Unit-1
Chapter-1
INTRODUCTION

→ Role of algorithms in computing :-

- Algorithms are tool for solving a well-specified computational problem.
- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired O/P.
- Algorithms are generally created independent of underlying languages, i.e, an algorithm can be implemented in more than one programming language.
- From the data structure point of view, following are some important categories of algorithms:
  - Search :- Algorithm to search an item in a data structure.
  - Sort :- Algorithm to sort items in a certain order.
  - Insert :- Algorithm to insert item in a data structure.
  - Update :- Algorithm to update an existing item in a data structure.
  - Delete :- Algorithm to delete an existing item from a data structure.

Roles:

> It depends on how efficient the algorithm when higher order of i/ps is given.
> The possible restrictions / constraints on the values
> The architecture of the computer & the kind of storage devices to be used.
> Another important aspect is the correctness of the algorithm implying that algorithm is correct if, for every instance, it produces correct O/P.
> An incorrect algorithm might not halt at all on some i/p instances, or give incorrect O/P.

i/P → Algorithms → O/P

- Algorithms must be :
  - Correct : For each i/P produce an appropriate O/P.
  - Efficient : Run as quickly as possible & use as little memory as possible.
- Algorithm is any well-defined computational procedure that takes some values or set of values, as i/P & produces some value or set of values as O/P.
- Algorithm is a method of solving a problem using a sequence of well-defined steps.

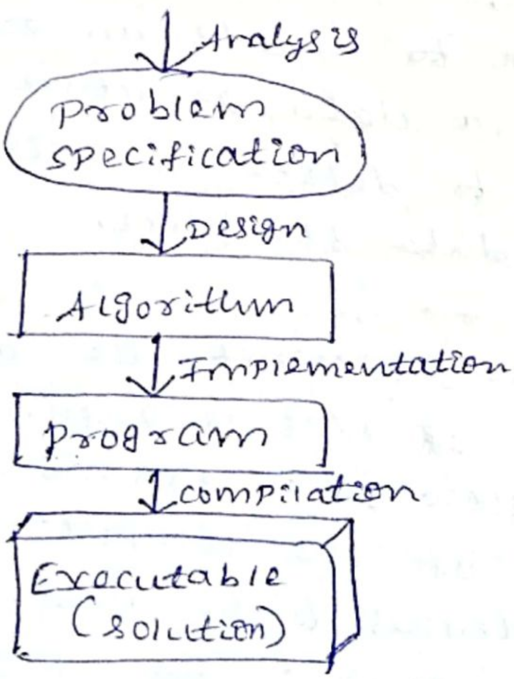
eg:- sorting problem $\langle a_1, a_2 - - a_n \rangle$

i/P ; A sequence of 'n' nos.

O/P ; A permutation of the i/P sequence ;

$\langle a'_1 < a'_2 < - - < a'_n \rangle$

↳ Problem - solving process :-



From algorithms to Programs :-

problem ⇒ Algorithm: A sequence of instructions describing how to do a task (process)

⇓

C++ Program

Examples :-

- Internet & Networks: The need to access large amount of information with the shortest time. The problem of finding the best routes for data to travel. Algorithms for searching this large amount of data to quickly find the pages on which particular information resides.
- Electronic commerce: The ability of keeping the information (credit card nos, passwords, bank statements) private, safe & secure. Algorithms involves encryption/decryption techniques.

Components of an algorithm :-

- variables & values.
- Instructions.
- Sequences :- series of instructions.
- Procedures :- A named sequence of instruction we also use the following words to refer to a "procedure" : • Sub-routine
  • Module
  • Function
- Selections :- An instruction that decides which of 2 possible sequences is executed. The decision is based on True/False condition
- Repetition :- Also known as iteration/loop.
- Documentation :- Records what the algorithm does.

↳ A simple algorithm :-

- I/P : A sequence of 'n' nos
  - T is an array of 'n' elements.
  - T[1], T[2] -- T[n]
- O/P : The smallest no among them.

Eg :-
```
min = T[1]
for i = 2 to n do
{
  if T[i] < min
     min = T[i]
}
output: min
```

- Performance of this algorithm is a function of 'n'.

⟶ Algorithm as a Technology :-

Efficiency :- Different algorithms solve the same Problem often differ noticeably in their efficiency.

- These differences can be much more significant than difference due to hardware & software.

- Consider two sort algorithms :

(i) Insertion sort :- roughly takes time equal to $c_1 n^2$ to sort n-items, where c_1 is a constant that does not depends on 'n'. It takes time roughly proportional to n^2.

(ii) Merge sort :- roughly takes time equal to takes $c_2 n \log(n)$ to sort 'n'-items. where c_2 is also a constant that does not depends on 'n'. $\log(n)$ stands for $\log_2(n)$. It takes time roughly proportional to $n \log(n)$

- Insertion sort usually has a smaller constant factor than merge sort so that, $c_1 < c_2$

* merge sort is faster than insertion sort for large i/p size;

consider now:

- A faster computer 'A' running insertion sort against,
- A slower computer 'B' running merge sort.
- Both must sort an array of one million nos,

suppose,

- computer 'A' execute one billion (10) instructions per second.
- computer 'B' execute ten million (10) instructions per second.
- So computer 'A' is 100 times faster than computer 'B',

Assume that:

$c_1 = 2$ & $c_2 = 50$

(i) To sort one-million nos;

- computer 'A' takes;

$$\frac{2 \cdot (10^6)^2 \text{ instructions}}{10^9 \text{ instructions/second}} = 2000 \text{ seconds.}$$

- computer 'B' takes;

$$\frac{50 \cdot 10^6 \cdot \lg(10^6) \text{ instructions}}{10^7 \text{ instructions/second}} = 100 \text{ seconds,}$$

- By using algorithm whose running time grows more slowly, computer - B runs 20 times faster than computer - A.

(ii) For ten million nos:

- Insertion sort takes : 2.3 days,
- merge sort takes : 20 minutes,

→ Analysing algorithms :-

- It is process of analysing the problem-solving capability of the algorithm in terms of the time & size required (size of the memory for storage while implementation). The main concern of analysis of algorithm is the required time or performance.
- Analysing an algorithm means predicting the resources that the algorithm requires.
- Resources such as memory, communication bandwidth or computer hardware are of primary concern.
- but most often it is computational time that we want to measure.

↳ Complexity of algorithms :-

- We can determine the efficiency of an algorithm by calculating its performance.
- Following are the 2 factors that help us to determine the efficiency of an algorithm:

(i) Total time required by an algorithm to execute.

(ii) Total space required by an algorithm to execute.

- Thus, the 2 main considerations required to analyse the algorithm are:

i) **Time complexity** :- It is a function that describes the time taken by an algorithm to solve a problem.

- The time complexity of an algorithm is the amount of computer time it needs to run for completion.
- Big-O notation is used to express the time complexity of an algorithm.

ii) Space complexity :- It is a function that describes the amount of memory / space required by an algorithm to run.

- A good algorithm has minimum no of space complexity.

- The complexity of an algorithm is the function f(n) which gives the running time & space requirement of the algorithm in terms of the size 'n' of the i/p data.
- Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'.
- The function f(n), gives the running time of an algorithm, depends not only on size 'n' of the i/p data, but also on particular data.

Example :-

Algorithm	statements / Instructions
A	x = x+1
B	for a=1 to n step-1 x = x+1 LOOP
C	for a=1 to n step-1 for b=1 to n step-1 x = x+1 LOOP

- In step-A, there is one independent statement 'x = x+1' & it is not within any loop.
- Hence, this statement will be executed only once. Thus, the frequency count of step-A of the algorithm is 1.

⊖ In step - B , there are 3 statements out of which '$x = x+1$' is an important statement, As the statement '$x = x+1$' is contained within the loop, the statement will be executed 'n' no of times,

- Thus, the frequency count of step - B in algorithm is 'n',

⊖ In step - C , the inner & outer loop runs in 'n' no of times, thus, the frequency count is n^2.

Total $= 1 + n + n^2$,

↳ Types of Analysis of complexity :-

(i) Best case time complexity :- (least)

- An algorithm will take minimum amount of time to solve a particular problem. In other words, the algorithm runs for a short - time.
- The best case efficiency of an algorithm is the efficiency for the best case i/p of size 'n'.
- Because of this i/p, the algorithm runs the fastest among all the possible i/ps of same size.
- Best case does not mean the smallest i/p. It means the i/p of size 'n' for which the algorithm runs the fastest.
- To analyse the best case efficiency, we have to 1st determine kind of i/ps for which the count C(n) will be the smallest among all possible i/ps of size 'n',

Eg:- In case of sequential search :
the best case for lists of size 'n' is when their 1st element is equal to the search key.

- Bubble sort has a best case time complexity of (n)

(ii) Worst case Analysis :-

- If an algorithm takes maximum amount of time to execute for a specific set of i/P, then it is called worst case time complexity [execute for long time]
- The worst case efficiency of an algorithm is the efficiency for the worst case i/P of size 'n'.
- The algorithm runs the longest among all the possible i/Ps of the similar size because of this i/P of size 'n'.

Eg :- In sequential search :
If the search element key is present at the nth position of the list, then the basic operations & time required to execute the algorithm is more.
Thus, it gives the worst case time complexity represented as :

$$C_{worst}(n) = n$$

- Quick sort has a worst case time complexity of n^2.

(iii) Average case Analysis :-

- If the time complexity of an algorithm for certain sets of i/Ps are on an average, then such a time complexity is called average case time complexity.
- It provides necessary information about an algorithm's behaviour on a typical / random i/P.

Eg :- In sequential search,
- The probability of successful search is equal to 't' i.e, $0 \leq t \leq 1$.

Scanned with CamScanner

- The probability of the 1st match occurring in the i^{th} position of the list is the same for all values of 'i'.

(i) In case of successful search $\frac{t}{n}$ the probability of 1st match occurring in the i^{th} position of the list is for all values of 'i', & the comparision made by the algorithm is also 'i'.

(i) In case of unsuccessful search, the probability of first match is $(1-t)$.

- <u>Quicksort</u> has an average case time complexity of $(n * \log(n))$.

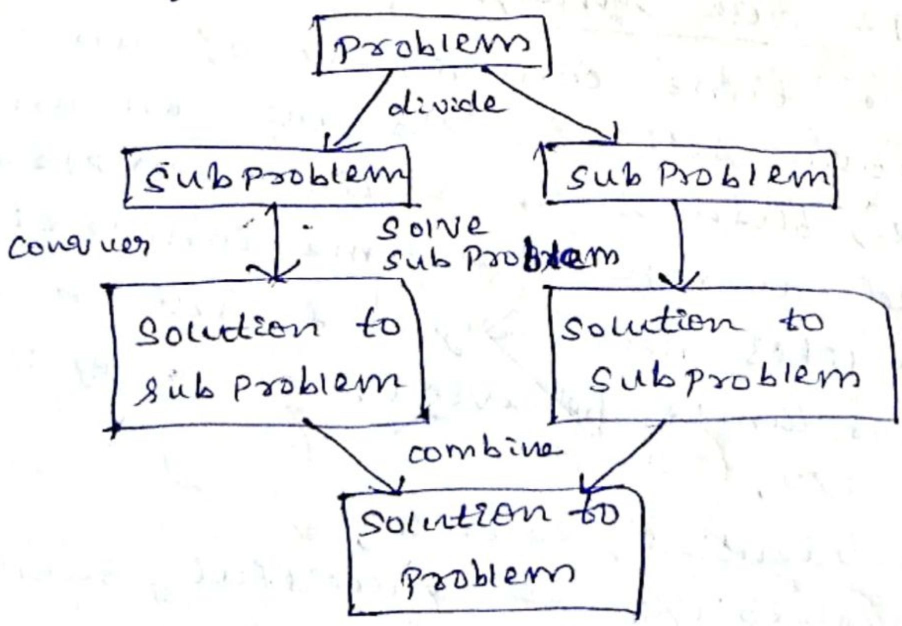
→ Designing algorithm :-

- It is a various design technique is avalable to design algorithm:

(i) <u>Divide & conquer :-</u> This method involves dividing the problem into sub-problem, recursively (individually) solving them, & then combining them for the final answer.

- It is a top-down approach.

<u>Eg :-</u> Binary search, quick-sort, merge sort.



(ii) Greedy technique :- In this method, at each step, a decision is made to choose the local optimum, without thinking about future consequences.

eg:- Fractional Knapsack, Activity selection.

- It is used to solve optimization problem: An obtimization problem is one in which for set of i/p values which are required either to be maximised (or) minimised.
- It always makes the choice looks best at a moment, to optize a given objective.
- It doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

eg:- selection sort.

(iii) Dynamic programing :- It is a bottom-up approach, we solve all possible small problems & then combine them to obtain solutions for bigger problems.

- This is particularly helpful when the no of copying subproblems is expontially large.

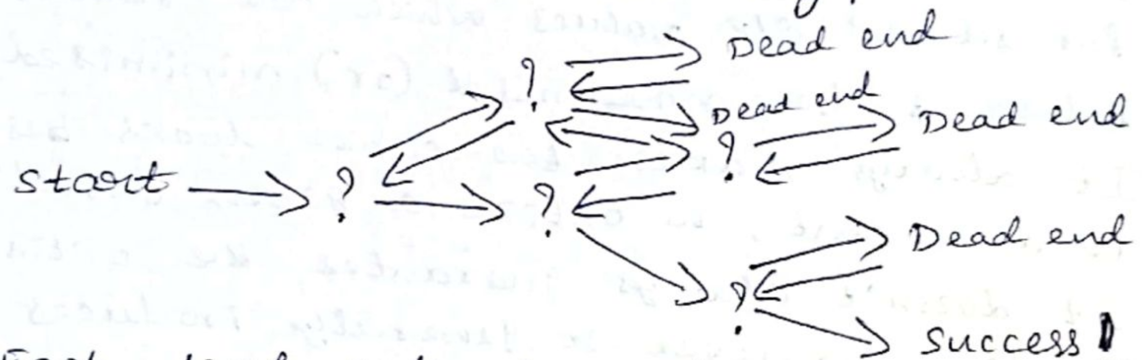
eg:- Insertion sort.

- It is similar to divide & conquer.
- The difference is that whenever recursive function calls with same results, instead of calling them again, we try to store the result in data structure in the form of table & retrieve the results from table.
- The overall time complexity is reduced.
- Dynamic means dynamically decide whether to call a function or retrieve values from the table.

eg:- 0-1 Knapsack, subset-sum problem.

(iv) Backtracking algorithm :- It is an optimiz-ation technique to solve combinational problems.

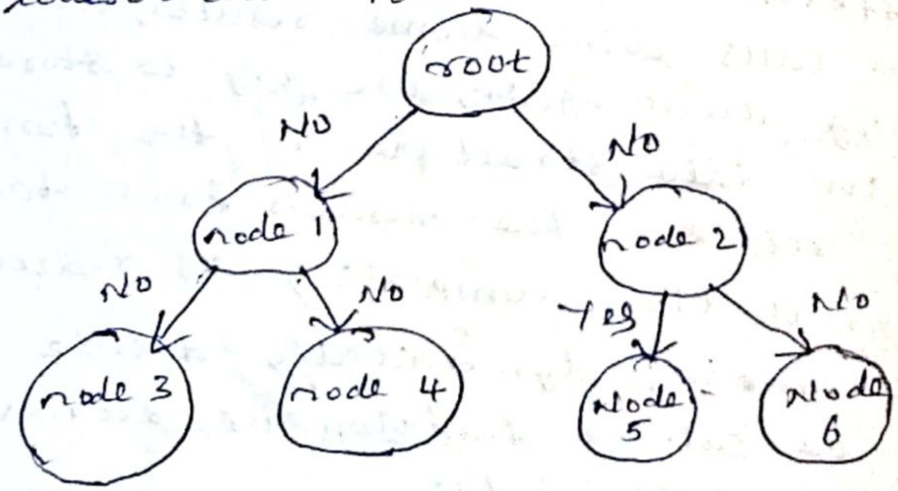
- It is applied to both programatic & real life problems.
- It is an algorithmic method to solve a problem with additional way.



