29/03/24 Date: / / Page no:

Assignment 01

(Q1.) What is ADA? What is the need to study algorithms? Explain in detail.

Ans. Analysis and Design of Algorithms = ADA is a branch of computer science that focuses on the study of algorithms, their applications in solving computations problems.

Need to study = An algorithm is a sequence of steps to solve a problem. It acts like a set of instructions on how a program should be executed.

- The main aim of designing of an algorithm is to provide a optimal solution for a problem. Not all problems must have some type of solutions, therefore we must adopt various strategies to provide feasible solutions for all types of problems.

- ADA covers the concept of designing an algorithm as to solve various problems in CS and IT and also analyse the complexity of these algorithms designed.

- A good algorithm should be small and less time consuming that means it should have less time and space complexities.

Q2. Give the divide and conquer solution for Quick sort and analyse its complexity.

ns. Quick sort is a highly efficient sorting algorithm and is based on partioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than pivot value.

Quick sort partitions of an array and then calls itself recursively twice to sort the 2 resulting subarrays.

- It is good for large-sized data sets as its average and worst case complexities are $O(n^2)$, respectively.

↳ Pseudo code for quick sort algo :-

```
Procedure : quicksort ( left , right )
    if left - right <= 0
      return
    else
      pivot  = A [right]
    partition = partitionfunction (left, right,
                                   pivot)
quick sort ( left, partition - 1)
quicksort (partition +1 , right)
 end if
end procedure
```

↳ Analysis :-

The worst case complexity of quick sort algorithm is $O(n^2)$. However

average cases generally we get the output in O(n log n) times.

↳ Implementation :-

```
# include <iostream>
using namespace std;
# define MAX 7
int int Array [MAX]= {4,6,3,2,1,9,7}
void display () {
   int i;
cout <<"[";
for (i=0 ; i < MAX ; i++) {
   cout << int Array [i] <<" " endl;
  }
  cout <<" ]" << endl;
 }
void swap (int num1 , int num2) {
int temp = int Array [num1];
int Array [num1] = int Array [num2];
int Array [num2] = temp;
}
int partition (int left, int right,
                int pivot) {
int left pointer = left - 1;
int right pointer = right ;
while (true) {
while (intArray [++ leftPointer] < pivot){
 }
while (rightPointer > 0 & intArray [--rightPointer] > pivot) {
}
if (leftpointer >= rightPointer) {
   break ; }
 else {
cout << "Item swapped : "<< intArray [leftPointer] << "," << intArray [rightPointer] >> << endl
swap (leftPointer, rightPointer);
}
} cout << endl << "Pivot swapped :"<<
intArray [leftpointer] <<"," << intArray [rightPointer] <<endl ;
swap << " updated array :";
display ();
return leftPointer;
}
void quicksort (int left, int right)
 { if (left - right <= 0 ) {
     return ;
```

```
} else {
int pivot = int Array [ right ];
int partition point = partition
( left , right , pivot );
quicksort ( lift , partition point - 1 );
quicksort ( partition Point + 1 , right);
}
}
int main()
{
cout << "Input array : " ;
display (1);
quicksort ( 0 , MAX - 1 );
cout << " Output array :";
display ; , return 0;
}
```

↳ Output

Input Array : [4, 6, 3, 2, 1, 9, 7]
Pivot swapped : 9, 7
Updated array : [4, 6, 3, 2, 1 , 7, 9]

Pivot swapped : 4, 1
Updated ~~st~~ array : [1, 6, 3, 2, 4, 7, 9]
Item swapped : 6, 2

Pivot swapped : 6, 4
Updated array : [1, 2, 3, 4, 6, 7, 9]

Pivot swapped : 3, 3
Updated array : [1, 2, 3, 4, 6, 7, 9]
Updated array : [1, 2, 3, 4, 6, 7, 9]
Output array : [1, 2, 3, 4, 6, 7, 9]

Q3. Define asymptotic Notations. Give different notations used to represent complexity of algorithms.

A. Time function of an algorithm is represented by T(n), where 'n' is input size.
Different types of asymptotic notations are used to represent the complexity of an algorithm.

