Birla Institute of Technology & Science, Pilani
Hyderabad Campus

DESIGN AND ANALYSIS OF ALGORITHMS

Project Report
CS F364 (DAA)

Submitted By:

Siddhant Panda	2020A7PS0264H
Archis Sahu	2020A7PS1692H
Raghuvir Singh	2020A7PS2061H
Dev Bansal	2020A7PS2051H

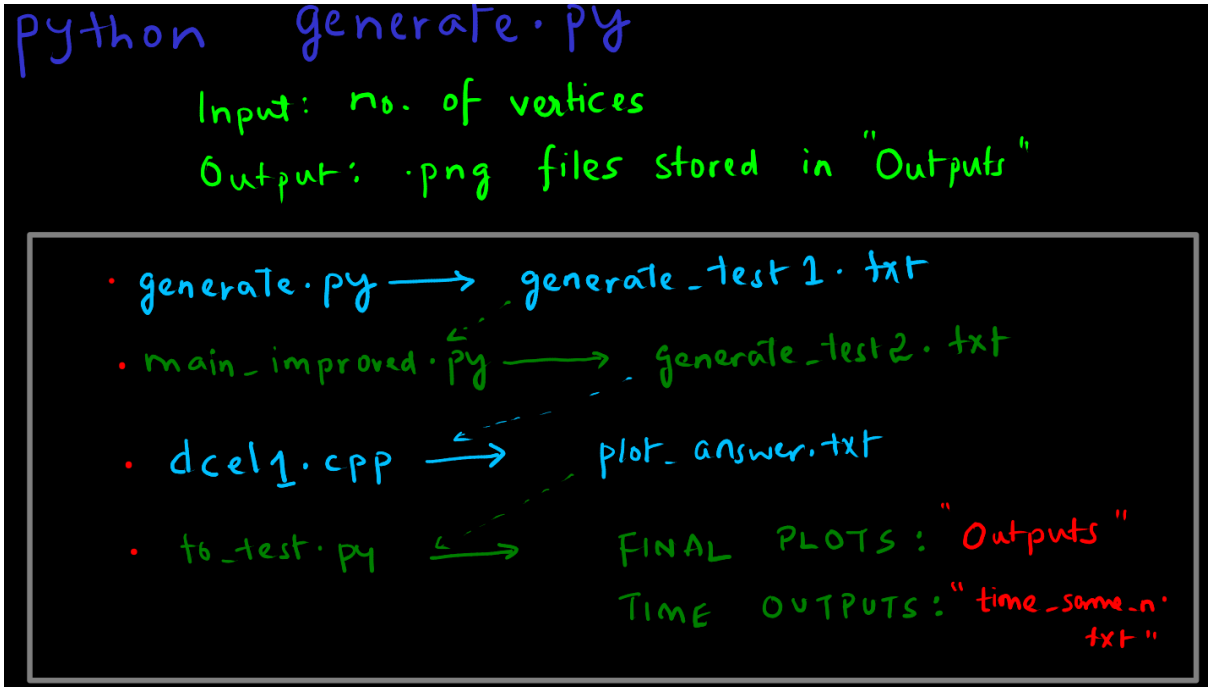
TABLE OF CONTENTS

Motivation :

- To understand and solidify our understanding of the design and analysis of decomposing any arbitrary polygon into convex polygon.
- In the following report, we understand the limitations and capabilities of the algorithm implemented, as mentioned in the given research paper, and the

object-oriented paradigm of code facilitated through the use of DCEL data structure.

