Abstracted from: Introduction to Design & Analysis of Algorithms, 2nd Ed, Anany Levitin, Pearson

UNIT - I

Subject: Design Methods & Analysis of Algorithms

Introduction

The name algorithm came from a mathematician named Muhammad ibn musa al-khowarizmi (lived in 850 A.D). He was a Persian mathematician, astronomer & geographer (780 -

- why do we need to study algorithms?

  - Practical reasons
  - Theoretical reasons

- computer programs could not exist without algorithms. And with computer applications becoming indispensable in almost all aspects of our professional and personal lives, studying algorithms becomes a necessity for more and more people.
- Another reason for studying algorithms is their usefulness in developing analytical skills. After all, algorithms can be seen as special kinds of solutions to problems - not answers but precisely defined procedures for getting answers. Consequently, specific algorithm design techniques can be interpreted as problem-solving strategies that can be useful regardless of whether a computer is involved.
- There are certain limits to problems that can be solved with an algorithm. We will not find, for example, an algorithm for living a happy life & becoming rich and famous.
- is it true that in order to obtain high speed computation, we need a very high speed computer?

Answer: it is not entirely true. A good algorithm implemented on a slow computer may perform much better than a bad algorithm implemented on a fast computer.

- what is an 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 a finite amount of time.

problem
↓
algorithm
↓
input → "computer" → output

notion of algorithm

- every algorithm must satisfy the following characteristics

1. Finiteness. terminate in finite no. of
2. Definiteness. (no ambiguity
3. Input. 0 or more
4. output. 1 or more
5. effectiveness (each operation should be sufficiently basic

- Important points in the design of algorithms in common are

- the nonambiguity requirement for each step of an algorithm cannot be compromised.
- the range of inputs for which an algorithm works has to be specified carefully.
- the same algorithm can be represented in several different ways
- several algorithms for solving the same problem may exist.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

Example
→ compute gcd(m,n), where m,n are two non negative, not-both-zero integers

gcd: the largest integer that divides both m & n evenly, i.e., with a remainder of zero.

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

gcd(60,24) can be computed as follows:

IDEA 1 method

$$gcd(60,24) = gcd(24,12) = gcd(12,0) = 12.$$

algorithm:

ALGORITHM Euclid(m,n)
// computes gcd(m,n) by Euclid's algorithm.
// Input: Two nonnegative, not-both-zero integers m and n.
// Output: Greatest common divisor of m and n.
while (n ≠ 0) do
  r ← m mod n
  m ← n
  n ← r
return m.

How do we know that Euclid's algorithm eventually comes to a stop?

The second number of the pair gets smaller with each iteration and it cannot become negative. Indeed, the new value of n on the next iteration is m mod n, which is always smaller than n. Hence, the value of the second number in the pair eventually becomes 0, and the algorithm stops.

IDEA 2 & Method 2

It is based on the definition of the greatest common divisor of m and n as the largest integer that divides both numbers evenly. Obviously, such a common divisor cannot be greater than the smaller of these numbers, which we will denote by $k = \min\{m, n\}$. So we can start by checking whether k divides both m and n; if it does, k is the answer; if it does not, we simply decrease k by 1 and try again. (How do we know that the process will eventually stop?) For example, for numbers 60 and 24, the algorithm will try first 24, then 23, and so on until it reaches 12, where it stops.

Consecutive integer checking algorithm for computing gcd(m,n)

Step 1: Assign the value of $\min\{m, n\}$ to k.

Step 2: Divide m by k. If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.

Step 3: Divide n by k. If the remainder of this division is 0, return the value of k as the answer and stop; otherwise, proceed to Step 4.

Step 4: Decrease the value of k by 1. Go to Step 2.

k ← min{m,n}
If (m mod k = 0)
    If (n mod k = 0)
        return k
else k ← k − 1.

k ← min{60, 24}
If (60 mod 24) ≠ 0

```
k ← min (m,n)
do {
  If ((m mod k) == 0)
    If ((n mod k) == 0)
      return k
  else
    k ← k - 1
}
while (k != 0)
```

```
k ← min (60,24)
do{ k = 24
  If ((60 mod 24) != 0) X
  else
    k ← 24 - 1
}while (23 != 0)
do { (If (60 mod 23) != 0) X
  else k ← 23 - 1
  :
do { If ((60 mod 12) == 0)
    If ((24 mod 12) == 0)
      return 12 ✓ Stop
}while (k != 0).
```

The above algorithm does not work correctly when one of its input numbers (either m & n) is zero. This example illustrates why it is so important to specify the range of an algorithm's inputs explicitly and carefully.

IDEA & method 3

middle-school procedure for computing gcd(m,n).

step1. Find the prime factors of m.

step2. Find the prime factors of n.

step3. Identify all the common factors in the two prime expansions found in step1 and step2. (If p is a common factor occurring p_m and p_n times in m and n, respectively it should be repeated min $\{p_m, p_n\}$ times.).

step4. Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

for the numbers 60 and 24, we get

$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$

$$24 = 2 \cdot 2 \cdot 2 \cdot 3$$

$$\text{gcd}(60, 24) = 2 \cdot 2 \cdot 3 = 12.$$

This procedure is much more complex and slower than euclid's algorithm.

