RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Analysis and Design of Algorithm**

Subject Code: **IT-403**

Semester: **4th**

Unit-1 Introduction of Algorithms:

Algorithm

The word "**Algorithm**" comes from the Persian author Abdullah Jafar Muhammad ibn Musa Al-khowarizmi in ninth century, who has given the definition of algorithm as follows:

- An Algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An Algorithm is a well defined computational procedure that takes input and produces output.
- An Algorithm is a finite sequence of instructions or steps (i.e. inputs) to achieve some particular output.
- An algorithm is finite set of instructions that is followed, accomplishes a particular task.

In addition, all algorithms must satisfy the following criteria:
1. Input. Zero or more quantities are externally supplied.

2. Output. At least one quantity is produced.

3. Definiteness. Each instruction is clear and produced.

4. Finiteness. If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

5. Effectiveness. Every instruction must be very basic so that it can be carried out, in principal, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.

Designing Algorithms

• Understanding the problem
• Ascertain the capabilities of the computational device
• Exact /approximate soln.
• Decide on the appropriate data structure
• Algorithm design techniques
• Methods of specifying an algorithm
• Proving an algorithms correctness
• Analysing an algorithm

- Understanding the problem:The problem given should be understood completely.Check if it is similar to some standard problems & if a Known algorithm exists.otherwise a new algorithm has to be devised.Creating an algorithm is an art which may never be fully automated. An important step in the design is to specify an instance of the problem.

- Ascertain the capabilities of the computational device: Once a problem is understood we need to Know the capabilities of the computing device this can be done by Knowing the type of the architecture,speed & memory availability.

- Exact /approximate soln.: Once algorithm is devised, it is necessary to show that it computes answer for all the possible legal inputs. The solution is stated in two forms,Exact solution or approximate solution.examples of problems where an exact solution cannot be obtained are i)Finding a squareroot of number.
  ii)Solutions of non linear equations.

- Decide on the appropriate data structure:Some algorithms do not demand any ingenuity in representing their inputs.Someothers are in fact are predicted on ingenious data structures.A data type is a well-defined collection of data with a well-defined set of operations on it.A data structure is an actual implementation of a particular abstract data type. The Elementary

- Algorithm design techniques: Creating an algorithm is an art which may never be fully automated. By mastering these design strategies, it will become easier for you to devise new anduseful algorithms. Divide & Conquer, Dynamic programming , Greedy strategy , such technique.

- Methods of specifying an algorithm: There are mainly two options for specifying an algorithm: use of natural language or pseudocode & Flowcharts.A Pseudo code is a mixture of natural language & programming language like constructs. A flowchart is a method of expressing an algorithm by a collection of connected geometric shapes.

- Proving an algorithms correctness: Once algorithm is devised, it is necessary to show that it computes answer for all the possible legal inputs .We refer to this process as algorithm validation. The process of validation is to assure us that this algorithm will work correctly independent of issues concerning programming language it will be written in.

- Analyzing algorithms: As an algorithm is executed, it uses the computers central processing unit to perform operation and its memory (both immediate and auxiliary) to hold the program and data. Analysis of algorithms and performance analysis refers to the task of determining how much computing time and storage an algorithm requires.

Analysis of algorithms:

There are any criteria upon which we can judge an algorithm for instance:
1. Does it do what we want to do?
2. Does it work correctly according to the original specifications to the task?
3. is there documentation that describes how to use it and how it works?
4. Are procedures created in such a way that they perform logical sub functions?
5. is the code readable?

The space complexity of an algorithm is the amount of memory it needs to run to completion.
The time complexity of an algorithm is the amount of computer time it needs to run to completion.

Algorithm Complexity

The complexity of an algorithm f(n) gives the running time and / or storage space required by the algorithm in terms of n as the size of input data.

Suppose X is an algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X.

- **Time Factor** − The time is measured by counting the number of key operations such as comparisons in sorting algorithm
- **Space Factor** − The space is measured by counting the maximum memory space required by the algorithm.

Space complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. Space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example simple variables & constant used, program size etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example dynamic memory allocation, recursion stack space etc.

