Module 1

1. Introduction:

1.1 What is an Algorithm?

An algorithm is a finite 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.

Every algorithm must satisfy the following criteria:

1) Input: There are zero or more quantities / inputs which are externally supplied.

2) Output: At least one quantity / output is produced, i.e an algorithm produce one or more outputs.

3) Definiteness: Each instruction must be clear and unambiguous.

4) Finiteness: If we trace out the instructions of the algorithm, then for all valid cases the algorithm will terminate after a finite number of steps & operations, an each operation must be finite.

5) Effectiveness: Every instruction must be sufficiently basic that it can in principle be carried out by a person using only a pencil and paper. It is not enough the each operation be definite, but it must be feasible. (Possible to do easily or conveniently)

Algorithm that are definite and effective are also called "Computational Procedures".

- The diagrammatic representation of the algorithm (notion of the algorithm) is shown below:

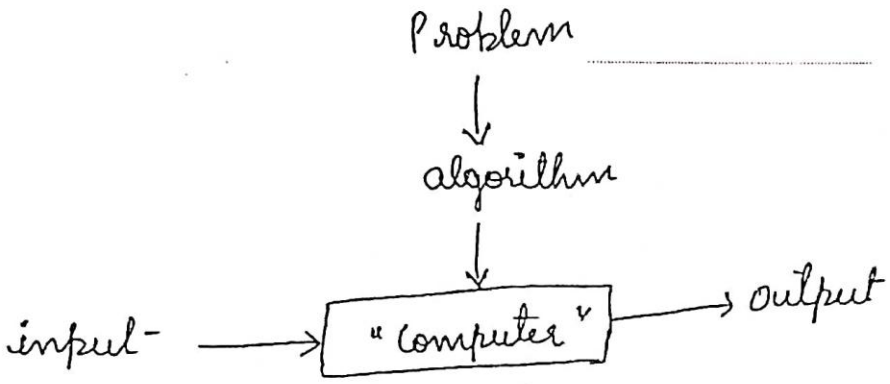


Fig: The notion of the algorithm

- Several other important points/properties of algorithm:
  - 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.
  - There may exist several algorithms for solving the same problem.
  - Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

1.2 Fundamentals of Algorithmic Problem Solving

A sequence of steps one typically goes through in designing and analyzing an algorithm :

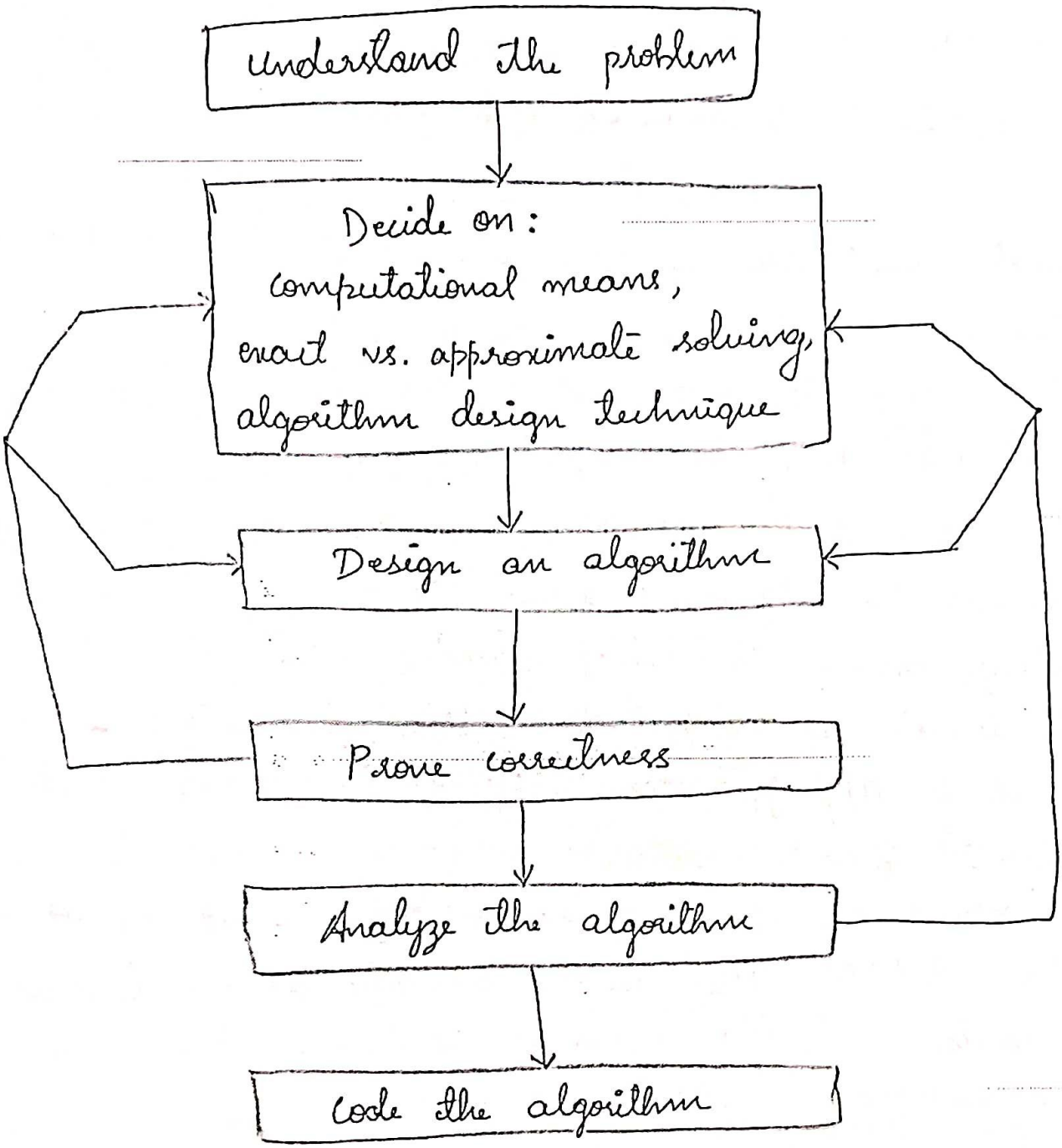


Fig : Algorithm design and analysis process.

We can consider algorithms to be procedural solutions to problems.

Step 1: Understanding the Problem

From a practical perspective, the 1st thing we need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if any doubts about the problem, do a few small examples by hand think about special cases, & ask questions again if needed.

Often we will not find a readily available algorithm and will have to design our own.

An input to an algorithm specifies an instance of the problem the algorithm solves. It is very important to specify exactly the set 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.

A correct algorithm is not one that works most of the time, but one that works correctly for all legitimate inputs.

Step 2: Decide on -

- Computational means

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

The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine.

The essence of this architecture is random-access machine (RAM). Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called Sequential algorithms.

The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e in parallel. Algorithms that take advantage of this capability are called Parallel algorithms.

The speed and memory available on a particular computer system are other factors to consider.

3) Exact vs Approximate Problem Solving

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

An algorithm which solves the problems exactly is called an exact algorithm. (Precise solution)

An algorithm which solves the problem approximately is called an approximation algorithm. (approximate solution)

Why opt for an approximation algorithm?

1) There are important problems that simply cannot be solved exactly for most of their instances.

Eg: evaluating square roots, solving non linear equations, evaluating definite integrals.

2) Available exact algorithms are unacceptably slow, because of the problem's intrinsic complexity.

3) An approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

4) Algorithm Design Techniques

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.

Design an algorithm to solve a given problem.

Algorithm design techniques make it possible to classify algorithms according to an underlying design idea, ∴ they can serve as a natural way to both categorize and study algorithms.

5) Step 3: Design an Algorithm (and Data Structures)

Designing an algorithm for a particular problem maybe a challenging task. Some design techniques can be simply inapplicable to the problem in question. Sometimes, several techniques need to be combined.

- Choosing data structures appropriate for the operations performed by the algorithm is important.

Algorithms + Data Structures = Programs.

In the new world of object-oriented programming, data structures remain crucially important for both design and analysis of algorithms.

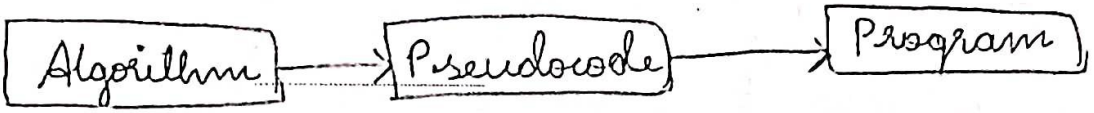
3) Methods of Specifying an Algorithm

Once the algorithm is designed, it should be specified in some fashion. The two options that are most widely used for specifying algorithms: Pseudocode & Flowchart.

Pseudocode is a mixture of a natural language and Programming language-like constructs.

Pseudocode is a step by step description of an algorithm.

Pseudocode is the intermediate state between an idea and its implementation (code) in a high-level language.



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

Flowchart is a pictorial representation of flow of an algorithm.

4) Step 4: Proving an Algorithm's correctness

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

b) Design an algorithm that can handle a range of inputs that is natural for the problem at hand. (Size of inputs it accepts)

Step 6 : Coding an Algorithm

Algorithms ultimately has to be implemented as computer Programs. Programming an algorithm presents both peril and an opportunity.

