18.04.24
Thursday

MODULE 1

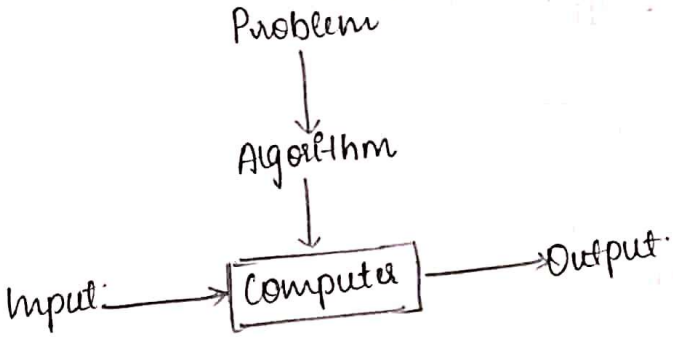
NOTION OF AN ALGORITHM.

*Algorithm is defined as step by step procedure used to solve a particular problem in a finite set amount of time.

*Algorithm is finite set of instructions that if followed accomplishes a particular task

Characteristics / Features of algorithm

1. Input - 0 or more quantities are externally supplied.
2. Output - atleast 1 o/p is produced.
3. Definiteness - no ambiguity while writing instructions. [each instr. must be clear & unambiguous]
4. Finiteness - result obtained in finite amount of time
5. Effectiveness - able to make out result (correctness) — able to trace the o/p by providing sample i/p (instance)



* gcd (m,n) = gcd (n, m mod n)
stopping condition gcd (m,0) = m.

* Algorithm Euclid (m,n)
// Compute gcd (m,n) by Euclid's algorithm.
// Input two non negative, not both zero integers m & n.
// Output Greatest common divisor of m & n.

```
while n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
return m
```

* Consecutive Integer Checking Algorithm

Step 1: Assign the value of min{m,n} to t.

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

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

Step 4: Decrease the value of t by 1 & go to step 2.

Eg. min{10,3}.

(1) t = 3

(2) m/t = 10/3 = 3 r=1.

(4) t = 2.

(2) m/t = 10/2 = 5 r=0.

(3) n/t = 3/2 = 1 r=1.

(4) t = 1.

(2) m/t = 10/1 = 10 r=0.

(3) n/t = 3/1 = 3 r=0.

* Middle School Procedure. to find gcd of {m,n}

Step 1: Find the prime factor of m.

Step 2: Find the prime factor of n.

Step 3: Identify all the common factors in the two prime expansions found in step 1 & 2. (if p is the common factor occuring P_m & P_n times in m & n respectively it should be repeated $\min\{P_m, P_n\}$

Step 4: Compute the product of all the common factor & return it as the greatest common divisor of the numbers given.

Eg. gcd {60,12}

60 = (2), (2), (3), 5.

12 = (2), (2), (3)

gcd {60,12} = 2 × 2 × 3 = 12

*. Sieve of Eratosthenes to find the prime factors of an integer Algorithm sieve(n)

// implements the ~~algorithm~~ of sieve of Eratosthenes

// Input an integer $n \geq 2$.

// Output: array L (of all prime numbers) $\leq n$.

```
for p ← 2 to n do A[p] ← p
for p ← 2 to ⌊√n⌋ do        ⌊ ⌋ floor symbol. [takes only integer value]
    if A[p] ≠ 0
        j ← p * p
        while j ≤ n do
            A[j] ← 0
            j ← j + p
// copy the remaining elements of A to array L of the primes.
p ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i + 1
return L.
```

Eg n = 16

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value ←			2	3	~~4~~ 0	5	~~6~~ 0	7	~~8~~ 0	~~9~~ 0	~~10~~ 0	11	~~12~~ 0	13	~~14~~ 0	~~15~~ 0	~~16~~ 0

p = 2

A[2] ≠ 0

j = 2 * 2 = 4.

4 ≤ 16

4 ≤ 16

A[4] = 0.

j = j + p.

= 4 + 2.

6 ≤ 16 ✓

A[6] = 0.

j = 6 + 2 = 8

8 ≤ 16 ✓

A[8] = 0.

j = 8 + 2 = 10.

10 ≤ 16 ✓

A[10] = 0.

j = 10 + 2 = 12

12 ≤ 16 ✓

A[12] = 0.

j = 12 + 2 = 14.