A simple algorithm for generating consecutive primes not exceeding any given integer n. (also called as Sieve of Eratosthenes).

```
Algorithm Sieve(n)
// Implements the Sieve of Eratosthenes.
// Input: An integer n ≥ 2.
// Output: Array L of all prime numbers less than & equal to n.
    for P ← 2 to n do A[P] ← P.
    for P ← 2 to ⌊√n⌋ do
        if A[P] ≠ 0 // P hasn't been eliminated on previous passes
            j ← P*P.
            while j ≤ n do
                A[j] ← 0    // mark element as eliminated
                j ← j+P
    // copy the 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.
```

eliminates list of all multiples of 2 i.e 2,4,6; multiples of 3, multiples of 4 & multiples of 5

The algorithm starts by initializing a list of prime candidates with consecutive integers from 2 to n. In the first iteration of the algorithm, it eliminates from the list all multiples of 2, i.e., 4, 6, and so on. Then it moves to the next item on the list, which is 3, and eliminates its multiples. No pass for number 4 is needed: since 4 itself and all its multiples are also multiples of 2, they were already eliminated on a previous pass. (By similar reasoning, we need not consider multiples of any eliminated number). The next remaining number on the list, which is used on the third pass, is 5. The algorithm continues in this fashion until no more numbers can be eliminated

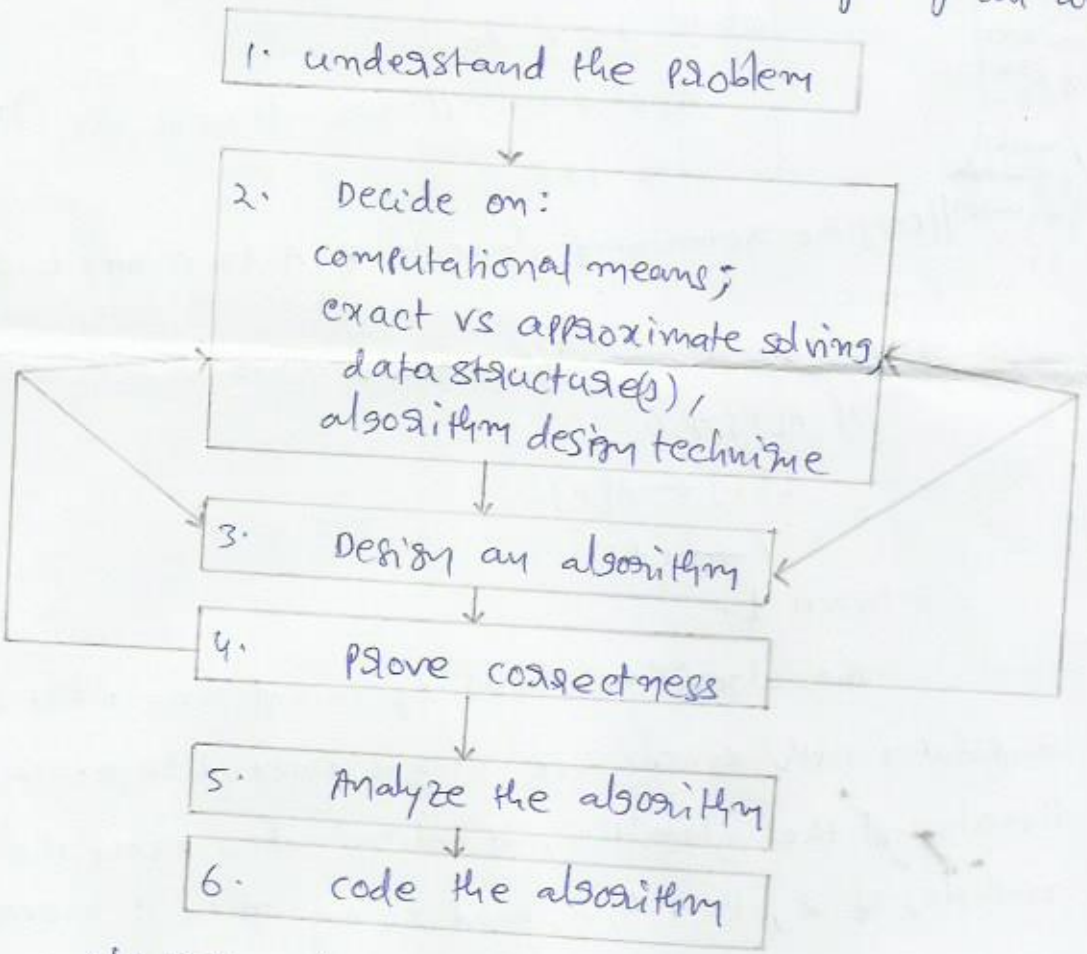
from the list. The remaining integers of the list are the primes needed.

As an example, consider the application of the algorithm to finding the list of primes not exceeding $n = 25$:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	3		5		7				11		13				17		19				23		25
2	3		5		7				11		13				17		19				23		

Fundamentals of Algorithmic problem solving

sequence of steps in designing and analyzing an algorithm are



Algorithm design and analysis process.

1. Understand the Problem: before designing an algorithm, we first need to understand completely the problem given.

An input to an algorithm specifies an instance of the problem the algorithm solves. It is very important to specify exactly the range of instances the algorithm needs to handle. If we fail to do this, our algorithm may work correctly for a majority

of inputs but crash on some boundary value. Remember that a correct algorithm is not one that works most of the time, but one that works correctly for all legitimate inputs.

2. Ascertaining the capabilities of a computational device:

Once we completely understand a problem, we need to ascertain the capabilities of the computational device the algorithm is intended for. Capabilities like

(i) RAM

(ii) sequential algorithms vs parallel

Choosing between exact and approximate problem solving

The next principal decision is to choose between solving the problem exactly or solving it approximately.

exact algorithm — approximation algorithm

Eg: extracting square roots.

Deciding on appropriate data structures

Algorithms + data structures = programs.

Algorithm design techniques

How do we design an algorithm to solve a given problem? Ans: using algorithm design technique

What is an algorithm design technique?

An algorithm design technique (or "strategy or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Methods of specifying an algorithm

Once we have designed an algorithm, we need to specify it in some fashion.

A pseudocode is a mixture of a natural language and programming language like constructs. some of the pseudocode conventions are.

