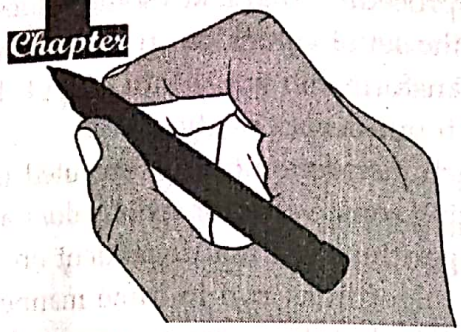
1 Chapter



FOUNDATION OF ALGORITHM ANALYSIS

CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Algorithm and its properties, RAM model, Time and Space Complexity, detailed analysis of algorithms (Like factorial algorithm), Concept of Aggregate Analysis
- Asymptotic Notations: Big-O, Big-Ω and Big-θ Notations their Geometrical Interpretation and Examples.
- Recurrences: Recursive Algorithms and Recurrence Relations, Solving Recurrences (Recursion Tree Method, Substitution Method, Application of Masters Theorem)

ALGORITHM

An algorithm can be defined as a well-defined computational procedure that takes some values, or the set of values, as an input and produces some value, or the set of values, as an output. An algorithm is thus a sequence of computational steps that transform the input into output. It describes specific computational procedures for achieving the input-output relationship.
An algorithm is a finite set of computational instructions, each instruction can be executed in finite time, to perform computation or problem solving by giving some value, or set of values as input to produce some value, or set of values as output. Algorithms are not dependent on a particular machine, programming language or compilers i.e. algorithms run in same manner everywhere. So the algorithm is a mathematical object where the algorithms are assumed to be run under machine with unlimited capacity.

ALGORITHMS PROPERTIES

- **Correctness:** It should produce the output according to the requirement of the algorithm
- **Finiteness:** Algorithm must complete after a finite number of instructions have been executed.
- **An Absence of Ambiguity:** Each step must be defined, having only one interpretation.
- **Definition of Sequence:** Each step must have a unique defined preceding and succeeding step. The first step and the last step must be noted.
- **Input/output:** Number and classification of needed inputs and results must be stated.
- **Feasibility:** It must be feasible to execute each instruction.
- **Flexibility:** It should also be possible to make changes in the algorithm without putting so much effort on it.
- **Efficient** - Efficiency is always measured in terms of time and space requires implementing the algorithm, so the algorithm uses a little running time and memory space as possible within the limits of acceptable development time.
- **Independent:** An algorithm should focus on what are inputs, outputs and how to derive output without knowing the language it is defined. Therefore, we can say that the algorithm is independent of language.

Need of Algorithm

- To understand the basic idea of the problem.
- To find an approach to solve the problem.
- To improve the efficiency of existing techniques.
- To understand the basic principles of designing the algorithms.
- To compare the performance of the algorithm with respect to other techniques.
- It is the best method of description without describing the implementation detail.
- The Algorithm gives a clear description of requirements and goal of the problem to the designer.

- A good design can produce a good solution.
- To understand the flow of the problem.
- To measure the behavior (or performance) of the methods in all cases (best cases, worst cases, average cases)
- With the help of an algorithm, we can also identify the resources (memory, input-output) cycles required by the algorithm.
- With the help of algorithm, we convert art into a science.
- To understand the principle of designing.
- We can measure and analyze the complexity (time and space) of the problems concerning input size without implementing and running it; it will reduce the cost of design.

RAM MODEL

Algorithms can be measured in a machine-independent way using the Random Access Machine (RAM) model. This model assumes a single processor. In the RAM model, instructions are executed one after the other, with no concurrent operations. This model of computation is an abstraction that allows us to compare algorithms on the basis of performance. The assumptions made in the RAM model to accomplish this are:

- Each simple operation takes 1 time step.
- Loops and subroutines are not simple operations.
- Each memory access takes one time step, and there is no shortage of memory.

For any given problem the running time of an algorithm is assumed to be the number of time steps. The space used by an algorithm is assumed to be the number of RAM memory cells. In computing time complexity, one good approach is to count primitive operations. This approach of simply counting primitive operations gives rise to a computational model called the Random Access Machine (RAM). The RAM model consists of following elements.

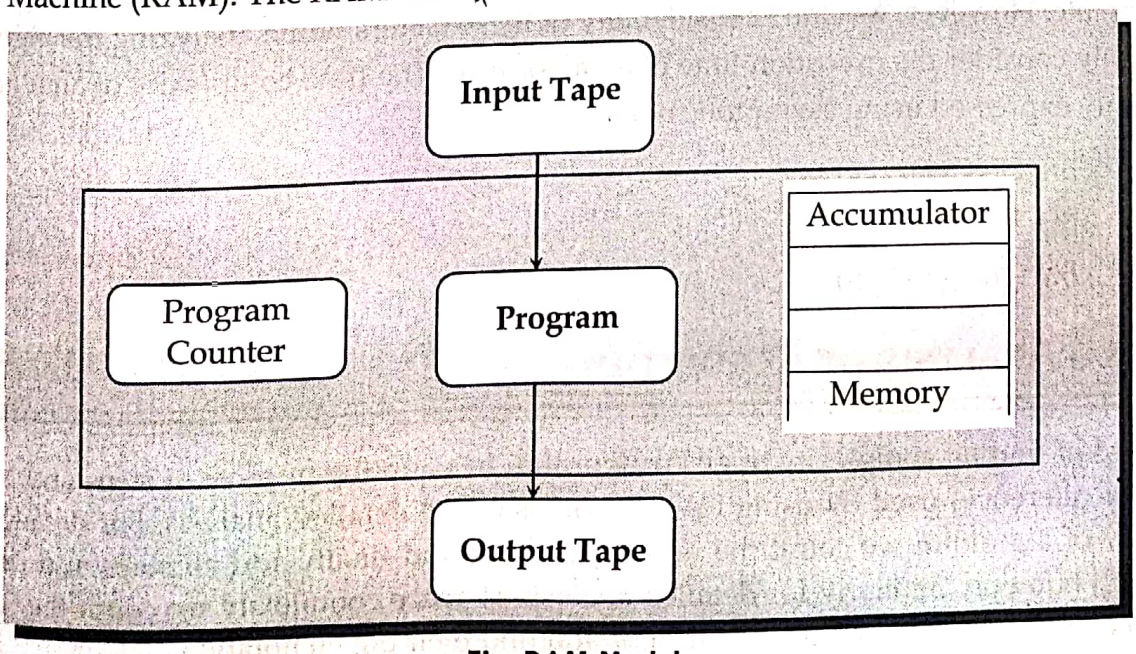


Fig: RAM Model

Input tape/output tape: input tape consists of a sequence of squares, each of which can sto integer. Whenever one square is read from the tape head moves one square to the right. Th output tape is also a sequence of squares, in each square an integer can be written. Output written on the square under the tape head and the after the writing, the tape head moves on square to the right over writing on the same square is not permitted.

Memory: The memory consists of a sequence of registers, each of which is capable of holding a integer.

Program: Program for RAM contains a sequence of labeled instructions resembling those foun in assembly language programs. All computations take place in the first register calle accumulator. A RAM program defines a mapping from input tape to the output tape.

Program counter: The program counter determines the next instruction to be executed.

Some examples of primitive operations are:

- Assigning a value to a variable
- Performing an arithmetic operation
- Indexing into an array
- Returning from method
- Calling a method
- Comparing two numbers etc

TIME AND SPACE COMPLEXITY

While analyzing an algorithm, we mostly consider time complexity and space complexity. Tim complexity of an algorithm quantifies the amount of time taken by an algorithm to run as function of the length of the input. Similarly, Space complexity of an algorithm quantifies th amount of space or memory taken by an algorithm to run as a function of the length of th input.

Time and space complexity depends on lots of things like hardware, operating system processors, etc. However, we don't consider any of these factors while analyzing the algorithm We will only consider the execution time of an algorithm. The time complexity of an algorithm is commonly expressed using asymptotic notations:

- Big O - O(n),
- Big Theta - Θ(n)
- Big Omega - Ω(n)

DETAILED ANALYSIS OF ALGORITHMS

The process of finding complexity of given algorithm by counting number of steps and numbe of memory references used by using RAM model is called detailed analysis. For detailed time complexity of algorithm we count the number of steps and finally add all steps and calculate their big oh notation. Similarly for detailed analysis of space complexity we count the numbe of memory references and finally add all steps and find their big oh notation.

Example 1: Find detailed analysis of following factorial algorithm

```
#include <stdio.h>
int main()
{
        int i, n, fact = 1;
        printf("Enter a number to calculate its factorial\n");
        scanf("%d", &n);
        for (i = 1; i <= n; i++)
                fact = fact * i;
        printf("Factorial of %d = %d\n", n, fact);
        return 0;
}
```

Time complexity

The declaration statement takes 1 step

Printf statement takes 1 step time

Scanf statement takes 1 step

In for loop,

i=1 takes 1 step

i<=n takes (n+1) step

i++ takes n step

within for loop fact = fact*i takes n step

printf statement takes 1 step

return statement takes 1 step

Total time complexity = 1+1+1+1+n+1+n+n+1+1

= 2n+7

= O(1) x O(n) +O(1)

= O(n) +O(1)

= O(n)

Similarly space complexity is,

Total memory references used = 3

= O(1)

Example 2: Find detailed analysis of Bubble sort

```
Bubble_Sort(A, n)
{
        for (i=1;i<=n;i++)
        {
                for (j=0; j<n-i; j++)
                {
                        if(A[j]>A[j+1])
                        {
                                temp=A[j];
                                A[j]=A[j+1];
                                A[j+1]=temp;
                        }
                }
        }
}
```

Analysis

Space complexity

S.C. =1+1+1+1+n=n+4=O (n)

Time complexity

Within first for loop:

i=1 takes 1 step

i<=n takes (n+1) steps

i++ takes n steps

Within second for loop:

j=0 takes n step

j<n-i takes [n + (n-1) + (n-2) + --------- + 2 + 1]

j++ takes [(n-1) + (n-2) + --------- + 2 + 1]

In if statement:

It takes at most 3* (n-1)

So total time complexity (T. C) = 1+(n+1)+n+[n+{n+(n-1)+(n-2)+.....+3+2+1]+ [(n-1)+(n-2) +.....+3+2+1}]+3*[(n-1)+(n-2) +..... ...+3+2+1]

=2n+2+ [n + n (n+1)/2+n (n-1)/2] + 3*n (n-1)/2

=2n+2+n+n^2 /2+n/2+ n^2 /2-n/2+2n^2/2-3n/2

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

$= [O(1) \times O(n^2) + O(1) \times O(n)]/O(1)$

$= [O(n^2) + O(n)]/O(1)$

$= O(n^2) + O(n)$

$= O(n^2)$

Example 3: Detailed analysis of sequential search algorithm

```
Sequential_Search(A, n, key)
{
    int i, flag=0;
    for(i=0;i<n;i++)
    {
        if(A[i]==key)
        {
            flag=1;
        }
    }
    if(flag==1)
        print"Search successful"
    else
        print"Search un-successful"
}
```

Space complexity

By counting number of memory references gives the space complexity of given algorithm.

