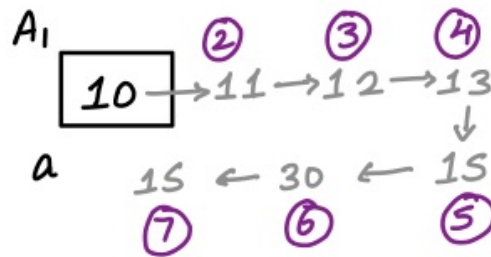
a 10

&a A₁

ptr A₁

*ptr 10

&ptr A₂



- 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

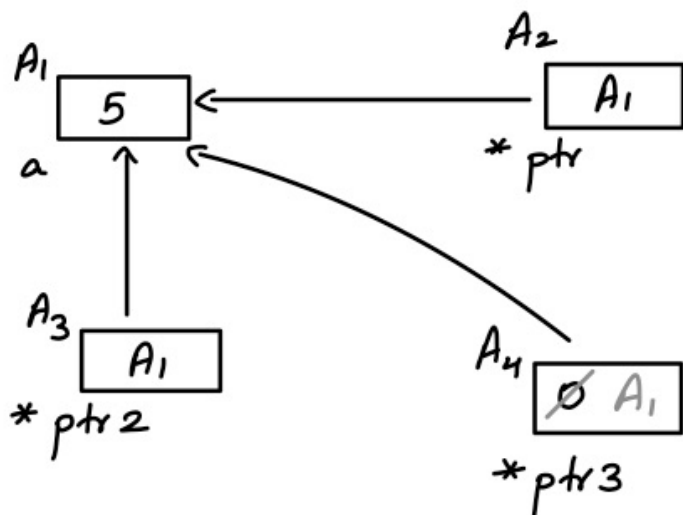
```
int a = 5;
```

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

```
{ int *ptr2 = ptr;
```

```
{ int *ptr3 = 0;
```

```
{ ptr3 = ptr;
```



ARRAYS & POINTERS

W6-L2

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

CONSTANT pointer to the first element of array

cannot change

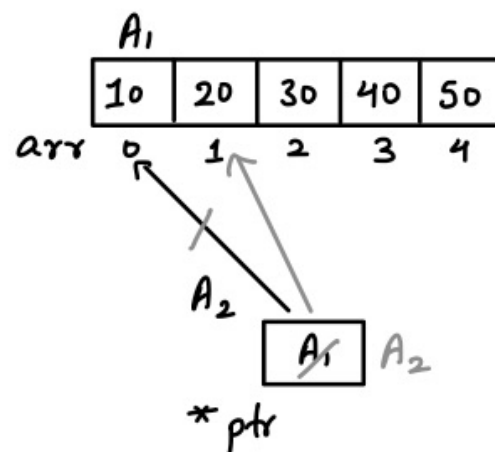
also does not have separate memory

$arr[i] \rightarrow$ element at index i

$\&arr[i] \rightarrow$ address of $arr[i]$

$\text{cout} \ll arr[0]; \rightarrow 10$

$\text{cout} \ll \&arr[0]; \rightarrow A_1$



$\text{cout} \ll arr; \rightarrow A_1$

array name returns add. of 1st index of array

$\text{cout} \ll \&arr; \rightarrow A_1$

from symbol table (add. of array)

same address unlike pointers

```
int *ptr = arr;
```

$\text{cout} \ll *ptr; \rightarrow 10$

$\text{cout} \ll ptr; \rightarrow A_1$

$\text{cout} \ll \&ptr; \rightarrow A_2$

$\text{cout} \ll *arr; \rightarrow 10$

$\text{cout} \ll *(\&arr); \rightarrow 10$

$\text{cout} \ll *(\&arr[0]); \rightarrow 10$

Symbol table

type	name	address
$\text{int} (*)[5]$	arr	A_1
$\text{int} *$	ptr	A_2

cout << *arr + 1; $\longrightarrow$ 11

cout << *(arr + 1); $\longrightarrow$ 20

cout << *(arr + 2); $\longrightarrow$ 30

cout << arr[2]; $\longrightarrow$ 30

cout << 2[arr]; $\longrightarrow$ 30

arr[i] $\longleftrightarrow$ *(arr + i) $\longleftrightarrow$ i[arr]

arr++; $\longrightarrow$ ERROR $\longrightarrow$ Entry in symbol table

ptr++; $\longrightarrow$ A₁+4 $\longrightarrow$ can't be change

$\longrightarrow$ You can access subpart of an array using pointers

int* ptr = arr;

cout << *(ptr + 2); $\longrightarrow$ 40

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

Arrays / Array pointers

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

① cout << arr; $\longrightarrow$ A₁ } same
cout << &arr; $\longrightarrow$ A₁

② arr++; $\longrightarrow$ ERROR

③ sizeof(arr); $\longrightarrow$ 3*4 = 12

Pointers / Normal pointers

int* ptr = arr;

① cout << ptr; $\longrightarrow$ A₁ } diff.
cout << &ptr $\longrightarrow$ A₄

② ptr++; $\longrightarrow$ VALID

③ sizeof(ptr) $\longrightarrow$ 8
 $\downarrow$
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

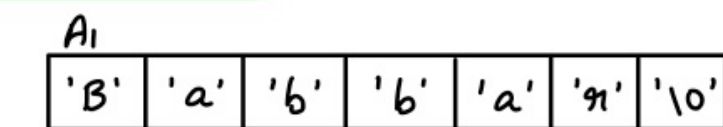
cout << &ch; —————> A₁

cout << ch[0]; —————> B

cout << &ch[0]; —————> Babbar

cout << *ptr; —————> B

cout << &ptr; —————> A₂



cout implementation
is diff. in char
pointers and char arrays

$ch[i] \longleftrightarrow *(ch + i) \longleftrightarrow i[ch]$

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

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

```
char *ptr = ch;
```

```
cout << ch;      _____> Sherbano
```

```
cout << *ch;     _____> S
```

```
cout << &ch;     _____> A1
```

```
cout << *(ch + 3); _____> n
```

```
cout << ptr;      _____> Sherbano
```

```
cout << &ptr;     _____> A2
```

```
cout << *(ptr + 3); _____> n
```

```
cout << ptr + 2;  _____> erbano
```

```
cout << *ptr;     _____> S
```

```
cout << ptr + 8;  _____> " _____> NULL char
```

```
cout << ptr + 9;  _____> g^
```

cout << ch[0]; $\longrightarrow$ S

cout << ch[0]; $\longrightarrow$ Sherbano

cont << &(*ch); $\longrightarrow$ Sherbano

CHAR AND POINTER

```
char ch = 'k';
```

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

$$\text{count} \leftarrow \text{ch}; \quad \longrightarrow \quad k$$

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

cont << ptr; $\longrightarrow$ k.....

`cout << &ptr;` $\longrightarrow$ A_2 $\begin{cases} \longrightarrow gv \\ \longrightarrow \text{print until it gets '\n'} \end{cases}$

cout << *ptr; $\longrightarrow$ k

Behind The Scenes

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

↳ 2 step process

①

'B'	'a'	'b'	'b'	'a'	'n'	'\0'
-----	-----	-----	-----	-----	-----	------

temp memory

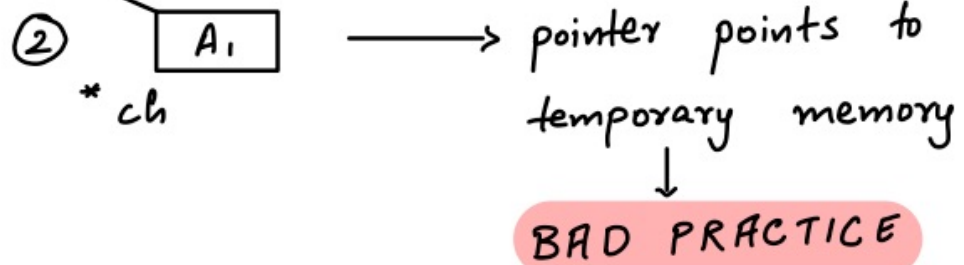
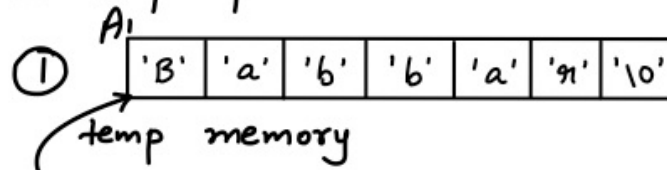
copy

② 'B' 'a' 'b' 'b' 'a' 'n' 'lo' g^v g^v g^v

ch

char * ch = "Babbar"

↳ 2 step process



POINTERS WITH FUNCTIONS

→ pass by value

a copy of pointer is made

→ but simulates pass by reference

↳ as in case of arrays & functions

int func (int arr[]) {

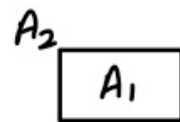
cout << arr; → A₁

cout << *arr; → 10

cout << &arr; → A₂

cout << sizeof(arr); → 8

}



*arr

copy of pointer
is created

```
int func2 ( int *arr ) {
```

→ pointer is passed

cout << arr; → A₁

cout << *arr; → 10

cout << &arr; → A₃

cout << sizeof(arr); → 8

A₃

A ₁

*arr

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(arr 2);
```

```
return 0;
```

```
}
```

A₁

10	20	0	0	0
----	----	---	---	---

arr

→ 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


```

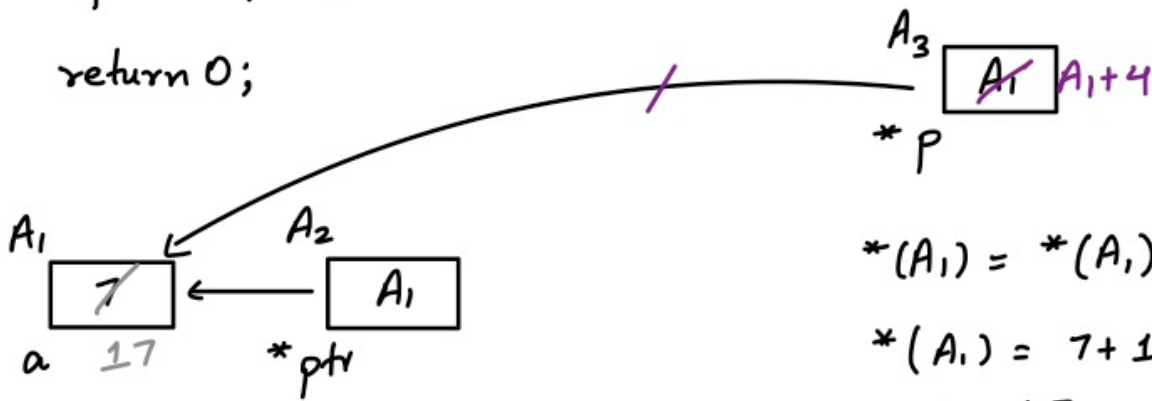
int main() {
    int a = 7;
    int * ptr = &a;
    update(ptr);
    return 0;
}

```

```

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

```



$*(A_1) = *(A_1) + 10$
 $*(A_1) = 7 + 10 = 17$
 $a = 17$

Function to Pointers in MW

Basic Mathematics for DSA

W6-L3 R

Sieve of Eratosthenes Theorem

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

Steps-

- 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 $\sqrt{n}$ if they are marked prime
- Count all remaining 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 $(i-1)$

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

$$SC - O(n)$$

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

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

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

Same with segmented sieve → having high & low

GCD/HCF and LCM

→ Greatest Common Divisor

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

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

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

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

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

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

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

Modulo Arithmetic

$$\rightarrow a \% n \rightarrow [0, n)$$

$$\rightarrow (a + b) \% n = (a \% n) + (b \% n)$$

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

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

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

Fast Exponentiation

$$\rightarrow a^b = a^{b/2} * a^{b/2}, \quad b \text{ is even}$$

$$a^b = [a^{b/2} * a^{b/2}] * a, \quad b \text{ is odd}$$

```

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;

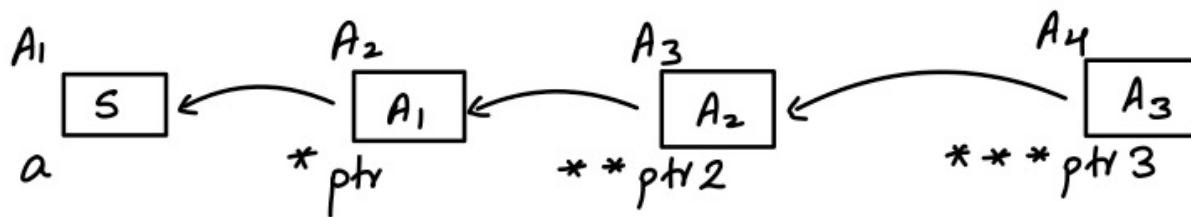
int *ptr = &a;

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;
}

```

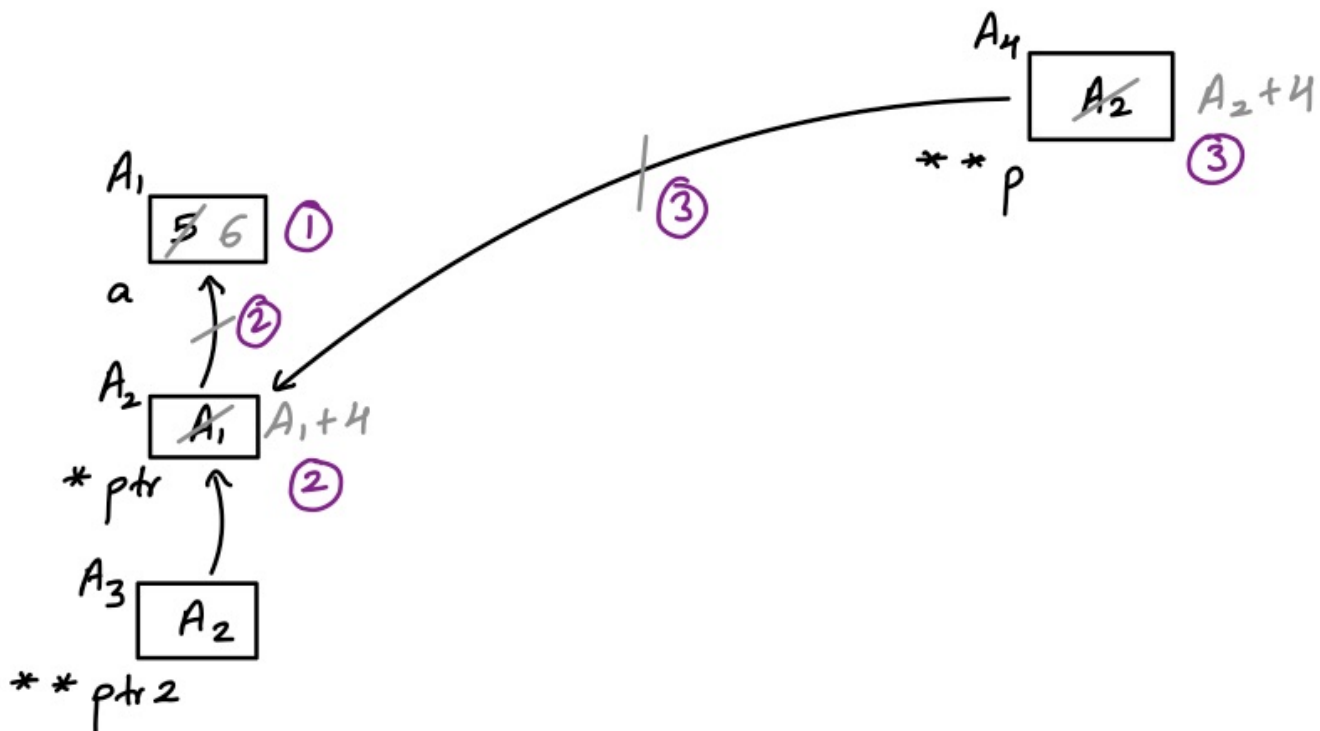
both are same

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
 - $a, *ptr, **ptr \rightarrow a$
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;

A_1

5

 a, b

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;  
}
```

```
void update (int &x) {
```

