unit-1

INTRODUCTION

Chetana Hegde
9844830189

Ever since man invented the idea of a machine which could perform basic mathematical operations the study of what can be computed and how it can be done in a better manner was started. This study has led to the discovery of many important algorithms and design methods. Gradually, the study of algorithms became a branch of computer science.

As computer applications are being used in every angle of human life now-a-days the study of algorithms is becoming inevitable for a computer professional. Moreover, the study of algorithms helps in developing the analytical skills of a person. This is because algorithms provide the procedures for getting the solution of a problem rather than the solution itself. So, algorithms can be thought of as problem solving strategies irrespective of the field of computer science. Thus, it is a general-purpose mental tool for

understanding other subjects like economics, linguistics, chemistry, psychology etc.

Notion of Algorithm :-

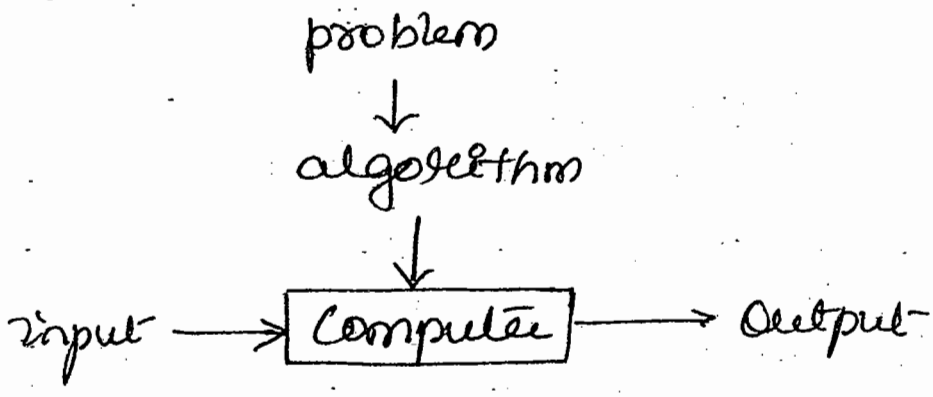
An algorithm is a sequence of unambiguous instructions for solving a problem, i.e. for obtaining a required output for any legitimate input in finite amount of time.

Thus, all the algorithms must satisfy the following criteria-

* Input : Zero or more quantities supplied externally.
* Output : At least one quantity must be produced.
* Definiteness : Each instruction of an algorithm must be clear and unambiguous.
* Finiteness : If we trace the algorithm, then for all the cases, the algorithm must terminate after finite number of steps.
* Effectiveness : Every instruction of the algorithm must be very basic and feasible so that it can be understood by common man.

Here, first two criteria indicates that input can be supplied to an algorithm and inturn it must produce at least one output. The third criteria indicates that the instructions should not contain unclear statements like "add a or 3 to a", "divide 5 by 0" etc.

The definition of algorithm can be depicted using the diagram-



Note that, here the term 'computer' may not exactly the device computer, it may be even a person who will do computation also.

While designing an algorithm for a particular problem, one must remember the following aspects-

* Non ambiguity must not be compromised at any step.

* The range of inputs for which an algorithm works has to be specified carefully.
* The same algorithm can be represented in different ways.
* Different algorithms may exist to solve the same problem.
* The different algorithms of a same problem may work at exclusively different speeds.

To illustrate these aspects, consider an example of finding the greatest common divisor of two numbers.

Let m and n be two non-negative integers such that at least one of them is not zero, then, the Euclid's algorithm says

$$gcd(m, n) = gcd(n, m \bmod n).$$

for ex, $gcd(75, 20) = gcd(20, 15)$

$= gcd(15, 5)$

$= gcd(5, 0)$

$= 5$ ...

This algorithm seems to be very strong when we choose m and n as large numbers.

Euclid's algorithm can be stated as follows.

step 1. If $n = 0$, return m and stop. Otherwise go to step 2.

step 2. Divide m by n and assign the reminder to r.

step 3. Assign the value of n to m and the value of r to n. Go to step 1.

The above algorithm may be expressed using pseudocode as below—

```
ALGORITHM Euclid(m, n)
  // Computes gcd(m, n) by Euclid's method.
  // Input: Two non-negative, not-both-zero
            integers m and n.
  // Output: GCD of m & n.
  while n ≠ 0 do
      r ← m mod n
      m ← n
      n ← r
  return m.
```

In this algorithm, as the second number is gradually decreasing and as it cannot be negative, we can easily say that the algorithm will not enter into infinite loop and it will stops after finite number of steps.

Let us go through one more algorithm for finding g.c.d. of two numbers. As we know, gcd of two numbers cannot be greater than the smallest of given nos, we will first check this smallest no is gcd or not. If it is, then we will stop. Otherwise, decrease that number by one and find out whether new number divides given two numbers. The procedure is continued till we get such a number.

i.e. algorithm can be written as —

Step 1. Assign $t \leftarrow \min\{m, n\}$

Step 2. Divide m by t. If the remainder is zero, goto step 3, otherwise go to step 4.

Step 3. Divide n by t. If the remainder of this division is zero, return t and stop. Otherwise goto step 4.

Step 4. Decrease value of t by 1. Go to Step 2.

Here, we can easily see that the algorithm does not work if any one of the given numbers is zero. Thus, this algorithm provides the proof for the aspect that ~~how~~ specifying the range of inputs for an algorithm is very important.

Now, let us consider one more algorithm for computing gcd of two numbers m and n.

Step 1. Find prime factors of m.

2. Find prime factors of n.

3. Identify all the common prime factors obtained in step 1 & 2.

4. Compute the product of these common prime factors and return the same.

For ex, gcd(75, 20) —

$75 = 3 \times 5 \times 5$

$20 = 2 \times 2 \times 5$

$\therefore$ gcd(75, 20) = 5

Chetana Hegde
9448301894

This algorithm also can not be supported as it will not tell how to ~~calculate~~ find prime factors and how to find out common

prime factors among two lists. Thus, there is ambiguity in the algorithm.

As we need algorithm for generating prime numbers now, consider one such algorithm called 'Sieve of Eratosthenes'. In this algorithm of finding all primes less than a particular number n, first we list all consecutive integers from 2 to n. In the first iteration, all the multiples of 2 are eliminated. Then in next step, multiples of 3 are eliminated. In the same manner we continue till we get a list in which the remaining elements have not got multiples. Note that multiples of 4 or such other nos are not taken for itera-tions as it is already eliminated by the number 2.

But, the problem in this case is, the numbers like 6 which is multiple of both 2 and 3 are eliminated more than once ~~at~~ different iterations and hence an overhead occurs.

While writing an algorithm for this, we will come across one question: what is the largest integer whose multiples still exist in the list? ie up to what number we have to consider for elimination? Let that number be p. Then its multiples are $2p, 3p, \ldots, (p-1)p, p.p$ etc. But in any case $p.p = p^2$ must be less than n.

ie $p^2 \leq n$

$\Rightarrow p \leq \sqrt{n}$.

Hence we will take $\lfloor \sqrt{n} \rfloor$, the largest integer less than $\sqrt{n}$ as the last number for iteration.

The algorithm is as below-

```
ALGORITHM Sieve(n)
// Implementation of Sieve of Eratosthenes.
// Input: A positive integer n ≥ 2
// Output: Array L of all prime numbers
           less than or equal to n.

for p ← 2 to n do A[p] ← p
for p ← 2 to ⌊√n⌋ do
    if A[p] ≠ 0    // p is not eliminated on previous passes.
        j ← p*p
        while j ≤ n do
```

```
        A[j] ← 0  // mark an element as
                     eliminated
        j ← j + p
// copy remaining elements of A to array
// L of the primes.
i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i + 1
return L
```

Now using this algorithm of finding prime factors, we can solve the problem of GCD.