16 ≤ 16 ✓

14 ≤ 16 ✗

p = 3.

A[3] ≠ 0

j = 3 * 3 = 9

9 ≤ 16 ✓

12 ≤ 16 ✓

15 ≤ 16 ✓

18 ≤ 16 ✗

p = 4.

j = 4 * 4 = 16

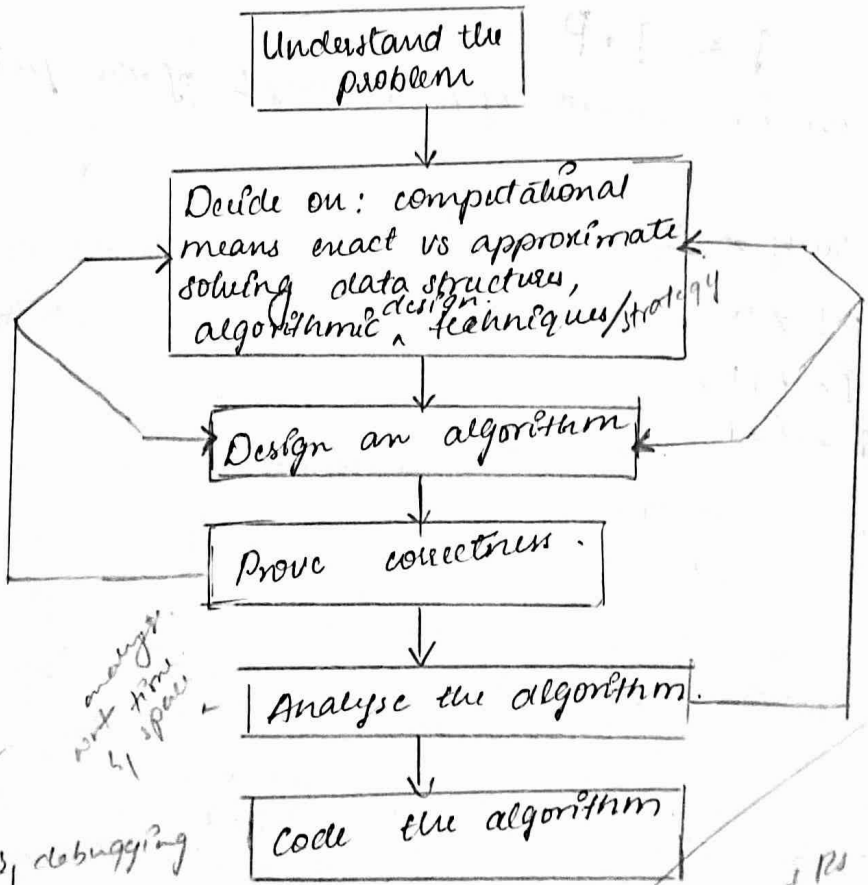
16 ≤ 16 ✓

20 ≤ 16 ✗

Array L contains : 2 3 5 7 11 13

22.04.24
Monday.

* Fundamentals of algorithmic problem solving.



program
↓
profiling & debugging
(measuring time & space efficiency)

Algorithm + PL = prog.

✓ Von Neumann - sequential architecture.

Specification of algorithm.

Chapter 2 - Fundamentals of the analysis of algorithmic efficiency.

Understanding the problem

- Comprehensive understanding entails dissecting the problem statement, identifying key requirements, constraints & objectives.
- It involves mentally walking through the problem, devising test cases, and visualizing potential edge cases or exceptional scenarios.
- Active engagement with the problem, including asking clarifying questions, can reveal hidden complexities or ambiguities.

Ascertaining the capabilities of computational device

- Understanding the computational environment involves considering factors like process architecture, memory hierarchy and ~~~~ available resources.
- For sequential algorithms, adherence to Von Neumann architecture principles is paramount, whereas parallel algorithms leverage concurrent processing capabilities.
- Awareness of computational constraints guides algorithm design, ensuring scalability and efficiency across different hardware configurations.

Choosing between Exact & Approximate problem solving!

- Decision-making process regarding whether an exact solution or an approximation is more suitable for the problem.
- Approximation algorithms are considered when exact solutions are computationally infeasible or too time-consuming.

Algorithm Design Techniques:

- Introduction to various algorithm design paradigms that provide general approaches to problem-solving.
- Learning these techniques is essential for effectively addressing new and diverse computational problems.