1. comments - //, /, /* ... */
2. compound statements - { and } braces.
3. identifier - letter
4. compound datatypes can be formed with structures.
5. Assignment statement - = (Assignment of values to variable)
6. boolean values - true & false.
   logical operators - and, or and not.
   relational operators - <, ≤, =, ≠, ≥, and >.
7. arrays, multidimensional arrays [ ]
8. loops: while

```
while (condition) do
{
    statement 1
    :
    statement n
}
```

for

```
for var = value1 to value2 step step do
{
    stmt 1
    :
    stmt n
}
```

do-while

```
do
{
    stmt 1
    :
    stmt n
} while(condition)
```

9. conditional stmts:

```
if (condn) then (statement)
if (condn) then (statement 1) else stmt 2.
switch(condn) {
        case 1: stmt 1; stmt 2; break;
        case 2: stmt 1; stmt 2; break;
        :
        default; stmt 1;
}
```

flowchart is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

Proving an Algorithm's correctness

Once an algorithm has been specified, you have to prove its correctness. i.e., you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.

A common technique to prove correctness is mathematical induction.

In order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.

Analyzing the Algorithm

After correctness, next is efficiency. There are two kinds of algorithm efficiency: time efficiency and space efficiency.

Time efficiency indicates how fast the algorithm runs; space efficiency indicates how much extra memory the algorithm needs.

other characteristics of algorithm are: simplicity, generality.

Coding an Algorithm

a good algorithm is a result of repeated effort and rework.

implementing an algorithm correctly is necessary. A better algorithm can make a difference in running time by orders of magnitude.

A coosking prossam provides an additional opposturity in allocsing an empisical analysis of the underlying algosithm. The analysis is based on several inputs and then analyzing the results obtained.

issues of algosithmic problem solving are

a) optimality - It is not about the efficiency of an algosithm but about the complexity of the problem it solves.

b) cohether or not every problem can be solved by an algosithm.

Chapter 2: Fundamentals of the Analysis of Algosithm efficiency

Analysis of Algosithms mean an investigation of an algosithm's efficiency w.r.t. two resources: running time and memory space.

Analysis frame cook: These are two kinds of efficiency: time efficiency & space efficiency.

Time efficiency indicates how fast an algosithm in question runs; space efficiency deals with the extra space the algosithm requires.

measuring an input's size: It is logical to investigate an algosithm's efficiency as a function of some parameter n indicating the algosithm's input size.

The choice of an appropriate size metric can be influenced by operations of the algosithm in question.

Ex: how should we measure an input's size for a spell-checking algosithm?

If the algosithm examines individual characters of its input, then we should measure the size by the number of characters it cooks by processing words, we should count their number in the input.

Units for measuring running time

Standard unit of time measurement – a second, a milli second, and so on.

identify the most important operation of the algorithm, called the basic operation, the operation contributing the most to the total running time, and compute the no. of times the basic operation is executed.

As a rule, it is not difficult to identify the basic operation of an algorithm: it is usually the most time-consuming operation in the algorithm's innermost loop.

Ex: most sorting algorithms work by comparing elements of list being sorted with each other; for such algorithms, the basic operation is a key comparison.

Thus, the established framework for the analysis of an algorithm's time efficiency suggests measuring it by counting the number of times the algorithm's basic operation needs to be executed ~~for this algorithm~~ on inputs of size n.

Let c_{op} be the execution time of an algorithm's basic operation on a particular computer, and let $C(n)$ be the no. of times this operation needs to be executed for this algorithm. Then we can estimate the running time $T(n)$ of a program implementing this algorithm on that computer by the formula

$$T(n) \approx c_{op} \cdot C(n).$$

How much faster would this algorithm run on a m/c that is ten times faster than the one we have?"

Ans: ten times.

assume that, $C(n) = \frac{1}{2} n(n-1)$.

How much ~~time~~ longer will the algorithm run if we double

its input size?

Ans $C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$

and therefore

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)}$$

$$\approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4 \text{ times longer.}$$

Orders of Growth

A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones. For large values of n, it is the function's order of growth that counts.

n	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n ↓ exponential growth functions ↘	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

values of several functions important for analysis of algorithms.

* The function growing the slowest among these is the logarithmic function.

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

ALGORITHM sequentialSearch(A[0...n-1], K)
// Searches for a given value in a given array by sequential search
// Input: An array A[0,...,n-1] and a search key K.
// output: The index of the first element of A that matches K or
-1 if there are no matching elements.

```
i ← 0
while (i < n and A[i] ≠ K do
    i ← i+1
if i < n return i
else return -1
```

Worst-case efficiency

The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or) of size n for which the algorithm runs the longest among all possible inputs of that size.

- for any instance of size n, the running time will not exceed $C_{worst}(n)$, its running time on the worst-case inputs.

Ex: for sequential search, $C_{worst}(n) = n$.

Best-case efficiency

The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input (of inputs) of size n for which the algorithm runs ~~the fastest among all~~ possible inputs of that size.

For ex: for sequential search, best-case inputs are lists of size n with their first elements equal to a search key; i.e $C_{best}(n) = 1$

Average-case efficiency. To analyze the algorithm's average-case efficiency, we must make some assumptions about possible inputs of size n. They are (Ex: linear search)

a) the probability of a successful search is equal to p ($0 \le p \le 1$)

b) the probability of the first match occurring in the ith position of the list is the same for every i; and the no. of comparisons made by the algorithm in such a case is

under these assumptions, we can find the avg no. of key comparisons $C_{avg}(n)$ as follows.

In the case of successful search, the probability of the first match occurring in the ith position of the list is p/n for every i,

and the no. of comparisons made by the algorithm in such a situation is obviously i.

In the case of an unsuccessful search, the no. of comparisons is n with the probability of such a search being (1-p). Therefore,

$$C_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right] + n(1-p)$$

$$= \frac{p}{n}[1+2+\cdots+i+\cdots+n] + n(1-p)$$

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

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

This general formula yields some quite reasonable answers.

ex; if p=1 (i.e., the search must be successful), the avg. no. of key comparisons made by sequential search is (n+1)/2; i.e., the algorithm will inspect, on average, about half of the list's elements if p=0 (i.e., the search must be unsuccessful), the avg. no. of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

investigation of avg-case efficiency is considerably more difficult than the investigation of other two efficiencies.