① $x++$;

}

PASS BY
REFERENCE

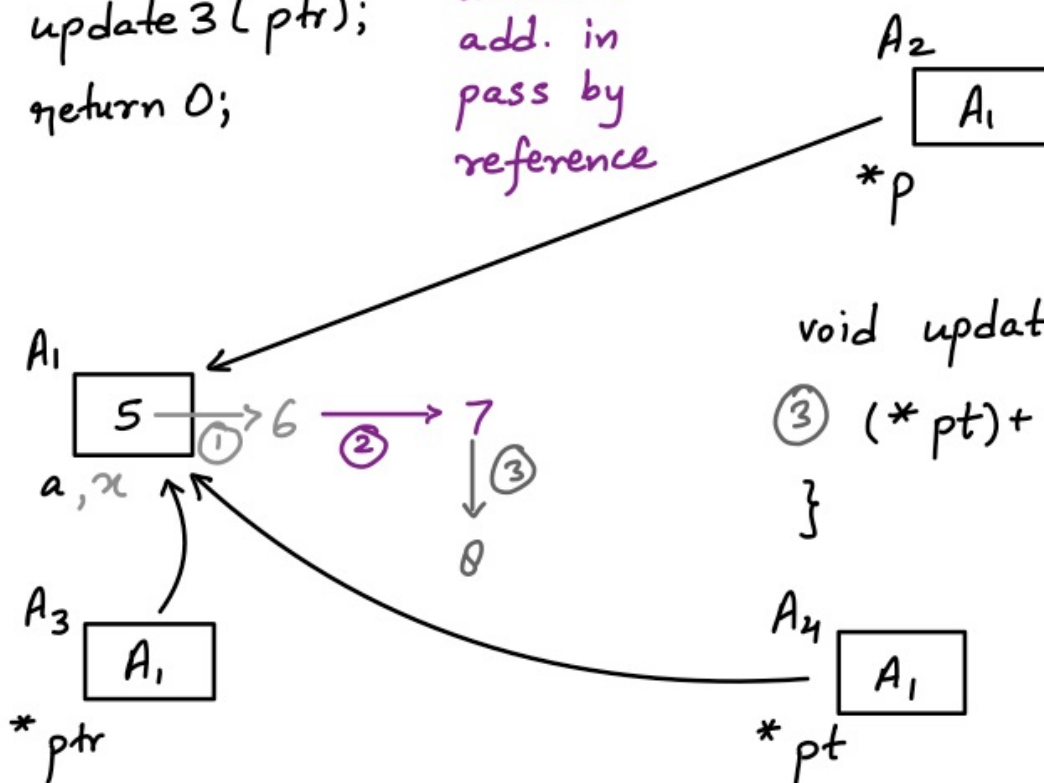
```
void update2 (int *p) {
```

② $(*p)++$;

}

PASS BY
VALUE

//update3 (&a); → ERROR
constant
add. in
pass by
reference



```
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;
}
```

→ this is reference variable
of type int

```
int solve (int &arr[])
```

```
int main () {
    vector<int> arr { 1, 2, 3, 4 };
    vector<int> ans = func(arr);
```

→ ERROR
array of reference
is not allowed

```
    for (auto i: ans)
        cout << i;           → 2 3 4 5
```

```
solve( arr ); → ERROR
```

vector != int

```
int arr2 [] = { 1, 2, 3 };
```

```
solve( arr2 ); → ERROR
```

int[3] != int

```
return 0;
```

```
}
```

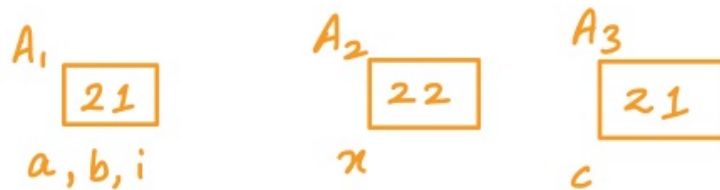
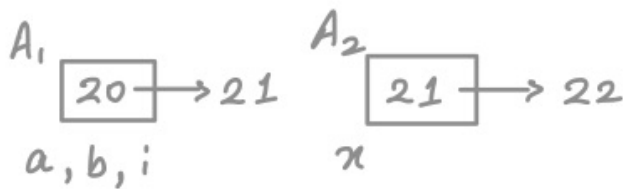
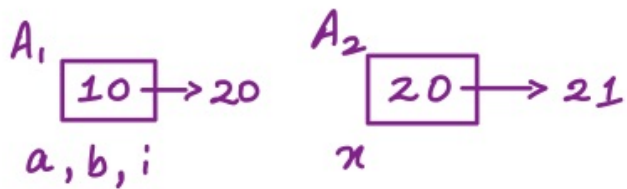
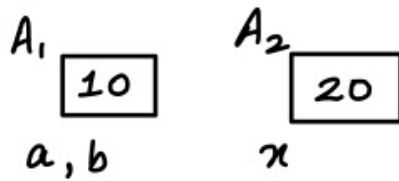
RETURN BY REFERENCE

- 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

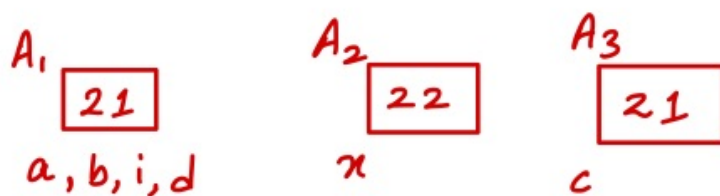
passing a reference variable

```
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;  
}
```

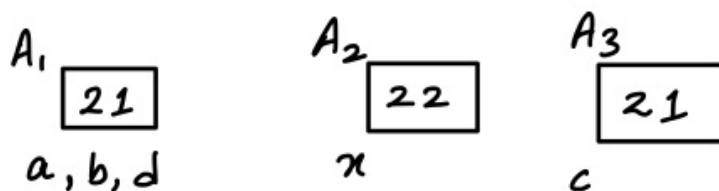
```
int &ref(int &i) {  
    return i;  
}
```



`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;
```

$x = 10;$

```
  ref2() = 20; —→ ERROR
```

```
  return 0;
```

```
}
```

```
int & ref() {
```

```
  return x;
```

```
int ref2() {
```

```
  return x;
```

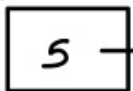
```
}
```

→ returning a value, not variable

$10 = 20;$

→ that's why ERROR

A_1



x

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;
```

```
}
```

returning a variable stored
in temporary memory

BAD PRACTICE

ERROR

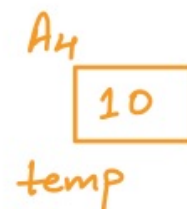
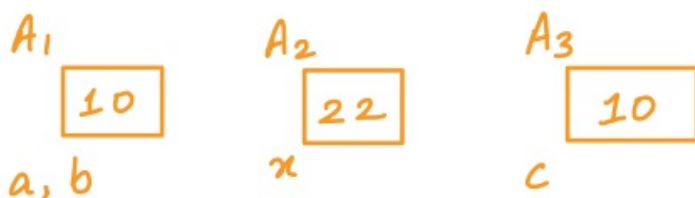
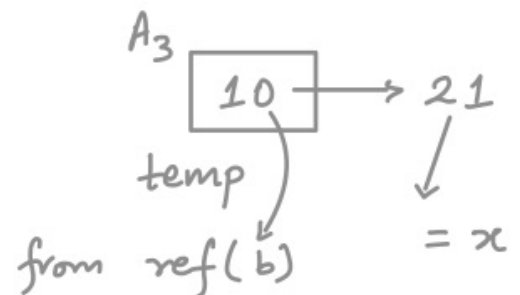
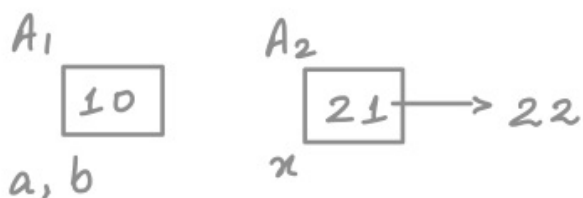
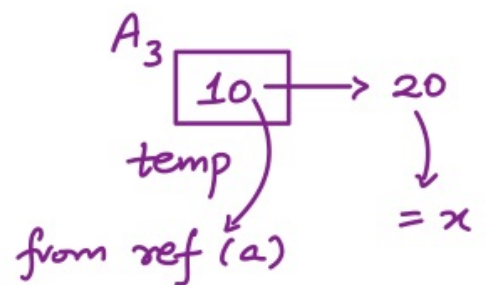
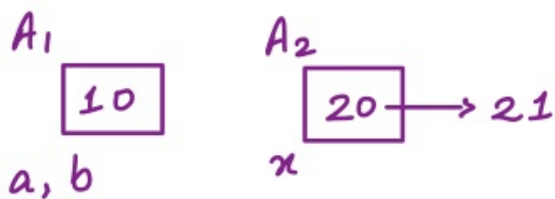
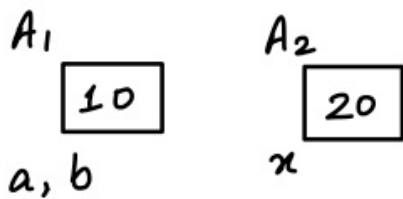
```
int c = ref(a);
```

```
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

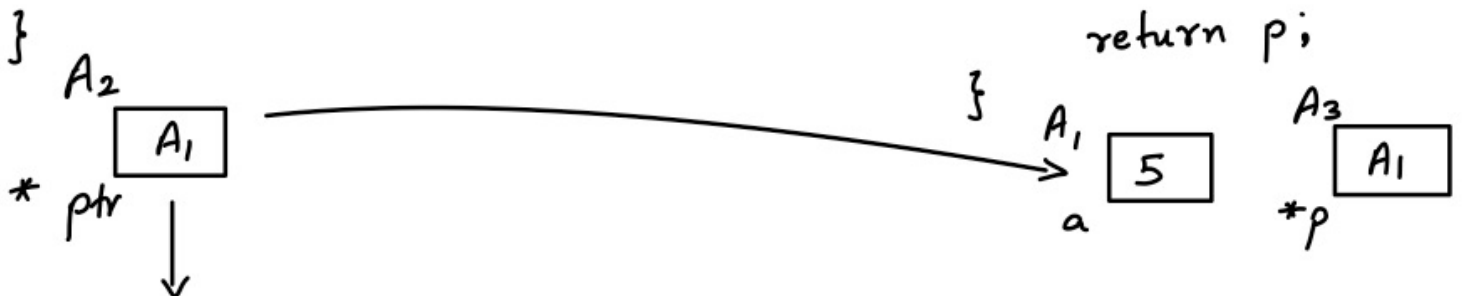
DON'T DO THIS AT HOME

Have similarity

← 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;` → runtime ERROR
`cout << *ptr;`

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

→ `int a[] = {1, 2, 3, 4, 5}`

`int *p = a++;` → ERROR

`cout << *p;`

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

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

`ch++;` → ERROR
`cout << ch;`

name of array refers
to base address

→ `double arr[] = {2.5, 7.9, 50.25};`

`double * ptr = arr;`

`ptr = ptr + 0.25;` → ERROR

`cout << ptr;`

↳ pointer arithmetic op-
can only be done
using integers

→ `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;`

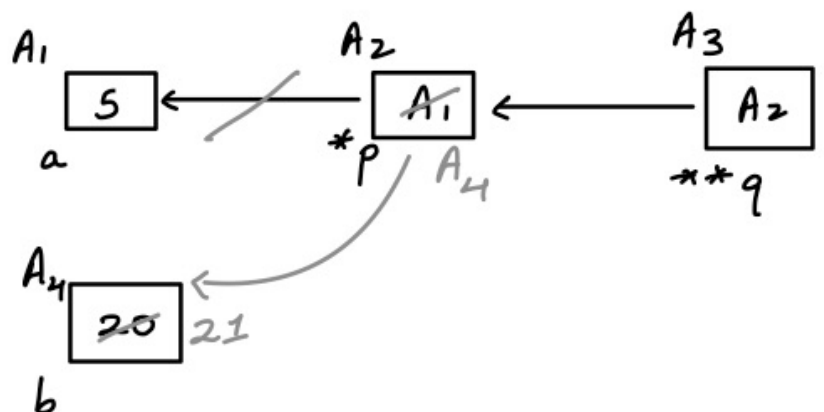
`int ** q = &p;`

`int b = 20;`

`*q = &b;`

`(*p)++;`

`cout << a << ' ' << b;` → 5 21



→ `int a = 5;`

`int * ptr = &a;`

`int ** ptr2 = &ptr;`

`int ** ptr3 = &(&a);` → ERROR

`cout << ** ptr3;`

↓
address of (address in
memory location)

(doesn't make any sense)

POINTER IMP DOUBT

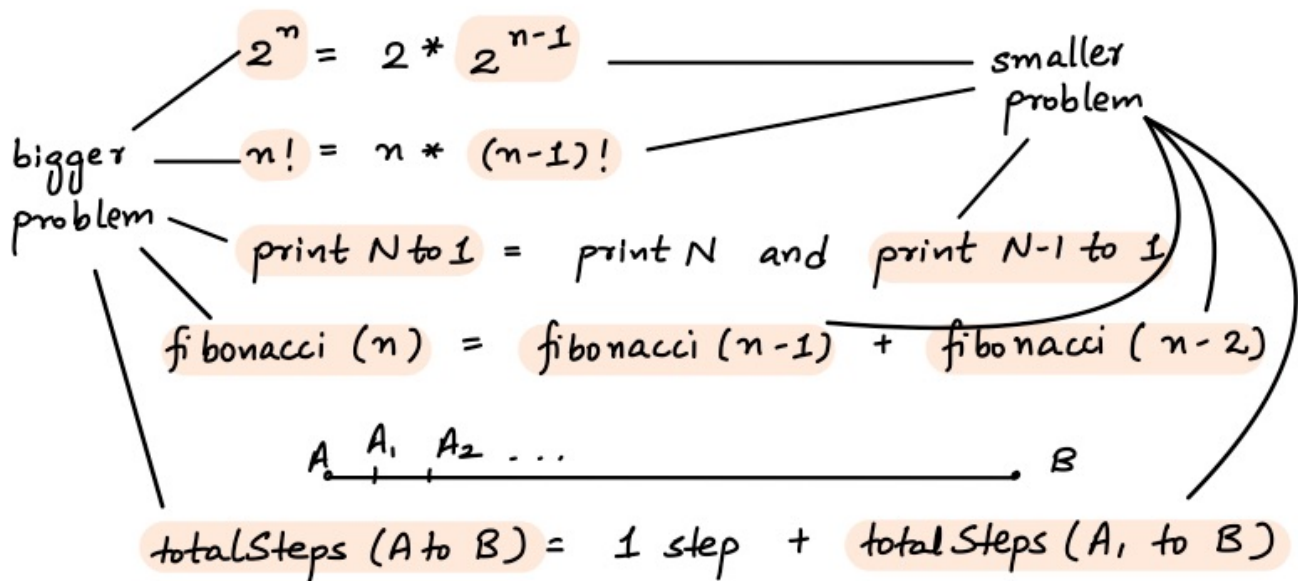
(IN DASH BOARD)

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;
}