Designing an algorithm & Data Structures.

- Discussion of the challenges in designing an algorithm tailored to a specific problem.

- Emphasis on choosing appropriate data structures to efficiently represent and manipulate problem instances.

Methods of Specifying an Algorithm:

- Overview of different methods for formally specifying algorithms, including natural language, pseudocode, and flowcharts.
- Emphasis on clarity & precision required in algorithm specifications to ensure understanding and correctness

Proving an Algorithm's correctness

- Proving that an algorithm produces the correct result for all legitimate inputs in a finite amount of time is crucial. Techniques like mathematical induction are often used for this purpose.

Analyzing an Algorithm:

- After correctness, efficiency is crucial. This involves analyzing time and space efficiency. Simplicity and generality are also desirable characteristics of an algorithm.

Coding an Algorithm

This involves implementing the algorithm as computer program, ensuring correctness, efficiency & effectiveness. Testing & debugging are essential steps in this process.

Iterative Improvement & Optimization

Even after obtaining a working algorithm, it's important to continually refine & optimize it for better performance. This iterative process may involve revisiting & fine-tuning the algorithm.

Challenges in Algorithmic Problem Solving

The text discusses challenges such as determining algorithm optimality, the complexity of problems and the question of whether every problem can be solved by an algorithm. It also emphasizes the creativity & rewarding nature of algorithm design.

Chapter 2 - Fundamentals of the analysis of algorithmic efficiency.

* Analysis framework :-

1) Time efficiency

2) Space efficiency → extra space an algorithm requires.

* Measuring the i/p size :-

→ Algorithm's efficiency is a function of some parameter indicating algorithm's i/p size.

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

→ computer scientists prefer measuring the size by b bits in the n's binary representation

* Units for measuring Running time

→ represented in 'ns'.

1) Speed of computer.

2) Quality of algorithm

3) Use of compiler

* Approaches.

1) Include count variable.

2) Basic operation — most time consuming part

COP

execution time of an basic operation on a particular computer.

C(n) - no. of times this operation needs to be executed for this algorithm.

T(n) - the running time T(n) of a program implementing this algorithm using the formula. $T(n) \approx COP \; C(n)$

24.04.24
Wednesday.

* Order of Growth

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10					
10^2	6.6	100					
10^3	10	1000					
10^4		10000					
10^5	17	100000					
10^6	20	100000					

* Best case, worst case, Average case efficiencies.

Algorithm SequentialSearch(A[0..n-1], K).

// Search is for a given value in a given array by sequential search

// Input an array A[0...n-1] & a search key K.

// Output: the index of the first element of A that matches K or -1 if there are no matching element.

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

worst case (2 cases)
run max time if
successful
or
unsuccessful

worst(n) = n

best(n) = 1

$0 \le p \le 1$

$1.\frac{p}{n} + 2\frac{p}{n} \ldots i\frac{p}{n} + \ldots + n \Rightarrow \frac{p}{n}[1+2+\ldots+n]$

$\Rightarrow \underbrace{\frac{p}{n}\left[\frac{n(n+1)}{2}\right]}_{\text{successful}} + \underbrace{n(1-p)}_{\text{not successful}}$

Worst case efficiency.

Worst case efficiency of an algorithm is its efficiency for the worst case input of size n which is an input of size n for which the algorithm runs the longest amount all possible inputs of that size.

$C_{worst}(n) = n$

Best case efficiency.

Best case efficiency of an algorithm is its efficiency for the best case input of size n which is an input of size n for which the algorithm runs the fastest amount all possible input of that size $C_{best}(n) = 1$

Average case efficiency

Probability of successful search is p $(0 \le p \le 1)$

Probability of the first match occuring in the i^{th} position of the list is same for every i

$$C_{average}(n) = 1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + 3 \cdot \frac{p}{n} + \ldots + i \cdot \frac{p}{n} + \ldots + n \cdot \frac{p}{n}$$

$$= \frac{p}{n}[1 + 2 + 3 + \ldots + n]$$

$$= \frac{p}{n}\left[n\left(\frac{n+1}{2}\right)\right] + n(1-p).$$

$p = 1$ (successful search)

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

$p = 0$ (unsuccessful search)

$C_{avg}(n) \approx n$

~~Amount~~ Amortized Efficiency:

Applies not a single run of algorithm rather to a sequence of operations performed on same data structure

Asympotic notations & basic efficiency classes.

