VIDUSH SOMANY INSTITUTE OF TECH AND RESEARCH

Question Bank

Design and Analysis of Algorithm/Fundamentals of Algorithms(CE/CSE/IT)

CE504-N/IT504-N

Q1 Give the definition of algorithm with examples.

A1 Definition of an algorithm:

An algorithm is a well defined computational procedure that takes some value or set of values as input and produces some value or set of values as output. An algorithm is thus a sequence of computational steps that transform the input into output. We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship.

Example of an algorithm:

We need to sort a sequence of numbers into

Nondecreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard techniques and analysis tools. Here is how we formally define the sorting problem:

Input: A sequence of n numbers <a1,a2,……..,an>

Output: A permutation (reordering) <a'1,a'2,…….,a'n> of the input sequence such that a'1<=a'2<=……..<=a'n.

For example, given the input sequence <31,41,59,26,41,58>, a sorting algorithm returns an output sequence <26,31,41,41,58,59>.Such an input sequence is called as an instance of the sorting problem. In general, an instance of a problem consists the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Because many problems use it as an intermediate step, sorting is a fundamental operation in computer science. As a result, we have a large number of good sorting algorithms at our disposal.

Which algorithm is best for a given application depends on among other factors- the number of items to be sorted, the extent to which the items are already somewhat sorted, possible restrictions on the item values, the architecture of the computer, and the kind of storage devices to be used: main memory, disks, or even tapes.

An algorithm can be specified in English as a computer program, or even as a hardware design. The only requirement is that the specification must provide a precise description of the computational procedure to be followed.

Practical application of algorithms are ubiquitous and include the following examples:

- The Human Genome Project has made great progress towards the goals of identifying all the 1,00,000 genes in human DNA, determining the sequences of the 3 billion chemical base pairs that make up human DNA, storing this information in databases, and developing tools for data analysis. Each of these steps requires sophisticated algorithms.
- The Internet enables people all around the world to quickly access and retrieve large amounts of information. With the aid of clever algorithms, sites on the Internet are able to manage and manipulate this large volume of data. Examples of problems that make essential use of algorithms include finding good routes on which the data

  Will travel (techniques for solving such problems appear)

  And using a search engine to quickly find pages to which

  Particular information resides.
- Electronic commerce enables goods and services to be negotiated and exchanged electronically, and it depends on the privacy of personal information such as credit

card numbers, passwords and bank statements. The core technologies used in electronic commerce include public key cryptography and digital signatures which are based on numerical algorithms and number theory.

- Manufacturing and other commercial enterprises often need to allocate scarce resources in the most beneficial way. An oil company may wish to know where to place its wells in order to maximize its expected profit. A political candidate may want to determine whether to spend money buying campaign advertising in order to maximize the chances of winning an election. An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and the government regulations regarding crew scheduling are met.An Internet service provider may wish to determine where to place additional resources in order to serve its customers more effectively. All of these are examples of problems that can be solved using linear programming.

We can also solve some specific problems including the following:

- We are given a road map on which the distance between each pair of adjacent intersections is marked, and we wish to determine the shortest route from one intersection to another. The number of possible routes can be huge, even if we disallow routes that cross over themselves. How do we choose which of all possible routes is the shortest? Here, we model the roadmap (which is itself a model of the actual roads) as a graph and we wish to find the shortest path from one vertex to another in the graph.

- We are given two ordered sequences, X = <x1,x2,……,xm> and Y = <y1,y2,……,yn>, and we wish to find a longest common subsequence of X and Y. A subsequence of X is just X with some (or possibly all or none) of its elements is removed. For example, one subsequence of X and Y gives one measure of how similar these two subsequences are. For example, if the two sequences are base pairs in DNA strands, then we might consider them similar if they have a long common subsequence. If X has m symbols and Y has n symbols, then X and Y have 2^m and 2^n possible subsequences, respectively. Selecting all possible subsequences of X and Y and matching them up could take a prohibitively long time unless m and n are very small.
- We are given a mechanical design in terms of a library of parts, where each part may include instances of other parts, and we need to list the parts in order so that each part appears before any part that uses it. If the design comprises n parts, then there are n! possible orders, where n! denotes the factorial function.Because the factorial function grows faster than even an exponential function, we cannot feasibly generate each possible order and then verify that, within that order, each part appears before the parts using it(unless we have only a few parts). This problem is an instance of topological sorting.
- We are given n points in the plane, and we wish to find the convex hull of these points. The convex hull is the smallest convex polygon containing the points. Intuitively, we can think of each point as being

represented by a nail sticking out from the board.The convex hull would be represented by a tight rubber band that surrounds all the nails. Each nail around which a rubber band makes a turn is a vertex of the convex hull. Any of the 2^n subsets of the points might be the vertices of the convex hull. Knowing which points are vertices of the convex hull is not quite enough, either, since we also need to know the order in which they appear.

