Unit Test - 1

PAGE NO.:
DATE: / /

Q.1 (A)

1. What is algorithm?

→ An algorithm is a finite set of instruction that, if followed, accomplishes a particular task.

2. What is running time of an algorithm?

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

3. Enlist the technique to design an algorithm?

→
- Divide & Conquer
- Greedy Algorithm
- Brute force / exhaustive search
- Back tracking
- Branch & Bound Algorith
- Dynamic Programing

4. How do we calculate worst case time complexity of an algorithm?

→ In the worst case analysis, we calculate upper bound on running time of an algorithm. Therefore, the worst case time complexity of linear search would be $\Theta(n)$.

Q-1 (B)

1. Differentiate pseudocode and real code?

→ Pseudocode is informal language used to describe an algorithm in a imprecise way for pedagogical purposes. It cannot be compiled in any meaningful sense.

Program code is written in a programming language which is indeed, extremely precise. It can be compiled into executable code, whether it's native code or byte code. For interpreted language, no executable code make be generated and the program code is simply perform line by line by a runtime program.

2. When an algorithm is said to be corrected?

→ An algorithm must satisfy all the following criteria:

① Input - Zero or more quantities are externally supplied

② Output - At least one quantity is produced

③ Definiteness - Each instruction is clear and unambiguous.

④ Finiteness - If we trace out the instruction of an

algorithm then for all cases, the algorithm terminates after a finite number of steps.

(5) Effectiveness - Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enought that each operation be defined as an operation 3, it must be feasible.

3. Enlist the properties that should be posses by loop invariant?

→ A loop variant is a statement about program variables that is true before and after each iteration of a loop.
A good loop invariant should satisfy three properties:

(1) Initialization - The loop invariant must be true before the first execution of the loop.

(2) Maintenance - If the variant is true before an iteration of the loop, it should be true also after~~t~~ iteration.

(3) Termination - When the loop is terminated the invariant should tell us something useful, something that help us understand the algorithm.

PAGE NO.:
DATE: / /

Q.2
(A) Derive the running time for binary search using recurrence relation?

→
```
int binarySearch (int num[], int low, int high,
                  int target)
{
    if (low > high)
    {
        return -1;
    }
    int mid = (low + high)/2;
    if (target == nums[mid])
    {
        return mid;
    }
    else if (target < nums[mid])
    {
        return binarySearch (nums, low, mid-1,
                             target);
    }
    else
    {
        return binarySearch (nums, mid+1, high, target);
    }
}
```

We know that at each step of the algorithm, our search space reduces to half. That means if initially, our ~~space~~ search space contain n

elements, then after one iteration it contains
$n/2$ then $n/4$ - - - - - $n - 1$

suppose our search space is hausted after k step
Then,

$$n/2^k = 1$$
$$n = 2^k$$
$$k = \log n$$

Therefore the time complexity of the binary search algorithm is $O(\log n)$, which is very efficient.
The auxiliary space required by the program is $O(1)$ for iterative implementation and $O(\log n)$ for recursive implementation due to call stack.

(B) Write an algorithm for sort an array using divide and conquer method?

→

```
Algorithm MergeSort (low, high)
// a[low : high] is a global array to be sorted
Small (P) is true if there is only one element
to sort. In this case the list is already sorted.
{
    if (low < high) then
    {
        // Divide P into subproblems
        // find where to split the set
```

```
    mid := [(low + high)/2];
    // solve the sub problem.
        MergeSort (low, mid);
        MergeSort (mid + 1, high);
    // Combine the solutions
        Merge (low, mid, high);
    }
}
```

[c] Discuss any three criteria to evaluate the efficiency of an algorithm?

→ Space Time trade off

It is a way of solving in less time by using more storage space or by solving given algorithm in very little space by spending more time.

To solve a given programming problem, many different algorithm may be used. Some of these algorithm may be extremely time efficient and other extremely space efficient.

It refers to a situation where you can reduce the use of memory at the cost of slower program execution, or reduce the running time at the cost of increased memory usage.

PAGE NO.:
DATE: / /

Asymptotic Notations

These are language that use meaningful statements about time and space complexity. The following three notation are used to represent time complexity of algorithm:

- Big O – often used to describe worst case of an algorithm.
- Big Ω – Big omega is reverse of Big O, if Big O is used to describe upper bond then this is used to describe lower bond.
- Big Θ – When algo^{n} has a complexity by Big Ω = Big O, then it actually has the complexity Θ (n log n), which means the running time of that algorithm always falls in n log n in best & worst case

There are 3 case
- Best case
- Average case
- Worst case

Let us assume a list of n number of value stored in an array. Suppose, if we want to search a particular element in this list then algorithm that search the key element in list among n element, by comparing the key element with each element in the list sequentially.

PAGE NO.:
DATE: / /

Q.3
(A

The best case would be if the first element in the list matches with the key element to be searched in a list of elements. The efficiency in that case would be expressed as O(1) because only one comparision is enough.

The worst case will be if the complete search list is searched and the element is found on at end of the list or is not found.

