KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

DATA STRUCTURES –ITT 201 (S3 IT)

MODULE -1

	Module 1: Introduction to data structures	9hrs
1.1	**Data Structures-Introduction and Overview**: Definitions, Concept of data structure, classifications of data structure- ADT and CDT- Linear and nonlinear.	1
1.2	**Arrays:** definition, Representation of Single/Two dimensional arrays, Applications of array – searching –Sorting - Sparse Matrix- conversion of sparse matrix into 3 tuple form.	2
1.3	**Algorithm/Program Development:** Analysis of algorithms. Space Complexity, Time Complexity - Best case, worst case, average case. **Searching** : linear and binary search – Complexity Analysis (Detailed analysis is not required)	2
1.4	**Sorting**: classifications- Internal sorting – External sorting , N^2 Sorting : Selection, bubble and insertion- Complexity analysis (Detailed analysis is not required)	2
1.5	N $\log_n$ Sorting : Quick Sort and Merge Sort (Recursive Algorithms)- Complexity Analysis (Detailed analysis is not required)	2

Data Structures – Introduction & Overview

Definition – Data Structure

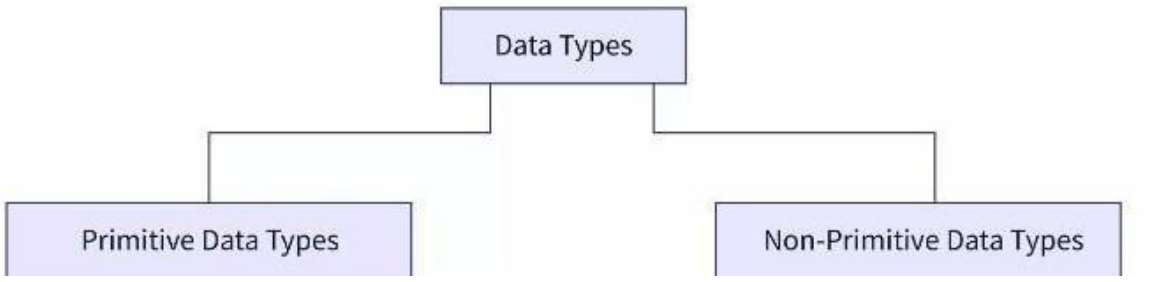
A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways. Most importantly, data structures frame the organization of information so that machines and humans can better understand it.

Data structures are the building blocks of any program. Data Structure is a way to store and organize data so that it can be used efficiently. Data structures are the main part of many algorithms as they enable the programmers to handle the data in an efficient way. The data structure is not any programming language like C, C++, Java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.

Types of Data Structures

There are two types of data structures:

- Primitive Data Structures
- Non-Primitive Data Structures



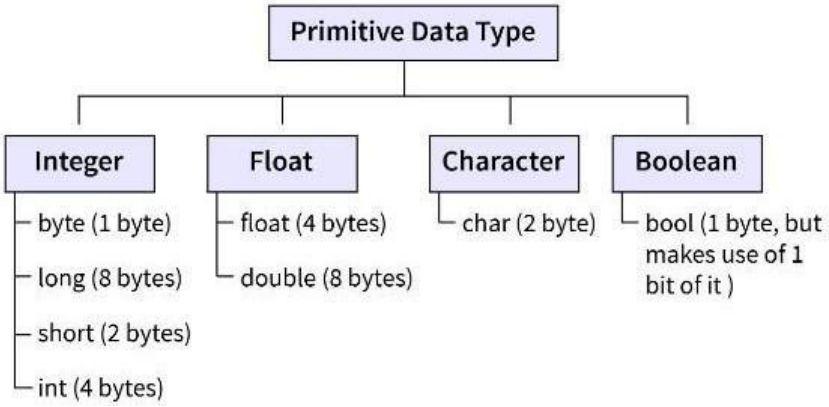
Primitive Data Structure

These are the fundamental data types which are directly supported by the machine & allows storing the values of only one data type.

The different types of primitive data types are :

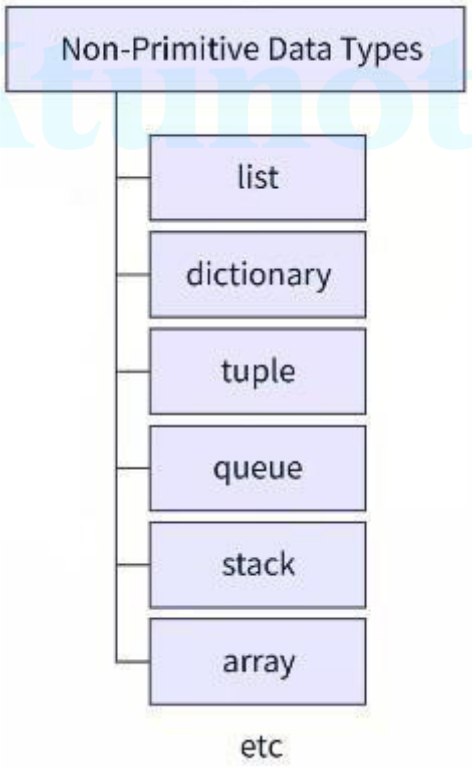
- Integer
- Float

- Double
- Character
- Boolean



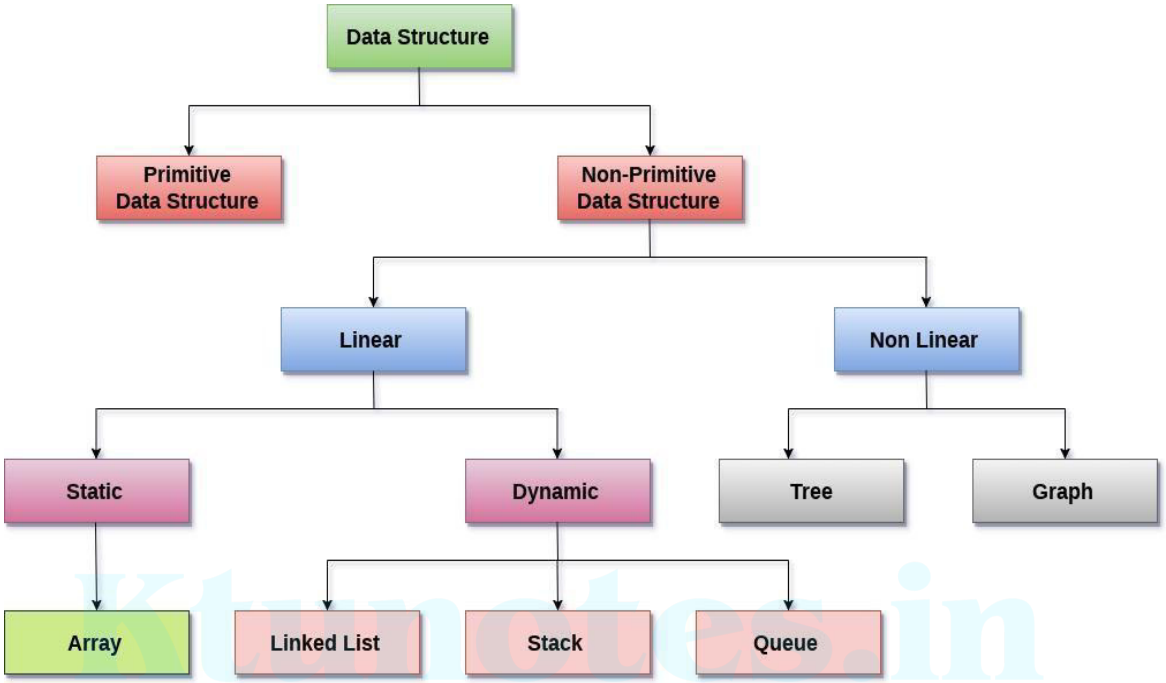
Non-Primitive Data Structure

Non-Primitive data structures are considered as the user-defined structure that allows storing values of different data types within one entity.



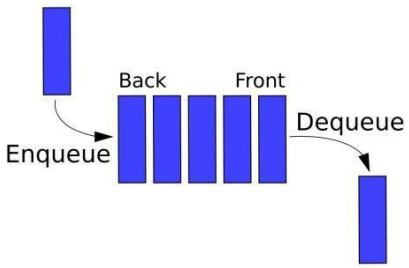
Classification of data structure

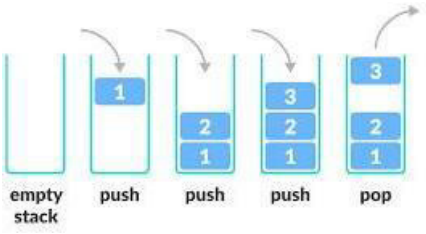
By using data structure, one can organize and process a very large amount of data in a relatively short period. Let's look at different data structures that are used in different situations.



Abstract data type (ADT)

- a mathematical model of a data structure
- specifies the type of data stored, the operations supported on them, and the types of parameters of the operations
- specifies what each operation does, but not how it does it can be implemented using one of many data structures
- examples : Stack, Queue





Concrete data type (CDT)

- It is a specialized solution-oriented data type that represents

a well-defined single solution domain concept.

- A concrete data type is a data type whose representation is known And relied upon by the programmers who use the datatype.
- Boolean,Integer,Floating Point,User defined Structures
- Arrays,linked lists,trees,graphs

Primitive Data Structure

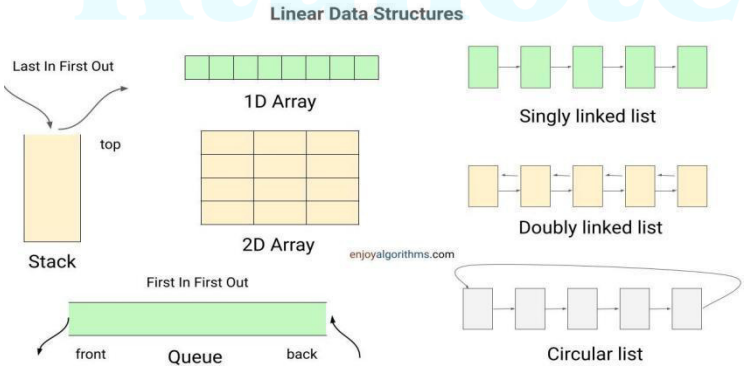
A primitive data structure can store the value of only one data type. For example, a char data structure (a primitive data structure) can store only characters.
Eg: integer, character, boolean, float, double, long, etc

Non primitive data structure

Unlike the primitive data structure, a non-primitive data structure can store the value of more than one data type.this can be classified into two categories : Linear data structure and Non linear data structure

Linear data structure

Data elements in a linear data structure are linked to one another in a sequential arrangement, with each element linked to the elements in front of and behind it.
Eg: array,linked list,stack,and queue



- Array

An array is a linear collection of values stored at contiguous memory locations. Array stores homogeneous values(similar data type values).