Space complexity **S(P)** of any algorithm **P** is **S(P) = C + SP(I)** Where **C** is the fixed part and **S(I)** is the variable part of the algorithm which depends on instance characteristic **I**.

Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)

Step 1 - START

Step 2 - C ← A + B +10

Step 3 - Stop

Here we have three variables A, B and C and one constant. Hence S(P) = 1+3 =4. Now space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time complexity

Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function **T(n)**, where **T(n)** can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c*n, where c is the time taken for addition of two bits. Here, we observe that T(n) grows linearly as input size increases.The time T (P) taken by a program P is sum of compile time and run time. The compile time does not depend on the instance characteristics. Also, we may assume that a compiled program will be run several time of a program. This run time is denoted by tp. Because of many of the factor tp depends on are not known at the time of a program is conceived, it is reasonable to attempt only to estimate tp . T(p)=c+tp

Worst-case, Best-case, Average case efficiencies

Algorithm efficiency depends on the input size n. And for some algorithms efficiency depends on type of input. We have best, worst & average case efficiencies.

Worst-case efficiency: Efficiency (number of times the basic operation will be executed) for the worst case input of size n. i.e. The algorithm runs the longest among all possible inputs of size n.

Best-case efficiency: Efficiency (number of times the basic operation will be executed) for the best case input of size n. i.e. The algorithm runs the fastest among all possible inputs of size n.

Average-case efficiency: Average time taken (number of times the basic operation will be executed) to solve all the possible instances (random) of the input. NOTE: NOT the average of worst and best case

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation, as mentioned below −

- ***A priori*** **analysis** − This is theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. processor speed, are constant and have no effect on implementation.
- ***A posterior*** **analysis** − This is empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We learn **a priori** algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. Running time of an operation can be defined as no. of computer instructions executed per operation.

Asymptotic Notations

Asymptotic analysis of an algorithm, refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case and worst case scenario of an algorithm.

Asymptotic analysis are input bound i.e., if there's no input to the algorithm it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. Which means first operation running time will increase linearly with the increase in n and running time of second operation will increase exponentially when n increases. Similarly the running time of both operations will be nearly same if n is significantly small.

Usually, time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

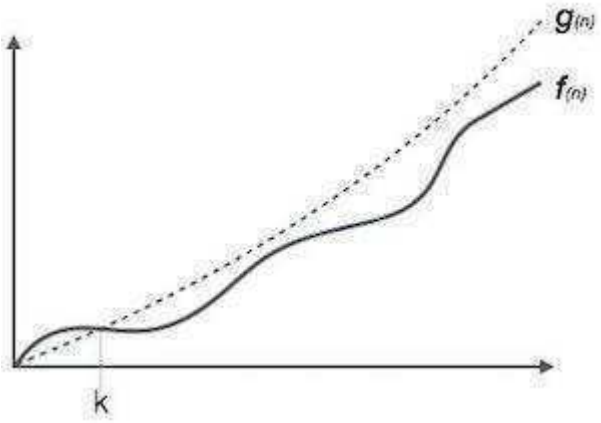
Following are commonly used asymptotic notations used in calculating running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

The O(n) is the formal way to express the upper bound of an algorithm's running time. It measures

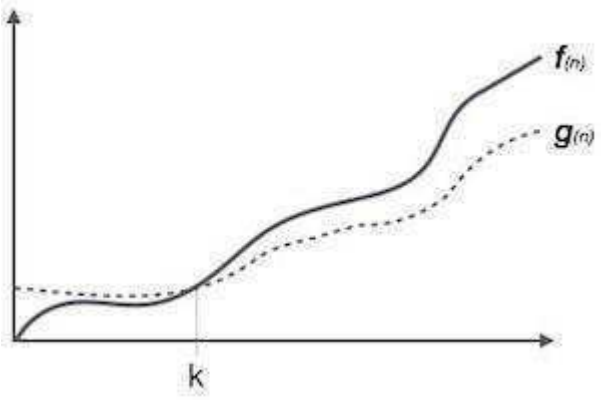
the worst case time complexity or longest amount of time an algorithm can possibly take to complete.
Definition:
A function t(n) is said to be in O(g(n)), denoted t(n) ∈ O(g(n)), if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that
t(n) ≤ cg(n) for all n ≥ n0