Average case could be obtained by finding Average number of comparisin.

minimum no = 1 maximum numbe = n if the elem not found.

Average = (n + 1)/2

Q.3 (A)

Describe P, NP and NP hard problem ?

* Polynomial algorithm:

The first set of problems are P- algorithm that we can solve in polynomial time, like logarithmetic, linear or quadratic time. If an algorithm is polynomial, we can formally defines its time complexity as:

where and where and are constant and is input size. In general, for polynomial time algorithms is expected to be less than. Many algorithms complete in polynomial time:

- All basic mathematical operations; addition, substraction, division and multiplication.
- Testing for primacy.
- Hashtable, lookup, string operation, sorting problem
- Shortest Path algorithm; Djikshtra, Bellman-ford, Floyd-warshall
- Linear and Binary search algorithms for a given set of numbers.

NP Algorithm:

The second set of problem cannot be solved in polynomial time. However, they can be verified in polynomial time. We expect these algorithm to have an exponential complexity, which we will defined as:

where, and where and are

constant and is the input size. is a function of exponential time where at least and as a result, we get .
For example, we'll see complexities like in the set of problems. There are several algorithms that fit this description. among them are:

- Integer factorization
- Graph Isomorphism

[B]

NP-Hard Algorithm

Our last set of problem contains the Hardest most complex problems in Computer Science. They are not only hard to solve but are hard to verify as well. In fact some of these problems aren't even decidable. Among the hardest Computer Science problems are:

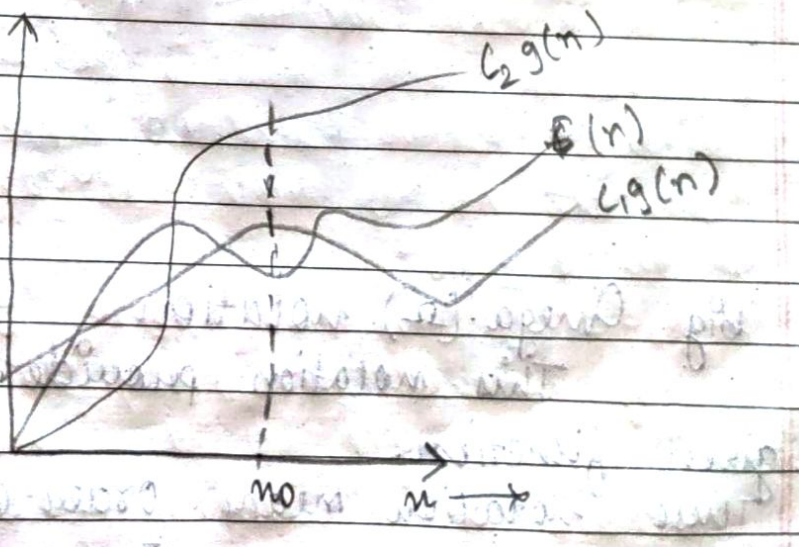
- K-means clustering
- Travelling Salesman problems
- Graph Coloring

These algorithms have a peculiarity similar to one in . They can all be reduced to any problem in . Because of that, these are in and are at least as hard as any other problem in . A problem can be both in and which is another aspect of being.

[B] Define asymptotic bound with graph?

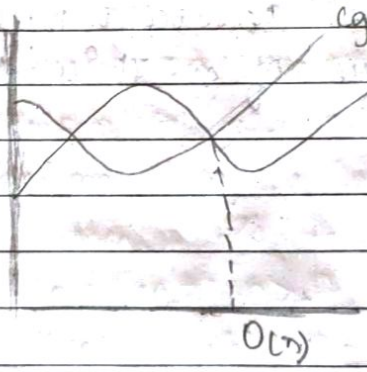
- Theta (Θ) Notation:

It provide both upper and lower bond for a given function.
It means order exactly. It implies a function is bounded above and bounded below both.
This notation provide both minimum and maximum time that a function can attain for any input size.
Let $g(n)$ be given function. $f(n)$ be the set of function defined as $\Theta(g(n)) = \{ f(n) :$ if there exist positive constant c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0 \}$
It can be written as $f(n) = \Theta(g(n))$ or $f(n) \in \Theta(g(n))$, here $f(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large value of n. It is described in the following figure.