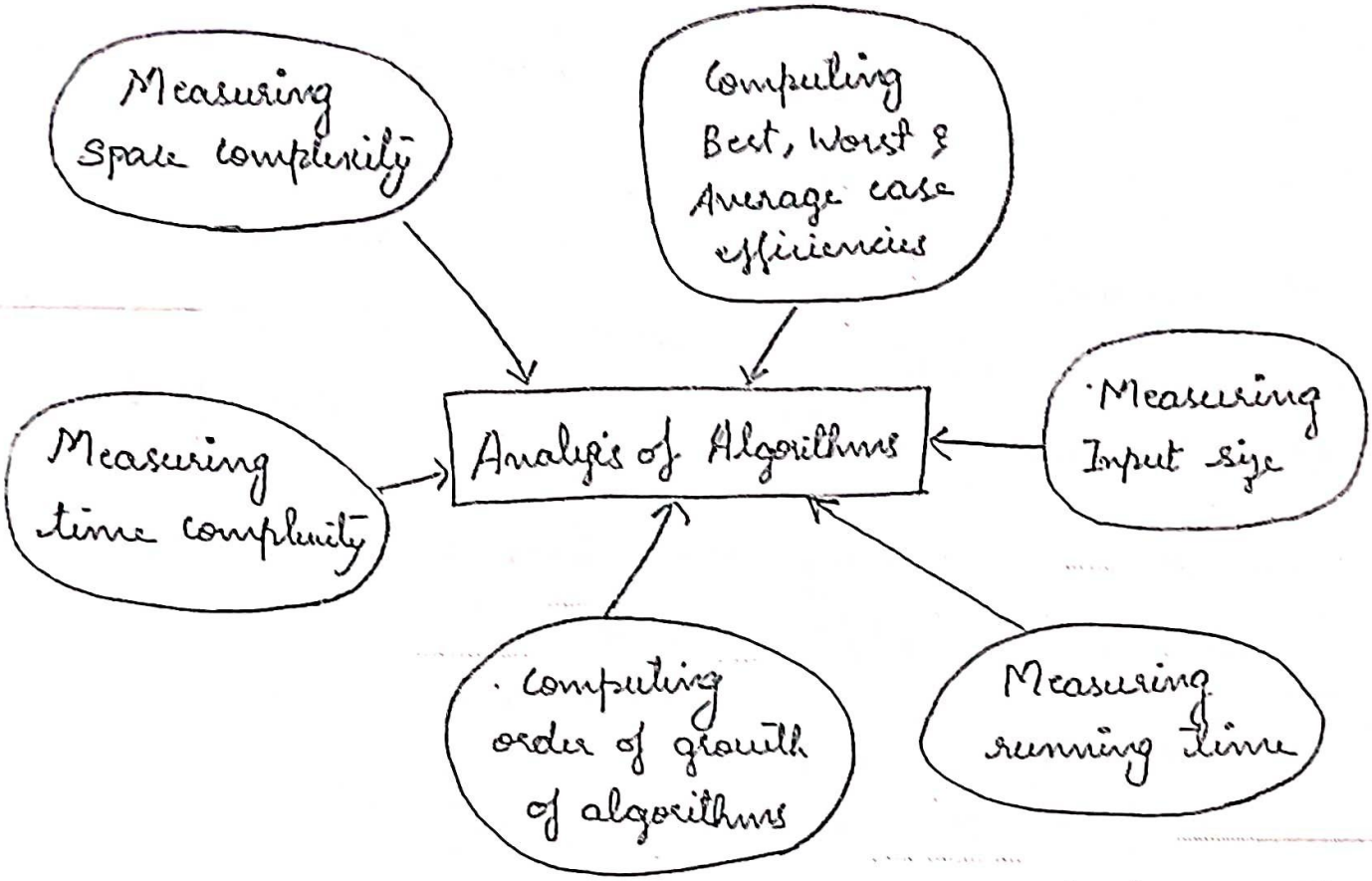
The peril (threats) lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently.

The validity of the program is established by testing and debugging programs, test and debug program thoroughly when an algorithm is implemented.

Code optimization may be required, such improvements can speed up a program (running time).

2.) Fundamentals of the Analysis of Algorithm Efficiency

2.1) Analysis Framework



- Efficiency of an algorithm can be in terms of time or space. Thus, checking whether the algorithm is efficient or not means analyzing the algorithm. There is a systematic approach that has to be applied for analyzing any given algorithm. This systematic approach is modelled by a framework called as Analysis Framework.
- The efficiency of an algorithm can be decided by measuring the performance of an algorithm.

We can measure the performance of an algorithm by computing two factors:

1) Amount of time required by an algorithm to execute
   - time complexity (or) Running time (or) time efficiency
2) Amount of storage required by an algorithm or the amount of memory units required by the algorithm including the memory needed for i/p & o/p.
   - space complexity (or) Memory space (or) space efficiency

The reason for selecting these two criteria are:
- Simplicity - Generality - Speed - Memory

Space Complexity:

- Amount of memory required by an algorithm to run.

To compute the space complexity we use two factors: Constant and instance characteristics

The space requirement S(P) can be given as

$$S(P) = C + S_P$$

where (C) is a constant, i.e fixed part and it denotes the space of inputs and outputs, amount of space taken by instructions, variables & identifiers.

(Sp) is a space dependent upon instance characteristics i.e a variable part, where space requirement depends on particular problem instance. Eg: The control statements

such as for, do, while, choice, switch), Recursion stack for handling recursive call.

For eg: add(a,b)

return a+b

$S(p) = c + Sp$

$= c + 0$ // a, b occupy one word size then total size come to be 2

$= 2 + 0 = \boxed{2}$

Time Complexity

Amount of time required by an algorithm to run to completion.

It is difficult to compute the time complexity in terms of physically clocked time.

For instance in multiuser system, executing time depends on many factors such as:

- System load
- Number of other programs running
- Instruction set used
- Speed of underlying hardware

Therefore, the time complexity is given in terms of frequency count.

The frequency count is a count that denotes how many times a particular statement is executed. (or) count denoting number of times of execution of statement.

Eg: ① Void fun()

```
{ int a;
  a = 10;          ---------- 1
  Printf("%d", a); ---- 1
}
```

∴ The frequency count is 2

Eg ② Void fun()

```
{ int a;
  a = 0;                  ---------- 1          1
  for(i=0 ; i<n; i++)     ---- n+1              1 + n+1 + n
   a = a+i;               ---------- n          n
  Printf("%d", a);        ---------- 1          3+3n
}
```

∴ The frequency count is $1+(n+1)+n+1 = 2n+3$

Note: The for loop in the snippet is executed n times when the condition is true & one more time when the condition is false. Hence the frequency count for 'for' loop is n+1.

Eg ③ Void fun (int a[][], int b[][])

```
{ int c[3][3];
  for (i=0; i<m; i++)                  ---------- m+1        ⇒ 1+m+1+n
  { for (j=0; j<n; j++)                ---------- m(n+1)     ⇒ 1.n + m(n+1)
    { c[i][j] = a[i][j] + b[i][j]; ---- m.n                    + m.n
                                                              ⇒ m.n
```

}

∴ The frequency count is $(m+1) + m(n+1) + m.n$

$= m+1 + mn + m + mn$

$= 2m + 2mn + 1$

$= 2m(1+n) + 1$

$3mn + 2(m+n+1)$

Calculating Sum of n numbers

```
for (i=0; i<n; i++)
{  sum = sum + a[i];
}
```

Statement	Frequency count
i = 0	1
i < n	n+1
i++	n
Sum = sum+a[i]	n
Total	3n + 2

Note: Time Complexity normally denotes in terms of Oh notation (O). Hence if we neglect the constants then we get the time complexity to be O(n).

---

⑤ Matrix addition

```
for(i=0; i<n; i++)
{  for(j=0; j<n; j++)
   {  c[i][j] = a[i][j] + b[i][j];
   }
}
```

Statement	Frequency count
i = 0	1
i < n	n+1
i++	n
j = 0	n * 1 = n (outer loop, initialization of j)
j < n	n * (n+1) = n²+n (outer loop, inner loop)
j++	n * n = n²
c[i][j] = a[i][j] + b[i][j]	n * n = n²
Total	3n² + 4n + 2 ⇒ O(n²)

(2.1.1) Measuring an Input Size

Efficiency measure of an algorithm is directly proportional to the input size or range. So an algorithm efficiency could be measured as a function of n, where n is the parameter indicating the algorithm input size.

For eg, when multiplying two matrices, the efficiency of an algorithm depends on the no of multiplication performed, not on the order of matrices. The i/p given may be a square or a non-square matrix.

- Some algorithm require more than one parameter to indicate the size of their i/p. In such situation, the size is measured by the no of bits in the n's binary representation: $B = floor(\log_2 n + 1)$

Eg: • Sorting • Naive algorithm - n^2 • Best Algorithm - $n \log n$

Algorithms run longer on longer inputs.

(2.1.2) Units for measuring Running time

- The running time of an algorithm usually depends on:
  - Speed of a particular computer (computer used)
  - Quality of a program
  - Compiler used to run the program
- Since we are after a measure of an algorithm's efficiency we like to have a metric that does not depend on extraneous factors as mentioned above.

A possible approach is to count the number of times each of the algorithm's operations is executed.

To measure the algorithm efficiency

- Identify the most important operation (core loop) of the algorithm called the Basic operation, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.
- 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.

Problem Statement	Input Size	Basic operation
1) Searching a key element from the list of n elements	List of n elements	comparison of key with every element of list
2) Perform matrix multiplication	two matrices with order $n \times n$	Actual multiplication of the elements in the matrices
3) Computing GCD of two numbers	two numbers	Division

- Of the 4 arithmetical operations : addition, subtraction, multiplication division, most time consuming operation is division, then multiplication, then addition and subtraction

3) Orders of measuring with A

Let C_{op} be the execution time of an algorithm's basic operation on a particular computer,

Let $C(n)$ be the number 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

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

- $T(n)$: Running time of basic operation
- C_{op}: Time taken by the basic operation to execute
- $C(n)$: Number of times the operation needs to be executed

- Assuming that $C(n) = \frac{1}{2} n(n-1)$, how much longer will the algorithm run if we double its input size?

The answer is about four times longer. Indeed, for all but very small values of n,

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

and therefore

$$\frac{T(2n)}{T(n)} \approx \frac{C_{op} \cdot C(2n)}{C_{op} \cdot C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2} n^2} = 4$$

The efficiency analysis framework ignores multiplicative constants and concentrates on the count's order of growth to within a constant multiple for large-size inputs.

3) Orders of Growth

Measuring the performance of an algorithm in relation with the input size 'n' is called order 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, which contains values of a few functions particularly important for analysis of algorithms.

