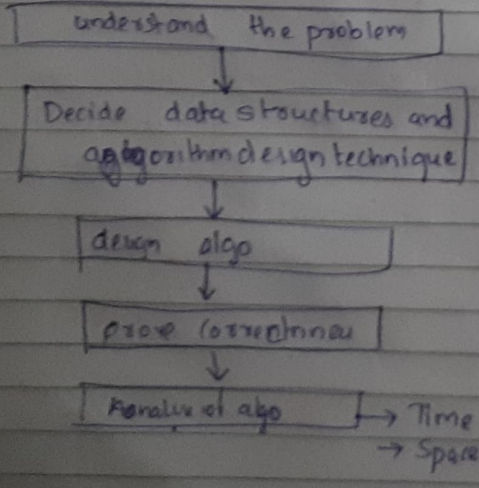
Mod-1

Date __/__/____

1) Time Complexity
2) Need for analyzing an algo
3) Big O
4) Merge Sort with eg.
5) Control abstraction for divide and Conquer — 3
6) measure performance of algo
7) algorithm to compute sum of n numbers and find its time and space complexity
8) Chara of algo
9) Dif time and Space complexity
10)

1) Linear Search algorithm, best, worst, average case
2) Merge Sort, Worst Case analysis.
3) asymptotic Complexity of algo, 3 notations used for asymptotic
4) divide and Conquer Strategy is used in Merge Sort



Saathi

Algorithm

- step by step finite sequence of instruction to solve well defined computational problem.

Algorithm	Program
- written at design time	- written at implementation time
- Language independent	- dependent
- independent on hardware and software	- dependent

Characteristics of Algo → Criteria for a good algo

① Input :- 0 or more value
② Output : 1 or more values
③ Definiteness :- Each instruction are clear and unambigious
④ effectiveness :- ~~do sum~~ avoid unnecessary steps
⑤ finiteness :- step are finite

Steady of Algo requires 3 steps
① Algo design
② Proving correctness of algo.
③ Analysis & algorithm

notes

Time Complexity

Time Complexity of an algorithm is the amount of time it needs to run to completion.

Running time depends on
① I/p to program
② Compiler
③ Nature and Speed of instruction

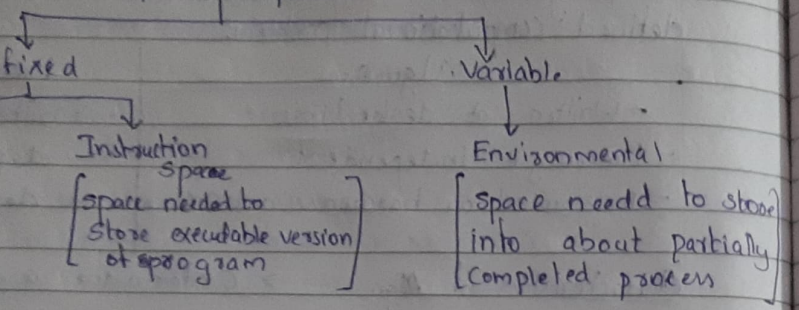
Space Complexity

The amount of memory needs to run an algorithm for Completion.

Date / /

Saathi

Space Complexity



Need for analysising an algorithm

① predict the behaviour of an algorithm without implementing it on a specific computer.
② compare performance of two algorithms for solve same problem
③ required to know the capability of the algorithm.

Time and Space Trade off

(notes)

Time Complexity	Space Complexity
def	def
- Calculates the time needed	- Calculates the memory space needed
- Counts time for all statements	- Counts memory space for all variables even inputs and outputs
- Depends mostly on input data size	→ Depends mostly on auxillary variable size
Time Complexity of merge Sort is O(n log n)	Space :- O(n)

Page No.

Date / /

Saathi

Analyzing time and space Complexity

① Frequency Count Method

A. Sum two numbers

	Cost	Frequency
Add() {		
a = 10	C_1	1
b = 10	C_2	1
C = a + b	C_3	1
Print (C)	C_4	1
}		

Time complexity = O(1)

A → 1
b → 1
C → 1
S(n) = O(1)

B. Sum of n numbers

	Cost	Frequency
S = 0	C_1	1
for (int i = 0; i < n; i++)	C_2	n+1
{		
S = S + A[i]	C_3	n
}		
return S	C_4	1

Time complexity = $C_1 + C_2(n+1) + C_3(n) + C_4$

$= n(C_2 + C_3) \rightarrow T(n) = O(n)$

Space Complexity.

S → 1
A → n
i → 1
n → 1
S(n) = O(n)

Page No.