Fundamentals of Algorithmic Problem Solving:-

While writing an algorithm for any problem, we have to follow certain steps. These steps are very helpful in analyzing and designing an effective and productive algorithm which will lead us to get the solution for the problem in question. The pictorial representation of steps to be followed is given as—

(i) Understanding the Problem : Before designing algorithm

Chetana Hegde
9448830189

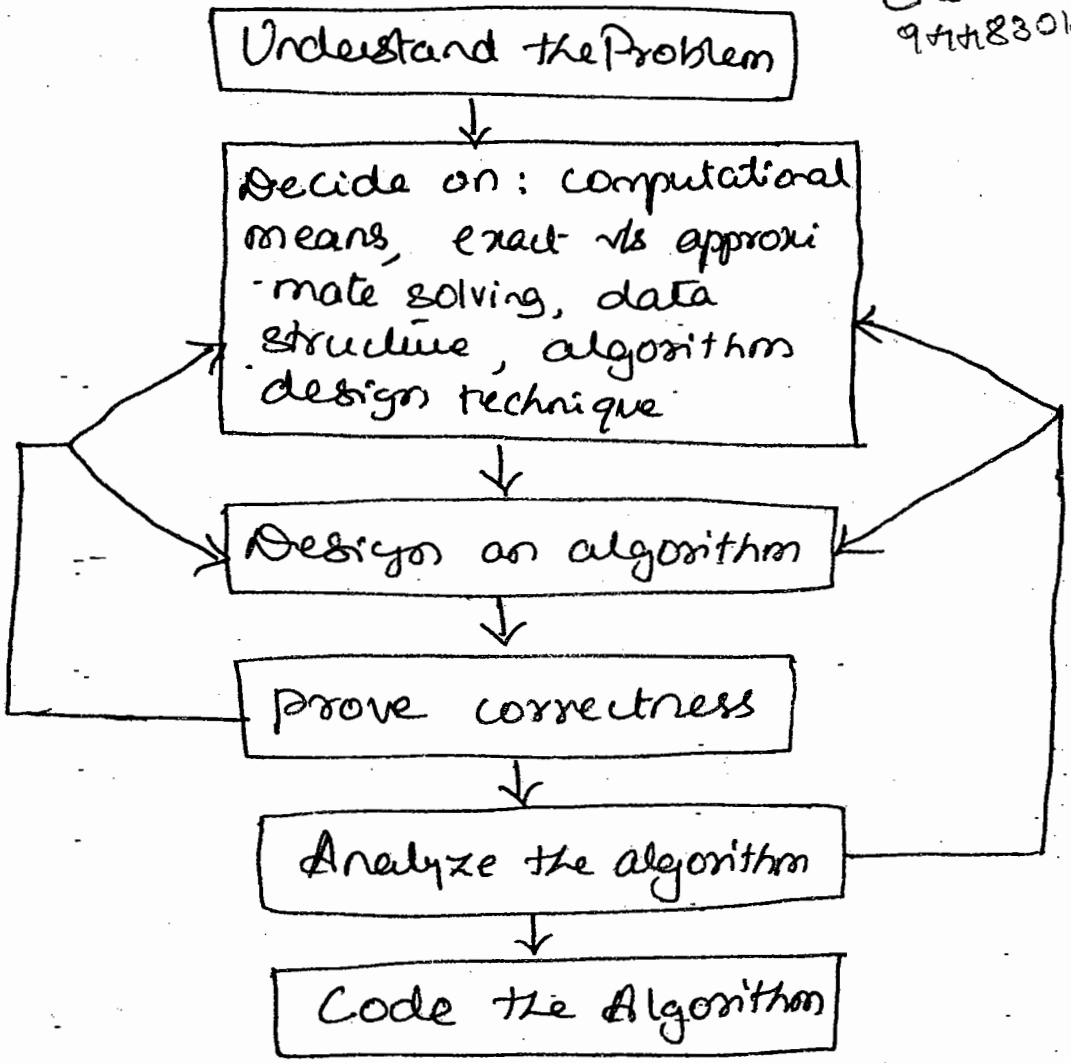


Fig: Algorithm Design & Analysis Process

1. Understanding the Problem : Before designing algorithm, one should understand the problem correctly. All the doubts must be cleared. Few problems which belongs to definite types of problems may have certain techniques. So, we have to

check whether the given problem falls under the known category or not. As we will be knowing weakness and strengths of such algorithms, we can set the appropriate one.

Any input to an algorithm specifies an instance or event of the problem. So, it is very important to set the range of inputs so that the algorithm works for all legitimate inputs.

2. Ascertaining the Capabilities of Computational Device :- After understanding the problem, one must think of the device which is going to solve the problem i.e. the device to which we are going to feed this algorithm. If the device is capable of executing the instructions one after the other, we go for sequential algorithms. As the newer devices are capable of executing instructions concurrently, parallel algorithms can also be used. If we are dealing with small problems, as a scientific excercise, we need not worry about time taken and memory required by the algorithm.

But, as a practicle tool, we do care about time and space.

3. Choosing between Exact and Approximate Problem Solving :- The algorithm which solves the problem and gives the exact solution is known as exact algorithm and the one which gives approximate results is known as approximate algorithm. Some problems like finding square roots, solving non-linear equations etc may not be solved exactly. Some other problems takes huge amount of time and space if we wish to get exact solution because of their complexity. In such cases, we will go for approximate algorithms.

4. Deciding on Appropriate Data Structures :- Algorithms use different data structures for their implementation. Some may use very simple one but some other may require quite difficult data structures. But, data structures are crucial requirement for design & analysis of algorithms.

5. Algorithm Design Techniques :- An algorithm design technique is a general approach for solving problems algorithmically that is applicable to variety of problems from different areas of computing. These techniques will provide guidance in designing algorithms for new problems and they will classify algorithms according to an underlying design idea. Thus, the algorithm design techniques serve as a natural way to categorize and study algorithms.

6. Methods of Specifying an algorithm :- After designing an algorithm it must be represented in some fashion. The two well-known methods are using English-like statements and using pseudocodes. A pseudocode is a mixture of natural language and programming language that is more effective and easier one.

7. Proving an Algorithm's Correctness :- One has to prove that the designed algorithm works for all legitimate inputs. For proving the correctness of an algorithm, one has to give the general proof i.e. one has to show that the algorithm

yields the required output for any arbitrary inputs within the range. But for disproving an algorithm, one example or one set of inputs for which algorithm fails, is enough. For proving we can go for mathematical induction.

8. Analyzing an Algorithm:- Any algorithm must be analyzed for its efficiency in time and space. Time efficiency indicates the time taken by the algorithm for its execution and space efficiency indicates the amount of memory space required for it. This study of analysis allows us to make quantitative judgements about the value of one algorithm over another.

9. Coding an Algorithm:- The designed algorithms obviously implemented as computer programs. After converting an algorithm into a program of required programming language, one has to test it. The testing consists of two phases viz. debugging and performance measurement. Debugging is a process of executing programs on sample data sets to determine whether faulty results occur and if so, to correct them. Performance measurement is the process of executing correct program on data sets and measuring time and space it takes.

Important Problem Types

There are some common problems that we come across while computing. Usually, many of the problems can be categorized into any one of these major problem types. The most important problem types are—

1. Sorting :— The problem involving the rearrangement of the items of a given list in some particular order is known as sorting problem. Sorting makes many questions about list easier to answer. ie by sorting a list, we can easily search for particular element, compare items etc. Even though, there are plenty of algorithms for sorting, none of these best suits for all possible situations. Some are very simple to follow but takes more time than few other which are complex. Some work better on randomly ordered list and some other suits for almost ordered list.

There are two properties of sorting algorithm. First one is stability. If in a particular sorting algorithm, the elements with equal values remain in the same relative order in the output as they were in the input, then such algorithm is treated

as stable. In otherwords, if list contains two equal elements at the positions i and j where $i<j$, then in the sorted list they have to be in positions i' and j' such that $i'<j'$. The second property is the amount of extra memory required by algorithm. An algorithm is said to be <u>in-place</u> if it does not require extra memory, except, possibly for a few memory units.

2. <u>Searching</u>:- The searching problem deals with searching of a given value called 'key' in the given list. There are several algorithms for searching, but as in case of sorting, no one suits best for all the cases. Few of them work sufficiently better on a sorted list. Some algorithms which works better requires more memory.

3. <u>String Processing</u>:- In rapid growing applications of computer science, processing of text takes an important role. Searching for a particular word in a text is known as string matching. There are several algorithm for string matching that we will come across in future study.

4. Graph Problems :- Graph is a set of points known as vertices some of which may be connected by a line / curve called edges. Graphs are helpful in problems like transp-ortation and communication networks, project management etc. The basic graph algorithms include traversal algorithms, shortest-path algorithms etc. The most popular problem that is computationally complex is travelling salesman problem. In this problem, a salesman has to traverse all the given cities exactly at once and come back to his hometown within minimum possible time with the note that all the cities are connected to each other. One more problem including graph is colouring problem, which asks to find out the minimum number of colours required for assigning each vertex of a graph so that no two adjacent vertices have same colour.

5. Combinatorial Problems :- The problems involving combinatorial elements like permuta-tions, combinations, sets etc are known as combinatorial problems. The difficulty with these problems is - the combinatorial objects grows extremely fast with the problem size and there are no such algorithm for giving accurate solutions within feasible amount of time for these problems.

6. Geometric Problems :- We will come accross geometric problems in the field of computer graphics, robotics etc. Very well-known problems are closest-pair problem that is used to find the closest pair of points among the given n points and the convex-hull problem, in which we have to find the smallest convex polygon covering all the given points.

