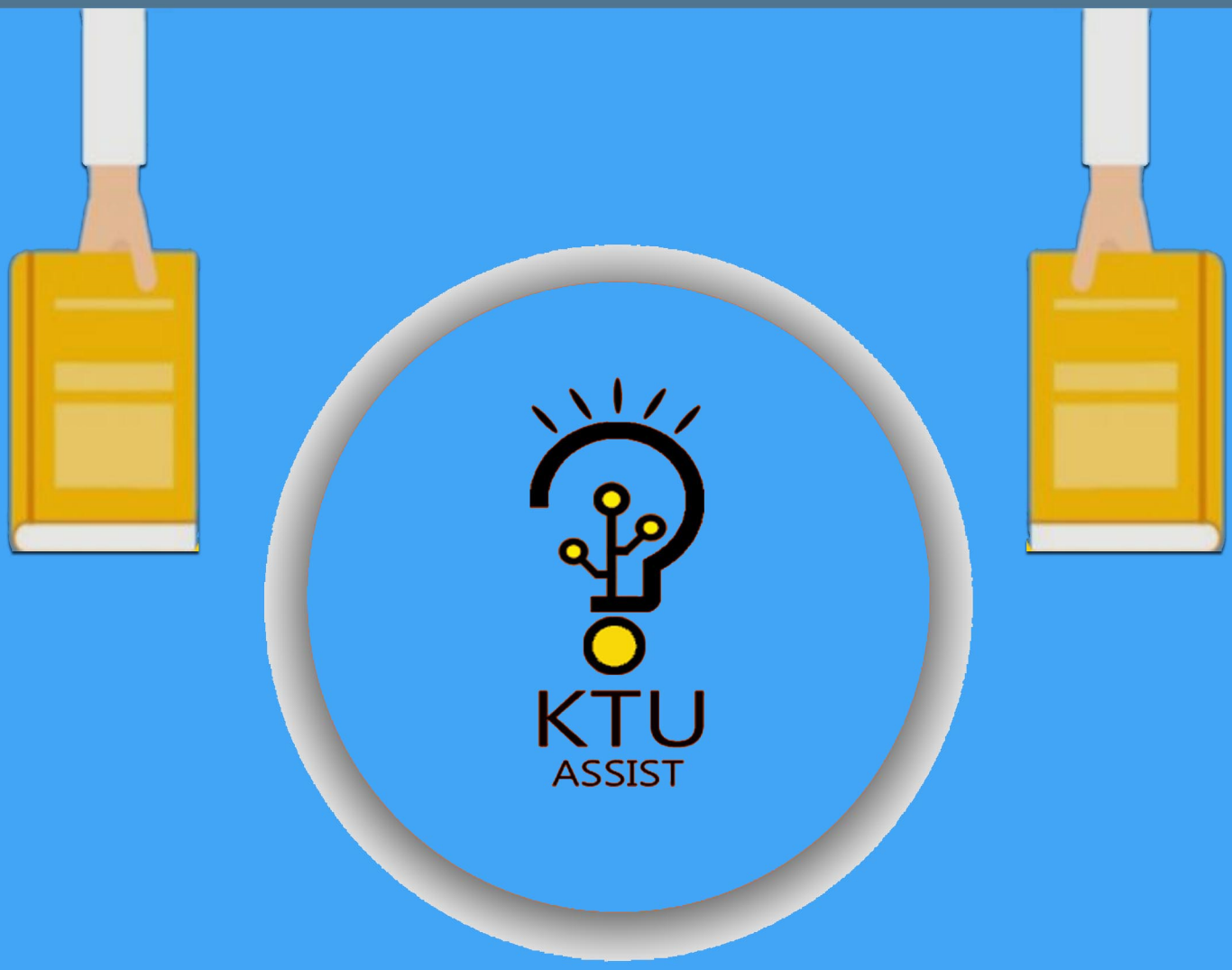
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS



Module 1

Algorithm Analysis & Recurrence Equations

ALGORITHM ANALYSIS

Ques 1) What is algorithm? What are the main features of algorithm?

Ans: Algorithm

An algorithm is defined as a finite set of well defined instructions for problem solving. It takes a valid input and produces desired output in a finite amount of time. It is illustrated in **figure 1.1**.

An algorithm is generally used for calculation and processing data using a sequence of finite instruction set.

An Algorithm provides step-by-step procedure for problem solving.

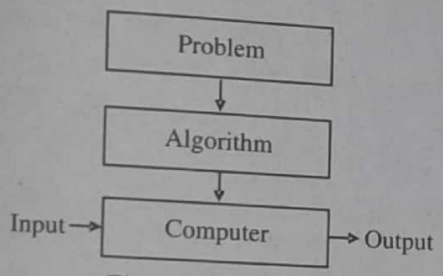


Figure 1.1: Algorithm

According to Stone and Knuth "an algorithm is a set of rules that precisely defines a sequence of operations in such a way that each rule is effective and definite and that the sequence terminates in a finite time."