```

```

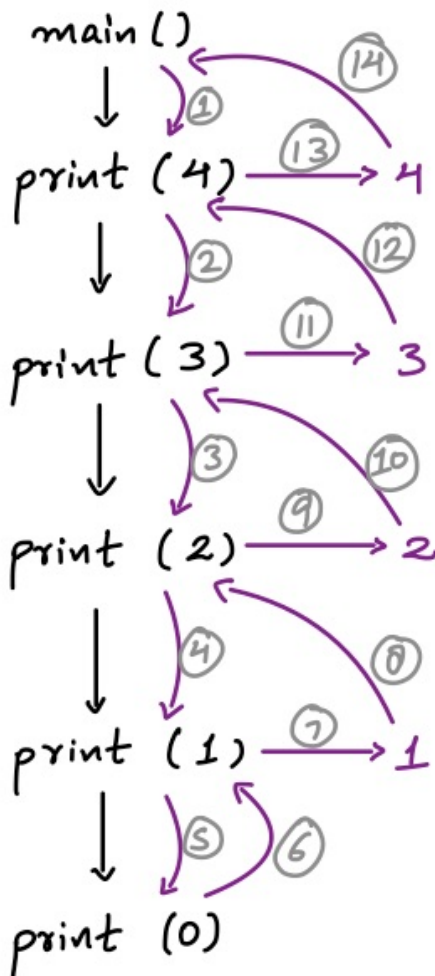
void print ( int n) {
    // base condition
    if ( n == 0)
        return;

    // recurrence relation
    print (n-1);

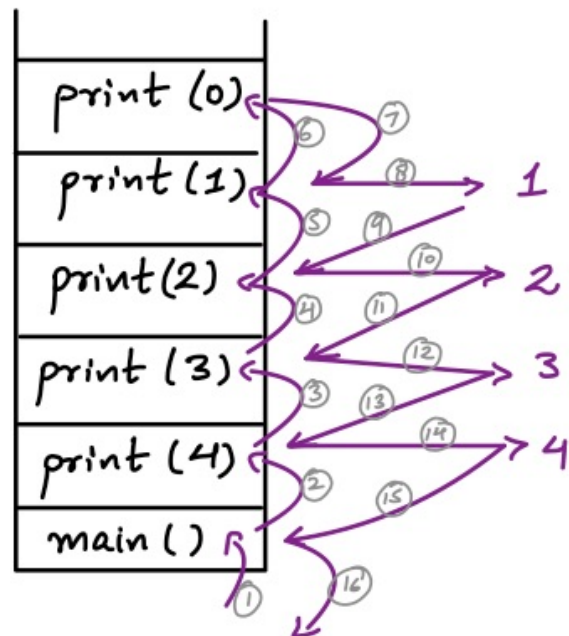
    // processing
    cout << n;
}