- Linked list

A linked list is a sequential collection of data elements connected via links. The data element of a linked list is known as a node which contains two parts namely- the data part and the pointer. The data part contains the actual data and the pointer part contains a pointer that points to the next element of the linked list.

- Stack

Stack in a linear data structure that stores data sequentially based on the Last In First Out (LIFO) manner. So, the data which is inserted first will be removed last. Since the last element inserted is served and removed first, the queue data structure is also known as the Last Come First Served data structure.

- Queue

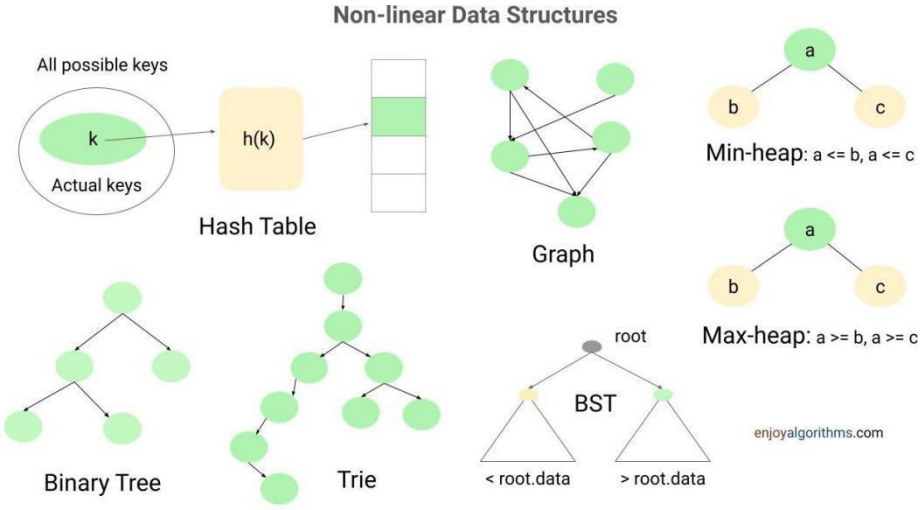
Queue in a linear data structure that stores data sequentially based on the First In First

Out(FIFO) manner. So, the data which is inserted first will be removed from the queue first.

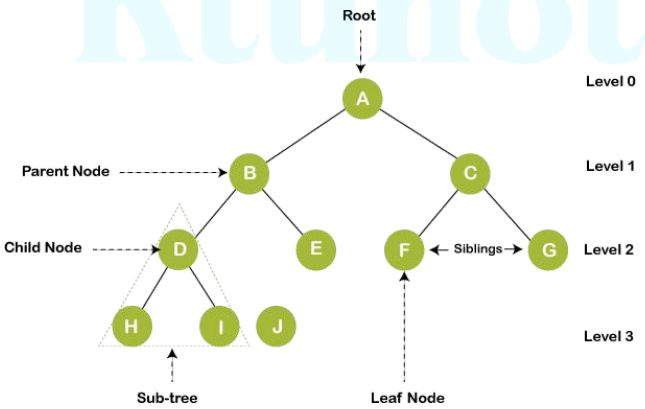
Since the first element inserted is served and removed first, the queue data structure is also known as the First Come First Served data structure

Non linear data structure

Here ,elements are not arranged sequentially.each data elements can be linked to more than one data element.eg:tree,graph,set etc



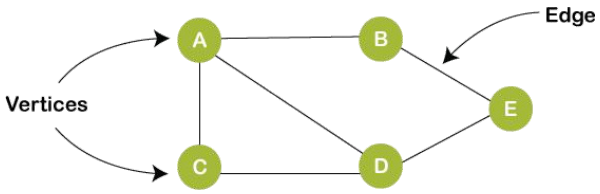
- Tree



The tree is a non-linear data structure that comprises various nodes. The nodes in the tree data structure are arranged in hierarchical order.The tree itself is a very broad data structure and is divided into various categories like Binary tree, Binary search tree, AVL trees, Heap, max Heap, min-heap, etc.

- graph

A graph is a non-linear data structure that has a finite number of vertices and edges, and these edges are used to connect the vertices. The vertices are used to store the data elements, while the edges represent the relationship between the vertices



Previous year questions

1.What is an Abstract Data Type? What is its use?
2.Explain the classifications of data Structures.?
3.Differentiate between ADT and CDT.

Array

>Definition

•An array is a consecutive set of memory locations. An array is a set of pairs, ie; index & values.

• For each index which is defined, there is a value associated with the index. In mathematical terms we call this a correspondence or a mapping.

• Array can be defined as

Structure - ARRAY(value,index)
Declare - CREATE()array
RETRIEVE(array,index)-->value
STORE(array, index, value)--> array

For all A is an array, i, j are index and x is value.

>Representation of Arrays

One Dimensional Array

•One dimensional array can be represented as followsA[lower bound : Upper bound]
Eg: A[3:4]

•The address A[i] can be calculated by
Base address +(i -Lowerbound)=α+(i-L)
Here; base address is the starting address.

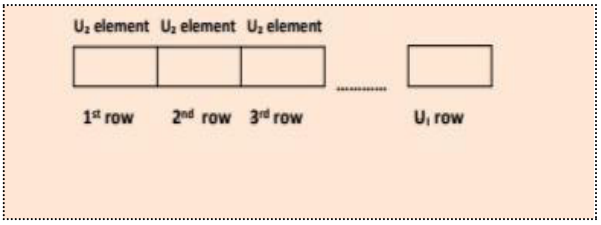
•Total number of elements can be calculated by
Upper bound – Lower bound+1=U-L+1

Two Dimensional Array

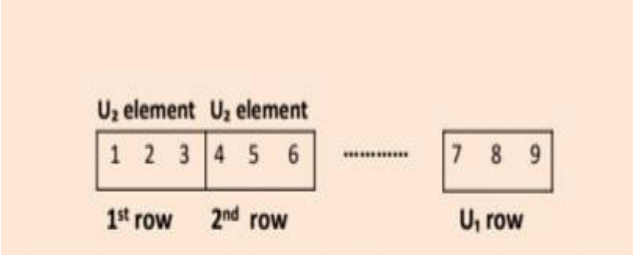
•It can be represented as
A[L_1…….U_1 , L_2…….U_2]
Row = U_1-L_1+1
Column = U_2-L_2+1
• So, Total number of elements =Row*Column
The two common ways to represent multidimensional arrays are
1. Row major order
2. Column major order

Row Major

As its name implies, row major order stores multidimensional arrays by rows
A[L_1……..U_1 , L_2…….U_2]
Where
U_1 = row representation
U_2 = column representation



Eg:1 2 3
4 5 6. A [3,3] in C, A[3][3]
7 8 9

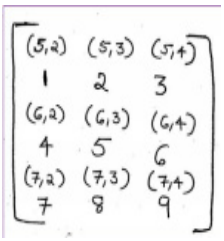


we can find the address of A[i, j] using row major,

Base address + $(i-L_1)U_2 + (j-L_2)$

Where, U_2 represents column

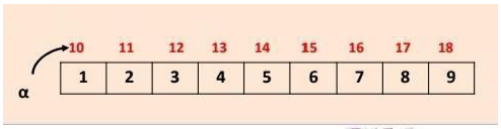
Eg: A[5:7 , 2:4] find the address of A[6,3] , base address α=10.



L_1 U_1 L_2 U_2.

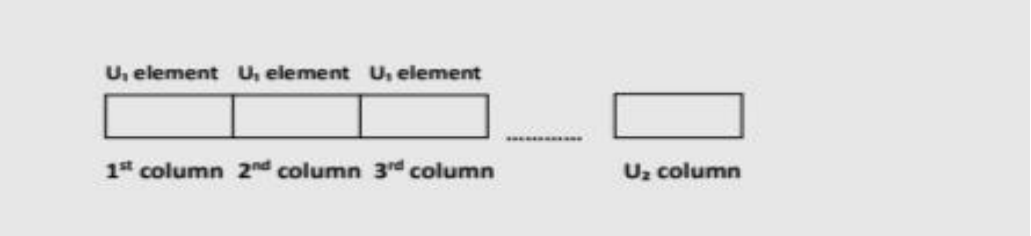
A [5 : 7 , 2 : 4]
m (row) = $U_1-L_1+1 = 3$
n (column) = $U_2-L_2+1 = 3$
A(6,3) =Base address + $(i-L_1)$ column + $(j-L_2)$
= 10+(6-5)3 + (3-2)

=10+4=14

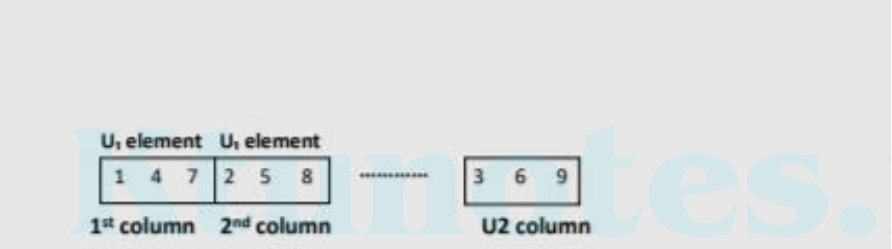


Column Major

As its name implies, Column major order stores multidimensional arrays by columns



Eg:1 2 3
4 5 6
7 8 9



we can find the address of A[i, j] using row major

Base address + $(i-L_1) + (j-L_2)m$

Where, m represents row

Eg: A[6:9 , 3:6] find the address of A[8,5] , base address α=10
.L_1 U_1 L_2 U_2
A [6 : 9 , 3 : 6]
m (row) = U_1-L_1+1 = 9-6+1 = 4
n (column) = U_2-L_2+1 = 6-3+1= 4
A(8,5)
= 10+(8-6) + (5-3)x 4
= 10+2+8
=20

APPLICATIONS OF ARRAY

● Arrays are used to Store List of values

One dimensional arrays are used to store a list of values of the same datatype. In other words, one dimensional arrays are used to store a row of values. In a one dimensional array, data is stored in linear form.