7. Numerical Problems :- This section involves solving system of equations, computing definite integrals, evaluating functions etc. Many of the numerical problems can be solved approximately as these problems involves the real values which can't be expressed in computers exactly. Using the rounding-off technique, many variables will loose their original value and may result in approxi-mate solution but not the exact one.

Chetana Hegde
9th830189H

Fundamental Data Structures

Most of all the algorithms operate on data. So, organization of data plays an important role in design & analysis of algorithms. Thus, data structure which specifies the structure of data in the memory is an important aspect to be considered here. The data used by algorithms may be any basic data types like integers, reals, characters etc or they may be derived data structures like arrays, linked lists, trees etc. Let us review one by one now.

1. Linear Data Structures :- In this category, we come across arrays and linked lists. Array is a sequence of elements of same data type that uses static memory allocation. Array elements are accessed using its index. The time required for accessessing one element from the array is constant irrespective of its position in the array. Arrays of characters is treated as string and we do the operations like finding string length, comparing two strings, concatinating two strings etc. The string made up of zeros and ones are known as binary strings or bit strings.

Linked list is a sequence of zero or more elements called nodes. In a singly

linked list, each node consists of data and one link field that links this node to next node. In a doubly linked list, node consists of one data field and two link fields containing the links for predecessor and successor nodes. Linked list uses dynamic memory allocation & hence no need to reserve the memory before. But, to access an item from a linked list, we have to traverse the list till that particular node is reached. Hence, the time required for accession depends on the position of a node in a list.

Stack and queue are the data structures under this category only. For stack the insertion and deletion are made at one end called 'top' whereas for queue, the insertion made at one end called 'rear' and the deletion is made at the other end called 'front'. For selection based on priority, we can go for priority queue. For the operations like finding largest element, deleting largest element etc. can be done on priority queue. But, heap will be considered as better implementation over priority queue.

2. Graphs :- A graph $G = \langle V, E \rangle$ is defined by two finite sets viz. V of vertices and E of edges. V consists of points whereas E consists of pair of these points. For ex.

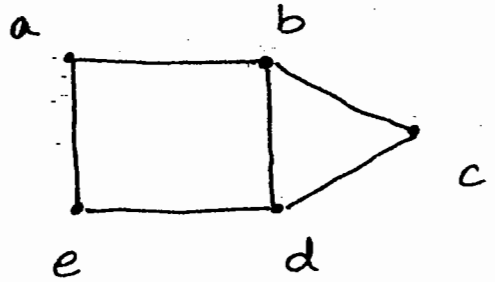


Fig (1)

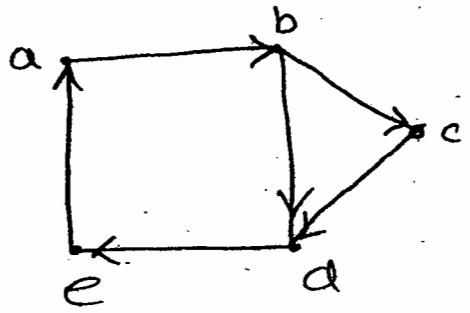


Fig (2)

For the fig(1) the set V will be-

$V = \{a, b, c, d, e\}$ and

$E = \{(a,b), (b,c), (c,d), (a,e), (e,d), (b,d)\}$

In each pair $(x, y) \in E$, the order is immaterial. i.e. (x, y) is treated same as (y, x).

But in fig(2), the order of vertices in pair is important. So,

$E = \{(a,b), (b,c), (c,d), (b,d), (d,e), (e,a)\}$

Such a graph having directed edges is known as directed graph or digraph.

In a graph, if any vertex has got edge with itself as two endpoints, then such edge is known as loop. If we consider a graph without loops then

$$0 \leq |E| \leq |V|(|V|-1)/2$$

here, $|E|$ denotes no. of edges & $|V|$ denotes no of vertices in a graph.

In a graph if every pair of vertices are connected, it is known as complete graph.

In algorithms, graphs are represented using adjacency matrix and adjacency linked lists. Adjacency matrix for a graph of n vertices is an $n \times n$ boolean matrix with one row and one column for each of the vertices, in which the element at i^{th} row and j^{th} column is 1 if there is an edge from i^{th} vertex and j^{th} vertex and is zero if there is no such edge. The adjacency matrix for graph of fig(1) is -

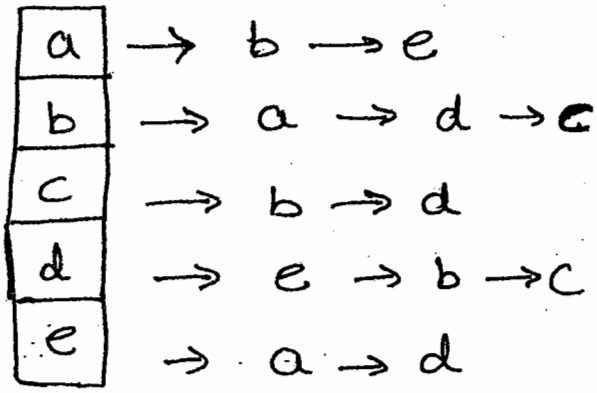
	a	b	c	d	e
a	0	1	0	0	1
b	1	0	1	1	0
c	0	1	0	1	0
d	0	1	1	0	1
e	1	0	0	1	0

Note that, the adjacency matrix for an undirected graph without loops is always symmetric. ie.

$$A[i, j] = A[j, i], \quad \forall\ 0 \le i, j \le n-1.$$

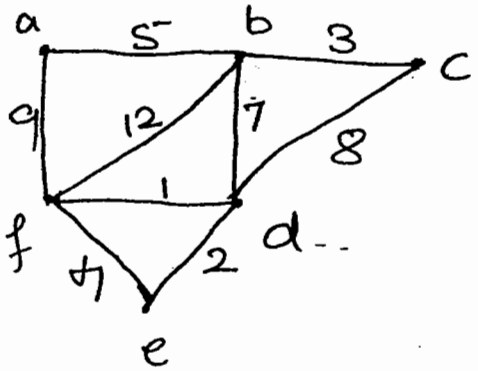
The adjacency linked list of a graph or diagraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex. Following is the adjacency linked list for the graph in fig(1).

Chetana Hegde
9448830184



A graph, in which the edges are assigned with some numbers is known as weighted graph. These numbers are called as weights or costs. These weighted graphs have got many realtime applications in the field of transportation network and communication network.

Consider an example—



The adjacency matrix for weighted graph contains weight at $(i, j)^{th}$ position if there is an edge between the vertices i and j, and contains ∞ if there is no edge. ie.

	a	b	c	d	e	f
a	∞	5	∞	∞	∞	9
b	5	∞	3	7	∞	12
c	∞	3	∞	8	∞	∞
d	∞	7	8	∞	2	1
e	∞	∞	∞	2	∞	4
f	9	12	∞	1	4	∞

The adjacency linked list for weighted graph contains weights along with the name of the vertex. This is as given below—

a	→ b, 5 → f, 9
b	→ a, 5 → f, 12 → c, 3 → d, 7
c	→ b, 3 → d, 8
d	→ f, 1 → b, 7 → c, 8
e	→ d, 2 → f, 4
f	→ a, 9 → b, 12 → d, 1 → e, 4

Path from vertex u to vertex v of a graph is defined as the sequence of adjacent vertices starting with u and ending with v. If all the edges of a path are distinct, the path is said to be simple. The total no of vertices in a path minus one is length of the path. A graph is said to be connected if for every pair of vertices u and v, there is a path from u to v. A cycle is a path which starts and ends with same vertex.

3. Trees :- A tree is a connected acyclic graph. For a tree, the number of edges are one less than the number of vertices. i.e. $|E| = |V| - 1$.

unit-2

ANALYSIS OF ALGORITHM EFFICIENCY

The analysis of algorithm means the investigation of an algorithm's efficiency with respect to two aspects viz. running time and memory space.

Chetana Hegde
9448301894

Analysis Framework :- In this section, we will concentrate on program performance. The performance of the program is measured by the efficiency of the program in terms of time required for its execution and extra memory space taken by it. But, now in new technological era, as we have got computers with huge storage space, we won't bother much about space complexity in practical problems. But, as a theoretical approach, we will study both time and space complexity by giving equal importance to them.

Following are some important aspects that we come across in Analysis Framework.

