UNIT-I

Algorithms:
- Step by step procedure or instruction to solve any problem.
- Algorithm + Data Structures = Programs.
- Algorithmics - Framework for designing & analyzing alg.

Need : Algorithmics:
- To develop a framework for instructing computers to perform tasks.
- To introduce notion of alg. as means of specifying how to solve a pbm.
- To introduce approaches for defining and solving very complex tasks.

Features of Algorithm:
- Finiteness - must have some way to terminate after solving the pbm.
- Definiteness - must be exact definition for each step.

- Input - Any number of inputs can be given.
- Output - Number of outputs produced (1 or more).
- Effectiveness - Computation should be simple.
Ex:

Algorithm: Finding largest number

Input: List of numbers \( L \)

Output: Largest number in the list \( L \).

\[ \text{Largest} \leftarrow \text{Null} \]

for each item in \( L \), do

if item > Largest, then

Largest \leftarrow \text{item} \]

return largest

Algorithm: Notion:

- Sequence of unambiguous instructions for solving a problem.

Problem

\[ \downarrow \]

Algorithm

\[ \downarrow \]

1/p

[computer] \rightarrow \text{output} /p

A same problem can be solved using different methods and procedures. For example, finding GCD (Greatest Common Divisor) of any two non-negative integer numbers.
3 Methods are there to find GCD of 2 numbers.

1. Euclid's Algorithm.

2. Consecutive Integer Checking Algorithm.

3. Middle School Procedure.

Euclid's Algorithm:

Based on applying repeatedly the equality.

Alg: Computing GCD

1/ Inputs: 2 non-negative, non both zero integers $m$ and $n$.
2/ Outputs: GCD of $m$ and $n$.

while $n 
eq 0$ then

\[ r = m \mod n \]

\[ m = n \]

\[ n = r \]

return $m$

Illustration: GCD (60, 24)

\[ \text{gcd} (m, n) = \text{gcd} (n, m \mod n) \]

\[ \text{gcd} (24, 60 \mod 24) \]

\[ = \text{gcd} (24, 12) \]

\[ = \text{gcd} (12, 24 \mod 12) \]

\[ = \text{gcd} (12, 0) \]

\[ = 12 \]
Algorithm: Steps:

Step 1: If \( n > 0 \), return the value of \( m \) as answer & Stop.
otherwise proceed.

Step 2: Divide \( m \) by \( n \) & assign the val. of remainder to \( r \).

Step 3: Assign the value of \( n \) to \( m \), value of \( r \) to \( n \).
Go to Step 1.

Consecutive Integer checking Algorithm:

- A common divisor cannot be the smaller greater than the smaller of the two numbers.

\[ t = \min \{ m, n \} \]

So, we check whether \( t \) divides both \( m \) and \( n \).
if it does, \( t \) is the answer.
otherwise, decrease \( t \) by 1 and try again.

Steps:

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 answer & Stop.
otherwise, proceed to Step 4.

Step 4: Decrease the value of \( t \) by 1. Go to Step 2.
Illustration: \( \text{GCD}(12, 8) \)

\[
\begin{align*}
12 \mod 8 & = 4 \\
5 & = 3 \\
8 \mod 6 & = 2 \\
7 & = 4 \\
8 \mod 4 & = 0 \\
9 & = 0
\end{align*}
\]

\( \text{GCD}(12, 8) = 4 \)

Middle School Procedure: Steps:

Step 1: Find the prime factors of \( m \).
Step 2: Find the prime factors of \( n \).
Step 3: Identify all the common factors of \( m, n \).
Step 4: Compute the product of all common factors and return it as \( \text{gcd} \).
Illustration: \[ \gcd(60, 24) \quad \gcd(120, 72) \]

\[
\begin{align*}
2 & \mid 60 & 2 & \mid 24 \\
2 & \mid 30 & 2 & \mid 12 \\
2 & \mid 15 & 2 & \mid 6 \\
3 & \mid 5 & 3 & \mid 3 \\
& 3 & & 3 \\
60 & = 2 \times 2 \times 3 \times 5 \\
24 & = 2 \times 2 \times 2 \times 3
\end{align*}
\]

\[
\begin{align*}
\gcd(60, 24) & = \\
120 & = 2 \times 2 \times 2 \times 3 \times 5 \\
72 & = 2 \times 2 \times 3 \times 3 \\
\gcd(120, 72) & = 2 \times 2 \times 3 \times 3 = 24
\end{align*}
\]