File Structure :

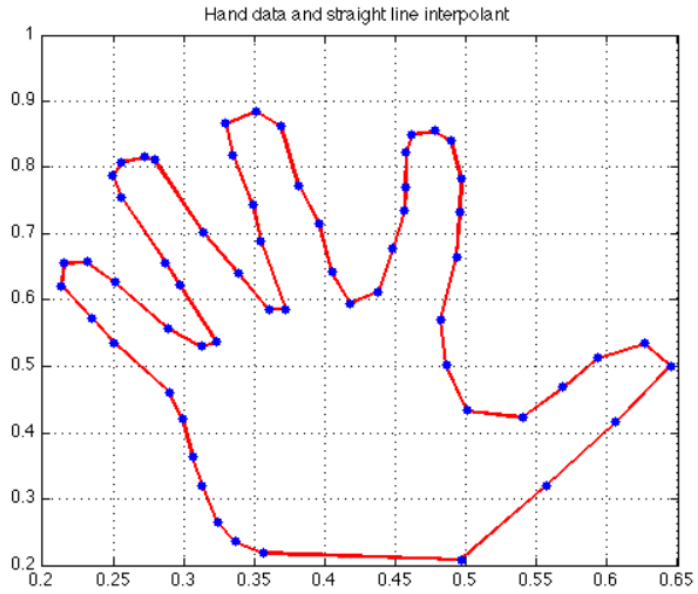


- The above flowchart shows the file structure of the code, where in generate.py file is used to generate random simple polygons with given no. of input vertices, which then calls the files as shown above using system calls, to finally give the output of the dissociated convex polygon as a .png file, and the time analysis is also recorded in a separate text file, later used to plot it.

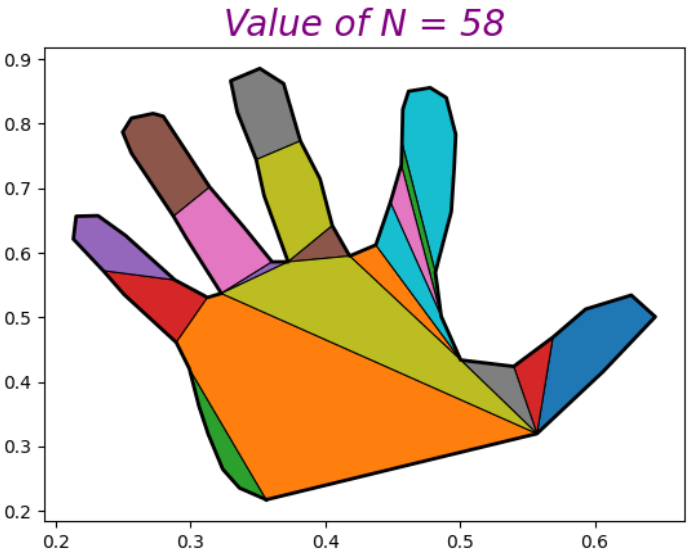
Visual Correctness and output of code on few well known cases :

- Human hand :

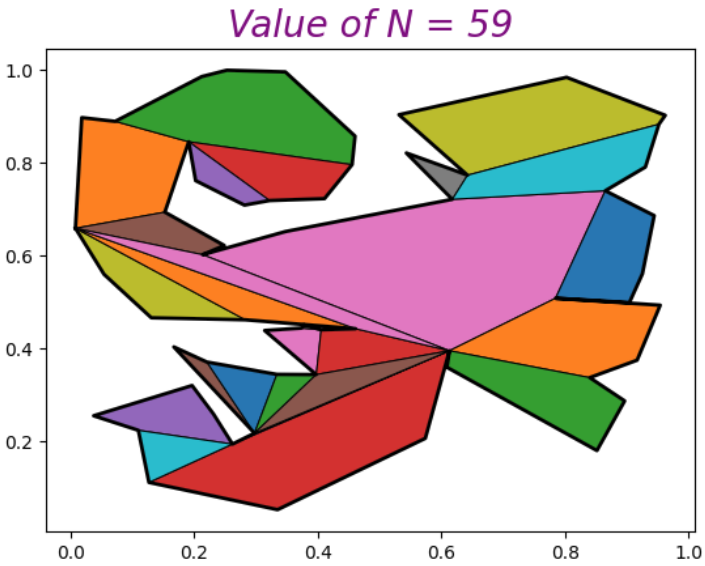
  - ❖ Unprocessed graph :