* Measuring an Input's Size :- It is obvious that all the algorithms take more time for execution if the input is large. For example, sorting a large list, multiplying matrices of big order etc takes much time. So, if we think logically, it is better put the efficiency of an algorithm with respect to time as a function of some parameter say, n denoting the

size of the input: In the problems like sorting, searching etc, the array size itself will decide the time taken for execution. In some problems, there may be more than one inputs, for example, say, matrix multiplication of order m x n. Here, even though two inputs are there, the total no of elements to be multiplied is a single quantity. So, we will be having some interrelationships between various inputs. For the algorithms involving properties of numbers, the input size is usually measured in bits as

$$b = \lfloor \log_2 n \rfloor + 1.$$

* Units for Measuring Running Time :- The algorithm's execution time can be measured in seconds, milliseconds etc. This execution time depends on -

- → Speed of a computer
- → Choice of the language to implement algorithm.
- → compiler used for generating code
- → Number of inputs.

Depending on all such aspects, it is quite difficult to find out the time required. So, we will go for one appreciable

and believable approach, where the time required for each of the executable statement of algorithm is considered. But, as we are more interested in the actual working of algorithm, we will ignore some non-important statements like input and output, and concentrate only on very important operation called, 'basic operation'. This will contribute more to the total running time and usually, it will be in the algorithm's innermost loop.

Consider an example of sorting in which the comparison of elements will be basic operation. In the example of matrix multiplication we have got two operations viz. multiplication and addition. As in many of the computers, multiplication takes more time than the addition, we may go for choosing multiplication as basic operation.

Thus, for the analysis of time, we can count the number of times the basic operation is executed for the input of 'n'.

Let c_{op} be the time required for a basic operation to execute and let $C(n)$ be the number of times, the basic operation executes. Then the time required for

algorithm, $T(n)$ can be approximately given as -

$$T(n) \simeq C_{op} \cdot C(n).$$

Note that here C_{op} and $C(n)$ both are approximate and hence $T(n)$ also will be an approximate value.

* Orders of Growth :- Suppose that we have got two algorithms for solving the same problem. If the size of the input is very small, then we can't compare the time complexities of these two algorithms and so, we can't judge, which is better one. We have to check the time taken by these algorithms as the input size increases. This is known as order of growth. It is measured in terms of some parameter 'n', which may be the size of input. The algorithms that we come across are having execution times proportional to any of the following functions.

1. Constant: If most of the instructions in a program are executed only once or very few number of times, we will say that the running time of a program is constant.

2. $\log n$: If the difference between the input n and running time increases logarithmically, as n increases, then we will say that running time is a function, $\log n$.

3. n : If the execution time is 'n' for the input n, then it indicates, algorithm is linear. Thus, if n is 10, then time is also 10 units. This kind of algorithms spend time in processing the input items.

4. $n \log n$: The time taken by algorithm for the input n is $n \log n$. This result is found in algorithms that solve the problem into number of smaller sub problems and then these subproblems are solved individually and finally combined to get final solution. Examples are quicksort, mergesort etc.

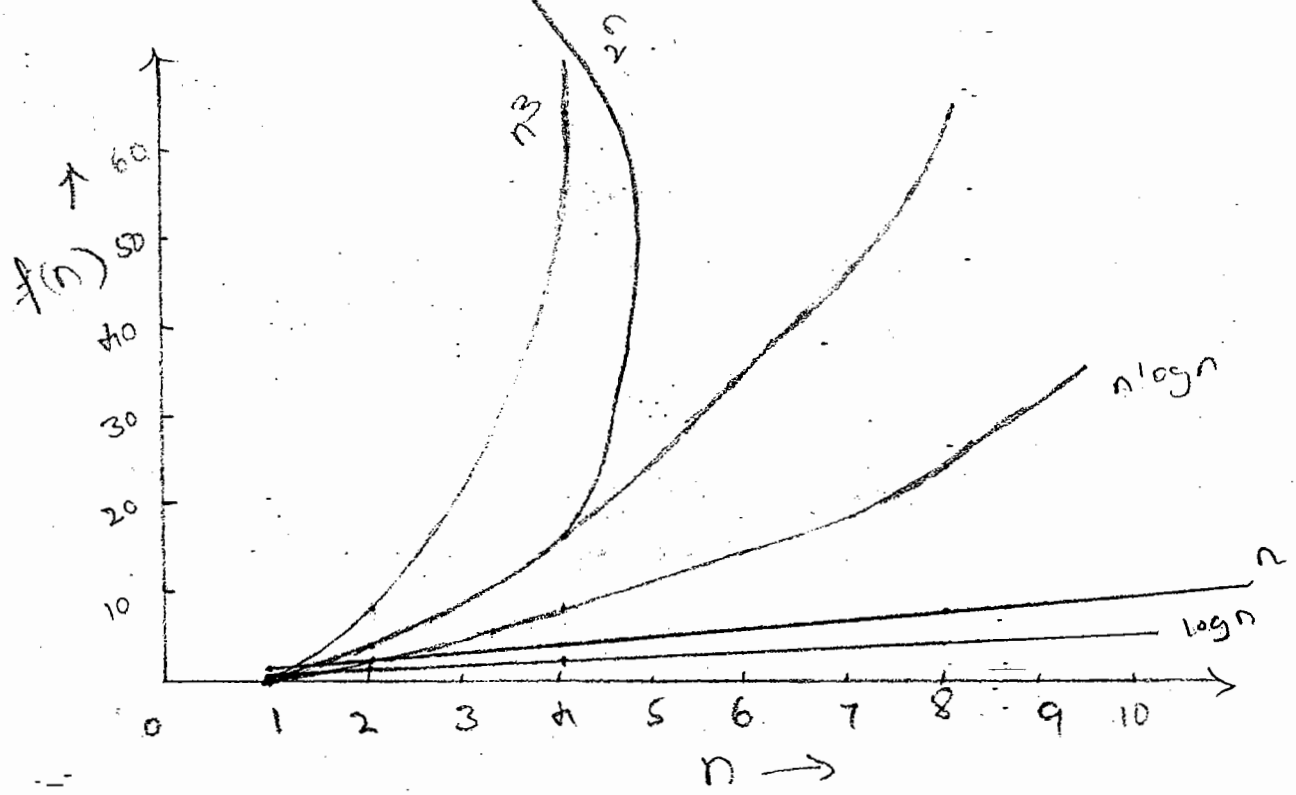
5. n^2 : Indicates, running time of algorithm is quadratic. This kind of algorithms are used when n is relatively small.

6. $n3$: Indicates that, running time is cubic and applicable for smaller problems.

7. 2^n and $n!$:- This indicates, the execution time is exponential.

Consider, the following table and graph -

n	$\log_2 n$	n	n log n	n^2	n^3	2^n	n!
1	0	1	0	1	1	2	1
2	1	2	2	4	8	4	2
4	2	4	8	16	64	16	24
8	3	8	24	64	512	256	40320
16	4	16	64	256	4096	65536	High
32	5	32	160	1024	32768	High	very high



These, are called basic efficiency classes of algorithms.

* Worst-Case, Best-Case and Average-Case Efficiencies :-

Chetana Hegde
9th830189th

The time complexity of some algorithms depends on size of input. But, in some other case this is not true. For example, in case of searching algorithm, the time depends on the position of the element to be searched. i.e. if the element for which we are searching is in first position of the list itself, then time taken by algorithm will be obviously less. But, if the element is in the last position or if the element does not exist at all in the list, then, certainly time taken will be more. Thus, for the same input size n, the time complexity ~~is~~ differs.

So, we will divide the efficiency of an the algorithm into three categories. Viz.

(a) Worst-Case efficiency : The efficiency of algorithm for the worst case input of size n for which the algorithm takes ~~largest~~ longest time to execute is called the worst case efficiency.

(b) Best Case efficiency :- The efficiency of an algorithm for the input of size n for which algorithm takes least time for execution is best case efficiency.

(c) Average Case efficiency :- Average number of steps required for execution for any random input.

Consider an example of linear search -

```
ALGORITHM linearSearch(A[0...n-1], K)
// Searches for a given value in a given
// array.
// Input : An array A[0..n-1] and search key k.
// Output : Returns the position of key if
//          it is found, Otherwise returns 0.

for i ← 0 to n-1 do
    if (A[i] == k)
        return i+1;
return 0;
```

Let us now analyze various efficiencies in all possible situations.

Worst-Case: In this case, maximum number of comparisons are required
→ if key is present in the last position, or
→ key is not present.

Thus, algorithm requires n comparisons in any of these cases. So, the worst case analysis for successful and unsuccessful search is.

$$C_{worst}(n) = n$$

This indicates the time required for this algorithm to run will not exceed $C_{worst}(n)$.

Best-Case: In best-case, time taken by the algorithm will be least for given input.
→ For successful search, the best case occurs when the element is found at first position itself. So, only one comparison is enough.
∴ $C_{best}(n) = 1$.