● Arrays are used to Perform Matrix Operations

We use two dimensional arrays to create a matrix. We can perform various operations on matrices using two dimensional arrays.

● Arrays are used to implement Search Algorithms

We use one dimensional arrays to implement search algorithms like

1. Linear search
2. Binary search

● Arrays are used to implement Sorting Algorithms

We use one dimensional arrays to implement sorting algorithms like

1. Selection sort
2. Bubble sort
3. Insertion sort
4. Quick sort
5. Merge sort

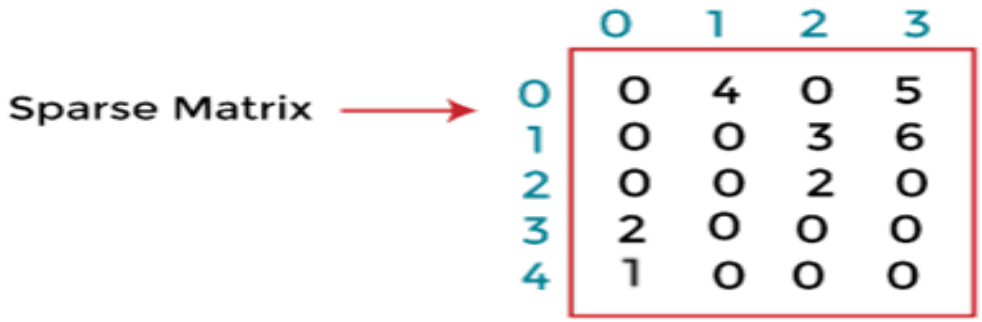
● Arrays are used to implement Data Structures

Arrays to implement data structures such as stacks ,queues etc..

● Arrays are also used to implement CPU Scheduling Algorithms

SPARSE MATRIX

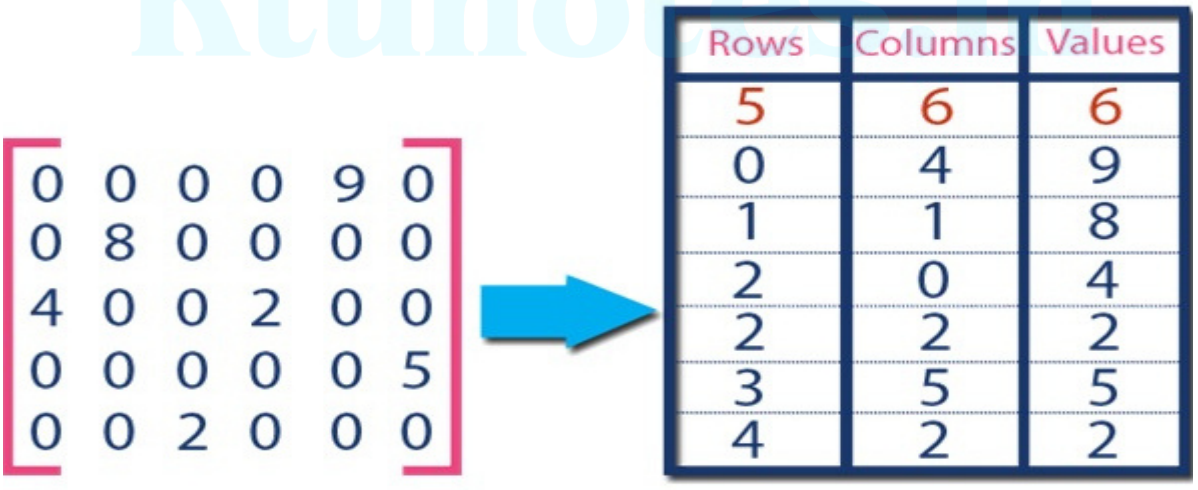
- In C programming ,a matrix can be defined with a two dimensional array.
- A matrix is a two dimensional array having ‘m’ rows and ‘n’ columns.
- A matrix with m rows and n columns is called m x n matrix.
- There may be a situation in which a matrix contains more number of zero values than non zero values.
- Such a matrix is known as Sparse matrix.
- *Sparse matrices are those matrices that have the majority of their elements equal to zero.In other words,the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.*



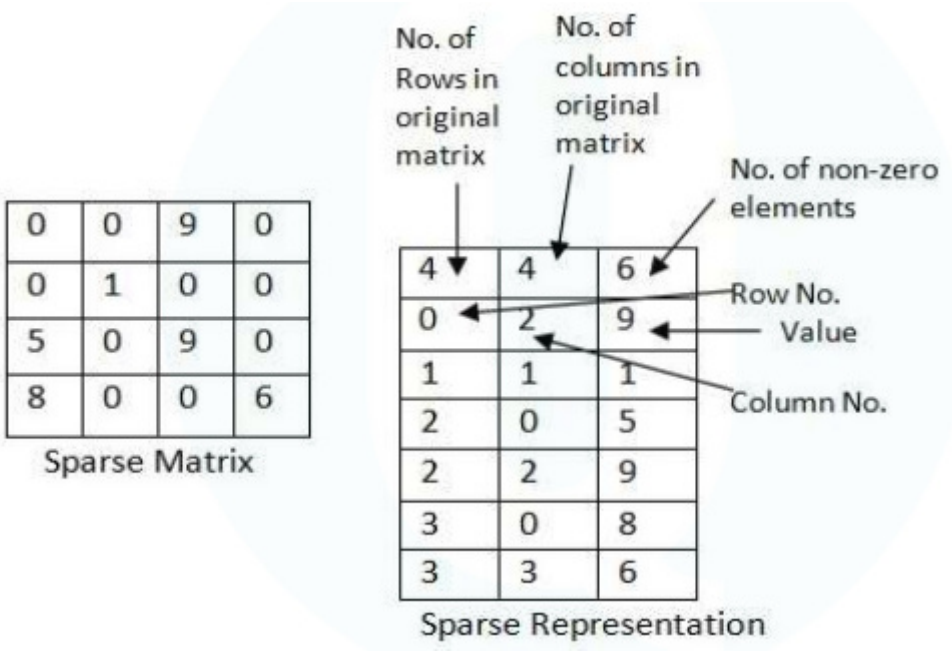
DENSE MATRIX:A dense matrix is one in which the majority of the elements are non zero values.

3 TUPLE REPRESENTATION OF A SPARSE MATRIX

- In this representation ,we consider only the non-zero values along with their row and column index values.
- In this representation ,the 0 th row stores the total number of rows,total number of columns and the total number of non-zero values in the sparse matrix.
- For example,consider a matrix of size 5 x 6 containing 6 non-zero values.



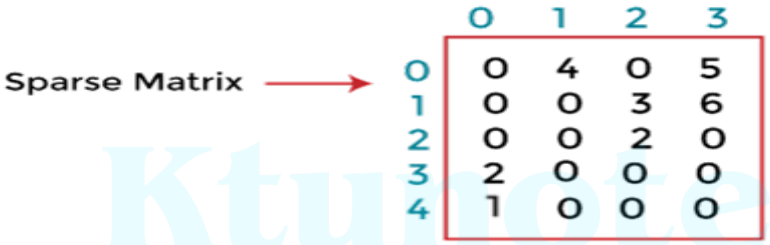
- In the above example matrix,there are only 6 non-zero elements(those are 9,8,4,2,5,2)and the matrix size is 5 x 6.
- We represent this matrix as shown in the above image.Here the first row in the right side table is filled with values 5,6 & 6 which indicates that it is a sparse matrix with 5 rows,6 columns & 6 non-zero values.
- The second row is filled with 0,4, & 9 which indicates the non-zero value 9 is at the 0 th-row and 4th column in the sparse matrix.
- In the same way,the remaining non-zero values also follow a similar pattern.



0	0	9	0
0	1	0	0
5	0	9	0
8	0	0	6

4	4	6
0	2	9
1	1	1
2	0	5
2	2	9
3	0	8
3	3	6

Example



SPARSE REPRESENTATION

ROW	COLUMN	VALUE
5	4	7
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
4	0	1

ADVANTAGES OF USING SPARSE MATRIX

Storage - We know that a sparse matrix contains lesser non-zero elements than zero, so less memory can be used to store elements. It evaluates only the non-zero elements.

Computing time: In the case of searching in a sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

DISADVANTAGES OF USING SPARSE MATRIX

- When a sparse matrix is represented with a 2-dimensional array,we waste a lot of space to represent that matrix.

PREVIOUS YEAR QUESTIONS

1. What is a sparse matrix?(IT DECEMBER 2020,2019 scheme)

   Ans:Refer topic sparse matrix

2. What is the complexity of finding maximum and minimum value from an array of n values? Explain the steps of deriving complexity.(DECEMBER 2018)

   Ans: Time Complexity is O(n) and Space Complexity is O(1).

3. Represent the following matrix using row major order and column major order.
   10 20 -32 44
   3 99 12 -20
   21 -4 33 89 (MAY 2019)
   Ans: Refer topics under representation of an array.

4. Write an algorithm/pseudo code to add a new element in a particular position of an array.(DECEMBER 2019)

Ans:

Algorithm

1. Get the element value which needs to be inserted.

2. Get the position value.

3. Check whether the position value is valid or not.

4. If it is valid,

Shift all the elements from the last index to position index by 1 position to the right.

insert the new element in arr[position]

5. Otherwise,

Invalid Position

5. Write an algorithm/pseudocode to delete a given element k from an array A of n elements? Assume that the element k is always present in A.(September 2020)

Ans: 1. Set **index** value as -1 initially. i.e. **index = -1**

2. Get **key** value from the user which needs to be deleted.

3. Search and store the index of a given key

[index will be -1, if the given key is not present in the array]

4. if index not equal to -1

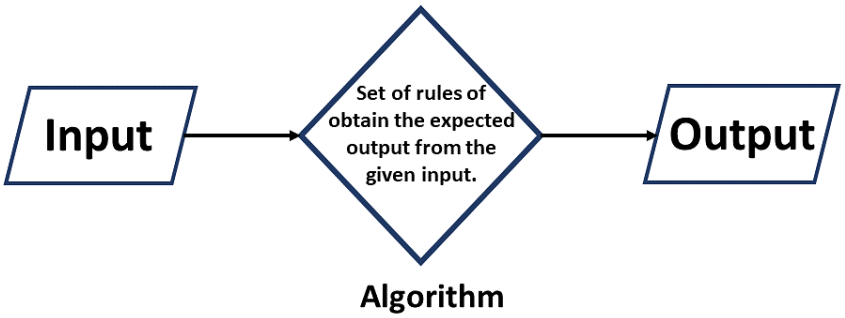
Shift all the elements from index + 1 by 1 position to the left.