- Big O Notation - O
- Big Omega notation - Ω
- Big Theata Notation - Θ
- ~~Big~~ little o notation - o

(i) Big O (O) = The notation O(n) is the formal way to express the upper bound

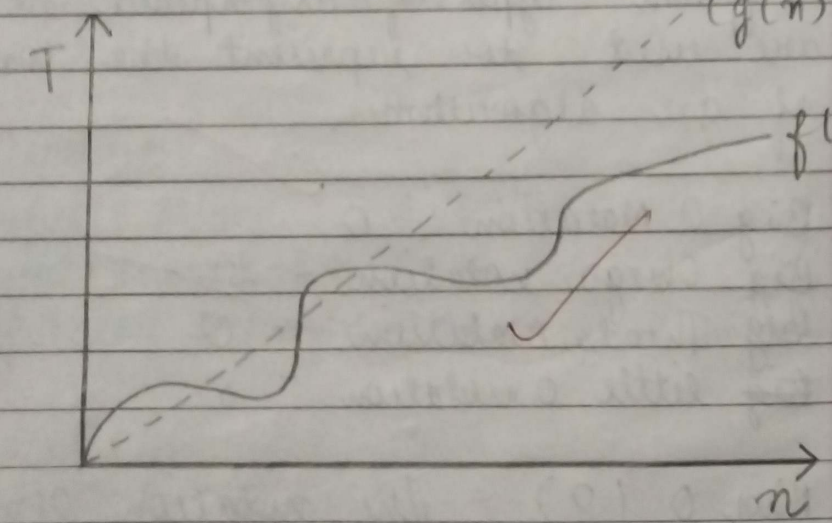
of an algorithm's running time complexity.

↳ It is most commonly used notation.

↳ It measures the worst time complexity.

1. A function $f(n)$ can be represented as the order of $g(n)$ that is $O(g(n))$, if there exists a value of positive integer n as n_0 and a positive constant C such that -

$f(n) \leq C \cdot g(n)$ for $n > n_0$ in all cases. Hence, function $g(n)$ is an upper bound for function $f(n)$ as $g(n)$ grows faster ~~that~~ than $f(n)$

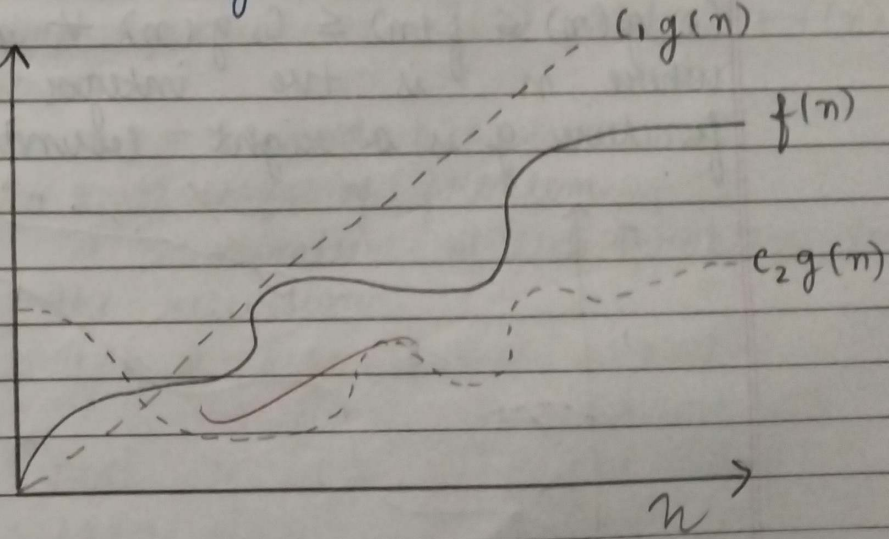


(ii) Big Omega (Ω) :- The notation $\Omega(n)$ is the formal way to express lower bound of an algorithm's running time complexity.

↳ It measures the best case complexity.

↳ We say that $f(n) = \Omega(g(n))$ when there exists constant C that is $f(n) \geq C \cdot g(n)$, $\forall\ n \geq n_0$, where n is positive integer.

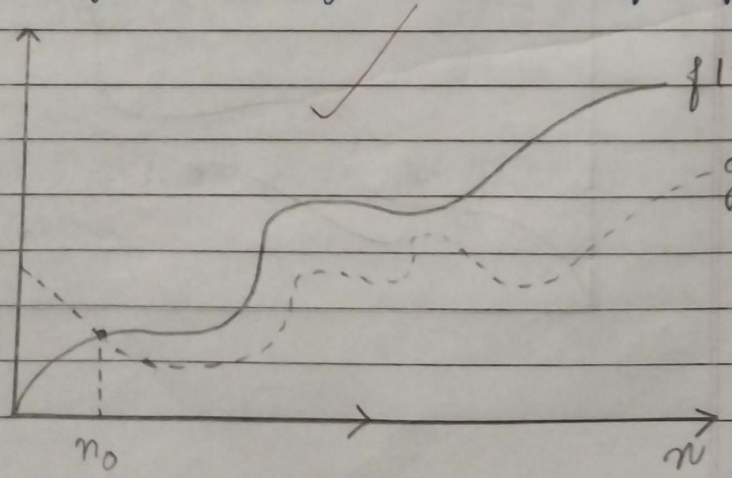
↳ It means function g is a lower bound for function f; after a certain value of n, f will never go below g.