```

Recursion Tree



Call Stack



Output - 1 2 3 4

```

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

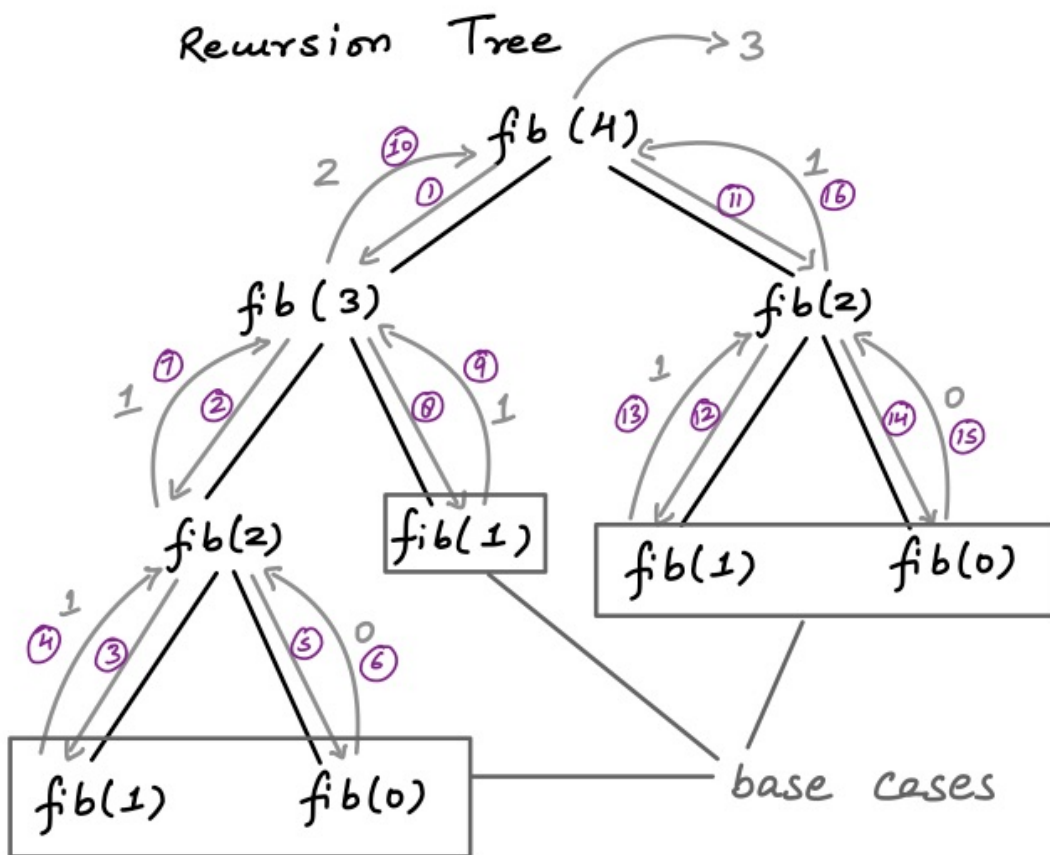
```

```

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)`

W7-L2

pass by reference

if we want to retain value of ans after function call

→ Try to pass everything by reference IF POSSIBLE

↳ 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;` → interpreted as an octal number

`cout << n;` → 64 → convert 0100 octal to decimal

`cin >> n;` → (0100)

`cout << n;` → 100 → no conversion as
cin reads 0100 as 100

Entry in function call stack

W7-L3

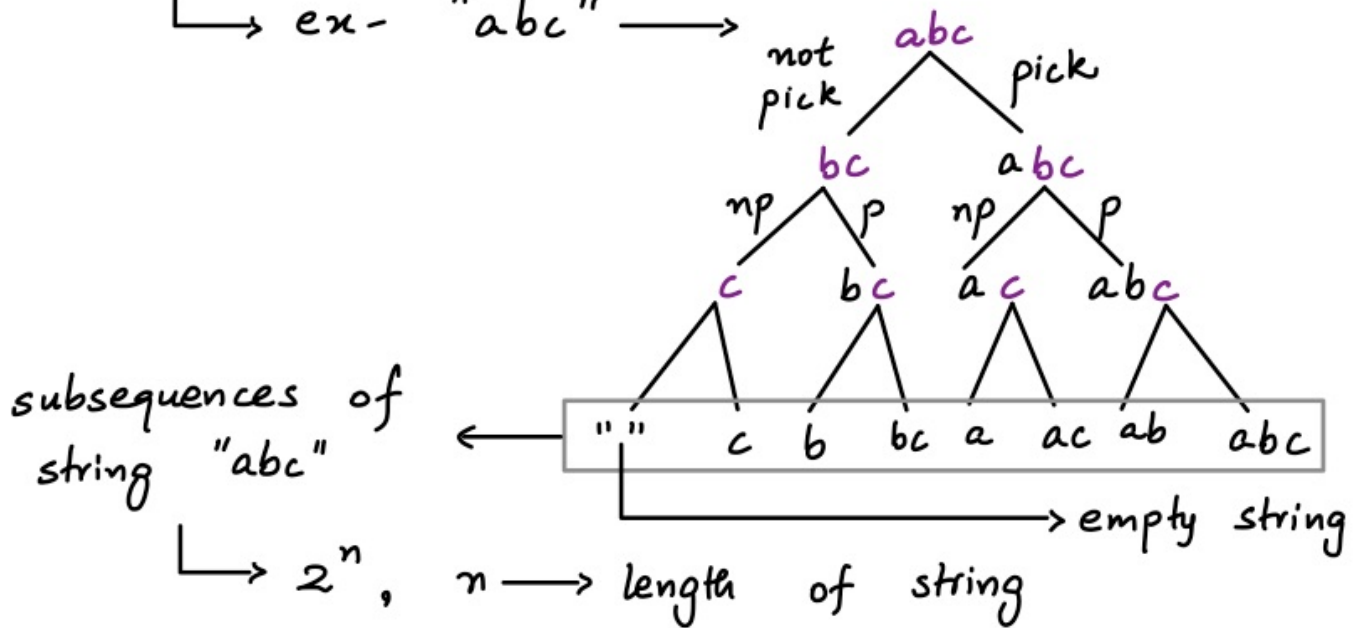
↳ Computer has track of variables of that function about their memory location and all in symbol table

→ Expression can't be pass by reference

Subsequences

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

→ ex- "abc" →



Arithmetic Operations on Strings

`s1 = "Ujjwal";`

`s2 = "2327";`

`cout << "Ujjwal" + "2327";` → ERROR

`s1 = s1 + s2;`

`cout << s1;` → "Ujjwal2327" → Concatenate

`-, *, /, %` → NOT VALID in strings

Coin change problem

W7-L4

→ find min. coins equivalent 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
either this or that

→ for using
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 -

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

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

⋮

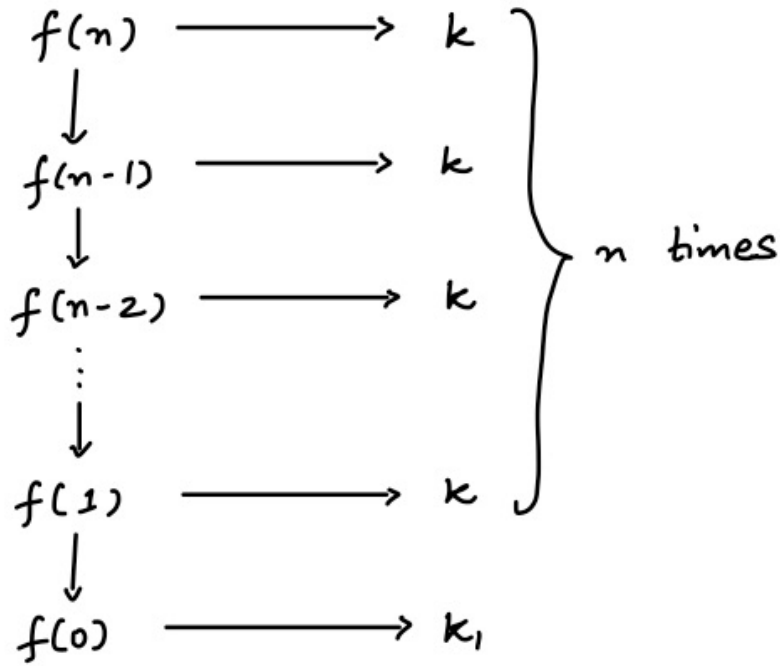
$$T(0) = k_1$$

$$T(n) = nk + k_1$$

$$T(n) = O(n)$$

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

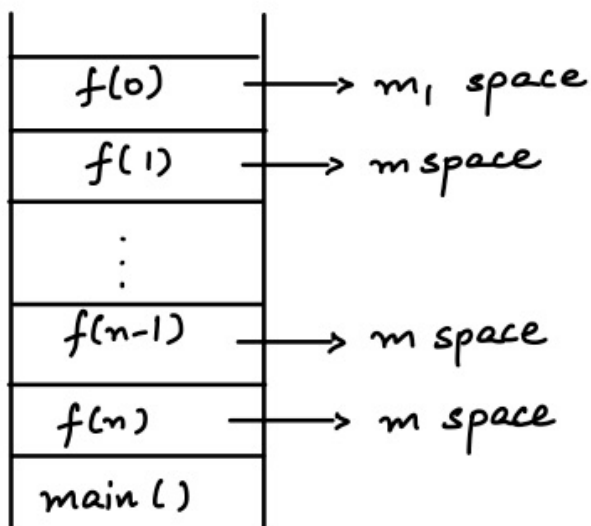
Recursive Tree Method - Best method



$$T(n) = nk + k_1$$

$$T(n) = O(n)$$

Space Complexity -



m and m_1 are constants

$$SC = mn + m_1$$

$$SC = O(n)$$

$\rightarrow$ max depth

Binary Search

$$f(n) = k + f(n/2);$$

base case $\rightarrow$ for $n=0$

Recursive Relation Method

$$\left. \begin{array}{l} T(n) = k + T(n/2) \\ T(n/2) = k + T(n/4) \\ \vdots \\ T(1) = k + T(0) \\ T(0) = k_1 \end{array} \right\} \begin{array}{l} a \text{ times} \\ \\ \\ \\ \end{array}$$

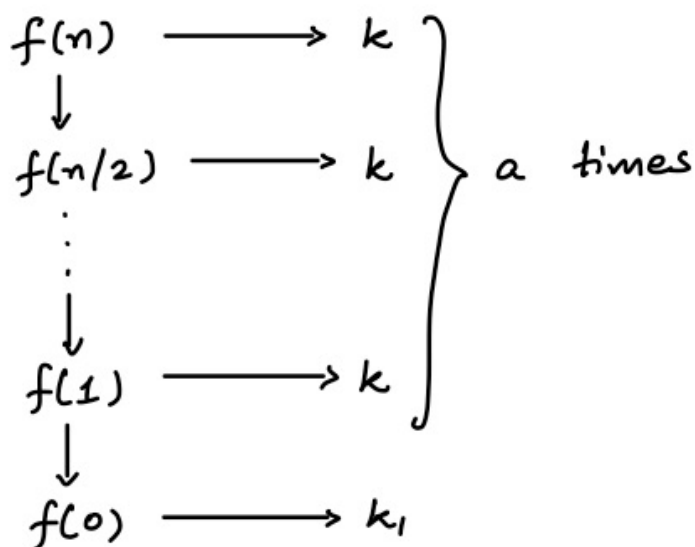
$$a = \log_2 n$$

$$T(n) = ak + k_1$$

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

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

Recursive Tree Method

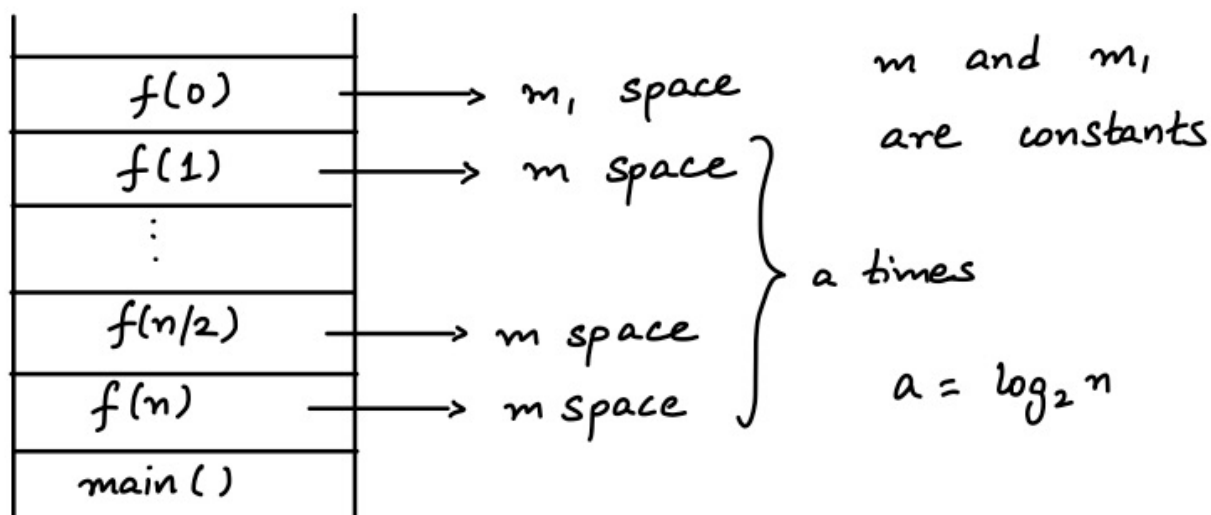


$$a = \log_2 n$$

$$TC = k \log_2 n + k_1$$

$$TC = O(\log_2 n)$$

Space Complexity -



$$SC - m \log_2 n + m_1$$

$$SC - O(\log_2 n)$$

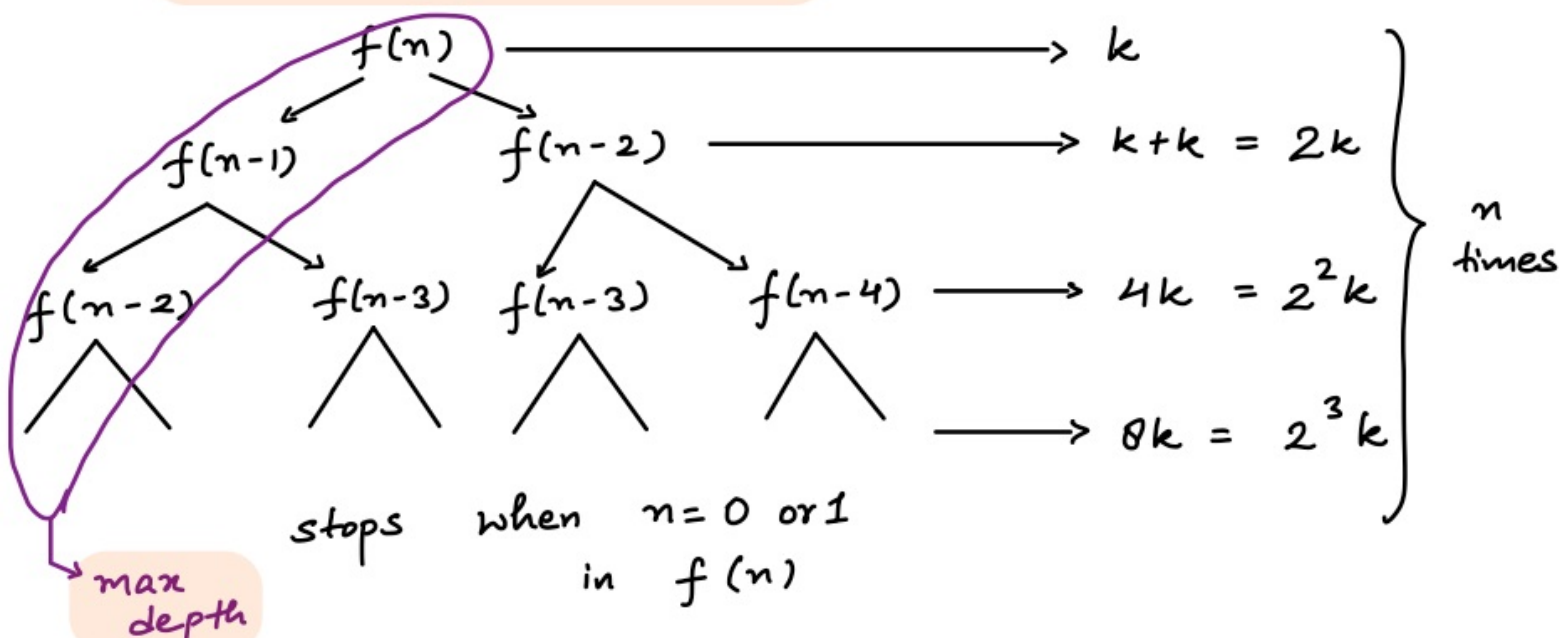
$\rightarrow$ max depth

Fibonacci Series

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

base case $\rightarrow$ for $n = 0$ or 1

Recursive Tree Method



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

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

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

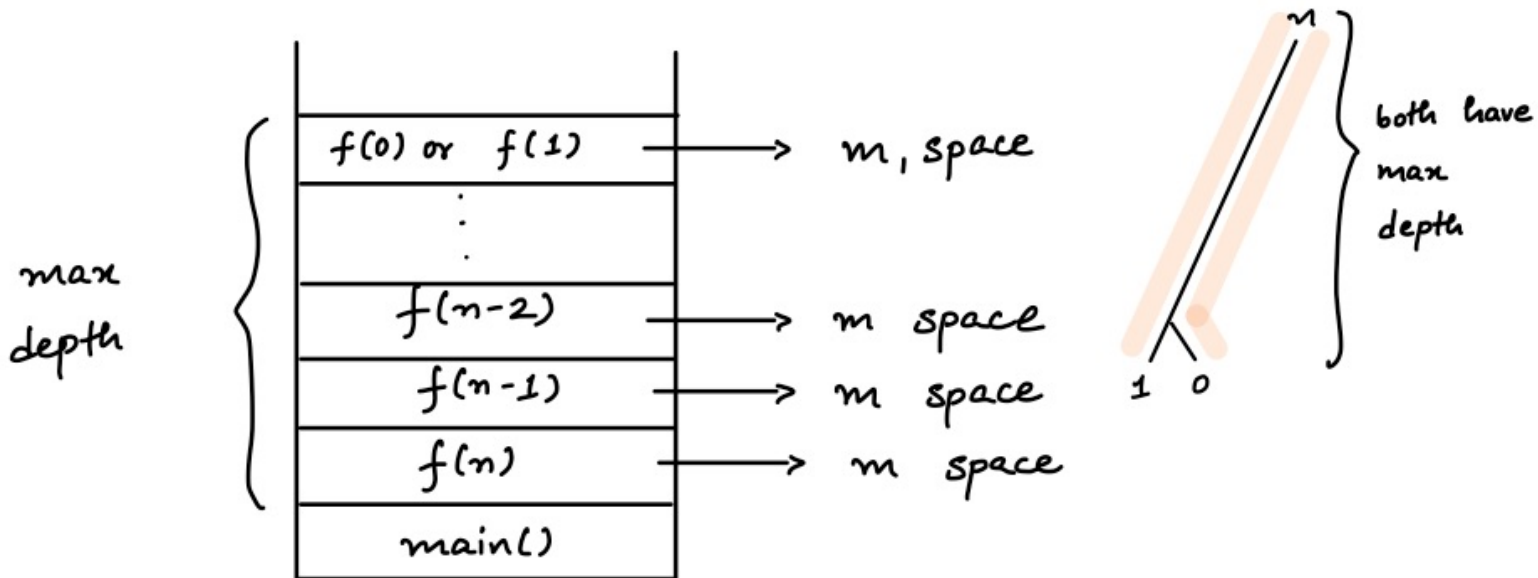
→ assume
for simplicity
and for large
numbers

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

→ exponential TC

→ very bad

Space Complexity



$$SC = m(n-1) + m,$$

$$SC = O(n)$$

→ max depth

if this is such a case

fun(0)	→ n^3 space / time
fun(1)	→ n^2 space / time
⋮	
fun(n-1)	→ n space / time
fun(n)	→ k space / time
main()	

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

Standard 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

→ in best case too →
 T.C. - $O(n \log n)$ → number of levels

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

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; ————→ This is one way  
    int pivot = arr[s]; Other ways - choose last  
                                - choose random  
                                ↓  
                                randomised quick sort  
    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
```

```

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. - $O(n \log n)$ $\longrightarrow$ $\log n$ number of function calls

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

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

Backtracking

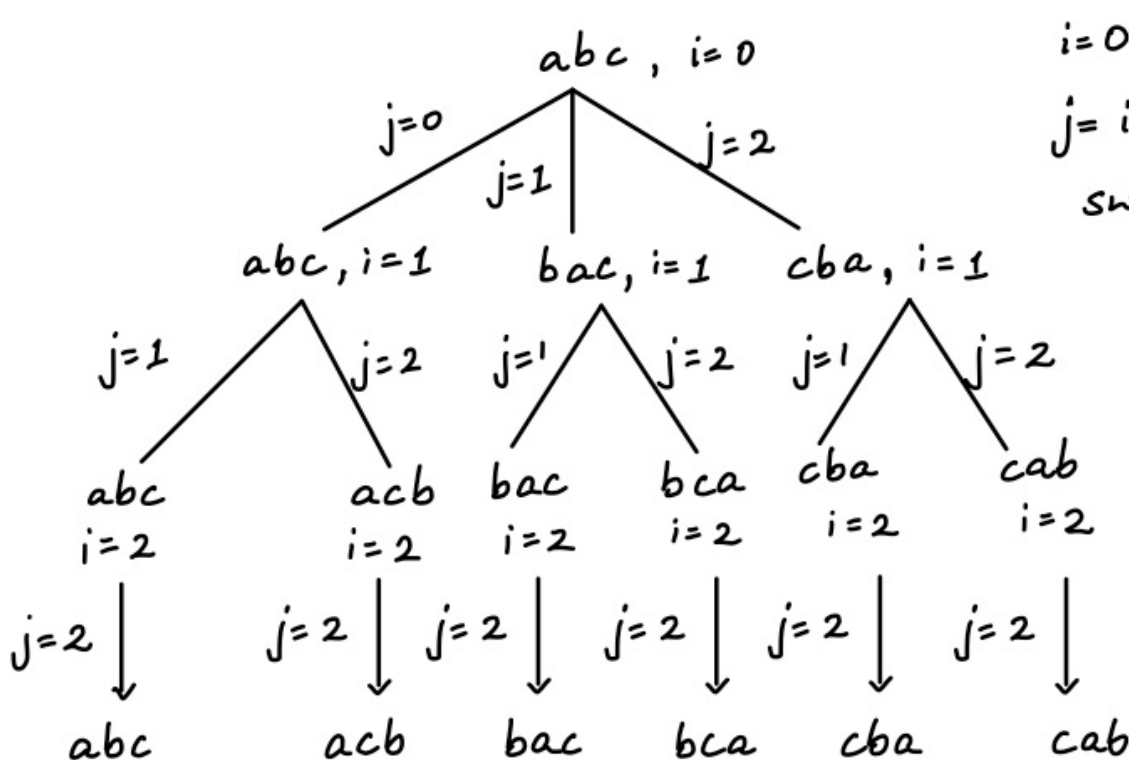
- specific form of recursion
- explore all possible solutions and
- don't check the solution you have already discarded
- brute force
- used for reference variables passed in functions

Permutation in a string

$s = "abc"$

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

→ total - $n!$



$i = 0$ to $< \text{str.length}()$
 $j = i$ to $< \text{str.length}()$
 $\text{swap}(\text{str}[i], \text{str}[j])$

T.C. - $n!$

S.C. - $s.\text{length}()$

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
```