The variable 'i' takes 1 unit space

The variable 'n' takes 1 unit space

The variable 'key' takes 1 unit space

The variable 'flag' takes 1 unit space

The variable 'A' takes n unit space

Now total space complexity is;

$T(n)=1+1+1+1+n$

$=4+n = O(1) + O(n)$

$=O(n)$

Time complexity

By counting number of statements takes the total time complexity of given algorithm.

Declaration statement takes 1 step time

In for loop;

i=0 takes 1 step time

i<n takes (n+1) step time

i++ takes n step time

Within for loop if condition takes n step time

Within if statement flag=0 takes at most n step time

If statement outside the for loop takes 1 step time

Print statement takes 1 step time

Now total time complexity is given by;

$T(n) = 1 + 1 + (n + 1) + n + n + n + 1 + 1$

$= 5 + 4n \leq 5n + 4n$

$= 9n = O(1) \times O(n)$

$\Rightarrow T(n) = O(n)$

AMORTIZED COMPLEXITY

Amortized analysis is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive. It refers to finding 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. The basic idea is that a worst case operation can alter the state in such a way that the worst case cannot occur again for a long time.

There are several techniques are used in amortized analysis:

- Aggregate analysis
- Accounting method and
- Potential method

Aggregate analysis

It determines the upper bound T(n) on the total cost of a sequence of n operations, then calculations the average cost to be T(n)/n. In aggregate analysis, there are two steps. First, we must show that a sequence of n operations takes T(n) time in the worst case. Then, we show that each operation takes T(n)/n time, on average. Therefore, in aggregate analysis, each operation has the same cost.

Example: A common example of aggregate analysis is a modified stack. Stacks are a linear data structure that has two constant-time operations. **Push(element)** puts an element on the top of the stack, and **Pop()** takes the top element off of the stack and returns it. These operations are both constant-time, so a total of n operations will result in O(n) time.

Now, a new operation is added to the stack. **Multipop(k)** will either pop the top k elements in the stack, or if it runs out of elements before that, it will pop all of the elements in the stack and stop. The pseudo-code for Multipop(k) would look like this:

```
Multipop(k):
    while stack not empty and k > 0:
        k = k - 1
        stack.Pop()
```

Looking at the pseudo-code, it's easy to see that this is not a constant-time operation. Multipop can run for at most n times, where n is the size of the stack. So, the worst-case runtime for multipop is O(n). So, in atypical analysis, that means that n multipop operations take $O(n^2)$ time.

However, that's not actually the case. Think about multipop and what it's actually doing. multipop cannot function unless there's been a push to the stack because it would have nothing to pop off. In fact, any sequence of n operations of multipop, pop and push can take at most O(n) time. multipop, the only non-constant-time operation in this stack, can only take O(n) time if there have also been n constant-time push operations on the stack. In the very worst case, there are n constant-time operations and just 1 operation taking O(n) time.

For any value of n, any sequence of multi-pop, pop, and push takes O(n) time. So, using aggregate analysis,

$$\frac{T(n)}{n} = \frac{O(n)}{n} = O(1)$$

Accounting method

The accounting method is aptly named because it borrows ideas and terms from accounting. Here, each operation is assigned a charge, called the amortized cost. Some operations can be charged more or less than they actually cost. If an operation's amortized cost exceeds its actual cost, we assign the difference, called a credit, to specific objects in the data structure. Credit can be used later to help pay for other operations whose amortized cost is less than their actual cost. Credit can never be negative in any sequence of operations.

The amortized cost of an operation is split between an operation's actual cost and credit that is either deposited or used up. Each operation can have a different amortized cost, unlike aggregate analysis. Choosing the amortized cost for each operation is important, but the costs must always be the same for a given operation no matter what sequence of operations, just like for any method of amortized analysis.

Potential method

The potential method is similar to the accounting method. However, instead of thinking abou the analysis in terms of cost and credit, the potential method thinks of work already done a potential energy that can pay for later operations. This is similar to how rolling a rock up a hi creates potential energy that then can bring it back down the hill with no effort. Unlike th accounting method, however, potential energy is associated with the data structure as a whole not with individual operations.

COMPUTATIONAL AND ASYMPTOTIC COMPLEXITY

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate complexity of an algorithm it does not provide exact amount of resource required. So instead of taking exact amount of resource we represent that complexity in a general form which produces the basic nature of that algorithm. We use that general form for analysis process.

Complexity analysis of an algorithm is very hard if we try to analyze exact. We know that the complexity (worst, best, or average) of an algorithm is the mathematical function of the size of the input. So if we analyze the algorithm in terms of bound (upper and lower) then it would be easier. For this purpose, we need the concept of asymptotic notations. Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes. Three notations are used to calculate the running time complexity of an algorithm.

Why asymptotic notations important?

- They give a simple characterization of an algorithm's efficiency.
- They allow the comparison of the performance of various algorithms.

Big Oh (O) notation

When we have only asymptotic upper bound then we use O notation. If f and g are any two functions from set of integers to set of integers then function f(x) is said to be big oh of g(x) i.e. f(x)=O(g(x))) if and only if there exists two positive constants c and x_0 such that for all x >= x_0, f(x) <= c*g(x)

The above relation says that g(x) is an upper bound of f(x)

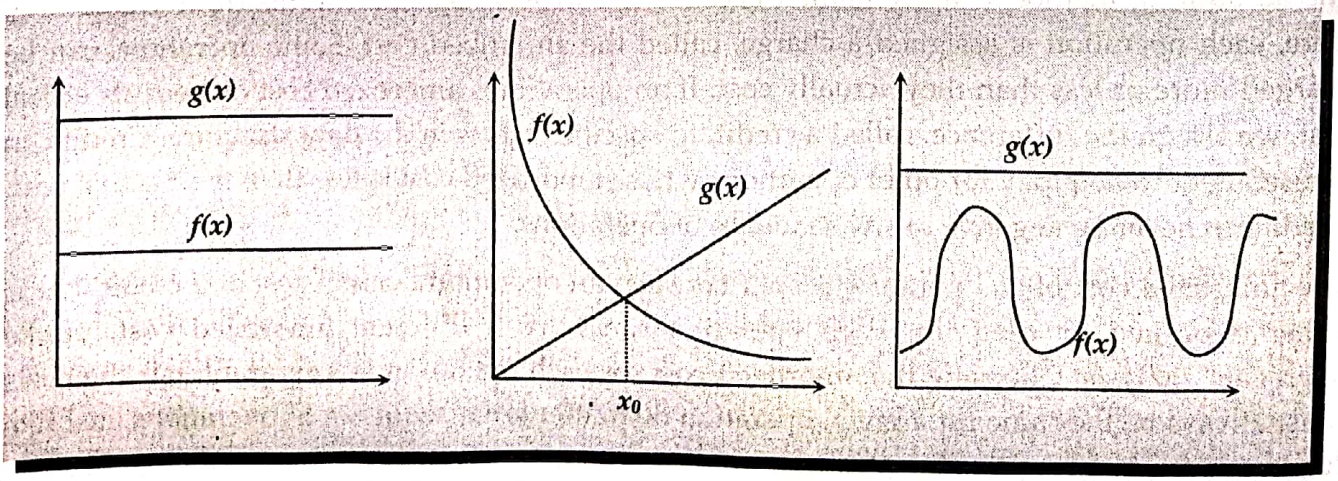


Fig: Geometric intrepretation of Big-Oh notation

Some properties

Transitivity: f(x) = O(g(x)) & g(x) = O(h(x)) then f(x) = O(h(x))

Reflexivity: f(x) = O (f(x))

Transpose symmetry: f(x) = O(g(x)) if and only if g(x)= Ω(f(x))

O (1) is used to denote constants.

For all values of n >= n0, plot shows clearly that f(n) lies below or on the curve of c*g(n)

Example: Find big oh of given function $f(n) = 3n^2 + 4n + 7$

Solution: we have $f(n) = 3n^2 + 4n + 7 <= 3n^2 + 4n^2 + 7n^2 <= 14n^2$

$f(n) <= 14n^2$

where, c=14 and $g(n) = n^2$, thus $f(n) = O(g(n)) = O(n^2)$

Big Omega (Ω) notation

Big omega notation gives asymptotic lower bound. If f and g are any two functions from set of integers to set of integers, then function f(x) is said to be big omega of g(x) i.e. f(x)=Ω(g(x)) if and only if there exists two positive constants c and x_0 such that

For all $x >= x_0$, f(x)>=c*g(x)

The above relation says that g(x) is a lower bound of f(x).

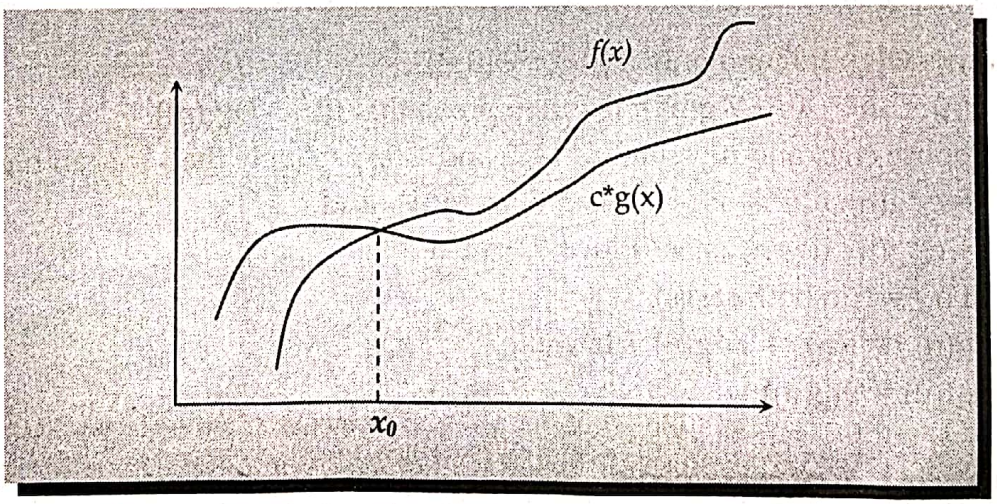


Fig: Geometric intrepretation of Big Omega notation

Some properties

Transitivity: f(x) = Ω(g(x)) & g(x) = Ω(h(x)) then f(x) = Ω(h(x))