(2) Another way of finding time complexity of M in polygomial form using 2 rules

① drop low order terms (because it is not significant)

② drop constant multiplier

eg:- ① $2n^3 + 3n^2 + 4n + 6$

Time Complexity = $O(n^3)$

② $17n^4 + 3n^2 + 4n + 8 \rightarrow O(n^4)$

③ $16n + \log n \Rightarrow O(n)$

Time and Space Complexity Analysis of loops

① Addition of 2 $n \times n$ matrix

```
add(A, B, n) {
    for(i=0; i<n; i++){                    C1   n+1
        for(j=0; j<n; j++){                C2   n(n+1)
            c[i][j] = a[i][j] + b[i][j]    C3   n x n
        }
    }
    return c                               C4   1
}
```

Time complexity = $C_1(n+1) + C_2(n^2+n) + C_3(n^2) + C_4$

$= n^2(C_2 + C_3) + n(C_1 + C_2) + C_1 + C_2$

$\approx O(n^2)$

Space Complexity:

$A \rightarrow n^2$

$B \rightarrow n^2$

$C \rightarrow n^2 \longrightarrow 3n^2 + 3 \approx O(n^2)$

$i \rightarrow 1$

$j \rightarrow 1$

$n \rightarrow 1$

Page No.

② Multiplication of 2 ~~Square~~ matrix

```
mul(A, B, n, m) {                          Frequency
    for(i=0; i<n; i++){                    n+1
        for(j=0; j<m; j++){                n(n+1)
            for(k=0; k<n; i++){            n^2(n+1)
                C[i][j] += A[i][k] * B[k][j]   n^3
            }
        }
    }
    return 1                               1
}
```

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

$T(n) \Rightarrow O(n^3)$

Space Complexity:

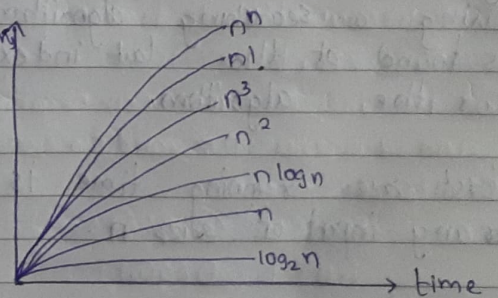
$A \rightarrow n^2$

$B \rightarrow n^2 \longrightarrow 3n^2 + 4 \approx O(n^2)$

$C \rightarrow n^2$

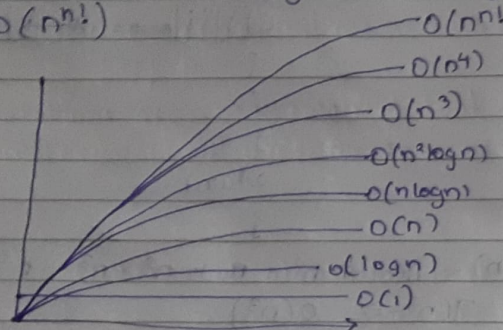
$i, j, k, n \rightarrow 1$

Growth Rate graph



- Arrange & illustrate the following growth rate in increasing order $O(n^4)$, $O(1)$, $O(n)$, $O(n\log n)$, $O(n^2\log n)$, $O(\log n)$, $O(n^3)$, $O(n^{n!})$

$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2\log n) < O(n^3) < O(n^4) < O(n^{n!})$



Worst Case

In this case, the algorithm will take maximum running time for input size n to execute

eg:- We are Searching an element in an array using a searching algorithm then if an element is found at the last index then it's worst case for the algorithm. → or not in the arra

Worst case running time is the longest running time for any input of size n.

Average Case

In this case, the algorithm will take the ~~minimum~~ average running time for input size.

eg:- Element found in middle

Average case running time assume that all inputs of a given size are equally likely.

Best Case

In this case, the algorithm will take the minimum running time for an input size.

eg:- at first element of array

In the best case analysis, we calculate lower bound on running time of an algorithm.

Amortized running time

It refers to the time required to perform a sequence of related operations averaged over all the operations performs.

Linear Search Algorithm

```
int search(int arr[], int n, int x)
{
    for(int i=0; i<n; i++)
    {
        if(arr[i] == x) {
            return i;
        }
    }
    return -1;
}
```

Worst Case time complexity : $O(n)$

Average Case " " :- $O(n)$

Best Case " " :- $O(1)$

Saathi

Date

Asymptotic notation

Answer paper

To enable us to make meaningful statements, about time and space complexity of an algorithm, asymptotic notations are used. It describes behaviour of time and space complexity of large instance characteristics.

Asymptotic notations are mathematical tool to represent the time complexity of algorithms for asymptotic analysis.