- Each leaf node in a tree is parent of one or more other nodes.
- Each node in the tree, other than the root, has exactly one parent.
- If N is a goal node, return "Success".
- If N is a leaf node, return "failure".

(v) Branch & Bound :- It is 'n' optimization technique to get an optimal solution to the problem.

- It looks for the best solution given problem in the entire space of the solution.
- The purpose of this search is to maintain lowest-cost path to a target.



→ Growth of functions :- [Asymptotic Analysis]

- Algorithm's rate of growth (characterization) enables us to figure out an algorithm's efficiency along with the ability to compare the performance of another algorithm.
- I/P size matters as constants & lower order terms are influenced by the large sized of i/ps.
- once the i/p size 'n' becomes large enough, merge sort, with its $O(n \log n)$ worst-case running time, beats insertion sort, whose worst-case running time is $O(n^2)$.
- Although we can determine the exact running time of an algorithm.

→ Asymptotic Notations :-

- A problem may have various algorithmic solutions,
- In order to choose the best algorithm for a particular process, you must be able to judge the time taken to run two solutions & choose the better among the solutions.
- To select the best algorithm, it is necessary to check the efficiency of each algorithm.
- The efficiency of each algorithm can be checked by computing its time complexity.
- The asymptotic notations help to represent the time complexity in a shorthand way.
- It can generally be represented as the fastest possible, slowest possible or average possible.

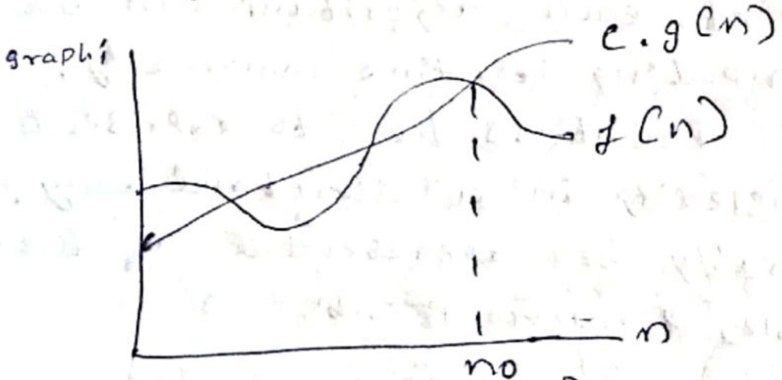
(5) Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

(i) Big-oh (O) notation:-

- The notation $O(n)$ is the formal way to express the upper bound of an algorithm's running time.
- It measures the worst-case time complexity.
- It calculates the maximum amount of time taken by an algorithm to compute a problem.
- It express the run-time in terms of – how quickly it grows relative to the i/p, as the i/p gets larger.
- $O(n)$, when passed 5 argument, it will take 5 times as long as when passed 1 argument.
- $O(n^2)$, when passed 5 argument, it will take 5^2 (25) times longer than when passed a single argument.
- If $f(n) = O.g(n)$, if there exists a positive no 'c' & positive integer 'n_0' such that: $\boxed{f(n) \leq c.g(n)}$

  for all : $\boxed{n \geq n_0}$

graph:



$f(n) = O.g(n)$

- If the equality that is $f(n) \leq c.g(n)$ holds good, we say that: $f(n) \in O.g(n)$

  $(f(n) = O.g(n))$

eg:- $f(n) = O \cdot g(n)$ $\quad n = 5$

$f(n) = n^2 \} 25$ $\quad 25 \leq 125$ (satisfied)
$g(n) = n^3 \} 125$

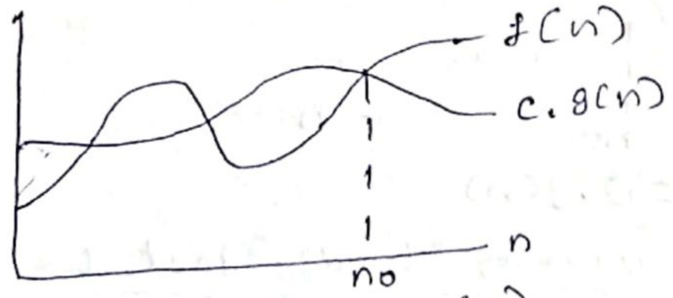
$f(n) = n^3 \} 125$ $\quad 125 \geq 25$ (not satisfied)
$g(n) = n^2 \} 25$

(ii) By - Omega (Ω) notation :-

- It describes which algorithm performs in the best-case time complexity.
- It provides the minimum amount of time taken by an algorithm to compute a problem.
- It gives the "lower bound" of the algorithm's run-time.
- If $f(n) = \Omega \cdot g(n)$, if there exists a positive integer 'no' & positive no 'c', such that : $\boxed{f(n) \geq c \cdot g(n)}$

$\forall \boxed{n \geq no}$

graph



$f(n) = \Omega \cdot g(n)$

- If the equality that is $f(n) \geq c \cdot g(n)$ holds good, we say that : $f(n) \in \Omega \cdot g(n)$
($f(n) = \Omega \cdot g(n)$)

eg:- $f(n) = n^3 \} 125$ $\quad 125 \geq 25$ (satisfied)
$g(n) = n^2 \} 25$

n=5

$f(2n+1) = \Omega\, 3n$
$f(2(5)+1) = \Omega\, 3(5)$
$10 + 1 = \Omega\, 15$
$11 = \Omega\, 15$
$f(n) \neq g(n)$
(not satisfied)

$f(n) = 2n+5$ | $n = 5$
$g(n) = 3n$

$2n+5 = \Omega\, 3n$
$2(5)+5 = \Omega\, 3(5)$
$10 + 5 = \Omega\, 15$
$15 \geq 15$
$f(n) = g(n) \longrightarrow$ satisfied

(iii) Big-Theta (Θ) notation :-

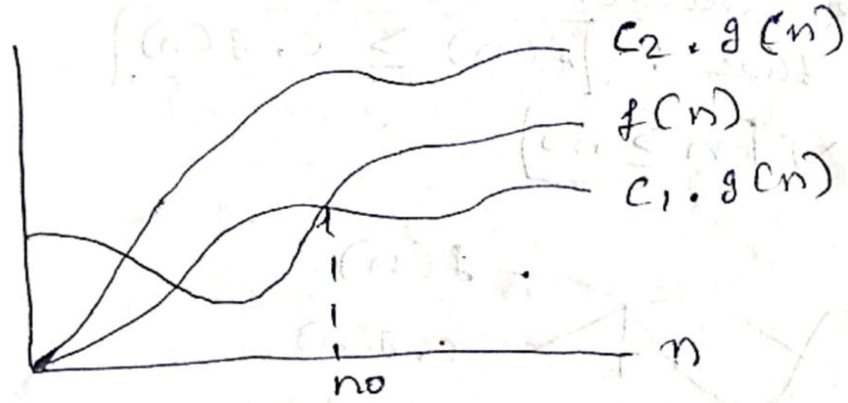
- It is used when the upper bound & lower bound of an algorithm are in the same order of magnitude.
- It is used to analysing the average case time complexity of an algorithm.
- $f(n) = \Theta . g(n)$, if there exists a positive integer 'n_0' & 2-positive constant 'c_1' & 'c_2' such that:

$$\boxed{c_1 . g(n) \leq f(n) \leq c_2 . g(n)}$$

$$\forall \boxed{n \geq n_0} \quad (n_0 > 1)$$

- The function $g(n)$ is both upper & a lower bound for function $f(n)$ for all values of 'n'.

graph:



$f(n) = \Theta . g(n)$

- If this equality holds good, we say that: $\boxed{f(n) \in \Theta . g(n)}$ (or) $\boxed{f(n) = \Theta . g(n)}$

eg:- $c_1 n^3 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2$

$n=5$, $c_1 125 \leq \frac{1}{2} 25 - 3(5) \leq c_2 25$

$125 \leq \frac{25}{2} - 15 \leq 25$

$125 \leq \frac{5}{2} \leq 25$ (Not satisfied)

$f(n)$		$g(n)$
$16n^2 + 30n^2 - 90$	n^2	$f(n) = \Theta(n^2)$
$7.2^n + 30n$	2^n	$f(n) = \Theta(2^n)$

eg:- $f(n) = 3n+2$

$g(n) = n$

$$c_1 . g(n) \le f(n) \le c_2 . g(n)$$

$c_1 = 1$
$c_2 = 4$

$n=4$,

$c_1 . g(n) \le f(n)$

$1. n \le 3n+2$

$1.4 \le 3(4)+2$

$4 \le 12+2$

$4 \le 14$

{$c_1 = 1$ is lower bound}

$f(n) \le c_2 . g(n)$

$3n+2 \le 4(n)$

$3(4)+2 \le 4(4)$

$12+2 \le 16$

$14 \le 16$

{$c_2 = 4$ upper bound}

(iv) Little-oh(o) notation :-

- For a given function $g(n)$, the set of little-oh, defined as;

$o(g(n)) = \{ f(n)$ is $\forall$ $c > 0 \equiv n_0 > 0$

such that $\forall$ $n \ge n_0$, we have;

$$\boxed{0 \le f(n) < c . g(n)}$$

- $g(n)$ is upper bound for $f(n)$ that is not asymptotically tight.

$$\boxed{f(n) = o . g(n)}$$

(v) Little-omega(ω) notation :-

- For a given function $g(n)$, set of little-omega, defined as;

$\omega . g(n) = \{ f(n) = \forall$ $c > 0 \equiv n_0 > 0$

such that $\forall$ $n \ge n_0$, we have;

$0 \le c g(n) < f(n)$

- $g(n)$ is lower bound for $f(n)$ that is not asymptotically tight.

$$\boxed{f(n) = \omega . g(n)}$$

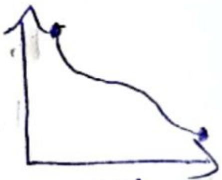
→ Standard notations & common functions:

↳ Monotonicity :-

- A function's increasing (or) decreasing tendency is called monotonicity on its domain.
- A function $f(n)$ is called montotonically increasing if for all x & y such that :

$$\boxed{x \le y \ , \ f(x) \le f(y)}$$

- A function $f(n)$ is called monotonically decreasing, if for all x & y such that :



$$\boxed{x \cancel{\ge} y \ , \ f(x) \ge f(y)}$$

- A function $f(n)$ is called strictly increasing, if for all x & y such that :

$$\boxed{x < y \ , \ f(x) < f(y)}$$

- A function $f(n)$ is called strictly decreasing, if for all x & y such that :

$$\boxed{x > y \ , \ f(x) > f(y)}$$

↳ Floor & ceiling functions :-

* **Floor :-** It is represented as floor($\lfloor x \rfloor$

- It gives the largest integer less than or equal to x [does not exceed 'x'].

eg :- floor(1.01) = 1 , floor(0) = 0,
floor(2.9) = 2 , floor(−3) = −3
floor(−1.1) = −2

* **ceiling :-** It is represented as ceiling ($\lceil x \rceil$

- It gives the smallest integer value greater than or equal to x.

eg :- ceiling(1.5) = 2, ceiling(0) = 0
ceiling(2) = 2 , ceiling(−3) = −3
ceiling(−1.1) = −1

Scanned with CamScanner

↳ Summation symbol :-

- Summation symbol is 'Σ'.
- Summation is the operation of combining a sequence of nos using addition.
- The result is the sum / total of all the nos.
- Apart from nos, other types of values such as, vectors, matrices, Polynomials & elements of any additive group ~~of~~ can also be added using summation symbol.

Eg :- consider a sequence : $a1, a2, a3 - a_{10}$.
The simple addition of this sequence :
$a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 + a_9 + a_{10}$
using mathematical notation we can shorten the addition, then the expression will be : $\boxed{\sum_{i=1}^{10} a_i}$

- 1 is 1st index
- 10 is last index.
- 'a' is variable.

↳ Exponent & Logarithm :-

*** Exponent :-** refers to no of times a number is multiplied by itself.

- Exponential function has the form $f(x) = a^x + B$ where, 'a' is the base, x is the exponent 'B' is any expression.
- If a is positive, the function continuously increases in value. As 'x' increases, the slope of the function also increases.

Eg :- $a^m = a + a + - - - ($m$ \text{ times})$

- For all real $a > 0$, m & n, we have the following identities :

$a^0 = 1$	$(a^m)^n = a^{mn}$
$a^1 = a$	$(a^m)^m = (a^n)^m$
$a^{-1} = \frac{1}{a}$	$a^m a^n = a^{m+n}$

* **Logarithms :-** It refers to how many times a certain number called the base is multiplied by itself to reach another no.

- A logarithm is an exponent.
- The logarithmic function is defined as:

$$f(x) = \log_b x.$$

- Here, the base of the algorithm is 'b'.

⊜ We shall use following notations:

$\lg n = \log_2 n$ (binary logarithm),
$\ln n = \log_e n$ (natural logarithm),
$\lg^k n = (\lg n)^k$ (exponentiation),
$\lg \lg n = \lg(\lg n)$ (composition.

Eg:- $y = \log_b x$ is equivalent to $b^y = x$.

- $\log_2^8 = 3$, since $2^3 = 8$
- $\log_{10} 100 = 2$, since $10^2 = 100$

↳ **Factorial :-** The symbol of factorial function is '!', '⌊'.

- The product (multiplies) a series of natural numbers that are in descending order from 1 to n.
- The factorial of a positive integer 'n' which is denoted by $n!$ (or) $\lfloor n$.

Eg:- $n! = n * (n-1) * (n-2) -- 2*1$

$4! = 4 * 3 \times 2 \times 1 = 24$

↳ **Fibonacci numbers :-** In fibonacci sequence, after 1st two numbers i.e, 0 & 1 in the sequence, each subsequent no in the series is equal to sum of previous two nos.

Eg:- 0, 1, 1, 2, 3, 5, --

3, 10, 13, 23, 36, --

- In mathematical terms; the sequence 'F_n' of fibonacci nos is defined as:

$$\boxed{F_n = F_{n-1} + F_{n-2}}$$

Chapter-2

Fundamental algorithms

1) Exchanging the values of 2 variables:-

- It is also called as swapping.
- It is a processing of exchanging the values of 2 variable with each other.

Algorithm:-

1) START
2) Read the value of num-1 & num-2
3) Assign values:
   Num-1 = num-1 + num-2
4) Num-2 = num-1 - num-2
5) num-1 = num-1 - num-2
6) print the value num-1 & num-2
7) STOP.

Tracing:- $num_1 = 5$, $num_2 = 10$

num-1 = 5 + 10 = 15
num-2 = 15 - 10 = 5
num-1 = 15 - 5 = 10

num-2 = 5
num-1 = 10

* Exchanging the given 2 nos using 3 variables:

1) START
2) Read the value of a & b, temp.
3) temp = a
4) a = b
5) b = temp
6) print the value of a & b:
7) STOP.

a = 5, b = 10
temp = 5(a)
a = 10(b)
b = 5(temp)

② COUNTING the no of digits in a given number :-

1) START
2) Read a number.
3) Initialise count = 0.
4) Repeat step-5 until number-0 otherwise step-7.
5) Divide number with 10.
6) count = count + 1.
7) write count.
8) STOP.

Tracing :- 123

- Count = 0
  $n = 123/10 = 12$
  count = count + 1
  = 0 + 1 = 1
- count = 1
  $n = 12/10 = 1$
  count = 1 + 1 = 2
- count = 2
  $n = 1/10 = 0$
  count = 2 + 1 = 3

∴ count = 3

③ Summation of a set of numbers :-

1) START.
2) Read the no.
3) Get a modulus/remainder of a no.
4) sum of remainder of no.
5) Divide by no - 10.
6) Repeat step-3 till greater than '0'.
7) STOP.