The direct approach for doing this involves dividing all instances of size n into several classes so that for each instance of the class the no. of times the algorithm's basic operation is executed is the same.

another type of efficiency is amortized efficiency. It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure. In some situations a single operation can be expensive. but the total time for an entire sequence of n such operations is always significantly better than the worst-case efficiency of that single operation multiplied by n.

Asymptotic notations

We use 3 asymptotic notations to measure the orders of growth. They are O (big-oh), Ω (big omega), and Θ (big theta)

algorithm's running time ← $t(n)$ and $g(n)$ → some simple function to compare the count with.

let $t(n)$ and $g(n)$ can be any nonnegative functions defined on the set of natural numbers.

O-notation Definition 1

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, 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

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

(8)

Definition 2

let $f(n)$ & $g(n)$ be two asymptotic positive real valued functions.

$f(n) = O(g(n))$ iff there exists two positive constants $c > 0$ and $n_0 \geq 1$ such that $|f(n)| \leq c\,|g(n)|$ for all n, $n \geq n_0$.

Informally, It gives an upperbound on the growth rate of a function.

Prove that $100n + 5 \in O(n^2)$

Soln:

$100n + 5 \leq 100n + n$ ($\forall n \geq 5$)

$\leq 101n$

$\leq 101n^2$.

$c = 101$, $n_0 = 5$

n	$100n + 5$	$101n^2$
1	105	101
2	205	404

$n_0 = 2$, $c = 101$

$100n + n = 101n$

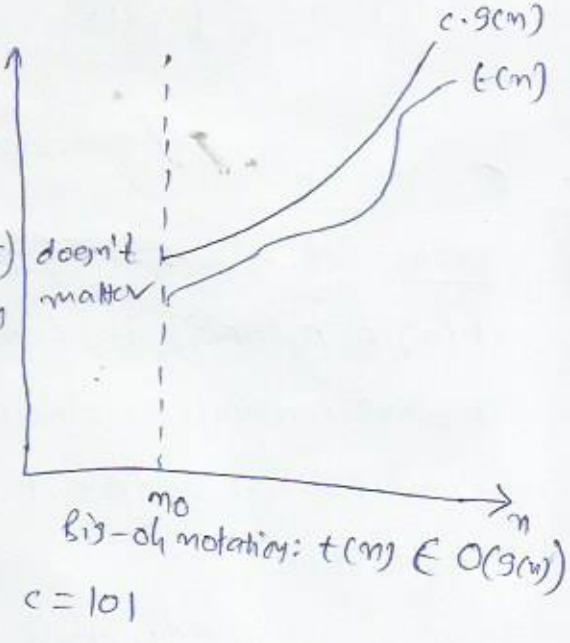
$100 \geq 101$

$n = 2$: $205 > 202$

$n = 3$: $305 > 303$

$n = 4$: $405 > 404$

$n = 5$: $505 = 505$



Examples 1. $10n^2+4n+2 = O(n^2)$. $c=11$ $n_0=5$

2. $1000n^2+100n-6 = O(n^2)$. $c=1001$ $n_0=100$

3. $10n^2+9 = O(n)$. ✗

4. WKT $10n^2+4n+2 = O(n^4)$. $c=10$ $n_0=2$.

5. P.T $n^2 = O(2^n)$ $c=2$ $n_0=1$

Ω - notation

Defn 1: A function $f(n)$ is said to be in $\Omega(g(n))$, denoted $f(n) \in \Omega(g(n))$, if $f(n)$ is bounded below by some positive 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

$$f(n) \geq c \cdot g(n) \quad \text{for all } n \geq n_0.$$

(09)

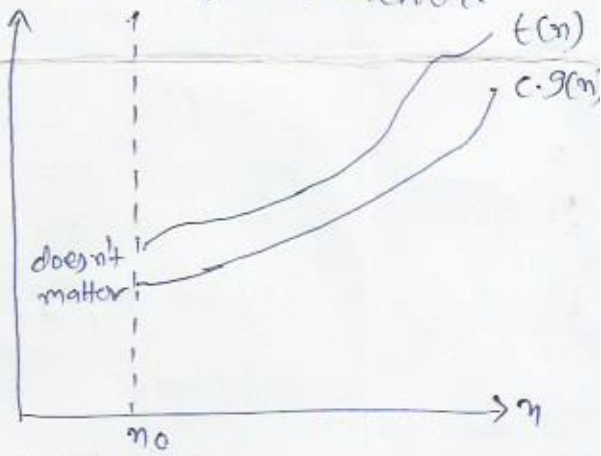
Defn 2: $f(n) = \Omega(g(n))$ iff there exists positive constants c and n_0 such that for all n, $n \geq n_0$ $|f(n)| \geq c \cdot |g(n)|$.

It gives a lower bound on to growth rate of a function.

Prove that $n^3 \in \Omega(n^2)$.

as $n^3 \geq n^2$ always for all $n \geq 0$,

i.e., we can select $c=1$ and $n_0=0$.



Big-omega notation : $f(n) \in \Omega(g(n))$

Examples 1. P.T $6 * 2^n + n^2 = \Omega(2^n)$ $c=1$ $n_0=1$

2. P.T $2^n = \Omega(2^{n+1})$ $c=1/4$ $n_0=1$

3. P.T $n^3 = \Omega(n^3+2)$ $c=1/2$ $n_0=2$

Θ - notation

Defn 1: A function $f(n)$ is said to be in $\Theta(g(n))$, denoted $f(n) \in \Theta(g(n))$, if $f(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constrant c_1 and c_2 and some non negative integer n_0 such that

$$c_2(g(n)) \leq f(n) \leq c_1 \cdot (g(n)) \quad \text{for all } n \geq n_0.$$

(09)

Defn2: $f(n) = \Theta(g(n))$ iff there exists positive constants c_1, c_2 and n_0 such that for all $n \geq n_0$,

$$c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)|$$

Prove that $\frac{1}{2} n(n-1) \in \Theta(n^2)$

Soln: First, we prove the right inequality (the upper bound):