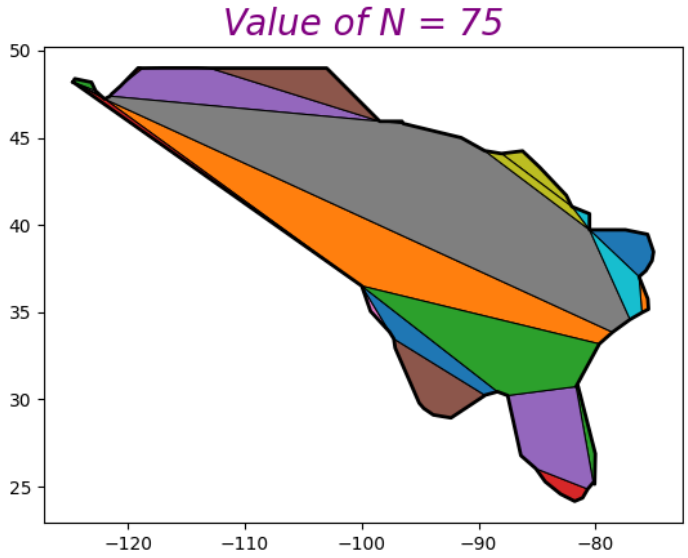
  - ❖ Processed graph :



- Scorpion :



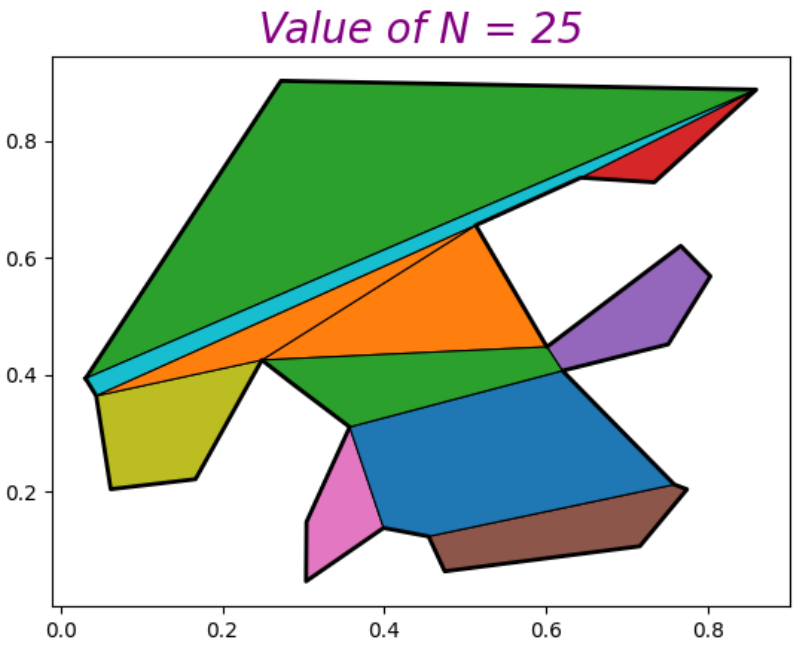
- USA Map :