5. else

print "Element Not Found"

What is an Algorithm?

- An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem-solving operations.
- According to its formal definition, an algorithm is a finite set of instructions carried out in a specific order to perform a particular task.
- It is not the entire program or code; it is simple logic to a problem represented as an informal description in the form of a flowchart or pseudocode.



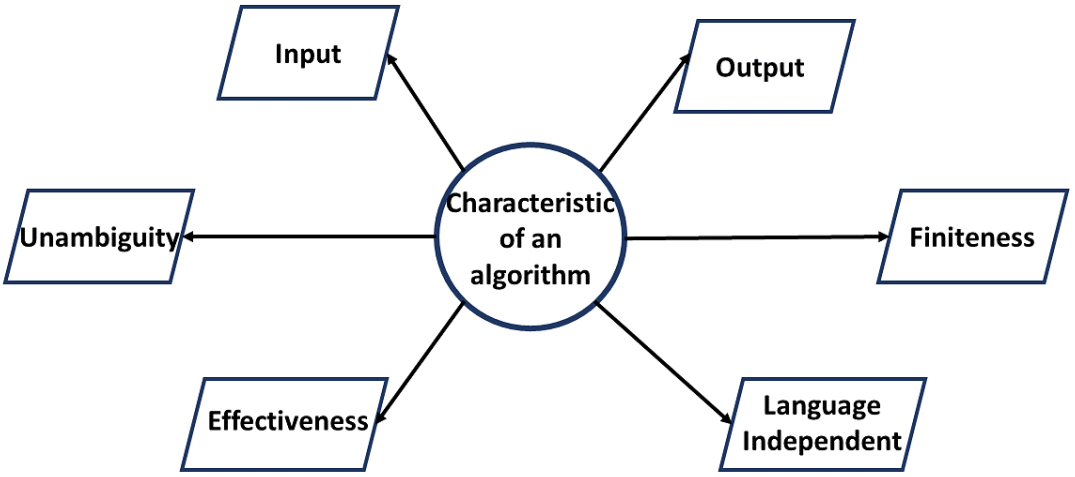
Algorithm

- Problem: A problem can be defined as a real-world problem or real-world instance problem for which you need to develop a program or set of instructions. An algorithm is a set of instructions.
- Algorithm: An algorithm is defined as a step-by-step process that will be designed for a problem.
- Input: After designing an algorithm, the algorithm is given the necessary and desired inputs.
- Processing unit: The input will be passed to the processing unit, producing the desired output.
- Output: The outcome or result of the program is referred to as the output.

After defining what an algorithm is, you will now look at algorithm characteristics.

Characteristics of an Algorithm

An algorithm has the following characteristics:



- Input: An algorithm requires some input values. An algorithm can be given a value other than 0 as input.
- Output: At the end of an algorithm, you will have one or more outcomes.
- Unambiguity: A perfect algorithm is defined as unambiguous, which means that its instructions should be clear and straightforward.
- Finiteness: An algorithm must be finite. Finiteness in this context means that the algorithm should have a limited number of instructions, i.e., the instructions should be countable.
- Effectiveness: Because each instruction in an algorithm affects the overall process, it should be adequate.
- Language independence: An algorithm must be language-independent, which means that its instructions can be implemented in any language and produce the same result.

What Is Time Complexity?

Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input. Time complexity is not a measurement of how much time it takes to execute a particular algorithm because such factors as programming language, operating system, and processing power are also considered.

Time complexity is a type of computational complexity that describes the time required to execute an algorithm. The time complexity of an algorithm is the amount of time it takes for each statement to complete. As a result, it is highly dependent on the size of the processed data. It also aids in defining an algorithm's effectiveness and evaluating its performance.

What Is Space Complexity?

When an algorithm is run on a computer, it necessitates a certain amount of memory space. The amount of memory used by a program to execute it is represented by its space complexity. Because a program requires memory to store input data and te A good algorithm executes quickly and saves space in the process. You should find a happy medium of space and time (space and time complexity), but you can do with the average. Now, take a look at a simple algorithm for calculating the "mul" of two numbers.

Best Case, Worst Case, and Average Case in Asymptotic Analysis

Best Case: It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time. In this case, the execution time serves as a lower bound on the algorithm's time complexity.

Average Case: You add the running times for each possible input combination and take the average in the average case. Here, the execution time serves as both a lower and upper bound on the algorithm's time complexity.

Worst Case: It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time possible. In this case, the execution time serves as an upper bound on the algorithm's time complexity.

PREVIOUS YEAR QUESTIONS

1.What do you understand about the complexity of an algorithm? Write worst case and best case

complexity of linear search.

2.Describe Big O notation used to represent asymptotic running time of algorithms.

Give the asymptotic analysis of any one iterative algorithm.

3. Write a recursive function to find the factorial of a given number. Write its time complexity

4.What is frequency count? With the help of an example, explain how frequency count is used to calculate the running time of an algorithm?

5.Give two methods through which the performance of an algorithm can be analyzed. Use an example each to illustrate.

ANSWERS

1. Algorithmic complexity is concerned about how fast or slow particular algorithm performs. Complexity is defined as a numerical function T(n) - time versus the input size n. Time taken by an algorithm is defined without depending on the implementation details.

For a list with n items, the best case of linear search is when the value is equal to the first element of the list, in which case only one comparison is needed. So time complexity in the best case would be Θ(1).

The worst case is when the value is not in the list (or occurs only once at the end of the list), in which case n comparisons are needed. Therefore, the worst case time complexity of linear search would be Θ(n).

2. Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- ☐ Big-O notation
- ☐ Omega notation
- ☐ Theta notation

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm. The time-complexity of 5.recursive factorial would be:

```
factorial (n) {
    if (n = 0)
        return 1
    else
        return n * factorial(n-1)
}
```


So,

The time complexity for one recursive call would be:

T(n) = T(n-1) + 3 (3 is for As we have to do three constant operations like multiplication,subtraction and checking the value of n in each recursive call)

= T(n-2) + 6 (Second recursive call)

= T(n-3) + 9 (Third recursive call)

.

.

.

.

= T(n-k) + 3k

till, k = n

Then,

= T(n-n) + 3n

= T(0) + 3n

= 1 + 3n

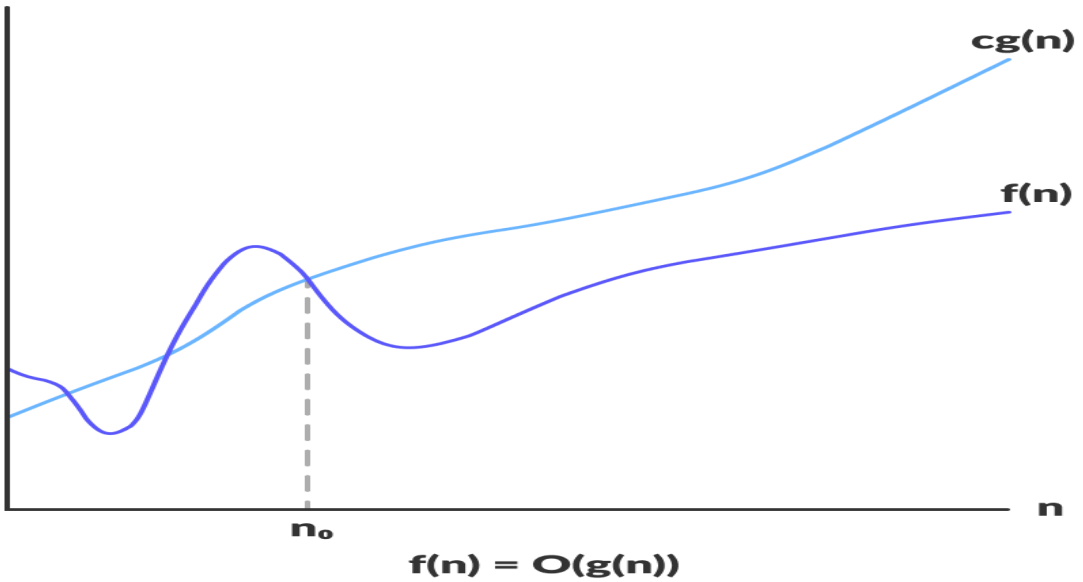
To represent in Big-Oh notation,

T(N) is directly proportional to n,

Therefore, The time complexity of a recursive factorial is O(n). As there is no extra space taken during the recursive calls,the space complexity is O(N).

6. Frequency count in the algorithm gives you how many times that your program is run, that is it depends on the loop used in the program.However the loop defines the time complexity of your program.

So it is mean that the frequency count defines the program time complexity.Therefore it is better to try to reduce the use of the loop because as the loop is more the program time complexity is more it mainly consist the polynomial time that is O(n^2) or O(n^3).



Big-O gives the upper bound of a function O(g(n)) = { f(n): there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$ }

SEARCHING

- Search is a very simple search algorithm.
- In this type of search, a sequential search is made over all items one by one.
- Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Linear Search

- Searching is a process of finding a particular data item from a collection of data items based on specific criteria.
- Every day we perform web searches to locate data items contained in various pages.
- A search is typically performed using a search key and it answers either True or False based on whether the item is present or not in the list.
- Linear search algorithm is the most simplest algorithm to do sequential search and this technique iterates over the sequence and checks one item at a time, until the desired item is found or all items have been examined.
- In linear search, we cannot determine that a given search value is present in the sequence or not until the entire array is traversed.

Algorithm:

1. Linear Search (Array A, Value x)
2.
3. Step 1: Set i to 1
4. Step 2: if i > n then go to step 7
5. Step 3: if A[i] = x then go to step 6
6. Step 4: Set i to i + 1
7. Step 5: Go to Step 2
8. Step 6: Print Element x Found at index i and go to step 8

9. Step 7: Print element not found
10. Step 8: Exit

Pseudocode:

```
int val;
scanf("%d",&val);
for (int i = 0; i < n; i++) {
if (A[i] == val) {
printf("Found %d at index %d\n", val, i);
break;
}
}
```

Time Complexity Of Linear Search:

- Any algorithm is analyzed based on the unit of computation it performs.
- For linear search, we need to count the number of comparisons performed, but each comparison may or may not search the desired item.