Reflexivity: f(x) = Ω(f(x))

Example: Find big omega of $f(n) = 3n^2 + 4n + 7$

Solution: Since we have $f(n) = 3n^2 + 4n + 7 >= 3n^2$

⇨ $f(n) >= 3n^2$

where, c=3 and $g(n) = n^2$, thus f(n) = Ω(g(n)) = Ω(n^2)

Big Theta (Θ) notation

When we need asymptotically tight bound then we use this notation. If f and g are any two functions from set of integers to set of integers then function f(x) is said to be big theta of g(x) i.e. f(x) = Θ (g(x)) if and only if there exists three positive constants c1, c2 and x_0 such that for all x >= x_0, c1*g(x) <= f(x) <= c2*g(x)

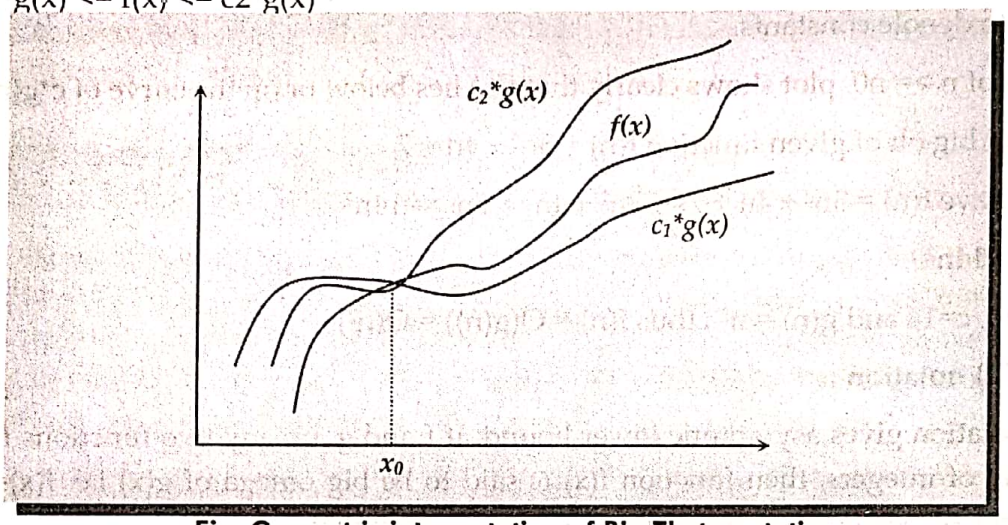


Fig: Geometric intrepretation of Big Theta notation

Some properties

Transitivity: f(x) = Θ(g(x)) & g(x) = Θ(h(x)) then f(x) = Θ(h(x))
Reflexivity: f(x) = Θ(f(x))
Symmetry: f(x) = Θ(g(x)) if and only if g(x) = Θ(f(x))
Example: If f(n) = $3n^2 + 4n + 7$ g(n) = n^2 , then prove that f(n) = Θ(g(n)).
Proof: let us choose c_1, c_2 and n_0 values as 14, 1 and 1 respectively then we can have,

f(n) <= c_1*g(n), n>=n_0 as $3n^2 + 4n + 7 <= 14*n^2$, and
f(n) >= c_2*g(n), n>=n_0 as $3n^2 + 4n + 7 >= 1*n^2$
For all n >= 1(in both cases).
So c_2*g(n) <= f(n) <= c_1*g(n) is trivial.
Hence f(n) = Θ(g(n)).

MATHEMATICAL FOUNDATION

Since mathematics can provide clear view of an algorithm. Understanding the concepts of mathematics is aid in the design and analysis of good algorithms. Here we present some of the mathematical concepts that are helpful in our study.

Exponents

Some of the formulas that are helpful are:

- $x^a x^b = x^{a+b}$
- $x^a / x^b = x^{a-b}$
- $(x^a)^b = x^{ab}$
- $x^n + x^n = 2x^n$
- $2^n + 2^n = 2^{n+1}$

Logarithms

Some of the formulas that are helpful are:

1. $\log_a b = \log_c b / \log_c a$; c>0 [making base same]
2. log ab = log a + log b
3. log a/b = log a - log b
4. log (a^b) = b log a
5. Log x < x for all x>0
6. Log 1 = 0, log 2 = 1, log 1024 = 10.
7. $a^{\log_b n} = n^{\log_b a}$

Series

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=0}^{n} a^i \le \frac{1}{1-a}; \text{if } 0 < a < 1$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

RECURRENCES

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence. To solve recursive algorithms we need to define their recurrence relation and by using any one of the recurrence relation solving method we calculate their complexity.

Examples

Recursive algorithm for finding factorial

T(n)=1 when n =1

T(n)=T(n-1) + O(1) when n>1

Recursive algorithm for finding N^{th} Fibonacci number

T(1)=1 when n=1

T(2)=1 when n=2

T(n)=T(n-1) + T(n-2) +O(1) when n>2

Recursive algorithm for binary search

T(1)=1 when n=1

T(n)=T(n/2) + O(1) when n>1

NOTE: Cost of solving recursive algorithm = cost of dividing problem + cost of solving sub problems + cost of merging solutions

Solving Recurrences

The process of finding solution of given recurrence relation in terms of big oh notation is called solving recurrences. There are a lot of methods for solving recurrence relations some of popular method are listed below:

- Iteration method
- Recursion Tree
- Substitution
- Master Method

Iteration method

Here we expand the given relation until the boundary is not meet. Expand the relation so that summation independent on n is obtained. The Iteration method is also known as the Iterative Method, Backwards Substitution, Substitution method, and Iterative Substitution. It is a technique or procedure in computational mathematics used to solve a recurrence relation that uses an initial guess to generate a sequence of improving approximate solutions for a class of problems, in which the n^{th} approximation is derived from the previous ones.

Example 1: Solve following recurrence relation by using iterative method

$T(n)= 2T(n/2) +1$ when $n>1$

$T(n)= 1$ when $n=1$

Solution:

$T(n) = 2T(n/2) +1$

$= 2 \{2T (n/4) + 1\} +1$

$= 2^2T (n/2^2) + 2 + 1$

$= 2^2 \{2T(n/2^3) +1 \} + 2 + 1$

$= 2^3 T(n/2^3) + 2^2+ 2 + 1$

.....

.....

$= 2^k T(n/2^k) + 2^{k-1} +$ ..... $+ 4 + 2 + 1.$

For simplicity assume:

$n/2^k =1$

or, $n=2^k$

Taking log on both sides,

$\log n=\log 2^k$

$\log n=k \log 2$

k=log n [since log 2=1]

Now, $T(n)= 2^k T(n/2^k) + 2^{k-1} + \ldots\ldots\ldots\ldots + 4 + 2 + 1.$

$T(n)= 2^k T(1)+ 2^{k-1} + \ldots\ldots\ldots\ldots + 2^2 + 2^1 + 2^0$

$T(n)= (2^{k+1} -1)/ (2-1)$

$T(n)= 2^{k+1} -1$

$T(n)= 2.2^k -1$

T(n)= 2n-1

T(n)= O(n)

Example 2 Solve following recurrence relation by using iterative method

T(n) = T(n/3) + O(n) when n>1

T(n) = 1 when n=1

Solution:

T(n) = T(n/3) + O(n)

T(n) = T(n/3) + cn

$T(n) = T(n/3^2) + cn/3 + cn$

$T(n) = T(n/3^3) + cn/3^2 + cn/3 + cn$

$T(n) = T(n/3^4) + cn/3^3 + cn/3^2 + cn/3 + cn$

$T(n)= T(n/3^k) + c\, n/3^{k-1} + \ldots\ldots\ldots\ldots + cn/3^2 + cn/3 + cn \ldots\ldots\ldots(1)$

Simplicity assume

$n/3^k=1$

or, $n= 3^k$

Taking log on both sides,

$\log n=\log 3^k$

$k = \log_3 n$

Now from given relation 1,

$T(n)<= T(1) + c\, n/3^{k-1}\ldots\ldots\ldots\ldots + c.n/3^2 + c.n/3 + c.n$

$T(n) <= 1 +\{ c.n/3^{k-1}\ldots\ldots\ldots\ldots + c.n/3^2 + c.n/3 + c.n\}$

$T(n)<= 1+ c.n\ \{ 1/(1-1/3) \}$

$T(n) <= 1+ 3/2\ c.n$

T(n) = O(n)

Example 3: Solve following recurrence relation by using iterative method

T(n)= T(n-1) +O(1)

Solution:

We have the recurrence relation,

T(n) = T(n – 1) + O(1)

= T(n – 2) + 1 + 1 [since O(1)=1 where choose c=1]

$= T(n-3) + 1 + 1 + 1$

$= T(n-4) + 1 + 1 + 1 + 1$

.....

$= T(n-k) + 1 + \ldots\ldots\ldots\ldots + 1$ (k times)

Lets choose n-k=1

⇨ $k = n - 1$

Now $T(n) = T(1) + 1 + \ldots\ldots\ldots + 1$ (k times)

$= 1 + k*1$

$= k + 1$

$= n - 1 + 1 = n$

$T(n)=O(n)$

Example 4: Solve following recurrence relation by using iterative method

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

Solution: We have the recurrence relation,

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

$= 2[2T(n/2^2)+ n/2] + n$

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

$= 2^2[2T(n/2^3)+ n/2^2]+n + n$

$= 2^3T(n/2^3)+ n + n + n$

.....

.....

$=2^kT(n/2^k)+ n + n + \ldots\ldots.. + n$ (k times)

Let's put $n/2^k=1$

⇨ $n= 2^k$

Taking log on both sides,

$\log n=\log 2^k$

or, $\log n=k \log 2$

$k=\log n$

Now, $T(n) = n\,T(1) + n + n + \ldots\ldots\ldots\ldots + n$ (k times)

$= n + k*n$

$= (k\,n + n)$

$= \log n*n + n$

$= n \log n + n$

$O(n \log n)$

Example 5: Solve following recurrence relation by using iterative method

$$T(n) = T(n/3) + n$$

Solution: We have the recurrence relation,

$T(n) = T(n/3) + n$

$= T(n/3^2) + n/3 + n$

$= T(n/3^3) + n/3^2 + n/3 + n$

$= T(n/3^4) + n/3^3 + n/3^2 + n/3 + n$

.....

.....