An algorithm has a specific format which is shown in **figure 1.2**.

ALGORITHM: Algorithm_Name

Input: .....

Step 1: .....
Step 2: .....
Step 3: .....
.
.
.
Step n: .....

Output: .....

Figure 1.2: Format of Algorithm

An algorithm is a sequence of operational steps. It transforms input into output. **For example,** let suppose one has a problem of finding the greatest number from a list of given numbers. To find the greatest number, he/she will write the algorithm as follows:

Algorithm: Largest_Number
Step 1) Largest $\leftarrow L_0$
Step 2) For every item in the list $L \geq 1$, do
Step 3) If the item > largest, then
Step 4) Largest $\leftarrow$ item
Step 5) Return largest.

Characteristics / Properties of Algorithm

Salient characteristics of an algorithm are as follows:

1) **Finiteness:** After finite number of steps it terminates.
2) **Definiteness:** Specified rigorously and unambiguously.
3) **Input:** It clearly specified the valid input.
4) **Output:** For a valid input it can be proved to give the accurate output.
5) **Effectiveness:** Simple and basic steps are used.

Ques 2) What are the advantages and disadvantages of algorithm?

Ans: Advantages of Algorithms

Following are the advantages of Algorithm:

1) It cases the process of actual development of program code.
2) It allows the programmers to use the most efficient solution as per time and space complexity.
3) It breaks down the solution of a problem into a series of simplified sequential steps.
4) It is a simplified way of representing program instructions enables other programmers to easily understand and modify it.

Disadvantages of Algorithms

The disadvantages of Algorithm are as follows:

1) For large algorithms, it becomes difficult to understand the flow of program control.
2) It lacks the visual representation of programming logic as is prevalent in flowcharts.
3) There are no standard conventions to be followed while developing algorithms.
4) It may take considerable amount of time to write the algorithm for a given problem.

Ques 3) What is meant by analysis of algorithm?

Or

What is space and time complexity?

Ans: Analysis of Algorithm

After designing an algorithm, it has to be checked and its correctness needs to be predicted; this is done by analysing the algorithm. The algorithm can be analysed by tracing all step-by-step instructions, reading the algorithm for logical correctness, and testing it on some data using mathematical techniques to prove it correct.

Another type of analysis is to analyse the simplicity of the algorithm. That is, design the algorithm in a simple way so that it becomes easier to be implemented. However, the simplest and most straightforward way of solving a problem may not be sometimes the best one. Moreover there may be more than one algorithm to solve a problem. The choice of a particular algorithm depends on following performance analysis and measurements:

1) **Space Complexity:** The space complexity of an algorithm measures the amount of memory space required for the loading of a program.

   The memory space requirement depends on following elements:

   i) **Instructions:** Space required to store executable instructions of the program implementing the algorithm.

   ii) **Data:** Space required to store all the variables, constants, pointers etc.

   iii) **Environment Stack:** Space required for recursive program.

   The space that a program uses is the sum of the following elements:

   i) Fixed sized components of the program which require space for storing code, fundamental variable, and others in memory.

   ii) Variable or dynamic components of the program which require space for storing composite or user defined variables whose size depend on the instance of specific problem. It also includes the stack space requirement in case of recursive procedures.

2) **Time Complexity:** The time complexity of an algorithm measures how much time an algorithm requires to run the program completely. It can be considered as the time t (n) required by the program having n as input size.

   It represents the total time (compilation time + running time) required by the program for its execution. The Big O notation is most commonly used to measure time complexity. Time complexity counts the number of instruction steps performed by the algorithm. If time complexity is lower, it means the algorithm is faster.

   For different cases of input data, the performance of algorithm can vary. For some cases of input, the performance of algorithm may be best and for others it can be worst. One should take into consideration the worst-case time complexity as it takes maximum time for input size.

Ques 4) Discuss about the different elementary operations with their growth function.

Ans: Elementary operations with their growth function are shown in **table below**:

Growth Function	Elementary Operations
O(1)	Time requirement is constant, and it is independent of the problem's size
O(n)	Time requirement for a linear algorithm increases directly with the size of the problem.
$O(n^2)$	Time requirement for a quadratic algorithm increases rapidly with the size of the problem.
$O(n^3)$	Time requirement for a cubic algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm.
$O(2^n)$	As the size of the problem increases, the time requirement for an exponential algorithm increases too rapidly to be practical.
n!	Factorial
$O(\log_2 n)$	Time requirement for a logarithmic algorithm increases slowly as the problem size increases.
$O(n*\log_2 n)$	Time requirement for an $n*\log_2 n$ algorithm increases more rapidly than a linear algorithm.

Ques 5) What is the frequency count? Explain with example.

Ans: Frequency Count

Frequency count means how many times certain instruction executed in a piece of code. The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed. These numbers of primitive operations are called frequency count.