For example, for a function *f*(n)
O(*f*(n)) = { *g*(n) : there exists c > 0 and n0 such that *g*(n) ≤ c.*f*(n) for all n > n0. }

Omega Notation, Ω

The Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or best amount of time an algorithm can possibly take to complete.
Definition:
A function t (n) is said to be in Ω (g(n)), denoted t(n) ∈ Ω (g (n)), if t (n) is bounded below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that
t(n) ≥ cg(n) for all n ≥ n0

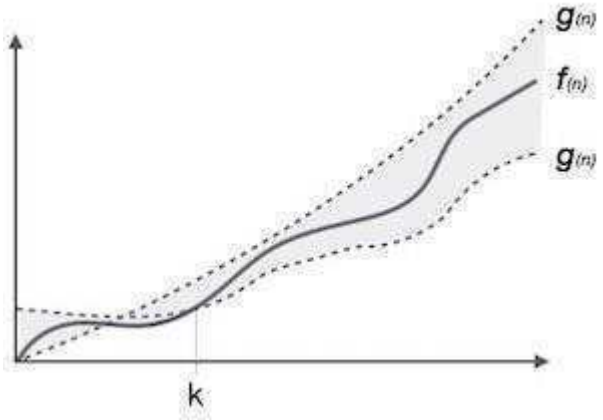


For example, for a function *f*(n)
Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n0 such that *g*(n) ≤ c.*f*(n) for all n > n0. }

Theta Notation, θ

The θ(n) is the formal way to express both the lower bound and upper bound of an algorithm's running time. Definition:

A function t (n) is said to be in Θ(g (n)), denoted t(n) ∈ Θ (g (n)), if t (n) is bounded both above and below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that
c2 g (n) ≤ t (n) ≤ c1 g (n) for all n ≥ n0
It is represented as following –



θ(*f*(n)) = { *g*(n) if and only if *g*(n) = O(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n0. }

Basic Efficiency classes

The time efficiencies of a large number of algorithms fall into only a few classes.

	Efficiency	Class	
Fast	1	constant	High time efficiency
↓	log n	logarithmic	↓
	n	linear	
	n log n	n log n	
	n 2	quadratic	
	n 3	cubic	
	2 n	exponential	
slow	n!	Factorial	low time efficiency

constant	–	O(1)
logarithmic	–	O(log n)
linear	–	O(n)
n log n	–	O(n log n)
quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$
polynomial	–	${}_nO(1)$
exponential	–	${}_2O(n)$

Recurrence relation:

Recurrence relations often arise in calculating the time and space complexity of algorithms. Any

problem can be solved either by writing recursive algorithm or by writing non-recursive algorithm. A recursive algorithm is one which makes a recursive call to itself with smaller inputs. We often use a recurrence relation to describe the running time of a recursive algorithm. A recurrence relation is an equation or inequality that describes a function in terms of its value on smaller inputs or as a function of preceding (or lower) terms.

Like all recursive functions, a recurrence also consists of two steps:
1. Basic step: Here we have one or more constant values which are used to terminate recurrence. It is also known as initial conditions or base conditions.
2. Recursive steps: This step is used to find new terms from the existing (preceding) terms. Thus in this step the recurrence compute next sequence from the k preceding values. This formula is called a recurrence relation (or recursive formula).
Hence a recurrence has one or more initial conditions and a recursive formula, known as recurrence relation.
Recurrence relations are used to determine the running time of recursive programs – recurrence relations themselves are recursive.
T (0) =time to solve problem of size 0 – Base Case
T (n) =time to solve problem of size n – Recursive Case
There are mainly three ways for solving recurrences.

1) Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the the guess is correct or incorrect.