Tracing :-

$n = 123$
$n = 123 \% 10 = 3$

add = 3 | $\left.\frac{123}{10} = 3\right\}$ add = 6

1) START.
2) Read a number.
3) Initialise sum = 0.
4) while num != 0
5) num % 10.
6) Sum = sum + num.
7) num + sum
8) STOP

Tracing :-

num = 123, sum = 0

num = 123 % 10 = 3

sum = 0 + 3 = 3

6

num + sum = 6

num = 3 % 10 = 3

sum = ~~3~~ + 3 = 6

$\therefore \boxed{sum = 6}$

(4) Factorial computation :-

* using recursion :-

1) START.
2) Read a no.
3) Initialise f = 1.
4) if(num == 1) then
5) return num.
6) else f = num * fact(num - 1)
7) return f.

Tracing :-

f = 1, num = 4

f = num * fact(num - 1)

= 4 * fact (4 - 1)

= 4 * 3 * fact(3 - 1)

= 4 * 3 * 2 * fact(2 - 1)

= 4 * 3 * 2 * 1

= 24

* Without recursion :-

1) START.
2) Read the no.
3) Initialise variable $i=1$, $fact=1$.
4) if $i<=n$, go to step-5, otherwise go to step 8
5) $fact = fact * i$.
6) Increment 'i' by 1 $(i=i+1)$ & go to step-4
7) Print fact.
8) STOP.

Tracing :-

$i=1, f=1, n=4$

$fact = fact * i$

$fact = 1 * 1 = 1$ $(1+1=2)$

$fact = fact * i$

$fact = 1 * 2 = 2$ $(2+1=3)$

$fact = 2 * 3 = 6$ $(3+1=4)$

$fact = 6 * 4 = 24$ $(i<=n)(4<=4)$ STOP

5 Fibonacci sequence :-

* Using Recursion :-

1) START.
2) Read number.
3) Initialise $a=0$, $b=1$ & $i=2$, $sum=0$.
4) if $i<=n$, then go to step-5 6 7.
5) $sum = a+b$, print sum.
6) $a = b$
7) $b = sum$, i++
8) Print sum.
9) STOP.

Scanned with CamScanner

Tracing :-

$i=2, n=4, a=0, b=1$

$sum = a+b$

$sum = 0+1$

$sum = 1$

$a = 1(b)$

$b = 1(sum)$, $i++$

$i=3$, $sum = a+b$

$sum = 1+1$

$sum = 2$, $i++$

$i=4$	$sum = a+b$
$a=1(b)$	$sum = 1+2$
$b=2(sum)$	$sum = 3$

Fibonacci :-

0	1	2	3	4
0	1	1	2	3

* Without recursion :-

1) START
2) Read a number
3) Initialise $i=0, n$
4) $f(n) = f(n-1) + f(n-2)$
5) Repeat step-4 until 'i' is lesser than equal to 'n'.
6) Print number.
7) STOP.

Tracing :- $n=5$, $\boxed{i=0}$

0	1	2	3	4	5
0	1	1	2	3	5

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

$f(5) = f(4) + f(3) = 5 = (3+2)$

~~f(.) = f(.) + f(.)~~

$f(4) = f(3) + f(2) = 3 = (2+1)$

$f(3) = f(2) + f(1) = 2 = (1+1)$

$f(2) = f(1) + f(0) = 1 = (1+0)$

~~f(1) = f(0)~~

(6) Reversing the digit of an integer:-

1) START
2) Read the number 'n'.
3) Initialise rev = 0
4) Repeat step - (5) (6) & (7) until n != 0
5) set r = n % 10
6) set rev = rev * 10 + r
7) set n = n/10.
8) print rev.
9) STOP.

Tracing:- rev = 0 , $\boxed{n = 123}$

r = 123 % 10 = 3
rev = 0 * 10 + 3 = 3
n = 123/10 = 12

---

r = 12 % 10 = 2
rev = 3 * 10 + 2 = 32
n = 12/10 = 1

r = 1 % 10 = 1
rev = 32 * 10 + 1 = 321
n = 1/10 = 0

rev = 321

→ Base conversion :-

(1) Binary (octal(8), hexadecimal(16)) to decimal (10) :-

1) Let 'n' be no of digits in number.
2) Let 'b' be base of the number.
3) Let 's' be running total initially '0'.
4) For each digit in the number working from left to right i.e, 1 from 'n' & multiply the digit 'base' times (b^n) & add "it to 'S'.
5) when we all done with digits in the number, the decimal value will be 'S'.

eg: $1011_{(2)}$ $n=4$, $b=2$, $S=0$

$n=n-1$
$n=4-1$
$n=3$

$1 \times b^n = 1 \times 2^3$
$= 8$
$\boxed{S=8}$

2nd digit 0 : $n=3$, $n=n-1$
$n=3-1$
$n=2$

$0 \times b^n = 0 \times 2^2$
$= 0$
$\boxed{S=0} + 8 = 8$

3rd digit 1 : $n=2$, $n=2-1$
$n=1$

$1 \times b^n = 1 \times 2^1$
$= 1 \times 2$
$\boxed{S=2} + 8 = 10$

4th digit 1 : $n=1$, $n=1-1$
$n=0$

$1 \times b^n = 1 \times 2^0$
$= 1$
$\boxed{S=1} + 10 = 11$

(or)

$1 \times b^n + 0 \times b^n + 1 \times b^n + 1 \times b^n$

$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

$8 + 0 + 2 + 1$

$(11)_{10}$

2) Decimal (10) to Binary (2) :- (Octal (8), Hexadecimal (16))

1) Let 'n' be a decimal number.
2) Let 'm' be the number initially '0'.
3) Let 'b' be the base of the number that we are converting to.
4) Repeat until 'n' number becomes '0'.
   - Divide 'n' by 'b'. letting result in 'b' remainder be 'r'.
   - write the remainder 'r' as the left most digit of 'm'.
   - Let 't' be the new value of 'n'.

Scanned with CamScanner

Eg:-

$n = 45$
$m = 0$
$b = 2$

$\frac{n}{b} = \frac{45}{2} = 22$

$\boxed{r = 1}$, $\boxed{t = 22}$, $m = 1$, $n = 22$

$\frac{22}{2} = 11$, $r = 0$ (remainder), $\boxed{t=11}$, $m = 01$ & $n = 11$

$\frac{11}{2} = 5$, $r = 1$, $\boxed{t=5}$, $m = 101$ & $n = 5$

$\frac{5}{2} = 2$, $r = 1$, $\boxed{t=2}$, $m = 1101$ & $n = 2$

$\frac{2}{2} = 1$, $r = 0$, $\boxed{t=1}$, $m = 01101$ & $n = 1$

$\frac{1}{2} = 0$, $r = 1$, $\boxed{t=0}$, $m = 101101$ & $n = 0$

$\boxed{\begin{array}{cccccc} 1 & 0 & 1 & 1 & 0 & 1_{(2)} \\ 32 & 16 & 8 & 4 & 2 & 1 \end{array}} = 45$

or

$2 \,|\, 45$
$2 \,|\, 22 - 1$
$2 \,|\, 11 - 0$
$2 \,|\, 5 - 1$
$2 \,|\, 2 - 1$
$1 - 0$

$(101101)_2$

(3) Character to number:-

1) START
2) Declare number n, i, value=0;
3) Read character representation of an number 'n'
4) LOOP over the characters in 'n' from left to right do the following step: for i = 1 to n

   Value = value $\times$10 + ASCII value of i^{th} character − ASCII of '0'. ~~char( )~~

5) Print value.
6) End.

Characters	0	1	2	3	4	5	6	7	8	9
ASCII code	48	49	50	51	52	53	54	55	56	57

eg:- n = 725, i = 1, value = 0

value = 0 * 10 + ASCII of i^{th} char (7) - ASCII of 0

= 0 + 55 - 48

= 0 + 7

= 7

i = 2, value = 7 | value = 7 * 10 + ASCII of i^{th} char (2) - ASCII of 0

= 70 + 50 - 48

= 70 + 2

= 72

i = 3, value = 72 | value = 72 x 10 + ASCII of i(2) - ASCII (0)

= 720 + 53 - 48

= 720 + 5

= 725

④ octal to binary :-

eg:- $(73.26)_8 \longrightarrow (\ )_2$

$= (111/011 . 010/110)_2$

⑤ Binary to octal :-

eg:- $(101111 . 110)_2 \longrightarrow (\ )_8$

$= (57.6)_8$

⑥ Hexadecimal to binary :-

eg:- $(12AB.3D)_{16} \longrightarrow (\ )_2$

$= (0001/0010/1010/1011 . 0011/1101)_2$

⑦ Binary to Hexa-decimal :-

eg:- $(0011/1011/1011/1111 . 1011/1100)_2 \longrightarrow (\ )_{16}$

$= (3BBF.BC)_{16}$

Scanned with CamScanner

UNIT - 2 [Chapter - 1]

C - Programming

→ Structure of Program :-

- Documentation (comment) section.
- Link (Library function / predefined preprocessor)
- Definition (Declaration)
- Global declaration section

- main() function section
  { Local declaration part
  Executable part
  }

- Sub_program section
  { Function - 1
  — — — —
  Function - n
  }

  } user-defined function (section)

→ How to execute program :-

1) create in text-editor.
2) save with '.c' extension.
3) compile & Linker, Loader.
4) Run / execute if no errors found by compiler.

→ Elements of C :-

→ character sets :-

It denotes any alphabets, digits, special symbols, whitespace (blank space, newline(\n), horizontal tab(\t)) used to represent information in any language.

- In C, the character set is as follows;

- Alphabets : A, B, — — Z, a, b, — — z.
- Digits : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Special symbols : blank, !, @, #, %, /, ¬, +, :, ;, ', { }, ( ), ?, =, < >, ", ~, ., _

C-Tokens:- C-tokens are the basic building blocks in C language which are constructed together to write a C-program.

- Tokens are each & every smallest individual units in a C-program.

Types:

- Keywords (int, float, while, double, char) (reserved words)
- Identifiers (main, total) (name of program)(constant)
- Constants (Pi) (10, 20)
  - Primary — char - int
  - Secondary — Array, structure, union, enum, Pointer
- Strings ("Total", "hello").
- symbols ((), {})
- operators (+, /, -, *)

*** Variable as constant :-**

```
int const x = 100;
{ x = 2; ⊗
}
```

Identifiers :- Each program elements in a C program are given a name called identifier.

- Names given to identify variables, functions, constants, arrays, structures etc.

e.g.:- x = 10, here 'x' is a name given to variable

Rules :-

- 1st character should be alphabet, underscore & followed by either alphabets or digits.
- length can be upto 32 character.
- It can contain both uppercase & lowercase.
- special characters except underscore should not be used.
- It should be a single word without space.
- Keywords cannot be used as identifiers.

* **Keywords :-** (Reserved words) These are predefined words with special meanings which cannot be changed.
- Keywords cannot be used as variable names.
- There are 12 keywords in C language.

Eg:- auto, default, float, register, static.

* **Constants :-** Does not change during the execution of a program.
- supports several types of constants;

(i) Integer : 426, +786, -9000. } Numeric constants.
Float (real) : 426.0, 4.1, 41e8,

(ii) Single character : ch = 'A'. } character constants
String : "hello"

* **Variables :-** Each variable represents the name of a memory location in which a value can be stored.
- A variable value can be changed during program execution.

Eg:-

x	y	→ variable's name
10	20	→ Data value of variable.

* **Datatypes :-** are used to inform the type of value that can be stored in a variable.

Syntax :- datatype var1, var2 - - ;

Eg:- int a, b;

Types :-

1) Primitive types : int (4), float (4), char (1) & double (8).
2) User-defined types : struct, union, enum & typedef
3) Derived types : Pointer, array & function Pointer.

- short = (4), short_int = (2), long_int = (4).
- char → ch, s. str = 's', string = "str".
- float → pi = 3.142.
- double → long double (10) d = 1.123456789

* Quantifiers :- (modifiers) are keywords used to alter basic datatype
- It can be applied to basic datatypes are :
• signed, unsigned, short, long.

Syntax :- <modifiers> <datatype> <var1>, --

eg :- signed int a;

* Declaration of variable :- informs the compiler to reserve enough space for the variable in memory.

Syntax :- datatype variablename;

eg :- int a;

* Variable assignment :-

1) Assignment (=) [int a = 10;]
2) Keyboard [scanf("%d", &a);]

* Declaring symbolic constant :-

define Pi = 3.142;
define MAX = 100;

→ Arithmetic expressions :-

- It is a combination of variables, constants operand & operator arranged as per syntax of the language. eg :- A + B - C / d

Types :-

(i) Integer Arithmetic expression :- Result is in integer format.

eg :- $\frac{5}{2} = 2.5 = \boxed{2}$ / $10.5 \times 3 = 31.5 = \boxed{31}$

(ii) Real Arithmetic expression :- Result is in float format.

eg :- $\frac{5}{2} = \boxed{2.5}$ / $10.5 \times 3 = \boxed{31.5}$

(iii) Mixed Arithmetic expression :- Result is in both integer & float.

eg :- $\frac{5.0}{2} = \boxed{2.5}$, int = $\boxed{2}$ float = $\boxed{2.5}$

$\frac{5}{2.0} = \boxed{2.5}$

Evaluation of Arithmetic expression :-

- It is finding the answer from given arithmetic expression.

Syntax :- variable = expression;

- Rules for evaluation of Arithmetic expression:
- If 2 or more parenthesized expression are used, then the left most parenthesized expressions are evaluated.

eg:- $3 + (9 * 2) + (8 - 2) + 5$
(1) (2)

$3 + 18 + 6 + 5 = \boxed{32}$

- If parenthesis are nested, then inner most expressions are evaluated 1st.

eg:- $(2 + ((5 * 2) + 3) * 4)$
(1) (2) (3)

$= (2 + (10 + 3) * 4)$
$= (2 + 13 * 4) = (2 + 52) = \boxed{54}$

- The associative rule is applied when 2 or more operators of the same precedence appear in a expression [Left to right]

B O D M A S
{ }, ( ), (or) /, *, +, -

eg:- $x = 2 * ((8 \% 5) * (4 + (15 - 3) / (4 + 2)))$
$= 2 * ((8 \% 5) * (4 + 12 / (4 + 2)))$
$= 2 * (3 * (4 + 12 / 6))$
$= 2 * (3 * (4 + 2))$
$= 2 * (3 * 6)$
$= 2 * 18$
$= \boxed{36}$

* Relation expression :- <, >, <=, >=, !=

Syntax :- exp1 <relational operator> exp2

eg:- $(20 + 15) > 20/5$
$35 > 4$ ✓ T (1)

* Logical expression :- &&, ||, !

Syntax :- exp1 <logical operator> exp2.

eg:- $\underset{T}{5>2}$ && $\underset{T}{6<=8}$ } T (1)

$\underset{F}{!(8==8)}$ || $\underset{F}{3<2}$ } F (0)

↳ Type conversion (casting) :- It is a process of converting one type (form) of data into another type (form) of data, either lower to high (or) higher to low.

Types :-

(i) Implicit type conversion (widening) (Automatic) :- It is automatic conversion by computer itself.

- It converts lower type of data into higher type of data (promotion)
- when the variable (or) constant of higher type is converted to lower type, we call it as (Demotion)

eg :-

Promotion	Demotion
int i=5;	float f = 3.142
float f;	int i;
f=i;	i=f;
f = 5.0	i=3

char → short → int → long → long long → float → double → long double
(lower) (higher)

(ii) Explicit type conversion (Narrowing) (Manual) :- Explicitly can convert one datatype to another datatype using cast.

Syntax :- (datatype) expression;

eg:- x = float (5+6)

float (6/4) = 1.5

int (6/4) = 1

float $\left(\frac{6.0}{4}\right)$ = 1.5

eg:- To demonstrate type conversion:-

```
#include<stdio.h>
void main()
{ printf("5/2 = %d \n", 5/2);  --> 2
  printf("5.0/2 = %f \n", 5.0/2);  --> 2.5
  printf("5/2 = %f \n", (float) 5/2);  --> 2.5
                        Manual
  getch();
}
```

(built-in)
→ Library functions :- It is a group of statements used to perform specified task.