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

$$\leq \frac{1}{2} n^2 \text{ for all } n \geq 0.$$

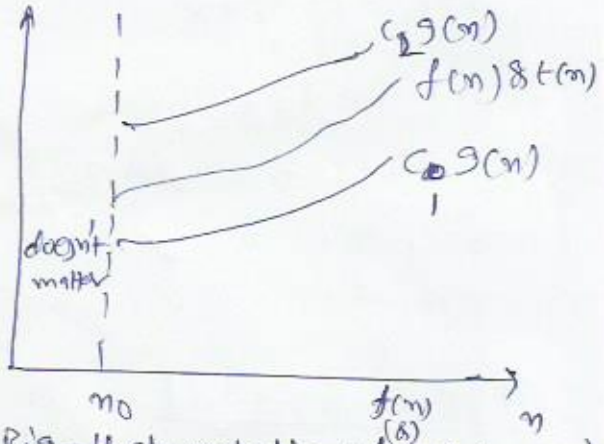
second, we prove the left inequality (the lower bound):

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

$$\geq \frac{1}{2} n^2 - \frac{1}{2} n \frac{1}{2} n \text{ (for all } n \geq 2)$$

$$= \frac{1}{4} n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.



Big-Theta notation: $t(n) \in \Theta(g(n))$

Examples 1. $10n^2 + 4n + 2 = \Theta(n^2)$

2. $6 * 2^n + n^2 = \Theta(2^n)$.

3. $6 * 2^n + n^2 \neq \Theta(n^2)$

4. $n^3 = \Theta(n^3 + 2)$.

Q- Examples solved

1. Prove that $3n+2 = O(n)$.

Soln: $3n+2 \leq 4n$ for all $n \geq 2$. $n_0 = 2$

$n=1, 3+2 \leq 4$ ✗

$n=2, 6+2 \leq 8$ ✓

P.T

3. $100n + 6 = O(n)$

Soln: $100n + 6 \leq 101n$ for all $n \geq 6$. $n_0 = 6$

$n=1, 100+6 \leq 101$ ✗

$n=2, 200+6 \leq 202$ ✗

$n=3, 300+6 \leq 303$ ✗

$n=4, 400+6 \leq 404$ ✗

$n=5, 500+6 \leq 505$ ✗

$n=6, 600+6 \leq 606$ ✓

2. Prove that $3n+3 = O(n)$.

$3n+3 \leq 4n$ for all $n \geq 3$

Soln: $n=1, 3+3 \leq 4$ ✗ $n_0 = 3$

$n=2, 6+3 \leq 8$ ✗

$n=3, 9+3 \leq 12$ ✓

4. P.T $6 * 2^n + n^2 = O(2^n)$.

$6 * 2^n + n^2 \leq 7 * 2^n$ for $n \geq 4$

$n=1, 12+1 \leq 14$ ✗ $n=4, 96+16 \leq 112$ ✓

$n=2, 24+4 \leq 28$ ✓ $n=5, 192+25 \leq 224$ ✓

$n=3, 48+9 \leq 56$ ✗

Ω - Examples solved

1. Prove that $3n+2 = \Omega(n)$

Soln $3n+2 \geq 3n$ for $n \geq 1$.

$n=1,\ 3+2 \geq 3$ ✓

$n=2,\ 6+2 \geq 6$ ✓

2. P.T $3n+3 = \Omega(n)$.

Soln $3n+3 \geq 3n$ for $n \geq 1$.

$n=1,\ 3+3 \geq 3$ ✓

$n=2,\ 6+3 \geq 6$ ✓.

3. P.T $100n+6 = \Omega(n)$

Soln: $100n+6 \geq 100n$ for $n \geq 1$.

$n=1,\ 100+6 \geq 100$ ✓

$n=2,\ 200+6 \geq 200$ ✓

4. P.T $10n^2+4n+2 = \Omega(n^2)$

$10n^2+4n+2 \geq n^2$ for $n \geq 1$.

$n=1,\ 10+4+2 \geq 1$ ✓

O - Examples solved

Ω O

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

1. Prove that $3n+2 = O(n)$

Soln $3n+2 \geq 3n$ for all $n \geq 1$

Ω $n=1,\ 3+2 \geq 3$ ✓

$n=2,\ 6+2 \geq 6$ ✓

$3n+2 \leq 4n$ for all $n \geq 2$

O $n=1,\ 3+2 \leq 4$ ×

$n=2,\ 6+2 \leq 8$ ✓

So $c_1=3,\ c_2=4$ & $n_0=2$.

2. P.T $3n+3 = O(n)$.

$3n+3 \geq 3n$ for all $n \geq 3$

$n=1,\ 3+3 \geq 3$

Ω $n=2,\ 6+3 \geq 6$

$n=3,\ 9+3 \geq 9$

$3n+3 \leq 4n$ for all $n \geq 3$

$n=1,\ 3+3 \leq 4$ ×

O $n=2,\ 6+3 \leq 8$ ×

$n=3,\ 9+3 \leq 12$ ✓

So $c_1=3,\ c_2=4$ & $n_0=3$.

Little "oh" notation The function $f(n) = o(g(n))$ iff

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Ex: 1. P.T $3n+2 = o(n^2)$.

Since $\lim_{n\to\infty} \frac{3n+2}{n^2} = \frac{3n}{n^2} + \frac{2}{n^2} = \frac{3}{n} + \frac{2}{n^2} = 0$

P.T

2. $3n + 2 = o(n \log n)$.