The magnitude of the numbers in below table has significance for the analysis of algorithms. There are 7 efficiency classes listed

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

Table: Values (some approximate) of several functions important for analysis of algorithms

The function growing the slowest among these is the logarithmic function. It grows so slowly, in fact, we should expect a program implementing an algorithm with a logarithmic basic-operation count to run practically instantaneously on inputs of all realistic sizes.

- Although specific values of such a count depend on the logarithm's base, the formula

$$\log_a n = \log_a b \cdot \log_b n$$

makes it possible to switch from one base to another, leaving the count logarithmic, but with new multiplicative constant.

- This is why we omit a logarithm's base and write simply $\log n$ in situations where we are interested just in a function's order of growth to within a multiplicative constant.

* The exponential function 2^n & the factorial function $n!$, both these functions grow so fast, that their values become astronomically large even for rather small values of n. There is a tremendous difference between the orders of growth of the functions 2^n & $n!$, yet both are often referred to as "exponential-growth functions".

* Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.

* Another way to appreciate the qualitative difference among the orders of growth of the functions in Table is to consider how they react to, say, a two-fold increase in the value of their argument n.

- The function $\log_2 n$ increases in value by just 1

$\because \log_2 2n = \log_2 2 + \log_2 n = 1 + \log_2 n$

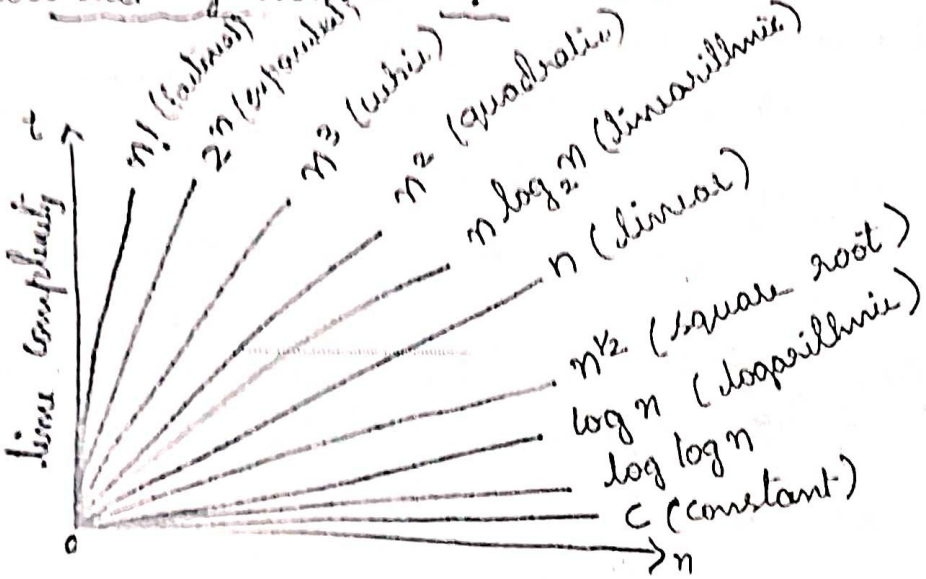
- The linear function increases twofold (n).
- The linearithmic function $n \log_2 n$ increases slightly more than twofold.
- The quadratic function n^2 & cubic function n^3 increases fourfold & eightfold respectively.

$\because (2n)^2 = 4n^2$ & $(2n)^3 = 8n^3$

- The value of 2^n exponential gets squared

$\because 2^{2n} = (2^n)^2$

- The factorial function $n!$ much more than 2^n exponential



- Rate of growth of common computing time function

2.1.4) Worst case, Best case, and Average case Efficiencies

There are many algorithms for which running time depends not only on an input size, but also on the specifics of a particular input.

- The Worst case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n, for which the algorithm runs the

longest among all possible inputs of that size.

If an algorithm takes maximum amount of time to run to completion for a specific set of input, then it is called worst case time complexity.

Eg: While searching an element by using linear searching method, if desired element is placed at the end of the list, then we get worst case time complexity.

$C_{worst}(n) = n$

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

If an algorithm takes minimum amount of time to run to completion for a specific set of input, then it is called best case time complexity.

Eg: While searching an element by using linear search, if the desired element is placed at the first place itself, then we get best case time complexity.

$C_{best}(n) = 1$

The Average case efficiency: neither the worst case analysis nor its best case counterpart yields the necessary information about an algorithm's behaviour on a "typical" or "random" input. To analyze the algorithm's average case efficiency, we must make some assumptions about

Possible inputs of size n.

Eg: Cavg(n) = Probability of successful search + Probability of unsuccessful search

- If P=1 (the search must be successful), the average 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 (the search must be unsuccessful), the average no of key comparisons made will be n, because the algorithm will inspect all n elements on all such inputs.

③ Time Space Tradeoff

Time space tradeoff is basically a situation where either a space efficiency can be achieved at the cost of time or a time efficiency can be achieved at the cost of memory.

(2.2) Asymptotic Notations and Basic Efficiency Classes

- Mathematical notation for determination of the running time (time Complexity) of an algorithm.
- There are notations for determination of order of magnitude of algorithms during priori analysis (before execution of the algorithm), these notations are called as "asymptotic notations. These notations help to make approximate (but meaningful) assumption about the time & space complexity of algorithms.
- To compare algorithms, we need to check efficiency of each algorithm, the efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity. Using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" or "average time".
- Commonly used mathematical asymptotic notations are

1) Big Oh notation 'O'
2) Big Omega notation 'Ω'
3) Theta notation 'θ'

- To compare and rank orders of growth, we use three asymptotic notations : O (big oh), Ω (big omega) and θ (big theta).

Let $t(n)$ and $g(n)$ can be any nonnegative functions defined on the set of natural numbers, $t(n)$ will be an algorithm's running time (indicated by its basic operation count $C(n)$) and $g(n)$ will be some simple function to compare the count with it.

- Informally, $O(g(n))$ is the set of all functions with a lower or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity).
- $\Omega(g(n))$ is the set of all functions with a higher or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity).
- $\Theta(g(n))$ is the set of all functions that have the same order of growth as $g(n)$ (to within a constant multiple) as n goes to infinity).

---

(1) Big oh notation : 'O'

Definition - 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 & some nonnegative integers n_0 (threshold problem size) such that

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

- Big 'O' refers to set of all functions whose growth rate is higher than that of the algorithm's growth rate.

- The Big oh notation refers to the "upper bound" of resources required for solving the problem.
- 'O' is a method of representing the upper bound of algorithm's running time.
- Using 'O' we can give longest amount of time taken by the algorithm to complete.

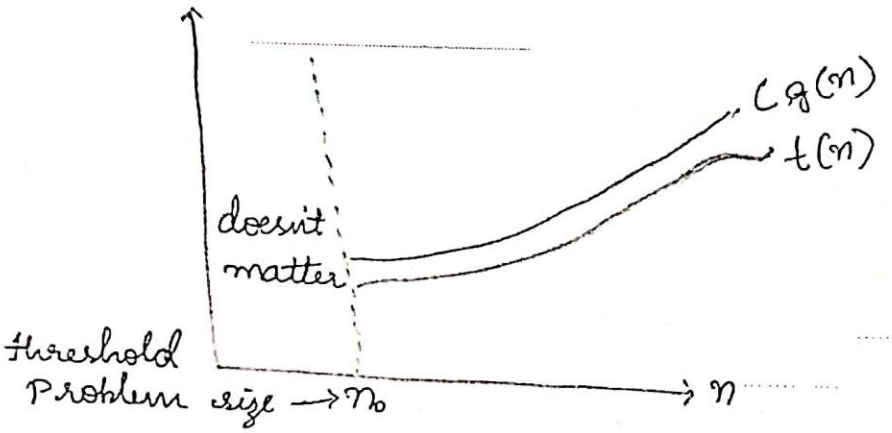


Fig: Big-oh notation: $t(n) \in O(g(n))$ $\forall\ n \geq n_0$
upper Bound of an algorithm

Eg: Consider function $t(n) = 2n+2$ and $g(n) = n^2$
Then we have to find some constant c, so that $t(n) \leq c.g(n)$

Solution: we find c for

$n = 1$, then $t(n) = 2n+2 = 2(1)+2 = 4$
and $g(n) = n^2 = 1^2 = 1$
$\therefore t(n) > cg(n)$

$n = 2$, then $t(n) = 2n+2 = 2.2+2 = 6$
and $g(n) = n^2 = 2^2 = 4$
$\therefore t(n) > c.g(n)$

$n = 3$, then $t(n) = 2n+2 = 2(3)+2 = 8$

and $g(n) = n^2 = 3^2 = 9$

$\therefore t(n) < g(n)$

Hence we can conclude that for $n \geq 2$, we obtain $t(n) < g(n)$

---

(2) Omega notation : 'Ω'

Definition - A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ 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 (threshold problem size) such that

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

- Omega refers to set of all functions whose growth rate is lower than that of the algorithm's growth rate.
- 'Ω' represents the "lower bound" of resources required for solving the problem.
- 'Ω' is a method of representing the lower bound of algorithm's running time.
- Using 'Ω' we can give shortest amount of time taken by the algorithm to complete.