Efficiency analysis framework concentrates on the order of growth of an algorithms basic operations count as the principle indicator of the algorithms efficiency to compare and rank them are three notations:

worst case O big O
best case Ω omega
avg case Θ theta

1) Big O - O worst case O(n)
2) Omega - Ω best case Ω(n)
3) Theta - Θ avg case Θ( )

- $f(n)$ & $g(n)$ can be any non negative functions defined on the set of natural numbers
- $f(n)$ algorithms running time indicated by its basic operation count $c(n)$
- $g(n)$ some simple function to compare the count with,

Informal introduction

- $O(g(n))$ is the set of all function with smaller or same order of growth as $g(n)$

Eg. 1) $n \in O(n^2)$
2) $100n + 5 \in O(n^2)$
} Linear & has smaller order of growth than $g(n) = n^2$.

3) $\frac{1}{2} n(n-1) \in O(n^2)$ } Quadratic & has same order of growth as n^2.

4) $n^3 \notin O(n^2)$
5) $0.00001 n^3 \notin O(n^2)$
} Cubic & has higher order of growth than n^2

6) $n^4 + n + 1 \notin O(n^2)$ } 4th degree polynomial

- $\Omega(g(n))$ set of all functions with larger or same order of growth as $g(n)$.

Eg. 1> $n3 \in \Omega(n^2)$

2> $\frac{1}{2} n(n-1) \in \Omega(n^2)$

3> $100n+5 \notin \Omega(n^2)$

- $\Theta(g(n))$ set of all functions that have same order of growth as $g(n)$

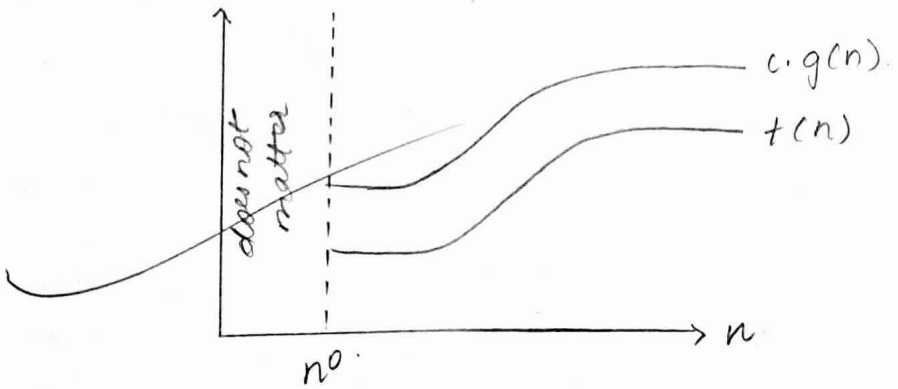
Big O notation

- A function $t(n)$ is said to be in $O(g(n))$ denoted as $t(n) \in O(g(n))$ if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n if there exists positive constant c and some non negative integer $n0$ such that $t(n) \leq c(g(n))$ for all n greater or equal to $n0$. $(n \geq n0)$

Eg: $100n+5 \in O(n^2)$

$101n \leq 101(n^2)$

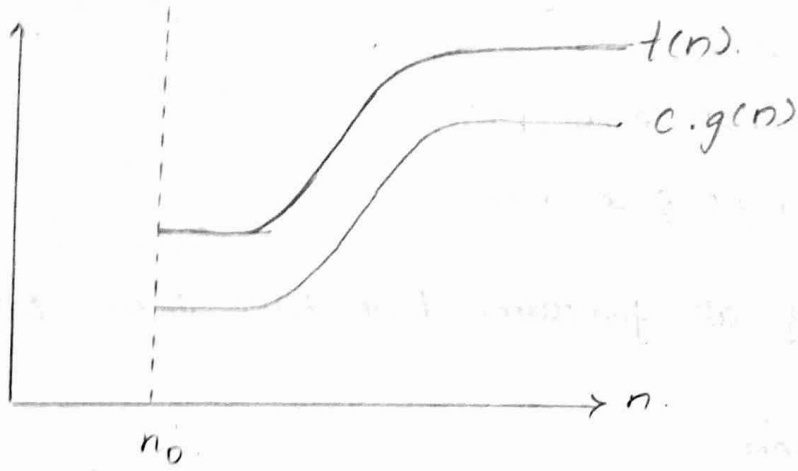
$100n+5 \leq 100n+n$



Omega notation

- A function $t(n)$ is said to be in $\Omega(g(n))$ denoted as. $t(n) \in \Omega(g(n))$ if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n if there. that is, if there exists some positive constant c and some non negative integer n_0 such that