$= T(n/3^k) + n/3^{k-1} + \ldots\ldots + n/3^2 + n/3 + n$

Lets put $n/3^k = 1$

$3^k = n$

Taking log on both sides

$\log 3^k = \log n$

$k \log 3 = \log n$

$k = \log n / \log 3$

Now $T(n) = T(1) + n/3^{k-1} + \ldots\ldots + n/3^2 + n/3^1 + n/3^0$

$= 1 + n\,[1/3^0 + 1/3^1 + 1/3^2 + 1/3^3 + \ldots\ldots\ldots\ldots + 1/3^{k-2} + 1/3^{k-1}]$

Since this is a geometric series of common ration $r = 1/3$

Thus $T(n) = 1 + n[1-(1/3)^k]/[1-1/3]$

$= 1 + n\,[1-1/3^k]/2/3$

$= 1 + 3n/2\,[1-1/n]$

$= 1 + 3n/2 * (n-1)/n$

$= 1 + 3n/2 - 3/2$

$= 3n/2 - 1/2$

$= O(n)$

Hence $T(n) = O(n)$

Sum of Geometric series

- $s_n = \dfrac{a(r^n - 1)}{r - 1}$ if $r > 1$
- $s_n = \dfrac{a(1 - r^n)}{1 - r}$ if $r < 1$

Example 6: Solve the recurrence relation $T(n) = 2\,T(n/2) + n/\log n$ by using iteration method,

$T(n) = 4\,T(n/4) + n/\log(n/2) + n/\log n$

$= 8\,T(n/8) + n/\log(n/4) + n/\log(n/2) + n/\log n$

$= 2^k\,T(n/2^k) + n/\log(n/2^{k-1}) + n/\log(n/2^{k-2}) \ldots\ldots + n/\log n$

When $n = 2^k$

$= n * T(1) + n/\log 2 + n/\log 4 + n/\log 8 \ldots\ldots + n/\log n$

$= c\,n + n\,(1/\log 2 + 1/\log 4 + 1/\log 8 + \ldots\ldots 1/\log n)$

$= c\,n + n\,(1/1 + 1/2 + 1/3 + \ldots\ldots 1/k)$

$= c\,n + n \log (k)$

$= c\,n + n \log (\log n)$

$= O(n \log (\log n))$

Recursion Tree

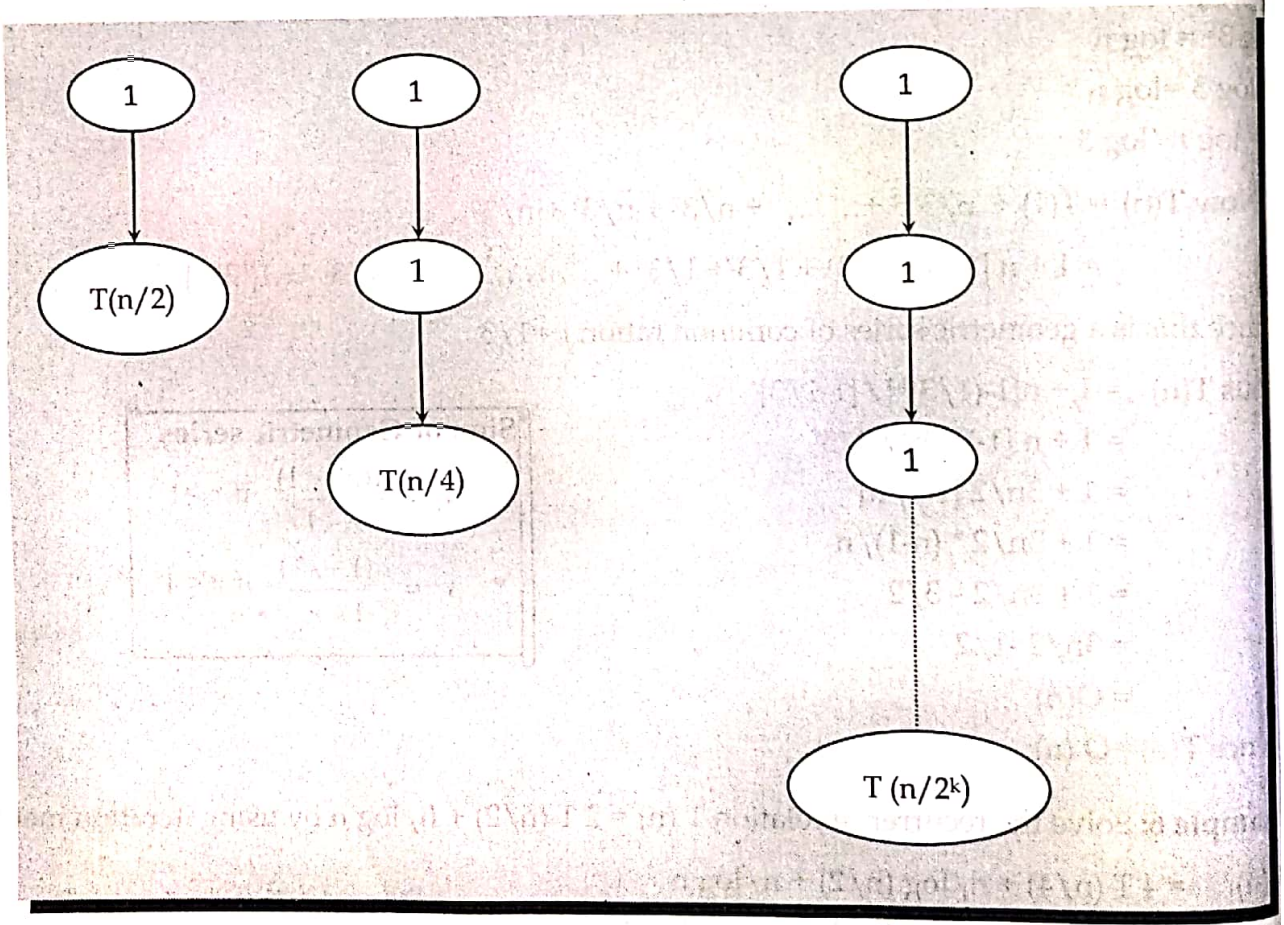
Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded. In general, we consider the second term in recurrence as root. It is useful when the divide & Conquer algorithm is used.

It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represents the cost of a single sub-problem. We sum the costs within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.

Example 1:- Solve following recurrence relation by using recursion tree method

$T(1)=1$ **when n=1**

$T(n)= T(n/2)+ 1$ **when n>1**



Cost at each level =1

For simplicity assume that $n/ 2^K=1$

⇨ $n=2^K$

Taking log on both sides,

⇨ $\log n = \log 2^K$

⇨ $k \log 2 = \log n$

⇨ $k = \log n$

Summing the cost at each level,

Total cost $= 1 + 1 + 1 + \ldots\ldots\ldots\ldots\ldots + T(n/2^K)$

$= 1+1+\ldots\ldots\ldots+1$ (k times) $+T(1)$

$= k*1+1$

$= (k+1)$

$= \log n+1$

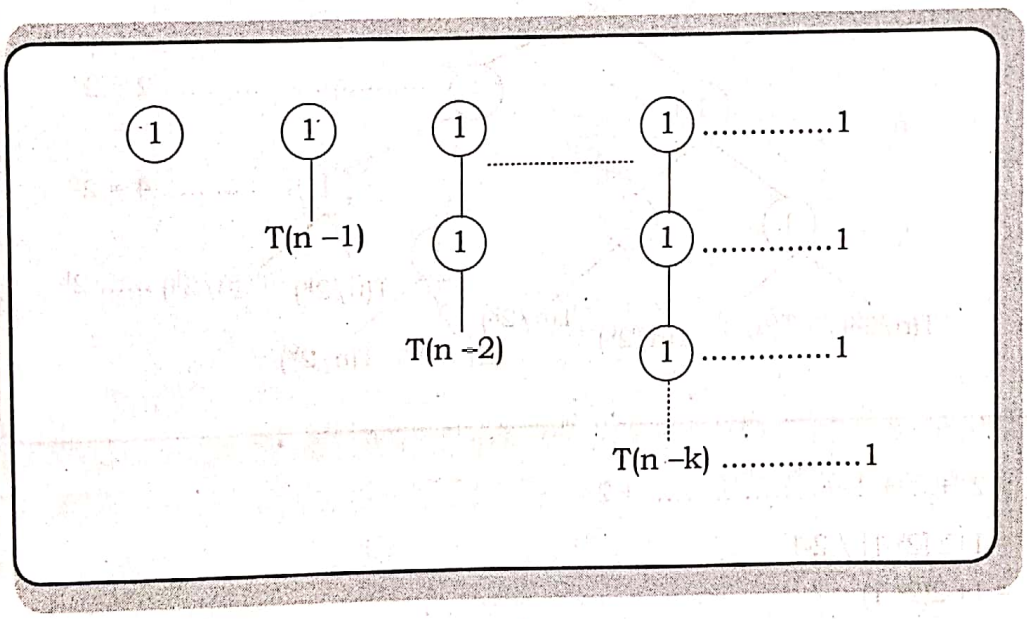
⇨ $T(n) = O(\log n)$

Example 2:- Solve following recurrence relation by using recursion tree method

$T(1) = 1$ **when n=1**

$T(n) = T(n-1) + 1$ **when n>1**

Solution:



T. C. = T (n) $= 1 + 1 + \ldots\ldots\ldots + 1$ (k times) $+ T(n-k)$

$= k * 1 + T(n-k)$

Let s put $n-k=1$

⇨ $k = n-1$

Now $T(n) = k + T(1)$

$= n-1+1 = n$

$= O(n)$

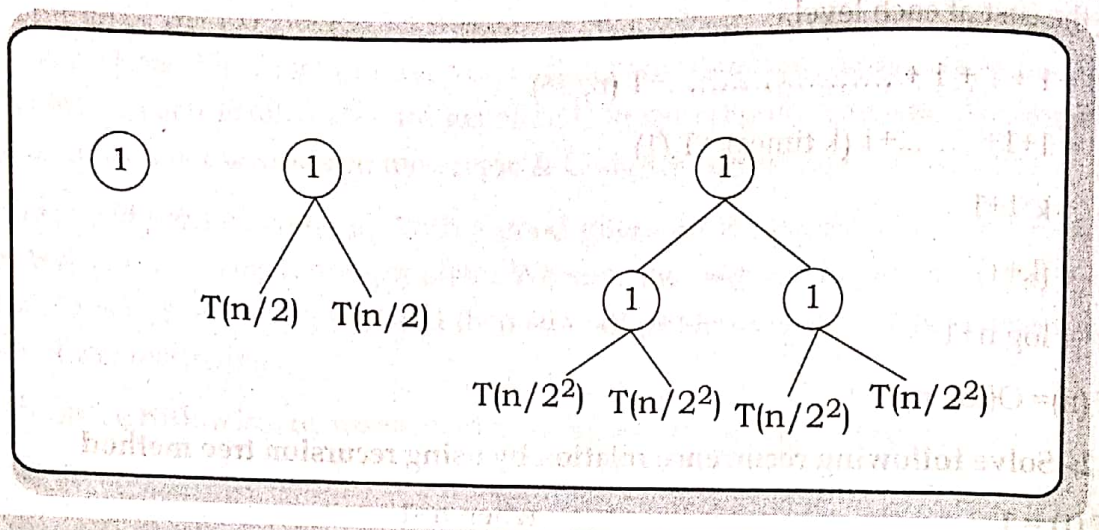
⇨ $T(n) = O(n)$

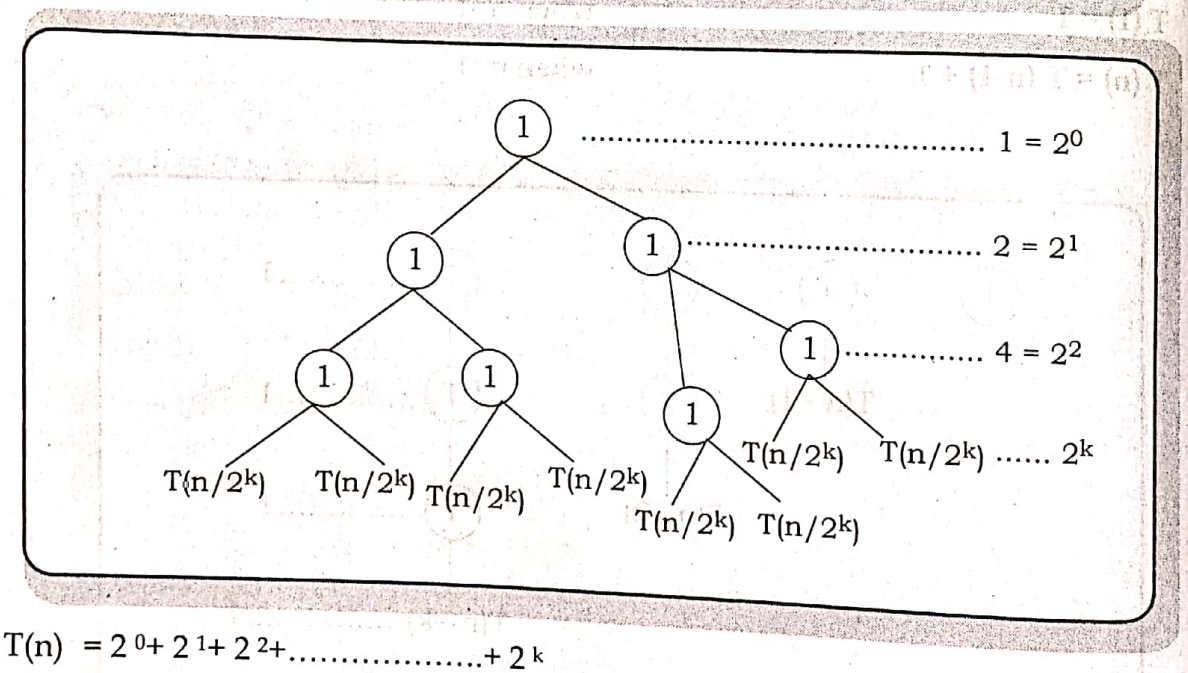
Example 3: Solve following recurrence relation by using recursion tree method

$T(1) = 1$ when n=1

$T(n) = 2\,T(n/2) + 1$ when n>1

Solution:





Now $T(n) = 2^0 + 2^1 + 2^2 + \ldots\ldots\ldots\ldots + 2^k$

$= 1+2\,[2^k-1] / 2-1$

$= 1+ 2(2^k-1)$

$= 2*2^k - 1$

For simplicity assume that $n/ 2^K=1$

⇨ $n=2^K$

Taking log on both sides,

⇨ $\log n=\log 2^K$

⇨ $k \log 2=\log n$

⇨ $k= \log n$

Now $T(n) = 2n - 1$

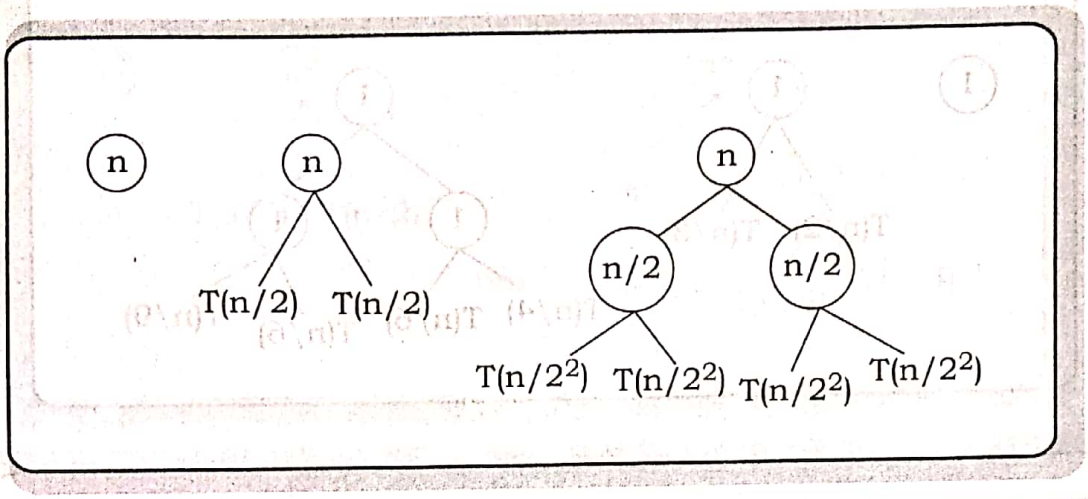
Hence, $T(n) = O(n)$

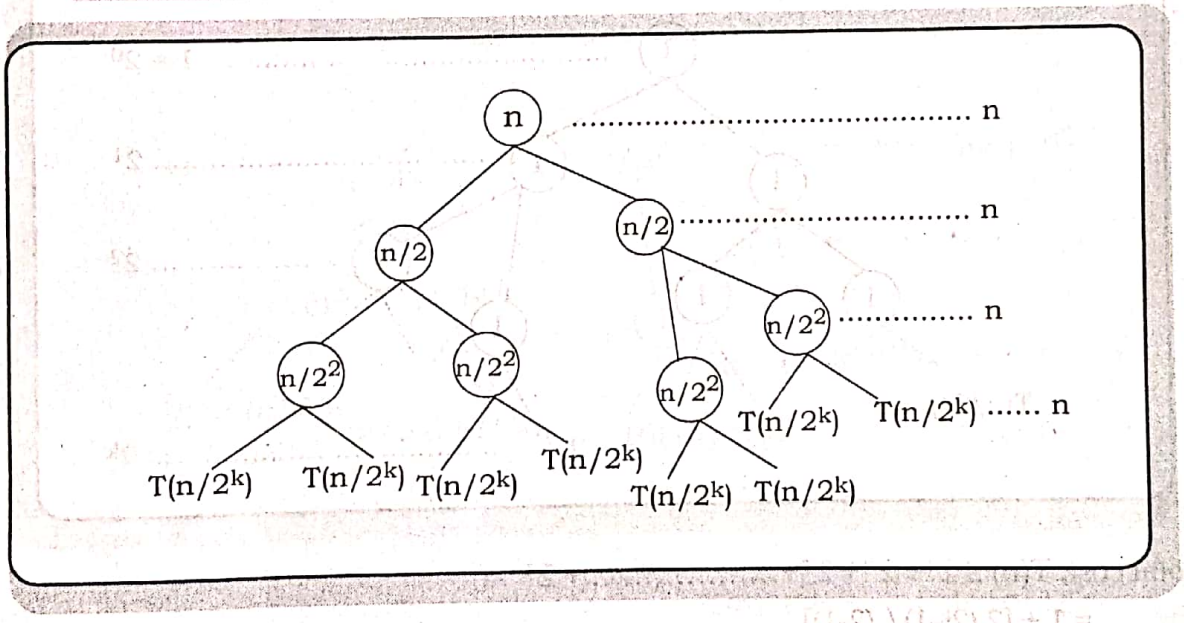
Example 4:- Solve following recurrence relation by using recursion tree method

T (1) = 1 **when n=1**

T (n) = 2 T (n/2) + n **when n>1**

Solution:





Now T(n) $= n + n + n + \ldots\ldots + n$ (k times) $+ T(n/2^k)$

$= n^* k + T(n/2^k)$

For simplicity assume that $n/ 2^K=1$

⇨ $n=2^K$

Taking log on both sides,

⇨ $\log n=\log 2^K$

⇨ $k \log 2=\log n$

⇨ $k= \log n$

Now T (n) = kn + T(1)

