MODULE-I

Algorithm

Algorithm has come to refer to a method that can be used by a computer for the solution of a problem. This is what makes algorithm different from words such as process, technique or method.

An algorithm is a finite set of instructions that is followed, accomplishes a particular task. All algorithms must satisfy the following criteria.

* Input : One or more quantities are externally supplied.
* Output : Atleast one quantity is produced.
* Finiteness : If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite no: of steps.
* Definiteness : Each instruction is clear & unambiguous.
* Effectiveness : Every instruction must be very basic so that it can be carried out, in principle by a person.

Properties of a Good Algorithm

⇒ Should be written in simple English
⇒ Each step of an algorithm is unique & should be self explanatory
⇒ An algorithm must have atleast one input
⇒ Should provide the correct solution
⇒ An algorithm has finite number of steps
⇒ Should have an end point
⇒ Every statement should be definitive.

Algorithm Efficiency

In computer science, algorithmic efficiency are the properties of an algorithm which relate to the amount of computational resources used by the algorithm. An algorithm must be analysed by determine its resource usage.

for maximum efficiency, we wish to minimize resource usage. The various resources (eg: time, space) cannot be compared directly.

* **Time Complexity:** amount of computer time it needs to run the completion of an algorithm/program. The actual running time depends on many factors.
  - → The speed of the computer
  - → The compiler, compiler options
  - → The quantity of data - Search a long list or short

The time T(P) taken by a program P, is the sum of compile time and the run time. Here compile time doesnot depend on the instance characteristics. We concern with the run time of a program. This run time is denoted by t_p.

eg: If we knew the characteristics of the compiler to be used, we could proceed to determine the no: of additions, subtractions, multiplications, divisions etc.

We could obtain an expression for t_p of the form:

$$t_p = C_a \text{ Add} + C_s \text{ Sub} + C_m \text{ mul} + C_d \text{ division} + \cdots$$

* **Space Complexity:** a measure of the amount of memory needed for an algorithm to execute.

Space needed by an algorithm is equal to the sum of the following two components.

a) **A fixed part:** that is a space required to store certain

data and variables (ie; simple variables and constants, program size etc), that are not dependent of the size of the problem.

b) A variable part: is a space required by variables, whose size is totally dependent on the size of the problem.

Space complexity $S(P)$ of any algorithm P is $S(P) = A + S_p(I)$ where A is treated as the fixed part and $S(I)$ is treated as the variable part of the algorithm. which depends on instance characteristic I.

eg:
```
Step 1 : START
Step 2 : R = P + Q + 10
Step 3 : Print the result in R
Step 4 : STOP
```

Here, we have 3 variables P, Q, R and one constant. Hence space complexity $S(P) = 1+3$. Now space is dependent on data types of given constant types and variables and it will be multiplied accordingly.

Asymptotic Notations

Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.

Asymptotic notations are languages that allow us to analyse an algorithms running time by identifying its behaviour as the input size of the algorithm. This is also known as an algorithms growth rate.

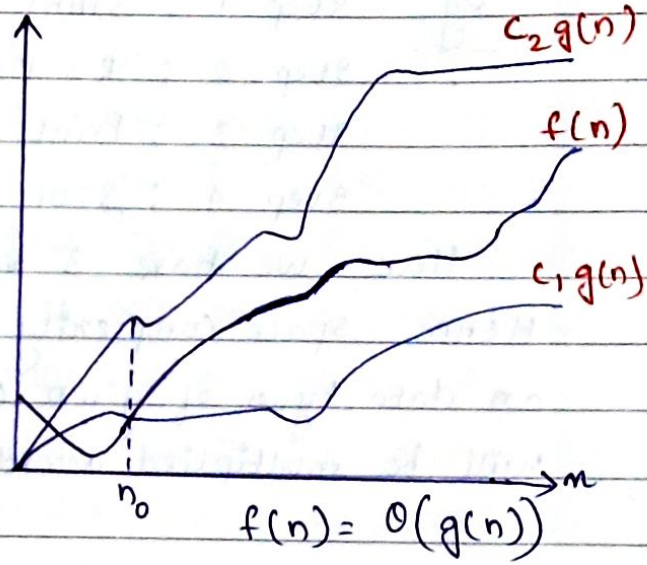
The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

1) **Θ Notation** - The 'Θ' Notation bounds a functions from above and below, so it defines exact asymptotic behaviour. A simple way to get theta (Θ) notation of an expression is to drop low order terms & ignore leading constants. for a given function $g(n)$, we denote $\Theta(g(n))$ is following set of functions.

