<table style="height: 120px;" width="203">

<tbody>

<tr>

<td style="width: 193.76px;">

<p>Annam Dheeraj</p>

<p>RA1811029010032</p>

<p>CSE-CN-L2</p>

</td>

</tr>

</tbody>

</table>

<p>&nbsp;</p>

<h1>DESIGN ANALYSIS AND ALGORITHM</h1>

<p>An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space.</p>

<p>An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way. If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the&nbsp;<strong>algorithm is independent from any programming languages</strong>.</p>

<h2>Algorithm Design</h2>

<p>The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.</p>

<p>To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. However, one has to keep in mind that both time consumption and memory usage cannot be optimized simultaneously. If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.</p>

<h2>Problem Development Steps</h2>

<p>The following steps are involved in solving computational problems.</p>

<ul class="list">

<li>Problem definition</li>

<li>Development of a model</li>

<li>Specification of an Algorithm</li>

<li>Designing an Algorithm</li>

<li>Checking the correctness of an Algorithm</li>

<li>Analysis of an Algorithm</li>

<li>Implementation of an Algorithm</li>

<li>Program testing</li>

<li>Documentation</li>

</ul>

<h2>Characteristics of Algorithms</h2>

<p>The main characteristics of algorithms are as follows &minus;</p>

<ul class="list">

<li>

<p>Algorithms must have a unique name</p>

</li>

<li>

<p>Algorithms should have explicitly defined set of inputs and outputs</p>

</li>

<li>

<p>Algorithms are well-ordered with unambiguous operations</p>

</li>

<li>

<p>Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point</p>

</li>

</ul>

<h2>Pseudocode</h2>

<p>Pseudocode gives a high-level description of an algorithm without the ambiguity associated with plain text but also without the need to know the syntax of a particular programming language.</p>

<p>The running time can be estimated in a more general manner by using Pseudocode to represent the algorithm as a set of fundamental operations which can then be counted.</p>

<h2>Difference between Algorithm and Pseudocode</h2>

<p>An algorithm is a formal definition with some specific characteristics that describes a process, which could be executed by a Turing-complete computer machine to perform a specific task. Generally, the word "algorithm" can be used to describe any high level task in computer science.</p>

<p>On the other hand, pseudocode is an informal and (often rudimentary) human readable description of an algorithm leaving many granular details of it. Writing a pseudocode has no restriction of styles and its only objective is to describe the high level steps of algorithm in a much realistic manner in natural language.</p>

<p>For example, following is an algorithm for Insertion Sort.</p>

<pre class="result notranslate"><strong>Algorithm: Insertion-Sort</strong>

Input: A list L of integers of length n

Output: A sorted list L1 containing those integers present in L

Step 1: Keep a sorted list L1 which starts off empty

Step 2: Perform Step 3 for each element in the original list L

Step 3: Insert it into the correct position in the sorted list L1.

Step 4: Return the sorted list

Step 5: Stop

</pre>

<p>Here is a pseudocode which describes how the high level abstract process mentioned above in the algorithm Insertion-Sort could be described in a more realistic way.</p>

<pre class="result notranslate">for i &lt;- 1 to length(A)

x &lt;- A[i]

j &lt;- i

while j &gt; 0 and A[j-1] &gt; x

A[j] &lt;- A[j-1]

j &lt;- j - 1

A[j] &lt;- x<br /><br /><br /></pre>

<h1 class="qa\_title">Convex Hull</h1>

<p>Here we will see one example on convex hull. Suppose we have a set of points. We have to make a polygon by taking less amount of points, that will cover all given points. In this section we will see the Jarvis March algorithm to get the convex hull.</p>

<p>Jarvis March algorithm is used to detect the corner points of a convex hull from a given set of data points.</p>

<p>Starting from left most point of the data set, we keep the points in the convex hull by anti-clockwise rotation. From a current point, we can choose the next point by checking the orientations of those points from current point. When the angle is largest, the point is chosen. After completing all points, when the next point is the start point, stop the algorithm.</p>

<p><strong>Input</strong>&nbsp;&minus; Set of points: {(-7,8), (-4,6), (2,6), (6,4), (8,6), (7,-2), (4,-6), (8,-7),(0,0), (3,-2),(6,-10),(0,-6),(-9,-5),(-8,-2),(-8,0),(-10,3),(-2,2),(-10,4)}</p>

<p><strong>Output</strong>&nbsp;&minus; Boundary points of convex hull are &minus;</p>

<pre class="result notranslate">(-9, -5) (6, -10) (8, -7) (8, 6) (-7, 8) (-10, 4) (-10, 3)</pre>

<h2>Algorithm</h2>

<pre class="result notranslate">findConvexHull(points, n)

Input: The points, number of points.

Output: Corner points of convex hull.

Begin

&nbsp; &nbsp;start := points[0]

&nbsp; &nbsp;for each point i, do

&nbsp; &nbsp; &nbsp; if points[i].x &lt; start.x, then // get the left most point

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp;start := points[i]

&nbsp; &nbsp;done

&nbsp; &nbsp;current := start

&nbsp; &nbsp;add start point to the result set.

&nbsp; &nbsp;define colPts set to store collinear points

&nbsp; &nbsp;while true, do //start an infinite loop

&nbsp; &nbsp; &nbsp; next := points[i]

&nbsp; &nbsp; &nbsp; for all points i except 0th point, do

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp;if points[i] = current, then

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; skip the next part, go for next iteration

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp;val := cross product of current, next, points[i]

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp;if val &gt; 0, then

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; next := points[i]

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; clear the colPts array

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp;else if cal = 0, then

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; if next is closer to current than points[i], then

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp;add next in the colPts

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp;next := points[i]

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; else

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp;add points[i] in the colPts

&nbsp; &nbsp; &nbsp; done

&nbsp; &nbsp; &nbsp; add all items in the colPts into the result

&nbsp; &nbsp; &nbsp; if next = start, then

&nbsp; &nbsp; &nbsp; &nbsp; &nbsp;break the loop

&nbsp; &nbsp; &nbsp; insert next into the result

&nbsp; &nbsp; &nbsp; current := next

&nbsp; &nbsp;done

&nbsp; &nbsp;return result

End</pre>