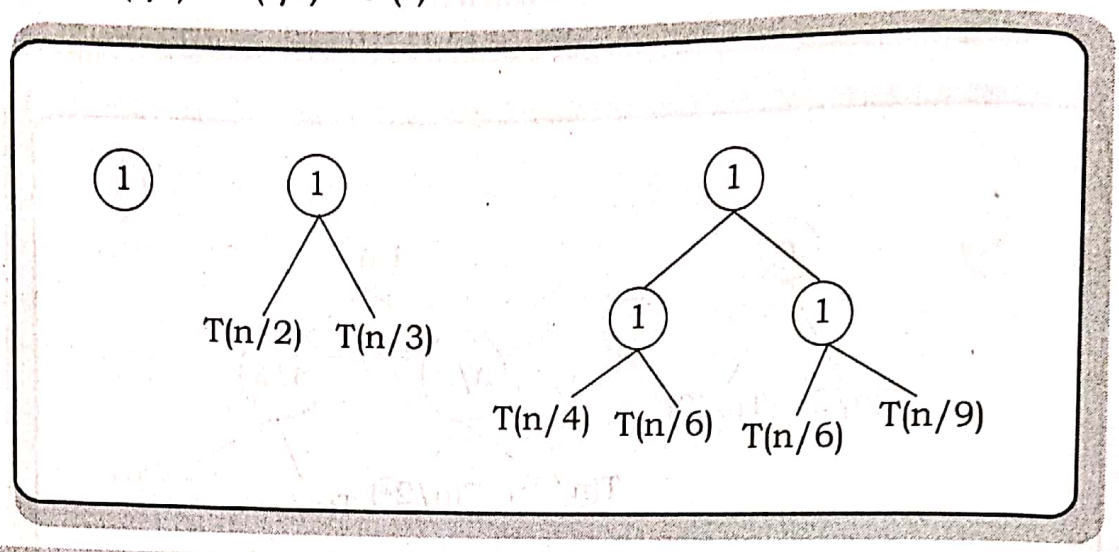
= n log n +1

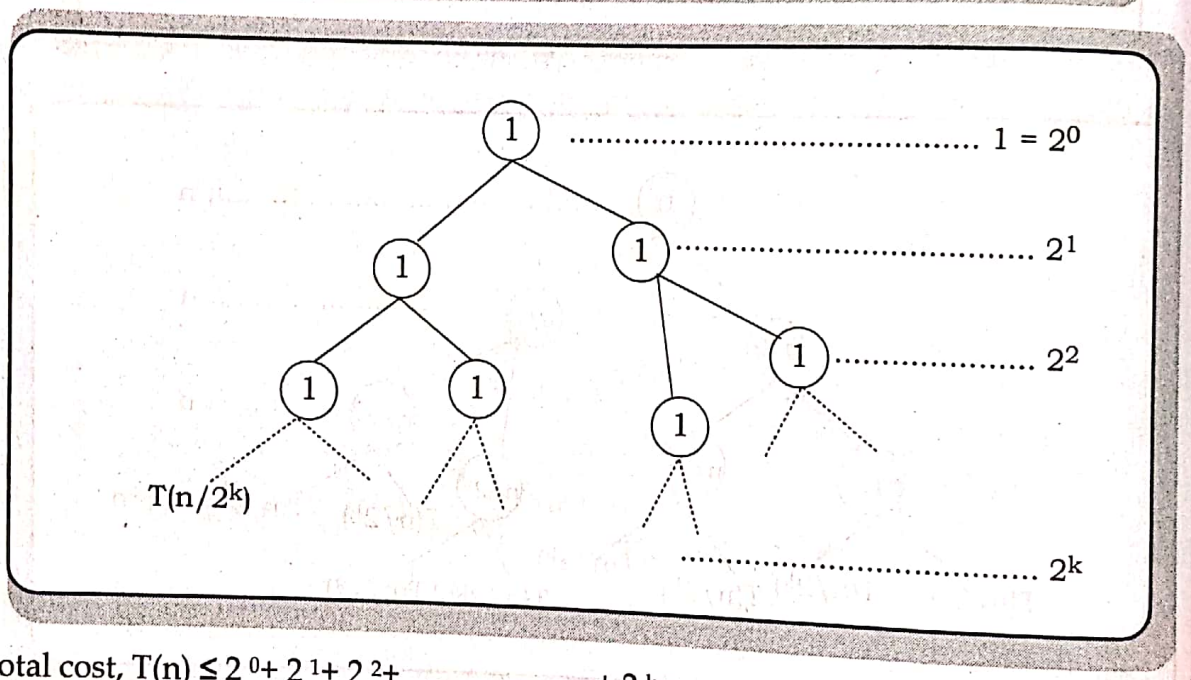
Hence, T (n) = O (n log n)

Example 5: Solve following recurrence relation by using recursion tree method

$T(1) = 1$ when n=1

$T(n) = T(n/2) + T(n/3) + O(1)$ when n>1





Now total cost, $T(n) \leq 2^0 + 2^1 + 2^2 + \ldots\ldots\ldots\ldots + 2^k$

$= 1 + [2(2^k-1)/(2-1)]$

$= 1 + 2*2^k - 2$

$= 2*2^k - 1$

Lets put $n/2^k=1$

⇨ $n=2^K$

Taking log on both sides,

⇨ $\log n = \log 2^K$

⇨ $k \log 2 = \log n$

⇨ $k = \log n$

Now $T(n) \leq 2*n - 1$

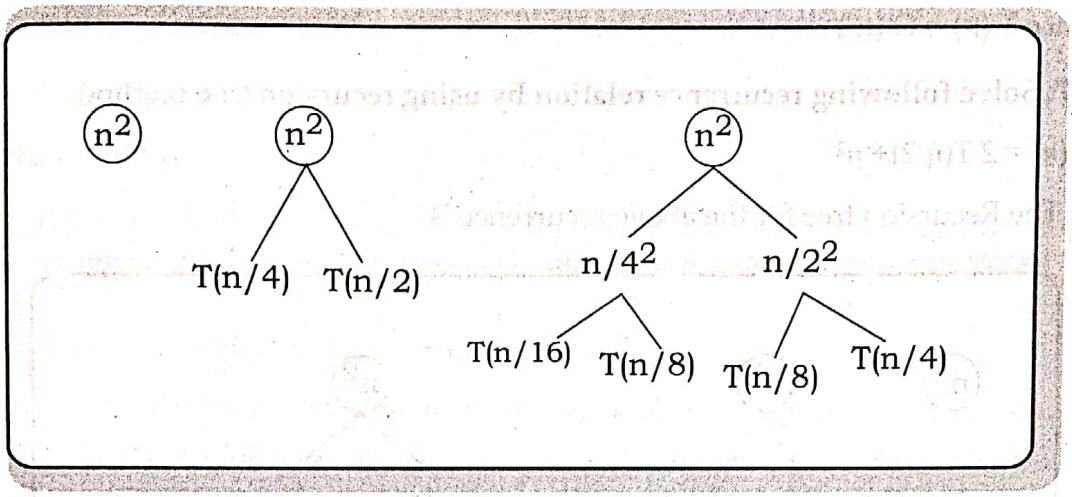
$\leq 2n-1$

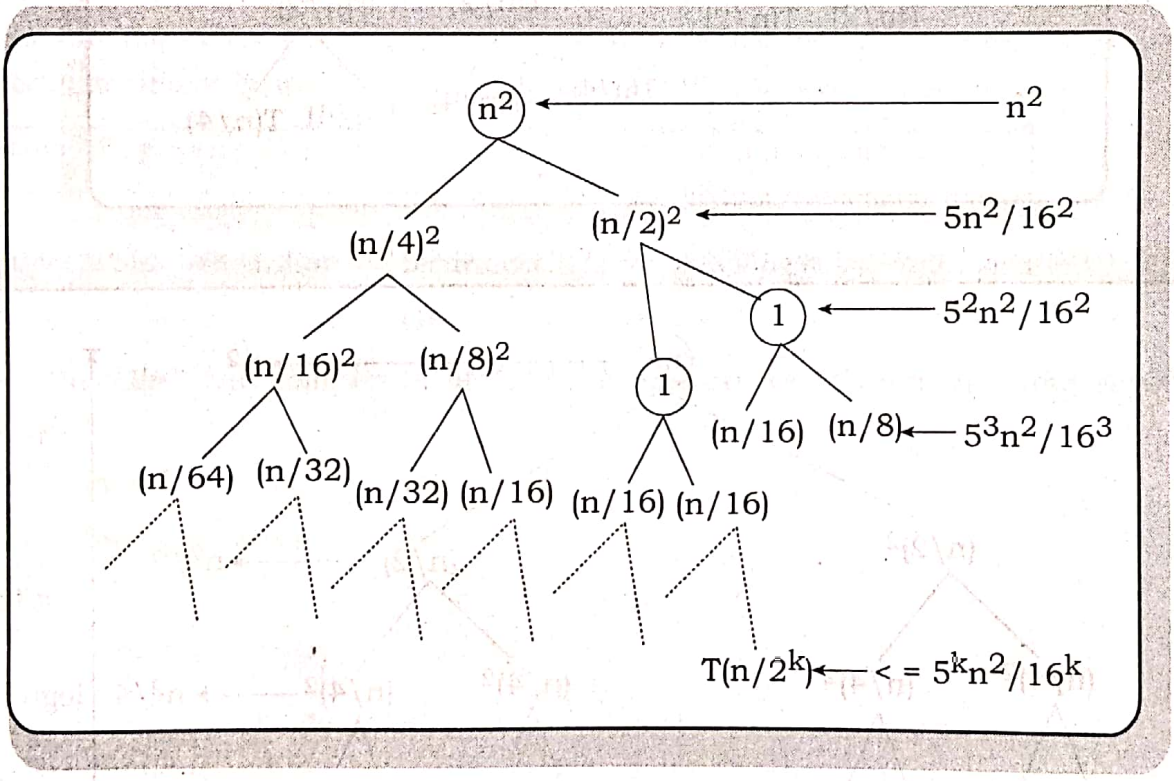
Hence, $T(n) = O(n)$

Example 6: Solve following recurrence relation by using recursion tree method

$T(n) = T(n/4) + T(n/2) + n^2$

Solution:





Total Cost ≤ $n^2 + 5n^2/16 + 5^2n^2/16^2 + 5^3n^2/16^3 + \ldots\ldots + 5^kn^2/16^k$

{Why ≤ ? Why not =?}

$\leq n^2 (1 + 5/16 + 5^2/16^2 + 5^3/16^3 + \ldots\ldots + 5^k/16^k)$

$\leq n^2 [1 + (5/16 + 5^2/16^2 + 5^3/16^3 + \ldots\ldots + 5^k/16^k)]$

$\leq n^2 + [1 + (1 - 5^k/16^k) / (1 - 5/16)]$

$\leq n^2 + [1 + 11(1 - 5^k/16^k) / 16]$

Let's put $n/2^k = 1$

⇨ $n = 2^k$

Taling log on both sides

$\log n = \log 2^k$

⇨ $\log n = k \log 2$

⇨ k=log n [since log 2=1]