Since. $\lim_{n\to\infty} \frac{3n+2}{n\log n} = \frac{3n}{n\log n} + \frac{2}{n\log n} = 0$

Ex: 2 P.T $3n+2 = o(n^2)$

3. P.T $n^3 = o(n^4)$

4. P.T $3n+2 = o(n \log n)$

little omega notation The function $f(n) = \omega(g(n))$ iff

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0.$$

Ex1: P.T $3n^2 =$ [illegible] $\Omega(3n+2)$

Ex2: P.T $n^4 = \Omega(n^3)$

Ex3: P.T $n^2 = \Omega(3n+2)$

Ex4: P.T $n \log n = \Omega(3n+2)$

Soln: $\lim_{n\to\infty} \frac{n}{3n+2} = \lim_{n\to\infty} \frac{3n+2}{n} = \frac{3}{n} + \frac{2}{n^2}$

$= 0$

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 (a simple fact about two arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2 \max\{b_1, b_2\}$.)

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \le c_1 \cdot g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 \cdot g_2(n) \quad \text{for all } n \ge n_2.$$

L'Hôpital's Rule

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Stirling's formula

$n! \simeq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ for all large values of n.

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \geq \max\{n_1, n_2\}$ so that we can use both inequalities. Adding the two inequalities above yields the following:

$$t_1(n) + t_2(n) \leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n)$$
$$\leq c_3 \cdot g_1(n) + c_3 \cdot g_2(n) = c_3[g_1(n) + g_2(n)].$$
$$\leq c_3 \cdot 2 \max\{g_1(n), g_2(n)\}.$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2\max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively.

So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a larger order of growth, i.e., its least efficient part:

$$\left.\begin{array}{l} t_1(n) \in O(g_1(n)) \\ t_2(n) \in O(g_2(n)) \end{array}\right\} t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

Basic efficiency classes The time efficiencies of a large no. of algorithms fall into only a few classes. They are

class	name	comments
1	constant	best-case efficiencies
$\log n$	logarithmic	cutting a problem's size by a constant factor
n	linear	scan a list of size n (seq search)
$n \log n$	"n-log-n"	many divide-and-conquer algorithms (merge, quick sort) fall into this category
n^2	quadratic	two embedded loops (nxn matrix)
n^3	cubic	matrix multiplication.
2^n	exponential	generate all subsets of an n-element set
$n!$	factorial	generate all permutations of an n-element set

Theorem 2

Show that $f(n) = O(n^m)$, where $f(n)$ is a polynomial of degree 'm'.

Proof let $f(n)$ be a polynomial of degree 'm'.

$f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_m n^m$ where $a_m > 0$

and $a_0, a_1, a_2, \cdots, a_m$ are constants.

We shall prove that $f(n) = O(n^m)$.

~~We have that~~

$\therefore$ let

$$p = \max\{|a_0|, |a_1|, |a_2|, \cdots, |a_m|\}$$

$$a_0 \le p \cdot n^m, \quad \forall\ n \ge 1 \longrightarrow (1)$$

$$a_1 \cdot n \le p \cdot n^m, \quad \forall\ n \ge 1 \longrightarrow (2)$$

$$a_2 \cdot n^2 \le p \cdot n^m, \quad \forall\ n \ge 1 \longrightarrow (3)$$

$$\vdots \quad \vdots \quad \vdots \quad \vdots$$

$$a_m n^m \le p \cdot n^m, \quad \forall\ n \ge 1 \longrightarrow (m+1).$$

adding all the above equations, we get

$$a_0 + a_1 n + a_2 n^2 + \cdots + a_m n^m \le \underbrace{p \cdot n^m + p \cdot n^m + \cdots + p \cdot n^m}_{m+1 \text{ times}}$$

$$\therefore f(n) \le (m+1)\, p \cdot n^m, \quad \forall\ n \ge 1$$

$$\boxed{f(n) = O(n^m).} \qquad c = (m+1)p \ \&\ n_0 = 1.$$

Ex: let $f(n) = n^2 + 4n + 3$, Prove that $f(n) = O(n^2)$.

let $p = \max\{1, 4, 3\}$

$= 4$

$$n^2 \le 4n^2, \quad \forall\ n \ge 1 \longrightarrow (1)$$

$$4n \le 4n^2, \quad \forall\ n \ge 1 \longrightarrow (2)$$

$$3 \le 4n^2, \quad \forall\ n \ge 1 \longrightarrow (3)$$

adding all the above equations, we get

$$n^2 + 4n + 3 \le 4n^2 + 4n^2 + 4n^2, \quad \forall\ n \ge 1$$

$$\le 12 n^2$$

$\therefore f(n) = O(n^2)$.

Theorem 3 show that $f(n) = \Omega(n^m)$, where $f(n)$ is a polynomial of degree 'm'.

Theorem 4 show that $f(n) = \Theta(n^m)$, where $f(n)$ is a polynomial of degree 'm'.

Time & space complexity

Time complexity It is defined as the amount of time taken to run an algorithm.

Space complexity It is defined as the amount of space required to run an algorithm.

Types of Time complexity

1. worst case - It is defined as the maximum amount of time taken to run the algorithm.
2. Best case - minimum amount of time taken to run the algorithm.
3. Average case - which is neither worst case nor Best case

Note: worst case, Best case Time complexity is unique.
Average case Time complexity is not unique.

Procedure for finding Time complexity

1. To execute an assignment statement once, 1 unit of time is required. Ex: a=b;
2. The amount of time required to call a function for 'x' times is 'x' units. Ex: add();
3. To execute a condition once, 1 unit of time is required.
   Ex: i<=n;
4. To execute unary operators like ++, --, 1 unit of time is required. Ex: i++, i--
5. To execute a 'return' statement once, 1 unit of time is required. Ex: return n;

→ Find the Time complexity of the following code snippet.

```
for( i=1; i<=n; i++)
  {
    printf("in DMAA");
  }
```

To execute $i=1$, 1 unit of time is required.

To execute $i \leq n$, $n+1$ units of time is required.

∵ we are checking the condition for $(n+1)$ times.

To execute $i++$, n units of time is required.