→ For unsuccessful search, one has to compare all the elements with key, to get the output as 'unsuccessful'.
So, there will be n comparisons.
∴ $C_{best}(n) = n$.

Average-Case: In reallife situations, we come accross worst case and best case very rarely. Usually, the elements of the list are randomly distributed. So, we will go for measuring average time. The average-case efficiency for sequential search is calculated as below—

Let p be the probability of successful search, where $0 \le p \le 1$. And, let the probability of this successful search at any i^{th} position be uniform for $0 \le i < n$.

→ Successful search :- The probability of ~~first match occuring at i^{th} position successful search at any position~~ is $p \cdot \frac{1}{n} = p/n$.

i.e., the probability of success at i^{th} position, where ~~$0 \le i < n$~~ $i = 1, 2, \ldots n$ is given by

$$\left(1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right)$$

$$= \frac{p}{n} \cdot (1 + 2 + \cdots + n)$$

$$= \frac{p}{n} \cdot \frac{n(n+1)}{2} = \frac{p(n+1)}{2}$$

→ Unsuccessful search :- If p is the probability of successful search, (1-p) is probability of unsuccessful search. In this case, the total number of comparisons required are n.

∴ probability of unsuccessful search for all n elements is given by -

$$n.(1-p).$$

Chetana Hegde 9th Aug 2018 9:47

∴ Average case is given by -

$$C_{avg} = \frac{p(n+1)}{2} + n.(1-p)$$

NOTE :

1. The average case is not computed as the average of best case and worst case.
2. If the search must be successful, then probability of successful search is 1.

So, $C_{avg} = \frac{1(n+1)}{2} + n.(1-1)$

$$= \frac{n+1}{2}.$$

And, if it is unsuccessful search, probability is zero & so,

$$C_{avg} = n.$$

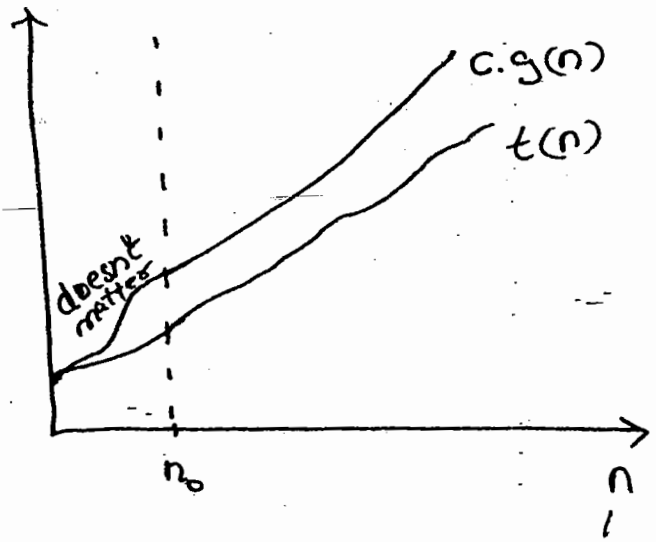
ASYMPTOTIC NOTATIONS

We know that, the order of growth of an algorithm's basic operation plays an important role in algorithm's efficiency. To compare the growth, we will use three different notations called asymptotic notation.

1. Big-O notation: A function $t(n)$ is said to be in $O(g(n))$, denoted by $t(n) \in O(g(n))$, 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 n_0 such that

$$t(n) \leq c.\, g(n), \quad \forall\, n \geq n_0.$$

This can be depicted using below given graph.



$t(n) \in O(g(n))$

for example,

① consider $100n + 5 \leq 100n + n, \quad \forall n \geq 5$

$= 101n$

$\leq 100n^2$

$\therefore 100n + 5 \in O(g(n))$

where, $g(n)$ can be taken as n^2 and,

$t(n) = 100n + 5, \quad c = 101 \quad \& \quad n_0 = 5$

i.e. $100n + 5 \leq 101 \cdot n^2 \quad \forall n \geq 5$

Note that, the same function $t(n)$ can be put in different way as—

$100n + 5 \leq 100n + 5n, \quad$ for $n \geq 1$

$= 105n$

$\leq 105n^2$

i.e. $100n + 5 \leq 105n^2 \quad$ for $n \geq 1$

Thus $t(n) \leq c \cdot g(n) \quad$ for $n > n_0$

Hence, $t(n) = 100n + 5 \in O(n^2)$.

② Express $t(n) = 10n^3 + 5$ using Big-Oh notation

Let $t(n) = 10n^3 + 5$

$\leq 15n^3 \quad$ for $n \geq 1$

i.e. $t(n) = 10n^3 + 5 \leq 15 \cdot n^3 \quad$ for $n \geq 1$

So, $t(n) \in O(n^3)$.

③ Let $t(n) = 6.2^n + n^2$. Express using O.

Consider $t(n) = 6.2^n + n^2$

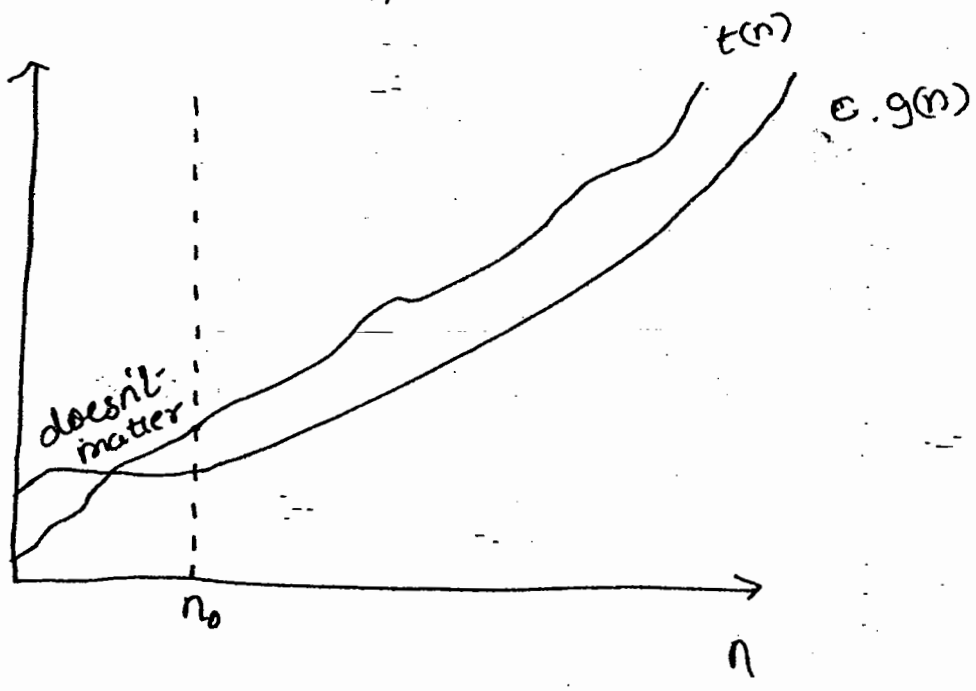
$\leq 7.2^n$ for $n \geq 4$

$\therefore t(n) \in O(2^n)$. for $n \geq 4$

2. Big-Omega (Ω) Notation :- A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, 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 n_0 such that

$$t(n) \geq c.g(n) \quad \text{for } n \geq n_0$$

It is as shown—



Chetana Hegde
9th 8301894H

For example—

① Let $t(n) = n^3$ then

$$t(n) \geq n^2 \quad \forall\, n \geq 0$$

So, we can take $g(n) = n^2$, $c = 1$ and $n_0 = 0$

Thus, ~~$t(n) \in \Theta(n^2)$~~

$$t(n) \in \Omega(n^2).$$

② Let $t(n) = 100n + 5$

then, $t(n) = 100n + 5$

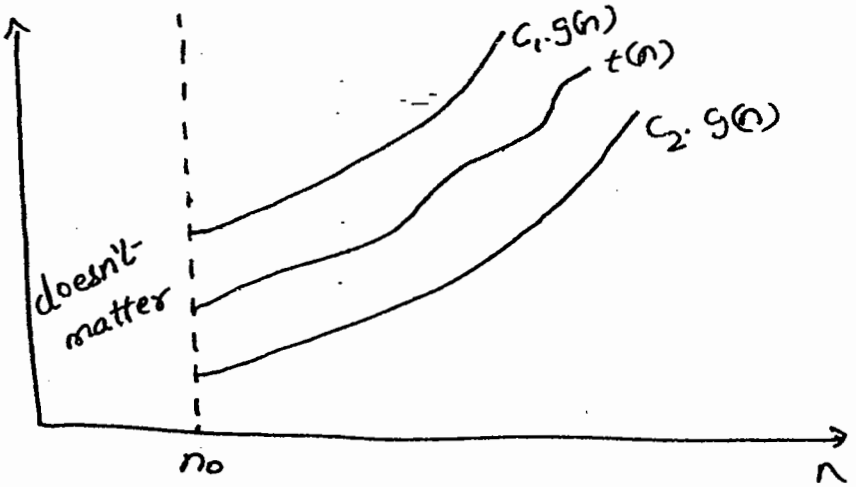
$$\geq 100n \quad \forall\, n \geq 1$$

$$\therefore t(n) \in \Omega(n).$$

3. Big-Theta (Θ) Notation :- A function $t(n)$ is said to be in $\Theta(g(n))$, denoted by $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n. ie. if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 \cdot g(n) \leq t(n) \leq c_1\, g(n), \quad \forall\, n \geq n_0.$$

ie.