- Library functions are declared & defined in special files called "Header files" which we can reference in our programs using "include" directive.

Syntax :- #include<filenam.h>

eg:- #include<stdio.h>
<conio.h>
<math.h>
<string.h>

→ Arithmetic operators :-

- An operator is a symbol used to indicate a specific operation on variables in a program.

eg:- '+' is an add operator that adds two data items called operands.

operator :- operation performed 'with'.
operand :- operation performed 'on'.

operation type	Types of operation	operator
① unary	(one operand) • Increment & decrement	• ++, --, -
② Binary (2-operand)	• Arithmetic	• +, -, /, %, *
	• Logical	• &&, ||, !
	• Relational	• =, >, <, ≤, ≥, !=
	• Bitwise	• <<, >>, &, |, ^
	• Assignment	• +=, -=, =, *=, /=, %=
③ Ternary (3/more operand)	• conditional	• ?:,

eg:- int a, b, sum, sub, mul, div, mod;

```
Printf(" enter the values ");
scanf("%d %d", &a, &b);
sum = a+b; sub = a-b; mul = a*b;
        div = a/b; mod = a%b;
printf(" addition ", sum);
Printf(" Subtraction", sub);
Printf(" multiplication", mul);
printf(" Division ", div);
Printf(" Modulus", mod);
getch();
}
```

(1) Unary operator :- It requires only one operand / data item.

- Unary minus (-) :- writen before numeric value, variable or expression. (Negation of operand)

eg:- -5, -2.933, -x, b = 5 ; a = -b

* Increment (++) :- It adds 1 to its operand

eg:- n = n+1 (n++), 1+n = n (++n)

Postfix (n++) :- It increments the value of 'n' after its value is used.

eg:- a = 5, b = a++ = 6 (5+1)

sum = x++;
sum = x;
x = x+1;

prefix (++n) :- It increments the value of 'n' before it is used.

eg:- a=5, b = ++a = 6 (1+5)
a=6
b=6

sum = ++x;
x = x+1;
sum = x;

* Decrement (--) :- It subtracts 1 from its operand.

eg:- n = n-1 (n--), 1-n = n (--n)

Postfix (n--) :- In this case, value of operand is fetched before subtracting 1 from it.

eg:- a = 5
5 fetch ← a-- = 4 (5-1)

sum = n-1;
sum = n;
n = n-1;

Prefix (--n) :- In this case, value of operand is fetched after subtracting 1 from it.

eg:- --a (1-a) = 3

sum = --a;
a = a-1;
sum = a;

eg:- If a = 5
--a = 4
a-- = 3

Eg:- int a, b, x = 10, y = 20
a = x * y ++;
b = x * --y;
printf("a = %d, b = %d\n", a, b);

Soln:- a = x * y ++; | a = x * y ++
a = 10 * 20 (21) = 200 | 10 * 20 (21) = 200
y = 21 | b = x * (++y)
b = x * --y; | 10 * 22 = 220
b = 10 * 20 = 200

Eg:- int a = 5, b; | a = 5
b = a ++ —only→ | a ++ = 6
b = 5 (6) → a | ++ a = 7

Post-increment 1st assign value & then increment.

Operation:-

(1) (b = a ++):- 1st assign the value of 'a' that is (5) to 'b'.

- Incrementing the 'a' by value of 'a' i.e, (5) is (6).

(2) (b = ++a):- 1st increment value of 'a' (5) by '1' then it becomes (6)

- Updated value (6) of 'a' is assigned to 'b' i.e, b = 6

Eg:- int a = 5, b;
b = (++a) 1+5
b = 6

(2) Binary operator:- It requires 2 operands to work with [left to Right]

* Arithmetic:- add, subtract, multiplies, divide, modulus of 2 operands (+, -, *, /, %)

* Relational:- It is used to compare two values & the result of such operation is always logical either True (1) or false (0).

Eg:- <, >, <=, >=, ==, !=.

syntax :- exp1 < relational operator > exp2

1 = 1

(True)

1 > 2

(False)

eg :-
```
{
int a = 10; b = 5, c = 15;
printf(" a==b \n", a==b);    0 (False)
printf(" a>=b \n", a>=b);    1 (True)
printf(" (a+b)>c \n", (a+b)>c);    0
printf(" (a+b)>=c \n", (a+b)>=c);    1
printf(" c <= (b*2) \n", c<=(b*2));    0
getch();
}
```

* **Logical :-** It is used to connect two relational expressions (or) logical expressions.

- The result of logical expressions is always an integer value either true(1) or False(0).

syntax :- exp1 < logical operator > exp2

- (8 < 15) && (5 < 8) = True
  1 ... 1
- (8 < 15) || (5 < 8) = True
  1 ... 1
- !(8 > 15) = True / !(5 > 3) = False

eg :-
```
{ int a = 10, b = 15;
printf(" 5>3 && 3<10 %d ", %d);  1
printf(" 7<4 || 3<8 %d ", %d);  1
printf(" !(8==8) %d ", %d);  0
printf(" !(8==9) %d ", %d);  1
      negate o/p
getch();
}
```

Precedence & Associativity :-	Activity
! (logical NOT), ++, --, sizeof()	Right to left
*, /, %	Left to right
+, -	Left to Right
<, <=, >, >=	Left to Right
==, !=	Left to Right
&&	Left to Right
||	Left to Right
?:	Right to left
=, +=, -=, *=, /=, %=	Right to left
,	Left to Right

High → Low

* **Assignment operators:-** It is used to assign the result of an expression to a variable.

Syntax:- variable_name = expression (or) value;

eg:- a = 20;   b = (2+5)*3;

- 2 cases:-

i) **compound assignment:-** To increment/decrement within single expression. (+=, -=, *=)

Operator	eg	Meaning
=	a=10	10 is assigned to variable 'a'.
+=	a+=10	10 is added to a number (variable)
-=	a-=10	10 is subtracted from no (variable)
*=	a*=10	10 is multiplied with no
/=	a/=10	10 is divided with no

eg:-
```
{ int a=10, b=20, c=0;
  printf("c=a+b", +c);   -> 30
  printf("c+=a=" +c);    -> 40
  printf("c-=a=" +c);    -> 30
  printf("c*=a=" +c);    -> 300
  printf("c/=a=" +c);    -> 1
  printf("c%=a=" +c);    -> 5
  getch();
}
```

eg:- a=10*b*c;

ii) multiple assignment:- int j=k=m=0;

* **Conditional operator (?:):-** It is called ternary operators as they are use 3 expressions.

- It is also called as shorthand version of if_else construct.
- It contains a condition followed by 2 statements/values, if condition is true, 1st statement is executed otherwise 2nd statement is executed.

Syntax:- result = exp1 ? exp2 : exp3;

- If exp1 is true then exp2 is evaluated else exp3 is evaluated.

eg:- int x=10, y=5, z;
```
z=(x>y)? x : y;  -> o/p: x=10
z=(x<y)? x : y;  -> o/p: y=5
```

Scanned with CamScanner

{ int a = 8, b = 4;
clrscr();
8 < 4 ? printf("true") : printf("false");
}

[Not applied to float/double] [manipulate data at bit level]

* **Bitwise operator :-** Corresponding bits of both operands are combined by the usual logic operations [used to test the bits, shifting them to right or left.

Operator	Example	meaning
& (AND)	X & Y	O/P : 1 if both i/ps are 1
| (OR)	X | Y	O/P : 1 if either i/ps are 1
^ (XOR)	X ^ Y	O/P : 1 if i/ps are different
~ (complement)	~X	Each bit is reversed
<< (shift left)	a << 2	moves the bits to left, it discards
	1010 → 1000 (<<)	the far left bit & assign 0
	1010 → 0010 (>>)	to right most bit.
>> (shift right)	a >> 2	moves the bits to right, it discard far right bit & assign 0 to left most bit.

Eg:- printf("12 & 8 = %d", 12 & 8); → 8
printf("8 | 4 = %d", 8 | 4); → 12
printf("8 ^ 4 = %d", 8 ^ 4); → 12
printf("~16 = %d", ~16);
printf("10 >> 2 = %d", 10 >> 2);
printf("10 << 3 = %d", 10 << 3);

→ **I/P & O/P functions :-** I/P function is used to read i/p from user through standard i/p device (keyboard) O/P function is to display O/P in the monitor.

- The console input/output functions are classified:

1) **Formatted I/O function :-**

* **Formatted I/P function :-** (scanf()) It is used to read i/p through standard i/p device.
• stores them at the address of variable in memory

syntax :- scanf("format specifiers", address of variable);

eg :- scanf("%d", &a);   & → address_of operator

format specifiers :
• %d → int, %f → float, %c → single character,
%s → string, %ld → long int, %lf → double.

⊙ Field width specification for inputting integers :
• scanf("%3d", "%3f", &a, &f);   a = 12 (12), f = 12 (12.000)

• scanf("%2d %4d %d", &a, &b, &c);
printf("a = %d; b = %d; c = %d", a, b, c);
a = 12   1234 _ 567 _ 89 | c = 567
b = 34   123 456 789 | 12 34 56 789

• Field width specification for i/puting characters:
%ws & %wc are two specifiers.

⇒ To read a character:

```
{
 char ch; string str;
 printf("enter a character");
 scanf("%s" &str);
 printf("entered character", ch);
}
```

⇒ character to ASCII:-

```
{
 char ch;
 printf("enter a character");
 scanf("%c", &ch);
 printf("entered character", ch);
 printf("ASCII value of entered character: %d", ch);
 getch();
}
```

⇒ ASCII to character:-

```
{
 int ch = 65;
 printf("character having ASCII value 65 is: %c", ch);
 getch();
}
```

⇒ read a string:-

```
{
 string str; / char name[20];
 printf("enter a string");
 scanf("%s", &str);
 printf("entered string", str);
 getch();
}
```

→ Blank space encounters

o/p:- Skyword BOOKS ✗

o/p:- ("%13c") → Skword BOOKS ✓

("%4s") → Skyw

("%[^\n]") → skyward Book

- %[a-z] → To read all small (lower) case
- %[A-Z] → To read all upper case.
- %[A-Z, 0-9] → To read upper case & nos.
- %[a-z, 0-9] → To read lower case & nos.
- %[a, e, i, o, u] → To read vowels.

* Formatted o/p function:- This is a function which transfers the data in various formats to the o/p device (monitor).

Syntax:- printf("format string", list of variables);

Eg:- printf("The no is %d", a);

* format string includes:

- User specified strings which will be printed as it is.
- Format specifier like %d, %c, %f etc along with optional field width.
- Escape sequences (\n, \t, \b)

eg:- printf ("The sum of %d, %d, %d", a, b, c);
printf ("The sum of %d, %d, %d \n", a, b, c);
printf ("enter the no. %d, %c, %s", a, b, c);
printf (" entered input %2d, %2f, %3f", a, b, c);
o/p:- entered input: 1 2.00 3.000

- **Field width justification:-**

syntax:- %wc (right justified) (right to left)
%-wc (left justified) (left to right)

w - Total no of column (width) to be printed (total width)

C - Format specifier (%d, %c, %f --).

eg:- • Let a = 254;
printf ("%5d", a); [ ][ ][2][5][4] right to left
printf ("%-5d", a); [2][5][4][ ][ ] Left to right
printf ("%2d", a); [2][5][4]

- Let a = 254768;
printf("%5d", a); [2][5][4][7][6][8]
Extra space

printf (" %06d", 1234); [0][0][1][2][3][4]

- Let a = 86.123;
printf ("%7.2f", a); [ ][8][6][.][1][2]
printf ("%-7.2f", a); [8][6][.][1][2][ ][ ]
printf ("%7.3f", a); [8][6][.][1][2][3]

- Let a = "COMPUTERSCIENCE";
printf ("%s", a); [C][O][M][P][U][T][E][R][S][C][I][E][N][C][E]
- Let a = 'A';
printf ("%4c", a); [ ][ ][ ][A]
printf("%-3c", a); [A][ ][ ]
- Let a = "COMPUTER"
printf ("%s", a); [C][O][M][P][U][T][E][R]
printf ("%5s", a); [C][O][M][P][U][T][E][R]
(when space is less, it will ignore width).

printf("%12S", a); [| | | | C | O | M | P | U | T | E | R]
printf("%12.6S", a); [| | | | | | C | O | M | P | U | T]
printf("%-12S", a); [C | O | M | P | U | T | E | R | | | |]

→ $\boxed{\%w.xf}$ → How many place after decimal

$\boxed{\%-w.xf}$

- a = 5432.123 → 8

printf("%f", a); [5 | 4 | 3 | 2 | . | 1 | 2 | 3]

printf("%12.2f", a); [| | | | | 5 | 4 | 3 | 2 | . | 1 | 2]

printf("%-12.2f", a); [5 | 4 | 3 | 2 | . | 1 | 2 | | | | |]

printf("%3.1f", a); [5 | 4 | 3 | 2 | . | 1]

printf("%e", a); [5 | . | 4 | 3 | 2 | 1 | 2 | 3 | e | 0 | 3]
exponent

printf("%13.2e", a); [| | | | 5 | . | 4 | 3 | e | + | 0 | 3]

printf("%-13.2e", a); [5 | 4 | 3 | e | + | 0 | 3]

- Field width specification to print string & characters:-

Syntax:- $\boxed{\%w.xs}$ Right
$\boxed{\%-w.xs}$ Left

- a = SRIKANTH

printf("%s", a); [S | R | I | K | A | N | T | H]

printf("%5s", a); [S | R | I | K | A | N | T | H]

Extra space
no truncation

printf("%12S", a); [| | | | S | R | I | K | A | N | T | H]

printf("%-12S", a); [S | R | I | K | A | N | T | H | | | |]

printf("%12.6S", a); [| | | | | | S | R | I | K | A | N]

printf("%-12.6S", a); [S | R | I | K | A | N | | | | | |]

printf("%12.0S", a); [| | | | | | | | | | |]
No characters

printf("%.6S", a); [S | R | I | K | A | N]
("%0.6S", a);
("%-.6", a);

ii) Unformatted input/Output function:-

* Unformatted i/p function:-

- getchar() :- It reads only a single character (one) at a time from keyboard & assigns that to the variable.

Syntax:- getchar(); / ch = getchar();

Eg:- char ch;
ch = getchar();

- getch():- & getche() are used to input only one character at a time through keyboard & they do not require to press enter key after i/p of a character.

- It will supply data to a program immediately without waiting for enter key to be pressed.

Syntax:- getch()

Eg:- char val;
val = getch();

- getche():- It will display character i/p while entering character.

Syntax:- getche()

Eg:- char val;
val = getche();

- gets() (puts) :- Read (write) a string or character array.

Syntax:- gets(array);

Eg:- gets (puts) (array); / gets(str);

* Unformatted o/p function:-

- putchar():- This function is used to print one character on the screen.

Syntax:- ~~char choice = 'Y';~~
putchar(variable_name);

Eg:- char choice = 'Y';
putchar(choice); putchar('A');

- putch():- This function displays single character through the standard o/p device like monitor.

Syntax:- putch(variable_name);

Eg:- char choice = 'Y';
putch(choice);
putch('A');

→ Variable length arguments list :-

- It is a programming construct that allows programers to pass n-no of arguments to a function.
- It is also known as var-tags.
- The function with varying no of arguments is defined by using a trailing ellipsis (...) in the argument list, declaring that there may be additional arguments, no & type unspecified.

Syntax:- return-type function-name(parameter-list, int num, ...);

- The library function like printf() & scanf() accept any no of arguments passed.

Eg:- int printf(char *format, arg1, arg2, ...);

• you can pass 'n' no of arguments to printf.

- <stdarg.h> headerfile defines libraries to handle variable nos of arguments.
- preprocessor macros included from are used to access the argument list for this kind of function:

• va-list :- Data type to define a va-list type variable.

• va-start :- Used to initialize va-list type variable.

• va-arg :- Retrieves next value from the va-list type variable.

• va-end :- Release memory assigned to a va-list type variable.