Case	Best Case	Worst Case	Average Case
If item is present	1	n	n/2
If item is not present	n	n	n

Binary Search

- In the Binary search algorithm, the target key is examined in a sorted sequence and this algorithm starts searching with the middle item of the sorted sequence.
- If the middle item is the target value, then the search item is found and it returns True.
    1. If the target item < middle item, then search for the target value in the first half of the list.
    2. If the target item > middle item, then search for the target value in the second half of the list.
- In binary search as the list is ordered, we can eliminate half of the values in the list in each iteration.

Algorithm:

- **Step 1**: Find middle element of the array.
- **Step 2**: Compare the value of the middle element with the target value.
- **Step 3**: If they match, it is returned.
- **Step 4**: If the value is less or greater than the target, the search continues in the lower or upper half of the array accordingly.
- **Step 5**: The same procedure as in **step 2-4** continues, but with a smaller part of the array. This continues until the target element is found or until there are no elements left.

Pseudocode:

```
\\ Assume int A[n] contains data
\\ & key the value to be searched
low = 0; high = n-1;
while (low <= high) {
mid = (low + high) / 2;
if (A[mid] == key) {
printf("Found at %d", mid);
return; }
if (key > A[mid]) {
low = mid+1; }
```

```
else {
high = mid-1; }
}
printf("Not Found");
```

Time Complexity Of Binary Search:

- In Binary Search, each comparison eliminates about half of the items from the list.
- Consider a list with n terms, then about n/2 items will be eliminated after first comparison. After second comparison, n/4 items of the list will be eliminated.
- If this process is repeated several times, then there will be just one item left in the list.
- The number of comparisons required to reach this point is n/2i= 1.
- If we solve for I, then it gives us I = log n.
- The maximum number comparison is logarithmic in nature, hence the time complexity of binary search is O(log n).

Case	Best Case	Worst Case	Average Case
If item is present	1	O(log n)	O(log n)
If item is not present	O(log n)	O(log n)	O(log n)

Previous Questions

1.What is linear search?

2.What is binary search?

3.What is the difference between linear and binary search?

4.Write an algorithm for linear search technique.

5.Write an algorithm for binary search technique.

6.Write the time complexity of linear search and binary search algorithm?

Sorting

Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order. A collection of records called a list where every record has one or more fields. The fields which contain a unique value for each record is termed as the key field. For example, a phone number directory can be thought of as a list where each record has three fields - 'name' of the person, 'address' of that person, and their 'phone numbers'. Being a unique phone number can work as a key to locate any record in the list.

Sorting is the operation performed to arrange the records of a table or list in some order according to some specific ordering criterion. Sorting is performed according to some key value of each record.

The techniques of sorting can be divided into two categories. These are:

- Internal Sorting
- External Sorting

Internal Sorting

Sorting algorithms that use main memory exclusively during the sort are called internal sorting algorithms. This kind of algorithm assumes high-speed random access to all memory. Some of the common algorithms that use this sorting feature are Bubble Sort, Insertion Sort, and Quick Sort.

Some common internal sorting algorithms include:

*Bubble Sort

*Insertion Sort

*Quick Sort

*Heap Sort

*Radix Sort

*Selection sort

External Sorting

Sorting algorithms that use external memory during the sorting come under this category. They are comparatively slower than internal sorting algorithms. For example, merge sort algorithms. It sorts chunks that each fit in RAM, then merges the sorted chunks together

Examples:

*Merge sort

*Tape sort

*Polyphase sort

*External radix

*External merge

Selection Sort Algorithm

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

The subarray which is already sorted.

The remaining subarray was unsorted.

In every iteration of the selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

How does selection sort work?

Let's consider the following array as an example: arr[] = {64, 25, 12, 22, 11}

First pass:

For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where 64 is stored presently, after traversing the whole array it is clear that 11 is the lowest value.

64 25 12 22 11

Thus, replace 64 with 11. After one iteration 11, which happens to be the least value in the array, tends to appear in the first position of the sorted list.

11 25 12 22 64

Second Pass:

For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.

11 25 12 22 64

After traversing, we found that 12 is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.

11 12 25 22 64

Third Pass:

Now, for third place, where 25 is present again, traverse the rest of the array and find the third least value present in the array.

11 12 25 22 64

While traversing, 22 came out to be the third least value and it should appear at the third place in the array, thus swap 22 with element present at third position.

11 12 22 25 64

Fourth pass:

Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array

As 25 is the 4th lowest value hence, it will place at the fourth position.

11 12 22 25 64

Fifth Pass:

At last the largest value present in the array automatically get placed at the last position in the array

The resulting array is the sorted array.

11 12 22 25 64

Bubble Sort Algorithm

Bubble Sort Algorithm is used to arrange N elements in ascending order, and for that, you have to begin with 0^{th} element and compare it with the first element. If the 0^{th} element is found greater than the 1^{st} element, then the swapping operation will be performed, i.e., the two values will get interchanged. In this way, all the elements of the array get compared.

How does bubble sort work?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with the very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.

The new array should look like this –

Next we compare 33 and 35. We find that both are in already sorted positions.

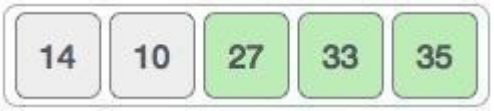
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller than 35. Hence they are not sorted.

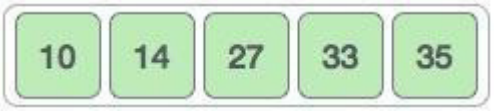
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted



Insertion sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in the sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of the sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of the third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list.

Algorithm

Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than
the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

The Complexity of Sorting Algorithm

The complexity of the sorting algorithm calculates the running time of a function in which 'n' number of items are to be sorted. The choice for which sorting method is suitable for a problem depends on several dependency configurations for different problems. The most noteworthy of these considerations are:

- The length of time spent by the programmer in programming a specific sorting program
- Amount of machine time necessary for running the program

- The amount of memory necessary for running the program

Time Complexities of all Sorting Algorithms

The efficiency of an algorithm depends on two parameters:

*** Time Complexity**
*** Space Complexity**

Time Complexity: Time Complexity is defined as the number of times a particular instruction set is executed rather than the total time taken. It is because the total time taken also depends on some external factors like the compiler used, processor's speed, etc.

Space Complexity: Space Complexity is the total memory space required by the program for its execution.

Both are calculated as the function of input size(n).

Types Of Time Complexity :

Best Time Complexity: Define the input for which algorithm takes less time or minimum time. In the best case, calculate the lower bound of an algorithm. Example: In the linear search when search data is present at the first location of large data then the best case occurs.

Average Time Complexity: In the average case take all random inputs and calculate the computation time for all inputs.

And then we divide it by the total number of inputs.

Worst Time Complexity: Define the input for which algorithm takes a long time or maximum time. In the worst case, calculate the upper bound of an algorithm. Example: In the linear search when search data is present at the last location of large data then the worst case occurs.

Previous year question papers

1- What is an algorithm? How is its complexity analyzed?

2- Give the heap sort algorithm. Write the complexity of your algorithm.

3- Write an algorithm/pseudo code to find the sum of two square matrices and find

the time complexity of the algorithm using frequency count method.

4- Design an algorithm/ pseudocode for selection sort. Illustrate the working of

selection sort on the following array with 7 elements : 30,45,25,32,55,60,49

5- Write the algorithm for insertion sort. Analyze its performance for sorted input.

6- Sort the following sequence using insertion sort

3, 10, 4, 2, 8, 6, 5, 1

7- What is a heap? Write an algorithm to perform heap sort

Quick sort

Quick Sort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quicksort that pick pivots in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot.
- Pick a random element as a pivot.
- Pick median as the pivot.

Quick sort algorithm basically takes the following steps

1. Choose a pivot element (the pivot element may be in any position). Normally the first element is chosen as pivot.
2. Perform the partition function in such a way that all the elements which are lesser than pivot go to the left part of the array and all the elements greater than pivot go to the right part of the array. The partition function also places pivot in the exact position.
3. Recursively perform a quick sort algorithm in these two sub arrays.

Quicksort Algorithm

```
QUICKSORT (array A, start, end)
{
  if (start < end)
  {
 p = partition (A, start, end)
 QUICKSORT (A, start, p - 1)
 QUICKSORT (A, p + 1, end)
 }
}
```

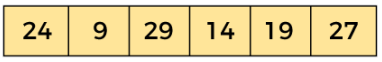
Partition

```
PARTITION (array A, start, end)
{
  pivot = A[end]
```

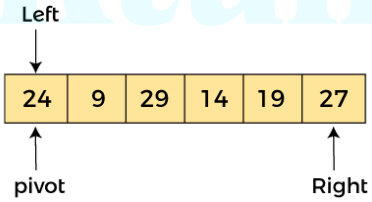
```
  i = start-1
  for j =start to end -1 {
  do if (A[j] < pivot) {
  then i = i + 1
  swap A[i] with A[j]
   }}
  swap A[i+1] with A[end]
  return i+1
}
```

Working of Quick Sort Algorithm

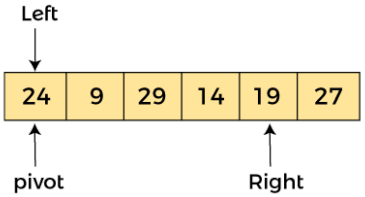
Let the elements of array are -



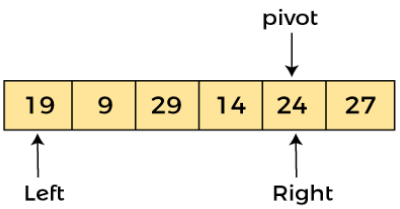
In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24. The pivot is at the left, so algorithm starts from right and move towards the left.



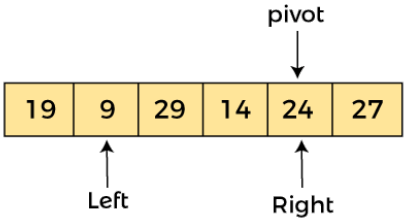
Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e.