Fundamentals of Algorithmic Problem Solving:

- Algorithms are not solutions to any problem. But they are instructions to reach the solutions to problems.
- Various steps involved in the algorithm design & analysis are

1. Understanding the problem
2. Ascertain the capabilities of a Computational device
3. Choose between exact & approximate problem solving
4. Decide appropriate data structures
5. Methods of specifying an algorithm
6. Proving an algorithm's correctness
7. Analysing an algorithm
8. Coding an algorithm
The following criteria are used to analyse the algorithm:

a) Correctness
b) Amount of work done
c) Amount of space used
d) Simplicity
e) Clarity
f) Optimality

![Algorithm design & Analysis Process]

Algorithm design technique is a general approach to solve problems algorithmically that is applicable to a variety of problems from different areas of computing.
Understanding the Problem:

- Read the problem description carefully.
- If any doubts in the problem statement, then ask question to clarify doubts.
- Do some examples, and then think about special cases, if required again ask questions.
- Once the problem is clearly understood, then determine the overall goals in a precise manner.
- Divide the problem into smaller problems of manageable size.

Ascertaining the capabilities of the computational device:

- Most algorithms are programmed for a computer (Von Neumann machine).
- In random access machine, instruction to be executed one after another; one operation at a time.
- In newer computers, the operation can be executed parallelly. The algorithms used in these computers are parallel algorithms.

Choosing between exact & approximate problem solving:

- Problem should be solved either exactly/approximately.
- Former algorithm is exact algorithm, latter is approximate algorithm.
Reason to Choose Approximate Algorithm:

- There are important problems that simply can't be solved exactly such as:
  - Extracting the square root.
  - Solving non-linear equations.
  - Evaluating definite integrals.
- Approximate algorithms are part of a more sophisticated algorithm that solves a problem exactly.

Algorithm Design Techniques:

- General approach to solve the problem algorithmically from different areas of computing.
- Provide guidance for designing algorithms.
- Act as a powerful collection of tools.
- Used to classify the algorithms based on the design idea.

Methods of Specifying the Algorithm:

- Presentation of algorithm is focused here.
- 2 ways of presenting - Pseudocodes & Natural languages.
  - Pseudocode is the mixture of natural language & programming language.
  - Pseudocode is more precise than natural language.
- Flowchart is a method of expressing an algorithm by a collection of connected geometric shapes containing description of the algorithm.
- Algorithm should prove that it yields a correct result for every input in a finite amount of time.
- Mathematical induction is the effective way of proving the correctness.
- It is also used to show under what condition an algorithm fails.

Analyzing the algorithm:
- Efficiency of an algorithm is determined by measuring time, space and amount of resources it uses for executing the program.
- Time taken by an algorithm can be calculated by finding the steps the algorithm executes.
- Optimality is the minimum amount of effort any algorithm will need to exert to solve the problem.
- 2 types of algorithm efficiency:
  - Time efficiency / Time complexity
  - Space efficiency / Space complexity

Time Efficiency:
- Time taken to solve the problem.
- Time required to complete the task is not always same.
Amount of time taken by the compiler.

- When implementing algorithms as programs to be used in actual applications, mathematical verifications must be done.
- Power of an algorithm depends on the coding.

Conclusion:
As a rule, a good algorithm is a result of repeated effort and network.
Important Problem types:

- Sorting
- String processing
- Geometric problem
- Searching
- Graph problems
- Numerical problems
- Combinatorial problems.