→ Control flow descission statements :-

- Descission making control statement (Selection) (conditional)
- Branching (loop) control statement

i) Descission making control statement :-

1) Simple - if statement :- It is one-way branching statement of decission-making statement.

- It is used to decide whether a certain statement will be executed or not.
- If a certain condition is true then a block of statement is executed otherwise not.

Syntax :-
```
if (condition)
{
  statement;
}
  statement-x;
```

Eg:-
```
{ int a,b;
  clrscr();
  printf("enter 2 values");
  scanf("%d %d", &a,&b);
  if(a>b)
{ printf("a is greater than b", a); }
  if(b>a)
{ Printf("b is greater than a", b); } }
```

2) If - else statement :- It is Two-way branching statement.

- It executes a block of code, if a specified condition is true, then if-block is executed & if it is false, then else-block of code is executed.

Syntax :-
```
if(condition)
{
  statement-1;
}
else
{
  statement-2;
}
```

Eg:-
```
#include <stdio.h>
#include <conio.h>
void main()
{ int a,b;
  clrscr();
  Printf("enter 2 values");
  Scanf("%d %d", &a,&b);
  if(a>b)
  {
   printf("a is greater", a);
  }
  else
  {
   Printf("b is greater", b);
  }
  getch();
}
```

③ **Nesting of if-else statement :-** When a if-else statement is present inside the body of another 'if' or 'else' then this is called nested if-else statement.

Syntax:-

```
if(condition-1)
{
  if(condition-2)
  {
    statement-1;
  }
  else
  {
    statement-2;
  }
}
else
{
  if(condition-3)
  {
    statement-3;
  }
  else
  {
    statement-4;
  }
}
```

outer if's if block

else block of inner if

outer if's else block

inner 'if'

else block of inner 'if'.

Scanned with CamScanner

* Different form of 'nested-if' statement:-

i) 'if' statement within another 'if' stmt:

```
if(condition-1)
{ if(condition-2)
{
 statement;
}
}
```

ii) Nesting of 'if-else' within an 'if' statement:

```
if(condition-1)
{
 if(condition-2)      -- if
 {
  statement;
 }
else
{
 statement;
}
}
```

iii) Nesting of 'if_else' within an 'if' block of 'if_else' statement:

```
if(condition-1)
{
 if (condition-2)      -- Inner if_else
 {
 stmt;
 }
 else
 {
 stmt;
 }
}
else                   -- outer if_else
{
stmt;
}
```

iv) Nesting of 'if_else' statement within an 'else' block of 'if_else' statement:

```
if(condition-1)
{ statement-1;
else
{ if(condition-2)
 { statement-2; }
 else
 { statement-3; }
}
}
```

Scanned with CamScanner

i) Nested of if_else within an if block & else block of if_else statement:

```
if(cond-1)
{
  if(cond-2)
  {
    stmt-1;
  }
  else
  {
    stmt-2;
  }
else
{
  if (cond-3)
  {
    stmt-3;
  }
  else
  {
    stmt-4;
  }
}
```

if_else

if_else

Eg:-

```
int a,b,c;
clrscr();
printf("enter values of a, b & c");
scanf("%d %d %d", &a,&b,&c);
if(a>b)
{
  if(a>c)
  {
    printf("a is greater than c", a);
  }
  else
  {
    printf("c is greater than a", c);
  }
}else {
  if(b>c)
  {
    printf("b is greater than c", b);
  }
  else
  {
    printf("c is greater", c);
  }
}
```

④ Else-if ladder :- It is multiple branching statement which contains 2 or more else-if stmt.
- It executes one condition from multiple stmts.
- In this, as one of the conditions become true, the statement associated with that is executed, and the rest of the ladder is bypassed.
- If none of the conditions is true, then the final else statement will be executed.

Syntax:-

```
1st if  <--- if (cond-1)
             {
               stmt-1;
             }
2nd if  <--- else if (cond-2)
             {
               stmt-2;
             }
               =
else break <--- else
             {
               stmt x;
             }
```

Eg:-

```
float percentage;
clrscr();
printf("enter the marks");
scanf("%f", &percentage);
percentage = obtained marks / 600;
if(percentage >=75)
{ printf("Distinction"); }
else if (percentage >=60)
{ printf("First class"); }
else if (percentage >= 50)
{ printf("Second class"); }
else if (percentage >=40)
{ printf("Third class"); }
else if (percentage <=40)
{ printf("Fail"); }
getch();
}
```

⑤ Switch statement :- It is multiple branching statement.

- It executes one statement from multiple conditions directly.
- It is like else-if ladder, but checks all condition.
- break & default are optional.

Syntax:- switch(exp)

```
{
    case label-1 : stmt-1;
                   break;
         =          =
    case label-n : stmt-n;
                   break;
    default : default statement;
              break;
}
stmt-x;
```

- No two values of labels should be same.

Eg:- Vowels or consonants :-

```
#include<stdio.h>
void main()
{
    char ch;
    clrscr();
    printf("enter any character");
    scanf("%c", &ch);
    switch(ch)
    {
        case 'a' : printf("vowel"); break;
        case 'e' : printf("vowel"); break;
        case 'i' : printf("vowel"); break;
        case 'o' : printf("vowel"); break;
        case 'u' printf("vowel"); break;
        default : printf("consonants");
    }
    getch();
}
```

Nested switch:- It refers to switch statements inside of another switch statements

syntax:-

```
switch(ch)
{
   case 1 : switch(a);
            {
              case 1
              case 2;
            }
   case 2: statement - 1;
           break;
}
```

eg:- Arithmetic calculation using **switch**:-

```
#include<stdio.h>
void main()
{
 int a, b, c, ch;
 clrscr();
 pf("\n 1. Addition");
 pf("\n 2. Subtraction");
 pf("\n 3. Multiplication");
 pf("\n 4. Division");
 pf("\n 5. Largest of 2 no");
 pf("\n enter your choice");
 sf("%d", ch);
 if(ch<=5 && ch>=0)
{ pf("enter 2 no");
  sf("%d%d", &a, &b);
} switch(ch)
{ case 1 : c=a+b;
           pf("Addition", c);
           break;
  case 2 : c=a-b;
           pf("subtraction", c);
           break;
  case 3 : c=a*b;
           pf("Multiplication", c);
           break;
  case 4 : c=a/b;
           pf("Division", c);
           break;
  case 5 : if(a>b)
           {
             largest=a;
           }
           else if(b>a)
           {
             largest=b;
           }
           else
           {
             pf("a & b are equal");
           }
           break;
  default : pf("Invalid choice");
            break;
}
getch();
}
```

* Conditional Operator in switch :-

- conditionals are expressions that evaluate to either true or false.
- Used to determine program flow.

Syntax :-	

if(condition)
  stmt-1;
else
  stmt-2;
	condition ? stmt 1 : stmt2

eg:- if(num % 2==0)
  printf("even");
else
  printf("odd");
	(num % 2 == 0) ?
	  printf("even") :
	  printf("odd");

eg:- if (a>b)
  printf("large = a");
else
  printf("large = b");
	(a>b) ? printf ("a") :
	  printf("b");

(ii) Branching (loop) control statement :-

Decision	Loop
- Execution only once	- Execution many times.
- Execution is until condition is true.	- Execution is until condition is false.
- It is certain statement to be executed once.	- It is certain statement to be executed again & again iteratively.
eg:- if, if_else, else_if ladder, switch	- eg:- while, do-while, for, nested-for.

(1) **While-loop :-** It is also known as Entry control loop construct.

- It is pre-tested loop construct.
- In this we can execute a set of statements as long as a condition is true.

Syntax :-
```
while(condition) → True
{
    stmt-1;
      =
    stmt-n;
}
```

Eg :- W.A.P to add 1 to 10 :
```
#include <stdio.h>
void main()
{
  int i=1, sum=0;
  clrscr();
  printf("Printing numbers");
  while (i<=10)
  {
    printf("%d", i);
    sum = sum + i;
    i++
  }
  printf("sum of 10 no is =%d", sum);
  getch();
}
```

O/P :- 1 2 3 4 5 6 7 8 9 10
sum of 10 no is : 55

Eg :- Factorial :-
```
while (n>=1)
{
  fact = fact * n;
  n--;
}
printf("Factorial of no is %d", fact);
```

n = 3

(3 >= 1) ✓ | fact = fact × n = 1 × 3 = 3 | n-- n = 2

(2 >= 1) ✓ | fact = 3 × 2 = 6 | n-- n = 1

(1 >= 1) ✓ | fact = 6 × 1 = 6 | n-- n = 0

eg:- Reverse of a given no:-

```
while (num > 0)
{
  digit = num % 10;
  num = num / 10;
  rev = rev * 10 + digit;
}
printf(" Reverse of no is %d", rev);
```

Tracing:-

① (num > 0)
(321 > 0) ✓
digit = 321 % 10 = 1
num = 321 / 10 = 32.
rev = 0 × 10 + 1
0 + 1 = $\boxed{1}$

② (32 > 0) ✓
digit = 32 % 10 = 2
num = 32/10 = 3
rev = 1 × 10 + 2
= 10 + 2
= $\boxed{12}$

③ (3 > 0) ✓
digit = 3 % 10 = 3
num = 3 / 10 = 1
rev = 12 × 10 + 3
= 120 + 3
= $\boxed{123}$

↳ Infinite loop:-

eg:-
```
a = 3;
while (a <= 0);
{
  a++;
}
```

```
while (1)
{
printf("hello");
}
```

Note:- In this piece of code, while stmt l... a semicolon which means that, it is end o... while.

- It means that as long as a <= 5 it is not do anything, it waits infinitely till the condition become false which will never happen, hence it gets into infinite loop.

② do-while loop :- It is also known as Exit - controlled loop construct.

- It is post tested loop construct.
- In while loop condition is testing in the beginning, if condition becomes false from (for) the 1st time, the body of the while loop will not be executed even once.
- But, sometimes we required to execute body of the statement ~~atleast~~ once even though the condition is 'false' for 1st time, In such situation, the do - while loop is used.

syntax :-

```
do
{
  statement-1;
      =
  statement-n;
}
while(condition);
```

eg:-

```
do
{
  printf("%d", i);
  i--;
}
while(i>0);
```

false to stop execution

→ W.a.P to print nos from 1 to 5 :

```
#include<stdio.h>
void main()
{
  int i=1;
  clrscr();
  printf("print nos from 1 to 5");
  do
  {
    printf("%d", i);
    i++
  }
  while(i<=5)
  getch();
}
```

O/P :-		
1	i=1	1<=5 i++
2	i=2	2<=5 i++
3	i=3	3<=5 i++
4	i=4	4<=5 i++
5	i=5	5<=5

Scanned with CamScanner

↳ W.A.P to demonstrate execution of do-while loop exactly once :

```
{
  int i = 1;
  do
  {
    printf("%d", i);
    i++;
  }
  while(i<1);
}
```

↳ W.A.P to display the digits & find the sum of digits in the no :

```
#include<stdio.h>
void main()
{
  int num, sum=0, digit;
  clrscr();
  printf("enter the digits ");
  scanf("%d", &num);
  do
  { digit = num % 10;
   printf("%d",digit);
   sum = sum +digit;
   num = num / 10;
  }
  while (num!=0);
  printf("sum of digits = %d", sum);
  getch();
}
```

Tracing :-

sum = 0 , num = 345

digit = 345 % 10 = 5

sum = 0 + 5 = 5

num = 345 / 10 = 34

num = 34

digit = 34 % 10 = 4

sum = 5 + 4 = 9

num = 34 / 10 = 3

num = 3

digit = 3 % 10 = 3

sum = 9 + 3 = 12

num = 3 / 10 = 0

O/P :-

5
4
3

3 + 4 + 5 = 12

③ **for loop :-** It is modified version / better version of while loop.

It is pre-tested nor post-tested.

It is fixed execution loop.

When we want to execute certain statements for certain no of times.

In this variable initialisation is done in for statement itself directly.

```
        (initialisation) (condition)  (inc/dec)
for ( exp1 ;  exp2 ;  exp3 )
                               (updation)
{
   statement - 1;
       =
}
   statement - x;
```

Syntax

eg :-

```
for (i = 2 ; i <= 10 ; i = i + 2)
{
   Print ("%d", i);
}
```

O/P :-

2
4
6
8
10

eg :-

```
float f;
for (f = 2.5 ; f <= 10.5 ; f + 10)
{
   Printf (f);
            2.5
}
```

O/P :-

2.5

eg :-

```
int i;
i = 1;
for ( ; i <= 5 ; i++)
{
   Printf ("%d", i);
}
```

eg :-

```
i = 1;
for ( ; i <= 5 ; )
{
   Print (i);
   i++;
}
```

Eg:- i=1;

```
for( ;  ; )
{
 Printf("%d", i)
   i++;
}
```

→ break;

[infinite for loop]

[To stop infinite for loop]

Eg:- i=1;

```
for( ;  ; )
{
 if(i==6)
 break;
 Printf("%d", i);
 i++;
}
statement - x;
```

False

Eg:- for(i=1, j=1; i<=10; i++; j++)

```
{
  Printf("%d %d", i, j);
}
```

Eg:- W.A.P to check prime or not using for loop

```
# include <stdio.h>
void main()
{
 int i=0, n, temp=0;
 printf(" Please input a no ");
 scanf("%d", &n);
 for (i=2, i<=(n/2); i++)
 {
  if(n%i==0)
  {
   temp=1;
   break;
  }
 }
```

```
if(temp==1)
printf(" given no is not prime");
else
printf(" given no is prime");
getch();
}
```

O/P:- please input a no ; 12
given no is not a prime.

→ Jumps in loops :-

- Jumps is used to skip certain part of statements (execution) of loop before the condition becomes false.
- It can be used to jump from one statement to another within loop as well as outside the loop.
- Jump can be achieved by using break / goto & continue statement.
- break & goto used to terminate the execution of a statement (jump out of the loop)
- continue is used to skip certain part of loop.

⊙ Jump can be used in loop construct like while, do-while & for along with if-stmt.

↳ break:-

i) Jump in while loop:

Syntax:-

```
while(condition)
{
  if(condition)
  break;
  =
}
statements;
```

True

ii) do-while loop:

Syntax:-

```
do
{
  - -
  if(condition)
  break;
  - - -
}
while(condition);
statements;
```

True

iii) for loop:

```
for(exp1; exp2; exp3)
{
  if(condition)
  break;
  --
}
statements;
```

True

e.g:- break

```
i=1;
while(i<=5)
{
  if(i==3)
  break;
  printf("%d", i);
  i++;
}
```

False / True

o/p:- 1
2

e.g:- break

```
i=1;
for( ; ; )
{
  printf("%d", i);
  i++
  if(i==6)
  break;
}
```

F / T

o/p:- 1 2 3 4 5

Jumps in continue:- Syntax:

i) using while:

```
while(condition)
{
  ---
  if(condition)
  continue;
  --
}
statements;
```

T

ii) do-while:

```
do
{
  if(condition)
  continue;
  =
}
while(condition);
statements;
```

T / F

iii) for loop:

```
for(exp1; exp2; ex
{
  if(condition)
  continue;
  --
}
statements;
```

T

e.g:-

```
for(i=1; i<=5; i++)
{
  if(i==2)
  continue;
  printf("%d", i);
}
```

T / F

o/p:- 1 3 4 5

e.g:-

```
sum=0;
for(i=1; i<=100; i++)
{
  if(i%2==0)
  continue
  sum = sum+i;
}
printf("sum of no=%d", sum);
```

T / F

o/p:- 1 3 5 7 — —
99

→ **goto statement:-** It allows us to transfer the control of the Program to specified label or location.

- It allows unconditional branching.
- It is an identifier because of label name used in goto statement.
- Goto statements + label.

Syntax:-

```
          loc
        (source)
goto  labelname;

labelname : statements;
(destination)
  location
```

where, labelname is not a case-sensitive.
& source labelname & destination labelname should be same.