$$\Theta(g(n)) = \{ f(n) : \text{there exists positive constants } c_1, c_2 \text{ \& } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \; \forall \; n \ge n_0 \}$$

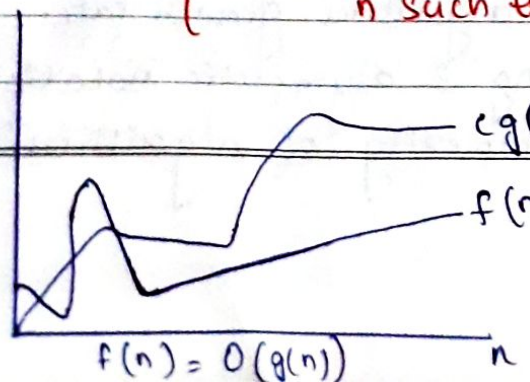
- $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$ since 'Θ notation' is stronger than 'O notation'.

$$\Theta(g(n)) \subseteq O(g(n))$$



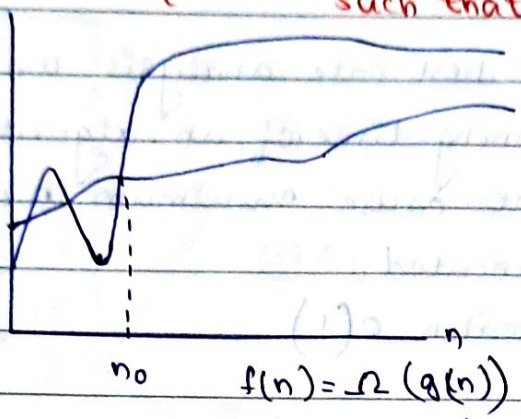
2) **Big O Notation** (O-Notation) - The Big O Notation defines an upper bound of an algorithm, it bounds a function only from above. for a given function $g(n)$, we denote by $O(g(n))$ the set of functions.

$$O(g(n)) = \{ f(n) : \text{there exist positive constants 'c' and 'n' such that } 0 \le f(n) \le g(n) \; \forall \; n \ge n_0 \}$$



3) **Ω - Notation** : Just as Big O notation, provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound. Ω Notation can be useful when we have lower bound on time complexity of an algorithm.

$\Omega(g(n)) = \{f(n)$: there exist positive constants 'c' & 'n_0' such that $0 \leq g(n) \leq f(n)$ $\forall n \geq n_0$



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

In each part, the value of 'n_0' shown is the minimum possible value; any greater value would also work.

ANALYSIS OF ALGORITHMS

We have 3 cases to analyse an algorithm.

1. Worst case analysis
2. Average case analysis
3. Best case analysis.

* **Worst case analysis** - In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed.

eg: Linear search - $O(n)$

* Average case analysis :- In average case analysis, we take all possible input and calculate completing time for all the inputs. Sum of all the calculated values and divide the sum by total number of inputs.

eg- Linear search : O(n)

* Best case analysis :- In best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that cause minimum number of operations to be executed.

eg : Linear search O(1)

Recursive Binary Search Algorithm

Step 1 : START

Step 2 : find the midpoint of the array. The midpoint divides the array into 2 smaller arrays.

Step 3 : The lower half of the array consisting of elements 0 to midpoint - 1 and the upper half of the array consisting of elements midpoint to size-1

Step 4 : Compare key to arr [midpoint] by calling the user function.