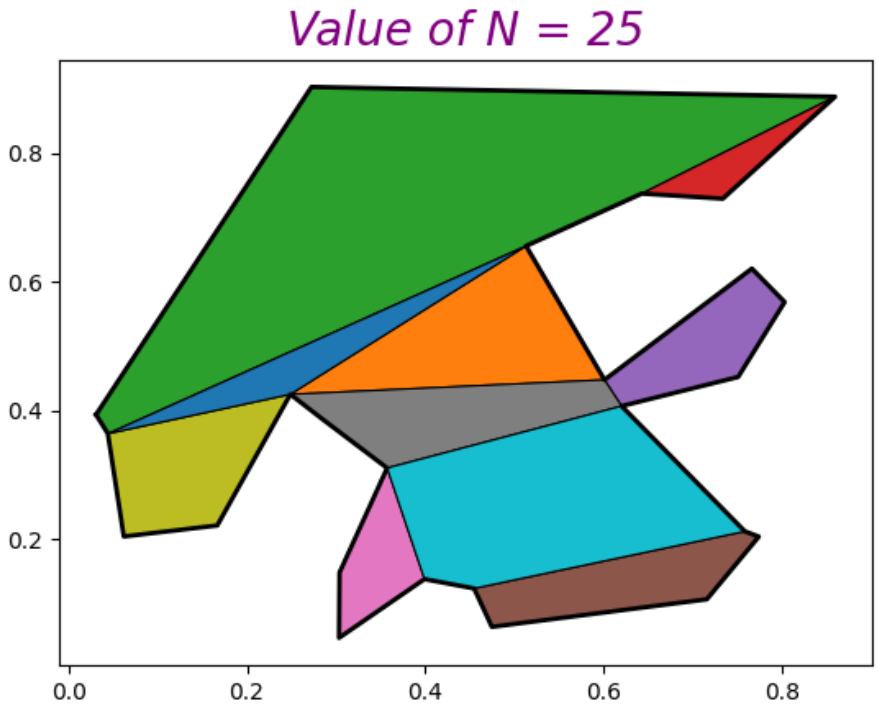
- The process of converting into convex polygon encompasses two stages -
    1. the *partitioning* of the edges in an apt manner, and
    2. the *merging* of redundant edges that do not cause the problem of notches in the given dataset. We refer to the second step as the ***merging process***.

Unmerged vs merged Processes :

- For N = 25 vertices

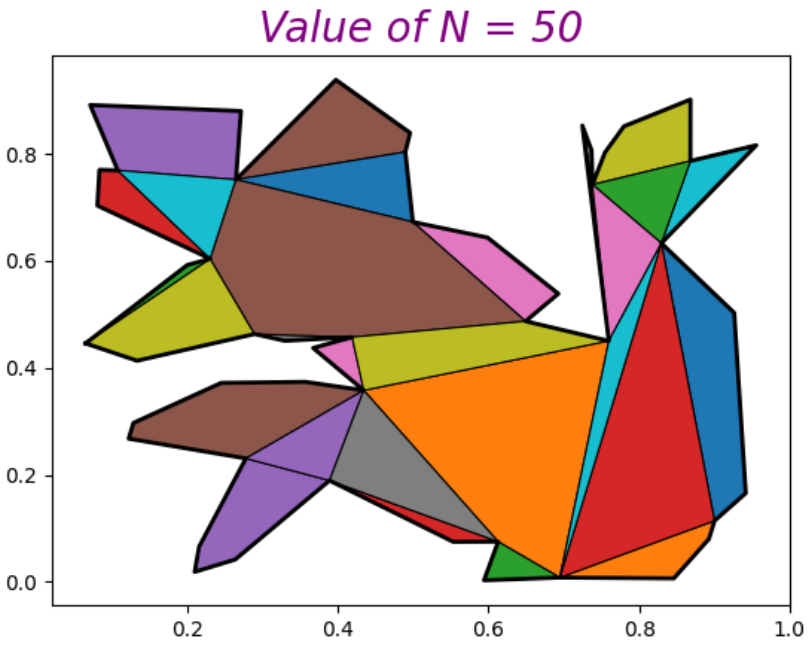


*The above picture shows the convex polygon that is only partitioned, but not merged