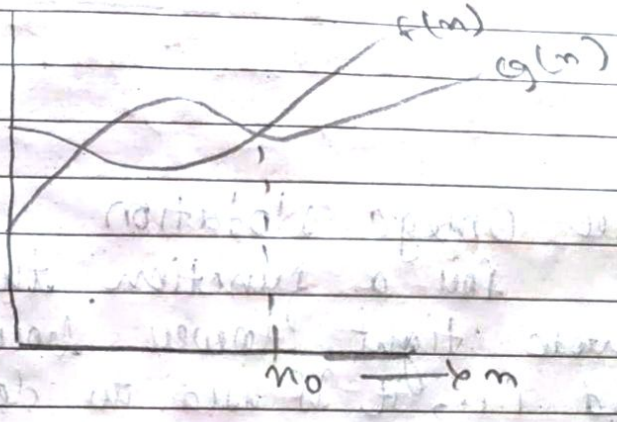
PAGE NO.:
DATE: / /

- Big Oh (O) notation
  This notation provide upper bound for a given function.
  O(Big Oh) notation means order at most i.e., bounded above or it will give maximum time required to run the algorithm.
  For a function having only asymptotic upper bound, Big oh 'O' notation is used.
  Let a given function $g(n)$, $O(g(n))$ is the set of function $f(n)$ defined as $O(g(n)) = \{f(n)$ if there exist positive constant c and n_0 such that $0 \leq$ such that $0 \leq cg(n)$ for all n, $n > n_0$, $f(n) = O(g(n))$ or $f(n) \in O(g(n))$, $f(n)$ is bounded above by some positive constant multiple of $g(n)$ for all large values of n. The definition is illustrated with the help of figure 3.



- Big Omega (Ω) notation
  This notation provide lower bound for a given function.
  These notation mean order at least ie., minimum time required to execute the algorithm or have lower bound.

for a function having only asymptotic lower bound, Ω notation is used.
Let a given function $g(n)$. $\Omega(g(n))$ is the set of functions $f(n)$ defined as $\Omega(g(n)) = \{f(n)$: if there exist positive constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all n, $n \geq n_0\}$
$f(n) = \Omega(g(n))$ or $f(n) \in \Omega(g(n))$, $f(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large value of n. It is described in the following figure.

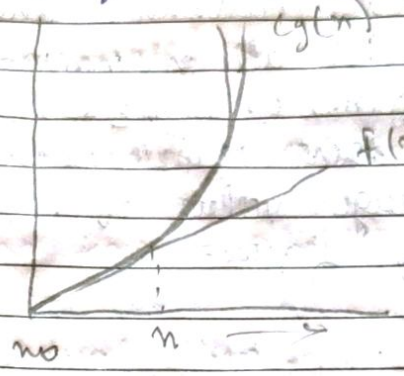


- Small O (o) Notation.

For a function that does not have asymptotic tight upper bound, O (small o) notation is used i.e., It is used to denote an upper bound that is not asymtotically tight.
Let a given function $g(n)$, $O(g(n))$ is the set of function $f(n)$ defined as $O(g(n)) = \{f(n)$: for any positive constant c there exist a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$

PAGE NO.:
DATE: / /

$f(n) = O(g(n))$ or $f(n) \in O(g(n))$, $f(n)$ is loosely bounded above by all positive constant multiple of $g(n)$ for all large n. It is illustrated by following figure:

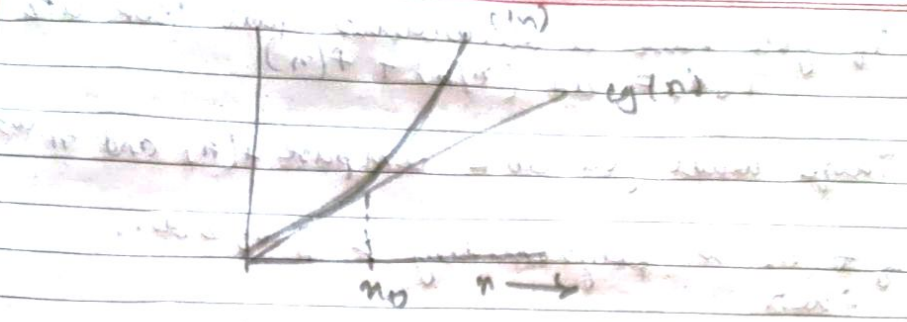


- Small Omega Notation

for a function that doesn't have asymtotic tight lower bound, ω notation is used i.e., it is used to denote lower bond that is not asymptotically tight.

Let a given function $g(n)$. $\omega(g(n))$ is the set of function $f(n)$ defined as $\omega(g(n)) = \{f(n):$ for any positive constant $c > 0$ there exist a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.

$f(n) = \omega(g(n))$ or $f(n) \in \omega(g(n))$, $f(n)$ is loosely bounded below by all positive constant multiple of $g(n)$ for all large n. It is described in following figure:



(c) Derive the running time for binary search using trace tree?

→

```
Algorith search (t, x)
{
  if (t = 0) then return 0;
  else if (x = t → data) then
        return Search (t → lchild, x);
  else
        return search (t → rchild, x);
}
```

We can write a recurrence for this algorithm:
Let T(n) represent the running time of binary search, then we can write

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

Where n = end - start + 1

There is something called master Theorem that says: (more or less)

PAGE NO.:
DATE: / /

If you have a recurrence that look like
$T(n) = aT(n/b) + f(n)$

Then what you do is compare $f(n)$ and $n^{\log_b(a)}$.

if $f(n)$ is polynomially $> n^{\log_b(a)}$,
then
running time $= \Theta(f(n))$
else
running time $= O(n^{\log_b(a)})$.

If both are equal then,
$\Theta(f(n) \log n)$.

In binary search,
$f(n) = 1$
$a = 1$
$b = 2$
$\log_b(a) = \log_2 1 = 0$.
$n^0 = f(n)$.

running time $= \Theta(\log n)$.