Now $T(n) \leq n^2 + [1 + 11(1-5^{\log n}/16^{\log n}) / 16]$

$\leq n^2 + \text{constant}$

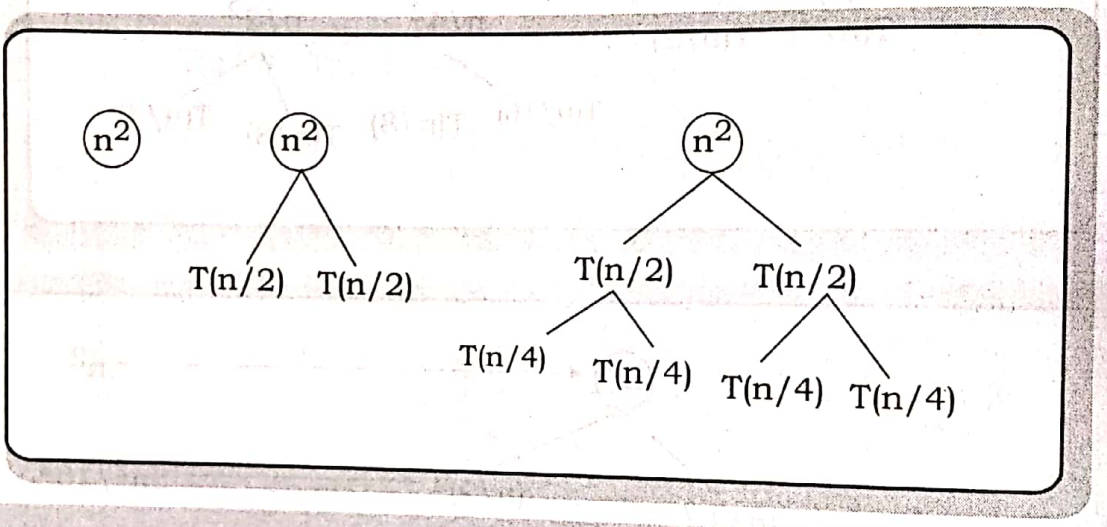
$\leq O(n^2)$

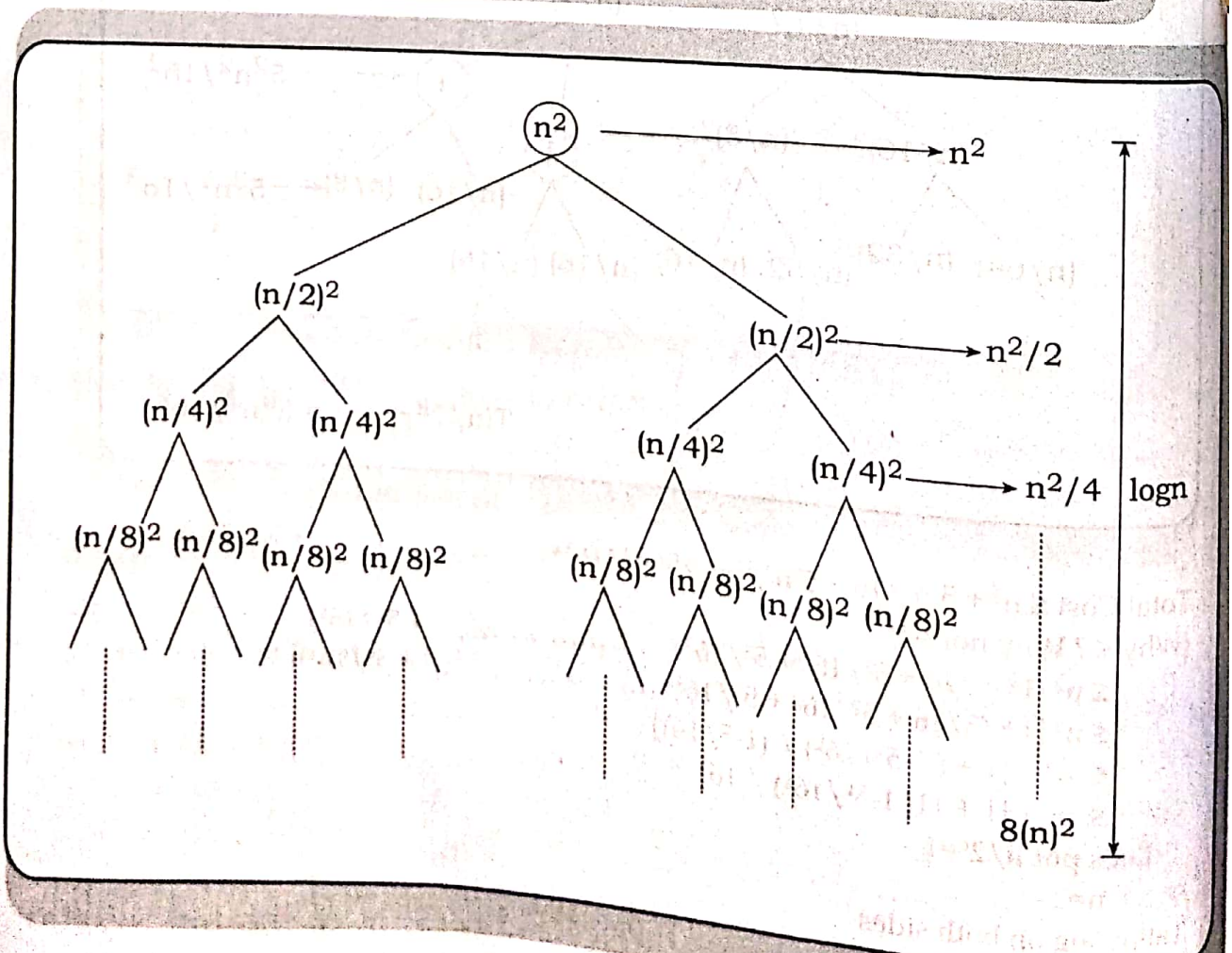
Hence $T(n) = O(n^2)$

Example 7: Solve following recurrence relation by using recursion tree method

$T(n) = 2\,T(n/2) + n^2$

Solution: The Recursion tree for the above recurrence is,





$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots\ldots\ldots\ldots$log n times

$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$

$\leq n^2 \left(\frac{1}{1-1/2}\right) \leq 2n^2$

$\Rightarrow \quad T(n) = O(n^2)$

Substitution Method

The substitution method for solving recurrences is famously described using two steps:

- Guess the form of the solution.
- Use induction to show that the guess is valid.

This method is especially powerful when we encounter recurrences that are non-trivial and unreadable via the master theorem. We can use the substitution method to establish both upper and lower bounds on recurrences. The name comes from the substitution of the guessed answer for the function when the inductive hypothesis is applied to smaller values. This method is powerful but it is only applicable to instances where the solutions can be guessed.

Note: Initially guessing the solution of a problem depends upon your practices.

Example 1: Solve the following recurrence relation by using substitution method.

OR

Show the $O(n^3)$ the complexity of following recurrence relation by using substitution method.

$T(n) = 1$ **n=1**

$T(n) = 4T(n/2) + n$ **n>1**

Solution:

Guess, $T(n) = O(n^3)$

$\Rightarrow T(n) \leq cn^3$, for all n >= n_0.....(1)

Now prove this by mathematical induction as,

Base step: For n=1

$T(n) = c*1^3$ Definition

$1 \leq c$ which is true for all +ve values of c

Inductive step: Lets assume that it is true $\forall$ k < n

Then $T(k) \leq ck^3$.....(2)

It is also true for k=n/2

Now equation 2 becomes,

$T(n/2) \le c\,(n/2)^3$

$= c\,n^3/8$

Now, $T(n) = 4T(n/2) + n$

$\le 4\,c\,n^3/8 + n$

$= c\,n^3 / 2 + n$

$= c\,n^3 - c\,n^3/2 + n$

$= c\,n^3 - n\,(c\,n^2/2 - 1) \le c\,n^3$

Hence $T(n) \le c\,n^3$ for $\forall n>0$

Thus $T(n) = O(n^3)$ Proved

Example 2: Show that $O(n^2)$ is the solution of following recurrence relation by using substitution method

$T(n) = 1$ $n=1$

$T(n) = 4T(n/2) + n$ $n>1$

Solution:

Guess: $T(n) = O(n^2)$.

$T(n) \le cn^2$ for $\forall\ n > n0$.....(1)

Now proof this relation by using mathematical induction

Base step: For n=1,

$T(n) = c*1^2$ Definition

$1 \le c$ which is true for all +ve values of c

Inductive step: Lets assume that it is true $\forall\ k < n$

Then $T(k) \le ck^2$.....(2)

It is also true for k=n/2

Now equation 2 becomes,

$T(n/2) \le c\,(n/2)^2$

$= c\,n^2/4$

Now, $T(n) = 4T(n/2) + n$

$\le 4\,c\,n^2/4 + n$

$= c\,n^2 + n$

$\Rightarrow T(n) = c\,n^2 + n$

It is not possible to show that $c\,n^2 + n \le c\,n^2\ \forall\ n>0$, thus we try to subtract lower order term as,

Since $T(n) = O(n^2)$

$\Rightarrow T(n) \le c\,n^2 - dn$ [since $c\,n^2 - dn \le c\,n^2$] .....(3)

Where c and d are +ve constants

Now proof this relation by using mathematical induction,

Base step: For n=1,

$T(n) = c*1^2 - d*1$ Definition

$1 \le c - d$ which is true for all +ve values of c and d < c

Inductive step: Lets assume that it is true $\forall$ k < n

Then $T(k) \le ck^2 - d\,k$.....(4)

It is also true for k=n/2

Now equation 4 becomes,

$T(n/2) \le c\,(n/2)^2 - d\,n/2$

$= c\,n^2/4 - d\,n/2$

Now, $T(n) = 4T(n/2) + n$

$\le 4[c\,n^2/4 - d\,n/2] + n$

$\le c\,n^2 - 2d\,n] + n$

$\le c\,n^2 - d\,n - d\,n + n$

$\le (c\,n^2 - d\,n) - n\,(d-1) \le (c\,n^2 - d\,n)$

$\Rightarrow T(n) \le (c\,n^2 - d\,n)\ \forall\ n>0$

Thus $T(n) = O(n^2)$ Proved

Example 3: show that $O(n^3)$ is the solution of following recurrence relation by using substitution method

$T(n)= 8\,T(n/2) + n^2$

Solution:

Let's guess, $T(n) = O(n^3)$

$T(n) \le cn^3$ for $\forall\ n > n_0$.....(1)

Now proof this relation by using mathematical induction

Base step: For n=1,

$T(n) = c*1^3$ Definition

$1 \le c$ which is true for all +ve values of c

Inductive step: Lets assume that it is true $\forall$ k < n

Then $T(k) \le ck^3$.....(2)

It is also true for k=n/2

Now equation 2 becomes,

$T(n/2) \le c\,(n/2)^3$

$= c\,n^3/8$

Now, $T(n) = 8T(n/2) + n^2$

$\le 8\,c\,n^3/8 + n^2$

$= c\,n^3 + n^2$

$\Rightarrow T(n) = c\,n^3 + n^2$

It is not possible to show that $c\,n^3 + n^2 \le c\,n^3\ \forall\ n>0$, thus we try to subtract lower order term as

Since $T(n) = O(n^2)$

$\Rightarrow T(n) \le c\,n^3 - dn^2$ [since $c\,n^3 - dn^2 \le c\,n^3$] .....(3)

Where c and d are +ve constants

Now proof this relation by using mathematical induction,

Base step: For n=1,

$T(n) = c*1^2 - d*1$ Definition

$1 \le c - d$ which is true for all +ve values of c and $d < c$

Inductive step: Lets assume that it is true $\forall\ k < n$

Then $T(k) \le ck^3 - d\,k^2$.....(4)

It is also true for k=n/2

Now equation 4 becomes,

$T(n/2) \le c\,(n/2)^3 - d(n/2)^2$

$= c\,n^3/8 - d\,n^2/4$

Now, $T(n) = 8T(n/2) + n^2$

$\le 8[c\,n^3/8 - d\,n^2/4] + n^2$

$\le c\,n^3 - 2d\,n^2 + n^2$

$\le c\,n^3 - d\,n^2 - d\,n^2 + n$

$\le (c\,n^3 - d\,n^2) - n\,(d\,n - 1) \le (c\,n^3 - d\,n^2)$

$\Rightarrow T(n) \le (c\,n^3 - d\,n^2)\ \forall\ n>0$

Thus $T(n) = O(n^3)$ **Proved**

Changing Variables

Sometimes a little algebraic manipulation can make an unknown recurrence similar to one we have seen.

