DATA STRUCTURE AND ALGORITHM

A data structure is an organization of the data to solve a problem in such a way that data can be access efficiently by a program. The choice of particular data structure depends upon the following considerations.

- It must be able to represent the inherent relationship of data in the real world.
- It must be simple enough so that it can be processed efficiently as and when necessary.

Data Structure

Linear Data Structure
- Arrays
- Linked list
- Stacks
- Queues

Non-Linear Data Structure
- Trees
- Graphs
- Tables
- Sets

Linear Data Structure: The elements form a sequence. For example: Arrays, Stacks, queues, linked list.

Non-Linear Data Structure: The elements doesn’t form a sequence. For example: Tree, Graph.

Algorithm: Outline, the essence of a computational procedure step by step instruction.

Program: An implementation of an algorithm in some programming language.

Every algorithm must satisfy the following criteria.

- **Input**: There are zero or more values which are externally supplied.
- **Output**: At least one value is produced.
- **Definition**: Each step must be clear and unambiguous.
- **Finiteness**: If we trace the steps of an algorithm. Then for all the cases, algorithm must terminate after a finite number of steps.
- **Effectiveness**: Each step must be sufficiently basic that it can in principle be carried out by a person using only paper and pencil.
ALGORITHMIC PROBLEM

Specification of Input $\Rightarrow$ ? $\Rightarrow$ Specification of output as a function of input.

Algorithmic solution:

Input instance adhering to the specification. $\Rightarrow$ Algorithm $\Rightarrow$ Output related to the input as required.

Algorithm describes actions on the input instances. Infinitely many correct algorithms for the same algorithmic problem are possible. Any algorithm which is efficient in terms of the following is good for us.

(a) **Running time**: Good algorithms take least time.
(b) **Space used**: Good algorithm occupies least space.

Measuring the Running Time:
Most Algorithms transform input objects into output objects. The running time of an algorithm typically grows with the input size average case running time is often difficult to determine. So, the focus is given on the worst case running time because it is easier to analyze.

Experimental Studies:
1. Write the program that implements the algorithm.
2. Run the program with data sets of varying size and composition.
3. Use a method system. `Currenttimeinmillis()` to get an accurate measure of the actual running time.
4. Plot the result.

![Graph](image-url)
Limitation of experimental studies:
(a) Results may not be indicative of the running time on other inputs not included in the experiments.
(b) In order to compare 2 algorithms the same, hardware and software environment must be used.

Theoretical Analysis:
It uses a high level description of the algorithm instead of an implementation and characterizes running time as a function of the input size. It takes into account all possible inputs theoretical analysis allow us to evaluate the speed of an algorithm independent of hardware/software environment.

Pseudocode: A mixture of natural language and high level programming constructs that describes the main ideas behind the generic implementation of data structure or algorithms.

Example: Algorithm ArraysMax(A, n)
Input: The Array A storing n integers.
Output: the maximum element of A.
—CurrentMax ← A[0]
For i ← 1 to n−1 do
    If current max < A[i] then
        Current max ← A[i]
return currentMax.

Primitive operations: These are the basic computation performed by an algorithm identifiable in psuedo code. They are independent of the programming language for example.
• Evaluating an expression.
• Returning from a method.
• Indexing into an array.
• Arithmetic and logical operation.

By inspecting the pseudo code we can count the total number of primitive operations executed by an algorithm.

For example:
Algorithm ArrayMax (A, n) Operations
Current max ← A[0] .......................... 1
For i ← 1 to n−1 do ..........................
    If (A[i] > current max) then ..........
        Current Max ← A[i] ................. (n−1)

return currentmax
Total (n+1)

Asymptotic Analysis:
Goal: To simplify the analysis of running time by getting rid of “details” which may be affected by specific implementation and hardware.
[1]. The “Big on” O-Notation: It gives us the upper bound (Worst case behaviour) of the running time.
If f(n) and g(n) are two increasing functions on non-negative numbers then.