Q2 How can we compare the performance analysis/efficiency of Algorithms?

A2 The performance of algorithms can be characterized on the basis of:

- Space Complexity
- Time Complexity

- Space Complexity:

  Space complexity deals with the extra memory required by the algorithm.

- Time Complexity:

  Time complexity indicates how fast the algorithm runs. Time efficiency is analyzed by determining the number of
  Repetitions of the basic operation as a function of input size. Basic operation: The operation that contributes most

towards the running time of the algorithm. The running time of an algorithm is the function defined by the number of steps (or amount of memory) required to solve input instances of size n.

Q3. What do you mean by Combinatorial Problem?
A3. Combinatorial Problems are problems that ask to find a combinatorial object- such as permutation, a combination, or a subset-that satisfies certain constraints and has some desired property.

Q4. What are the characteristics of an algorithm?
A4. Every algorithm should have the following five characteristics.

i. Input
ii. Output
iii. Definiteness
iv. Effectiveness
v. Termination

Therefore an algorithm can be defined as a sequence of definite and effective instructions, which terminates with the production of correct output from the given input. In other words, viewed little more formally, an algorithm is a step by step formalization of a mapping function to map input set onto an output set.

Q5. What do you mean by Amortized Analysis?

A5. Amortized Analysis finds the average running time per operation over a worst case sequence of operations. Amortized analysis differs from average case performance in that probability is not involved.Amortized analysis guarantees the time per operation over worst case performance.

Q6. What are important problem types? (or) Enumerate some important types of problems.

A6.

1. Sorting
2. Searching
3. Numerical problems
4. Geometric problems
5. Combinatorial Problems
6. Graph Problems
7. String processing Problems

Q7. What are the algorithm design techniques?

A7. Algorithm design techniques (or strategies or paradigms) are general approaches to solving problems algorithmatically, applicable to a variety of problems from different areas of computing. General design techniques are:

(i) Brute force

(ii) divide and conquer

(iii) decrease and conquer

(iv) transform and conquer

(v) greedy technique

(vi) dynamic programming

(vii) backtracking

(viii) branch and bound

Q8. Analyze the time complexity of the following segment:

```
for(i=0;i<N;i++)
for(j=N/2;j>0;j--)
sum++;
```

A8. Time Complexity= N * N/2

= N2 /2

Є O(N2)

Q9. Explain the various asymptotic notations with examples.

A9. Asymptotic Notation: When it comes to analysing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the space required by the algorithm, instead we express it using some standard notations, also known as Asymptotic Notations.

Types of Asymptotic Notations:-

We use three types of asymptotic notations to represent the growth of any algorithm, as input increases:

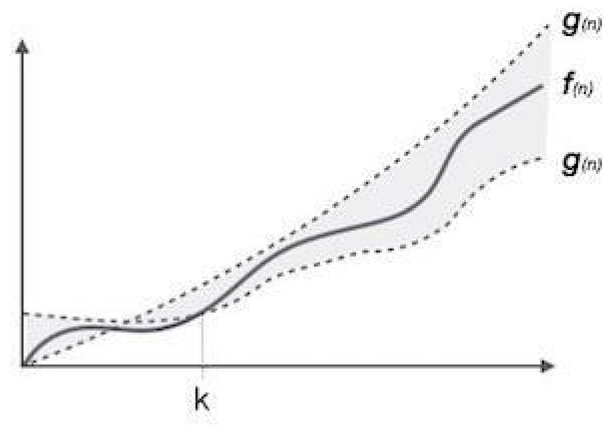
1. Big Theta (Θ)
2. Big Oh(O)
3. Big Omega (Ω)

Tight Bounds: Theta

When we say tight bounds, we mean that the time compexity represented by the Big-Θ notation is like the average value or range within which the actual time of execution of the algorithm will be.