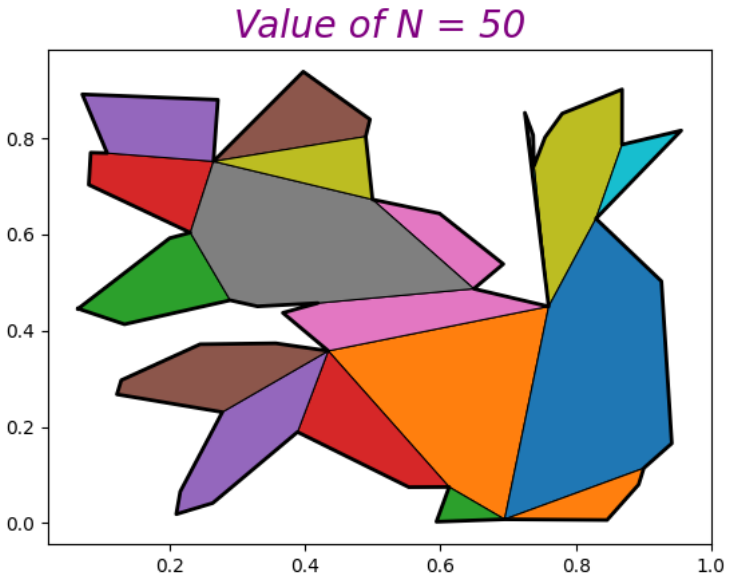


This picture shows the ***merged*** and ***partitioned*** convex polygon. The merging process adds a bit of an extra overhead, which can be compared in the timing graphs later in the report.

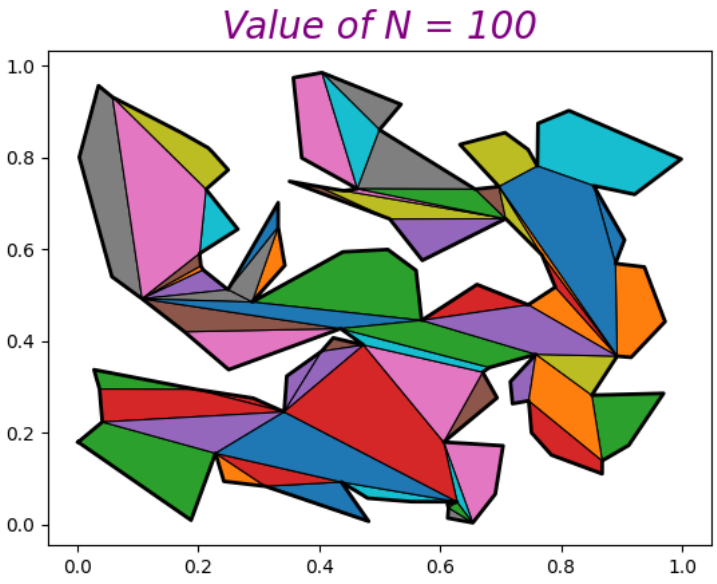
- For N = 50 vertices :



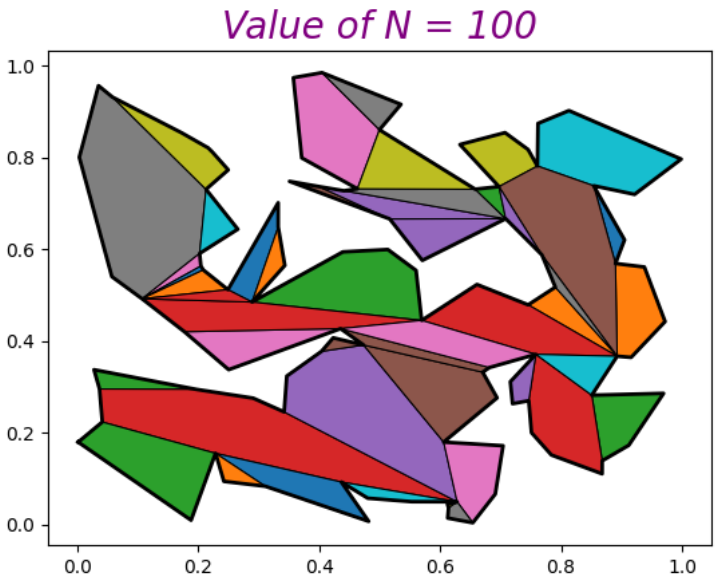
The above process is unmerged, and the following picture clearly shows the merged convex polygon :