\[ f(n) = O(g(n)) \text{ if there exists constant } c \text{ and } n \geq n_0 \text{ such that } f(n) \leq cg(n) \forall n \geq n_0. \]
For example: If \( f(n) = 60n^2 + 5n + 11 \) then 
\[ 60n^2 + 5n^2 + 11n^2 \]
\[ f(n) < 76n^2 \text{ where } c = 76 \]
\[ < 76n^2. \]
\[ \therefore \quad n > n_0 = 1 \]
\[ \therefore \quad f(n) = O \left( n^2 \right) \]

**Simple Rule:** Pop lower order terms and constant factors.

e.g. \(-5n^2 + 6n + 20 = O \left( n^2 \right)\)

and \(-5n^2 \log n + n + 7 = O \left( n^2 \log n \right)\)

**Relatives of Big-O:**

**Big Omega:** It gives the tight bound (Average case behaviour) on running time. If \( f(n) \) and \( g(n) \) are two increasing functions over non-negative numbers. Then

\[ f(n) = \Omega \left( g(n) \right) \quad \text{if there exists a constant } c \text{ and } n \geq n_0 \]

such that \( f(n) \geq cg(n) \quad \forall \ n \geq n_0 \)

For example: \( f(n) = 60n^2 + 5n + 11 > 60n^2 \quad \forall \ n > 1 \)

Therefore, \( f(n) = \Omega \left( n^2 \right) \) where \( c = 60 \) and \( n > n_0 = 1 \)

**Big Theta:** It gives the tight bound (Average case behaviour) on running time. If \( f(n) \) and \( g(n) \) are two increasing function defined over non-negative then

\[ f(n) = \Theta \left( g(n) \right) \quad \text{if there exists constants } c_1 \text{ and } c_2 \text{ and } n \geq n_0 \text{ such that} \]

\[ c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall \ n \geq n_0 \]

\( f(n) \) is sandwiched between \( c_1 g(n) \) and \( c_2 g(n) \)
If \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \)

Then \( f(n) = \theta(g(n)) \) and vice-versa.

As we have, \( f(n) = 60n^2 + 5n + 11 \) and \( f(n) = O(n^2) \) and \( f(n) = \Omega(n^2) \)

Therefore, \( f(n) = \theta(n^2) \)

**PROBLEM**

1. Consider the following three claims: \[ \text{[GATE-2003]} \]
   (i) \( (n + k)^m = O(n^m) \)
   (ii) \( 2^{n+1} = O(2^n) \)
   (iii) \( 2^{2n+1} = O(2^n) \)

   which of the above claims are correct.
   (a) 1, 2  (b) 1, 3  (c) 2, 3  (d) 1, 2, 3

   **Ans.** (a)

2. Consider the following function \[ \text{[GATE-2008]} \]
   \[ f(n) = 2^n \quad g(n) = n! \quad h(n) = n^{\log n} \]

   Which of the following statements about the asymptotic behaviour of \( f(n) \), \( g(n) \) and \( h(n) \) is TRUE.
   (a) \( f(n) = O(g(n)); \quad g(n) = O(h(n)) \)
   (b) \( f(n) = \Omega(g(n)); \quad g(n) = O(h(n)) \)
   (c) \( g(n) = O(f(n)); \quad h(n) = O(f(n)) \)
   (d) \( h(n) = O(f(n)); \quad g(n) = \Omega(f(n)) \)

   **Ans.** (d), since fact value larger than exponential values.

3. Two alternative packages A and B are available for processing a database having \( 10^6 \) records. Package A requires \( 0.0001 n^2 \) time units and packages B requires \( 10\log_{10}n \) unit to pass \( n \) records. What is the smallest value of \( k \) for which packages B will be performed over A \[ \text{[GATE-2010]} \]
   (a) 12  (b) 10  (c) 6  (d) 5

   **Ans.** (d)
4. Let \( n \) be a large integer which of the following statement is true

\[
\textbf{[TIFR-2012]}
\]

(a) \( 2^{\sqrt{\log n}} < \frac{n}{\log n} < \frac{1}{n^3} \)

(b) \( \frac{n}{\log n} < n^{\frac{1}{3}} < 2^{\sqrt{\log n}} \)

(c) \( 2^{\sqrt{\log n}} < n^{\frac{1}{3}} < \frac{n}{\log n} \)

(d) \( n^{\frac{1}{3}} < 2^{\sqrt{\log n}} < \frac{n}{\log n} \)

Ans. (d)

**RRECURRENCES**

A recurrence is an equation or inequality that describes a function in terms of its smaller inputs.