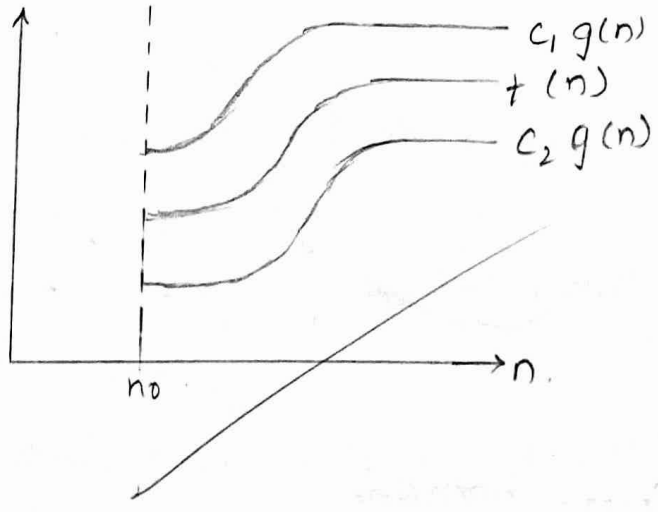
$f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.



Theta (Θ) notation

A function $f(n)$ is said to be in $\Theta(g(n))$ denoted as $f(n) \in \Theta(g(n))$ if $f(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n. If there exists some positive constant c_1 & c_2 & some non-negative integer n_0 such that

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



Example 2: to solve time complexity.

$f(n) = 6 \times 2^n + n^2$. (apply $n = 1, 2, 3, 4$ and find which has greater value to select $g(n)$)

$n = 1$

$6 + (2) + 1$

↳ has the greater value

Big O:- This should be more than 6.

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

$6 \times 2^n \leq 7 \times 2^n$

$n = 0, 1, 2$

for $n = 0$

$6 \times 1 \leq 7 \times 1$

for $n = 1$

$6 \times 2 \leq 7 \times 2$

for $n = 2$

$6 \times 4 \leq 7 \times 4$.

} it satisfies all the 3 conditions. therefore $n_0 = 0$

Big Ω :- less than 6.

$f(n) \geq c \cdot g(n)$.

$6 \times 2^n \geq 5 \times 2^n$.

$n = 0, 1, 2$.

for $n = 0$.

$6 \times 1 \geq 5 \times 1$

for $n = 1$

$6 \times 2 \geq 5 \times 2$.

for $n = 2$

$6 \times 4 \geq 5 \times 4$

} it satisfies all the 3 condition therefore $n_0 = 0$.

average (Big θ) :-

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

$5 \times 2^n \leq 6 \times 2^n + n^2 < 7 \times 2^n$

Satisfies all the 3 conditions then, $n_0 = 0$. for all these 3 cases $n_0 = 0$

Write a program to print a message on screen and figure out the time complexity.

```
public class fourthsem {
    psvm (String[] args) {
        SOP ("Hello Varsha!");
    }
}
```

The statement is getting exactly once. So the time complexity is

1. $T(n) = 1$, $T(n) \in \Omega(1)$ (Best case)

2. Write a program to print a message 'n' no. times and figure out the time complexity.

```
package T1;
import java.util.*;
public class Prg1 {
    psvm (String[] args) {
        Scanner sc = new Scanner (System.in);
        int n = sc.nextInt();
        for (int i=0; i<n; i++) {
            SOP ("Hello world");
        }
    }
}
```

Time complexity : $T(n) = n$.

$T(n) \in O(n)$ => worst case.

3. Write a program to add n numbers in an array and figure out the time complexity

```
public class Prg1. {
    psvm (String[] args) {
        Scanner sc = new Scanner (System.in);
        System.out.print ("Enter n: ");
        int n = sc.nextInt();
        int arr[] = new int[n];
        int sum = 0;
        SOP ("Enter array : ");
        for (int i=0 ; i<n; i++)
        {
            arr[i] = sc.nextInt();
            sum += arr[i];
        }
        SOP ("sum =" + sum);
    }
}
```

Time complexity : $T(n) = n$.