For example consider the recurrence T(n) = 2T(n/2) + n

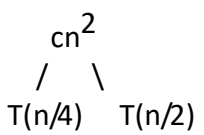
We guess the solution as T(n) = O(nLogn). Now we use induction to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.

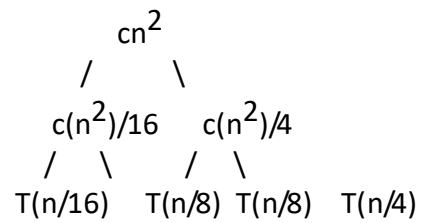
T(n) = 2T(n/2) + n
<= cn/2Log(n/2) + n
= cnLogn - cnLog2 + n
= cnLogn - cn + n
<= cnLogn

2) Recurrence Tree Method: In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

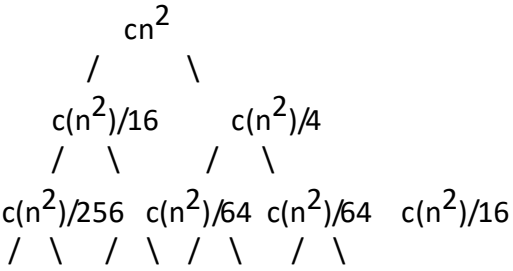
For example consider the recurrence relation
$T(n) = T(n/4) + T(n/2) + cn^2$



If we further break down the expression T(n/4) and T(n/2),
we get following recursion tree.



Breaking down further gives us following



To know the value of T(n), we need to calculate sum of tree nodes level by level. If we sum the above tree level by level, we get the following series
T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...
The above series is geometrical progression with ratio 5/16.

To get an upper bound, we can sum the infinite series.
We get the sum as $(n^2)/(1 - 5/16)$ which is $O(n^2)$

3) Master Method:

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:

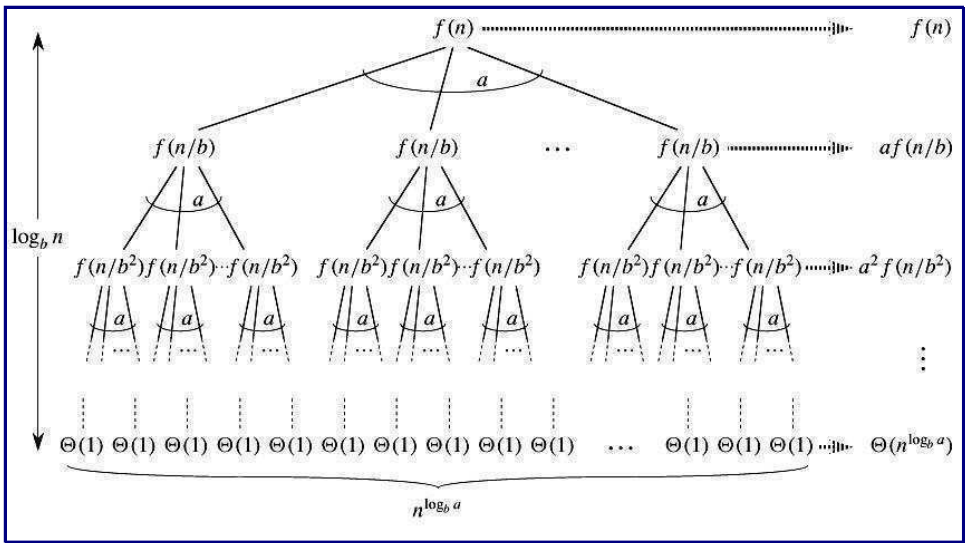
1. If $f(n) = \Theta(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$

2. If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$

3.If $f(n) = \Theta(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

How does this work?

Master method is mainly derived from recurrence tree method. If we draw recurrence tree of T(n) = aT(n/b) + f(n), we can see that the work done at root is f(n) and work done at all leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$. And the height of recurrence tree is $\text{Log}_b n$



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case 3).