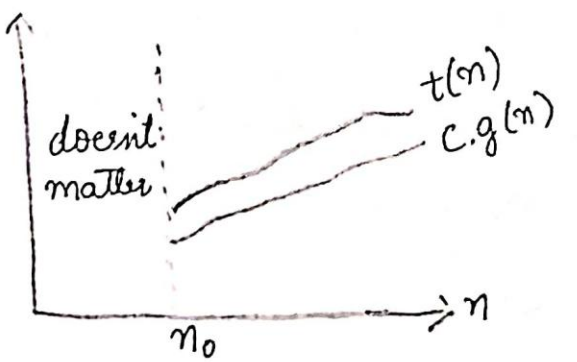


Fig: Big omega notation

$t(n) \in \Omega(g(n))$

- lower bound of an algorithm

Eg: Consider $t(n) = 2n^2+5$ and $g(n) = 7n$

we have to find a constant c, so that $t(n) \geq c.g(n)$

Sol: If $n=0$, $t(0) = 2(0)^2+5 = 5$

$g(0) = 7(0) = 0$

$\therefore t(n) > g(n)$

if $n=1$, $t(1) = 2(1)^2+5 = 7$

$g(1) = 7(1) = 7$

$\therefore t(n) = g(n)$

if $n=2$, $t(2) = 2(2)^2+5 = 13$

$g(2) = 7(2) = 14$

$\therefore t(n) < g(n)$

if $n=3$, $t(3) = 2(3)^2+5 = 23$

$g(3) = 7(3) = 21$

$\therefore t(n) > g(n)$

Hence we can conclude that for $n \geq 3$, we get $t(n) \geq c.g(n)$

It can be represented as $2n^2+5 \in \Omega(n)$

---

3) Theta notation : 'Θ'

Definition - A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ 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 constants c_1 & c_2 & some nonnegative integer n_0 such that

$c_2.g(n) \leq t(n) \leq c_1.g(n)$ for all $n \geq n_0$

C. Theta 'Θ' refers the set of all functions whose growth rate is between upper bound & lower bound of the algorithm growth rate.

. In 'Θ', the running time of the algorithm is between upper bound & lower bound.

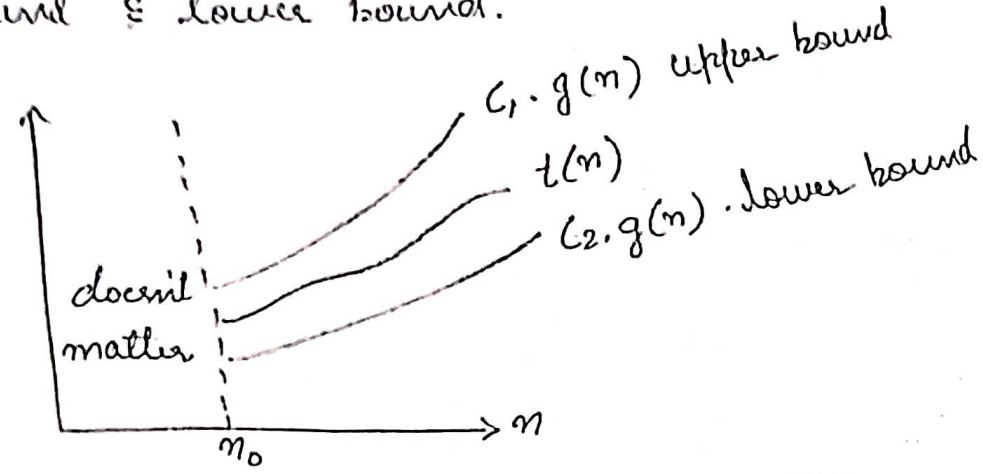


Fig: Big-theta notation : $t(n) \in \Theta(g(n))$

Eg: Consider $t(n) = 2n+8$ & $g_1(n) = 7n$ & $g_2(n) = 5n$

i.e $5n \leq 2n+8 \leq 7n$ for $n = 2$

Here $c_1 = 1$ $c_2 = 1$ with $n_0 = 2$

• Find the Big oh 'O' notation for the following :

• 1) $\log n + \sqrt{n}$

• Finding big oh means finding the upper bound in $\log n$ & $\sqrt{n}$

• upper bound is $n^{1/2}$

• Here in terms of 'O' notation it will be $O(n)$.

2) $n + n \log n$

In this expression $n \log n$ is the upper bound value. Hence it will be $O(n \log n)$

3) $6n + 2n^4 + 4n^5$

By neglecting constants the upper bound of this polynomial in n^5.

Hence the time complexity in terms of 'O' is $O(n^5)$

Note: When we calculate big O

a) only dominant terms matters

e.g. $O(n^4 + n^2 + 64) \Rightarrow O(n^4)$. all terms other than highest degree are ignored

b) Coefficients and constant factors are not significant or ignored or doesn't matter

e.g. $O(3n^5) \Rightarrow O(n^5)$

Basic Efficiency Classes

Classifying algorithms by their asymptotic efficiency

Class (order of growth)	Name (of efficiency class)	Description	Example
1	constant	when its input size grows infinitely large, algorithms running time typically goes to infinity.	1. Scanning array elements 2. Best case of linear search & binary search
log n	logarithmic	When we get logarithmic running time, then it is sure that the algorithm does not consider all its input, rather the problem is divided into smaller parts on each iteration.	1. Performing binary search operation 2. Worst case of binary search
n	linear	The running time of algo depends on the input size n.	1. Performing Sequential search operation 2. Worst case of linear search
n log n	linearithmic	Many divide & conquer algorithms, some instance of input is considered for the list of size n.	1. Sorting the elements using merge sort or quick sort
n^2	quadratic	When the algorithm has two nested loops then this type of efficiency occurs	1. Scanning matrix elements 2. Selection sort, Bubble sort

Class	Name	Description	Example
n^3	cubic	When algorithm has three embedded loops or nested loops, then this efficiency occurs.	1. Performing matrix multiplication 2. Linear algebra
2^n	exponential	When the algorithm has very faster rate of growth, then this efficiency occurs	1. Generating all subsets of an n-element set. 2. Knapsack Problem
$n!$	factorial	When the algorithm is computing all the permutations	1. Generating all Permutations of an n element set 2. Travelling Salesman Problem

• Properties of Asymptotic Notation

Asymptotic Notation	Reflexivity	Transitive	Symmetry
Big oh 'O'	✓ $t(n) = O(t(n))$	✓ if $t(n) = O(g(n))$ & $g(n) = O(h(n))$, then $t(n) = O(h(n))$	✗
Omega 'Ω'	✓ $t(n) = \Omega(t(n))$	✓ if $t(n) = \Omega(g(n))$ & $g(n) = \Omega(h(n))$, then $t(n) = \Omega(h(n))$	✗
Theta 'Θ'	✓ $t(n) = \Theta(t(n))$	✓ if $t(n) = \Theta(g(n))$ & $g(n) = \Theta(h(n))$, then $t(n) = \Theta(h(n))$	✓ $t(n) = \Theta(g(n))$ iff $g(n) = \Theta(t(n))$

Transpose Symmetry $\Rightarrow t(n) = O(g(n))$ iff $g(n) = \Omega(t(n))$

Comparing orders of Growth

1) $\log_2 n$ and $\sqrt{n}$

Sol: For comparison, we will consider various values of n

if $n = 2$ $\quad \log_2 n = \log_2 2 = 1$

$\sqrt{n} = \sqrt{2} = 1.414$

if $n = 64$ $\quad \log_2 n = \log_2 64 = 6$

$\sqrt{n} = \sqrt{64} = 8$

if $n = 256$ $\quad \log_2 256 = 8$

$\sqrt{256} = 16$

$\therefore$ All these assumptions show that $\boxed{\log_2 n < \sqrt{n}}$

---

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

Sol: For comparison, we will consider various values of n

if $n = 2$ $\quad \frac{1}{2} n(n-1) = \frac{1}{2} \cdot 2(2-1) = 1$

$n^2 = 2^2 = 4$

if $n = 4$ $\quad \frac{1}{2} n(n-1) = \frac{1}{2} \cdot 4(4-1) = 6$

$n^2 = 4^2 = 16$

if $n = 8$ $\quad \frac{1}{2} n(n-1) = \frac{1}{2} \cdot 8(8-1) = 28$

$n^2 = 8^2 = 64$

$\therefore$ All these computations indicate that $\boxed{\frac{1}{2} \cdot n(n-1) < n^2}$

Recurrence Relations

A recurrence relation is an equation which is defi... in terms of itself.

Many algorithms, particularly divide and conquer algorithms have time complexities which are naturally modeled by recurrence relations.

The recurrence equation can have infinite number of sequences.

Solving Recurrence Equations :

The recurrence relation can be solved by following methods

1. Substitution method
2. Master's method (Master theorem)

Substitution Method :

It is a kind of method in which a guess for the solution is made.

There are two types of substitutions -

a) Forward Substitution b) Backward Substitution

Forward substitution method -

This method makes use of an initial condition in the initial term & value for the next term is generated. This process is continued until some formula is guessed. Thus in this kind of method, we use recurrence equations to generate the few terms.

Eg: $T(n) = T(n-1) + n$ with initial condition $T(0) = 0$.
Solve it using forward substitution method.

Sol: Let $T(n) = T(n-1) + n$

If $n=1$, then $T(1) = T(0) + 1 = 0+1 = 1$

If $n=2$, then $T(2) = T(1) + 2 = 1+2 = 3$

If $n=3$, then $T(3) = T(2) + 3 = 3+3 = 6$

By observing the above generated equations, we can derive a formula

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

we can also denote $T(n)$ in terms of big oh notation

$$T(n) = O(n^2)$$

But in practise, it is difficult to guess the pattern from forward substitutions. Hence this method is not very often used.

---

b) Backward Substitution Method:

In this method backward values are substituted recursively in order to derive some formula.