Sorting:

- Rearrange the items of a given list in some order.
- Sorting can be done on numbers, characters, strings, and records.
- In case of sorting the records, we need to use piece of information as key to guide the sorting work.
- Two properties of sorting algorithms are:
  - Stable property
  - In-place property
- Algorithm is said to be stable, if it preserves the relative order of any two equal elements in its input.
  (i.e.) if an input list contains two equal elements in position i and j, then in the sorted list they have to be in positions i' and j', such that i' < j'.
- Algorithm is said to be in-place, if it doesn't require extra memory, except possibly for few memory units.
Searching:
- It is an activity by which we can find out the desired element from the list. The element which is to be searched is called 'search key'.
- Many searching algorithms - sequential search, binary search, Fibonacci search and many more.
- There is no single algorithm that fits all situations.
- Efficiency of searching algorithm has to be considered along with the two other operations: Addition, Deletion.

String Processing:
- String is a sequence of characters from an alphabet.
- Bit strings which comprises zeros and ones.
- Searching for a given word in text is called 'String matching'.
- Text strings which comprises letters, numbers and special characters.

Graph Problems:
- Graph comprises collection of points called vertices, some of which are connected by line segments called edges.
- Basic graph algorithms include graph traversal, shortest path & topological sorting for graphs with directed edges.
Geometric problems:
- It deals with geometric objects such as points, lines and polygons.
- Geometric algorithms also find its applications in computer graphics, robotics and tomography.

Numerical problems:
- This involves mathematical objects of continuous nature.
- Solving equations, computing definite integrals, evaluating functions and so on.
- This can only be solved approximately.
- The error can distort the output.

Combinatorial problems:
- These are related to the problems like computing permutations and combinations. This kinds of problems are more difficult due to the following reasons:
- As problem size grows, combinatorial objects grow rapidly.
- No algorithm is available to solve these problems within an amount of time.
- Many of these problems fall under the category of unsolvable problems.
Fundamentals of the analysis of algorithm efficiency:
- Process of investigation of an algorithm's efficiency with respect to two resources.
- Running time
- Memory Space
- Simplicity & generality measures efficiency in quantitative terms.
- Speed & memory are the efficiency consideration of modern computers.

Analysis frame work:

Time efficiency: It indicates how fast an algorithm runs.
Space efficiency: It deals with extra space the alg. requires.
- use of secondary memory devices will affect the running time of the algorithm thus affecting the time efficiency.

Measuring an input size:
- Efficiency of an alg. is directly proportional to the 1/p size.
- Alg. efficiency could be measured in terms of 'n'.
  where 'n' is the input size of an algorithm.
- Some alg. require more than 1 parameter to indicate the size of their input. In such cases, size is measured by the number of bits 'b' in the n's binary representation:
  \[ b = \text{floor} \left( \log_2 n + 1 \right) \]
**Ex:** Number of bits in binary representation for the numbers

<table>
<thead>
<tr>
<th>Number</th>
<th>( \log_2 n )</th>
<th>( \lfloor \log_2 n \rfloor )</th>
<th>( \text{No of bits in binary rep. b} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0000</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.0000</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1.5850</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2.0000</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2.3219</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>2.5850</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>2.8074</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>3.0000</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**Units for measuring running time:**

- Running time of an alg. depends on:
  - Speed of a particular computer on which Pgm is to run.
  - Quality of the Pgm. Implementing the alg.
  - The compiler used to generate the machine code.
  - Difficulty of clocking the running time of the Pgm.

In order to avoid the above drawbacks, we have the following alternatives.

- Identify the most important operation of an alg. (i.e.)
  - Identify the basic operation (operation which contributes more to the total running time).
- Compute the no. of times, the basic operation is executed will give the running time.
- Usually, the basic operation is the most time consuming operation of an algorithm.
- \( T(n) \approx C_{op} \cdot C(n) \)
- \( T(n) \) - Running time
- \( C_{op} \) - Execution time of basic operation
- \( C(n) \) - No of times the operation needs to be executed.
- Apart from basic operation, there may be other operations that will contribute to the running time.
- For these reasons, that the efficiency analysis framework ignores multiplicative constants & focuses on \( n \)‘s order of growth.