Eg:- W.a.P to check the no is even or odd using goto;

```
#include<stdio.h>
void main()
{
 int n;
 clrscr();
 Printf("enter the number");
 scanf("%d", &n);
{ if(n%2==0)
  goto even;
 else
  goto odd;
}
  even : Printf("even");
  odd  : printf("odd");
 getch();
}
```

→ **Nested loops :-** It refers to having any of loops nested.

- It is used to declare a loop inside with in another loop.
- The nesting is necessary when a set of statements are to be executed as long as 2 (or more) different conditions are to be satisfied.

i) **Nesting of for loop :-**

```
for(exp1; exp2; exp3)  --> outer for loop
{
  for(exp1; exp2; exp3) --> inner for loop
  {
    - -
  }
}
```

ii) **Nesting of while loop :-**

```
while(condition)  --> outer while
{
  - -
  while(condition) --> Inner while
  {
    - -
  }
}
```

iii) **Nesting of do-while loop :-**

```
do
{
 do
 {
   - -
 }
 while(condition);
}
while(condition);
```

outer loop

Scanned with CamScanner

iv) while in for loop :-

```
for (exp1; exp2; exp3)
{
  while(condition)
  {
    - -
  }
}
```

Inner loop

outer loop

v) while in nested-for loop :-

```
for (exp1; exp2; exp3)
{
  for (exp1; exp2; exp3)
  {
    while(condition)
    {
    }
    - -
  }
}
```

vi) Nesting of nested for in do-while loop :-

```
do
{
  for (exp1; exp2; exp3)
  {
    - -
  }
  for (exp1; exp2; exp3)
  {
  }
}
while(condition);
```

↳ Using break & continue in nested loops

(i) break :-

```
for (exp1 ; exp2; exp3)
{
  - -
  while (condition)
  {
   - -
   if (condition)
     break;
   - -
  }
  - -
}
```

Inner loop

F T

- In case break is used in inner loop with 2 nested loops, execution of break terminates the execution of inner-loop & control is sent to outer-loop [for]

(ii) continue :-

```
for (exp1; exp2; exp3)
{
  - -
  while (condition)
  {
   if (condition)
    continue;
   - -
  }
  - -
}
```

Inner loop

- In case continue is used in inner loop with 2 nested loops, execution of continue terminates the execution of inner-loop & control is sent to inner-loop [while].

Eg :-

```
int i, j;
for(i=1; i<=3; i++)
{
for(j=1; j<=5; j++)
{
printf("\n i=%d  j=%d", i, j);
}
}
```

O/P :-

i = 1	j = 1	i = 3	j = 1
i = 1	j = 2	i = 3	j = 2
i = 1	j = 3	i = 3	j = 3
i = 1	j = 4	i = 3	j = 4
i = 1	j = 5	i = 3	j = 5
i = 2	j = 1		
i = 2	j = 2		
i = 2	j = 3		
i = 2	j = 4		
i = 2	j = 5		

Eg :- → multiplication table :

```
#include<stdio.h>
void main()
{
 int i, j;
 clrscr();
 ~~printf("~~
 for(i=1; i<=10; i++)
 {
 for(j=1; j<=5; j++)
 {
 printf("\n %d * %d = %d", i, j, i*j);
 }
 }
```

O/P :-

i	j	i*j
1	1	1
1	2	2
1	3	3
1	4	4
1	5	5
1	6	6
1	7	7
1	8	8
1	9	9
1	10	10

— — 10 × 1 = 10

Eg:- i=1, j=1, sum=8;

```
for(i=1; i<=2; i++)
{
 for(j=1; j<=2; j++)
 {
  for(k=1; k<=2; k++)
  {
   printf("\n %d + %d + %d = %d", i, j, k,
                                  (i+j+k));
```

o/p:-

i	j	k	(i+j+k)
1	1	1	3
1	1	2	4
1	2	1	4
1	2	2	5
2	1	1	4
2	2	2	6

Eg:- W.a.p to print pattern :-

```
for(int i=1; i<=5; i++)
  {
   print(" ");
   if(i>=5)
   break;
  }
for(int j=1; j<=5; j++)
 {
  if(j==i)
  continue;
 }
 printf(" * ");
```

```
*
* *
* * *
* * * *
* * * * *
```

Eg:

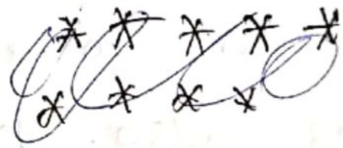
```
int i, n;
printf("enter no of lines :");
scanf("%d", &n);
for (i=5; i>=1; i--)
{
for(j=1; j<=5; j++);
{
printf("*");
}
printf("\n");
}
}
```

F

After print

T

4



o/p:

```
* * * * *
* * * *
* * *
* *
*
```

UNIT-3

Factoring Methods

- It is a no or arithmetic expression or algebraic equation and which on di[vision] gives remainder '0'.
- It is method of multiplication with smaller or equal no to get the number back.
- Factorisation or factoring is used to get small writen as product & on decomposition of a no in to a smaller (or) smaller object.
- It is also known as reverse of product (multiplication) which gives list of factors.

eg:- 12 — 1×12, 2×6, 3×4, 4×2, 6×3, 12×1

eg:- $x^2 - 2x$
$= (x)(x - 2)$
$= x^2 - 2x$

∴ divide by small no. (x)

Problems:-

- 'n' of "a" no is "m"
$n \times n = m$ / $n^2 = m$

$n = 2$, $m = 4$
$2 \times 2 = 4$

methods (ways):
(i) Repeated subtraction
(ii) square root using prime factorization
(iii) square root by long decision
(iv) square root by approximation method (Newton method).

- $\sqrt{6}$

$6 \times 6 = 36$ ✗ → high
$5 \times 5 = 25$ ✗ → high
$4 \times 4 = 16$ ✗ → high
$3 \times 3 = 9$ ✗ → high
$2 \times 2 = 4$ → low
$2.4 \times 2.4 = 5.76$ → low

* Tolerance level :- maximum difference value b/w 'n' & root allowed.

m, L

$$\boxed{root = 0.5 * \left(n_{guess} + (m/n)\right)}$$

used in loop & try to guess n, starting from no 'm' (or) $\frac{m}{2}$ (or) 1. we will update with each iteration with new guessed root

① Find square root : 256 using newton's method :-

Soln :- $m = 256$, $n_{guess} = m$ (or) $\frac{m}{2}$ (or) 1

$n = \frac{m}{2} = \frac{256}{2} = 128$ ('n' guess of m) 65

L = 0.1

$$root = 0.5 * \left(128 + \left(\frac{256}{128}\right)\right)$$

$$0.5 * (128 + 2)$$

$$0.5 * 130 = 65$$

∴ n = 128 & root = 65, the difference (n = root) is greater than tolerance limit 0.1 & hence ~~let~~ us continue with next iteration. [~~new~~ n = new guessed root = 65]

- assign root to n, ∴ n = 65

Iteration-2 :- $root = 0.5 * (n + m/n)$

$$= 0.5 * \left(65 + \left(\frac{256}{65}\right)\right)$$

$$root = 34.46$$

We will check accuracy for each iteration, if it is less or equal to tolerance level 'n' we can come out of the loop & return the value.

Iteration-3 :- $n = 34.46$

$$root = 0.5 * \left(n + \frac{m}{n}\right)$$

$$= 0.5 * \left(34.46 + \left(\frac{256}{34.46}\right)\right)$$

$$= 20.94$$

Iteration-4 :- $n = 20.94$

$$root = 0.5 * \left(20.94 + \frac{256}{20.94}\right)$$

$$root = 16.58$$

Iteration - 5 :- $n = 16.58$

$$root = 0.5 * \left(16.58 + \frac{256}{16.58}\right)$$

$$= 16.01$$

Iteration - 6 :- $n = 16.01$

$$root = 0.5 * \left(16.01 + \frac{256}{16.01}\right)$$

$$\boxed{root = 16} \longrightarrow \begin{array}{r} 16.01 \\ 16.00 \\ \hline 00.01 \end{array}$$

$L = 0.1$

$< = L$ ✓

∴ Since The difference is less than tolerance limit 0.1, ~~which~~ ~~it~~ will return the value of the root i.e, 16.

∴ square root of 256 is 16.

↳ Algorithm to find square root of a no using newton's method:

1) START
2) Declare variables m, n, L, root.
3) Read a no m and to tolerance limit L
4) Initialize root to 0 (root = 0)
5) Set initial guess $n = m/2$
6) Repeat the following steps until absolute difference b/w 'n' & 'root' is less than or equal to tolerance limit L,
   a) $root = 0.5 * n + \left(\frac{m}{n}\right)$
   b) $n = root$
7) Print root
8) End

Scanned with CamScanner

↳ program to find square root of a no using newton's method :

```
# include <stdio.h>
          <stdlib.h>
~~void main()~~
{
double square_root( double m, double L)
{ ~~clrscr();~~
  double root n;
  root 0.0 ;
  n = m/2 ;
  while (1)
  {  root = 0.5 * (n+(m/n)) ;
     if (abs (root - n) <=L)
       break;
     n = root ;
  }
 }
 void main()
{
 double m, L;
 printf("\n enter a number to find a
            square root(m) : ");
 scanf ("%f", &m);
 printf (" enter tolerance limit (L) : ");
 scanf ("%f", &L);
 printf ("square root of %f is : %f",
         m, squareroot (m, L));
}
```

② Find the smallest divisor of an integer?

9 — 1, 3, 9 } divisor [1 or same integer, less than integer]

cross-over

36 = { 1, 2, 3, 4, 6, 9, 12, 18 }

$$\boxed{\text{smaller divisor} \times \text{larger divisor} = n}$$

~~If no. is not even, then use mod~~
~~85 = {1,~~

Steps:-

1) If 'n' is even, 2 is smallest divisor.
2) div = div + 2

- If n mod 2 = 0, then the 2 is smallest divisor, otherwise if n mod 2 ≠ 0 compute $r = \sqrt{n}$.
- Take a variable & initialise div to 3 (div = 3). while div is less than or equal to 'r', do the following step.:
1) If (n mod div is equal to 0) then div is smallest divisor.
2) else div = div + 2

eg:- n = 85

div = 3

85 mode 3 = 1

div = div + 2

div = 3 + 2

div = 5

85 mode 5 = 0

div = small < div 5

n = 73

div = 3

73 mod 3 = 1

div = div + 2
= 3 + 2
= 5

73 mod 5 = 3

div = div + 2
= 5 + 2
= 7

73 mod 7 = 3

div = div + 2
= 7 + 2
= 9

73 mod 9 = 1

div = div + 2
= 9 + 2
= 11

73 mod 11 = 7

div = div + 2
= 11 + 2
= 13

73 mod 13 = 8

div = div + 2
= 13 + 2
= 15

73 mod 15 = 13

div = div + 2
= 15 + 2
= 17

73 mod 17 = 5

div = div + 2
= 17 + 2
= 19

73 mod 19 = 16

This follows until 'O' otherwise it is prime no which does not become 'O'.

Algorithm :-

1) Establish 'n' the integer whose smallest divisor is required [initialize integer n]
2) If 'n' is not odd then return 2 as smallest divisor [
else [r = √n]
   (a) compute 'r' the square root of 'n'.
   (b) Initialize divisor 'd' to 3 [div = 3]
   (c) while not an exact divisor & square root limit not reached do [while (div <= r)]
   (C.1) generate next member in odd sequence 'd'; [div = div + 2]
   • [n mod div == 0, then smallest divisor is div]
   (d) If current odd value 'd' is an exact divisor of 'n' then return it as the exact divisor of 'n'
   else return 1 as the smallest divisor of 'n'
   [If div is exact divisor & div ≠ n then div is smallest divisor of 'n' else 'n' is prime]

(or)

1) check whether n is divisible by 2, if yes, then 2 is the smallest divisor.
2) Iterate, from $i=3$ to squareroot(n) and making a jump of 2.

Eg:- $8 \times 8 = 64$

$2 \times 32 = 64$
$4 \times 16 = 64$
$8 \times 8 = 64$
$16 \times -$
$32 \times -$ } → cross over.

- If any value divides the 'n' then that will be the smallest divisor.
3) If nothing divides, then 'n' is a prime no. & it is the smallest divisor.

~~n = 81~~
~~div = 3~~
~~81 mod 3 = 0~~

$r = \sqrt{71}$
$r = 8$
div <= r
9 <= 8 ✗ [come out of loop]

n = 71 [prime]
div = 3
n mod div
71 mod 3 = 2
div = div + 2
= 3 + 2 = 5

71 mod 5 = 1
div = div + 2
= 5 + 2
= 7
71 mod 7 = 1
div = div + 2
= 7 + 2
= 9

~~71 mod 9 ≠~~
~~div = div + 2~~
~~= 9 + 2~~
~~= 11~~

n = 88 [even]

∴ If n is even, 2 is small divisor of 'n'

[GCD]

3) Find the greatest divisor of an integer :-

gcd (6 , 8) = 2

6 → 1 , 2 , 3 , 6

8 → 1 , 2 , 4 , 8

[gcd (a , b)]

↳ Steps to find GCD of a 2 numbers using Euclid's alg (a, b):

1) If a = 0 , then GCD of a, b = b, because gcd of 0 and b = b, return 'b' as gcd.

eg:- gcd ($\overset{a}{0}$, $\overset{b}{7}$) , then 'b' i.e, 7 is gcd.

otherwise go to step-2.

2) If b = 0 , then GCD of a , b = a , because gcd of 0 and a = a , return 'a' as gcd , otherwise go to step-3.

eg:- gcd ($\overset{a}{7}$, $\overset{b}{0}$) , then 'a' i.e , 7 is gcd.

3) [If a ≠ 0 , b ≠ 0 , then] let 'r' be the remainder of dividing a & b . Assuming a > b . otherwise r = a % b

a) If r = 0 , then gcd (a , b) = b , return value of 'b' as gcd & stop. otherwise (r ≠ 0) → go to step - 4.

eg:- 10 mod 5 = 0 , ($\overset{a}{0}$, $\overset{b}{5}$) → 5 i.e, 'b' is gcd

4) The smaller integer 'b' taken as larger integer a & r is taken as the divisor(b)

eg:- gcd (a, b) r

b = a , r = b ,

a ← b

b ← r

a) assign the value of 'b' to 'a' & 'r' to 'b'.

b) Go to step - 3 to find gcd of (b , r)

c) The process continues until 'r' becomes '0' in step - 3.

Scanned with CamScanner

Eg:- Find GCD of (285, 741):

$a = 285$, $b = 741$.

$gcd(741, 285)$, $a = 741$, $b = 285$

- $r = 741 \% 285 = 171$

$$285\overline{)741}(2 \quad 570 \quad \overline{171}$$

$285 + 285 = 570$

- $a = 285$, $b = 171$
  $r = 285 \% 171 = 114$
- $a = 171$, $b = 114$
  $r = 171 \% 114 = 57$
- $a = 114$, $b = 57$
  $r = 114 \% 57 = 0$ $\therefore$ $gcd(741, 285) = 57$

- ~~$a = 57$, $b = 2$~~
  ~~$r = 57 \% 2 = 1$~~
- ~~$a = 2$, $b = 1$~~
  ~~$r = 2 \% 1 = 0$~~

~~$1\overline{)2}(2 \quad 2 \quad \overline{0}$~~

* Algorithm:-

1) Establish the two positive non-zero integers smaller & larger whose gcd is being sought.
2) Repeatedly:
   a) get the remainder from dividing the larger integer by the smaller integer,
   b) Let the smaller integer assume the role of the larger integer,
   c) Let the remainder assume the role of the divisor,

   until a zero remainder is obtained.
3) Return the gcd of the original pair of integers.

4) Generation of prime numbers :-

- The numbers which has factor as 1 & itself

eg :- $\underline{2 \times 5} = 10$ i.e, 2 & 5 are factors of 20

- Factors are multiplying the nos to get what we want.

- Prime factors refers to the numbers of factors are prime [Factors which is prime no]

20

4 5 } Not prime factors.