Consider the example

$T(n) = 2T(\lfloor n^{1/2} \rfloor) + \log n$

Looks Difficult: Rearrange like

Let $m = \log n$

$\Rightarrow n = 2^m$

Thus,

$$T(2^m) = 2T(2^{m/2}) + m$$

Again let $S(m) = T(2^m)$

$S(m) = 2S(m/2) + m$

We can show that

$S(m) = O(m \log m)$

$T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log \log n)$

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.

$$T(n) = a\,T(n/b) + f(n)$$

Where, $a \geq 1$, $b > 1$ are constant, $f(n)$ asymptotically positive function

If the recurrence relations is in this form then there are following four possible cases occurred:

Master Method Case 1

If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constants $\varepsilon > 0$

Then

$$T(n) = O(n^{\log_b a})$$

Master Method Case 2

If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constants $\varepsilon > 0$

Then

$$T(n) = O(f(n))$$

Master Method Case 3

If $f(n) = \Theta(n^{\log_b a})$ for some constants $\varepsilon > 0$

Then

$$T(n) = O(f(n) \cdot \log n)$$

In the above three cases we are comparing the values of $f(n)$ and $n^{\log_b a}$ and then find complexity of the given recurrence relation.

Master Method Case 4: In this case the master method cannot be applied

Example 1: Solve the following recurrence relation by using Master's method

$$T(n)= 3\,T(n/2) + n$$

Solution:

Here we have a=3, b=2 and f(n) = n

Now, $n^{\log_b a} = n^{\log_2 3} = n^{(\log 23/ \log 22)} = n^{1.584}$

Also $f(n) = n^1$

Since $f(n) \le n^{\log_b a - \varepsilon}$ where choose $\varepsilon = 0.1$

Thus it satisfy the first case of Master's method

Thus it's complexity,

$$T(n)= \Theta (n^{\log_b a}) = \Theta (n^{\log_2 3}) = \Theta (n^{1.58})$$

Thus $T(n)= \Theta (n^{1.58})$

Example 2: Solve the following recurrence relation by using Master's method

$$T(n) = 4\,T(n/2) + n^2$$

Solution:

Here we have a=4, b=2 and $f(n) = n^2$

Now, $n^{\log_b a} = n^{\log_2 4} = n^{2 \log_2 2} = n^2$

Also $f(n) = n^2$

Since $f(n) = n^{\log_b a}$

Thus it satisfy the third case of Master's method

Thus it's complexity,

$$T(n)= \Theta (f(n) \log n) = \Theta (n^2 \log n)$$

Thus $T(n) = \Theta (n^2 \log n)$

Example 3: Solve the following recurrence relation by using Master's method

$$T(n) = 9\,T(n/3) + n$$

Solution:

Here we have a=9, b=3 and f(n) = n

Now, $n^{\log_b a} = n^{\log_3 9} = n^{2 \log_3 3} = n^2$

Also $f(n) = n$

Since $f(n) \le n^{\log_b a - \varepsilon}$ where choose $\varepsilon = 0.1$

Thus it satisfy the first case of Master's method

Thus it's complexity,

$T(n)= \Theta (n^{\log_b a}) = \Theta (n^{\log_3 9}) = \Theta (n^2)$

Thus $T(n)= \Theta (n^2)$

Example 4: Solve the following recurrence relation by using Master's method

$T(n) = 3\,T(n/4) + n \log n$

Solution:

Here we have a=3, b=4 and $f(n) = n \log n$

Now, $n^{\log_b a} = n^{\log_4 3} = n^{0.658}$

Also $f(n) = n \log n$

Since $f(n) \geq n^{\log_b a + \varepsilon}$ where choose $\varepsilon = 0.1$

Thus it satisfy the second case of Master's method

Thus it's complexity,

$T(n)= \Theta (f(n)) = \Theta (n \log n)$

Thus $T(n)= \Theta (n \log n)$

Example 5: Solve the following recurrence relation by using Master's method

$T(n) = 2\,T(n/4) + \sqrt{n}$

Solution:

Here we have a=2, b=4 and $f(n) = \sqrt{n} = n^{1/2} = n^{0.5}$

Now, $n^{\log_b a} = n^{\log_4 2} = n^{0.5}$

Also $f(n) = n^{0.5}$

Since $f(n) = n^{\log_b a}$

Thus it satisfy the third case of Master's method

Thus it's complexity,

$T(n)= \Theta (f(n) \log n) = \Theta (n^{0.5} \log n)$

Thus $T(n) = \Theta (n^{0.5} \log n)$

Example 6: Solve the following recurrence relation by using Master's method

$T(n) = 2\,T(2n/3) + 1$

Solution:

At first convert this relation into Master's form as,

$T(n) = 2\,T(n/3/2) + n^0$

$\Rightarrow T(n) = 2\,T(n/1.5) + n^0$

Now compare this with standard form of master's method $T(n)=aT(n)/b + f(n)$

Where, a=2, b=1.5 and $f(n) = n^0$

Now, $n^{\log_b a} = n^{\log_{1.5} 2} = n^{\log_{10} 2 / \log_{10} 1.5} = n^{1.709}$

Since $f(n) \leq n^{\log_b a}$

Thus requird solution is,

$T(n) = O(n^{\log_b a})$

Thus, $T(n) = O(n^{1.709})$

Example 7: Solve following recurrence relation by using master method,

$T(n) = 4\,T(n/2) + n^2/\log n$

Solution:

Comparing this relation with the general form of master relation

$T(n) = a\,T(n/b) + f(n)$

Where $a = 4$, $b = 2$ and $f(n) = n^2/\log n$

Now, $n^{\log_b a} = n^{\log_2 4} = n^{2 \log_2 2} = n^2$

Test case 1:- $f(n) = O(n^{\log_b a - \varepsilon})$

$\Rightarrow f(n) \leq n^{\log_b a - \varepsilon}$

Or, $f(n) \leq n^{2-\varepsilon}$

$$\text{Or, } \frac{n^2}{\log n} \leq \frac{n^2}{n^{\varepsilon}}$$

$$\text{Or, } \frac{n^2}{\log n} \leq \frac{n^2}{n^{0.1}}$$ where we choose $\varepsilon = 0.1$

$\Rightarrow n^2 n^{0.1} \leq n^2 \log n$

To satisfy this relation the value of log n must be greater than $n^{0.1}$,

But $n^{0.1}$ is a polynomial in 0.1 thus $n^{0.1}$ must be greater than log n

i.e. $n^{0.1} > \log n$

Thus master method failed in this case

Test case 2:- $f(n) = \Omega(n^{\log_b a + \varepsilon})$

$\Rightarrow f(n) \geq n^{\log_b a + \varepsilon}$

$$\text{Or, } \frac{n^2}{\log n} \geq n^{2+\varepsilon}$$

Or, $\frac{n^2}{\log n} \geq n^{2+0.1}$

Or, $n^2 \geq n^2 n^{0.1} \log n$ which is false

Test case 3:- $f(n) = \Theta(n^{\log_b a})$

⇨ $f(n) = n^{\log_b a}$

Or $\frac{n^2}{\log n} = n^2$

Or, $n^2 = n^2 \log n$ which is false

Since master method is false in all three cases thus, in this recurrence relation master method can not be applied.

DISCUSSION EXERCISE

1. What is the importance of asymptotic notation? Explain.
2. What is detailed analysis of algorithm? Detailed analyze the sequential search algorithm.
3. Show that the solution of T(n) = 2T(n / 2) + n is Ω(n log n). Conclude that solution is O(n log n).
4. Define big oh, big omega and big theta with suitable example.
5. What do you mean by best case and worst case complexity of the algorithm?
6. Show that the solution to T(n) = 2T(n / 2 + 17) + n is O(n log n).
7. Write recursive Fibonacci number algorithm derive recurrence relation for it and solve by substitution method. {Guess 2n}
8. Argue that the solution to the recurrence T(n) = T(n/3) + T(2n/3) + n is (n log n) by appealing to a recursion tree.
9. Use iteration to solve the recurrence T(n) = T(n-a) + T(a) + n, where a >=1 is a constant.
10. The running time of an algorithm A is described by the recurrence T(n) = 7T(n/2) + n2. A competing algorithm A' has a running time of T'(n) = a T'(n/4) + n^2. What is the largest integer value for 'a' such that A' is asymptotically faster than A?
11. Find big oh and big omega of following function,
    $F(x)=5n^3 + 6n^2 + 9n + 3$
12. Define RAM model. Why do you need the algorithm analysis? Discuss Big-O and Big-Ω notations used in algorithm analysis with suitable examples.

13. Write recurrence relation for following segment of code and then find their big oh.

```
void main ( )
{
int L, M, N, i, j, k, a[ ], b[ ], c[ ];
printf ("Enter value of L, M and N");
scanf ("%d %d %d", &L, &M, &N);
for (i=0; i<=L; i++)
{
for (j=1; j<=M; j++)
{
b[j]=2*j;
}
    }
for (k=N; k>=0; k--)
c[k]=-k;
}
```

14. Make tight big oh analysis of following code segment.

```
void main ( )

{
int  sum=0,i,j, a[ ][ ];
for(i=0; i<n;i++)
{
  for(j=1; j<n-i; j++)
  {
            sum=sum+a[i][j];
  }
}
}
```

15. What is the importance of asymptotic notations in DAA? Write down the algorithm f selection sort then find their tighter big oh by using RAM model.

16. What is the main concept behind big oh notation? Find big oh of following function:

$F(x) = 9x^3+8x+8$

17. What is the time complexity of fun()?

```
int fun(int n)
{
int count = 0;
for (int i = 0; i < n; i++)
  for (int j = i; j > 0; j--)
          count = count + 1;
return count;
}
```

18. Use master method find solution of following recurrence relation
    I. $T(n)=5T(n/2)+n^3$
    II. $T(n)=6T(2n/3)+n \log n$
19. Explain the properties of an algorithm with an example.
20. Give the algorithm for matrix multiplication and find the time complexity of the algorith using step – count method.

❑❑❑