(iii) Big Theta (Θ) :- The notation $\Theta(n)$ is formal way to find / express both the lower and upper bound of an algorithm's running time.

↳ Some may confuse the theta notation as the average case time complexity; while big theta notation could be almost accurately used to describe the average case, other notations could be used as well.

↳ We say that $f(n) = \Theta(g(n))$ when there exists a constant c_1 and c_2 that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ $\forall$ $n > n_0$ where n is +ve integer. This means function g is a right bound for fun f.



Q4. Write all three cases of Master theorem for the equation :

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

Ans. Master Theorem :- It is one of the many methods that area applied to calculate time complexity of algorithms

↳ In analysis time complexities are calculated to find out the best optimal logic of an algorithm.

↳ Master's theorem is applied on recurrence relations

⟶ Given equation is : $T(n) : aT\left(\frac{n}{b}\right) + f(n)$

where,

$a >= 1$ and $b > 1$

n -> size of problem

a -> number of sub problems in the recursion

n/b -> size of sub problems based on the assumptions that all sub-problems are of same size.

$f(n)$ - represents the cost of work done outside the recursion → $\Theta(nK \log^p n)$, where $K >= 0$ and p is a real number.

→ In this case, there is three cases in the master theorem to determine asymptotic notations:-

If $a > bK$, then $T(n) = \Theta(n \log b/a)$

[$\log a = \log a / \log b$]

If $a = bK$,

If $p > -1$, then $T(n) = \Theta(n \log b\, a \log^{p+1} n)$

If $p = -1$, then $T(n) = \Theta(n \log b\, a \log \log n)$

If $p < -1$, then $T(n) = \Theta(n \log b\, a)$

If $a < bK$:

If $p >= 0$, $T(n) = \Theta(nK \log^p n)$

If $p < 0$, $T(n) = \Theta(nK)$

This was Master Theorem for dividing function.

Q5 How can we prove that Strassen's Matrix multiplication is advantageous over ordinary matrix multiplication.

Ans: Strassens Matrix multiplications is the divide and conquer approach to solve the matrix multiplication.

↳ The ordinary matrix multiplication method multiplies each row with each column to achieve productive matrix: time complexity taken by this algorithm / approach is $O(n^3)$

↳ Strassens method reduces time complexity from $O(n^3)$ to $O(n^{\log 7})$

Naive Method :-

First we will discuss Naive method and its complexity.

Here, we are calculating $Z = X \times Y$ using Naive method two matrices X & Y can be multiplied if the order of these matrices are $p \times q$ and $q \times r$ and the resultant matrix will

Date: / / Page no:

6s of order p x q.

Pseudo code :-

→ Algorithm : Matrix - Multiplication (X, Y, Z)

```
for i = 1 to p do
for j = 1 to r do
  Z [i, j] #: = 0
for K = 1 to q do
  Z [i, j] : = z [i, j] + x [i, K]
            + y [K, j]
```

→ Complexity :- Here we assume that integer operations take O(1) time, There are three for loops in this algorithms and one is nested in other, thus algorithm takes $O(n^2)$ time.

* Strassen's Matrix Multiplication Algorithm :-

In this context, using strassens matrix multiplication algorithm, time, complexity can be reduced to a little bit.

- It can be performed on only square matrices where n is a power of 2. Order of both matrices are n x n.

Divide X, Y and Z into four $(\frac{n}{2}) \times (\frac{n}{2})$ matrices as represents below

$$Z = \begin{bmatrix} I & J \\ K & L \end{bmatrix} \qquad X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

$$Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Using strassens algorithm compute the following :-

$M_1 = (A+C) \times (E+F)$
$M_2 = (B+D) \times (G+H)$
$M_3 = (A-D) \times (E+H)$
$M_4 = (A) \times (F-H)$
$M_5 = (L+D) \times (E)$

$M_6 = (A+B) \times (H)$
$M_7 = D \times (G-E)$

Then,

$I = M_2 + M_3 - M_6 - M_7$
$J = M_4 + M_6$
$K = M_5 + M_7$
$L = M_1 - M_3 - M_4 - M_5$

↳ Analysis,

$$T(n) = \begin{cases} c & , \text{if } n = 1 \\ 7 \times T\left(\frac{n}{2}\right) + d \times n^2 & , \text{otherwise} \end{cases}$$

where c & d are constants.

Using this recurrence relation, we get $T(n) = O(n^{\log 7})$

Hence the complexity of strassens matrix multiplications algorithm would be $O(n^{\log 7})$

- Thus the strassens algorithms reduces the number of multiplications required compared to naive method, resulting in improved time complexity.
Hence this algorithm is advantageou over the ordinary matrix multiplicati

Strassen's
11/3/24.