Eg 1): Solve $T(n) = T(n-1) + n$ with $T(0) = 0$

Sol: Consider the recurrence relation

$$T(n) = T(n-1) + n \quad \text{---} (1)$$

To find $T(n-1)$

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

$$T(n-1) = T(n-2) + (n-1) \quad \text{---} (2)$$

Substituting eq" (2) in eq" (1)

$T(n) = T(n-2) + (n-1) + n$ ——— (3)

To find $T(n-2) = T(n-3) + (n-2)$ ——— (4)

Substituting eq" (4) in eq" (3)

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

$\vdots$

$= T(n-K) + (n-K+1) + (n-K+2) + \ldots + n$

$T(0)$

If $K = n$ then

$T(n) = T(0) + 1 + 2 + \cdots + n$

$T(n) = 0 + 1 + 2 + \ldots + n$

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

$\therefore T(n)$ in terms of 'O' notation can be represented as $T(n) \in O(n^2)$

---

2) Solve recurrence relation $T(n) = \begin{cases} T(n-1)+1 & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$

Sol: $T(n) = T(n-1)+1$ ——— (1)

$T(n-1) = T(n-2)+1$ ——— (2)

$T(n-2) = T(n-3)+1$ ——— (3)

$\therefore T(n) = \underline{T(n-2)} + 1 + 1$ ⇒ substituting (2) in (1)

$= T(n-2) + 2$

$= \underline{T(n-3)} + 1 + 2$ ⇒ substituting (3) in (2)

$= T(n-3) + 3$

$\vdots$

$= \underbrace{T(n-K)}_{T(0)} + K$

if $K = n$,

$T(n) = T(0) + n = 0 + n = n$

$\therefore T(n) = O(n)$

Solve recurrence relations $T(n) = \begin{cases} 2T(n/2) + c & \text{otherwise} \\ 1 & \text{if } n = 1 \end{cases}$

Sol: $T(n) = 2T\left(\frac{n}{2}\right) + c$ —— (1)

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + c$ —— (2)

$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + c$ —— (3)

By substituting

$T(n) = 2\left(2T\left(\frac{n}{4}\right) + c\right) + c$ ⇒ substituting (2) in (1)

$= 4T\left(\frac{n}{4}\right) + 3c$

$= 4\left(2T\left(\frac{n}{8}\right) + c\right) + 3c$ ⇒ (3) in (2)

$= 8T\left(\frac{n}{8}\right) + 7c$

$= 2^3 T\left(\frac{n}{2^3}\right) + \left(2^3 - 1\right)c$

$= 2^K T\left(\frac{n}{2^K}\right) + \left(2^K - 1\right)c$

If we put $2^K = n$, then

$T(n) = 2^K T\left(\frac{2^K}{2^K}\right) + \left(2^K - 1\right)c$

$T(n) = nT\left(\frac{n}{n}\right) + (n-1)c$

$= n(T(1)) + (n-1)c$

$= n \cdot 1 + (n-1)c$

$T(n) = n + (n-1)c$

$T(n) = T\left(\frac{n}{3}\right) + c \qquad T(1) = 1$

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T\left(\frac{n}{3}\right) + c & \text{otherwise} \end{cases}$$

$T(n) = T\left(\frac{n}{3}\right) + c$

$T(n/3) = T(n/9) + c$

$T(n/9) = T(n/27) + c$

By substitution

$T(n) = T(n/9) + c + c$

$= T(n/27) + c + c + c$

$= T(n/27) + 3c$

$= T(n/3^3) + 3c$

$\vdots$

$= T(n/3^k) + kc$

If we put $3^k = n$, then

$T(n) = T(n/n) + kc$

$= T(1) + \log_3^n c$

$T(n) = 1 + c.\log_3^n$

---

1) $T(n) = 4T(n/3) + n^2$ for $n \geq 2$ & $T(1) = 1$
2) $T(n) = 2T(n/2) + n$
3) $T(n) = T(n/2) + 1$

Mathematical Analysis of Non-Recursive Algorithms

General Plan for Analyzing the Time Efficiency of Non-Recursive Algorithms:

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, then the worst case, average case, and if necessary, best case efficiencies have to be investigated / analyzed separately.
4. Set up a sum for expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas & rules of sum manipulation, either find a closed form formula for the count or, at the very least, establish its order of growth.

Sum manipulation Rules:

1. $\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i$

2. $\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$

3. $\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i$, where $l \leq m \leq u$

4. $\sum_{i=l}^{u} (a_i - a_{i-1}) = a_u - a_{l-1}$

Important Summation Formulas :

1. $\sum_{i=l}^{u} 1 = \underbrace{1+1+\cdots+1}_{u-l+1 \text{ times}} = u-l+1$ (l, u are int limits $l \le u$)

$\sum_{i=1}^{n} 1 = n$

2. $\sum_{i=1}^{n} i = 1+2+\cdots+n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in O(n^2)$

3. $\sum_{i=1}^{n} i^2 = 1^2+2^2+\cdots+n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$

4. $\sum_{i=1}^{n} i^K = 1^K+2^K+\cdots+n^K \approx \frac{1}{K+1}n^{K+1}$

5. $\sum_{i=0}^{n} a^i = 1+a+\cdots+a^n = \frac{a^{n+1}-1}{a-1}$ $(a \neq 1)$; $\sum_{i=0}^{n} 2^i = 2^{n+1}-1$

6. $\sum_{i=1}^{n} i\,2^i = 1.2 + 2.2^2 + \cdots + n.2^n = (n-1)\,2^{n+1} + 2$

7. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + \gamma$, where $\gamma \approx 0.5772$ (Euler's constant)

8. $\sum_{i=1}^{n} \lg i \approx n \lg n$

1): Finding the value of the largest element in a list of n number. (array)

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
```

Mathematical Analysis:

Step 1: The input size is n. i.e total number of element in array..

Step 2: The basic operation is comparison in loop for finding largest value.

Step 3: The comparison is executed on each repetition of the loop. as the comparison is made for each value of n, there is no need to find best case, worst case & average case analysis.

Step 4: Let C(n) be the number of times the comparison is executed. The algorithm makes comparison each time the loop executes. That means with each new

value of i the comparison is made.

Hence for $i = 1$ to $n-1$ times the comparison is made

$\therefore$ we can formulate $C(n)$ as.

$C(n)$ = "One comparison made for each value of i"

Step 5 : Let us simplify the sum

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

$$c(n) = n-1 \in \Theta(n) \quad \text{using the rule } \sum_{i=1}^{n} 1 = n \in \Theta(n)$$

Thus the efficiency of above algorithm is $\Theta(n)$.

Ex 2 : Element uniqueness problem. Check whether all the elements in a given array of n elements are distinct. This problem can be solved by the steps.

Algorithm UniqueElements $(A[0...n-1])$

// Determines 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
```

- Mathematical Analysis :

Step 1 : The input size is n.

Step 2: The basic operation is the comparison operation of two elements. (if $A[i] = A[j]$)

Step 3: The number of comparisons (basic operation) will depend not only on 'n', but also on whether there are equal elements in the array or not. ∴ we limit the process to worst case only.

Step 4: The worst case is denoted by $C_{worst}(n)$. But there are two types of worst case inputs.

(i) When there are no equal elements in the array

(ii) The last two elements are equal in the array.

$$C_{worst}(n) = \text{outer loop} \times \text{inner loop}$$

$$= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

Step 5: Simplifying $C_{worst}(n)$ as follows:

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

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

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

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

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

$$C_{worst}(n) = \frac{n(n-1)}{2} \qquad \frac{1}{2} \cdot n^2$$

i.e $\frac{1}{2} n^2 \in \Theta(n^2)$

Eg 3: Matrix Multiplication

```
Algorithm MatrixMulti(A[0..n-1, 0..n-1], B[0..n-1, 0..
// Multiplies two square matrices of order n
// Input : Two n x n matrices A and B
// Output : Matrix C = A.B
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
```

. Mathematical Analysis :

Step 1 : The input size of above algorithm is simply order of matrices, i.e n

Step 2 : The basic operation is in the innermost loop & which is

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

we should note that in this basic operation both addition & multiplication are performed. But we will not choose any one of them as basic operation because on each repetition of innermost loop, each of the two will be executed exactly once. So by counting one automatically other will be counted. Hence we consider multiplication as a basic operation.

Step 3 : The basic operation depends only upon input size. There are no best case, worst case & average case efficiencies.

Step 4: Now we will compute sum. There is just one multiplication which is repeated on each execution of innermost loop (a for loop using variable K).

Hence we will compute the efficiency for innermost loop

$$\sum_{K=0}^{n-1} 1$$

Step 5: The sum can be denoted by $M(n) \rightarrow$ total number of multiplications

$M(n)$ = outerloop × innerloop × innermost loop

$$= \left[\begin{matrix}\text{for loop}\\ \text{using } i\end{matrix}\right] \times \left[\begin{matrix}\text{for loop}\\ \text{using } j\end{matrix}\right] \times \left[\begin{matrix}\text{for loop}\\ \text{using } K\end{matrix}\right] \left[1 \text{ execution}\right]$$

$$= \sum_{j=0}^{n-1} \sum_{j=0}^{n-1} \sum_{K=0}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n$$

$$= \sum_{i=0}^{n-1} n^2 \qquad \because \sum_{j=1}^{n-1} = n$$

$$M(n) = \underline{n^3}$$

Thus the simplified sum is n^3. Thus the time complexity of matrix multiplication is $\underline{\Theta(n^3)}$

(Eg 4). Find the number of binary digits in the binary representation of a positive decimal number.

Counting number of bits in an integer

Algorithm Binary(n)

// Input : A positive decimal integer n

Output: Returns the number of binary digits in n's binary representation

```
count ← 1
while n > 1 do
    count ← count + 1
    n ← [n/2]
return count
```

Mathematical Analysis:

Step 1: The input size is n, i.e the +ve integer whose binary digits in binary representation needs to be checked.

Step 2: The basic operation is denoted by while loop and it is each time checking whether $n > 1$. The while loop will be executed for the number of time at which $n > 1$ is True. It will be executed one more when $n > 1$ is false. But when $n > 1$ is false, the statements inside while loop wont get executed.

Step 3: The value of n is halved on each repetition of the loop. Hence efficiency of algorithm is equal to $\log_2 n$.

Step 4: Hence total number of times the while loop gets executed is $\lfloor \log_2 n \rfloor + 1$. (Floor)

Hence time complexity for counting number of bits of given number is $\Theta(\log_2 n)$.

Note: The $\lfloor \rfloor$ indicates floor value of $\log_2 n$

2.4 Mathematical Analysis of Recursive Algorithms

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size
2. Identify the algorithm's basic operation
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, then the worst-case, average case and best case efficiencies must be investigated separately.
4. Setup a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

While solving the recurrence we will use the forward and backward substitution method. And then correctness of formula can be proved with the help of mathematical induction method.

Eg 1: Computing factorial of some number n

Algorithm Factorial(n)
// Computes $n!$ recursively
// Input : A nonnegative integer n
// Output : The value of $n!$
if $n == 0$ return 1
else return Factorial($n-1$) * n

Mathematical Analysis:

Step 1: The factorial algorithm works for input size n

Step 2: The basic operation in computing factorial is multiplication.