∵ we are executing $i++$ for n times

∴ Total amount of time is $1 + (n+1) + n$

$= (2n+2)$

$= O(n)$

→ find the time complexity of the following algorithm.

```
for( i=1; i<=n; i++)
{
    for( j=1; j<=n ; j++)
    {
        Printf ("DMAA\n");
    }
}
```

when $i=1$, the inner for loop runs for 'n' times.

when $i=2$, the inner loop run for n times.

⋮

when $i=n$, the inner loop run for n times }

∴ Time complexity $= \underbrace{n+n+\cdots+n}_{n \text{ times}}$ }

$= O(n^2)$.

→ write an algorithm to search the given array using linear search and hence find the worst, average and best case time complexities.

array - a[ ]

element to be found - x in the list of n elements.

```
for (i=0; i<=n-1; i++)
{
    if( a[i] == K)
    {
      Printf("\n element is found at %d th position", i);
      break;
    }
}
```

Ans Suppose if the array contains 'n' elements then array elements are $a[0], a[1], \cdots, a[n-1]$. let k be the element to be searched in the given array.

Best-case Time complexity Best-case occurs, if k match with starting element of the array. i.e $k = a[0]$.

In this case, the time complexity is $O(1)$.

worst-case Time complexity It occurs either if k is found at last location of the array (or) if the element doesn't found at any location of the array. in this case the time complexity is $O(n)$.

Average-case Time complexity:

$$\text{Avg. case Time complexity} = O\left(\frac{1+n}{2}\right)$$
$$= O(n) \quad \text{for successful search.}$$

→ Find the time complexity of an algorithm which searches an element in the given array using binary search.

let 'n' be the no. of elements in the given array. let $T(n)$ be the time taken to search an element in the given array of 'n' elements using binary search.

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ 1+T(n/2) & \text{if } n \geq 2 \end{cases}$$

$$T(n) = 1 + T(n/2)$$
$$= 1 + (1 + T(n/4)) \quad \text{Substitute } n/2 \text{ in place of } 'n'.$$
$$= 2 + T(n/4)$$
$$= 2 + (1 + T(n/8)) \quad \text{Substitute } n/2 \text{ in place of } 'n'$$
$$= 3 + T(n/2^3).$$
$$\vdots$$
$$T(n) = p + T(n/2^p).$$

let $n = 2^p$

$$= p + T(1)$$
$$= p + 1$$
$$= \log_2 n + 1$$
$$\Longrightarrow O(\log_2 n)$$

Best case Time complexity

It occurs either if array contains only single element (or) element to be searched is found at middle location of the array.

∴ In this case Time complexity = $O(1)$.

Worst case Time complexity:

Worst case Time complexity is derived from the relation

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

∴ WC TC = $O(\log_2 n)$.

Average case T.C:

$$O\left(\frac{\log_2 n + 1}{2}\right) = O(\log_2 n)$$

→ Find the time complexity of an algorithm which prints n^{th} fibonocci number. Fibonacci series is 1, 1, 2, 3, 5, 8, 13, 21, 34 ....

```
Algorithm fib(n)                                    1
{                                                   1
    if (n:=1) then
        return n;
    else
        a := 0;                                     1
        b := 1;                                     1
        for i = 2 to n do   i=2=1 unit  (n-1) units  i++ n (n-2) times
        {
            c := a+b;                               n
            a := b;                                 n
            b := c;                                 n
        }
    return c; // printf("%d", c);                   1
}
```

∴ Time complexity = $4 + 1 + 3(n-1) + (n-2) + 1$

$= 5n$

$= O(n)$.

→ calculate the time complexity of armstrong number algorithm

```
algorithm armstrong(n).
{
    sum=0; t=n;
    while(n>0) do
    {
      n=n%.10;
      sum:= sum + n*n*n;
      n:= n/10;                                  n= 153
    }
    if (sum==t)
     Printf("given no. is armstrong number");
    else
     Printf("not armstrong");
}
```

Time complexity

- To execute the assignment statement t=n, 1 unit of time required.
- To execute the assignment statement sum=0, 1 unit of time required.

∴ The condition n>0 is checked for k+1 times, where k is the no. of digits in the given number n, the amount of time taken is k+1.

∴ The condition n>0 is true for k times, the three assignment statement inside the while loop will be executed for k times each, the amount of time taken is 3k units.

To check the condition whether sum=t or not, the amount of time taken is 1unit.

∴ Time complexity = 1+1+(k+1)+3k+1

= 4k+4

= O(k).

→ write an algorithm for strong number and also calculate the time complexity.

```
Algorithm strong(n)
{
    sum=0;
```

```
while(n>0)                    P+1
{
   m = n % 10;                P
   f = fact(m);               P.(a+1) times
   sum = sum + f;             P
   n = n/10;                  P
}
if sum == n then              1
  printf("strong no");        1
else
  printf("not strong");
}
int fact(int a)
{
   for i=1 to a do            (a) times
     j = j * i;
   return j;
}
```

$\therefore$ Time complexity $= 1+P+1+P+P.(a+1)+P+P+1+1$

$= 5P+4+aP.$

$= O(aP).$

→ write a recursive algorithm to find the factorial of a given positive integer & find its time complexity.

```
Algorithm fact(n)
{
    if(n==1) then
      return n;
    else
      return(n * fact(n-1));
}
```

Ex: $n=5$

5 * Fact(4)
↓
4 * Fact(3)
↓
3 * Fact(2)
↓
2 * Fact(1)
↓
1

Total amount of time taken = $n+n$ units

where first 'n' units is the time taken to check the condition $n=1$, the second n units is the time taken to call the function itself.

TC = $2n$ units = $O(n)$.

→ write a recursive algorithm to find nth fibonacci number.