For example—

① Let $t(n) = 100n + 5$.

then, $100n \leq 100n + 5 \leq 105n \quad \forall n \geq 1$

$\Rightarrow c_2 \cdot g(n) \leq t(n) \leq c_1 g(n) \quad \forall n \geq n_0$

$\therefore t(n) \in \Theta(n)$

② Let $t(n) = \frac{1}{2} n(n-1)$

Then, $t(n) = \frac{n^2}{2} - \frac{n}{2}$

We can easily say that-

$$\frac{n^2}{2} - \frac{n}{2} \leq \frac{n^2}{2} \quad \forall n \geq 0$$

Also, $\frac{n^2}{4} \leq \frac{n^2}{2} - \frac{n}{2} \quad \forall n \geq 2$

$\therefore \frac{n^2}{4} \leq t(n) \leq \frac{n^2}{2} \quad \forall n \geq 2$

$\therefore c_2 g(n^2) \leq t(n) \leq c_1 g(n) \quad \forall n \geq n_0$

Here, $c_2 = 1/4$, $g(n) = n^2$ & $c_1 = 1/2$, $n_0 = 2$

$\therefore t(n) \in \Theta(n^2)$.

Property of Asymptotic Notation

Theorem: If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then,

$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

Proof: (NOTE: If $a_1, b_1, a_2, b_2 \in R$ such that $a_1 \le b_1$ & $a_2 \le b_2$, then $a_1 + a_2 \le 2 \cdot \max\{b_1, b_2\}$.)

Let $t_1(n) \in O(g_1(n))$,

$\Rightarrow \exists\ c_1 \in R^+$ and $n_1 \in I^+$ such that

$t_1(n) \le c_1 \cdot g_1(n) \quad \forall\ n \ge n_1$.

Also, $t_2(n) \in O(g_2(n))$

$\Rightarrow \exists\ c_2 \in R^+$ and $n_2 \in I^+$ such that

$t_2(n) \le c_2 \cdot g_2(n) \quad \forall\ n \ge n_2$.

Let $c_3 = \max\{c_1, c_2\}$

& $n_3 = \max\{n_1, n_2\}$.

Then, $t_1(n) + t_2(n) \le c_1 g_1(n) + c_2 g_2(n)$, $\forall\ n \ge n_3$.

$\le c_3 g_1(n) + c_3 g_2(n)$

$= c_3 [g_1(n) + g_2(n)]$

$\le c_3 \cdot 2 \max\{g_1(n), g_2(n)\}$

$\therefore \quad t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

NOTE : This properly holds good for both Ω and Θ.

Using Limits :-

Instead of using the formal definitions of O, Ω and Θ to find the order of growth of algorithms, we will use one more method for this, we will compute the limit of the ratio of orders of growth of two algorithms. Depending on the ratio, we will take decision as below -

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0, & \Rightarrow t(n) \text{ has smaller order of growth than } g(n). \\ c, & \Rightarrow t(n) \text{ has the same order of growth as } g(n). \\ \infty, & \Rightarrow t(n) \text{ has the larger order of growth than } g(n). \end{cases}$$

NOTE :

1. In first two cases, we can say that $t(n) \in O(g(n))$.

   In last two cases, we can say that

$t(n) \in \Omega(g(n))$.

By this, we can easily say that in the second case, $t(n) \in \Theta(g(n))$.

2. For taking decisions about order of growth based on limit, we will use the following rules -

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)} \quad \text{(L' Hospital's Rule)}$$

&

$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ for large n (Stirling's formula).

Ex: Compare orders of growth of $\frac{1}{2}n(n-1)$ and n^2.

Sol: Let $t(n) = \frac{1}{2}n(n-1)$

$g(n) = n^2$.

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{\frac{1}{2}(n(n-1))}{n^2}$$

$$= \frac{1}{2} \lim_{n\to\infty} \frac{n^2-n}{n^2}$$

$$= \frac{1}{2} \lim_{n\to\infty} \left(1 - \frac{1}{n}\right)$$

$$= \frac{1}{2}, \text{ which is a positive constant.}$$

$\therefore$ $t(n)$ has got same order of growth as $g(n)$.

i.e. $t(n) \in \Theta(n^2)$.

Ex: Compare order of growth of $\log_2 n$ and $\sqrt{n}$.

Sol?: Let $t(n) = \log_2 n$ & $g(n) = \sqrt{n}$

Then, $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}}$

As the given limit is in indeminate form, $\frac{\infty}{\infty}$. So, we will use L'Hospital rule.

i.e. $$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'}$$

$$= \lim_{n\to\infty} \frac{\frac{1}{n}\cdot \log_2 e}{\frac{1}{2}(n)^{-1/2}}$$

$$= 2\log_2 e \lim_{n\to\infty} \frac{\sqrt{n}}{n}$$

$$= 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}}$$

$$= 0$$

∴ $t(n)$ has smaller order of growth than $g(n)$. We will denote it by small-o notation.

i.e. $t(n) \in o(\sqrt{n})$.

Ex: Compare orders of growth of $n!$ and 2^n.

Sol?: Let $t(n) = n!$ & $g(n) = 2^n$

Then, $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{n!}{2^n}$

$= \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}$

$= \sqrt{2\pi} \lim_{n\to\infty} \sqrt{n}\cdot\left(\frac{n}{2e}\right)^n$

$= \infty$

$\therefore$ $t(n)$ has got larger growth compared to $g(n)$.

i.e. $n! \in \Omega(2^n)$.

Mathematical Analysis

To analyze the efficiency of algorithms, we go for mathematical approach. This approach is different for recursive and non-recursive algorithms. Here we will study the methodologies used for both the kinds of algorithms.

Non-Recursive Algorithms

General Plan for Analysis:

Step 1. Based on the input size, decide the various parameters to be considered.

2. Identify the basic operation ~~to be~~ of the algorithm.

3. Compute the number of times the basic operation is executed. Check whether ~~the~~ this depends only on the size of the input. If the basic operation depends on some other conditions also, then one has to compute worst-case, best case and the average case seperately.

4. Obtain the total number of times a basic operation is executed.

5. Simplify using standard formulae and compute the order of growth & then express using asymptotic notations.

Formulae Used:

1. $\sum_{i=l}^{u} c.a_i = c. \sum_{i=l}^{u} a_i$

2. $\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$

3. $\sum_{i=l}^{u} 1 = u - l + 1 \quad , \quad l \leq u.$

d. $\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n$

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

$\approx \frac{1}{2} n^2$

e. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2$

$= \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3} \cdot n^3$

Ex.1 Consider a problem of finding the value of the largest element in a list of n numbers. The algorithm is as below—

```
ALGORITHM Max(A[0,..n-1])
// To determine maximum value in an array.
// Input: An array A[0..n-1] of real numbers.
// Output: The value of the largest element in A.
max ← A[0]
for i ← 1 to n-1 do
    if A[i] > max
        max ← A[i]
return max
```

Chetana Hegde
9448301894

Analysis:

1. As, the algorithm depends only on input size here, we will consider n.

2. There are two operations viz. comparison (A[i] > max) and assignment (max ← A[i]). As assignment happens only once, but comparison will happen for all possible n. So, hence, comparison is taken as basic operation.

3. As loop is executed n-1 times, the comparison is also executed n-1 times.

Thus, we have one comparison for (n-1) times.

So, $$C(n) = \sum_{i=1}^{n-1} 1$$

$$= (n-1) \cdot 1 + 1$$

$$= n-1$$

$$= t(n).$$

So, For all $n \geq 2$,

$\frac{n}{2} \leq n-1 \leq n$. is always true.

i.e. $\frac{1}{2} \cdot g(n) \leq t(n) \leq 1 \cdot g(n) \quad \forall n \geq 2.$

Thus, $t(n) \in \Theta(n)$.

Ex.2. Consider a problem of finding whether all the elements of a given array are distinct or not.