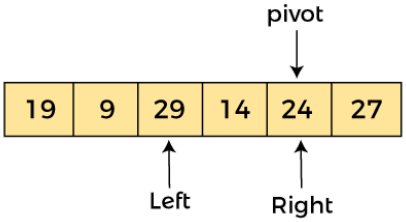
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24. Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as –



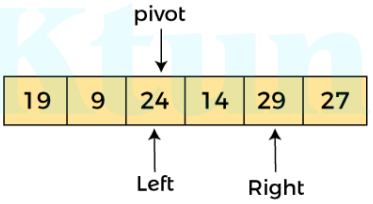
Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, the pivot is at right, so the algorithm starts from left and moves to right. As a[pivot] > a[left], so algorithm moves one position to right as -



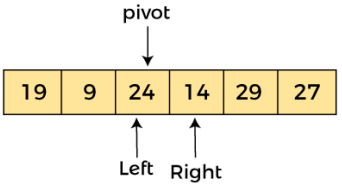
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



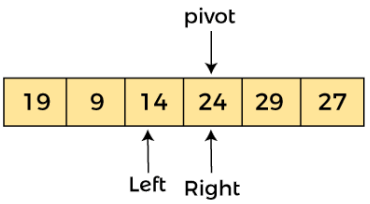
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



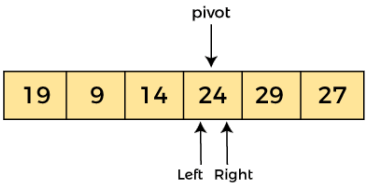
Since, the pivot is at left, so the algorithm starts from right, and moves to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



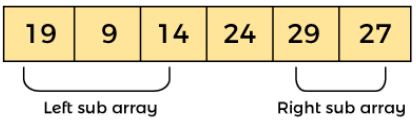
Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and moves to right.



Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing to the same element. It represents the termination of procedure. Element 24, which is the pivot element, is placed at its exact position. Elements that are on the right side of element 24 are greater than it, and the elements that are on the left side of element 24 are smaller than it.



Now, in a similar manner, the quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -

Time complexity Analysis

The recurrence relation of quicksort in worst case is

T(n)=T (n/2) +T (n/2) + (n^2)

T(n)=2T (n/2) + (n ^2)

Comparing with the standard recurrence equation

T(n)=aT (n/b) + f(n)

Substitute the values of a and b in n^log b a and compare it with f(n).

In this case a=2, b=2 then n^log b a=n

Whereas f(n)= n^ 2

Therefore, the time complexity of quick sort is O (n^2).

Best: O (n log(n))

Average: O (n log(n))

Worst: O (n^2)

Merge Sort

It is a Divide and Conquer technique.

- Divides the input array into two sub-lists.
- Here 2 sub-lists are again divided into 4 sub lists.
- The process is continued until subsists contain a single element.
- Then repeatedly merge these two sub lists to a single sub list. So that a sorted array is created from sorted sub lists.
- The process continues until a sub list contains all the elements that are sorted.

Algorithm

1. Start
2. if (start! = end)
    1. mid= (start+end) /2
    2. Mergesort (start, mid)
    3. Mergesort (mid+1, end)
    4. Merge (start, mid, end)
3. end if
4. stop

Algorithm for merge

Merge (start, mid, end)
1 i=start
2. j=mid+1
3. k=start
4. while i<=mid and j<= end do
    1. if a[i]<=a[j]
        1.temp[k]=a[i]
        2. i=i+1
        3. k=k+1
    2. Else
        1.Temp[k] =a[j]
        2. J=j+1
        3. k=k+1
    3. end if
5.end while
6. while i<=mid do
    1.temp[k]=a[i]
    2. i=i+1

    3. k=k+1
7. end while

8. While j<=end
    1.Temp[k]=a[j]
    2. j=j+1
    3. k=k+1
9. end while
10. k=start
11. while k<=end
    1.a[k]=temp[k]
    2. k=k+1
12.end while
13.stop.

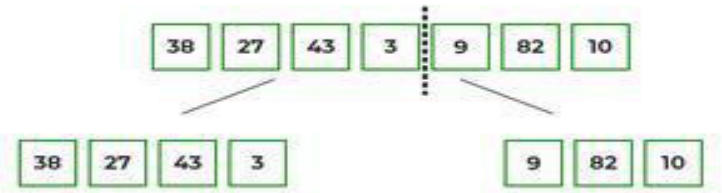
Working of Merge Sort

To know the functioning of merge sort, let's consider an array arr[ ] = {38, 27, 43, 3, 9, 82, 10}

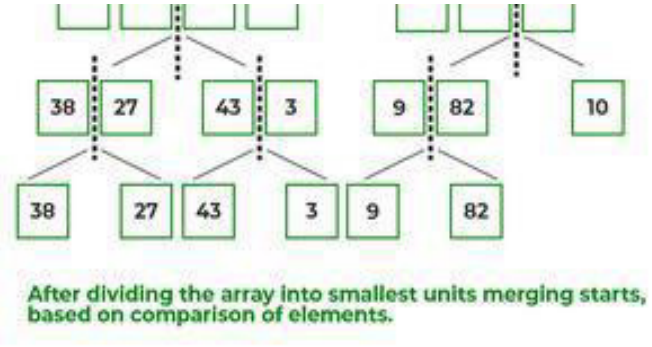
At first, check if the left index of the array is less than the right index, if yes then calculate its midpoint.

Now, as we already know, merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved. Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.

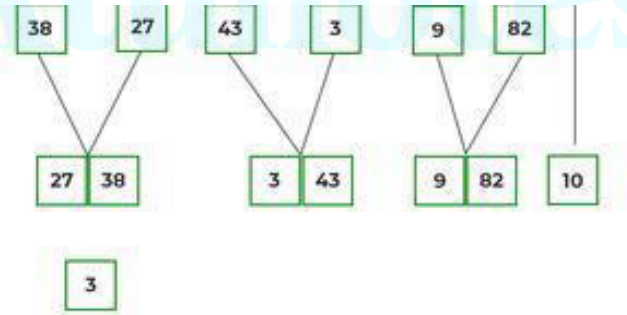
Now, again find that the left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.



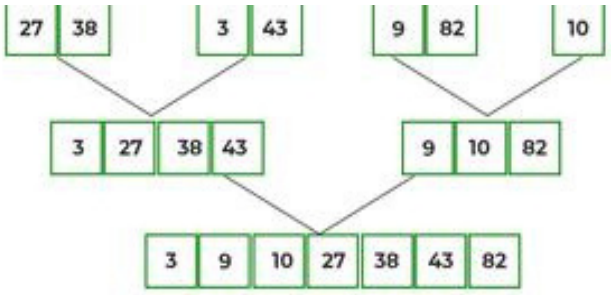
Now, further divide these two arrays into further halves, until the atomic units of the array are reached and further division is not possible.



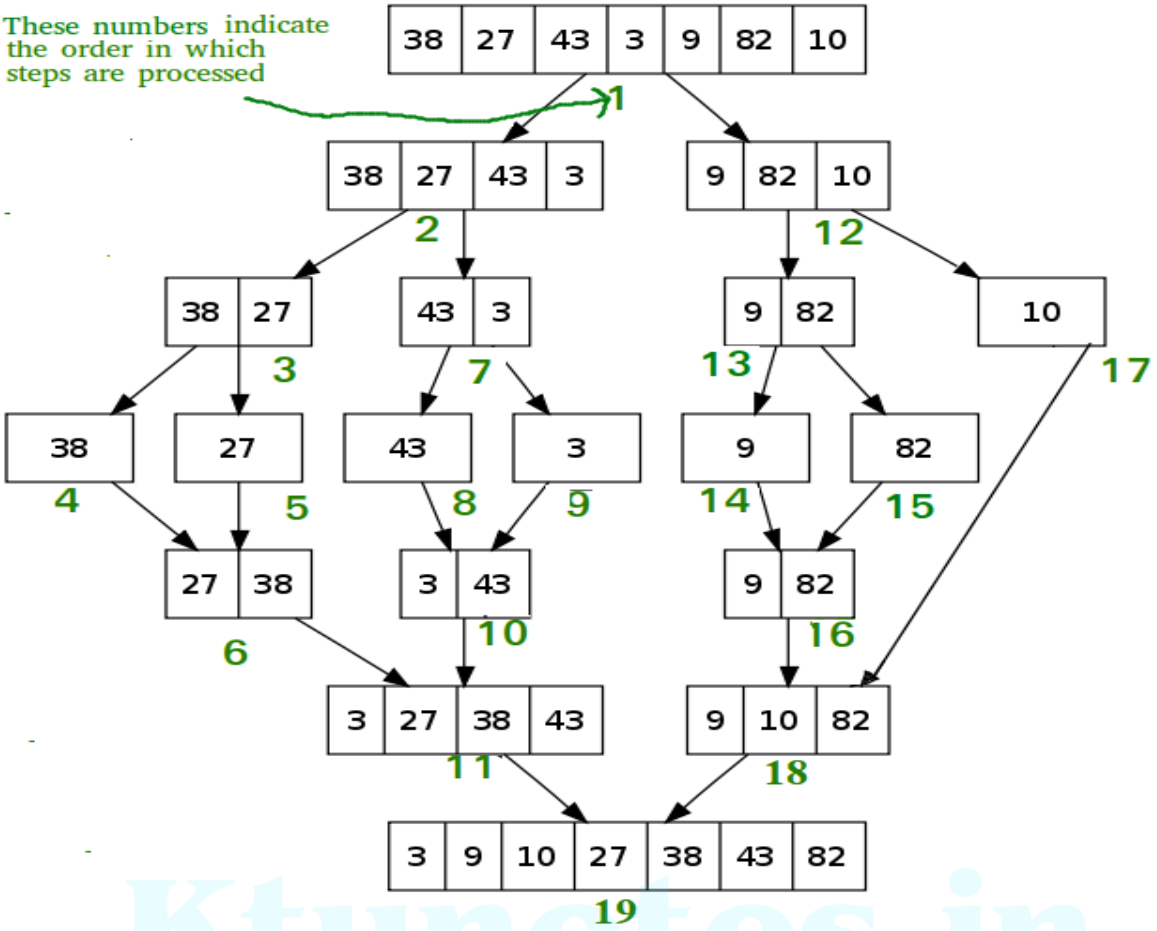
After dividing the array into smallest units, start merging the elements again based on comparison of size of elements. Firstly, compare the elements for each list and then combine them into another list in a sorted manner.