Examples:

\[
T(n) = \begin{cases} 
\theta(1) & \text{if } n = 1 \\
4T\left(\frac{n}{2}\right) + \theta(n) & \text{if } n > 1
\end{cases}
\]

Or in general, \( T(n) = aT(\frac{n}{b}) + f(n) \) where \( a \geq 1 \) and \( b > 1 \)

In the above recurrence the whole problem of input size \( n \) is divided into subproblems of \( \frac{n}{b} \) size and the cost of dividings and combining the solution is \( f(n) \).

Let us consider the following recurrence:

\[
T(n) = T(n-1) + n
\]

We can further expand it as follows:

\[
T(n) = T(n-2) + (n-1) + n \\
= T(n-3) + (n-2) + (n-1) + n \\
\vdots \\
= T(1) + 2 + 3 + 4 + \ldots + (n-1) + n \\
= 1 + 2 + 3 + 4 + \ldots + n \left[ \text{if } T(1) = 1 \right] \\
= \frac{n(n+1)}{2} = \theta(n^2)
\]

**Solution of recurrence relations:**

\[ \bullet \text{Recurrence tree method.} \]

\[ \bullet \text{Master method.} \]

**Recurrence tree method:**

Suppose:

\[
T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n) \quad \Rightarrow \quad T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn
\]

Which says problem of \( n \) size is divided into two parts of size \( \frac{n}{3} \) and \( \frac{2n}{3} \) respectively and dividing and combing of their solution takes \( cn \) time. So tree can be constructed as:

\[
\begin{array}{c}
T\left(\frac{n}{3}\right) \\
\uparrow cn \\
T\left(\frac{2n}{3}\right)
\end{array} \quad \Rightarrow \quad \begin{array}{c}
T\left(\frac{n}{3}\right) \\
\uparrow c\left(\frac{n}{3}\right) \\
T\left(\frac{2n}{3}\right)
\end{array} \quad \begin{array}{c}
T\left(\frac{n}{3}\right) \\
\uparrow c\left(\frac{2n}{3}\right) \\
T\left(\frac{2n}{3}\right)
\end{array}
\]
The longest path from root to leaf is \( n \rightarrow \frac{2}{3} n \rightarrow \left( \frac{2}{3} \right)^2 n \rightarrow \ldots \rightarrow 1 \) since \( \left( \frac{2}{3} \right)^k n = 1 \Rightarrow k = \log_{3/2}^n \) i.e. the height of the tree is \( \log_{3/2}^n \). We calculate the cost at each level. Then summing the whole cost we get \( T(n) = \Theta(n \log n) \).

**MASTER METHOD**

Master method is used for the following recurrences of the form.

\[
T(n) = aT\left(\frac{n}{b}\right) + f(n)
\]

where \( a \geq 1 \) and \( b > 1 \) are constants and \( f(n) \) is an asymptotically positive function then.

**Case 1:** If \( f(n) = O\left(n^{\log_b a - \epsilon}\right) \) for some constant \( \epsilon > 0 \) then

\[
T(n) = \Theta\left(n^{\log_b a}\right)
\]

**Case 2:** If \( f(n) = \Omega\left(n^{\log_b a}\right) \) then \( T(n) = \Theta\left(n^{\log_b a} \log n\right) \)

**Case 3:** If \( f(n) = \Theta\left(n^{\log_b a + \epsilon}\right) \) for some constant \( \epsilon > 0 \) and if \( af\left(\frac{n}{b}\right) \leq c f(n) \) for \( c < 1 \) and all sufficiently large \( n \) then

\[
T(n) = \Theta\left(f(n)\right)
\]