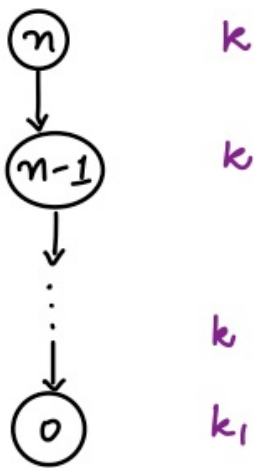
```
  fun(n-1)
```

```
}
```

k_1
 k_2 } k

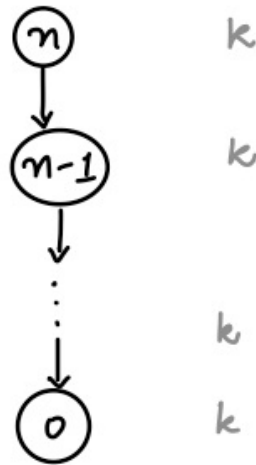
pass by value
 1 variable n is created

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

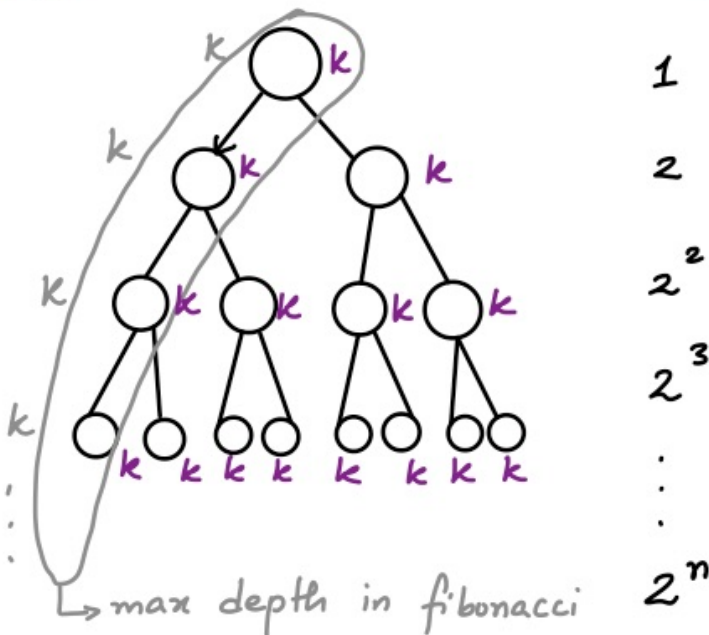
power of 2

check sorted

max in array

min in array

Type 2 Non Linear Tree



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

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

$$S.C. - nk$$

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

fibonacci

jump stairs

subsequences

Type 3 Logarithmic



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

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

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

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

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

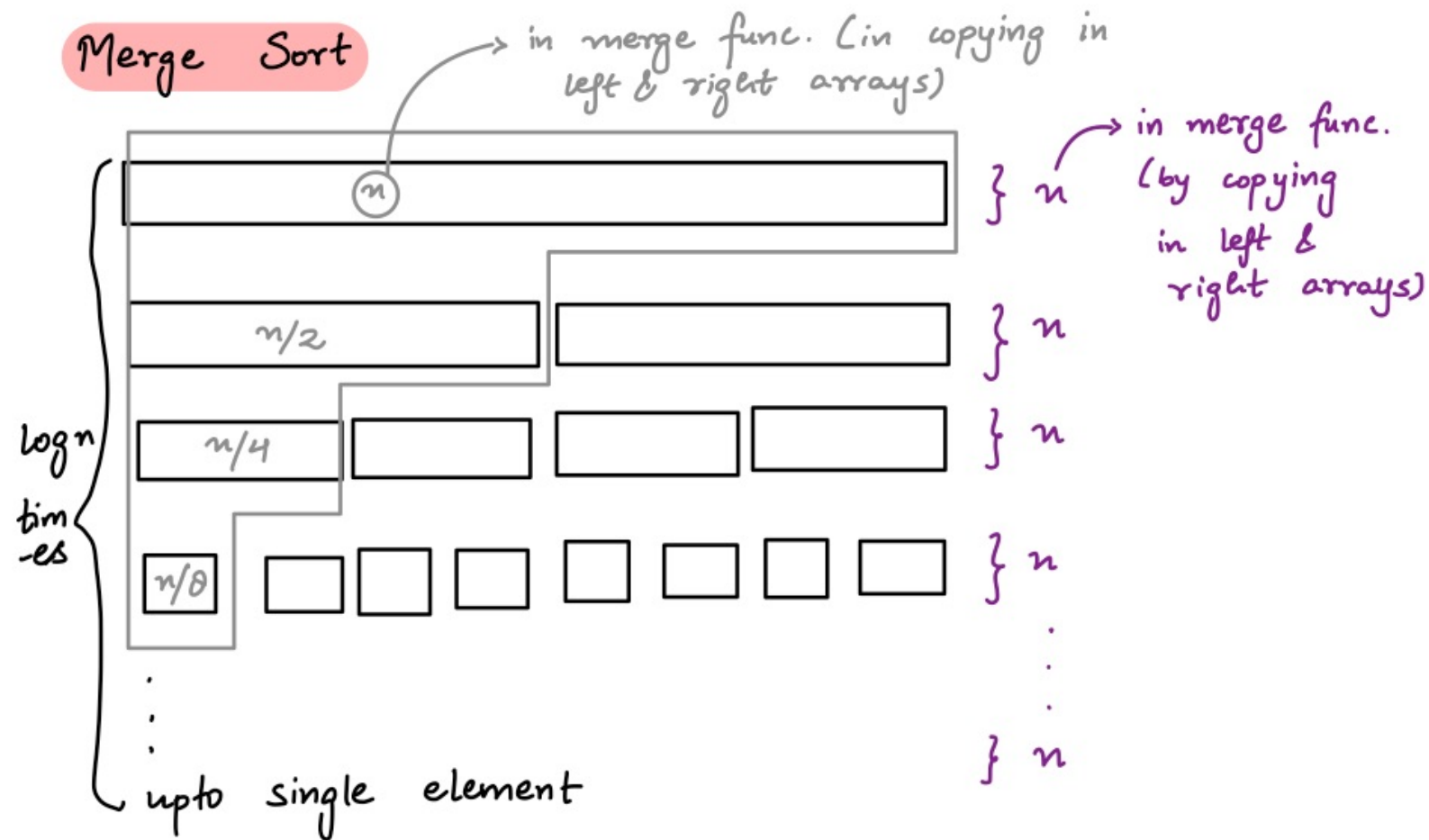
$$S.C. - n \log n$$

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. - $O(n \log n)$

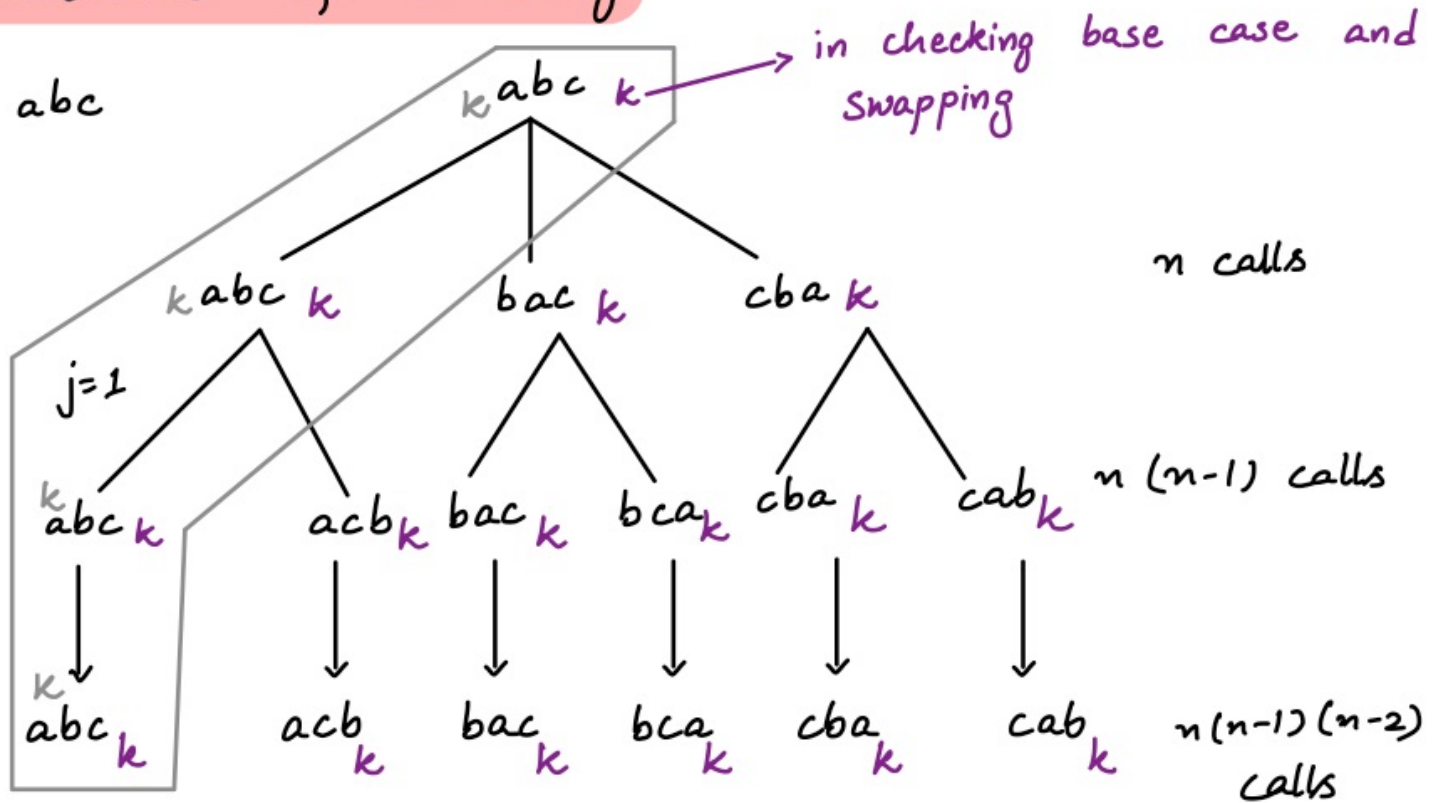
S.C. - $n + \frac{n}{2} + \frac{n}{4} + \dots + 2$

S.C. - $O(n)$

Backtracking Questions

Permutations of a string

$s = abc$



$$T.C. - k + nk + n(n-1)k + n(n-1)(n-2)k + \dots + \boxed{n!k}$$

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

$$S.C. - n * k$$

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

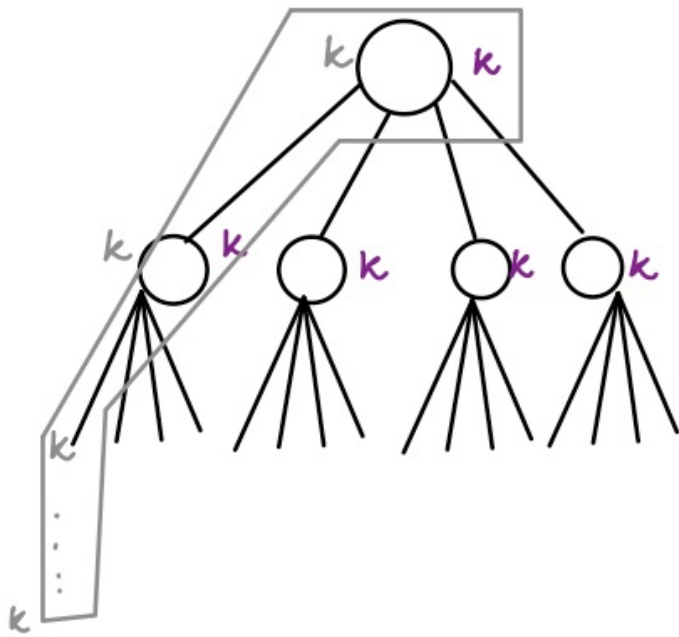
Rat in a maze

m rows
n cols

func ()
base case

4 recursive calls
↳ down, left, right, up

}



1
4
16
⋮

} m * n levels

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

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

$$S.C. - k * mn + mn \xrightarrow{\text{depth}} \text{of visited 2D array \& pass by reference}$$

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

Sudoku Solve

$n \times n$ matrix

standard

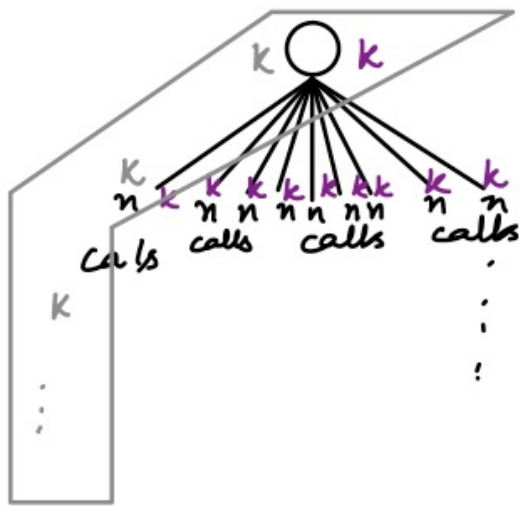
$\rightarrow n = 9$

```
func() {
```

base case

n recursive calls + isSafe func.

```
}
```



$n \times n$ levels

$\rightarrow$ number of elements
in a matrix

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

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

S.C. - $k * (n \times n)$ $\xrightarrow{\text{depth}}$

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

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

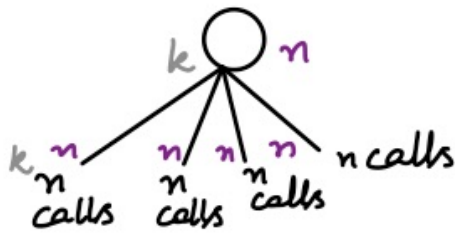
N Queens

$k \begin{cases} k_1 \\ k_2 \end{cases}$

```

func ( )
    base case  $\longrightarrow k$ 
    n is Safe function  $\longrightarrow O(n)$ 
    n recursive calls
}
    
```

$\} O(n)$



1

n

n^2

$\vdots$

n^n

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

T.C. - $O(n^n) \longrightarrow$ worst case

T.C. - $O(n!)$ $\longrightarrow$ average case

S.C. - $k * n$

S.C. - $O(n)$

S.C. - $k * n + n \xrightarrow{\text{in optimized sol for creating 3 maps}}$

S.C. - $O(n)$

$\longrightarrow n * n!$ can be written as $n!$

$\longrightarrow n * 2^n$ can be written as 2^n

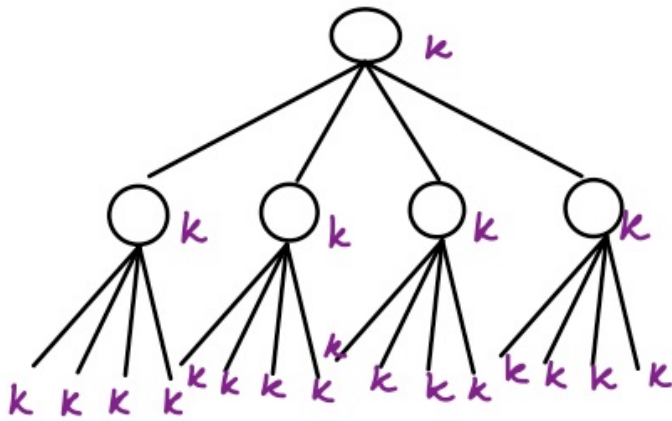
Phone Keypad Problem

1 2 3
 abc def

4 5 6
 ghi jkl mno

7 8 9
 pqrs tuv wxyz

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



1
 4
 4^2
 ...
 4^n

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

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

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

in max depth length of 1 output string
 ↑ ↑
 (k + 2k + ... + nk) (4^n * n)

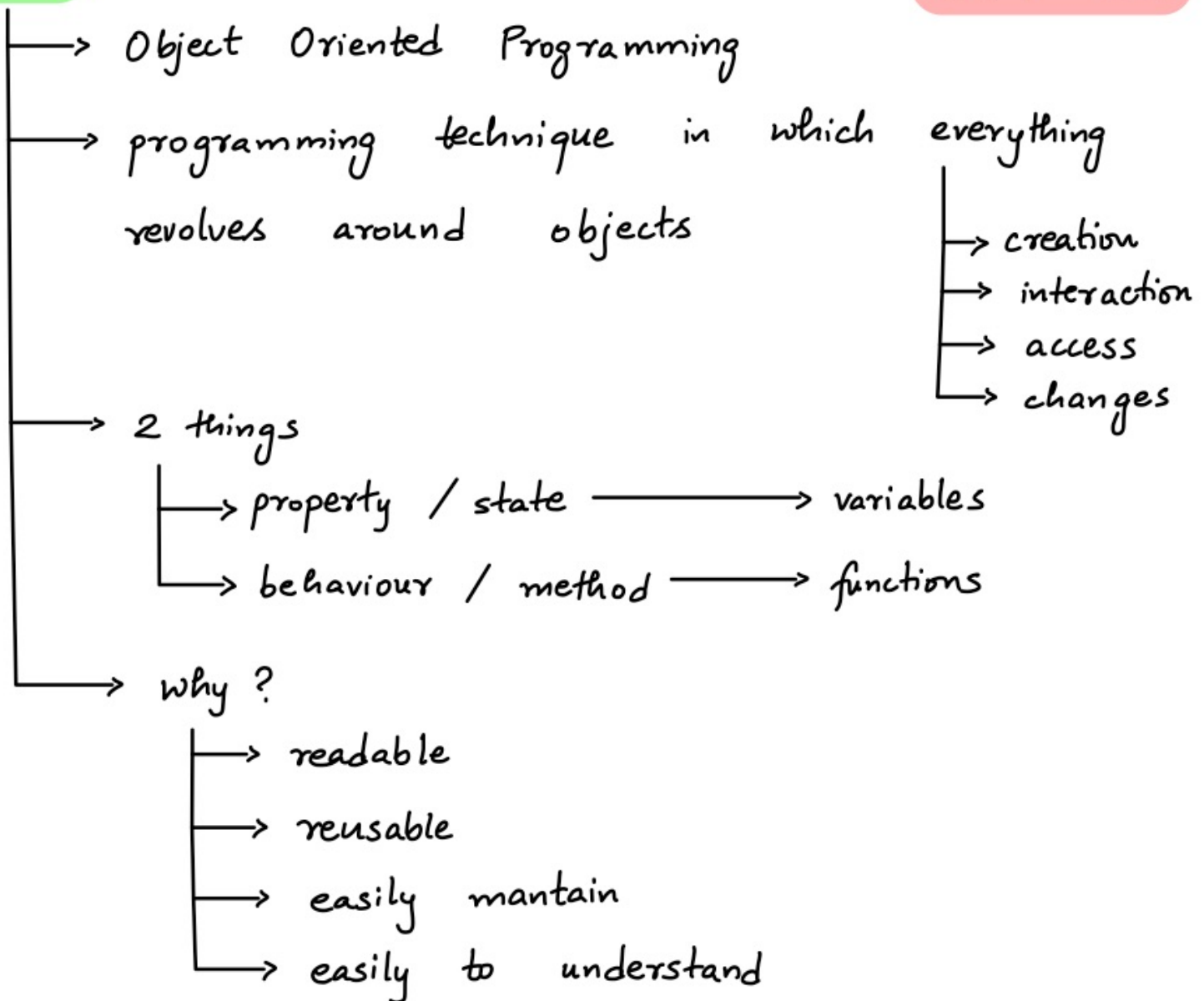
$$S.C. - O(4^n)$$

store
 output in
 base case

no. of strings
 stored in ans
 array

OPP

W9-L1



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() {  
    }  
};
```

size of (class) -

```
class animal {  
  
};
```

cout << sizeof (animal); → 1 to give existence for identifying this class, so min.