For example, if for some algorithm the time complexity is represented by the expression 3n2 + 5n, and we use the Big-Θ notation to represent this, then the time complexity would be Θ(n2), ignoring the constant coefficient and removing the insignificant part, which is 5n.

Here, in the example above, complexity of Θ(n2) means, that the average time for any input n will remain in between, k1 * n2 and k2 * n2, where k1, k2 are two constants, thereby tightly binding the expression representing the growth of the algorithm.



Upper Bounds: Big-O

This notation is known as the upper bound of the algorithm, or a Worst Case of an algorithm.

It tells us that a certain function will never exceed a specified time for any value of input n.

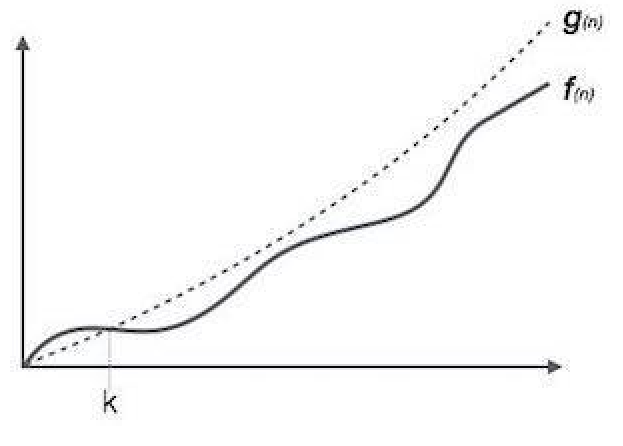
The question is why we need this representation when we already have the big-Θ notation, which represents the tightly bound running time for any algorithm. Let's take a small example to understand this.

Consider Linear Search algorithm, in which we traverse an array elements, one by one to search a given number.

In Worst case, starting from the front of the array, we find the element or number we are searching for at the end, which will lead to a time complexity of n, where n represents the number of total elements.

But it can happen, that the element that we are searching for is the first element of the array, in which case the time complexity will be 1.

Now in this case, saying that the big-Θ or tight bound time complexity for Linear search is Θ(n), will mean that the time required will always be related to n, as this is the right way to represent the average time complexity, but when we use the big-O notation, we mean to say that the time complexity is O(n), which means that the time complexity will never exceed n, defining the upper bound, hence saying that it can be less than or equal to n, which is the correct representation.

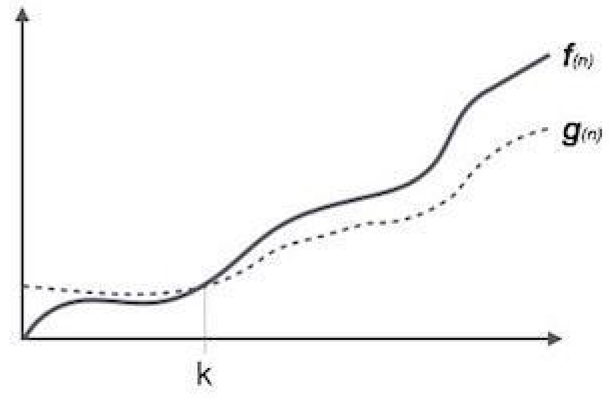


Lower Bounds: Omega

Big Omega notation is used to define the lower bound of any algorithm or we can say the best case of any algorithm.

This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.

In simple words, when we represent a time complexity for any algorithm in the form of big-Ω, we mean that the algorithm will take atleast this much time to complete it's execution. It can definitely take more time than this too.



Q10.Explain the insertion sort algorithm with example.

A10.