**Examples:**

(a) \( T(n) = 9T\left(\frac{n}{3}\right) + n \)

here \( a = 9, \quad b = 3 \) and \( f(n) = n \).

\[
n^{\log_b a} = n^{\log_3 9} = n^2 \quad \text{since} \quad f(n) = O\left(n^{\log_3 9 - \epsilon}\right)
\]

Where \( \epsilon = 1 \) we can apply case 1 of master method and calculate that \( T(n) = \Theta\left(n^2\right) \).
(b) \( T(n) = T\left(\frac{2n}{3}\right) + 1 \)

here \( a = 1 \quad b = \frac{3}{2} \quad f(n) = 1; \quad n^{\log_b^a} = n^{\log_{\frac{3}{2}}} = n^0 = 1 \)

Case 2 of master method applies and thus the solution to the recurrence relation is \( T(n) = \Theta(\log n) \)

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

We have \( a = 3, b = 4; \quad f(n) = n \log n \)

\( n^{\log_b^a} = n^{\log_4^3} = O\left(n^{0.793}\right) \quad \text{Since} \quad f(n) = \Omega\left(n^{\log_4^3 + \varepsilon}\right) \)

Where \( \varepsilon \approx 0.2 \) case 3 applies as

\( af\left(\frac{n}{4}\right) = 3\left(\frac{n}{4}\right)^{\log_4^3} \leq n \log n = c f(n) \quad \text{for} \quad c = \frac{3}{4} \)

Therefore by case of master theorem:

\( T(n) = \Theta(n \log n) \)

(d) \( T(n) = 2T\left(\sqrt{n}\right) + n \log n \)

Here \( a = 2, b = 2 \quad \text{and} \quad f(n) = n \log n \)

\( n^{\log_b^a} = n^{\log_2^2} = n \quad \text{as} \quad f(n) \quad \text{is not polynomially larger than} \quad n^{\log_2^2}. \quad \text{Therefore it does not follow any case of master theorem.} \)

(e) \( T(n) = 2T\left(\left\lfloor n^{\sqrt{n}} \right\rfloor \right) + \log n \)

**Soln.** Let \( n = 2^m \) and ignoring the \( \lfloor \rfloor \) sign.

\( T(2^m) = 2T\left(\frac{2^{m/2}}{2}\right) + m \)

Now let \( T(2^m) = S(n) \quad \text{then} \)

\( S(m) = 2S\left(\frac{m}{2}\right) + m \)

here \( a = 2 \quad b = 2 \quad \text{and} \quad f(m) = m \)

\( m^{\log_b^a} = m^{\log_2^2} = m = f(m) \quad \text{so by case II of master method.} \)

\( S(m) = \Theta(m \log m) \quad \text{Now changing back.} \)

\( T(n) = S\left(2^m\right) = \Theta(m \log m) \)

\( = \Theta(\log n \log \log n) \)
AKRABAZZI METHOD

If the recurrence has this form

\[ T(n) = \sum_{i=1}^{n} a_i \left( T \left( \frac{n}{b_i} \right) \right) + f(n) \]

\[ a_i > 0, \ b_i > 1 \quad f(n) = \text{it should be polynomial} \quad f(n) = \theta \left( n^d \right) \]

where,

\[ \sum a_i / b_i = 1 \quad \text{(row) } p = ? \]

\[ T(n) = \theta \left( n^p \left( 1 + \int_1^n f(u) \left( \frac{u}{u+1} \right) du \right) \right) \]

\[ f(n) = f(u) \]

e.g. \[ T(n) = T \left( \frac{3n}{4} \right) + T \left( \frac{n}{4} \right) + n \]

\[ T(n) = a_1 T \left( \frac{n}{b_1} \right) + a_2 T \left( \frac{n}{b_2} \right) + f(n) \]

\[ a_1 = 1 \quad a_2 = 1 \]

\[ b_1 = \frac{4}{3} \quad b_2 = 4 \]