Orders of growth:
- Measuring the efficiency of an alg. with the input size \( n \)
  is called order of growth.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( n \cdot \log_2(n) )</th>
<th>( n \cdot \log_3(n) )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( 2^n )</th>
<th>( n! )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.3</td>
<td>3.3x10</td>
<td>10^2</td>
<td>10^3</td>
<td>10^7</td>
<td>3.6x10^6</td>
</tr>
<tr>
<td>10^2</td>
<td>6.6</td>
<td>6.6x10</td>
<td>10^4</td>
<td>10^6</td>
<td>1.3x10^3</td>
<td>9.3x10^5</td>
</tr>
<tr>
<td>10^3</td>
<td>10</td>
<td>10x10^3</td>
<td>10^6</td>
<td>10^9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10^4</td>
<td>13</td>
<td>13x10^4</td>
<td>10^8</td>
<td>10^12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- when the value of \( n \) is 10, \( \log(n) \) gives result as \( 2^n \) gives \( 10^3 \) (1000).
- \( \log \) functions grow slow even for high range of input.
- whereas \( \exp(1/n) \) grows fast for small increase in \( 1/n \) size.

Worst Case, Best Case, Avg case efficiencies:
- Algorithm efficiency is measured in the form of alg. input size.
- If an alg. takes min. amount of time to complete the task for specific input, it is called best time complexity.
- If an alg. takes max. amount of time to complete the task for specific input, it is called worst case time complexity.

\( T(n) \) is different for different input.

Time taken to solve problem depends on \( 1/p \).

Ex: Search for \( k \) in the array \( A \):

\[ i \leftarrow 0 \]
\[ \text{while } i < n \text{ do} \]
\[ i \leftarrow i + 1 \]
\[ \text{if } i < n \text{ return} \]
\[ \text{else return} \]

Case 1: \( k = A[i] \), loop exits in 1 step.

Case 2: \( k \neq A[i] \), loop takes \( n \) steps.

Best Case

\( k \) is not in \( A \).

Worst Case
Ex: Best case: In the prev. example, if \( k \) is the first element of an array \( A \) (ie. \( A[0] \)), loop exits. This is the best case scenario.

Worst case: If \( k \) is not present in the array \( A \), the loop takes \( n \) no of steps. This is the worst case scenario.

Illustrations:

Array:
\[ A = [45, 86, 87, 25] \]

Search Key to be found out:
\( k = 25 \)

Because search key 25 is the last element in the array, loop exits at the last step. (4 times executed).

\( C_{\text{worst}}(n) = n \)

Best case scenario:
\[ A = [25, 45, 86, 87] \]

\( k = 25 \)

\[ C_{\text{best}}(n) = 1 \]

Worst case efficiency will give the upper bound for the running time.

Best case efficiency will give the lower bound for the running time. It is rarely used.
Average case time complexity.

This will give the behavior of the algorithm on specific or random input.

Example: Sequential search in an array of size `n`. Probability of successful search.

Illustration: 1st match is at location `i`.

Probability of occurring first match is `p_i` for every `i`th element.

Probability of getting unsuccessful search is 

\[(1-p)\]

\[c_{avg}(n) = \text{Prob. of Successful search} + \text{Prob. of unsuccessful search}\]

\[= \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + 3 \cdot \frac{p}{n} + \ldots + i \cdot \frac{p}{n} + \ldots + n \cdot \frac{p}{n}\right] + n \cdot (1-p)\]

\[= \frac{p}{n} \left[1 + 2 + 3 + \ldots + i + \ldots + n\right] + n \cdot (1-p)\]

\[= \frac{p}{n} \cdot \frac{n(n+1)}{2} + n \cdot (1-p)\]

\[c_{avg}(n) = \frac{p \cdot (n+1)}{2} + n \cdot (1-p)\]

\[\text{If } p = 1\]

\[= 1 \cdot \frac{(n+1)}{2}, \ n \cdot (1-1)\]

\[c_{avg}(n) = \frac{n+1}{2}\]

Successful search.

\[\text{If } p = 0\]

\[c_{avg}(n) = n + n = 2n\]

Unsuccessful search.

www.studentsfocus.com
In successful search, an average, about half of the elements will be inspected by the algorithm. 

\[ \frac{n+1}{2} \]

In unsuccessful search, the avg. no of key comparisons will be \( n \) (i.e. total size of an array).