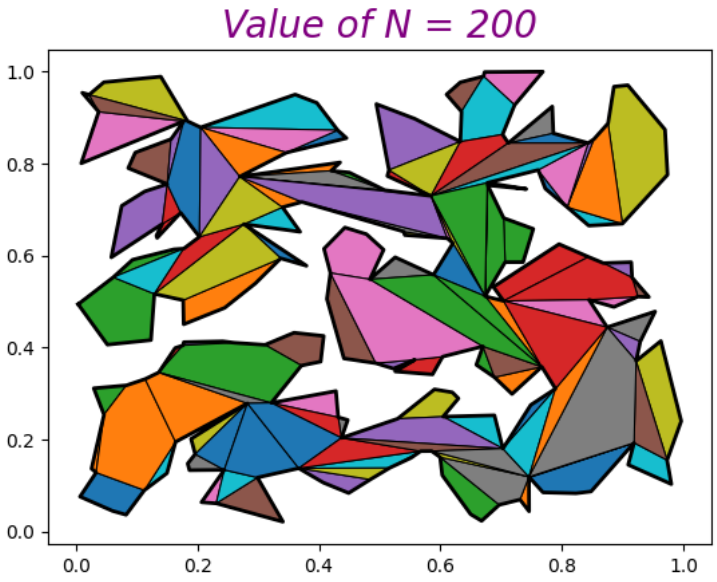
- N = 100 vertices :



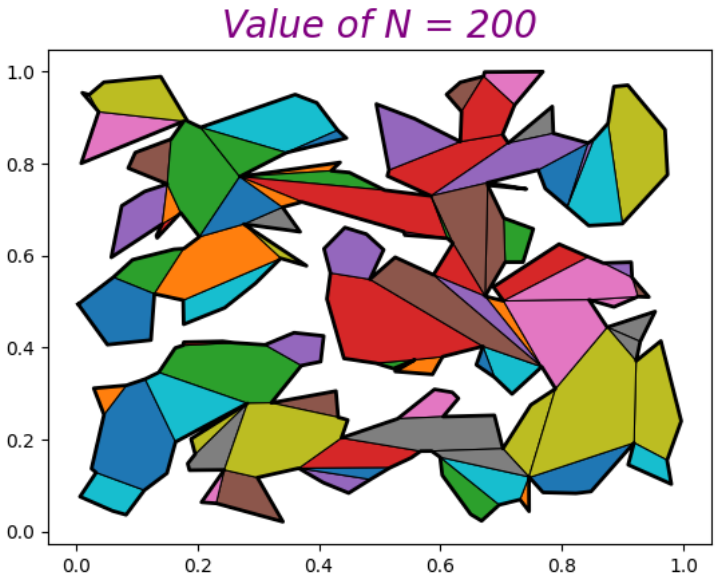
The above graph is unmerged, and the following graph shows the merged polygon :



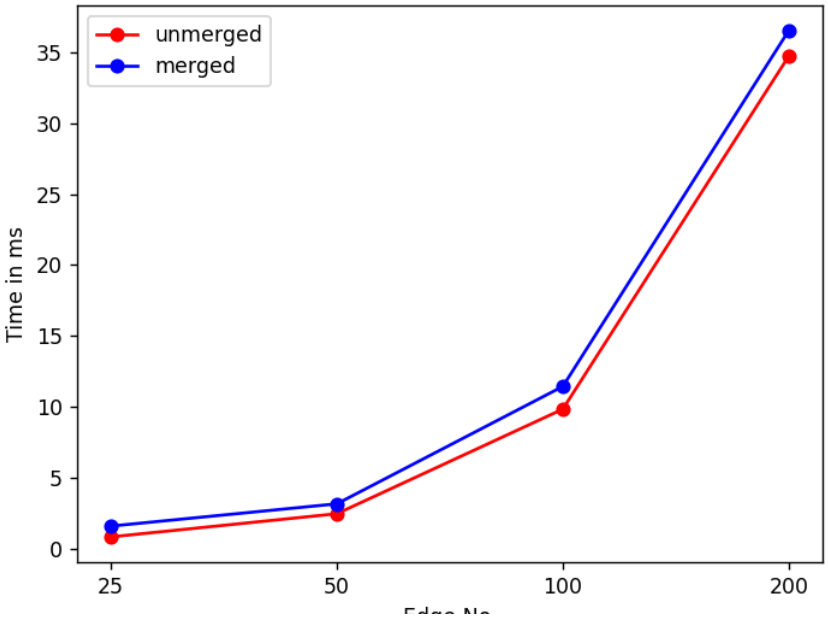
- N = 200 Vertices :