$T(n) \in O(n)$ worst case.

Using method

```
public class prg1.
{
    static int sum (int n, int[] arr) {
        int sum=0;
        for (int i=0 ; i<n; i++)
        {
            sum += arr[i];
        }
        return sum;
    }
```

```
psvm (string[] args) {
    Scanner sc = new Scanner(System.in);
    SOP ("Enter n:");
    int n = sc.nextInt();
    int arr[] = new int[n];
    SOP ("Enter array");
    for (int i=0 ; i<n; i++)
    {
        arr[i] = sc.nextInt();
    }
    SOP ("sum=" + sum(n, arr));
    sc.close();
}
}
```

4. Implement linear search concept and find time complexity.

```
public class Prg2 {
    psvm (String[] arg)
    {
        Scanner sc = new Scanner (System.in);
        SOP ("Enter n:");
        int n = sc.nextInt();
        int arr[] = new int[n];
        int sum = 0;
        SOP ("Enter array:");
        for ( int i=0 ; i<n; i++)
        {
            arr[i] = sc.nextInt();
        }
        SOP ("Enter search element :");
        int ele = sc.nextInt();
        int i=0;
        for (i=0; i<n; i++) {
```

```
            if (a[i] == ele)
            {
                SOP("search successful!");
                break;
            }
        }
        if (i==0)
        {
            SOP("search unsuccessful");
        }
        sc.close();
    }
}
```

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

$2^n < n!$ (max)

Note: theorem imp

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

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

eg $O(\max\{O(n^2), O(n)\})$

$\Rightarrow O(n^2)$.

comparing a time complexity in an algorithm.

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

- $a_1, b_1, a_2, b_2 \rightarrow$ arbitary constants.

$a_1 \leq b_1$

$a_2 \leq b_2$

this property holds good for Ω & Θ

$a_1 + a_2 \leq 2 \max\{b_1, b_2\}$

$t_1(n) \leq O(g_1(n))$ $\quad n \geq n_1$

$t_2(n) \leq O(g_2(n))$ $\quad n \geq n_2$.

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

consider $n \geq \max\{n_1, n_2\}$

$t_1(n) + t_2(n) \leq c_1 * g_1(n) + c_2 * g_2(n)$

$\leq c_3 * g_1(n) + c_2 * g_2(n)$

$\leq c_3 [g_1(n) + g_2(n)]$

$\leq c_3 * 2 \{\max(g(n), g_2(n))\}$

$\therefore t_1(n) + t_2(n) \in O\{\max(g_1(n), g_2(n))\}$ with constant c and no replaced by '0' (zero).

$2c_3 = 2 \max\{c_1, c_2\}$

02.05.24
Thursday

Ex. Array has identical elements

- Sort the array $\Rightarrow \frac{1}{2} n(n-1) = O(n^2)$.
- Scan the sorted array to check its consecutive elements for equality $\Rightarrow (n-1)$ comparison $= O(n)$

$\therefore$ Efficiency is given by $\Rightarrow O(\max\{g_1(n), g_2(n)\})$.

$= O(\max\{O(n^2), O(n)\})$.

$= O(n^2)$.

* Using limits for comparing order of growth.

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

- $0 \leq c < \infty \Rightarrow t(n) \in O(g(n))$
- $c > 0 \leq \infty \Rightarrow t(n) \in \Omega(g(n))$.
- $c > 0 \Rightarrow c_1 * g_1(n) \leq t(n) \leq c_2 * g_2(n)$. or $t(n) \in \Theta(g(n))$

* L Hospital's Rule.

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}.$$

* Stirling's formula.

$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ for larger values of n

Ex 1. Compare the order of growth of $\frac{1}{2}n(n-1)$ & n^2

Assume $\frac{1}{2}n(n-1)$ as $f(n)$ & n^2 as $g(n)$

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

$$\neq \frac{\infty}{\infty}$$

$$= \frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right) \quad \{1/\infty = 0\}.$$

$= 1/2$ ∴ This implies that $f(n)$ has same order of growth than $g(n)$.

2. Compare the order of growth of $\log_e n$ and $\sqrt{n}$

$$= \lim_{n\to\infty} \frac{(\log_e n)}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_e n)'}{(\sqrt{n})'}$$

$$= \lim_{n\to\infty} \frac{\frac{1}{n}\log_2 e}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{\sqrt{n}}{n} = 0$$

$= 0$ ∴ this implies that $f(n)$ has smaller order of growth than $g(n)$.