After the final merging, the list looks like this:



If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Analysis of mergesort

Suppose the merge sort follows the recurrence relation.

T(n)=2T(n/2) +n

Comparing with standard recurrence equation T(n)=aT(n/b) +f(n)

a=2, b=2, f(n)=n

then n^logb a=n

therefore, time complexity is O (n log(n)) in all the cases (Best, Average And Worst)

PREVIOUS YEAR QUESTIONS

1)Write a program to perform quick sort on a set of 'n' values given as input

2)Derive the worst case and average case,best case complexity of quick sort.

3)Write an algorithm for Quick sort.

4)Trace the working of the algorithm on the following input

38, 8, 0, 28, 45, -13, 89, 66, 42

5)Write an algorithm/ C program for merge sort technique. Illustrate with an example. Give its Complexity(best,worst, average).

6) Given the following list of numbers:

[21, 1, 26, 45, 29, 28, 2]

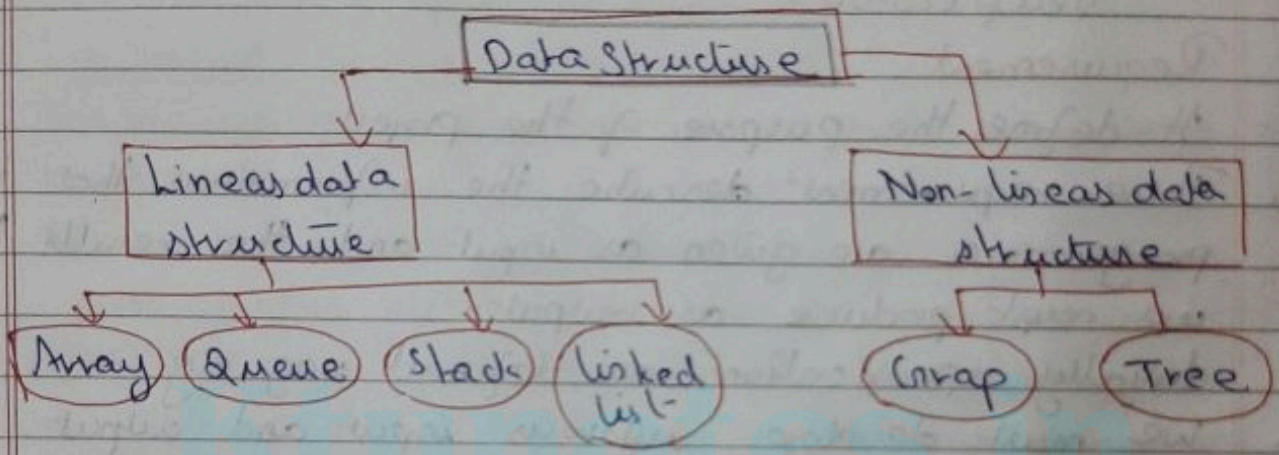
find the output obtained after each recursive call of merge sort algorithm.

Module - I

* Data Structure

→ A data structure is a particular way of organizing data in a computer so that it can be used effectively.

→ It is a specialized format for organizing, processing retrieving and storing data



→ The basic operations that are performed on data structures are

- Insertion
- Deletion
- Traversal
- Sorting
- Merging
- Searching

* System Life Cycle

→ System life cycle is a process of developing different components of a s/w system.

→ It discuss the tools and techniques necessary to design and implement large scale computer systems.

→ Good programmers regard large scale computer programs as systems that contain many complex interacting parts.

⇒ There programs undergo a development process called the system life cycle.

⇒ It consist of
- Requirements
- Analysis
- Design
- Refinement and coding
- Verification.

Requirement

⇒ It defines the purpose of the project.

⇒ There requirement describe the information that the programmes are given as input and the results that we must produce as output.

⇒ Initially specifications are defined vaguely.

⇒ We must develop rigorous input and output descriptions that include all cases.

Analysis

⇒ In this phase, we break the problem down into manageable pieces.

⇒ There are two approaches to analysis
- * Bottom up
- * Top down

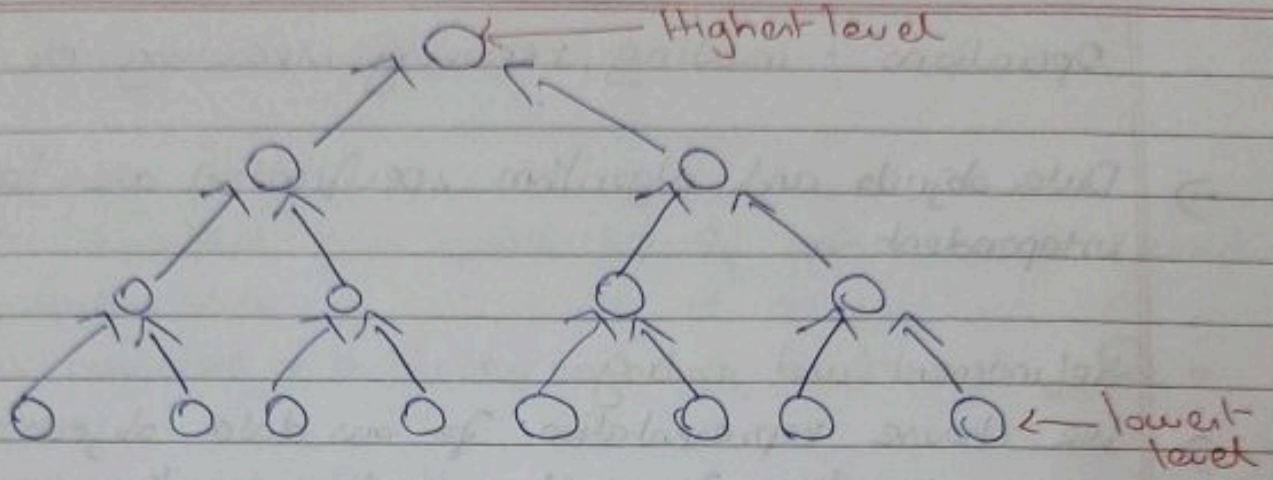
* Bottomup

⇒ It is older and unstructured strategy

⇒ The design starts with the lowest level components and subsystems.

⇒ By using these components, the next immediate highest level components and subsystems are created.

⇒ The process is continued till all the components and subsystems are composed into a single component, which is considered as a complete system

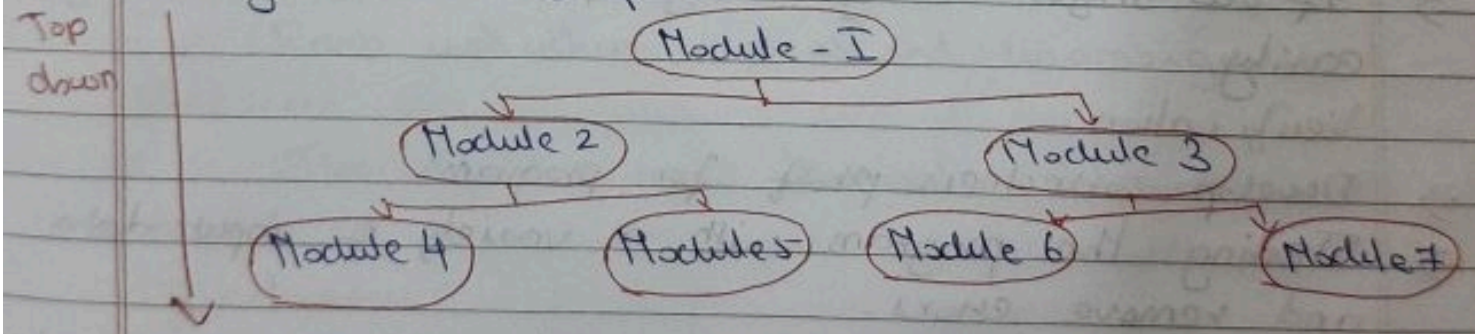


* Top-down

⇒ Each system is divided into several subsystems and components.

⇒ Each subsystem is futher divided into set of subsystems and components.

⇒ The design is started initially by defining the system as a whole and then keeps on adding definitions of the subsystem and components.



Design

⇒ The designes approaches the system from the perspective of both data objects that the program needs and the operations performed on them.

⇒ Data objects leads to the creation of abstract data types

⇒ Operation leads to the specification and design strategies & algorithms

⇒ Ex. Scheduling s/m of a university
Data objects - student, course, professor

Operations - inserting, removing, reaching ele.

⇒ Data objects and algorithm specification are language independent.

Refinement and coding

⇒ We choose representation for our data objects and write algorithms for each operations on them.
⇒ The order in which we do this is crucial because a data object's representation can determine the efficiency of the algorithms related to it.
⇒ Write the algorithms that are independent of the data objects, initially
⇒ Adapt better design changes.
⇒ The code for the system is written in this phase
⇒ If our original design is good, it can absorb changes easily.

Verification

⇒ Develop correctness proof for programs
⇒ Testing the program with a variety of input data and remove errors.
⇒ Correctness proof : programs can be proven correct using mathematical techniques
* These are very time consuming and difficult to develop for large projects.
* Select the algorithms that has been proven earlier and thus can reduce the number of errors
⇒ Testing : requires working code and set of test data
* Designed to test all posible scenarios
* Syntax and semantic errors are checked
* Program's running time is also considered.
⇒ Error removal - Remove errors depends on the design

and coding decisions made earlier.

* Algorithms

→ An algorithm is a finite set of instructions to accomplish a particular task.

→ It is considered as a finite number of set steps to solve a problem

→ A well defined algorithm always provides an answer and is guaranteed to terminate.

→ All algorithms must satisfy the following criteria:

Input

These are zero or more quantities that are externally supplied

Output

At least one quantity is produced

Definiteness

Each instruction is clear and unambiguous

Finiteness

The algorithm must terminate after a finite number of steps.

Effectiveness

Every instruction must be basic enough to be carried out. It must be feasible.

Performance Evaluation

Priori Estimates - theoretical analysis

→ It is machine independent technique

→ It determine the frequency count of each statement ie how many times a statement is executed

→ It is independent of the programming language in which the algorithm is written.