**Amortized Efficiency/complexity:**

- Amortized complexity is defined as that finding the avg. running time/operation over a worst-case sequence of operations.

**Rent:** Buy – One-time cost

Pay per use.

- Rent/DVD: $10
- Buy/DVD: $50

Asymptotic Notation & its Properties:

- Algorithm efficiency is determined by the order of growth of that algorithm.
- Order of growth is determined by the brain operation count: \( c(n) \)
To compare and rank the order of growth of an algorithm, we use three notations:
- \( O \) (Big oh), \( \Omega \) (Big Omega), \( \Theta \) (Big theta)
- \( t(n) \), \( g(n) \) are non-negative functions
- \( t(n) \) - Algorithm running time - Indicated by basic operations count
- \( g(n) \) - Simple function

**Big Oh notation:**
\[ g(n) \text{ is tightest upper bound for } t(n) \]

\[ \frac{1}{c} \]

\[ t(n) \leq c \cdot g(n) \quad n \geq n_0 \]
\[ \frac{t(n)}{g(n)} \leq c \quad n \geq n_0 \]

- If the \( \frac{t(n)}{g(n)} \) increases, the time taken to solve the problem \( t(n) \) also gets increased.
- After \( n_0 \) time, the value of \( c \cdot g(n) \) is always greater than \( t(n) \).

**Example:**
\[ t(n) = 3n + 2 \quad g(n) = n \]
\[ t(n) \leq c \cdot g(n) \quad c > 0, n \geq 1 \]
\[ 3n + 2 \leq cn \]
Assume \( c = 4 \) (Any \( c \) above 3 is good)
\[ 3n + 2 \leq 4n \]
\[ 2 \leq n \implies n \geq 2 \]

Proved.
For every \( n \geq n_0, \quad C = 4 \), \( f(n) \leq C \cdot g(n) \).

\[ E(n) = O(g(n)) \]

- \( E(n) \) is always bounded by \( g(n) \) provided it is multiplied with a constant \( C \).
- \( E(n) = O(g(n)) \).

If this is proved, \( g(n) = n, n^2, n^n, 2^n, n! \) (All these for bound \( E(n) \)).

**Big Omega (\( \Omega \))**

\( g(n) \) is an tightest lower bound for \( E(n) \).

Ex:

\[ E(n) \geq 3n + 2, \quad g(n) = n \]

Can \( E(n) \) be bounded by \( g(n) \)?

(a) \( g(n) \) is lower than \( E(n) \)?

\[ E(n) \geq g(n) \]

\[ E(n) \geq c \cdot g(n) \]

\[ 3n + 2 \geq c \cdot n \quad (C = 1) \]

\[ 3n + 2 \geq n \quad \text{Proved} \]

\( c \cdot g(n) \) is less than \( t(n) \) after \( n_0 \).

- A function \( E(n) \) is said to be \( \Omega(g(n)) \), if \( E(n) \) is bounded below by some positive constant multiple of \( g(n) \).

**Big Theta (\( \Theta \))**

Ex:

\[ E(n) \geq 3n + 2, \quad g(n) = n \]

\[ f(n) \leq C \cdot g(n) \]

\[ 3n + 2 \leq 4 \cdot n \quad n \geq 21 \quad \text{Proved} \]
Relation between $f(n)$, $g(n)$, and $h(n)$:

For any two functions $g(n)$ and $h(n)$,

\[ f(n) = \Theta(g(n)) \iff f(n) = \Omega(g(n)) \text{ and } f(n) = \omega(g(n)) \]

\[ \Theta(g(n)) = O(g(n)) \cap \Omega(g(n)) \]

Tight bounds are obtained from tight upper and tight lower bounds.