Step 3: The recursive function call can be formulated as

$F(n) = F(n-1) * n$ where $n > 0$

Then the basic operation multiplication is given as $M(n)$.

$M(n)$ is multiplication count to compute factorial (n).

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

↑ These multiplications are required to compute factorial $(n-1)$ ↖ To multiply factorial $(n-1)$ by n

Step 4: In step 3 the recurrence relation obtained is

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

Initial condition: $M(0) = 0$

Initial condition is obtained by looking at the condition that makes the recursive call to stop.

- The Recurrence relation $M(n)$ is given by:

$$M(n) = \begin{cases} 0 & \text{if } n = 0 \\ M(n-1) + 1 & \text{otherwise} \end{cases}$$

- Backward Substitution:

$M(n) = M(n-1) + 1$ ——(1)

$M(n-1) = M(n-2) + 2$ ——(2)

$M(n-2) = M(n-3) + 3$ ——— (3)

From the substitution methods we can establish a general formula as:

$$\boxed{M(n) = M(n-i) + i}$$

Now let us prove correctness of this formula using mathematical induction as follows:

From the initial condition, i.e $n=0$, substitute $i = n$ in the above equation to get

$$M(n) = M(n-i) + i$$
$$= M(n-n) + n$$
$$= M(0) + n$$
$$= 0 + n$$
$$\boxed{M(n) = n}$$

Thus the time complexity of factorial function is $\underline{\Theta(n)}$

Eg 2. Towers of Hanoi

We have n disks of different sizes and three pegs. Initially all the disks are on the first peg, such that the largest is at the bottom and smallest is on the top.

We have to move all the disks to the third peg using second one as auxiliary.

- We have to move only one disk at a time.
- It is forbidden to place a larger disk on top of a smaller one.

This problem can be solved by recursive technique.

Algorithm TOH(n, A, C, B)

// Moving n disks from source to destination using auxiliary peg

// Input : n disks & 3 pegs

// Output : Moving n disks from source to destination such that largest is at the bottom & smallest is on top.

if (n = 1) then

write ("The peg is moved from A to C")

return

else

TOH(n-1, A, B, C)

move disk from source to destination

TOH(n-1, B, C, A)

- Mathematical Analysis

Step 1 : The input parameter is n, i.e the total number of disks.

Step 2 : The basic operation of the algorithm is moving disks from one peg to another.

If n = 1, then we simply move the disk from Peg A to Peg C.

Step 3 : The number of disk movements depends only on no of disks we have to move, i.e n

Step 4 : The recurrence relation for basic operation count is given by

$M(n) = M(n-1) + 1 + M(n-1)$ for $n > 0$

- $M(n-1)$: to move $(n-1)$ disks from Peg A to B
- 1: To move the largest disk from Peg A to C
- $M(n-1)$: To move $(n-1)$ disks from Peg B to C

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

Initial condition: $M(1) = 1$

Step 5: Solving the recurrence relation using backward substitution method -

$M(n) = 2M(n-1) + 1$ —— (1)

$M(n-1) = 2M(n-2) + 1$ —— (2)

$M(n-2) = 2M(n-3) + 1$ —— (3)

Substituting the above equations

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

$= 4M(n-2) + 3$

$= 4(2M(n-3) + 1) + 3$

$= 8M(n-3) + 7$

$\vdots$

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

Put $i = n-1$

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

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

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

$\therefore \boxed{M(n) \in \Theta(2^n)}$

(3) Computing the sum of first n cubes

$$S(n) = 1^3 + 2^3 + 3^3 + \dots + n^3$$

Algorithm Sum(n)

// Computing the sum of first n cubes

// Input : The value of n

// Output : returns the sum of first n cubes

if (n=1) then return 1

else return (S(n-1) + n * n * n)

Mathematical Analysis :

Step 1 : The input size is n, an integer

Step 2 : The basic operation is multiplication

Step 3 : The number of times the basic operation performed is only dependant on input n.

Step 4 : The recurrence relation is

$$M(n) = M(n-1) + \underbrace{1+1+1}_{\text{To multiply } S(n-1) \text{ by } n*n*n} \quad \text{for } n>1$$

$M(n-1)$: To compute $S(n-1)$

Step 4 : Recurrence relation obtained is $M(n) = M(n-1) + 3$

Initial condition : $M(1) = 0$

$$M(n) = \begin{cases} 0 & \text{if } n=0 \\ M(n-1)+3 & \text{Otherwise} \end{cases}$$

Backward Substitution :

$$M(n) = M(n-1) + 3 \quad \text{---(1)}$$

$$M(n-1) = M(n-2) + 3 \quad \text{---(2)}$$

$$M(n-2) = M(n-3) + 3 \quad \text{---(3)}$$

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

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

$= M(n-2) + 6$

$= M(n-3) + 3 + 6$

$= M(n-3) + 9$

$\vdots$

$= M(n-i) + 3 * i$

Put $i = n-1$

$M(n) = M(n-n-1) + (3 * n-1)$

$= M(1) * 3n - 3$

$M(n) = 0 + 3n - 3$

$\therefore M(n) \in O(n)$

Eg 4). Find number of binary digits in n's binary representation in a recursive version.

Algorithm BinRec(n)

// Counting the binary digits from a decimal integer

// Input : The decimal integer n

// Output : The number of binary digits in n's binary representation

```
if n == 1
    return 1
else
    return BinRec(⌊n/2⌋) + 1
```

Mathematical Analysis :

Step 1 : The input size n, a +ve decimal number

Step 2: The basic operation is division by 2 and addition

Step 3: We will set up recurrence relation.

Let $A(n)$ be the number of times basic operation gets executed.

when $n=1$, then there is no division operation performed

$\therefore A(1) = 0$

when $n>1$, then BinRec(n) makes a recursive call with $\lfloor n/2 \rfloor$

$$A(n) = A(\lfloor n/2 \rfloor) + 1$$

↙ To compute BinRec $\lfloor n/2 \rfloor$ ↘ To compute $\lfloor n/2 \rfloor$

$\therefore$ The recurrence relation is

$$A(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ A\lfloor n/2 \rfloor + 1 & \text{otherwise} \end{cases}$$

Step 4: Solving recurrence for n to be powers of 2, we can compute the efficiency of an algorithm

Using substitution method:

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{---} (1)$$

$$A(n/2) = A(\lfloor n/4 \rfloor) + 1 \quad \text{---} (2)$$

$$A(n/4) = A(\lfloor n/8 \rfloor) + 1 \quad \text{---} (3)$$

$$A(n) = A(\lfloor n/2 \rfloor) + 1$$

$$= A(\lfloor n/4 \rfloor + 1) + 1$$

$= A(\lfloor n/4 \rfloor) + 2$

$= A(\lfloor n/5 \rfloor + 1) + 2$

$= A(\lfloor n/5 \rfloor) + 3$

$\vdots$

$= A(\lfloor n/2^k \rfloor) + k$

Put $2^k = n$

$= A(\lfloor n/n \rfloor) + \log_2 n$

$= A(1) + \log n$

$= 0 + \log n$

$A(n) = \log n$

$\therefore A(n) \in \Theta(\log n)$

Algorithm Design Techniques

Brute force

Greedy technique

Divide and conquer

Decrease and conquer

Transform and conquer

Dynamic Programming

Backtracking

Branch and Bound

Space and time tradeoffs

Important Problem Types

Sorting - Stable, Inplace

Searching

Numerical problems

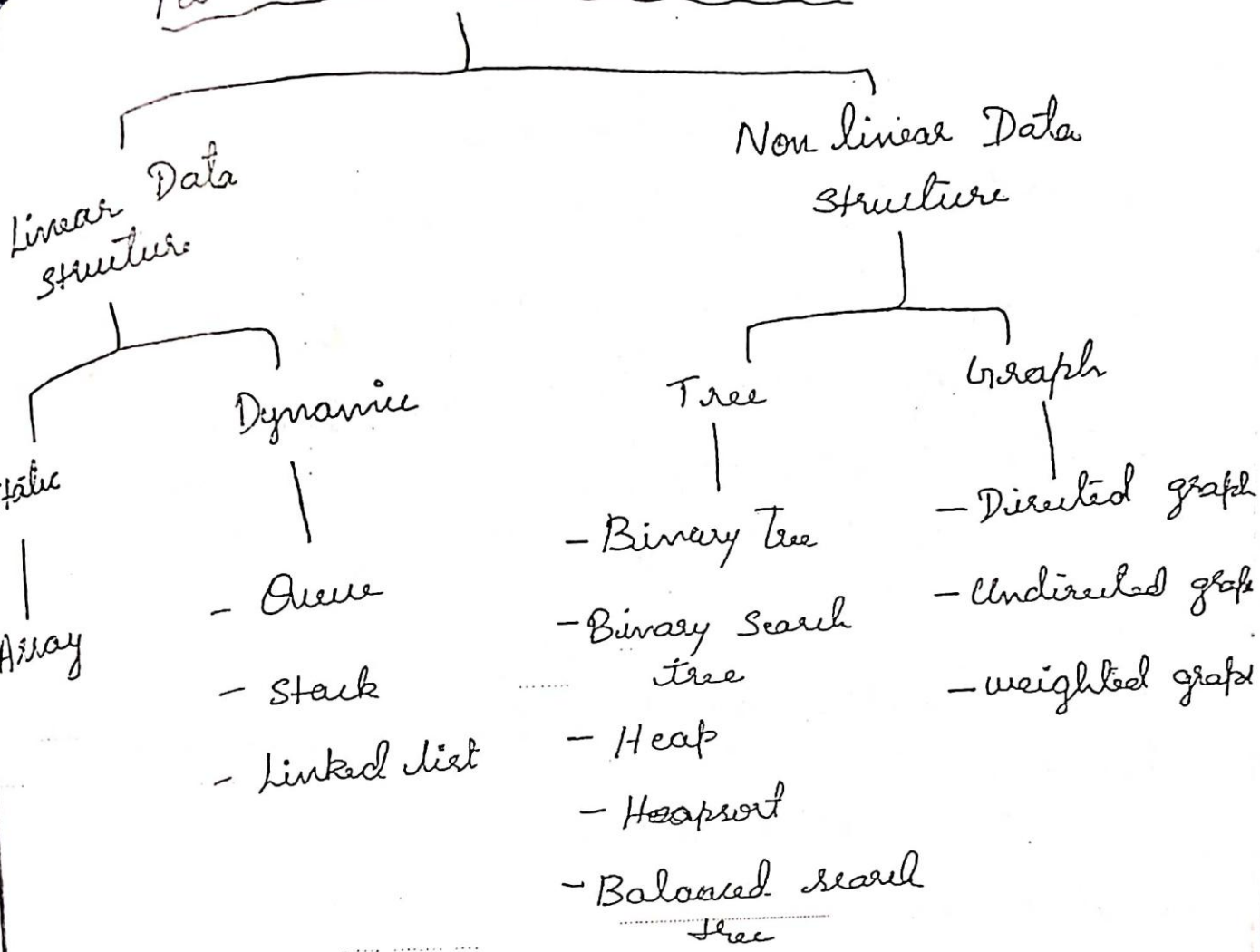
Geometric problems

Combinatorial Problems - Permutations & Combinations

Graph problems - Graph traversal, shortest path

String Processing Problems

Fundamental Data Structures



3. Brute Force Approaches:

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

Brute force approach finds all the possible solutions to find a satisfactory solution to a given problem. It is mainly used for solving simple and small problems. Brute force techniques are insufficient for large real issues.

Some of the algorithms based on Brute force principle:

- Selection Sort
- Bubble Sort
- Sequential search
- Brute force String matching
- Exhaustive Search (Travelling Salesman problem and knapsack problem)

3.1 Selection Sort and Bubble Sort:

Selection Sort:

Selection Sort starts by scanning the entire given list of data items to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Now the first data item is sorted.

Then it scans the list, starting with the second element

find the smallest among the last n-1 elements and exchange it with the second element, putting the second smallest element in its final position. Now the second element is sorted.

The above steps are repeated n-1 times, At the end of n-1 time, the entire data set is sorted

$A_0 \le A_1 \le \ldots \le A_{i-1}$ | $A_i, \ldots, A_{min}, \ldots, A_{n-1}$

(sorted) in their final position — the last n-i elements (unsorted)

Algorithm SelectionSort(A[0...n-1])

// Sorts a given array by selection sort

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

// Output: Array A[0..n-1] sorted in increasing order

for i ← 0 to n-2 do

  min ← i

  for j ← i+1 to n-1 do

    if A[j] < A[min]

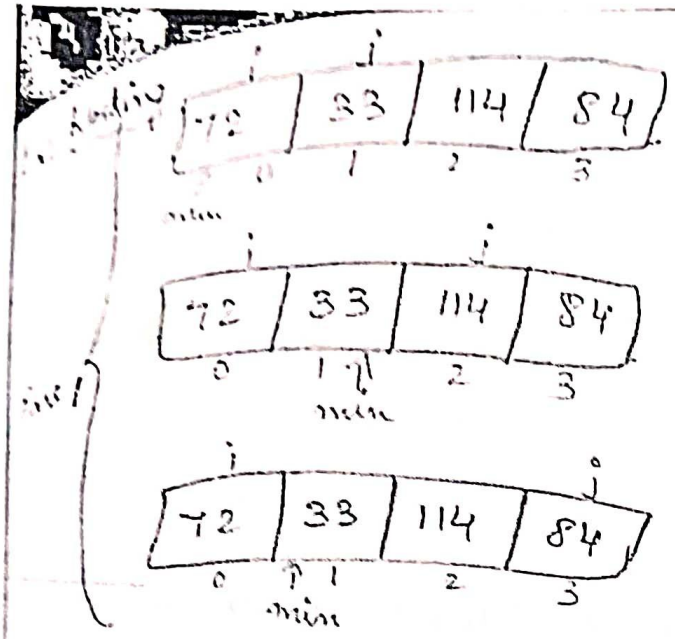
      min ← j

  swap A[i] and A[min]

The selection sort algorithm repeatedly selects the smallest element from the unsorted portion of the list & swaps it with the first element of the ~~unsorted~~ unsorted part.

This process is repeated for the remaining unsorted portion until the entire list is sorted.

The sorted part is to the left, unsorted part is to the right.



if A[j] < A[min] ?

33 < 72 ? Yes

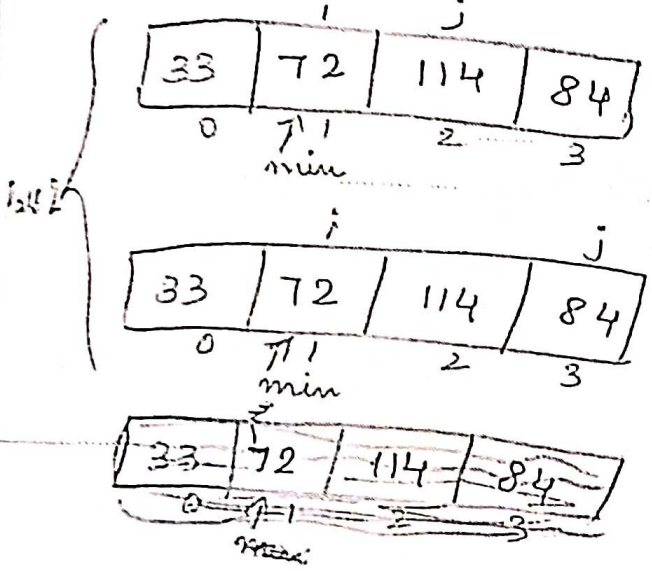
min = j

114 < 33 ? No

84 < 33 ? No

j loop is done once, then swap A[i] and A[min]

72 33

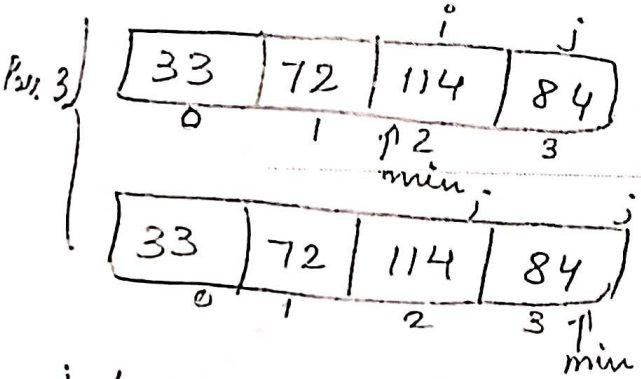


114 < 72 ? No

84 < 72 ? No

j loop is done twice, then swap A[i] and A[min]

72 72

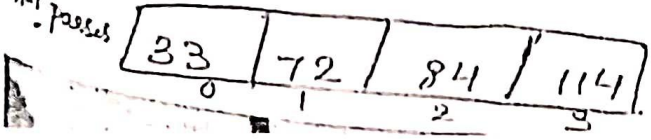


84 < 114 ? Yes

min = j

j loop is done thrice, then swap A[i] and A[min]

114 84



Analysis of Selection sort algorithm:

Input: The input size is given by the number of elements n.

Basic operation: key comparison $A[j] < A[min]$

Basic operation count: The number of times the basic operation is executed depends only on the array size and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] \quad \text{(upper bound - lowerbound + 1)}$$

$$= \sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + (n-3) + \cdots + 2 + 1$$

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

Thus, Selection sort is $O(n^2)$ algorithm on all inputs (Best, worst, Average).

- Best case complexity - no sorting required, the array is already sorted. (10 20 30 40) Ascending order
- Worst case complexity - array elements are required to be sorted in reverse order. (40 30 20 10) Descending order
- Average case complexity - array elements are in jumbled order (nor ascending neither descending) (20 10 40 30)

Bubble Sort:

Bubble sort compares adjacent elements of the list (data items) and exchange them if they are out of order (or wrong order or not in intented order). By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after n-1 passes the list is sorted.

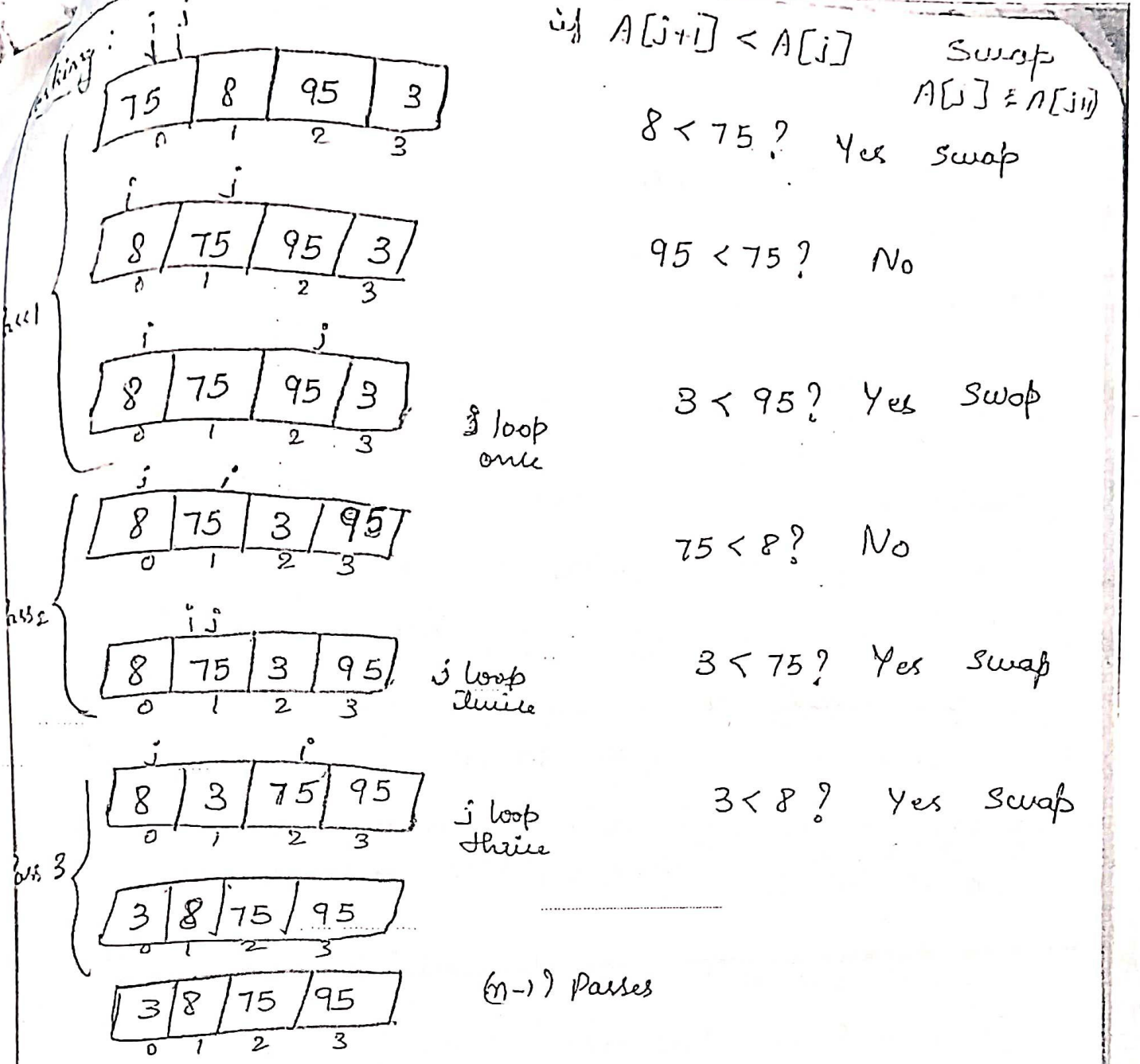
Pass i ($0 \le i \le n-2$) of bubble sort can be represented as:

$$\underbrace{A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1}}_{\text{unsorted}} \mid \underbrace{A_{n-i} \le \ldots \le A_{n-1}}_{\substack{\text{sorted} \\ \text{in their final positions}}}$$

Algorithm BubbleSort ($A[0..n-1]$)
// Sorts a given array by bubble sort
// Input: An array $A[0..n-1]$ of orderable elements
// Output: Array $A[0..n-1]$ sorted in increasing order

```
for i ← 0 to n-2 do
    for j ← 0 to n-2-i do
        if A[j+1] < A[j]
            Swap A[j] and A[j+1]
```

In Bubble sort, there are 'n' data items to be sorted and the main objective is to move the larger items towards the end of the list. This is repeated till the entire list is sorted.



Analysis of Bubble Sort Algorithm:

Input: The input size is given by the number of elements n.

Basic operation: Key comparison $A[j+1] < A[j]$

Basic operation count: The number of times the basic operation is executed depends only on the input array size & is given by the following Sum.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

$$= \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$

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

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

Thus, Bubble sort is $O(n^2)$ algorithm on all inputs. Worst, Avg

- Best case complexity - no sorting required, the array is already sorted (10, 20, 30, 40) Ascending order $O(n)$
- Worst case complexity - array elements are required to be sorted in reverse order (40, 30, 20, 10) Descending order.
- Average case complexity - array elements are in jumbled order (20 10 40 30) $O(n^2)$

Sequential Search and Brute Force String Matching

Sequential Search (linear search)

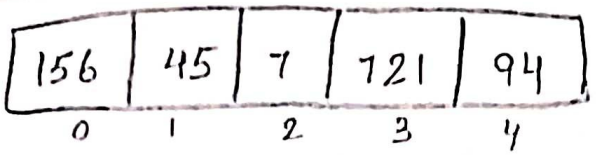
A sequential search algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search).

This is the most basic search technique. In this type of search, we go through the entire list and try to find a match for a single element (Key element). If we find a match, then the address of the matching target is returned.

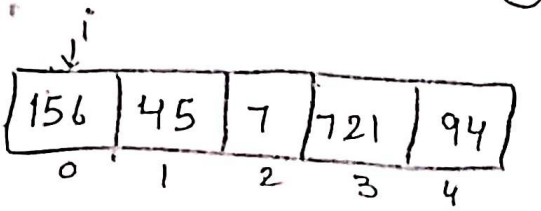
Algorithm SequentialSearch(A[0..n], K)
// Implements sequential search with a search key as a sentinel
// Input: An array A of n elements & a search key K
// Output: The index of the 1st element in A[0...n-1] whose value is equal to K or -1 if no such element is found

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

Looking:

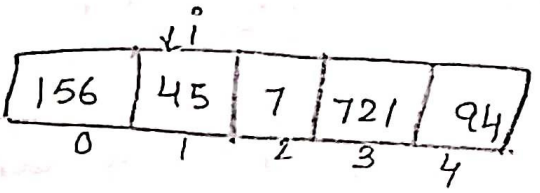


Data / element to be searched is (7)

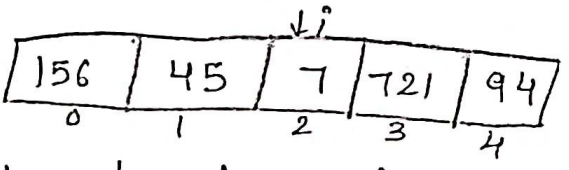
1 

$A[i] \neq K$ $i \leftarrow i+1$

156 = 7? No

2 

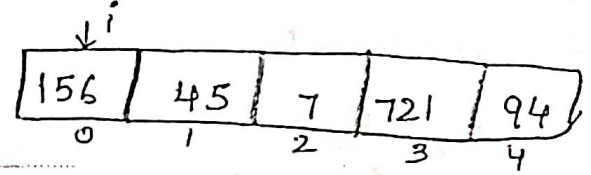
45 = 7? No

3 

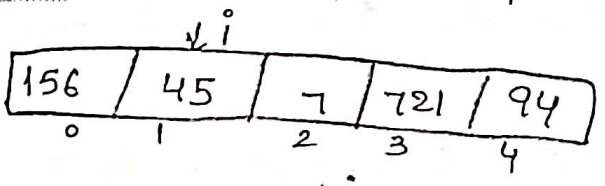
7 = 7? Yes

element found at position 2.

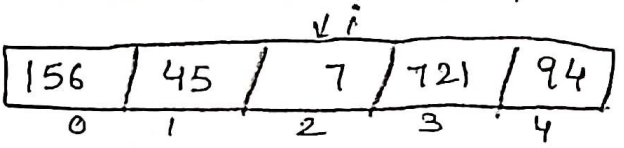
5) Data / element to be searched is (14)

1 

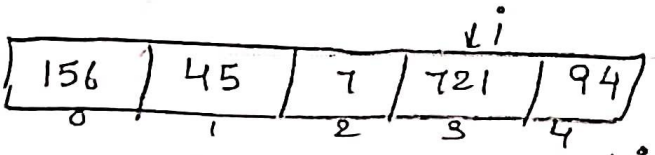
156 = 14? No

2 

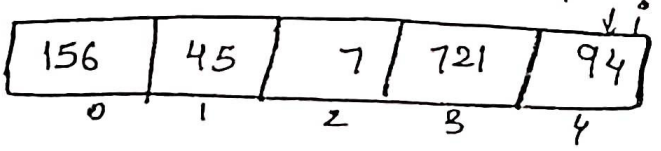
45 = 14? No

3 

7 = 14? No

4 

721 = 14? No

5 

94 = 14? No

Now the end of the list is reached. There are no more elements in the list. So the item 14 is not found.

Analysis of sequential search algorithm:

Input size: The input size is given by the number of elements n.

Basic operation: Key comparison $A[i] \neq K$

Worst operation count:

Worst case: The number to be searched is present as the last element or not present at all. The algorithm makes the largest number of key comparisons among all possible inputs of size n.

$C_{worst}(n) = n$

$T(n) = O(n+K) = \boxed{O(n)}$

Best case: The number to be searched is present as the first element in the given set of numbers.

$C_{best}(n) = 1$ (only one comparison)

$T(n) = \boxed{O(1)}$

Average case: Each number in the array (and the number not in the array) is equally likely to be the number being searched.

$C_{avg.}(n) =$ Probability of successful search + Probability of unsuccessful search

$T(n) = O(n/2) = \boxed{O(n)}$

2) Brute-Force String Matching:

A string is a sequence of characters, given a string of 'n' characters called the text & a string of 'm' characters ($m \leq n$) called the pattern, find a substring of the text that matches the pattern. Find i, the index of the leftmost character of the first matching substring in the text, such that

$t_0 \ldots t_i \ldots t_{i+j} \ldots t_{i+m-1} \ldots t_{n-1}$ Text T

$\updownarrow \quad \updownarrow \quad \updownarrow$

$P_0 \ldots P_j \ldots P_{m-1}$ Pattern P

Brute force algorithm for string matching align the pattern against the first m characters of the text & start matching the corresponding pairs of characters from left to right until either all the m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered, then shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern & its counterpart in the text.

The simplest algorithm, where we simply try to match the first character of the pattern with the first character of the text, and if it succeeds, try to match the 2nd character and so on; if we hit a failure point, slide the pattern over one character and try again. When we find a match, return its starting location.

Problem: BruteForceStringMatch(T[0..n-1], P[0..m-1])

Implements brute-force string matching

Input: An array T[0..n-1] of n characters: text and an array P[0..m-1] of m characters: pattern.

Output: The index of the first character in the text that starts a matching substring or -1 if the search is unsuccessful

```
for i ← 0 to n-m do
    j ← 0
    while j < m and P[j] = T[i+j] do
        j ← j+1
    if j = m return i
return -1
```

The time complexity of the algorithm is $O(m \times n)$, where n is the length of the text & m is the length of the pattern. $\boxed{O(n.m)}$

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13
NOBODY_NOTICED   - text      NOT - pattern
NOT
 NOT
  NOT
   NOT
    NOT
     NOT
      NOF
       NOT
        NOT
```

index 7 is returned