2	108
2	54
3	27
3	9
3	3
	1

$2 \times 2 \times 3 \times 3 \times 3 = 108$ } prime factors.

simple way to get prime factors.

30

6 5

$2 \times 3 \times 5 = 30$

* Sieve of Eratosthenes :-

- It is used to list out the numbers sequentially.

n = 36

2	3	~~4~~	5	~~6~~
7	~~8~~	~~9~~	~~10~~	11
~~12~~	13	~~14~~	~~15~~	~~16~~
17	~~18~~	19	~~20~~	~~21~~
~~22~~	23	~~24~~	~~25~~	~~26~~
~~27~~	~~28~~	29	~~30~~	31
~~32~~	33	~~34~~	~~35~~	~~36~~

$\sqrt{36} = 6$ } To stop the process, until square root of 'n'.

2, 3, 5, 7, 11, 13, 17, 19, 23, ~~25~~, 29, 31, 33.

n = 25

2	3	~~4~~	5	~~6~~
7	~~8~~	~~9~~	~~10~~	11
~~12~~	13	~~14~~	~~15~~	~~16~~
17	~~18~~	19	~~20~~	~~21~~
~~22~~	23	~~24~~	~~25~~	

$\boxed{\sqrt{25} = 5}$

2, 3, 5, 7, 11, 13, 17, 19, 23.

Steps - 1 :- $\boxed{n = 18}$

1) Let n = 18,
2) List all the numbers between 2 to 18.

2	3	4	5	6
7	8	9	10	11
12	13	14	15	16
17	18			

3) Circle 1st no : 2 & cross the multiple of that no.

2	3	~~4~~	5	~~6~~
7	~~8~~	9	~~10~~	11
~~12~~	13	~~14~~	15	~~16~~
17	~~18~~			

4) consider / circle the next no which is neither crossed nor circled 3 :

2	3	~~4~~	5	~~6~~
7	~~8~~	~~9~~	~~10~~	11
~~12~~	13	~~14~~	~~15~~	~~16~~
17	~~18~~			

5) consider the next no which is not crossed or circled 5 !

2	3	~~4~~	5	~~6~~
7	~~8~~	~~9~~	~~10~~	11
~~12~~	13	~~14~~	~~15~~	~~16~~
17	~~18~~	19		

6) $\boxed{\sqrt{sqr(n)} = \sqrt{18} = 4}$ is the cross-over (boundary) & no need to proceed further for crossing & circling.

7) List all the nos which are circled & not crossed i.e, remaining nos:

∴ 2, 3, 5, 7, 11, 13, 17.

* Computing prime factors of an integers :-

Integer:

$20 \;—\; 2 \times \underbrace{10}_{2\times 5\,(\text{factors})} \longrightarrow 2 \times 2 \times 5$ (prime factors)

$30 \;—\; 3 \times \underbrace{10}_{2\times 5\,(\text{factors})} \longrightarrow 3 \times 2 \times 5$ (prime factors)

(or)

$$\begin{array}{r|l} 2 & 20 \\ \hline 2 & 10 \\ \hline 5 & 5 \\ \hline & 1 \end{array} \qquad 2 \times 2 \times 5 = 20$$

Algorithm :-

1) START
2) Declare variable n and i :
3) Read number n.
4) while n is divisible by 2, repeat the following steps.
   a) Print '2'
   b) Divide n /2
5) After step - 4, 'n' must be odd i = 3 to the square root of 'n', repeat.
   a) while 'i' divide 'n', print 'i' & divide by 'i'

5) After 'i' fails to divide n, increment by 2 & continue.

6) If 'n' is greater than 2, print n.

7) End.

Eg:- $\boxed{n=36}$

1) Since n = 36 divisible by 2 print 2

$\frac{36}{2} = 18$

- n = 18, print - 2 [18 is divisible by 2]

$\frac{18}{2} = 9$

- n = 9 → at the end.

2) square root = $\sqrt{9} = 3$

i = 3 to 3 → $\boxed{n=3}$, i = 3

(It means that, this step is executed once) (Iterate this step from i = 3 to 3)

- check n is divisible by i:

$\boxed{n=3}$ i = 3

~~print 3~~

- Since 9 % 3 = 0, print 3

$\boxed{n=3}$

- Divide $\frac{9}{3}$, so 'n' becomes 3 (n = 3)
- Increment 'i' by 2: i = 3 + 2 = 5

'i' becomes 5

- loop terminates as 'i' value.

n = 3 at the end of step-1

3) check if 'n' is greater than '2'

since (3 > 2) is true, print the value of n print 3

∴ 2, 2, 3, 3 i.e, 2 × 2 × 3 × 3 = $\boxed{36}$

Eg:- $\boxed{n = 108}$

1) Since $n = 108$ divisible by 2, Print 2

$\frac{108}{2} = 54$

- $n = 54$, Print 2 [54 is divisible by 2]

$\frac{54}{2} = 27$

- $n = 27 \rightarrow$ at the end.

2) ~~Square root = $\sqrt{27} = 5$~~

~~$i = 3$ to $5 \rightarrow \boxed{n = 5}$, $i = 3$~~

If we divide 27 by 2, we get a fractional no. so proceed with next prime factor; $i = 3$

- Check 'n' is divisible by 'i';

$\boxed{n = 27}$ $i = 3$

- Since $27 \div 3 = 9$, Print 3
- Divide again by 3: $9 \div 3 = 3$, Print 3
- Divide again by 3: $3 \div 3 = 1$, Print 3
- ∴ Finally, we receive the number 1 at the end of the division process. We cannot proceed further.
- So, the prime factors of 108 are written as: $2 \times 2 \times 3 \times 3 \times 3$ where 2 & 3 are prime no ~~= 108~~ (false)

5) Generation of Pseudo-random number :-

- To generate pseudo-random no in the field of encryption: Linear congruential generator algorithm (method) is useful.
- The sequence of pseudo-random numbers; x_1, x_2 — — between (0 & $m-1$) can be generated by using linear congruential generator algorithm which uses the following expression:

$$\boxed{x_{n+1} = (a x_n + b) \bmod m \text{ for } n \geq 0}$$

[Express (equation) is used to generate successive pseudo-random number] condition to use the expression :-

1) $m > 0$ & is positive & 'm' is modulus.

2) $2 < a < m$ (where, 'a' is multiplier which is positive but less than modulus 'm')

3) $0 \leq b < m$ (the increment (b) 'b' is positive & less then 'm')

4) $0 \leq x_0 < m$ (the seed (x_0) is positive & less then 'm')

Algorithm :- [to obtain pseudo-random no using Linear congruential Generator]

1) Set parameters values for multiplier a, increment 'b', modulus 'm' & initial seed value 'x_0'.

2) Generate successive member of linear congruential sequence using :

$\boxed{x_{n+1} = (ax_n + b) \bmod m \text{ for } n \geq 0}$, where n's no of pseudo numbers to be generated

3) Repeat step-2 for 'n' numbers.

Steps :-

1) Accept some initial values i.e, 'x_0' which is a seed/key value.

2) Apply the seed value in a mathematical expression to get the result. This result is first random number.

3) Use that resulting random number as the seed of next iteration.

4) Repeat the process to generate pseudo-random numbers upto 'n'.

Problems:

1) Using linear congruential method generate a sequence of pseudorandom numbers where $x_0 = 27$, $a = 17$, $b = 43$, $m = 100$.

$$\boxed{x_{n+1} = (ax_n + b) \bmod m}$$

$x_0 = x_{0+1} = (ax_0 + b) \bmod m$

$x_1 = (17 \times 27 + 43) \bmod 100$

$$\boxed{x_1 = 2}$$

$x_2 = (ax_1 + b) \bmod m$

$= (17 \times 2 + 43) \bmod 100$

$$\boxed{x_2 = 77} \leftarrow 77 \bmod 100$$

$x_3 = (ax_2 + b) \bmod m$

$= (17 \times 77 + 43) \bmod 100$

$= (1309 + 43) \bmod 100$

$= 1352 \bmod 100$

$$\boxed{x_3 = 52}$$

$x_4 = (ax_3 + b) \bmod m$

$= (17 \times 52 + 43) \bmod 100$

$= 927 \bmod 100$

$$\boxed{x_4 = 27}$$

$x_5 = (ax_4 + b) \bmod m$

$= (17 \times 27 + 43) \bmod 100$

$$\boxed{x_5 = 2}$$

$x_6 = (ax_5 + b) \bmod m$

$= (17 \times 2 + 43) \bmod 100$

$$\boxed{x_6 = 77}$$

$\therefore$ If 'n' value is not given, the iteration continues until repeated numbers occurs.

(2) $n = 5$, $a = 109$, $b = 853$, $m = 40960$

$x = 3553$

$$\boxed{x_{n+1} = (a x_n + b) \bmod m}$$

$x_{0+1} = (a x_0 + b) \bmod m$

$x_1 = (109 \times 3553 + 853) \bmod 40960$

$x_1 = (387277 + 853) \bmod 40960$

$x_1 = 388130 \bmod 40960$

$$\boxed{x_1 = 19490}$$

$x_2 = (a x_1 + b) \bmod m$

$x_2 = (109 \times 19490 + 853) \bmod 40960$

$x_2 = (2124410 + 853) \bmod 40960$

$x_2 = 2125263 \bmod 40960$

$$\boxed{x_2 = 36303}$$

$x_3 = (a x_2 + b) \bmod m$

$x_3 = (109 \times 36303 + 853) \bmod 40960$

$x_3 = 3957880 \bmod 40960$

$$\boxed{x_3 = 25720}$$

$x_4 = (a x_3 + b) \bmod m$

$x_4 = (109 \times 25720 + 853) \bmod 40960$

$x_4 = 2804333 \bmod 40960$

$$\boxed{x_4 = 19053}$$

$x_5 = (a x_4 + b) \bmod m$

$x_5 = (109 \times 19053 + 853) \bmod 40960$

$x_5 = 2077630 \bmod 40960$

$$\boxed{x_5 = 29630}$$

6) Raising a number to a larger powers:

$\boxed{x^n}$ $x \rightarrow$ base
$n \rightarrow$ exponent

1) $x^n \longrightarrow 2^6 = 2\times2\times2\times2\times2\times2 = 64$
2) $(x^3 * x^3) = 2^3 * 2^3 = 8*8 = 64$

* Methods to obtain larger Power :-

1) Naive. (Do as it is) (long time)
2) Exponentiation by square.

$\rightarrow$ odd $\begin{cases} x^{11} = x^6 * x^5 \,/\!/\, \boxed{x^{10} * x^1} \,/\, x^7 * x^4 \,/ \\ x^{10} = \boxed{x^5 * x^5} \,/\, x^6 * x^4 \,/\, x^9 * x^1 \,/ \end{cases}$ 1-less

* Algorithm:-

1) START
2) Declare x, n, product = 1.
3) Repeat x & n
4) while $n > 0$
   a) if 'n' is odd, multiply the result by 'x'.
   Product = product * x
   b) Divide 'n' by 2
   c) multiply 'x' by by itself $x = x * x$
5) Print Product.
6) End

eg: $3^9 = x^n$ | $x = 3$, $n = 9$, product = 1

1) $n>0$ $n=9$
   $9>0$ ✓, 9 is odd.
   Product = Product * x
   Product = 1 * 3 = 3
   $n = \frac{n}{2}$ $n = \frac{9}{2} = 4$
   $x = x * x$
   $x = 3 \times 3 = 9$

$n=4$
2) $4>0$ ✓, 4 is not odd
   $n = \frac{n}{2}$ $n = \frac{4}{2} = 2$
   $x = 9 * 9 = 81$

$n=2$
3) $2>0$ ✓, 2 is not odd
   $n = \frac{2}{2} = 1$
   $x = 81 * 81 = 6561$

4) $n = 1$, $1 > 0$ ✓, '1' is odd

$product = product * x$

$= 3 * 6561$

$= 19683$

$n = \frac{n}{2} = \frac{1}{2} = 0$

$x = x * x$

$= 6561 * 6561$

$x = 43046721$

5) $n = 0$, $0 > 0$ ✗,

Print product

$\boxed{3^9 = 19683}$

→ Array techniques :-

- Array is indexed collection of data items which are of same datatype.

eg:- A[10]
↓ array name
↓ size of array.

- A[0]
A[1]
:
A[9]
} Accessing array elements

- Index - Giving numbers
- Array is contiguous (continous)
- Array element is stored in consequent memories.
- Every array has a name.

- malloc, calloc & realloc used in array.

↳ ① Array order reversing :-

1	2	3	4	5
A[0]	A[1]	A[2]	A[3]	A[4]

} A[5] } original array

Array order reversing :

5	4	3	2	1
A[0]	A[1]	A[2]	A[3]	A[4]

} A[5] } order reversed array

Replacement : (i) (j)

A[0] → A[4]
A[1] → A[3]
A[2] → A[2]
A[3] → A[1]
A[4] → A[0]

Interchanging the values not positions.

1st element → last element (~~4~~ 4)
2nd element → 2nd last element (~~4~~ 3)
3rd element → 3rd last element (2)
4th element → 4th last element (1)
5th element → 5th last element (0)
(i) (j)

1st half part of array is swaped with 2nd half part of array.

$$\overset{1}{A[0]} \,/\, \overset{2}{A[1]} \,/\, \overset{3}{A[2]} \,/\, \overset{4}{A[3]} \,/\, \overset{5}{A[4]} \,/\, \overset{6}{A[5]}$$

6 - elements

Algorithm :- ['n' be no of elements stored in a [0 -- n-1] 'i' 'j' for

1) START
2) i, j, n, temp, a[ ]
3) Read 'n' elements of array 'A'.
4) Initialize $\boxed{i=0}$ $\boxed{j=n-1}$
5) while (i < j), do the following change :
   a) Exchange i^{th} element with j^{th} element.
   b) Increment 'i' by 1 and decrement 'j' by 1
6) Print the elements of array 'A'.
7) End.

ODD

* $\boxed{n=5}$, $i=0$, $j=n-1=4$

A[0]	A[1]	A[2]	A[3]	A[4]
14	21	67	32	45

1) (i<j), 0<4 ✓, swap : a[i] with a[j]

swap : a[0] & a[4]

swap : 14 & 45

A[0]	A[1]	A[2]	A[3]	A[4]
45	21	67	32	14

$i = i + 1 = 0 + 1 = 1$
$j = j - 1 = 4 - 1 = 3$

2) (i<j), 1 < 3 ✓, swap : a[1] & a[3]

swap : 21 & 32

32 & 21

A[0]	A[1]	A[2]	A[3]	A[4]
45	32	67	21	14

$i = i+1 = 1+1 = 2$

$j = j-1 = 3-1 = 2$

3) $i < j$, $2 < 2$ ✗ stop

EVEN

* $\boxed{n=6}$, $i=0$, $j = n-1 = 5$

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
14	21	67	32	45	52

1) $i < j$, $0 < 5$ ✓, swap : a[i] with a[j]

swap : a[0] & a[5]

swap : 14 & 52

52	21	67	32	45	14
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

$i = i+1 = 0+1 = 1$

$j = j-1 = 5-1 = 4$

2) $i < j$, $1 < 4$ ✓, swap : a[1] & a[4]

swap : 21 & 45

52	45	67	32	21	14
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

$i = i+1 = 1+1 = 2$

$j = j-1 = 4-1 = 3$

3) $i < j$, $2 < 3$ ✓, swap : a[2] & a[3]

swap : 67 & 32

52	45	32	67	21	14
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

$i = i+1 = 2+1 = 3$

$j = j-1 = 3-1 = 2$

4) $i < j$, $3 < 2$ ✗, stop

$$\left[\; i \leq \frac{n}{2} - 1 \;\right]$$

Eg:-
```
#include <stdio.h>
int main()
{
  int a[ ] = {1, 2, 3, 4, 5};
  int length = sizeof(a);
  Printf("original array : \n");
  for (int i=0; i<j; i++)
  {
    printf(" %d", a[i]);
  }
  Printf("\n");
  printf("Array in reverse order :\n");
  for(int i = j ; i>=0; i--)
  {
    Printf(" %d ", a[i]);
  }
  return 0;
}
```

O/P:- original array : 1 2 3 4 5
Array in reverse order : 5 4 3 2 1

② Finding the maximum number in an (set) given number of array (n-elements):-

max

70	89	-32	12	9	73	12	38	19
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]

max:
1) a[0] , 70 = max
2) a[1] & max

a[1] < max
a[1] > max
a[1] = max

max = updated → a[1]
i=i+1

3) a[2].