Using limits for comparing orders of growth:

**Three principal cases are there**:

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \begin{cases} 
0 & \text{case 1: } f(n) \text{ has a smaller order of growth than } g(n) \\
c > 0 & \text{case 2: } f(n) \text{ has the same order of growth than } g(n) \\
\infty & \text{case 3: } f(n) \text{ has the larger order of growth than } g(n).
\end{cases}
\]

Basic Efficiency Classes:

<table>
<thead>
<tr>
<th>Order of Growth</th>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>constant</td>
<td>Scanning away elements</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>logarithmic</td>
<td>Performing binary search</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>linear</td>
<td>Performing sequential search</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>quadratic</td>
<td>Sorting using merge/quick sort</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>cubic</td>
<td>Scanning matrix elements</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>exponential</td>
<td>Performing matrix multiplication</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>factorial</td>
<td>Generate all subsets of $n$ elements</td>
</tr>
<tr>
<td>$O(n!)$</td>
<td></td>
<td>Generate all permutations</td>
</tr>
</tbody>
</table>

www.studentsfocus.com
Pseudocode:
- Starts with the keyword `ALGORITHM` with the name of the algorithm and the input parameters
- Next few lines of pseudocode should be the comment statements.
  - Problem description
  - Input
  - Output

Conventions used in pseudocode:

Assignments: `variable <- expression`

Comments: `//` This is comment line

Block of Statements:
- Statement 1
- Statement 2
- Statement n

Conditional Construction: `if cond then_stmt`

Loop Construction: `for var <- initial_val to final_val stmt`

Procedure: `Procedure max(L)`

Using Procedure in other Procedure:

Properties of Asymptotic Notations:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Reflective</th>
<th>Transitive</th>
<th>Symmetric</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>!O</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>!O</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Mathematical Analysis of non-recursive algorithms:

- The common way for analyzing the time efficiency of non-recursive algorithms is:
- Decide on a parameter indicating an input's size.
- Identify algorithm's basic operation.
- Check whether the number of times the basic operation is executed depends on the size of an input.
- Set up a sum expressing the number of times the basic operation is executed.

X(Solve the recurrence or attempt to ascertain the order of growth of its solution)X
- 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.

Before proceeding with further examples, summation formulas & rules that are often useful in analysis of algorithms shall be reviewed.

Basic rules of sum manipulation:

\[
\sum_{i=1}^{n} c \cdot a_i = c \sum_{i=1}^{n} a_i \quad \text{Rule 1}
\]

\[
\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i \quad \text{Rule 2}
\]

And two summation formulas are given below:
\[ \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \]

\[ \frac{\sum_{i=0}^{n-1} i}{\sum_{j=1}^{n} j} = 1 + 2 + \ldots + n = \frac{n(n+1)}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2) \]

Example: Algorithm: Element Uniqueness Problem

Input: Integer array \( A \) of size \( n \).

Output: true if all elements of \( A \) are distinct.

\[
\begin{align*}
\text{For } i &= 1 \ldots n-1, \text{ do} \\
\text{For } j &= i+1 \ldots n, \text{ do} &\text{ compare } i \text{ and } j \text{ until } n, n-1
\end{align*}
\]

\[
\text{if } a_i = a_j \text{ then} \quad \text{return false} \]

\[
\text{End} \\
\text{End} \\
\text{return true}
\]

For this algorithm, what is the elementary operation?

What is the input size?

Does the elementary operation depend on \( n \)?

Outer loop is run \( n-2 \) times. More formally, it contributes

\[
\sum_{i=1}^{n-2} 1 = n-2
\]

Inner loop depends on the outer loop, so it contributes

\[
\sum_{j=i+1}^{n} 1 = n - i - 1
\]
Based on this, we observe that the elementary operation $b$ executed once in each iteration. Thus we have

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

Example 2: Checking the 8th no. is odd/even parity.