Example 1: Following is a program fragment that will be used for a frequency count analysis:

```
cout << "Enter your name:";
string name;
cin >> name;
```

The frequency analysis of this fragment is as follows:

```
cout << "Enter your name:"; // add 1 to the time count
string name;              // add 1 to the space count
cin >> name;              // add 1 to the time count
```

Notice that the data declaration code is not counted in the time count. This example has a frequency count of 2 for time and a frequency count of 1 for space.

Example 2: Let consider the following to see what happen when the number of repetitions in a loop or number of elements in a data structure is unknown. In this example, as the number of elements in use in the array grows, the number of repetitions required to sum all of the populated elements increases in a 1-to-1 relationship.

Typically, the letter n is used to represent such growth. Here is a fragment that repeats an unknown number of times:

```
int sum=0;
for(int i=0; i<n; i++) {
    sum += a[i];
}
cout << sum;
```

In this example, a is an integer array with elements that have been given values in such a way that the number of elements that are to be used (n) is not knowable in advance (when the program is written). Here is the analysis of this algorithm.

```
creation of array a // add n to the space count
int sum=0; // add 1 to the time count, add 1 to the space count
for(int i=0; i<n; i++) {     // add n+1 to the time count, add
                                1 to the space count
    sum += a[i];             // add n to the time count
}
cout << sum;                 // add 1 to the time count
```

This is the summary of the time analysis for the example:

```
    1
+ n + 1
    + n
    + 1
-------
  2n + 3
```

Therefore, the algorithm and data structures have the frequency counts of n+1 for space and 2n+3 for time.

Example 3: Algorithms involving nested loops are very common. The following is a fragment that uses a nested for-loop:

```
int b;
for (int x=0; x<5; x++)
    for (int y=0; y<4; y++) {
        b = x * y;
        cout << b;
}
```

Since there are three simple variables, space use is 3. The only complication in analysing this fragment for time is the nested for-y-loop, which will be performed in its entirety each time the for-x-loop makes one pass. The number of times that the for-y-loop will be started from its beginning conditions (the number of passes through the body of the for-x loop) will have to be used to multiply the time count of the statements in the for-y-loop. The frequency count analysis of this fragment is as follows:

```
int b;                      // add 1 to space
for (int x=0; x<5; x++)     // add 1 to space, 6 to time
for (int y=0; y<4; y++) {   // add 1 to space, 5 times 5 to time
        b = x * y;          // add 4 times 5 to time
        cout << b;          // add 4 times 5 to time
}
```

This gives:

Time	Space
6	1
+25	+1
+20	+1
+20	3
71	

Ques 6) Discuss how to compute the time complexity of an algorithm. Explain with example.

Ans: Computation of Time Complexity

Generally, we determine the time complexity nothing but a running time for a piece of code but the code may contain different types of statements. So we find the time complexity of each statement then counting the number of statements performed by a code.

The time complexity of different type of statements is shown below:

1) **Sequence of Statements:** The syntax of sequence statement is:

```
statement 1;
statement 2;
....
statement k;
```

The total time is found by adding the times for all statements:

Total time = time (statement 1) + time (statement 2) + ... + time (statement k)

If each statement is simple because only involves basic operations, then the time for each statement is constant and the total time is also constant: **O(1).**

2) **"if-then-else" Statements:** The syntax of if statement is:

```
if (condition)
{
    sequence of statements 1
}
else
{
    sequence of statements 2
}
```

Here, either sequence 1 will execute, or sequence 2 will execute.

Therefore, the worst-case time is the slowest of the two possibilities: max(time(sequence 1) and time(sequence 2)).

For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be **O(N)**.

3) **'for' Loops:** The syntax of for loop statement is:

```
for (i = 0; i < N; i++)
{
    sequence of statements.
}
```

The 'for' loop executes N times so the sequence of statements also executes N times. Since we assume the statements are O(1), the total time for the for loop is N * O(1), which is **O(N)**.

4) **Nested Loops:** Consider loops where the number of iterations of the inner loop is independent of the value of the outer loop's index.

For example: The syntax of inner loop is:

```
for (i = 0; i < N; i++)
{
    for (j = 0; j < M; j++)
    {
        sequence of statements;
    }
}
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of $N \times M$ times. Thus, the complexity is $O(N \times M)$.

In a common special case where the stopping condition of the inner loop is $j < N$ instead of $j < M$. that is the inner loop also executes N times, Therefore, the total complexity for the two loops is $O(N^2)$.

Example: If the progression of the loop is loop is logarithmic such as the following:

```
Count=1
While(count<n)
{
    Count*=2;
    /* Some sequence of O(1) steps */
}
```

This loop running rime is O(log n). Note that we use a logarithm in an algorithm complexity, we almost always mean log base 2. This can be explicitly written as $O(\log_2 n)$. Since each time through the loop the value of count is multiplied by 2, The number of times the loop is executed is $O(\log_2 n)$.

Ques 7) Consider the following code and find the time complexity:

```
Public void example2 (int n)
{
    printSum(n);            /* This method call is O(1)
    for (int count = 0;count<n; count++)
                            /* this loop is O(n)
        printSum(count);
    for(int count=0;count<n; count++)
                            /* this loop is O(n^2)
        for(int count2 = 0;count2 < n; count2++)
            system.out.println(count, count2);
}
```

Ans: The initial call to the printSum method run at O(1) time.

The 'for' loop containing the call to the printSum method with parameter count is O(n). The nested loops running time is $O(n^2)$ .The growth function for the code is given by:

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

Now, eliminate the constants and all but the dominant term then the time complexity is $O(n^2)$.

Ques 8) What is best, worst and average case complexity? Explain with example.

Ans: Best, Worst, and Average-Case Complexity

1) **Worst-Case Complexity:** The worst-case complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n.

For example, the linear search on a list of n elements. In the worst case, the search must visit every element once. This happens when the value being searched for is either the last element in the list, or is not in the list.

2) **Best-Case Complexity:** The best-case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n.

For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list.

3) **Average-Case Complexity:** The average-case complexity of the algorithm is the function defined by an average number of steps taken on any instance of size n.

For example, the linear search on a list of n elements. The average case is assumed when the value searched for is in the list and each list element is equally likely to be the value searched for, the search visits only n/2 elements.

Ques 9) Discuss about the asymptotic analysis of binary search algorithm?

Ans: Binary Search

It is different from linear search as items are not searched in sequential manner. In binary search, the list is divided into two parts and then the item is searched. As two partitions are made, hence it is called binary search.

Binary search is faster than linear search and works only for sorted lists. Prior to application of binary search, linear data should be sorted either in ascending or descending order.

This approach is not suitable when elements are not in sorted form. It employs the divide-and-conquer approach. The element is first matched with the middle element. If the desired element is present before the middle element, the left portion is searched and vice-versa.

Algorithm: Binary Search

Input: L[1 . . . n] is a list
R = Number of elements
x = Item to be searched

Output: If x is present in list L, its index is returned else – 1 is returned.

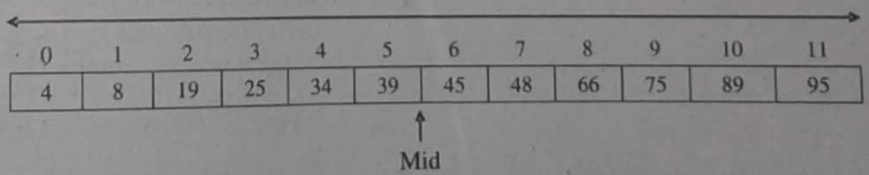
BINARY_SEARCH (L, n, x)

Step 1: Begin
Step 2: First: != 1;
Step 3: Last: = n;
Step 4: While First <= Last Do
Step 5: Begin
Step 6: Mid := $\left[\frac{\text{First} + \text{Last}}{2}\right]$;
Step 7: If List [Mid] = x Then
Step 8: Return Mid;
Step 9: Elseif List [Mid] > x Then
Step 10: Last: = Mid − 1;
Step 11: Else
Step 12: First: = Mid + 1;
Step 13: End If
Step 14: Return;
Step 15: End

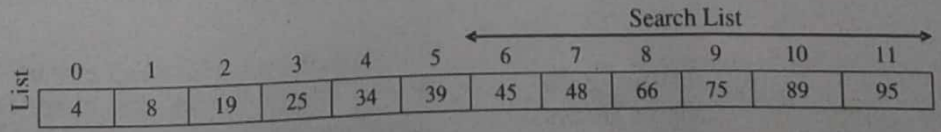
For example, let us take a sorted array list consisting of 12 elements.

	0	1	2	3	4	5	6	7	8	9	10	11
List	4	8	19	25	34	39	45	48	66	75	89	95

1) Let us consider that 75 is to be searched
2) Element 75 is compared with middle element of list, i.e., 39. Element 39 is present at index 5 (i.e., list[5]).
3) Since 75 is in list[6] to list[11], hence search is limited.



4) The process is now repeated for the right sub-list, i.e., list[6] to list[11]. Right sub-list consists of 6 elements.



Analysis of Binary Search

The block of items to be searched is divided into two halves at every step of the algorithm. Thus a set of n items can be divided in half maximum $\log_2 n$ times. The operational time of a binary search is directly proportional to log n, hence the complexity of binary search algorithm is O(log n).

For a small n, binary search will run slower than linear search as it requires a complex program than the original search. But, for large n:

$$\lim_{n\to\infty} \frac{\log n}{n} = 0$$

For large n, log n is smaller as compared to n. Hence, O(log n) algorithm is faster than O(n).

The number of possible searching is reduced by a factor of 2 when each comparison of binary search is performed. This shows that at most $\log_2 n$ key comparisons are performed. The complexity of binary search algorithm is O(log n).

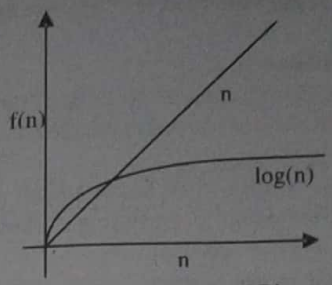


Figure 1.3: Analysis of Binary Search

Ques 10) Analyse complexity of the Euclid Algorithm For GCD

Ans: Euclid Algorithm for Greatest Common Divisor (GCD)

GCD of two non-negative, non-zero (both) integers i.e. m and n, is the largest integer that divides both m and n with a remainder of zero. Complexity analysis of an algorithm for computing GCD depends on which algorithm will be used for GCD computation.

The algorithm for calculating GCD will be explained in two steps:

Step I: Pseudo code for Computing GCD(m,n) by Euclid's Method

// m and n are two positive numbers where m is dividend and n is divisor

1) If n=0 , return m and exit else proceed to step 2).
2) Divide m by n and assign remainder to r.
3) Assign the value of n to m and value of r to n. Go back to step 1).

Step II: Algorithm for Computing GCD(m, n) by Euclid's Method

Input: Two non-negative, non-zero integers m and n
Output: GCD of m and n