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

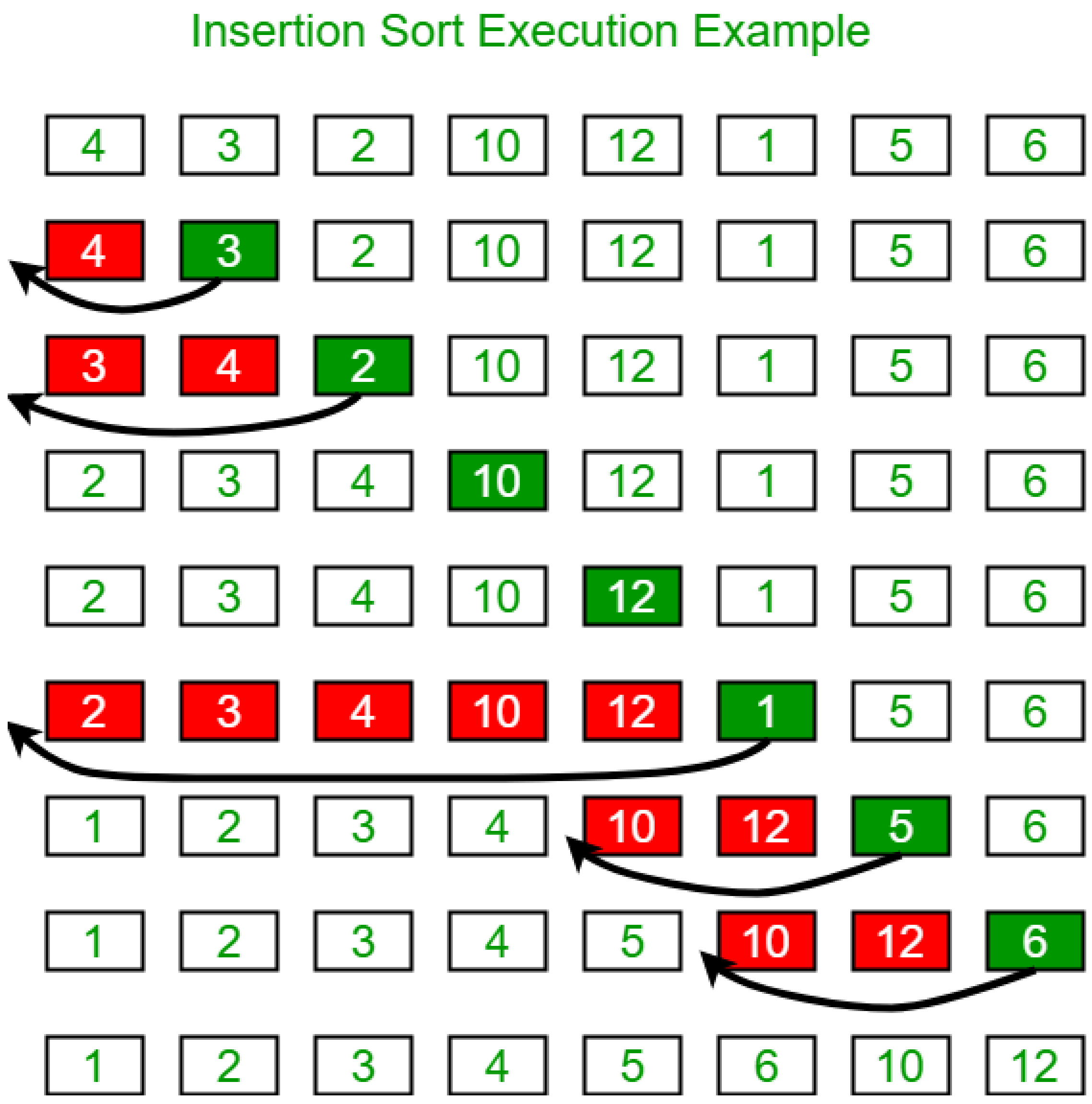
To sort an array of size n in ascending order:

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Example:



Q11.Explain the Radix Sort algorithm with example.

A11. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

Example:

Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives:

[*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs
170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives:

[*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by the most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802.

Q12.Explain Selection Sort with suitable example.

A12. The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.

2) Remaining subarray which is unsorted.

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

Following example explains the above steps:

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Q13. Explain the Bubble sort with suitable example.