Input: An integer $n$ in binary ($b[i]$)

Output: 0 if the parity of $n$ is even, otherwise.

Parity = 0

while $n > 0$, do

if $b[i] = 1$, then

Parity ← Parity + 1 mod 2

right shift (n)

End

End

return Parity

The while loop will be executed as many times as there are 1-bits in its binary representation. In the worst case, we will have a bit string of all ones.

The number of bits required to represent an integer $n$ is $\lceil \log n \rceil$.

So, the running time is simply $\log n$. 
Mathematical analysis of recursive algorithms:
- Decide on parameter $n$ indicating input size.
- Identify algorithm's basic operation.
- Determine worst, average, and best case for input of size $n$.
- Setup a recurrence relation and initial condition(s) for $c(n)$. The no. of times the basic operation will be executed for an input of size $n$.
- Solve the recurrence to obtain a closed form or estimate the order of magnitude of the solution.

Algorithm: Factorial function:
Input: A non-negative integer $n$.
Output: The value of $n!$.

if $n = 0$ then
  return 1
else
  return factorial($n-1$) * $n$

Recurrence for factorial function:
- Let $M(n)$ = multiplication count to compute factorial ($n$).
- $M(0) = 0$ because no multiplications are performed to compute factorial ($0$).
- If $n > 0$, then factorial ($n$) performs recursive call + one multiplication.
\[ M(n) = M(n-1) + 1 \]

To compute factorial (n-1)
To multiply factorial (n-1) by n.

Solving the factorial recurrence:

Make a guess.

Forward Substitution:
- \( M(1) = M(0) + 1 = 1 \)
- \( M(2) = M(1) + 1 = 2 \)
- \( M(3) = M(2) + 1 = 3 \)

Backward Substitution:
- \( M(n) = M(n-1) + 1 \)
- \( = [M(n-2) + 1] + 1 = M(n-2) + 2 \)
- \( = [M(n-3) + 1] + 2 = M(n-3) + 3 \)

Prove \( M(n) = n \) by mathematical induction

Basis: if \( n = 0 \), then \( M(n) = M(0) = 0 = n \)

Induction: if \( M(n-1) = n-1 \), then
- \( M(n) = M(n-1) + 1 \)
- \( = (n-1) + 1 = n \) Proved.

Tower of Hanoi Algorithm:

Algorithm: Tower of Hanoi \((n, i, j)\)

Step 1: Move \( n \) disks from Peg \( i \) to Peg \( j \)

Input: Integer \( n > 0 \), \( 1 \leq i, j \leq 3 \), \( i \neq j \)

Output: Specifies disk moves in correct order

Input Size: Use number \( n \)

Basic Operation: moving a disk
if \( n = 1 \) then
move disk 1 from peg i to peg j
else
Towers \((n-1, i, b-i-j)\)
move disk 1 from peg i to peg j
Towers \((n-1, b-i-j, i)\)

Recurrence for Tower of Hanoi:
- Let \( M(n) \) = more count to compute Towers \((n, \cdot, \cdot)\)
- \( M(1) = 1 \), because 1 move is needed for Towers \((1, \cdot, \cdot)\)
- If \( n > 1 \), then towers \((n, \cdot, \cdot)\) performs 2 recursive calls plus
  one more.
  \[
  M(n) = 2M(n-1) + 1
  \]

Make a reasonable guess.
Forward Substitution:
- \( M(2) = 2M(1) + 1 = 3 \)
- \( M(3) = 2M(2) + 1 = 7 \)
- \( M(4) = 2M(3) + 1 = 15 \)

Prove \( M(n) = 2^n - 1 \) by mathematical induction

\[ \text{Basis:} \quad \text{if} \quad n = 1, \quad \text{then} \quad M(n) = 1 = 2^n - 1 \]

\[ \text{Induction:} \quad \text{if} \quad M(n-1) = 2^{n-1} - 1, \quad \text{then} \]
\[
M(n) = 2M(n-1) + 1 = 2 \times (2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1
\]

Time complexity of Tower of Hanoi is \( \Theta(2^n) \)