The algorithm for determining uniqueness is as below -

```
ALGORITHM Unique(A[0..n-1])
// To check whether all elements of array are
// distinct.
// Input: An array A[0..n-1]
// Output: Returns 'true' if elements are distinct
        -- otherwise 'false'.

for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if A[i] == A[j]
            return false
return true.
```

Analysis:

1. The parameter to be considered here is input size n.
2. As there is only one operation of comparing A[i] with A[j], we will treat this as basic operation.
3. Here, the time taken by algorithm not only depends on input size n, but also on the fact that whether any duplicate

entry is there. So, if any duplicate is there, the algorithm stops at that position. So, Thus, algorithm depends on 'n' and the position of duplicate entry. But, we can't guess this position. So, we will assume the worst case, treating that all the comparisons are made. ie even if the duplicate entry exists, it is there at last position.

Thus, each operation of comparison is done for (i+1) to (n-1) times for each value of i from 0 to n-2.

Thus,

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1)-(i+1)+1]$$

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

$$= (n-1)+(n-2)+(n-3)+\cdots+3+2+1+0$$

$$= 1+2+3+\cdots+(n-1)$$

$$= \frac{(n-1).n}{2}$$

Now, Let $t(n) = \frac{n(n-1)}{2}$ Chetana Hegde 9448301894

$\leq \frac{n^2}{2} \quad \forall\ n \geq 0$

$\leq n^2 \quad \forall\ n \geq 0$

$\therefore\ t(n) \leq c\ g(n) \quad \forall\ n \geq n_0$

where $c = 1$, $g(n) = n^2$ & $n_0 = 0$

Thus, we can say that $t(n) \in O(n^2)$.

Ex.3. Analyze the algorithm for finding multiplication of two $n \times n$ matrices A & B.

Let A and B be two $n \times n$ matrices. Let C be the the product of A & B which will again be of $n \times n$ matrix.

i.e. if $A[i, j]$ & $B[i, j]$ then,

$C[i, j] = A[i, 0].B[0, j] + \cdots + A[i, k].B[k, j] + \cdots + A[i, n-1].B[n-1, j]$

for all $i, j \leq n-1$ & $0 \leq i, j$

So, the algorithm can be as below —

ALGORITHM MatrixProduct (A[0..n-1, 0..n-1],
B[0..n-1, 0..n-1])
// Multiplication of two n×n matrices.
// Input: Two n×n matrices A & B.
// Output: One n×n matrix C = AB

```
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        C[i,j] ← 0
        for k ← 0 to n-1 do
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
return C.
```

Analysis:

1. The input size is n & is the parameter.
2. There are two basic operations - addition and multiplication. As multiplication takes more time than addition, we will take it as basic operation.
3. The basic operation depends only on n & not on any other factors. It is done for each value of k, j and i in reverse order.

Hence,

So,

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} [(n-1) - 0 + 1]$$

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

$$= \sum_{i} n \cdot \sum_{j=0}^{n-1} 1$$

$$= n \cdot \sum_{i} (n-1-0+1)$$

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

$$= n^2 \sum_{i=0}^{n-1} 1 \quad = n^3$$

Chetana Hegde
9448830189

Thus, $C(n) = n^3$.

Note that, the number of additions is same as no of multiplications. So, there are n^3 no of multiplications & n^3 additions.

If we consider C_m as the time required for multiplication & C_a as the time required for addition on a particular system, then,

$$C(n) = C_m \cdot n^3 + C_a \cdot n^3$$

$$= (C_m + C_a) n^3$$

$$\in \Theta(n^3)$$

Recursive Algorithms

General plan for analyzing the efficiency of recursive algorithms is as below-

1. Based on input size, decide the various parameters to be considered.
2. Identify the basic operation.
3. Obtain the no of times the basic operation is executed on different inputs of the same size. If it varies, then we have to go for bestcase, worst case and average case analysis.
4. Obtain a recurrence relation with an appropriate initial condition.
5. Solve the recurrence relation and obtain the order of growth & then express using asymptotic notation.

Example 1. Compute the factorial of a nonnega-tive number & findout its time complexity.

Sol: We know that,

~~$F(n) = n(n-1)!$~~

$F(n) = n * F(n-1) \quad n \geq 1$

& $F(n) = 1 \quad$ if $n = 0$

So, the algorithm can be given as-

```
ALGORITHM F(n)
// computes n! recursively.
// Input : A non-negative integer
// Output : The value of n!
if n = 0
    return 1;
else
    return F(n-1)*n;
```

Analysis:

1. As the input size is n, we will consider it as parameter.
2. Multiplication is the basic operation.
3. The total no of multiplications is the recursive function can be obtained as below-

Let, M(n) be the number of multiplications required for the function F(n).

Here, $F(n) = F(n-1) * n$.

Then, M(n) must satisfy.

$M(n) = M(n-1) + 1$, for $n > 0$.

& $M(0) = 0$

Here, $M(n-1)$ is no of multiplications required for computing $F(n-1)$ and 1 multiplication to multiply $F(n-1)$ by n.

Similarly, $M(n-1) = M(n-2) + 1$.

So, $M(n) = M(n-1) + 1$

$= \{M(n-2) + 1\} + 1$

$= M(n-3) + 3$

$= \vdots$

$= M(n-n) + n$

$= M(0) + n$

$= n.$

Thus, the time complexity of the factorial function is given by,

$C(n) = n.$

So, we can say that $C(n) \in \Theta(n)$.

Ex. 2. Consider a Tower of Hanoi problem. Here, there are three pegs (or poles) treated as source, temporary and destination. The problem is to transfer n discs on the source to destination. These n discs are arranged in a source peg such that the smaller disc is always above the larger disc. We have to transfer one disc at a time keeping in mind the

fact that the smaller disc should not be kept below larger disc at any situation.

The process of transfer includes the following steps -

1. Transfer n-1 discs from source to temporary.
2. Transfer nth disc from source to destination.
3. Transfer n-1 discs from temporary to destination.

The source, temporary and destination are denoted as A, B & C respectively.

The algorithm is as below -

```
ALGORITHM Tower (n, s A, t B, d)
// To move n discs from s to d.
// Input : n no. of discs.
// Output: d is containing all discs in order.
if (n = 1)
    write ('Move disk 1 from' s 'to' d)
    Return
Tower (n-1, s, d, t)
write ('Move disk' n 'from' s 'to' d)
Tower (n-1, t, s, d)
```

Analysis:

1. As the number of moves or the time required depends only on no of discs, we need consider, n itself as parameter.
2. The basic operation is movement of discs.
3. The recurrence relation is calculated as below.

Let M(n) be the number of moves required for moving n discs as per the algorithm.

As our algorithm, says that we have to transfer n-1 discs from source to tempary. then n^{th} disk to destination, then again n-1 discs from tempary to destination.

So, $M(n) = M(n-1) + 1 + M(n-1)$

$= 2M(n-1) + 1$

But, $M(n-1) = 2M(n-2) + 1$

So, Continuing in this way-

$M(n) = 2M(n-1) + 1$

$= 2[2 \cdot M(n-2) + 1] + 1$

$= 2^2 M(n-2) + 2 + 1$

$$= 2^2[2M(n-3)+1] + 2 + 1$$

$$= 2^3 M(n-3) + 2^2 + 2 + 1$$

$$= \vdots$$

$$= 2^n M(n-n) + 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2 + 1$$

$$= 2^n \cdot M(0) + 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2^1 + 2^0$$

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

$$= \frac{a1(2^n - 1)}{2-1}$$ (by geometric progression, $\sum_{r=0}^{n} a^r = \frac{a(r^n-1)}{r-1}$)

$$= 2^n - 1$$

Thus, the number of movements required for transferring n discs from source to destination is - $2^n - 1$.

So, $C(n) = 2^n - 1$

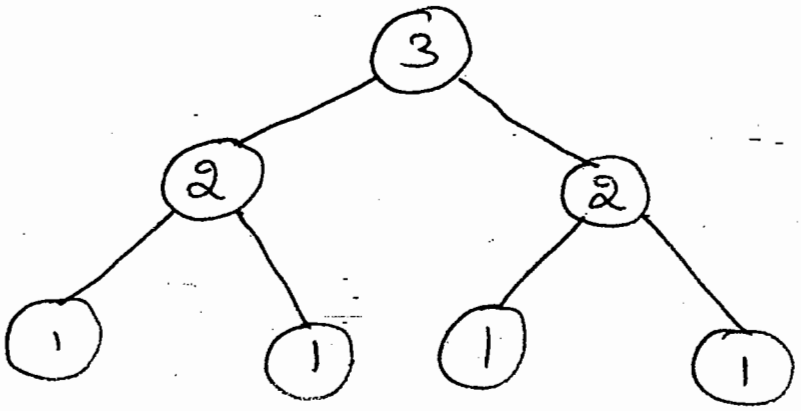
$$\in \Theta(2^n)$$

Thus, the time complexity of Tower of Hanoi algorithm is $2^n - 1$.