Examples of some standard algorithms whose time complexity can be evaluated using Master Method

Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$. It falls in case 2 as c is 1 and $Log_b a]$ is also 1. So the solution is $\Theta(n Logn)$

Binary Search: $T(n) = T(n/2) + \Theta(1)$. It also falls in case 2 as c is 0 and $Log_b a$ is also 0. So the solution is $\Theta(Logn)$

Notes:

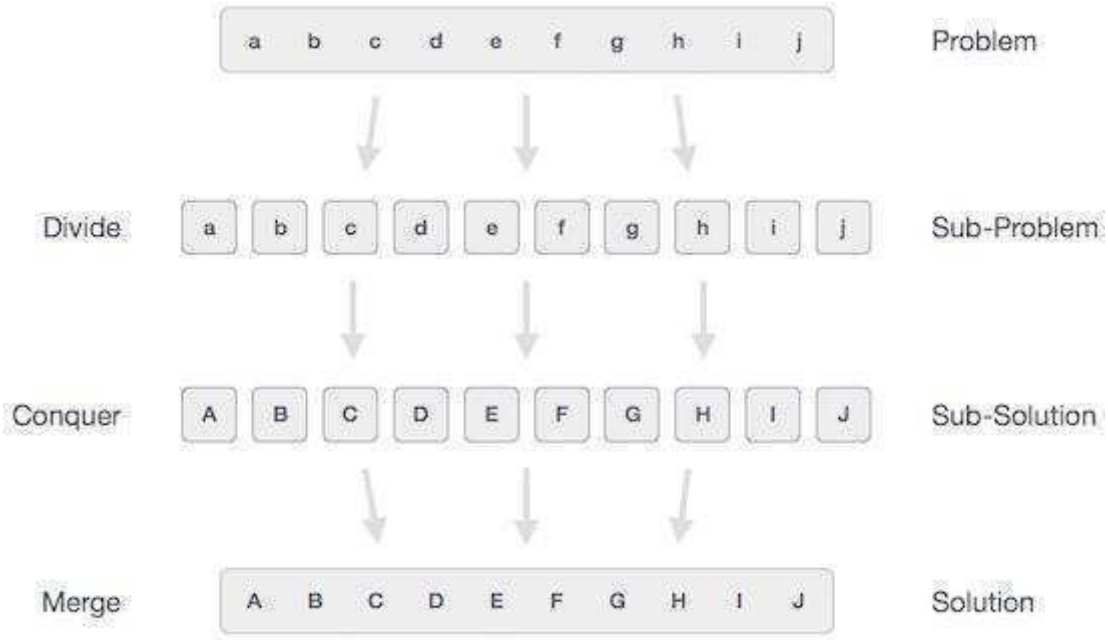
1) It is not necessary that a recurrence of the form T(n) = aT(n/b) + f(n) can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence T(n) = 2T(n/2) + n/Logn cannot be solved using master method.

2) Case 2 can be extended for $f(n) = \Theta(n^c Log^k n)$

If $f(n) = \Theta(n^c Log^k n)$ for some constant k >= 0 and $c = Log_b a$, then $T(n) = \Theta(n^c Log^{k+1} n)$

Divide & Conquer Method

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the sub-problems into even smaller sub-problems, we may eventually reach at a stage where no more dividation is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of original problem.



Broadly, we can understand **divide-and-conquer** approach as three step process.

Divide/Break

- This step involves breaking the problem into smaller sub-problems. Sub-problems should represent as a part of original problem. This step generally takes recursive approach to divide the problem until no sub-problem is further dividable. At this stage, sub-problems become atomic in nature but still represents some part of actual problem.

Conquer/Solve

- This step receives lot of smaller sub-problem to be solved. Generally at this level, problems are considered 'solved' on their own.

Merge/Combine

- When the smaller sub-problems are solved, this stage recursively combines them until they formulate solution of the original problem.