```
function gcd(m,n)
{
    while (n 0)
    {
        r= m mod n
        m=n
        n=r
    }
    return m
}
```

Example: Find the GCD of 662 and 414
Let m= 662 and n=414

Divide m by n to obtain quotient and remainder.
662=414 · 1+248 ...(1) // here 1 is quotient and 248 is remainder

In subsequent iterations dividend and divisor are based on what number we get as a divisor and as a remainder respectively of previous iteration.

So, subsequent iterations are as follows:

414=248 · 1+166 ... (2) // now m is 414 and n is 166
248=166 · 1+82 ... (3) // now m is 248 and n is 82
166=82 · 2 + 2 ... (4) // now m is 166 and n is 2
82= 2 · 41 + 0 ... (5) // now m is 82 and n is 0

According to Euclid's algorithm,

Step 1) gcd(662,414)=gcd(414,248)
Step 2) gcd(414,248)=gcd(248,166)
Step 3) gcd(248,166)=gcd(166,82)
Step 4) gcd(166,82)=gcd(82,2)
Step 5) gcd (82,2)= gcd (2,0)

Combining above all gives gcd(662,414)=2 which is the last divisor that gives remainder 0.

Complexity Analysis

In function gcd(m,n), each iteration of while loop has one test condition, one division and two assignment that will take constant time. Hence number of times while loop will execute will determine the complexity of the algorithm. Here it can be observed that in subsequent steps, remainder in each step is smaller than its divisor i.e., smaller than previous divisor. The division process will definitely terminate after certain number of steps or until remainder becomes 0.

1) **Best Case:** If m=n then there will be only one iteration and it will take constant time i.e., O(1).

2) **Worst Case:** If n=1 then there will be m iterations and complexity will be O(m). If m=1 then there will be n iterations and complexity will be O(n)

3) **Average Case:** By Euclid's algorithm it is observed that:
GCD (m, n) = GCD (n, m mod n) = GCD (m mod n, n mod (m mod n))

Since m mod n = r such that m = nq + r, it follows that r < n, so m > 2r. So after every two iterations, the larger number is reduced by at least a factor of 2 so there are at most O (log n) iterations.

Complexity will be O(log n), where n is either the larger or the smaller number.

Ques 11) Calculate the runtime complexity of following program segment.