Algorithm :- Let a[0 -- n-1] be the array, where n ≥ 1.

1) START
2) Declare a[ ], n, i, max
3) Read n. ~~max~~
4) Set 1st array element as max value & initialize i to 1 :
   max = a[0]
   i = 1
5) While i < n do
   (a) If next array element a[i] > current max then
   (b) i = i + 1
6) Print largest element max.

Tracing :-

1) n = 9, i = 1, max = a[0] = 70
   a[ ] = {70, 89, -32, 12, 9, 73, 12, 38, 19}

2) 1 < 9 ($i < n$) ✓, a[1] > max ⇒ 89 > 70 ✓
   update max = a[1] = 89.
   max = 89, i = i + 1 = 1 + 1 = 2

3) 2 < 9 ✓, a[2] > max ⇒ -32 > 89 ✗
   (Only 'i' is incremented), i = i + 1 = 2 + 1 = 3

4) 3 < 9 ✓, a[3] > max ⇒ 12 > 89 ✗
   i = i + 1 = 3 + 1 = 4

5) 4 < 9 ✓, a[4] > max ⇒ 9 > 89 ✗
   i = i + 1 = 4 + 1 = 5

6) 5 < 9 ✓, a[5] > max ⇒ 73 > 89 ✗
   i = i + 1 = 5 + 1 = 6

7) 6 < 9 ✓, a[6] > max ⇒ 12 > 89 ✗
   i = i + 1 = 6 + 1 = 7

8) 7 < 9 ✓, a[7] > max ⇒ 38 > 89 ✗
   i = i + 1 = 7 + 1 = 8

9) (8 < 9) ✓, a[8] > max ⇒ 19 > 89 (X)

i = i + 1 = 8 + 1 = 9

10) (9 < 9) (X)

Print : max = 89

Eg: # include<stdio.h>

void main()

{

int a[]; ~~= {70, 89, 32, 12, 9, 73, 12, 38, 19}~~

int length = sizeof(a);

int n, i, max = a[0];

printf("enter the no of elements of an array");

scanf("%d", &n);

printf("enter the elements of an array");

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

scanf("%d", &a[i]);

max = a[0];

while (i<n) do

{

a[i] > max; max = a[0];

~~if (max = a[i])~~ i = i+1;

~~return max;~~

else

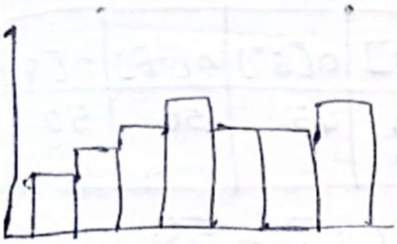
~~i = i+1~~

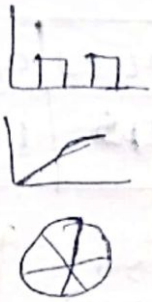
}

printf("largest element is:", max);

}

(3) Array counting or histograming :-





- Histogram similar to Bar graph but continuous count
- Set of nos :
  ↳ Frequency count continuous.

eg:- {2, 3, 2, 5, 3, 4, 7}

2 → 2
3 → 2
5 → 1
4 → 1
7 → 1

[How many times, a no is repeted (occuring) → Frequency]

- Given a set of nos (or) array of nos we need to findout the frequency count of each no.

Algorithm:- Let a[0 -- n-1] be a array where $n \geq 1$

1) START
2) Input 'n' (no of integers / numbers)
3) Initialize counting ~~array~~ array elements : (a[0, -- 100]) to zero.
4) While n>0 do
   (a) Read next (number) mark m.
   (b) Increase the count by one in ~~the~~ location ~~m~~ 'm' in counting memory.
   i.e, $a[m] = a[m] + 1$
5) output frequency count distribution or histogram for numbers.
6) Exit.

Eg:- n=10

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
37	45	50	45	100	0	45	50	50	100

a[i] = 0

1) a[1] = 37
a[m] = a[m] + 1
a[37] = 0 + 1
= 1

2) a[2] = 45
a[m] = a[m] + 1
a[45] = 0 + 1
= 1

3) a[3] = 50
a[m] = a[m] + 1
a[50] = 0 + 1
= 1

4) a[4] = 45
a[m] = a[m] + 1
a[45] = 1 + 1
= 2

5) a[5] = 100
a[m] = a[m] + 1
a[100] = 0 + 1
= 1

6) a[6] = 0
a[m] = a[m] + 1
a[0] = 0 + 1
= 1

7) a[7] = 45
a[m] = a[m] + 1
a[45] = 2 + 1
= 3

8) a[8] = 50
a[m] = a[m] + 1
a[50] = 1 + 1
= 2

9) a[9] = 50
a[m] = a[m] + 1
a[50] = 2 + 1
= 3

10) a[10] = 100
a[m] = a[m] + 1
a[100] = 1 + 1
= 2

(numbers) → (frequency)
37 → 1
45 → 3
50 → 3
100 → 2
0 → 1

* program:-

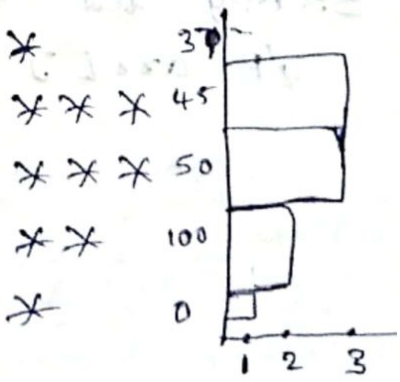
```
# include<stdio.h>
int main()
{
  int m, n, i, a[100];
  for (i=0; i<=100; i++)
   a[i]=0;
  printf("enter no of integers");
  scanf("%d", &n);
  printf("enter the number");
  for (i=1; i<=n; i++)
  {
    scanf("%d", &m);
    a[m] = a[m]+1;
  }
  printf("Marks (numbers) \t frequency \n");
  for (i=0; i<=100; i++)
  if (a[i]!=0)
  printf("\n3d \t %3d", i, a[i]);
}
```

O/P:- enter no of integers: 10

enter the number:

37 45 50 45 100 0 45 50 50 100

numbers	Frequency.
37	1
45	3
50	3
100	2
0	1



(4) To remove duplicate elements or numbers from an ordered array!

- Ordered array → Sorted array (Ascending / descending)

- A = {1, 1, 2, 2, 3, 4, 4, 5, 6, 6}

• duplicates elements : 1, 2, 3, 4, 6

• After removing duplicates elements :

A = {1, 2, 3, 4, 5, 6}

- To remove duplicates elements there are

(i) Using temporary array.

(ii) Without using temporary array.

(i) Using temporary array, we can remove duplicate elements of an ordered array by following steps!

Step-1:- Create a temporary array temp[ ] to store unique elements.

Step-2:- Traversing i/p array arr[ ], ~~an~~ i.e, reading 1 by 1 elements of array, we need to copy unique elements of arr[ ] to temp[ ] & we also need to keep track of count of unique elements. For this we use variable 'j' (to keep count of unique elements).

3:- Copy the ~~ten~~ j-element of temp[ ] to i/p arr[ ] & print the same.

(extra)
* Algorithm :- [using temporary array]

1) START
2) Declare arr[ ], temp[ ], i, j, n;
3) Read 'n' elements of array.
4) Initialize i = 0 & j = 0;
5) for i = 0 to n-2 do
   (a) If i^{th} element is not equal to $i+1^{th}$ element of arr[ ], then temp[j] = arr[i].
   (b) j = j + 1
6) Store last element of i/P array, arr[ ] to temp[ ].
   temp[j] = arr[n-1]
   j = j + 1
7) copy elements from temp[ ] to arr[ ].
   for 0 to j - 1 do
   arr[i] = temp[i]
8) Print final array arr[ ] from i = 0 to j = j - 1.
9) End.

- Check whether ith element of arr[ ] is equal to the $(i+1)^{th}$ element.
- If arr[i] & arr[i+1] are same, then we will increment the value of 'i' by 1.
- If arr[i] & arr[i+1] are not same i.e, unique then we will store the ith element in temp[ ] the we will increment 'i' by 1 & 'j' by 1.
- for loop will run till 2^{nd} last element of arr[ ] ∵ there is no $(i+1)^{th}$ element for comparision.
- After looping, copy the 'i' elements of temp[ ] to arr[ ] & print the same.

(ii) Algorithm :- Without using temporary array :

Let arr[0 -- n-1] where array is sorted array, where $n \geq 1$.

1) START
2) Declare arr[ ], i, j, n.
3) Read 'n' no of array.
4) Initialise i = 0 & j = 0
5) for i = 0 to i = n - 2 do
   (a) If ith element is not equal to i+1th element of arr[ ] then.
   arr[j] = arr[i].
   (b) j = j + 1.
6) store last element of arr[ ] to arr[j]
   arr[j] = arr[n - 1]
   j = j + 1.
7) Print the final array arr[ ] from i = 0 to j = j - 1.
8) STOP.

Tracing :-

(i) Let arr[ ] =

0	1	2	3	4	5
1	3	5	5	7	9

i = 0, j = 0

temp[j] = arr[i]

i → will point to index of each element of arr[ ]

j → will point to index of elements of new array temp[ ] → non-unique elements will be stored.

initialize j = 0 ∵ temp in initially empty.

Scanned with CamScanner

(•) arr

1	3	5	5	~~X~~	7	9

i

$1 \neq 3$

temp

1					

j = 1, i = i + 1 = 3

(•) compare next 3 & 5 are not same, so 3 will be stored in temp & 'i' will incremen-ted by 1.

arr (i+1)

1	3	5	5	7	9

i

$3 \neq 5$

temp

1	3				

j

j = 2, i = i + 1 = 5

(•) i = 5, i+1 = 5

5 = 5, so i th element will be stored & 'j' is not incremented

i = i+1 (if both elements are same)

(•) arr (i+1)

1	3	5	5	7	9

i

5 = 5

temp

1	3				

j

j = 2

(•) $5 \neq 7$, so 5 will be stored in temp & 'i' = increment & j = increment.

arr (i+1)

1	3	5	5	7	9

i

$5 \neq 7$

temp

1	3	5			

j

j = 3

(•) i = 7, & 9, $7 \neq 9$, so 7 will be stored in temp & i = increment

arr (i+1)

1	3	5	5	7	9

i

$7 \neq 9$

temp

1	3	5	7		

j

j = 4

(•) At last we will add 9 to temp:

temp

1	3	5	7	9	

j

j = 5

- So, the array temp contains all non-duplicate elements of arr. Copy the j elements of temp to arr & print them.

temp | 1 | 3 | 5 | 7 | 9 | | $j = 5$

copy 'j' elements of temp to arr

arr | 1 | 3 | 5 | 7 | 9 | 9 |

Print 'j' elements

Eg:-
```
#include <stdio.h>
void removeDuplicates(int arr[], int n)
{
    int i, j=0;
    int temp[n];
    for(i=0; i<n-1; i++)
    {
      if(arr[i] != arr[i+1])
      {
        temp[j] = arr[i];
        j++;
      }
    }
    temp[j++] = arr[n-1];
    for(i=0; i<j; i++)
      arr[i] = temp[i];
    printf("Array elements after removal of
                duplicates :");
    for(i=0; i<j; i++)
    printf("%d", arr[i]);
}
void main()
{
    int n, i;
    printf("enter no of elements:");
    scanf("%d", &n);
    int arr[n];
    printf("\n enter %d elements in sorted order");
    for(i=0; i<n; i++)
    scanf("%d", &arr[i]);
    removeDuplicates(arr, n);
}
```

O/P: enter no of elements : 10
enter 10 elements in sorted order : 1 2 2 3 4 5
5 5 8 8

Array elements after removal of duplicates :
1 2 3 4 5 8

(ii) Tracing :-

Let us consider the sorted array arr[] of 6 elements. n = 6
initially, i=0 & j=0

0	1	2	3	4	5
11	21	21	34	34	50
i, j					

- In 1st iteration of the for loop, 'i' will point to the index of 1st element (1). then we will check if the ith element is equal to (i+1)th element.
- If ith element is not equal to (i+1)th element of arr[], then store ith value in arr[j].

Step-1 :- The element at 1st position (a[i] = 11) is compared with element at 2nd position (a[i+1] = 21).

- Elements compared are different (11, 21), 11 is unique element placed at 1st position,

arr[j] = arr[i]
arr[j] = arr[0]
arr[j] = 11
arr[0] = 11
increment 'i' by 1 & 'j' by 1
so, i=1 & j=1

0	1	2	3	4	5
11 ✓	21	21	34	34	50
	i, j				

Step-2 :- The element at 2nd position (a[i] = 21) is compared with element at 3rd position (a[i+1] = 21). Elements compared are same (21, 21). If it is same, then just increment 'i' by 1.

(i=2), No change in j-value so, (j=1)

0	1	2	3	4	5
11	21	21	34	34	50
✓	j	i			

Step-3 :- The element at 3rd position (a[i]=21) is compared with element at 4th position (a[i+1]=34).

- Elements compared are different (21,34)
- 21 is unique element placed at 2nd position.

arr[j] = arr[i]
arr[j] = arr[2]
arr[j] = 21
arr[1] = 21

Increment i by 1 & j by 1
so, (i=3) & (j=2)

0	1	2	3	4	5
11	21	21	34	34	50
✓	✓	j	i		

Step-4 :- The element at 4th position (a[i]=34) is compared with element at 5th position (a[i+1]=34).

- Elements compared are same (34,34).
- If it is same, then just increment 'i' by 1.

(i=4), No change in 'j' value, so (j=2)

0	1	2	3	4	5
11	21	21	34	34	50
✓	✓	j		i	

Step-5 :- The element at 5th position (a[i]=34) is compared with element at 6th position (a[i+1]=50).

- Elements compared are different (34,50).
- 34 is unique element placed at 3rd position.

arr[j] = arr[i]
arr[j] = arr[4]
arr[j] = 34
arr[2] = 34

increment 'i' by 1 & 'j' by 1

so, i = 5 & j = 3

0	1	2	3	4	5
11	21	34	34	34	50
✓	✓	✓	j		i

- Since 'i' value is n-1, we can't compare with a[i+1]. so, we stop iterating the loop.
- Finally store the last element of an array in arr[j].

  arr[j] = arr[n-1] (or) arr[i]

  arr[3] = arr[5]

  arr[3] = 50

  increment j by 1, so j = 4

0	1	2	3	4	5
11	21	34	50	34	50
✓	✓	✓	✓	j	

- Now, Print 1st 'j' element of array arr[ ].

  arr[0] = 11, arr[1] = 21, arr[2] = 34 & arr[3] = 50,

- Note that, length of an array should be considered as 'j'. If we iterate the elements from 0 to j-1, we get the unique elements.

eg:-
```
#include<stdio.h>
void removeDuplicate(int arr[], int n)
{
    int i, j=0;
    for(i=0; i<n-1; i++)
    {
        if(arr[i]!=arr[i+1])
        {
            arr[j]=arr[i]};
            j++;
        }
    }
    arr[j++]=arr[n-1];
    printf("\n Array elements after removal
            of duplicates: ");
```

```
for(i=0; i<j; i++)
 printf("%d ", arr[i]);
}
void main()
{
 int n,i;
 printf("enter no of elements:");
 scanf("%d", &n);
 int arr[n];
 printf("\n enter %d elements in sorted
        order:", n);
 for(i=0; i<n; i++)
 scanf("%d", &arr[i]);
 removeDuplicates(arr, n);
}
```

Scanned with CamScanner