A13. Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

First Pass:

(5 1 4 2 8) –> (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 5 4 2 8) –> (1 4 5 2 8), Swap since 5 > 4

(1 4 5 2 8) –> (1 4 2 5 8), Swap since 5 > 2

(1 4 2 5 8) –> (1 4 2 5 8), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(1 4 2 5 8) –> (1 4 2 5 8)

(1 4 2 5 8) –> (1 2 4 5 8), Swap since 4 > 2

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:

(1 2 4 5 8) –> (1 2 4 5 8)

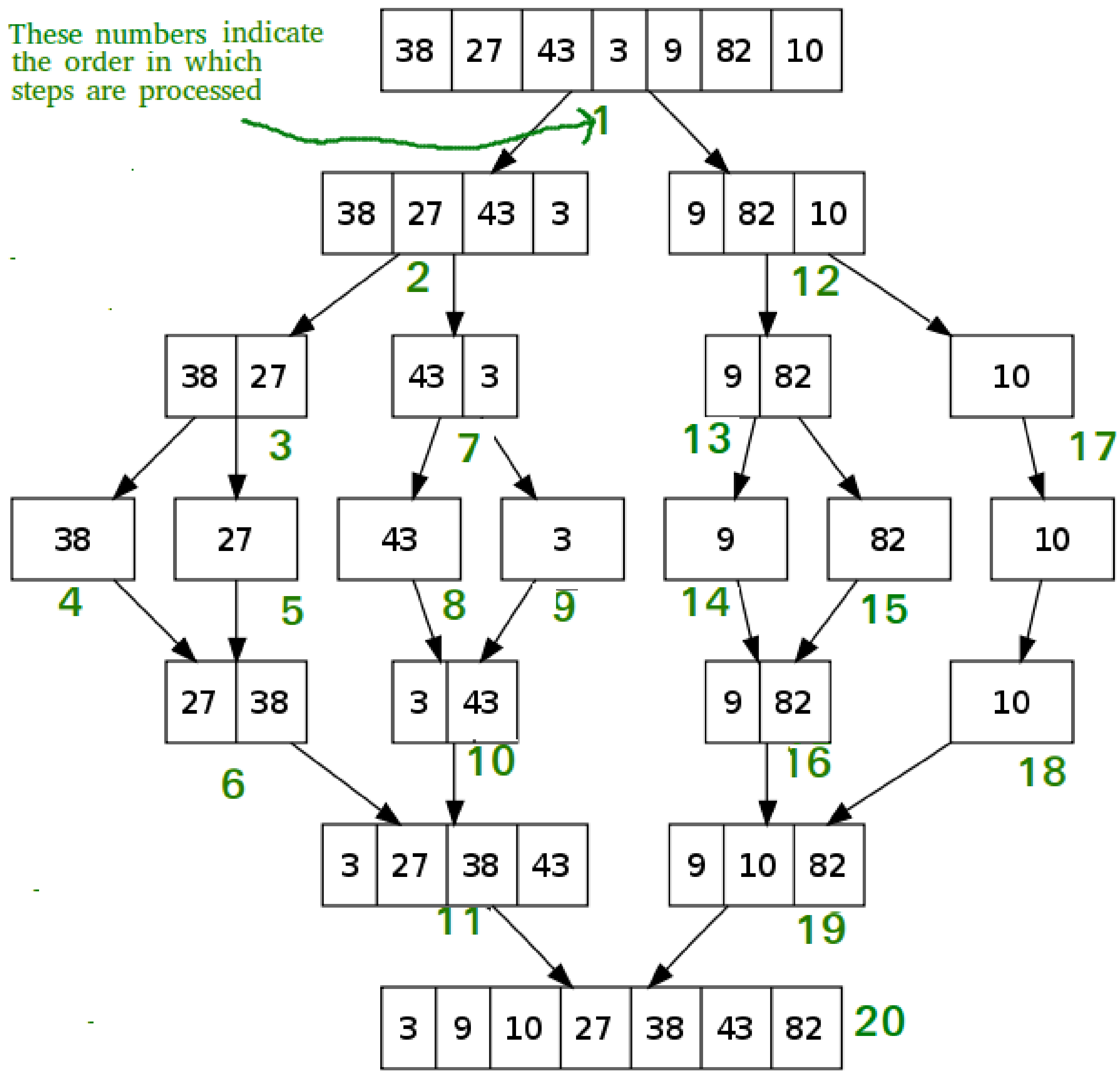
(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

Q14. Explain the Merge Sort with suitable example.

A14. Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.



Q15.Explain the recurrence relation and list the methods to solve the recurrence and explain any one method.

A15. A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Recurrences are generally used in divide and conquer paradigm.