This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Some Examples

The following computer algorithms are based on **divide-and-conquer** programming approach –

- Binary Search
- Merge Sort
- Quick Sort
- Strassen's Matrix Multiplication

Binary search

Binary Search is applied on the sorted array or list. In binary search, we first compare the key value with the element in the middle position of the array. If the value is matched, then we return the position. If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array. We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large

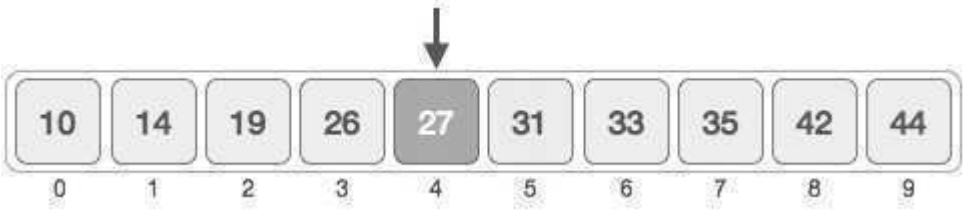
numbers of elements in an array.

The below given is our sorted array and assume that we need to search location of key value 31 using binary search.

First, we shall determine the mid of the array by using this formula –

mid = (LB + UB) / 2

Here it is, 0 + 9 / 2 = 4 (integer value of 4.5). So 4 is the mid of array.



Now we compare the value stored at location 4, with the value being searched i.e. 31. We find that value at location 4 is 27, which is not a match. Because key value is greater than 27 and we have a sorted array so target value must be in upper portion of the array.

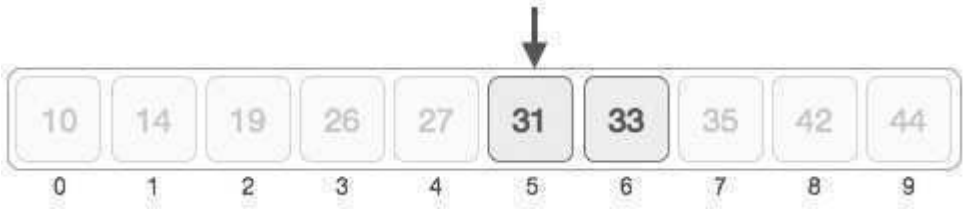
We change our low to mid + 1 and find the new mid value again.

low = mid + 1
mid = (LB + UB) / 2 = 5+9/2 =7

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is greater that what we are looking for. So the value must be in lower part from this location.

So we calculate the mid again. This time it is 5.



We compare the value stored ad location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Algo Binary Search ( ) :

Description: Here A is a sorted array having N elements. ITEM is the value to be searched. BEG denotes first element and END denotes last element in the array. MID denotes the middle value.

1. Set BEG = 1 and END = N

2. Set MID = (BEG + END) / 2

3. Repeat While (BEG <= END) and (A[MID] ≠ ITEM)

4.If (ITEM < A[MID]) Then

5.Set END = MID – 1

6.Else

7.Set BEG = MID + 1

8.Set MID = (BEG + END) / 2

9.If (A[MID] == ITEM) Then

10.Print: ITEM exists at location MID

11.Else

12.Print: ITEM doesn't exist

13.Exit

Merge sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n).Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How merge sort works

To understand merge sort, we take an unsorted array as depicted below –

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly same manner they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order 19 and 35. 42 and 44 are placed sequentially.

In next iteration of combining phase, we compare lists of two data values, and merge them into a list of foud data values placing all in sorted order.

After final merging, the list should look like this –

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then merge sort combines smaller sorted

lists keeping the new list sorted too.
Step 1 – if it is only one element in the list it is already sorted, return.
Step 2 – divide the list recursively into two halves until it can no more be divided.
Step 3 – merge the smaller lists into new list in sorted order.

Merge Sort (A, BEG, END) :
Description: Here A is an unsorted array. BEG is the lower bound and END is the upper bound.

1. If (BEG < END) Then
2. Set MID = (BEG + END) / 2
3. Call Merge Sort (A, BEG, MID)
4. Call Merge Sort (A, MID + 1, END)
5. Call Merge Array (A, BEG, MID, END)
6. [End of If]
7. Exit

Merge Array (A, BEG, MID, END)
Description: Here A is an unsorted array. BEG is the lower bound, END is the upper bound and MID is the middle value of array. B is an empty array.