```
i = 1
loop (i < = n)
print i
i = i + 1
```

Ans: We will first compute the frequency count for given program segment:

i = 1	1
loop (i < = n)	n
print i	n
i = i + 1	n
Frequency count	3n + 1

Thus frequency count is 3n + 1. Neglecting the constants and by considering the order of magnitude, we get O(n) as runtime complexity.

Ques 12) An algorithm runs a given input of size n. if n is 4096, the runtime is 512 millisecond. If n is 16384, the runtime 1024 millisecond. What is the complexity? What is the big-O notation?

Ans: n = 4096,
Runtime = 512
Runtime * 8 = n

i.e., 512 * 8 = 4096

Similarly, n = 16384
Runtime = 1024
Runtime * 8 = n
Runtime* 8 = 16384

That means runtime increases linearly with input size n. Hence time complexity is linear ,i.e., O (n).

RECURRENCE EQUATIONS

Ques 13) What do you understand by recurrence equations?

Ans: Recurrence Equation
A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

Any equation involving several terms of a sequence is called a recurrence relation.

A recurrence relation is then an equation of the type
$F(n, a_n, a_{n+1}, \ldots, a_{n+k}) = 0$, where $k \in N$ is fixed.

Or

In other words, for a numeric function $(a_0, a_1, a_2, \ldots, a_r, \ldots)$ a_n equation relating a_r to one or more a_i $(i < r)$ is called a recurrence relation or difference equation.

The numeric function is referred to as the solution of the recurrence relation.

A recurrence relation is an equation, which is defined in terms of itself. Solving a recurrence relation means obtaining a closed-form solution: a non-recursive function of n.

For example, following are recurrence relations:
i) $a_r = a_{r-1} + a_{r-2}$, $a_0 = 0, a_1 = 1$
ii) $a_r = 3\, a_{r-1}$, $a_0 = 1$

Ques 14) Write the steps for analysing efficiency of recursive algorithm.

Ans: Steps for Analysing Efficiency of Recursive Algorithm
Step 1: Decide on parameter n indicating input size.
Step 2: Identify algorithm's basic operation.
Step 3: Determine worst, average, and best case for input of size *n*.
Step 4: Set up a recurrence relation and initial condition(s) for C(n)-the number of times the basic operation will be executed for an input of size n.
Step 5: Solve the recurrence to obtain a closed form or estimate the order of magnitude of the solution.

Ques 15) What is the order and degree of recurrence relation?

Ans: Order and Degree of Recurrence Relation
The order of a recurrence relation is the difference between the greatest and lowest subscripts of the terms of the sequence in the equation. In other words, a recurrence relation or order k is said to be linear if it is **linear** in a_n, $a_{n+1}, \ldots, a_{n+k}$. Otherwise, the recurrence relation is said to be **non-linear**.

The highest power of a_r or y_n or f(x) is called the degree of the recurrence relation or difference equation.

For example, recurrence relation $a_r = 3a_{r-2}$ is of degree 1 and called linear recurrence relation, whereas the recurrence relation
$y_4^n - 3y_{n-1}^3 + 2y_n = f(n)$ is of degree 4.