\[ \frac{a_1}{b_1^p} + \frac{a_2}{b_2^p} = 1 \]

\[ \frac{1}{\left( \frac{4}{3} \right)^p} + \frac{1}{4p} = 1 \quad \Rightarrow \left( \frac{3}{4} \right)^p + \left( \frac{1}{4} \right)^p = 1 \quad ; \quad p = 1 \quad \frac{3}{4} + \frac{1}{4} = 1 \]

\[ T(n) = \theta \left( n^p \left( 1 + \int_1^n u \left( \frac{u}{u+1} \right) du \right) \right) = \theta \left( n + n \left[ \log u \right] \right) \]

\[ = \theta \left( n + n \left( \log n - \log 1 \right) = \theta \left( n + n \log n \right) \right. \]

\[ n \log n > n \]

\[ T(n) = n \log n \]

e.g. \[ T(n) = T(n-1) + T(n-2) \]

\[ T(1) = 1 \quad \text{let } T(n) = x^n \]

\[ T(2) = 1 \]

\[ f_n = f_{n-1} + f_{n-2} \]

1, 1, 2, 3, 5, 8, ......................

\[ x^n = x^{n-1} + x^{n-2} \quad \Rightarrow \quad x^n = \frac{x^n}{x} + \frac{x^n}{x^2} \]

\[ 1 = \frac{1}{x} + \frac{1}{x} \quad \Rightarrow \quad 1 = \frac{x+1}{x^2} \quad \Rightarrow \quad x^2 - x - 1 = 0 \]
\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \implies x = \frac{1 \pm \sqrt{5}}{2} \]

\[ \alpha = \frac{1 + \sqrt{5}}{2}, \quad \beta = \frac{1 - \sqrt{5}}{2} \]

Solution is \( T(n) = c_1\alpha^n + c_2\beta^n = c_1 \left( \frac{1 + \sqrt{5}}{2} \right)^n + c_2 \left( \frac{1 - \sqrt{5}}{2} \right)^n \)

\[ T(n) = \theta \left( \frac{1 + \sqrt{5}}{2} \right)^n = \theta \left( \frac{1 + 2.14}{2} \right)^n = \theta \left( \frac{3}{2} \right)^n \]

**PROBLEMS**

1. The running time of an algorithm is represented by the following recurrence relation \[ T(n) = \begin{cases} 
n \leq 3 \\
T \left( \frac{n}{3} \right) + c_n, \quad \text{otherwise}
\end{cases} \] which of the following represents the complexity of the above recurrence relation.

(a) \( \theta(n) \) \hspace{1cm} (b) \( \theta(n \log n) \) \hspace{1cm} (c) \( \theta(n^2) \) \hspace{1cm} (d) \( \theta(n^2 \log n) \)

**Ans.** (a)

2. The recurrence relation that arises with the complexity of binary search is: \[ \begin{cases} 
T(n) = 2T \left( \frac{n}{2} \right) + k \hspace{1cm} (a)
T(n) = T \left( \frac{n}{2} \right) + \log n \hspace{1cm} (b)
\end{cases} \]

**Ans.** (b)

3. Consider the following c-code.

```c
int j, n;
j = 1;
while (j < n)
    j = j*2;
```

The numbers of comparison made in the execution of the loop for any \( n > 0 \) is \[ \begin{cases} 
\lceil \log n \rceil + 1 \hspace{1cm} (a)
\lceil \log n \rceil \hspace{1cm} (b)
\end{cases} \]

**Ans.** (a)

4. How to find the complexity of a program

**Soln.** Complexity is basic rough estimate complexity is depend on number of loops

\[
\begin{align*}
\text{for} \ (i = 0; \ i < n; \ i++) \ n & \quad \theta(n^2) \\
\text{for} \ (j = 0; \ j < n; \ j++) \ n & \quad \theta(n) \\
\end{align*}
\]

5. Find out the complexity of a program.

\[
\begin{align*}
i = n; \\
\text{while} \ (i > 0)
\end{align*}
\]