3. Compare the order of growth of $n!$ and 2^n

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

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

$= \infty$ ∴ this implies that $f(n)$ has larger order of growth than $g(n)$.

Basic asymptotic efficiency class

Class	Name	Comments
1	Constant	Best case efficiency.
$\log n$	Logarithmic	

n	linear.
$n \log n$	$n \log n$
n^2	Quadratic
n^3	Cubic
2^n	Exponential.
$n!$	Factorial

* Mathematical analysis of non recursive algorithm (iterative)

General plan for analysing time efficiency of non recursive algorithm

1. Decide on parameter indicating an i/p size.
2. Identify the algorithms on basic operations
3. Check whether the basic operation executed depends only on size of input. If it also depends on some additional property the worst case, the average average case and if necessary best case have to be investigated separately
4. Step up a sum expressing the number of time basic operation is executed.
5. Using standard formula and rules of some manipulation either find a closed form formula for the count / establish its order of growth.

* Standard formulas

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

2. $\sum_{i=1}^{n} a_i \pm b_i = \sum_{i=1}^{n} a_i \pm \sum_{i=1}^{n} b_i$

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

4. $\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \ldots + n$

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

Ex 1. Maximum Element

ALGORITHM MaxElement(A[0...n-1])

// Determines the value of the largest element in a given array

// Input: An array A[0..n-1] of real numbers

// Output: The value of the largest element in A

maxval ← A[0]

for i ← 1 to n-1 do

    if A[i] > maxval

        maxval ← A[i]

return maxval.

ANALYSIS

1. Input parameter: n

2. Basic operation: comparison

A[i] > max.

$C(n) = \sum_{i=1}^{n-1} 1.$ $\quad \left\{ \sum_{i=l}^{u} 1 = u - l + 1 \right.$

$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 - 1 + 1 = n - 1 \in \Theta(n).$

Ex 2: Element uniqueness problem

ALGORITHM Unique Element(A[0..n-1])

// Determine whether all the elements in a given array are distinct.

// Input: An array A[0..n-1]

Output: Returns "true" if all the elements in A are distinct and "false" otherwise.

```
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. Input parameter: n
2. Basic operation: comparision
   A[i] == A[j]
3. $C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 \quad \frac{(n-2)(n-1)}{2}$$

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

4. We also could have computed the sum $\sum_{i=0}^{n-2} (n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \ldots + 1 = \frac{(n-1)n}{2} \in \Theta(n^2)$$

$O(n^2)$ can be worst/avg case

Ex 3 Matrix multiplication

ALGORITHM Matrix Multiplication (A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])

// Multiplies two n-by-n matrices by the definition-based algorithm

// Input: Two n-by-n matrices A and B

// Output: Matrix C = AB.

```
for i ← 0 to n ← 1 do
    for j ← 0 to n-1 do
        C[i,j] ← 0.0
        for k ← 0 to n-1 do
            C[i,j] ← C[i,j] + A[i,k] * B[k,j]

return C
```

ANALYSIS

1. Input parameter is input size $n \times n$

2. Basic operation: comparison.

$$C[i,j] == C[i,j] + A[i,k] * B[k,j]$$

3. $\sum_{k=0}^{n-1} 1$ and the total number of multiplications $M(n)$ is expressed by the following triple sum:

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

Now, we can compute this sum by using formula (S1) and rule (R1) given above starting with the innermost sum $\sum_{k=0}^{n-1} 1$.

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = n \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} = n^2 \sum_{i=0}^{n-1} = n^2 \cdot n = n^3$$

$\in O(n^3)$

EX4 Number of binary digits

Algorithm to find number of binary digits in the binary representation of a positive decimal number.