```
class animal {  
    int age;  
};
```

possible memory is allocated

cout << sizeof (animal); → 4

```
class animal {  
    string name;  
};
```

cout << sizeof (animal); $\longrightarrow$ 24 Why?

```
class animal {  
    int age;  
    char ch;  
};
```

cout << sizeof (animal); $\longrightarrow$ 0

$\hookrightarrow$ 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

object Name . state ;

object Name . 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) { setter
    name = newName;
}

};

int main () {
    animal ramesh;

    cout << ramesh.name; —————> ERROR

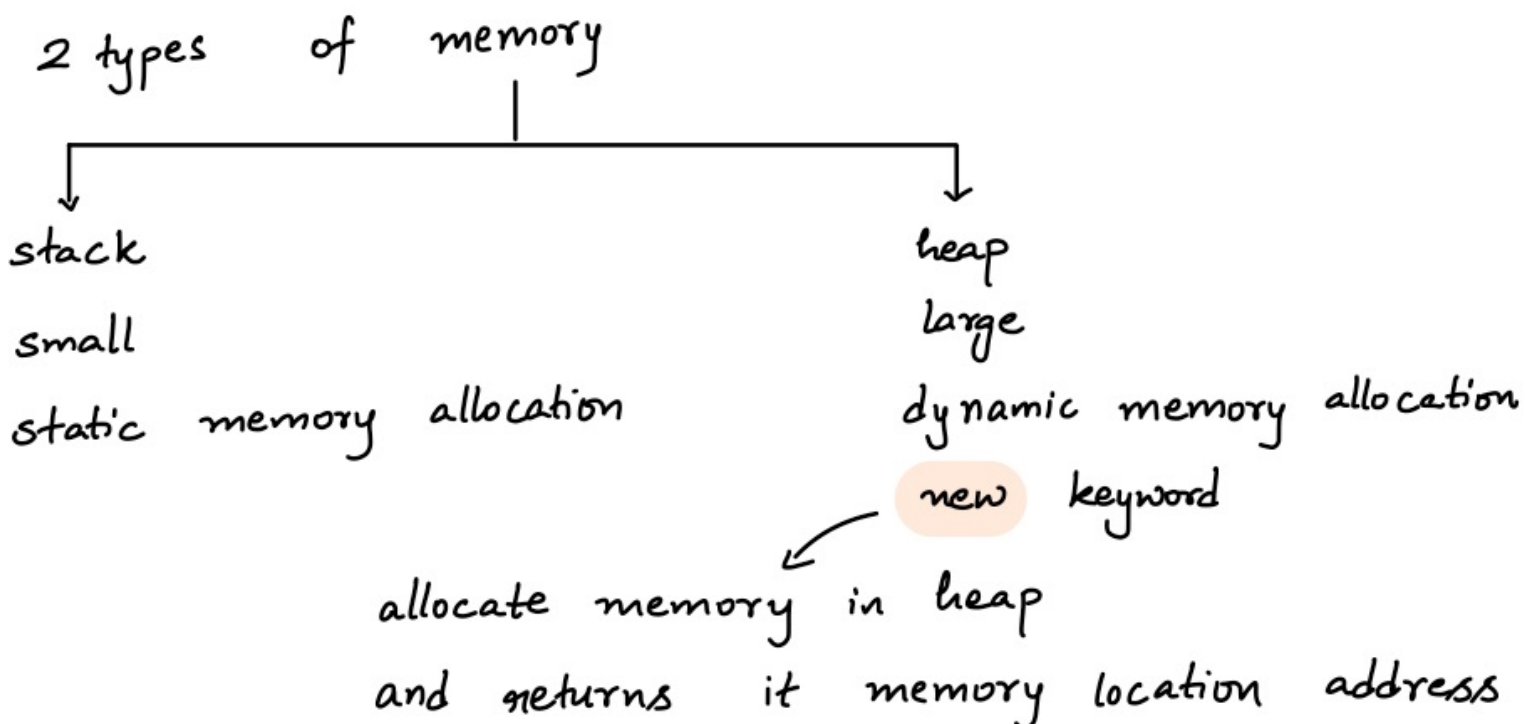
    cout << ramesh.getName(); —————> gv

    ramesh.setName ("Lion");

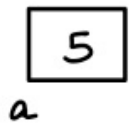
    cout << ramesh.getName(); —————> Lion

```

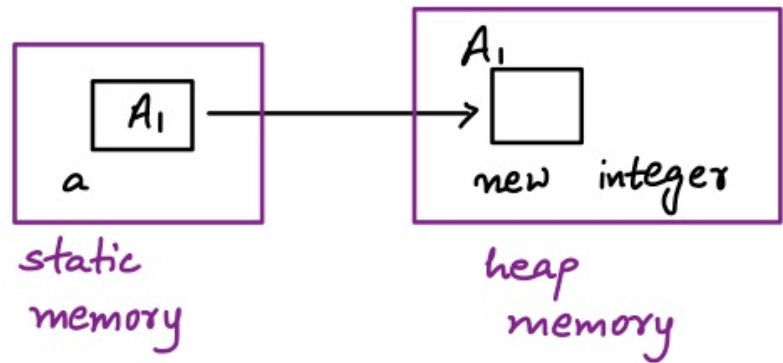
Dynamic Memory Allocation



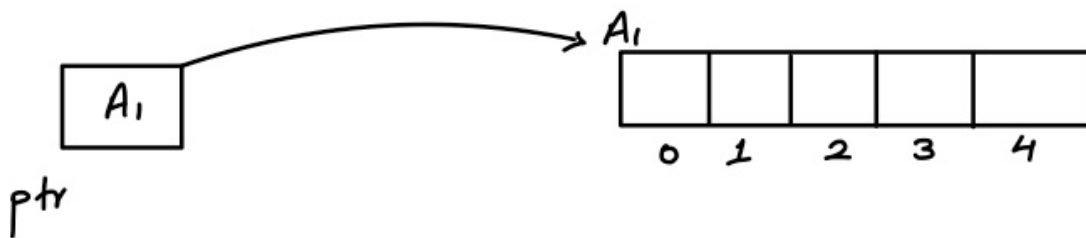
`int a = 5;`



`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];`

Deallocation

`delete a;`

`delete arr[];`

Object Creation using dynamic memory allocation

```
animal * suresh = new animal;
```

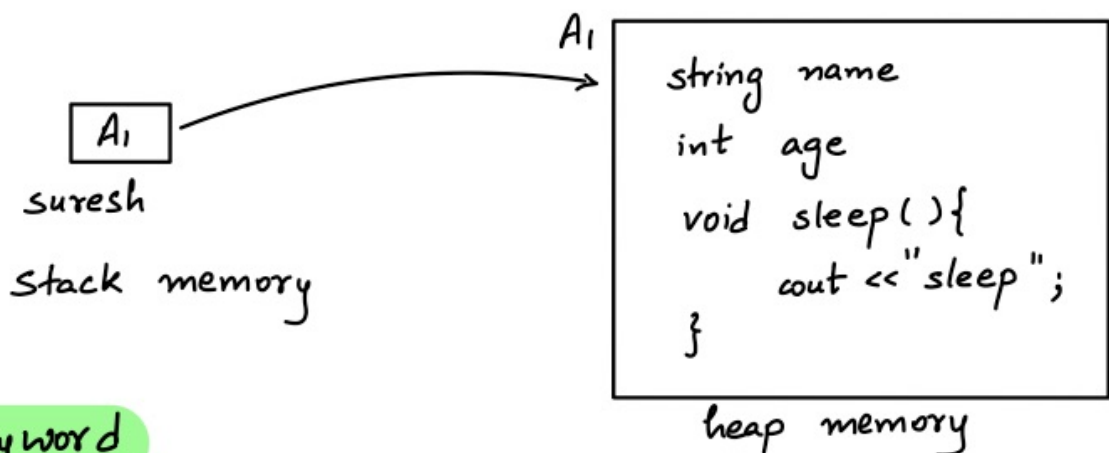
suresh.name; $\longrightarrow$ ERROR as suresh is a pointer, *suresh is object

(*suresh).name;
(*suresh).sleep();

} 1st way of access

suresh $\rightarrow$ name;
suresh $\rightarrow$ sleep();

} 2nd way of access



this keyword

$\longrightarrow$ pointer to current object

```
class animal {
```

```
    int age;
```

```
    public:
```

```
    void setAge ( int age) {
```

```
        this  $\rightarrow$  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;
```

```
}
```

→ accessing members and methods

using this keyword is **GOOD PRACTICE**

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

// 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";  
}
```

```
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

// copy constructor (right way)

```
animal (animal &obj) {  
    this → age = obj. age;  
    this → weight = obj. weight;  
    cout << "Copy Constructor Called";  
}
```

}

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";  
    }  
}
```

4 Pillars of OOPs

W9-L2

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction (Pitaji of all OOPs pillars)

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

Inheritance

- inherit some properties (data members and data functions) from parent / super / base class to sub / child / derived class
- Objective → Reusability
- is a relationship
- syntax
 - ↳ *childName is a parentName*
class childName : mode1 parentName 1,
..., mode N parentName N {
//additional states and methods
}

```
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 inheritance		
	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

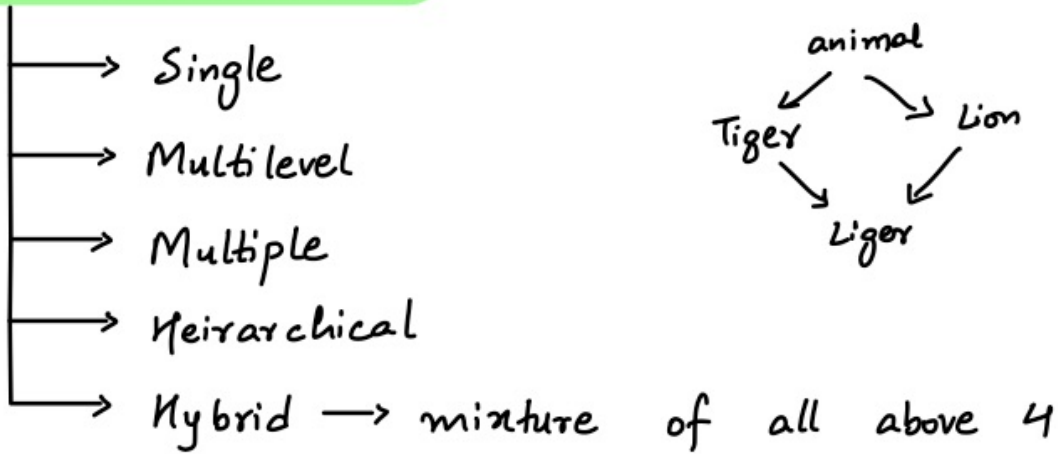
↳ can access inside class but not outside
↳ cannot be inherited



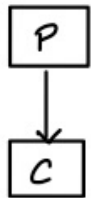
cannot be accessed inside child class

Type of Inheritance

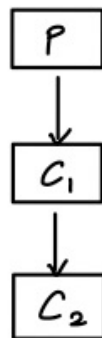
P- Parent C- Child



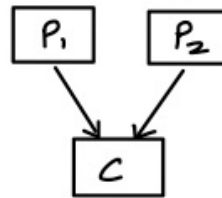
Single



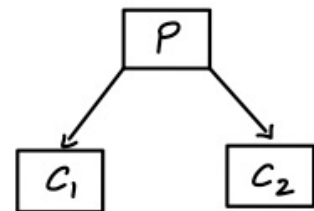
Multilevel



Multiple



Hierarchical

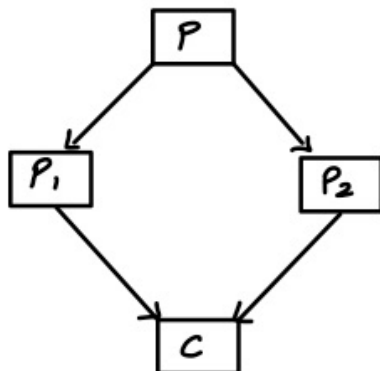


Diamond Problem / Inheritance Ambiguity Problem

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

→ `objectName . ParentClass Name :: property ;`

scope resolution operator

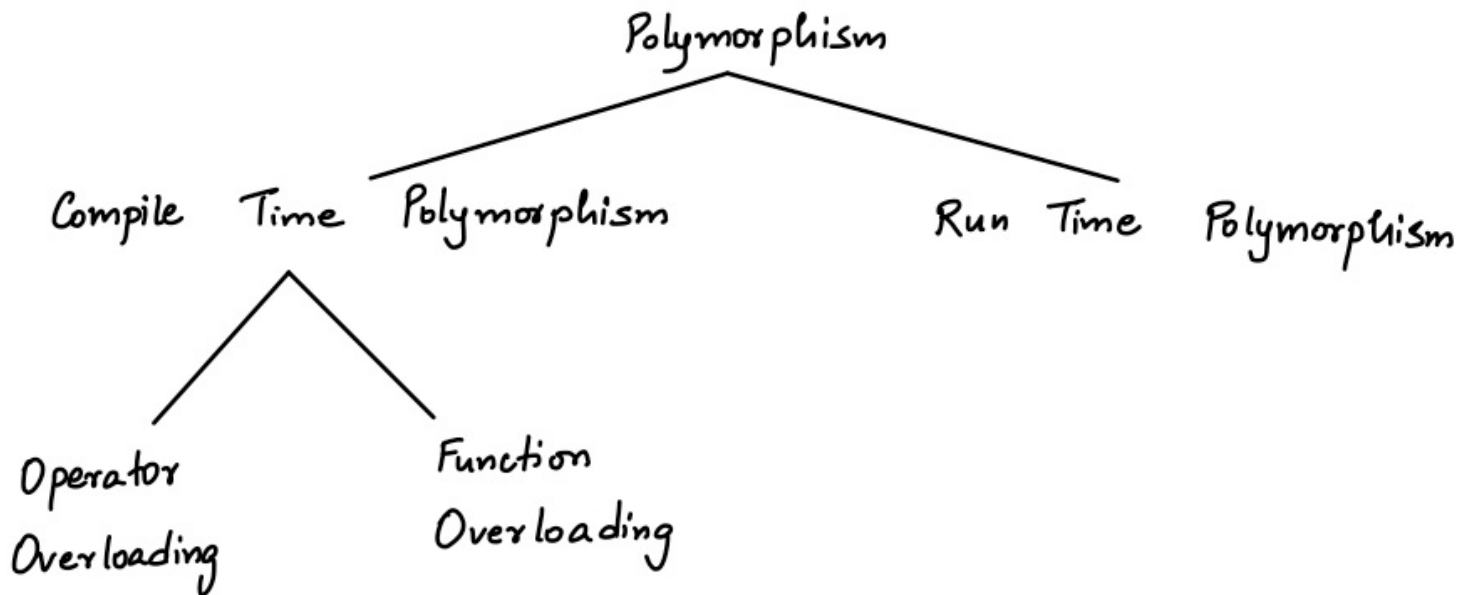


→ If parent and child have same property, then parent property inside child class will get overwrite by child property