↓

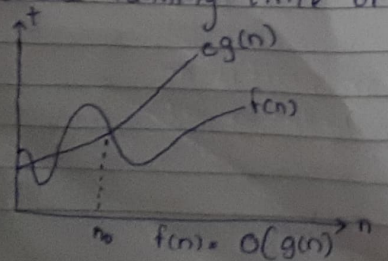
To measure the efficiency of algorithm that.
→ don't depend on machine specific constants
→ don't require algorithms to be implemented

Asymptotic notations:

(1) Big O
- used for expressing the upperbound of an algorithm running time
- Measure longest time possibly taken for algorithm to complete.
- f(n) and g(n) are non-negative function, if there exists an integer n_0 and a constant $c>0$, such that for integer $n \geq n_0$

$$f(n) \leq c \cdot g(n)$$
$$f(n) = O(g(n))$$

- Big O gives Worst case running time of an algo.



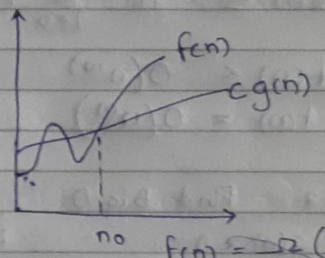
Date

Saathi

(2) Omega (Ω)
- used to represent lower bound of an algorithm
- f(n) and g(n) be non-negative integer, if there exists an integer n_0 and a constant $c>0$, such that $\forall$ integer $n \geq n_0$

$$f(n) \geq c \cdot g(n)$$
$$f(n) = \Omega(g(n))$$

- It provides best case complexity of an algorithm.
- Measure of minimum time taken for algorithm to complete



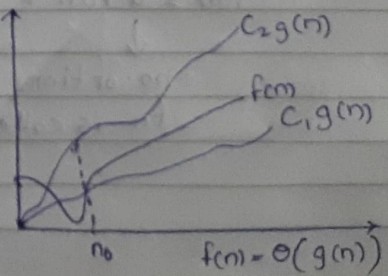
(3) Theta (θ)

If the function is b/w the lower and upper bound, the function is denoted as θ notation.

For any non negative function f(n) and g(n), if there exist an integer n_0 and positive constant C_1 & C_2, $C_1 > 0$ and $C_2 > 0$, such that for every integer $n \geq n_0$

$$C_1 g(n) \geq f(n) \geq C_2 g(n)$$

which is denoted as $f(n) = \theta(g(n))$



Proof:

If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n^1 + a_0$

$f(n) = O(g(n))$

$f(n)$ → function or a statement

$a_m > 0$ → mean value of m is greater than zero

$$f(n) \leq \sum_{i=0}^{m} |a_i| n^i$$

$$f(n) \leq n^m \sum_{i=0}^{m} |a_i| n^{i-m}$$

$f(n) \leq O(n^m)$ → dropped low order term

$f(n) = O(n^m)$

3. $f(n) = 3n+2$ Find Big O

Sol: Assume $g(n) = n$ $\quad f(n) \leq C\, g(n)$

(notes)

Recurrence Equations

- A recurrence ~~equation~~ is an equation or inequality that describes a function in terms of its values of smaller i/p.
- useful for expressing the running time of recursive algorithms.

General form:

$T(n) = aT(n) + g(n)$ → time required for doing other operation.

↓ no of times recursive fn is called

eg: Recurrence eqn for factorial

$T(n) = 0$ if $n = 0$

$T(n) = 1 \cdot T(n-1) + b$ for $n > 0$

↓ constant

Solving Recurrence Equations

Three methods for solving recurrence are

1. Substitution method
2. ~~Recursion Tree~~ Iteration Method
3. Masters Theorem

Subsitution Method

Subsitution method use mathematical induction to find the constant and show that the solution works. The subsitution method can be used to establish either upper or lower bound on a recurrence.

Date / /

Divide and Conquer

It is a top-down approach for designing algorithm that consist of dividing the problem into smaller sub problems hoping that the solution of sub problems are easier to find.

The composing the partial solutions into the solution of the original problem.

① Divide

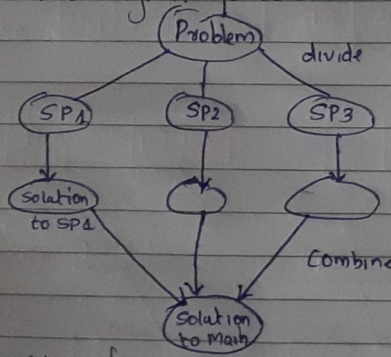
Dividing the problem into number of sub problem that are smaller instances of same problem

② Conquer

Conquer (solve) the subproblems by solving them recursively.

③ Combine

Combining the solutions of sub problems into solutions of the original problem.