1, 1, 2, 3, 5, 8

```
Algorithm fib(n)
{
    if (n==0) then
      return 0;
    if (n==1) then
      return 1;
    else
      return (fib(n-1) + fib(n-2));
}
```

fib(5) + fib(4)

fib(4) + fib(3)

fib(3) + fib(2)

fib(2) + fib(1)

fib(1) + fib(0)

→ write a recursive algorithm to compute ^nC_r & find its time complexity

```
Algorithm NCR(n, r)
{
    if ( n==r or r==0) then
      return 1;
    else
      return (NCR(n-1, r) + NCR(n-1, r-1));
}
```

$^nC_r = {}^{n-1}C_r + {}^{n-1}C_{r-1}$

In worst case, it occurs if $n \neq r$. but after some time n becomes equal to 'r'.

n becomes equal to 'r', if n is decrement for $(n-r)$ times

∴ The amount of time taken to call the function NCR itself is $(n-r)$ units.

∴ Time complexity = $O(n-r)$.

Best case It occurs if $n = r$, the amount of time time taken is 2 units. (1 unit - to check the condition $n=r$ & 1 unit - to execute statement return)

∴ Time complexity = $O(1)$.

→ Derive the function $f(n) = 12n^2 + 6n$ in $O(n^3)$ & $\omega(n)$.

we have to first prove that

$12n^2 + 6n = O(n^3)$

$12n^2 + 6n \leq c \cdot n^3$, $\forall n \geq n_0$ where c is a real number. n_0 is a natural number.

$p = \max\{12, 6\} = 12$

$\therefore 12n^2 \leq 12 \cdot n^3$, $\forall n \geq 1$ → ①

$6n \leq 12n^3$, $\forall n \geq 1$ → ②

① + ②, we get

$12n^2 + 6n \leq 12n^3 + 12n^3$

$12n^2 + 6n \leq 24n^3$, $\forall n \geq 1$

$\therefore c = 24, n_0 = 1$

hence, $12n^2 + 6(n) = O(n^3)$.

now, we have to prove that $f(n) = \omega(n)$.

here $g(n) = n$

consider, $\lim_{n \to \infty} \frac{g(n)}{f(n)}$

$\lim_{n \to \infty} \frac{n}{12n^2 + 6n}$

$\therefore$ by L'Hopital's rule, $\lim_{n \to \infty} \frac{g(n)}{f(n)} = \lim_{n \to \infty} \frac{g'(n)}{f'(n)}$

$= \lim_{n \to \infty} \frac{1}{24n + 6}$

$= \frac{1}{24 \cdot \infty + 6}$

$= \frac{1}{\infty} = 0$

hence, $12n^2 + 6n = \omega(n)$.

Exercise 2.4

→ $x(n) = x(n-1) + 5$ for $n > 1$,
$x(1) = 0$. $n = 1$.

Solution:

$x(n) = x(n-1) + 5$

$x(1) = 0$

$x(2) = x(1) + 5 = 5$

$x(3) = x(2) + 5 = 5 + 5 = 10$

$x(4) = x(3) + 5 = 10 + 5 = 15$

⋮ ⋮

$x(n-1) = x(1) + 5(n-1)$

$= 5(n-1)$

$\therefore x(n) = O(n)$.

→ $x(n) = 3x(n-1)$ for $n > 1$,
$x(1) = 4$.

Solution:

Given $x(n) = 3x(n-1)$.

$x(1)$ = back ward substitution

$= 3x(n-2)$

$= 3x(n-3)$

$= 3x(n-(n-1))$

$= 3x(1)$

$= 3 \cdot 4$

$= 12$.

(or)

$x(1) = 4$

$x(2) = 3 \cdot x(1) = 12$

$x(3) = 3 \cdot x(2) = 3 \cdot 12 = 36$

$x(4) = 3 \cdot x(3) = 3 \cdot 36 = 108$

$x(n-1) = 3 \cdot x(n-2)$

$4[1 + 3 + 9 + 3^3 + \cdots 3^{n-2}]$

$4[1 + 3 + 3^2 + 3^3 + \cdots]$

$x(n) = 4 \cdot 3^{n-1}$.

$\therefore x(n) = O(3^n)$.

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

Soln: $x(1) = x(0) + 1 = 1$

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

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

⋮ ⋮

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

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

$\therefore x(n) = O(n^2)$.

4. $x(n) = x(n/2) + n$ for $n > 1$,
$x(1) = 1$ (solve for $n = 2^k$).

Soln:

$x(1) = 1$

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

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

$x(4) = x(2) + 4 = 1 + 2 + 4$

$x(5) = x(5/2) + 5 = 1 + 5$

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

sub $n/2$ in n $= \left(x(n/4) + \frac{n}{2}\right) + n$

$n/4$ in n $= x(n/8) + \frac{n}{4} + \frac{n}{2} + n$

$n/8$ in n $= x(n/16) + \frac{n}{8} + \frac{n}{4} + \frac{n}{2} + n$

$$= x\left(\frac{n}{2^k}\right) + \frac{n}{2^{k-1}} + \frac{n}{2^{k-2}} + \frac{n}{2^{k-3}} + \frac{n}{2^{k-4}}$$

let $n = 2^k$

$k = \log_2 n$

$$= x(1) + \frac{2^k}{2^{k-1}} + \frac{2^k}{2^{k-2}} + \frac{2^k}{2^{k-3}} + \frac{2^k}{2^{k-4}}$$

$$= 1 + 2 + 4 + 8 + 16 + \cdots$$

$$= 2^n - 1.$$

$\therefore x(n) = O(2^n)$.

5. $x(n) = x(n/3) + 1$ for $n > 1$,

$x(1) = 1$ (solve for $n = 3^k$)

soln

$$x(n) = x(n/3) + 1$$

sub $n = n/3$

$$= \left[x\left(\frac{n/3}{3}\right) + 1\right] + 1$$

$$= x(n/9) + 2$$

sub $n = n/9$

$$= \left[x\left(\frac{n/9}{3}\right) + 1\right] + 2$$

$$= x(n/27) + 3.$$

$$= x\left(n/3^k\right) + k$$

let $n = 3^k$

$k = \log_3 n$

$$= x(1) + \log_3 n$$

$$= 1 + \log_3 n$$

$\therefore x(n) = O(\log_3 n)$.