1. Repeat For I = BEG to END
2. Set B[I] = A[I]
3. [End of For Loop]
4. Set I = BEG, J = MID + 1, K = BEG
5. Repeat While (I <= MID) and (J <=
6. If (B[I] <= B[J]) Then
7. Set A[K] = B[I]
8. Set I = I + 1 and K = K
9. Else
10. Set A[K] = B[J]
11. Set J = J + 1 and K = K
12. [End of If]
13. [End of While Loop]
14. If (I <= MID) Then
15. Repeat While (I <= MID)
16. Set A[K] = B[I]
17. Set I = I + 1 and K = K
18. [End of While Loop]
19. Else
20. Repeat While (J <= END)
21. Set A[K] = B[J]
22. Set J = J + 1 and K = K
23. [End of While Loop]
24. [End of If]
25. Exit

Quick sort
Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than

specified value say pivot based on which the partition is made and another array holds values greater than pivot value. The quick sort partitions an array and then calls itself recursively twice to sort the resulting two subarray. This algorithm is quite efficient for large sized data sets as its average and worst case complexity are of O(nlogn) where n are no. of items.
Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

The steps are:
1. Pick an element, called a pivot, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, whileall elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

QuickSort Algorithm
Using pivot algorithm recursively we end-up with smaller possible partitions. Each partition then processed for quick sort. We define recursive algorithm for quicksort as below –
Step 1 – Make the right-most index value pivot
Step 2 – partition the array using pivot value
Step 3 – quicksort left partition recursively
Step 4 – quicksort right partition recursively

QuickSort Partition Algorithm
Based on our understanding of partitioning in quicksort, we should now try to write an algorithm for it here.
Step 1 – Choose the highest index value has pivot
Step 2 – Take two variables to point left and right of the list excluding pivot
Step 3 – left points to the low index
Step 4 – right points to the high
Step 5 – while value at left is less than pivot move right
Step 6 – while value at right is greater than pivot move left
Step 7 – if both step 5 and step 6 does not match swap left and right
Step 8 – if left ≥ right, the point where they met is new pivot

```
void Quicksort(int left, int right)
{
if (left >= right) return(0);
int pivot = Partition(left, right + 1);
Quicksort(left, pivot – 1);
Quicksort(pivot + 1,right);
return(0);
}

int Partititon(int left, int right)
{
int pivot_element = A[left];
int left_to_right = ++left;
int right_to_left = --right;
```



```
while(A[left_to_right] < pivot_element) ++left_to_right;
while(A[right_to_left] > pivot_element) ++right_to_left;
//posititon for pivot element is at right_to_leftA[left] = A[right_to_left], A[right_to_left] = pivot_element;
return (right_to_left);
}
```

Strassen's Matrix Multiplication

Given two square matrices A and B of size n x n each, find their multiplication matrix.
Following is a simple way to multiply two matrices.

```
void multiply(int A[][N], int B[][N], int C[][N])
{
   for (int i = 0; i < N; i++)
   {
      for (int j = 0; j < N; j++)
      {
         C[i][j] = 0;
         for (int k = 0; k < N; k++)
         {
            C[i][j] += A[i][k]*B[k][j];
         }
      }
   }
}
```

Divide and Conquer
Following is simple Divide and Conquer method to multiply two square matrices.
1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

2) Calculate following values recursively . ae + bg, af + bh, ce + dg and cf + dh.

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. So the time complexity can be written as $T(N) = 8T(N/2) + O(N^2)$.Time complexity of above method is $O(N^3)$.

Strassen'smethod:

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

$$p1 = a(f - h) \qquad p2 = (a + b)h$$
$$p3 = (c + d)e \qquad p4 = d(g - e)$$
$$p5 = (a + d)(e + h) \qquad p6 = (b - d)(g + h)$$
$$p7 = (a - c)(e + f)$$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Time Complexity of Strassen's Method

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as $T(N) = 7T(N/2) + O(N^2)$

From Master's Theorem, time complexity of above method is $O(N^{Log7})$ which is approximately $O(N^{2.8074})$

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in