Let us consider T(n) to be the running time on a problem of size n.

If the problem size is small enough, say $n<c$ where c is a constant,the straightforward solution takes constant time which is written as $\Theta(1)$.

If the division of the problem yields a number of subproblems with size n/b.

A recurrence relation can be solved using the following methods:

Substitution Method

Recursion Tree Method

Master's Theorem

Master's Theorem:-

Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$

There are following three cases:

1. If f(n) = O(nc) where c < Logba then T(n) = Θ(nLogba)

2. If f(n) = Θ(nc) where c = Logba then T(n) = Θ(ncLog n)

3.If f(n) = Ω(nc) where c > Logba then T(n) = Θ(f(n))

Q16. Explain the Quick Sort with example.

A16. It is an algorithm of Divide & Conquer type.

Divide: Rearrange the elements and split arrays into two sub-arrays and an element in between search that each element in left sub array is less than or equal to the average element and each element in the right sub- array is larger than the middle element.

Conquer: Recursively, sort two sub arrays.

Combine: Combine the already sorted array.

Example of Quick Sort:

1. 44 33 11 55 77 90 40 60 99 22 88

Let **44** be the **Pivot** element and scanning done from right to left

Comparing **44** to the right-side elements, and if right-side elements are **smaller** than **44**, then swap it. As **22** is smaller than **44** so swap them.

22 33 11 55 77 90 40 60 99 **44**
88

Now comparing **44** to the left side element and the element must be **greater** than 44 then swap them. As **55** are greater than **44** so swap them.

`22  33  11` **44** `77  90  40  60  99` **55** `88`

Recursively, repeating steps 1 & steps 2 until we get two lists one left from pivot element **44** & one right from pivot element.

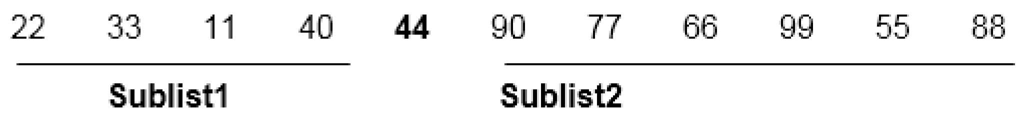
`22  33  11` **40** `77  90` **44** `60  99  55  88`

Swap with 77:

`22  33  11  40` **44** `90` **77** `60  99  55  88`

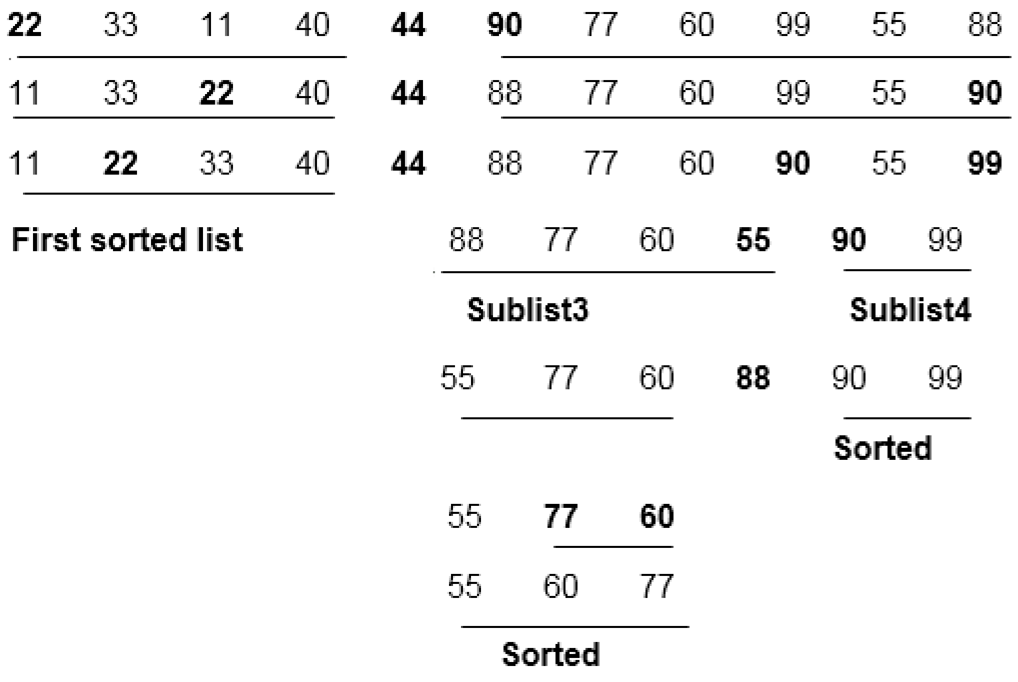
Now, the element on the right side and left side are greater than and smaller than **44** respectively.