→ Multiple inheritance 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

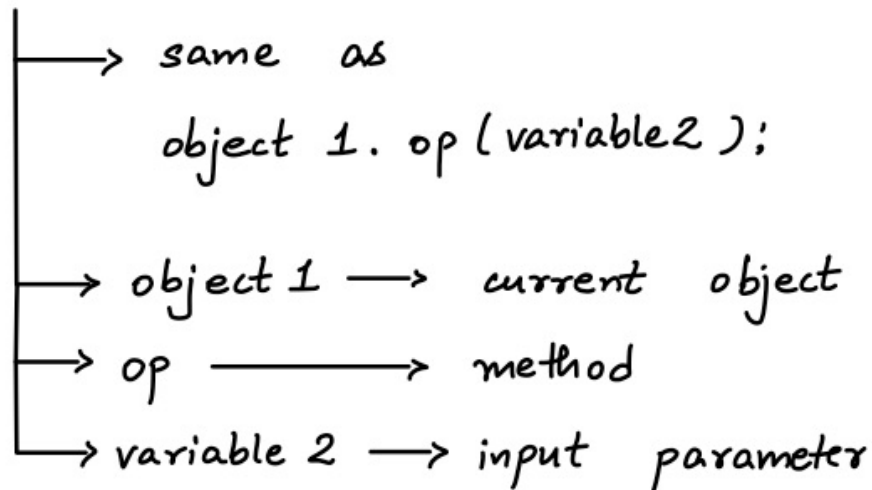
    return 0;
}

```

Operator Overloading -

↳ to make new implementation of an operator

object 1 op variable 2



```
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;  —————> same as a.+(b)
                                ↳ 3
```

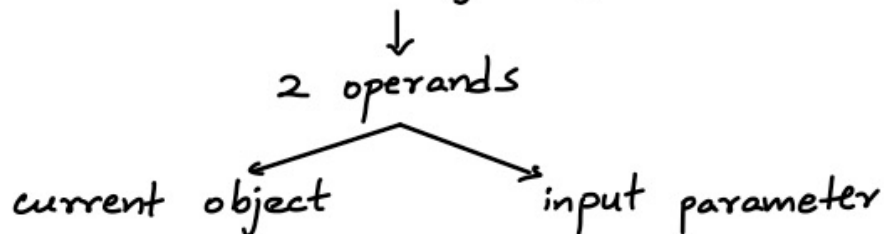
```
    a + 'c';  —————> 15
```

```
    a + '7';  —————> 15
```

```
    return 0;
```

```
}
```

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



Run Time Polymorphism -

↳ polymorphism after execution

Function Overriding -

Function / Method Over-riding

- when function of parent class is defined separately inside child class
- reusable + custom behaviour wherever you want

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

```
class Animal {  
    public:
```

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

```
int main () {  
    Animal a;  
    cout << a.speak();  
  
    Dog d;  
    cout << d.speak();  
}
```

// upcasting

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

```
cout << b. speak(); —————> speak —> function of  
parent class  
called
```

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

```
cout << b. speak (); —————> bark
```

// down casting

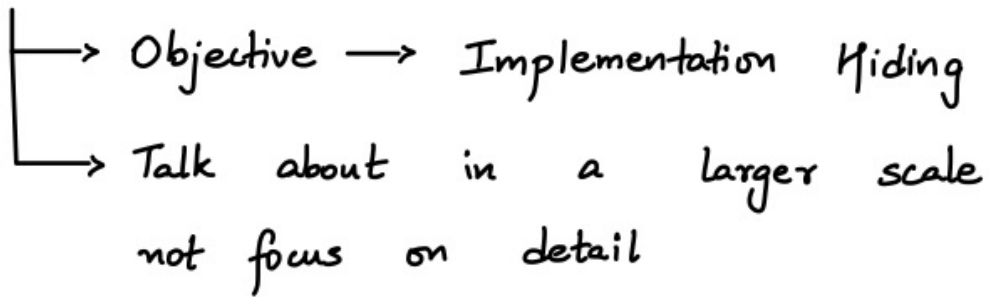
```
Dog * c = new Animal ; —————> Compiler dependent  
(ERROR or not)
```

```
cout << c. speak ; —————> bark
```

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

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

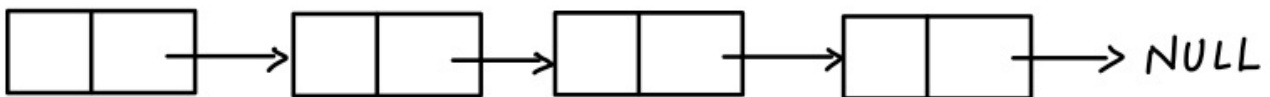
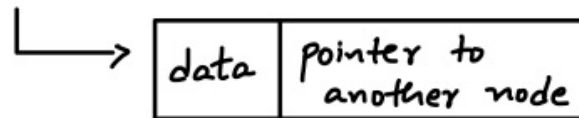
Abstraction



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

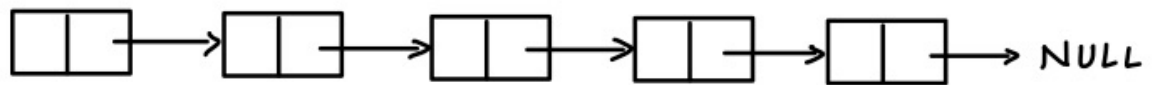


node creation (simplest)

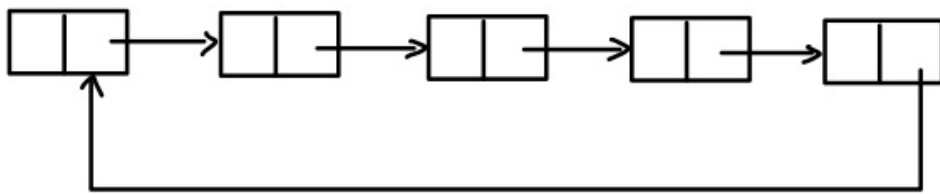
```
class node {  
    public:  
    int data;  
    node * next;  
}
```

Types of linked list

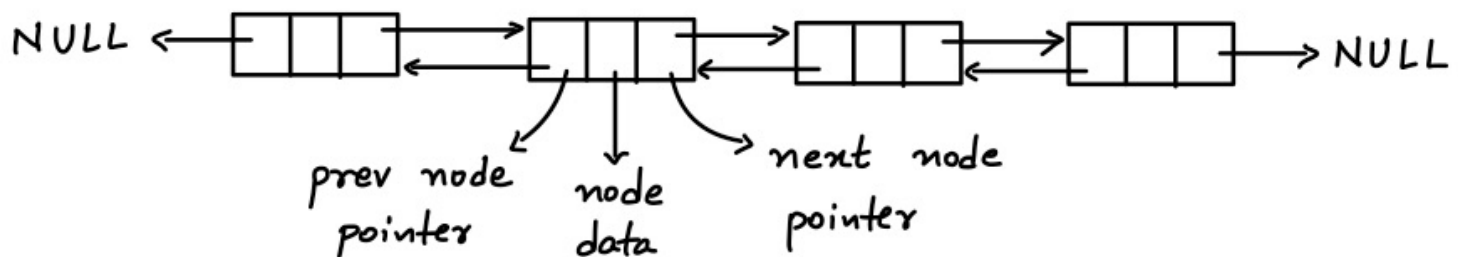
→ Singly LL



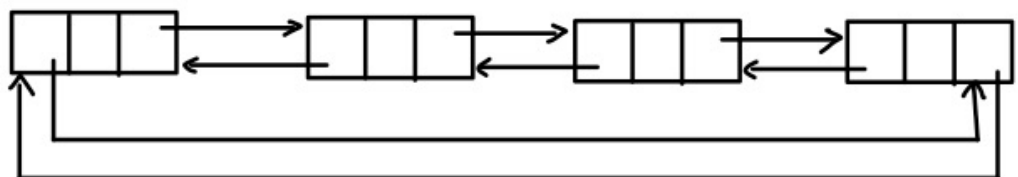
→ Circular LL



→ 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 Linkedlist

→ 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

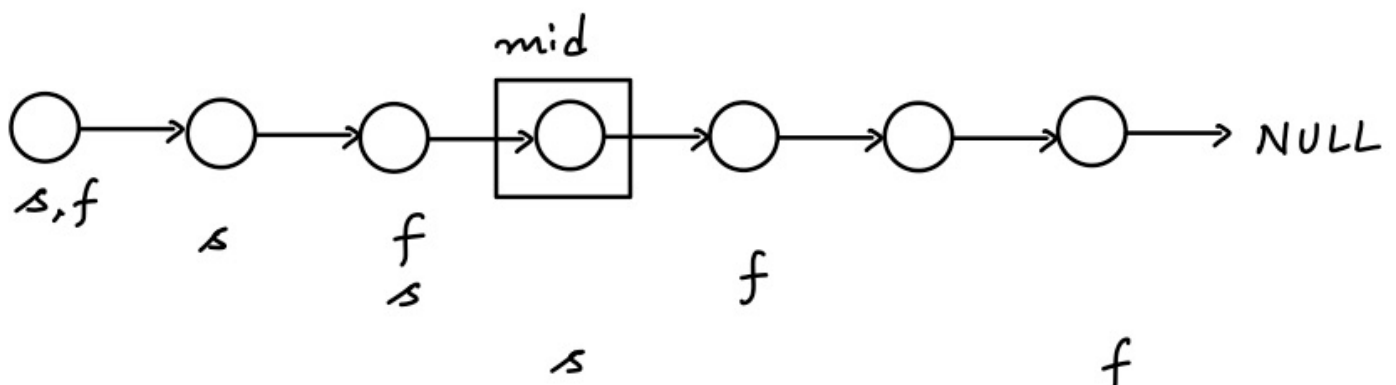
→ Stopping condition in loop acts as base case

Tortoise Algorithm

W10-L3

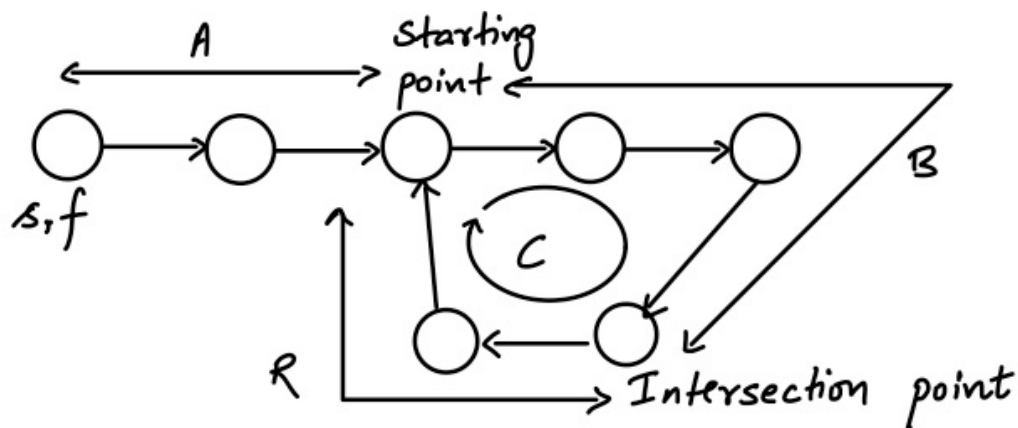
→ fast and slow pointer approach
→ get middle

```
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



$$\text{distance (f)} = 2 * \text{distance (slow)}$$

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

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

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 $\Rightarrow$ reach starting point

Also head + A $\Rightarrow$ reach starting point

$$R + zL = A$$

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

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

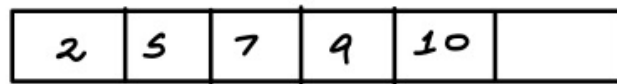
} insert

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

creation

queue <int> q;

rear always shows empty block

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 $\text{front} == \text{rear} \rightarrow$ 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.