→ Control Abstraction for divide and conquer algorithm.

```
divide_and_conquer(P)
{
    if small(P)  // P is very small, soln is trival
        return solution(S);
    else;
        divide the problem P in to k instances P1, P2 ---Pk
        return( combine( divide_and_conquer(P1),
                         divide_and_conquer(P2),
                         .
                         .
                         divide_and_conquer(Pk)))
}
```

Date / /

Merge Sort

- uses divide and conquer strategy
- The basic idea is to divide the list into sublist
  - sort each sublist
  - finally merge them to get a single sorted list

```
Merge (A,p,q,r)
{
    n1 = q-p+1
    n2 = r-q
    L1 = [1, ---, n1+1]   } be new arrays
    R [1 ---, n2+1]       }
    for (i=1 to n1)
        L[i] = A[p+i-1]
    for (j=1 to n2)
        R[j] = A[q+j-1]
    L[n1+1] = ∞
    R[n2+1] = ∞
    i=1  j=1
    for (k=p to r)
        if (L[i] ≤ R[j])
            A[k] = L[i]
            i = i+1
        else
            A[k] = R[j]
            j = j+1
}
```

Saathi

Date

For eg:

1 5 7 8 | 2 4 6 9

n_1 n_2

Step 1:

Create two new list $L = n_1 + 1$ $R = n_2 + 1$

$L = [1, 5, 7, 8]$ $R = [2, 4, 6, 9]$

and place ∞ at n

$L = [1, 5, 7, 8, \infty]$ $R = [2, 4, 6, 9, \infty]$

Step 2:

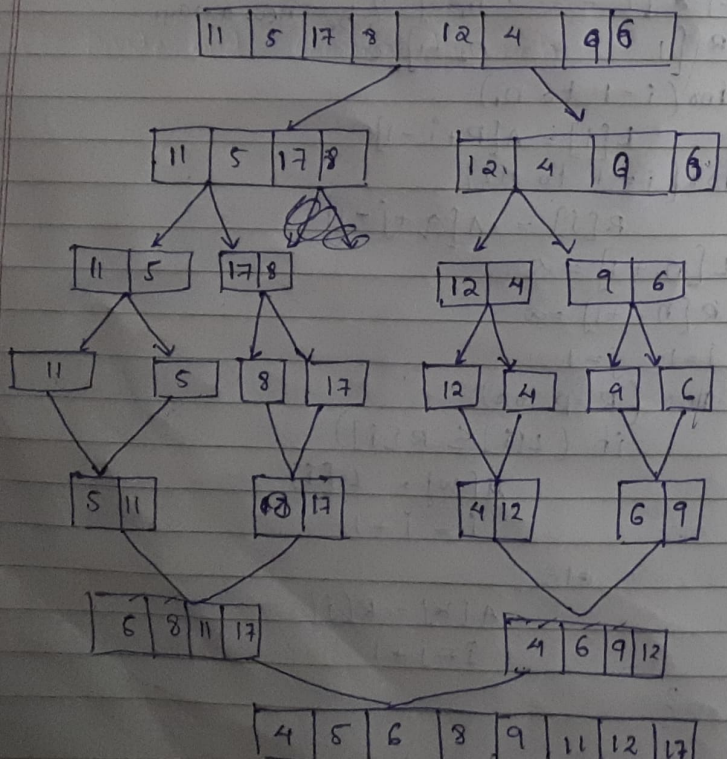
~~Compare and fill A~~

Create another array A[k]

$A[k] = [1, 5, 7, 8, \infty]$

Step 3:

Compare and fill A[k]



Page No.

Saathi

Date

Merge Sort (A, P, γ)

{

if $P < \gamma$

$q = P + \gamma/2$

Merge Sort (A, P, q) — $T(n/2)$

Merge Sort (A, q+1, γ) — $T(n/2)$

Merge (A, P, q, γ) — $O(n)$

}

The total time taken for merge sort is

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

Using Master's Theorem

$a = 2$

$b = 2$

$k = 1$

$p = 0$

$T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

$= \Theta(n \log n)$

$\Theta(n)$

Quick Sort

→ Recursive algorithm that follows divide and conquer approach.

→ The best case and average case time complexity is nlogn.

→ Worst case time complexity is $O(n^2)$

- Given an array, take any element from the array. Call it as pivot value and put pivot value in the position in such a way that elements less than the pivot value comes on left hand side and element greater than the pivot value comes on right hand side.

Page No.

```
QUICK-SORT(A,P,r)
{
  if(P<r)
  {
  q = PARTITION(AP,r)
  QUICKSORT(A,P,q-1)
  QUICKSORT(A,q+1,r)
  }
}
```

Matrix Multiplication

3 methods

(1) Naive Method

Time Complexity - $O(n^3)$ - 3 loops

(2) Divide and Conquer Approach