Now we get two sorted lists:

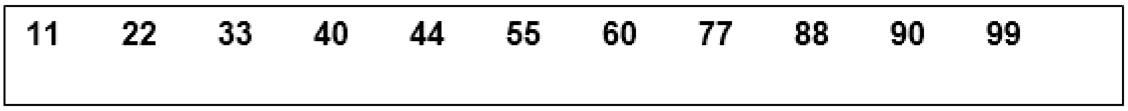


And these sublists are sorted under the same process as above done.

These two sorted sublists side by side.



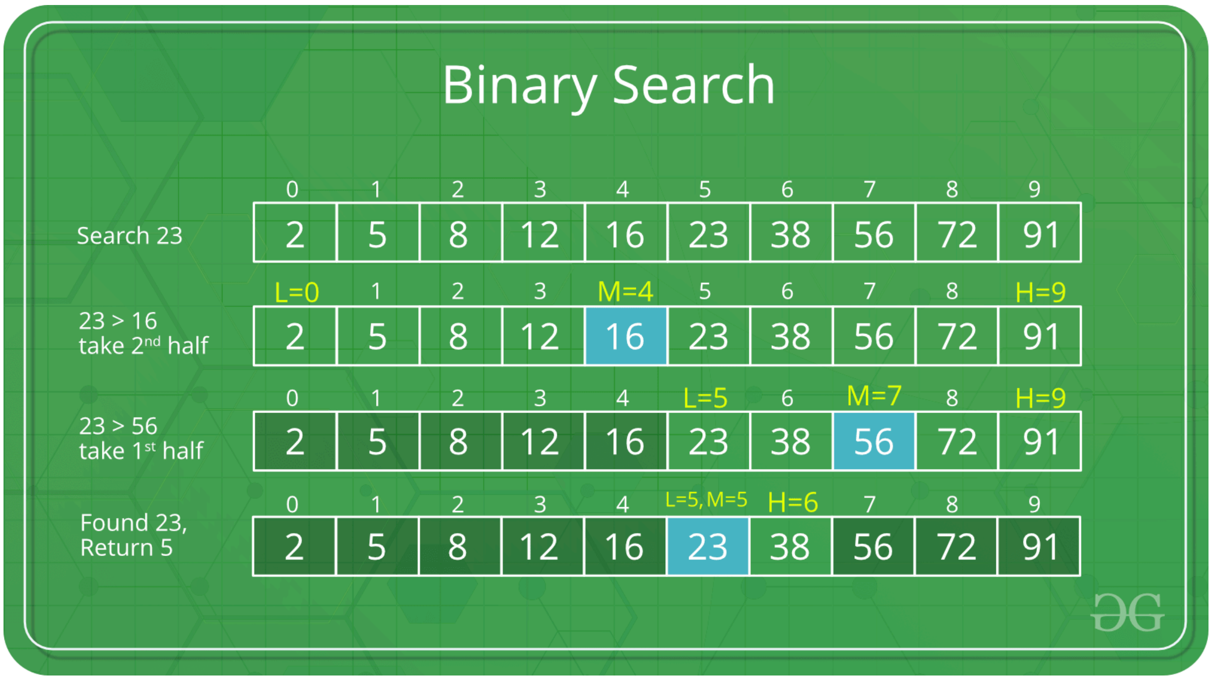
Merging Sublists:



11	22	33	40	44	55	60	77	88	90	99

Q17.Explain Binary Search with an example.

A17. Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].
A simple approach is to do a **linear search**. The time complexity of the above algorithm is O(n). Another approach to perform the same task is using Binary Search.
Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

Kindly note: Some of the answers may be quite long. Hence, I request all the students to shorten it and prepare according to the importance of the answers in examination.

Prepared By: Bhoomi C. Parikh