```
ALGORITHM Binary(n)
// input a positive decimal number n
// output the number of binary digits in n's binary representation
```

```
count ← 1
while  n > 1  do
        count ← count + 1
          n ← ⌊ n/2 ⌋ - floored value
        return count
```

{ When diriding a number by 2 repeatedly the time complexity $\log_2 n$ }

ANALYSIS

$$C(n) = O(\log_2 n) + \underbrace{1}_{\text{checking the condition for the last time}}$$
$$\approx O(\log_2 n)$$

MATHEMATICAL ANALYSIS OF RECURSIVE ALGORITHM

General plan for analyzing the efficiency of recursive algorithms.

1> Decide on parameter indicating an input size

2> Identify the algorithm's basic operation.

3> Check whether the number of time the basic operation is executed can vary on different input of same size. if it can the worst, average and best case efficiencies must be investigated separately.

4> Set up a rigorous relation with an appropriate initial condition.

5> Solve the recurrence relation.

06.08.24
Monday.

Computing factorial function $f(n) = n!$ for an arbitary +ve integer n since $n! = n(n-1)\ldots 1$

$= n(n-1)! \quad n \geq 1$

$0! = 1$

ALGORITHM F(n)

// Computes n! recursively.
// Input : positive integer n.
// Output : value of n!

if (n = 0) return 1.
else
return F(n-1) * n.

time complexity

logical formula

$F(n) = F(n-1) * n \quad n > 0$

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

time complexity of multiplication is 1

$M(n) = M(n-1) + 1 \quad n > 0$

$M(0) = 1 \quad n = 0$

time complexity of n

ANALYSIS.

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

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

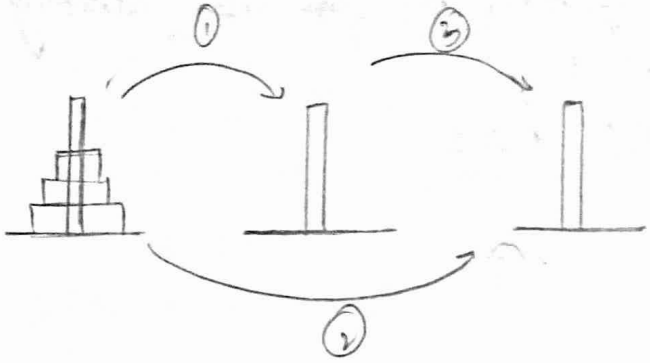
$= [M(n-3) + 1] + 2$

$= [M(n-n)] + n$

$= M(0) + n$

$= 1 + n \approx n$

We have n disc of different sizes and three pegs. Initially all the discs are on the first peg in order of size larger disc on the buttom and smallest on top, goal is to move all the disc to the third peg using second one as auxillary (if necessary). Conditions a) one disc at a time b) can't place larger disc on top of smaller one



Recursive solution is the Tower of Hanoi Puzzle.

To move n>1 disc from 1 to 3 (peg 2). First move n-1 disc from 1 to 2 then move largest disc from 1 to 3. Finally move n-1 disc from 2 to 3 (peg 1). If n=1 then, quickly move single disc from 1 to 3

ALGORITHM Hanoi (disk, source, dest, aux)

if disk = 1 then
move disk from source to destination

else
Hanoi (disk-1, source, aux, dest)
move disk from source to dest (constant time (1))
Hanoi (disk-1, aux, dest, source)
end if
end algorithm

ANALYSIS

M(n) - time required to move disk from source to destination

$M(n) = M(n-1) + 1 + M(n-1)$ $\quad n>1$

$M(1) = 1$ $\quad n=1$

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

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

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

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

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

$$= 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \dots + 1.$$

Replace $i = n-1$

$$= 2^{n-1} M(n-n+1) + 2^i - 1.$$

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

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

$$= 2^n - 1. \approx 2^n$$

Algorithm to find number of binary digits in the binary representation of a positive decimal number (Recursion)

ALGORITHM BinaryRec(n)

if $n=1$ return 1

else

return BinaryRec($\lfloor n/2 \rfloor + 1$)

* Input size n
* No. of addition made by the algorithm → A(n)
* No. of additions made in binary recursion is $\frac{n}{2}$ => $A(n/2)$ + one more addition

ANALYSIS

$$A(n) = \begin{cases} A(\lfloor n/2 \rfloor + 1) & n>1 \\ A(1) = 1 & n=1 \end{cases}$$

$$A(n) = A(n/2) + 1$$

$$n = 2^k$$

$$A(2^k) = A\left(\frac{2^k}{2}\right) + 1$$

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

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

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

$$= A(2^{k-3}) + 3$$

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

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

$$A(2^k) = A(2^0) + k$$

$$= 1 + k.$$

$$n = 2^k.$$

$$k = \log_2 n$$

$$= 1 + \log_2 n.$$

$$\approx \log_2 n.$$