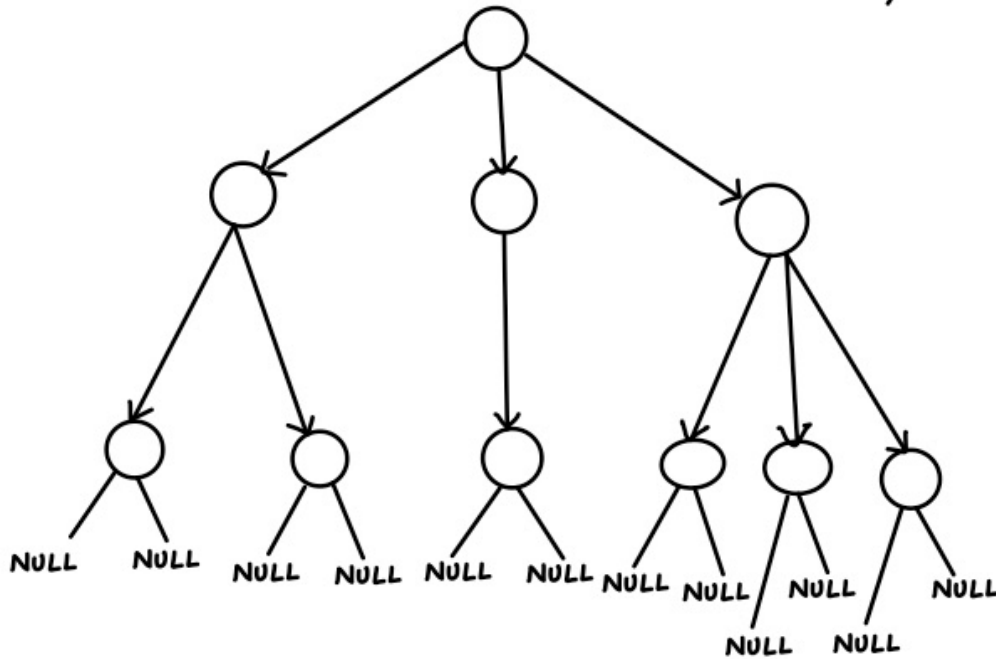
sliding window

- 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 hierarchical 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

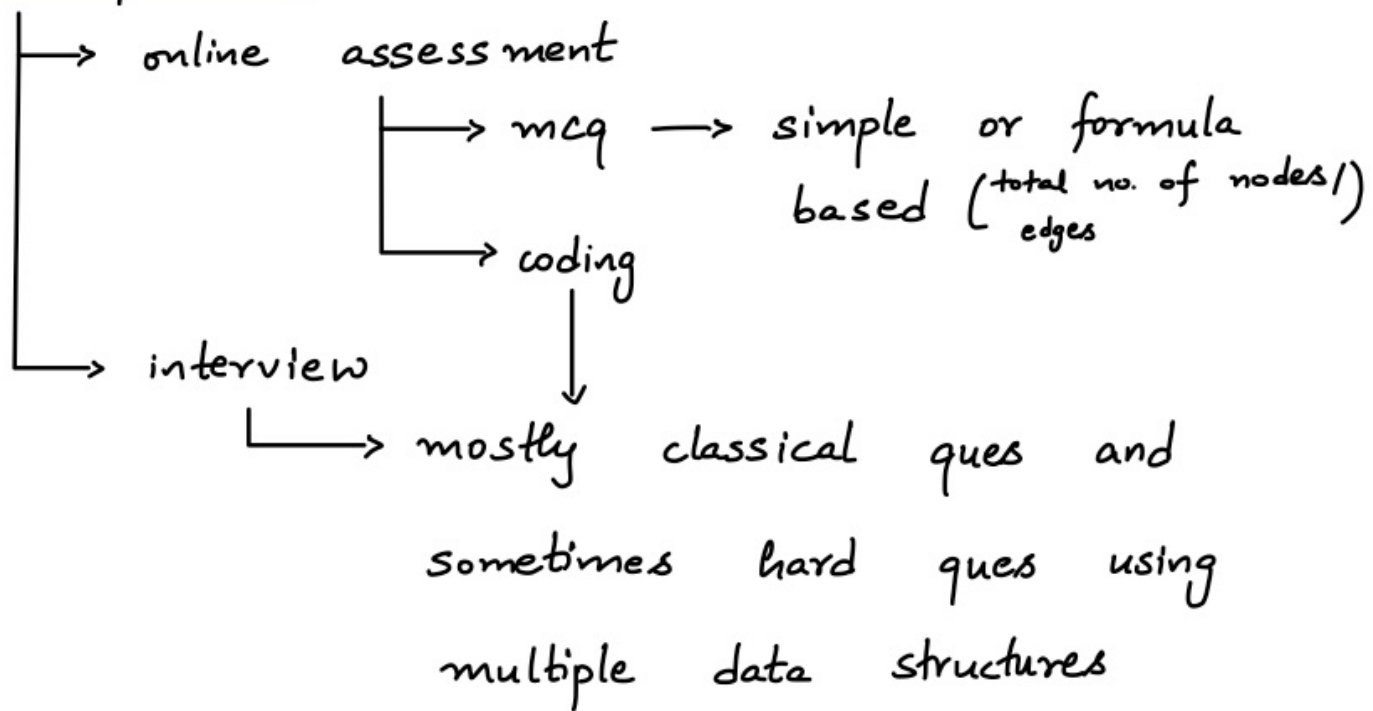
height of node → no. of ^{acc. to ques} nodes/edges from leaf to node in longest path

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

height of tree

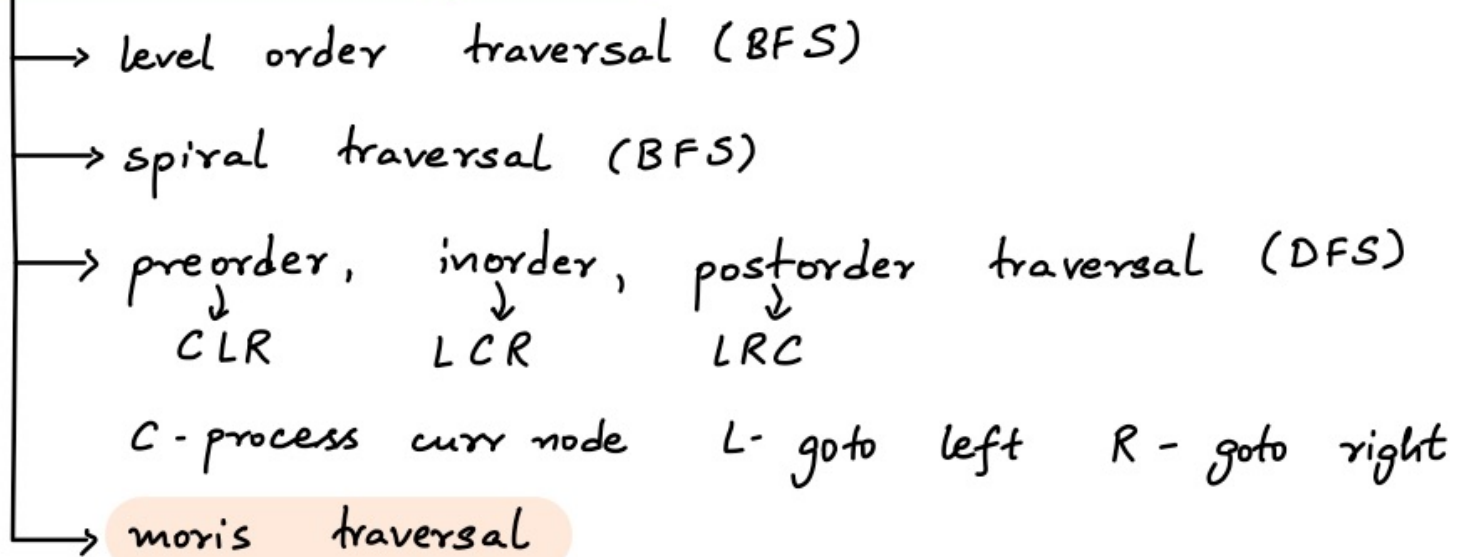
↳ height of root node

→ trees questions



→ tree is aasaan if you know recursion

Tree traversal algorithms



→ 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;

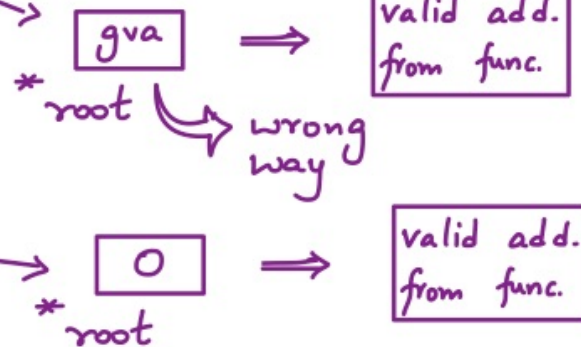
root = func();

}

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.

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 don't find logic to solve tree question, just start running recursion in examples and you will 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 (C[→]LR)

→ build tree from inorder and postorder (L[←]RC)
 ↳ right recursion call before
 left recursion call

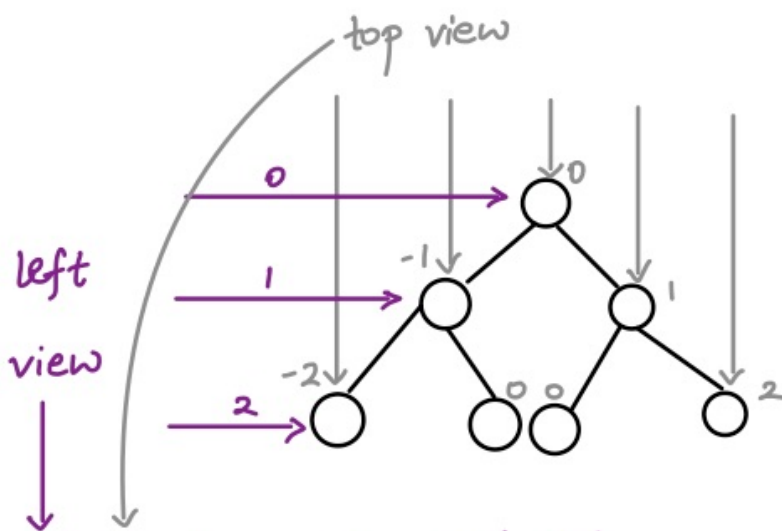
W13-L5

→ In ques. where we find views

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

we use 3 concepts

- map
- level order traversal
- horizontal distance from root



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

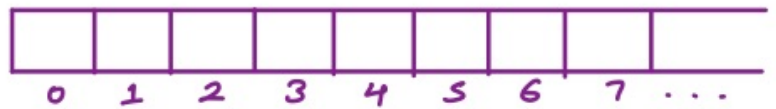
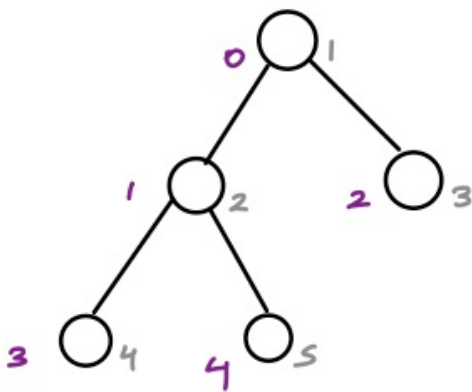
HEAPS

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 $\rightarrow$ left child - $2i+1$, right child - $2i+2$
 $\rightarrow$ parent - $(i-1)/2$

i^{th} node $\rightarrow$ left child - $2i$, right child - $2i+1$
 $\rightarrow$ 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 $(\frac{n}{2} + 1)^{\text{th}}$ 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. - $O(n \log n)$

DYNAMIC PROGRAMMING

W17-L1

→ technique used to optimize sol.

→ key point

→ those who forget the past are condemned to repeat it



don't 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

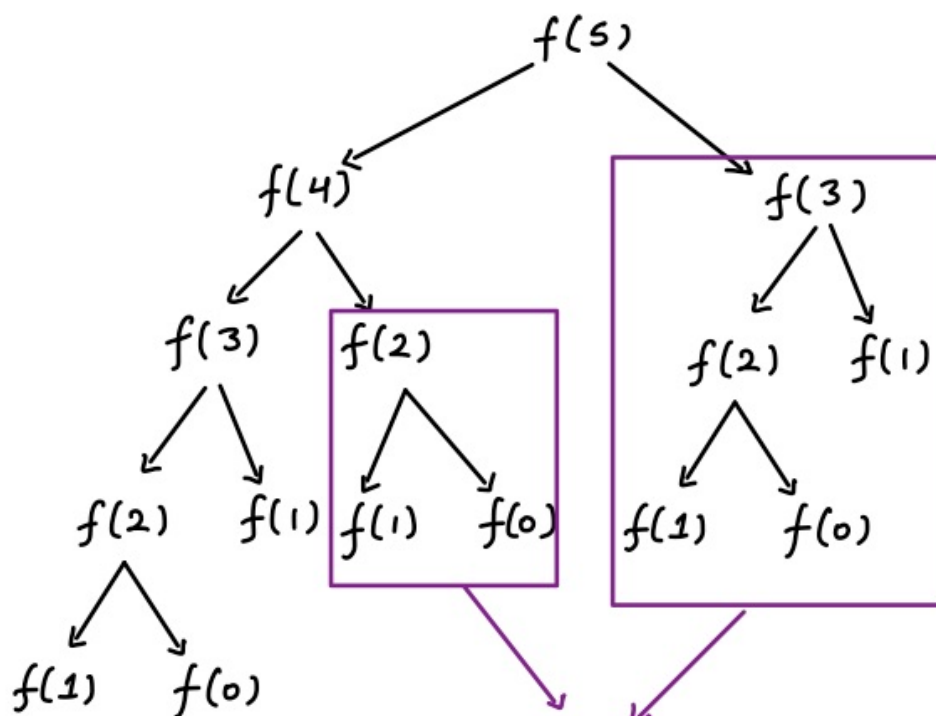
Top Down (Recursion + Memoisation)

- 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;

Space Optimization Sol.

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

ex- fibonacci



Solving overlapping problems again

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. - $O(n)$

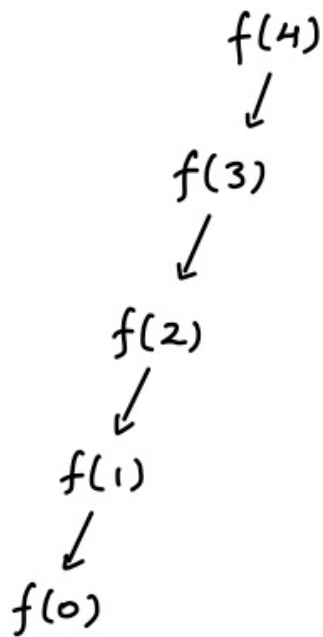
space optimization

T.C. - $O(n)$

S.C. - $O(1)$

DP can be used

factorial



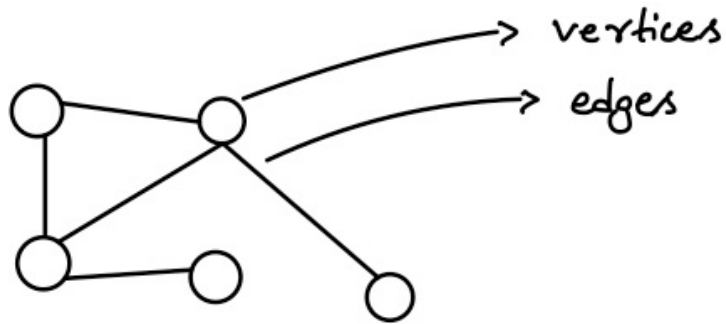
no overlapping problems

only have optimal substructure

DP cannot be used

GRAPHS

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



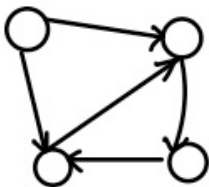
→ use case

↳ google map, facebook

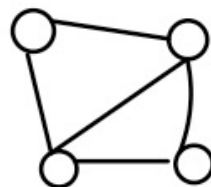
Types of graphs

→ Directed Graph

→ Undirected Graph

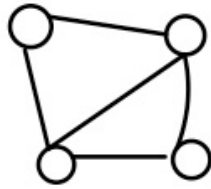
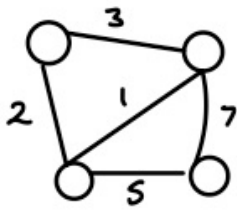


Directed



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

↳ in degree

↳ 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

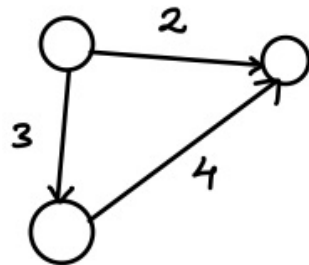
Cyclic Graph

- 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