NOTE - that, we can analyze about no of calls using binary tree technique also.

i.e. taking n as a node, we can construct binary tree as below -

if n = 3,



The labels of nodes represent the value of n which is passed as a parameter.

The no of nodes represent the no of calls made. So, we can find total no of nodes by finding no of nodes at each level.

So, no of nodes at level 0 = $2^0 = 1$

" " " " 1 = $2^1 = 2$

" " " " 2 = $2^2 = 4$

∴ Total nodes = 7 = total recursive calls for n = 3.

But, if we continue this way,

total nodes = $2^0 + 2^1 + \cdots + 2^{n-1}$

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

Ex 3. Write an algorithm and analyze it to find the number of binary digits in a binary representation of a positive decimal integer.

ALGORITHM Bin(n)
// Input: A positive decimal integer n
// Output: The no of binary digits in n's binary
// representation.

if n = 1
    return 1
else
    return Bin(⌊n/2⌋) + 1

Chetana Hegde
9448830184

Analysis:

1. The time required depends only on input size n. So that will be the parameter.
2. The basic operation is addition.
3. Let A(n) be the no of additions required for the input of n.

Then, $A(n) = A(\lfloor \frac{n}{2} \rfloor) + 1$.

Here, $A(\lfloor \frac{n}{2} \rfloor)$ is ~~time~~ no of additions for computing Bin(⌊n/2⌋) and 1 is required for adding it to 1.

Now, if we take n as any positive integer, it is quite difficult to solve recursive relation. So, as per standard

approach, we will take n ~~as even integer~~

i.e. $n = 2^k$

$\therefore A(n) = A(\lfloor n/2 \rfloor) + 1$

$= A(n/2) + 1$ ($\because$ if n is integral power of 2, then $\lfloor n/2 \rfloor = n/2$)

$= A(2^k/2) + 1$

$= A(2^{k-1}) + 1$, for $k > 0$

$= \{A(2^{k-2}) + 1\} + 1$

$= A(2^{k-2}) + 2$

$= \vdots$

$= A(2^{k-k}) + k$

$= A(1) + k$ ($\because A(1) = 0$, as when $n=1$, there are no additions)

$= k$

Thus, $A(2^k) = k$

As $n = 2^k$, we get $k = \log_2 n$

$\therefore A(n) = \log_2 n$

$\in \Theta(\log_2 n)$

Fibonacci Numbers

The following sequence is known as fibonacci sequence- 0, 1, 1, 2, 3, 5, 8, 13, ...

The formula is given by-

$$F(n) = \begin{cases} F(n-1) + F(n-2) & \text{for } n > 1 \\ 0, & n = 0 \\ 1, & n = 1 \end{cases}$$

Here, we will discuss the explicit formula for finding the n^{th} Fibonacci number-

Usually, recurrence relations are solved by using backward substitution or by forward substitution.

For example, the relation like

$$x(n) = 2x(n-1) + 1, \quad n > 1$$

$$x(1) = 1$$

can be solved as-

$x(1) = 1$

$x(2) = 2.x(1) + 1 = 3$

$x(3) = 2.x(2) + 1 = 7$ & so on.

This type of solving is known as forward substitution.

Again, $x(n) = x(n-1) + n$

$= x(n-2) + (n-1) + n$

⋮

This is known as backward substitution.

Some of the recurrence relations can not be solved by either of these methods. These will be of the form-

$$a.x(n) + b.x(n-1) + c.x(n-2) = 0 \quad \text{— (1)}$$

Here, a, b & c are real numbers with $a \neq 0$. Such a relation is known as linear, second-order recurrence with constant coefficients. This is a homogeneous equation. This equation will be having infinitely many solutions if $b \neq 0$ & $c \neq 0$.

For finding the general solution of the above relation, we will consider a characteristic equation in the form-

$$a.r^2 + b.r + c = 0 \quad \text{— (2)}$$

Depending on the roots, say r_1 and r_2, we will ~~decide~~ frame the general solution for equation (1) as below-

Case(i) If r_1 and r_2 are real and distinct, the general solution for (1) is obtained by -

$$x(n) = \alpha r_1^n + \beta r_2^n, \quad \alpha, \beta \in R$$

Case(ii) If r_1 and r_2 are equal then general solution of ① is

$$x(n) = \alpha r^n + \beta . n . r^n$$

where $r = r_1 = r_2$ & $\alpha, \beta \in R$.

Case(iii) If r_1 and r_2 are distinct complex numbers say, $r_1 = u + iv$

$$r_2 = u - iv$$

Then, general solution of ① is –

$$x(n) = r^n [\alpha \cos n\theta + \beta \sin n\theta]$$

where $r = \sqrt{u^2 + v^2}$, $\theta = \tan^{-1} v/u$

& $\alpha, \beta \in R$.

Based on all these concepts, we will find the explicit formula for n^{th} Fibonacci number.

As $F(n) = F(n-1) + F(n-2)$, $n > 1$

we have,

$$F(n) - F(n-1) - F(n-2) = 0 \quad \text{—— ①}$$

Its characteristic equation will be –

$$r^2 - r - 1 = 0$$

$$\therefore r_{1,2} = \frac{-(-1) \pm \sqrt{(-1)^2 - 4.1.(-1)}}{2.1}$$

$$= \frac{1 \pm \sqrt{5}}{2}$$

As roots are real and distinct, the general solution of ① is given by -

$$F(n) = \alpha \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n + \beta \left(\frac{1-\sqrt{5}}{2}\right)^n \quad -②$$

But, we know that

$$F(0) = 0 \quad \& \quad F(1) = 1.$$

∴ we get

$$0 = \alpha \left(\frac{1+\sqrt{5}}{2}\right)^0 + \beta \left(\frac{1-\sqrt{5}}{2}\right)^0$$

$$0 = \alpha + \beta$$

And,

$$1 = \alpha \left(\frac{1+\sqrt{5}}{2}\right)^1 + \beta \left(\frac{1-\sqrt{5}}{2}\right)^1$$

So, we get simultaneous equations in two unknowns -

$$\alpha + \beta = 0$$

$$\alpha\left(\frac{1+\sqrt{5}}{2}\right) + \beta\left(\frac{1-\sqrt{5}}{2}\right) = 1$$

Solving - we will get -

$$\alpha = 1/\sqrt{5} \quad \text{and} \quad \beta = -1/\sqrt{5}$$

Substituting in ② —

$$F(n) = \frac{1}{\sqrt{5}} \left\{ \left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n \right\}$$

i.e. $F(n) = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n)$ —— ③

Here, $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$

& $\hat{\phi} = -1/\phi \approx -0.61803$

Thus ③ gives general formula for n^{th} fibonacci number.

By looking at ③ we can easily say that $F(n) \in \Theta(\phi^n)$.

Algorithm for fibonacci numbers:

We discussed the explicit formula for n^{th} fibonacci number & then found complexity. Let us do the same now, through usual approach -

```
ALGORITHM F(n)
// computes nth fibonacci no
// Input: A nonnegative integer n
// Output: The nth fibonacci no
if n ≤ 1
    return n
else
    return F(n-1) + F(n-2).
```

Analysis:

1. The parameter is input size n
2. The basic operation is addition.
3. Let A(n) be the no of additions required for computing nth fibonacci no.

Then, $A(n) = A(n-1) + A(n-2) + 1$, $n > 1$

with $A(0) = 0$ & $A(1) = 0$

Now,

$$A(n) - A(n-1) - A(n-2) = 1$$

This equation is not homogeneous.

So, let us rewrite it as-

$$[A(n)+1] - [A(n-1)+1] - [A(n-2)+1] = 0$$

Let $B(n) = A(n) + 1$

Then, equation becomes,

$$B(n) - B(n-1) - B(n-2) = 0$$

with $B(0) = 1$ & $B(1) = 1$

This equation is same as the one we used in finding explicit for $F(n)$. But there we had $F(0) = 0$ & $F(1) = 1$

But, here we have $B(0) = 1$

& $B(1) = 1$

Chelana Hegde
9448830189

So, it can easily be seen that $B(n)$ is one place ahead of $F(n)$.

i.e. $B(n) = F(n+1)$

But, $A(n) = B(n) - 1$

$\therefore A(n) = F(n+1) - 1$

$$= \frac{1}{\sqrt{5}} \left(\phi^{n+1} - \hat{\phi}^{n+1} \right) - 1$$

Hence, $A(n) \in \Theta(\phi^n)$

Chetana Hegde
9448301894