Step 5 : If the key is match, return arr[midpoint]. otherwise -

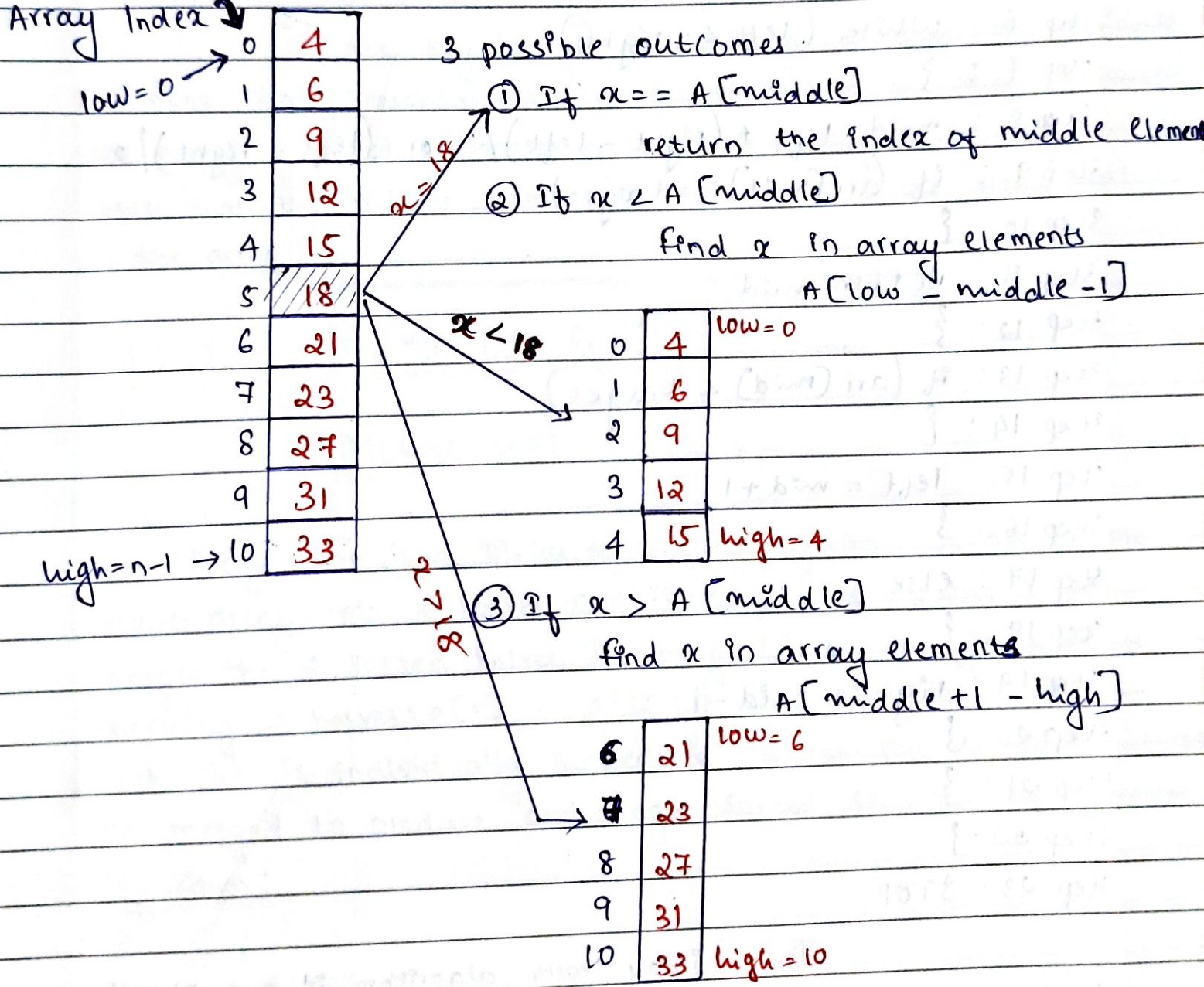
Step 6 : If the array consists of only one element return NULL (indicating that there is no match: otherwise -

Step 7 : If the key is less than the value extracted from arr[midpoint] search the lowerhalf of the array by recursively calling search. otherwise.

Step 8 : Search the upper half of the array by recursively calling search

Step 9 : STOP

Example : Find x in array elements A[low -- high]



Non-Recursive Binary Search Algorithm

Step 1 : START
step 2 : Algorithm binary search (int [] arr, int target)
Step 3 : {
Step 4 : left = 0, right = arr. length - 1
Step 5 : res = -1
Step 6 : while (left <= right)
Step 7 : {
Step 8 : mid = left + (right - left)/2 or (left + right)/2
Step 9 : if (arr[mid] == target)
Step 10 : {
Step 11 : Retuen mid
step 12 : }
Step 13 : if (arr[mid] < target)
Step 14 : {
Step 15 : left = mid + 1
Step 16 : }
Step 17 : else
Step 18 : {
Step 19 : right = mid - 1
Step 20 : }
Step 21 : }
Step 22 : }
Step 23 : STOP

The binary search algorithm is one of the fundamental computer science algorithms & used to search an element in a sorted input set.

In a binary search algorithm, you first

find the middle element of the array & compare that with the number you are searching. If it's equal then you return true or index of that number and your binary search is complete but if it doesn't match then you divide the array in two-part based upon whether the middle element is greater than or less than the key value, which you are searching.

If the key (target value) is greater than the middle element than search values in the second half of the array because the array is sorted in increasing order. Similarly, if the key is lower than the middle element it means it's in the first part of the array.