Ques 16) Consider the example and find the order of each given recurrence relation.
1) $a_n = 5a_{n-2} + 3a_{n-1}$
2) $a_n = 5a_{n-1} + n^2$
3) $a_n = 2a_{n-1}2 + 2^n$
4) $a_n = \sqrt{a_{n-1} + a_{n-2}^2}$

Ans:
1) $a_n = 5a_{n-2} + 3a_{n-1}$
The order is 2, degree is.
2) $\alpha_n = 5a_{n-1} + n^2$
The order is 1, degree is

3) $a_n = 2a_{n-1}2 + 2^n$
The order is 2, degree is

4) $a_n = \sqrt{a_{n-1} + a_{n-2}^{\ 2}}$
The order is 2, no degree

Ques 17) What are the different methods of solution of recurrence equations?

Or

Explain the iteration and recursion tree method with example.

Ans: Methods for Solving Recurrences

The recurrences relation can be solved by following methods:

1) **Substitution Methods:** In substitution method, the given recurrence relation for a_n is used repeatedly to solve a general expression for a_n in terms of n such that the expression does not contain any other term of the sequence except those given by initial conditions.

Example: $T(n) \le 2c\lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor) + n$. Show that $T(n) \le c \cdot n \lg n$ for large enough c and n. Assume that it is true for n/2, then

Solution: $T(n) \le 2 (c\lfloor n/2 \rfloor) \lg (\lfloor n/2 \rfloor) + n$

$$\le cn\lg \frac{n}{2} + n$$

$$= cn\lg n - cn\lg 2 + n = cn\lg n - cn + n$$

$$T(n) \le cn\lg n \ , c \ge 1$$

Starting with basis cases $T(2) = 4$, $T(3) = 5$, lets us complete the proof for $c \ge 2$.

2) **Iteration Method:** In iteration method the basic idea is to expand the recurrence and express it as a summation of terms dependent only on initial condition. Also known as **Try backsubstituting** until you know what is going on. Plug the recurrence back into itself until you see a pattern.

Example: $T(n) = 3T(\lfloor n/4 \rfloor) + n$.

Solution: Try backsubstituting:

$$T(n) = n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor))$$
$$= n + 3\lfloor n/4 \rfloor + 9(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor))$$
$$= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor)$$

The $(3/4)^n$ term should now be obvious.

Although there are only $\log_4 n$ terms before we get to T(1), it doesn't hurt to sum them all since this is a fast growing geometric series:

$$T(n) \le n \sum_{i=0}^{\infty} (\frac{3}{4})^i + \Theta\left(n^{\log_4 3} \times T(1)\right)$$

$$T(n) = 4n + o(n) = O(n)$$

3) **Recursion Trees:** Recursion tree method is a pictorial representation of an iteration method, which is in the form of a tree, where at each level nodes are expanded. It is useful when divide and conquer algorithm is used.

A recursion tree T(p) of degree p is either null or has p children which are recursion trees.

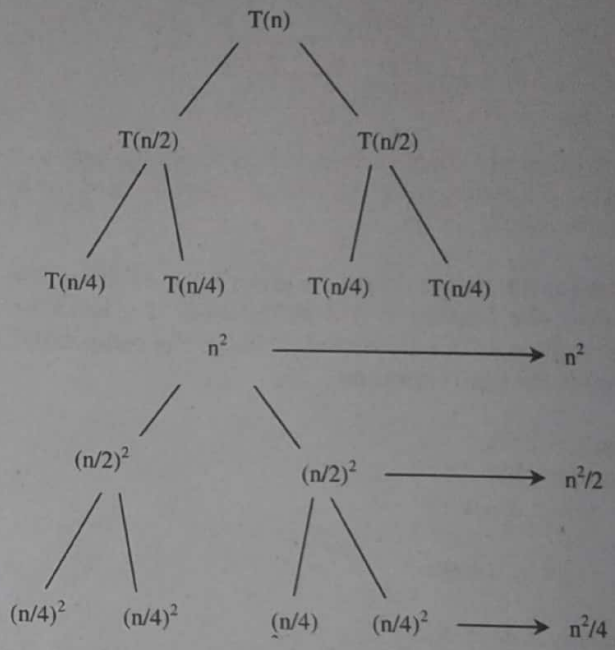
One must keep track of two things

i) The size of the remaining argument to the recurrence, and

ii) The additive stuff to be accumulated during this call.

Example: $T(n) = 2T(n/2) + n^2$

Solution:



The remaining arguments are on the left, the additive terms on the right.