* Posteriori Testing - empirical analysis
→ It is a machine dependent technique
→ Consider the characteristics of the machine in which we run the algorithm
→ dependents on the language used to implement the algorithm.

* Performance Analysis
→ Analysing an algorithm means determining the amount of resources needed to execute it.
→ Performance analysis of an algorithm is performed by using
  * Space required to complete the task of that algorithm. It includes program space and data space.
  * Time required to complete the task of that algorithm.
→ Space complexity of a program is the amount of memory that is needed to run to completion
→ Time complexity of a program is the amount of computer time that it needs to run to completion.

* Space Complexity
The space needed by a program is the sum of
  * Fixed space requirements
  * Variable space requirements
x Fixed space
→ It do not depend on the number and size of the program's inputs and outputs.
→ It includes the space needed for storing instructions, constants, variables and structured variables (like arrays and structures)

Variable part-

→ It includes space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.

→ The variable space requirement of a program P working on an instance I is denoted as $S_p(I)$

The total space requirement $S(P)$ of any program is

$$S(P) = c + S_p(I)$$

where c is the fixed space

→ When analyzing the space complexity of a program, we consides only variable space requirement.

Eg: float abc(float a, float b, float c)
{
return a+b+b*c+(a+b-c)/(a+b)+4.00
}

This function has only fixed space requirement

$S(P) = c + S_p(I)$

$S_p(I) = 0$

Eg: float rsum(float list[], int n)
{ if(n)
return (rsum(list, n-1) + list(n-1))
return(0)
}

Type	Name	No: of bytes
parameters: array pointer	list[]	4
parameter: integes	n	4
returnaddress:		4
		12

The variable space is 12 for one time recursion

If n is the size of an array then
Space complexity = n x 12

* Time Complexity
→ The time T(P) taken by a program P is the sum of its compile time and its run time.
→ Time complexity only consider execution (run) time.
→ Types of time complexities are
* Worst case time complexity - The maximum value of f(n) for any input
* Average case time complexity - The expected value of f(n)
* Best case time complexity - The minimum possible value of f(n)

Ex: $5n^4 + 7n^3 + 10n^2 + n + 100$
Then Time complexity = $O(n^4)$

1.
```
int sum(int a[], int n)        → 0
{                              → 0
s = 0;                         → 1
for(i=0; i<n; i++)             → (n+1)
s = s + a[i];                  → n
return s;                      → 1
}                              → 0
                               ---
                               2n+3
```
ie O(n)

2.
```
float sum(float list[], int n)      s/e   F.   Total step
{                                    0     0      0
float tempsum = 0                    0     0      0
int i;                               1     1      1
for(i=0; i<n; i++)                   0     0      0
                                     1   (n+1)    n+1
```

```
tempsum += list[i];          1      n      n
return tempsum;              1      1      1
}                            0      0      0
                                          2n+3
```

$O(n)$

3.
```
float rsum(float list[], int n)          0      0      0
{                                        0      0      0
if(n)                                    1     n+1    n+1
return rsum(list, n-1) + list[n-1];      n      n      n
return list[0];                          1      1      1
}                                        0      0      0
                                                      2n+1
```

$O(n)$

* Linear loops

eg `for(i=0; i<n; i++)` $T(n) = n$
```
    stmt
```

2. `for(i=0; i<n; i=i+2)` $T(n) = n/2$
```
    stmt
```

* Logarithmic loop

```
for(i=1; i<n; i=i*2)
stmt
```
$T(n) = \log n$
```
for(i=n; i>=1; i=i/2)
  stmt
```

* Linear logarithmic loop

```
for(i=0; i<n; i++)
  for(k=1; k<n; k=k*2)
```
$T(n) = n \log(n)$

* Quadratic loop

```
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
```
$T(n) = n^2$

4. void add()
{ int i, j, m, n, a[] → 0
for(i=0; i<n; i++) → n+1
for(j=0; j<m; j++) → n(m+1)
s = s+a[i][j] → n*m
}
2nm + 2n + 1

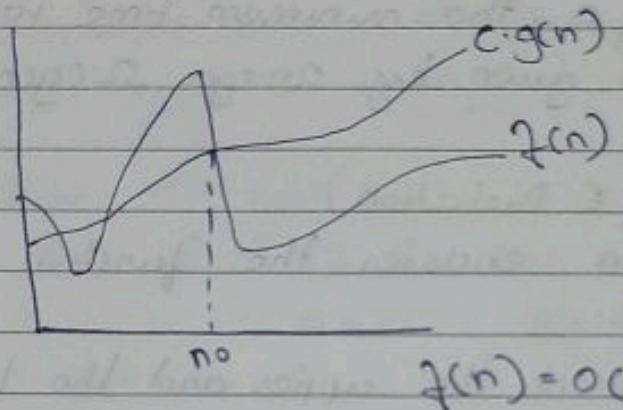
* Growth rate
Fundions in order of increasing growth rate is as follows

fast ↑	log N	n	O(1)	O(log n)	O(n)	O(n log n)	O(n²)	O(n³)
	N	1	1	1	1	1	1	1
	N log(N)	2	1	1	2	2	4	8
	N^2	4	1	2	4	8	16	64
	N^3	8	1	3	8	24	64	512
slow ↓	2^N	16	1	4	16	64	256	4096

* Asymptotic Notations
Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value.
An algorithm may not have the same performance for different types of inputs With the increase in the input size, the performance will change.
The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis
Ex: Bubble sort - input array sorted - best case
- reverse condition - worst case
- neither sorted nor reverse - average time

Big O Notation (O notation)

Big O notation represents the upper bound of the running time of an algorithm.
Thus it gives the worst-case complexity of an algorithm



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

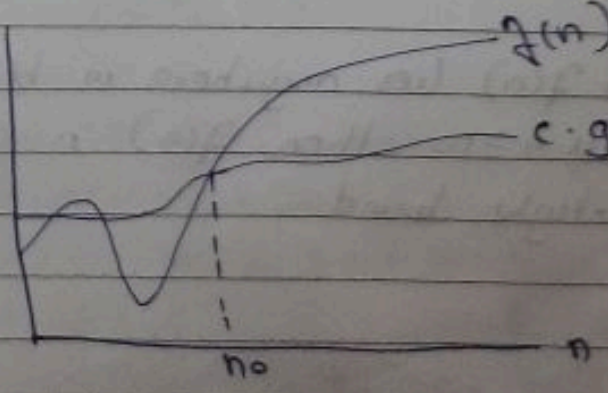
$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0 \}$

⇒ For any value of n, the running time of an algorithm does not cross time provided by $O(g(n))$

⇒ It is the measure of longest amount of time it could possibly take for an algorithm to complete.

Omega Notation (Ω notation)

Omega notation represents the lower bound of the running time of an algorithm.
Thus it provides the best case complexity of an algorithm.



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

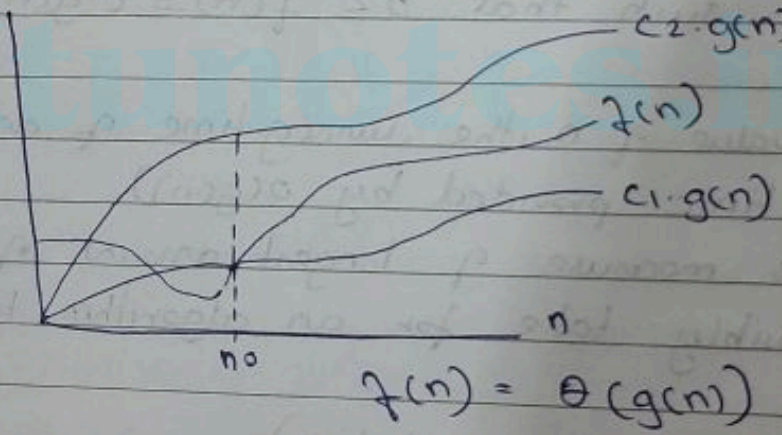
$\Omega(g(n)) = \{ f(n)$: there exist positive constant c and n_0 such that $0 \leq c\,g(n) \leq f(n)$ for all $n \geq n_0 \}$

For any value of n, the minimum time required by the algorithm is given by Omega $\Omega(g(n))$

Theta Notation (Θ Notation)

Theta notation encloses the function from above and below.

Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average case complexity of an algorithm.



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

$\Theta(g(n)) : \{ f(n)$: there exist positive constant c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$

If a function $f(n)$ lies anywhere in b/w $c_1 g(n)$ and $c_2 g(n)$ for all $n \geq n_0$, then $f(n)$ is said to be a asymptotically tight bound.

? $3n^2 + 8n + 4$ is $O(n^2)$

$3n^2 + 8n + 4 \leq 3n^2 + 8n^2 + 4n^2$

when $n \geq 1$ $(k = 1)$

$3n^2 + 8n + 4 \leq 15n^2$

$c = 15$

$O(n^2)$

? Prove that $n^3 + 20n^2 + 2 \neq O(n)$ when $n \geq 10$

$n^3 + 20n^2 + 2 \leq n + 20n + 2n$

$n \geq 10$ $k = 10$

$3002 \not\leq 230$

no constant

? Derive Big oh notation for $f(n) = 2n + 3$

$2n + 3 \leq 2n + 3n$

$2n + 3 \leq 5n$ $\forall$ $n \geq 1$

Hence $c = 5$

$g(n) = n$

$f(n) = O(n)$

? Derive Big Omega notation for $f(n) = 2n + 3$

$2n + 3 \geq 1n$ for all $n \geq 1$

$c = 1$

$g(n) = n$

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

? Derive the Big o notation of $n^2 + 2n + 5$

$f(n) = n^2 + 2n + 5$

$n^2 + 2n + 5 \leq n^2 + 2n^2 + 5n^2$

$\leq 8n^2$ $\forall$ $n \geq 1$

so $c = 8$ $f(n) \leq c \cdot g(n)$

$O(n^2)$

7 $f(n) = 2n^2 + 3n + 4$

$\Rightarrow 2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2$

$\leq 9n^2 \quad \forall\ n \geq 1$

$c = 9 \quad g(n) = n^2$

$f(n) \leq c \cdot g(n)$

$O(n^2)$

7 $2n^2 + 3n + 4 \geq 1n^2$

$c = 1 \quad g(n) = n^2$

$\Omega(n^2)$

$1n^2 \leq 2n^2 + 3n + 4 \leq 9n^2$

$\Theta(n^2)$