Although this tree has height lg n, the total sum at each level decreases geometrically, so:

$$T(n) = \sum_{i=0}^{\infty} n^2/2^i = n^2 \sum_{i=0}^{\infty} 1/2^i = \Theta(n^2)$$

The recursion tree framework made this much easier to see than with algebraic back substitution.

4) **Change Variables:** In change variable method as a name implies we change any complex term presents in recurrence by any variable.

Step 1) First select a variable in place of complex term and put the variable as a substitute in the recurrence equation.

Step 2) Then for finding actual result replace the chosen variable by the complex term.

Example: Solve the following recurrence by using change variable method.

$$T(n) = 2T\left(\lfloor \sqrt{n} \rfloor\right) + 1$$

Solution: Given that:

$$T(n) = 2T\left(\lfloor \sqrt{n} \rfloor\right) + 1 \quad \ldots\ldots (1)$$

Let $m = \log_2 n$

$\Rightarrow n = 2^{mv}$

$\Rightarrow n^{\frac{1}{2}} = 2^{\frac{m}{2}}$ ..... (2)

put the value of $n^{\frac{1}{2}}$ in (1) we get,

$$T(2^m) = 2T\left(2^{\frac{m}{2}}\right) + 1 \quad \text{..... (3)}$$

put $x(m) = T(2^m)$ ..... (4)

putting the value of x(m) in equation (3) we have

$$x(m) = 2x\left(\frac{m}{2}\right) + 1 \quad \text{..... (5)}$$

The solution for equation (5) is given as
$x(m) = O(m \log m)$

Since, $T(n) = x(m)$
$\Rightarrow T(n) = O(m \log m)$
$\Rightarrow T(n) = O((\log n)(\log \log n))$

Ques 18) Show that solution to $T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor + 17\right) + n$ is O(n lg n).

Ans: Let us guess solution is O(n lg n) then we have to prove $T(n) \le cn \lg n$, substituting this we get,

$$T(n) \le 2\left(\frac{cn}{2}\lg\left(\frac{n}{2}\right) + 17\right) + n$$

$$= cn \lg\left(\frac{n}{2}\right) + 34 + n = cn \lg n - cn \lg 2 + 34 + n$$

$$= cn \lg n - cn + 34 + n = cn \lg n - (c-1)n + 34$$

$$= cn \lg n - bn + 34 \le cn \lg n$$

if $c \ge 1$, where b is constant. Hence, $T(n) = O(n \lg n)$

Ques 19) Solve $T(n) = 3Tn(\lfloor n/2 \rfloor) + n$ by iteration.

Ans: $T(n) = n + 3T(\lfloor n/2 \rfloor)$

$$= n + 3(\lfloor n/2 \rfloor) + 3.T(\lfloor n/4 \rfloor))$$

$$= n + 3(\lfloor n/2 \rfloor) + 3(\lfloor n/4 \rfloor) + 3T(\lfloor n/16 \rfloor)))$$

$$= n + \frac{3n}{2} + \frac{9n}{4} + \ldots + 3^i.T\left(n/2^i\right)$$

When series terminates

$$\frac{n}{2^i} = 1 \Rightarrow n = 2^i \text{ or } i = \log_2 n$$

Hence, $T(n) \le n + \frac{3n}{2} + \frac{9n}{4} + \ldots + 3^{\log_2 n}\Theta(1)$

$$\le n\sum_{i=0}^{\infty}\left(\frac{3}{2}\right)^i + n^{\log_2 3}.\Theta(1), \text{as } 3^{\log_2 n} = n^{\log_2 3}$$

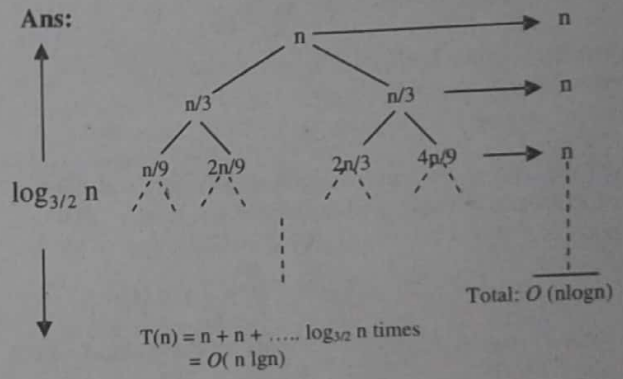
$$\le n.\frac{1}{1-\frac{3}{2}} + n^{\log_2 3}, \Theta(1) = 1$$

$$\le n^{\log_2 3} - 2n$$

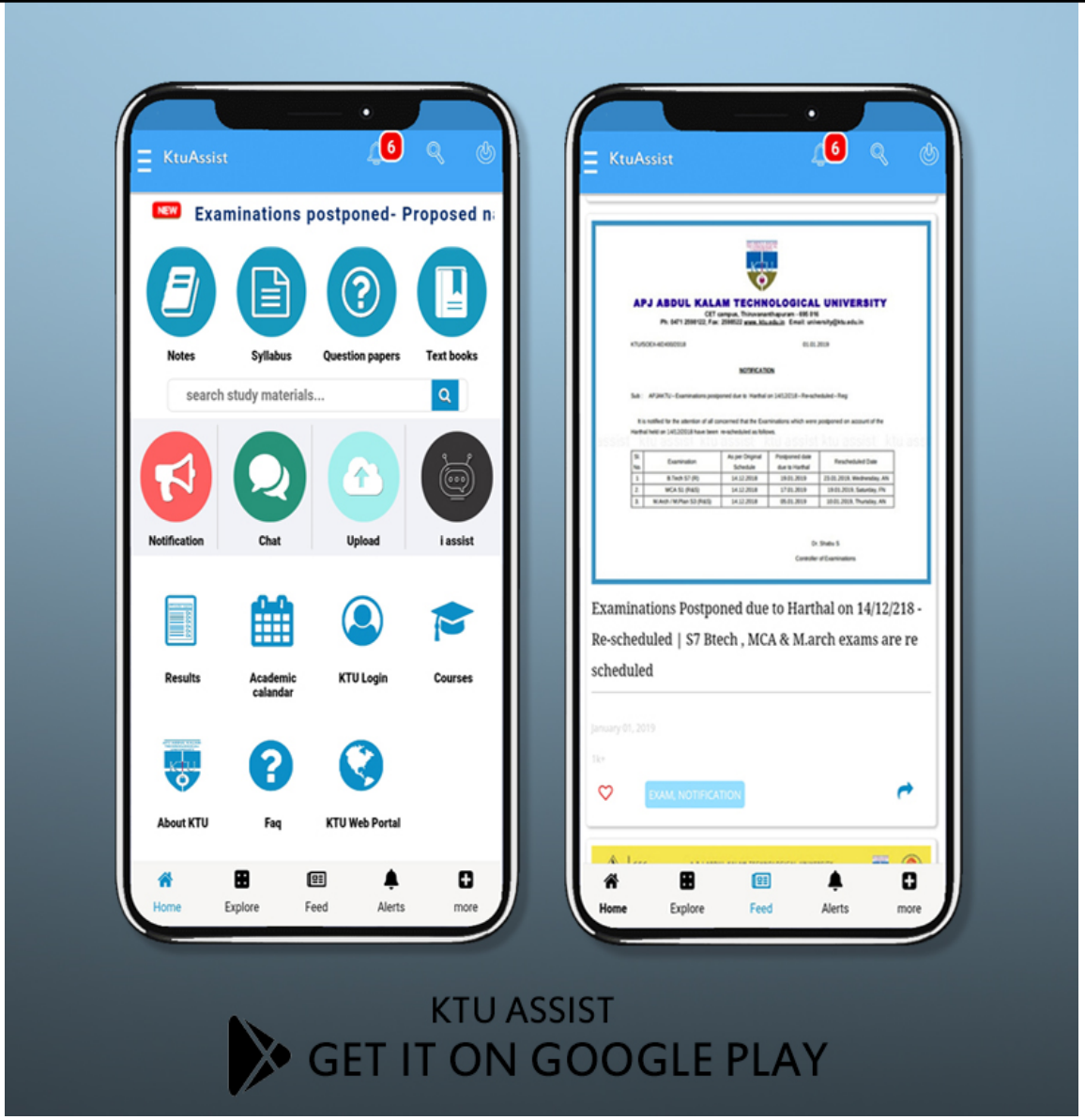
$$= \Theta(n^{\log 3}), \text{as} \log_2 3 > 1$$

Ques 20) Solve the recurrence $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$ by recursion trees.

Ans:



$T(n) = n + n + \ldots$ $\log_{3/2} n$ times
$= O(n \lg n)$



END

facebook.com/ktuassist

instagram.com/ktu_assist