The merged one looks as follows :



Timing analysis of these both algorithms – MERGED v/s UNMERGED :

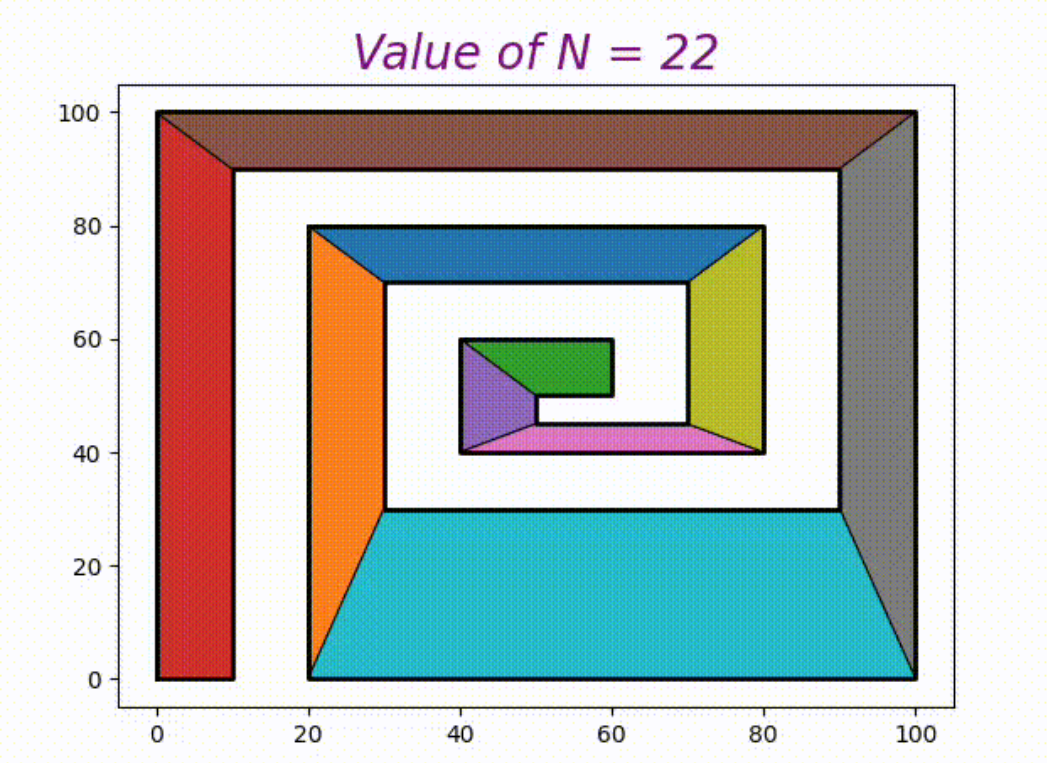


- We see that there is a very close correlation between how the merged and unmerged algorithms work with respect to different input sizes.
- This correlation can be explained by how merging various redundant edges results in an extra overhead of going through a lot of edges and finding out if removing them would result in a notch in the graph.

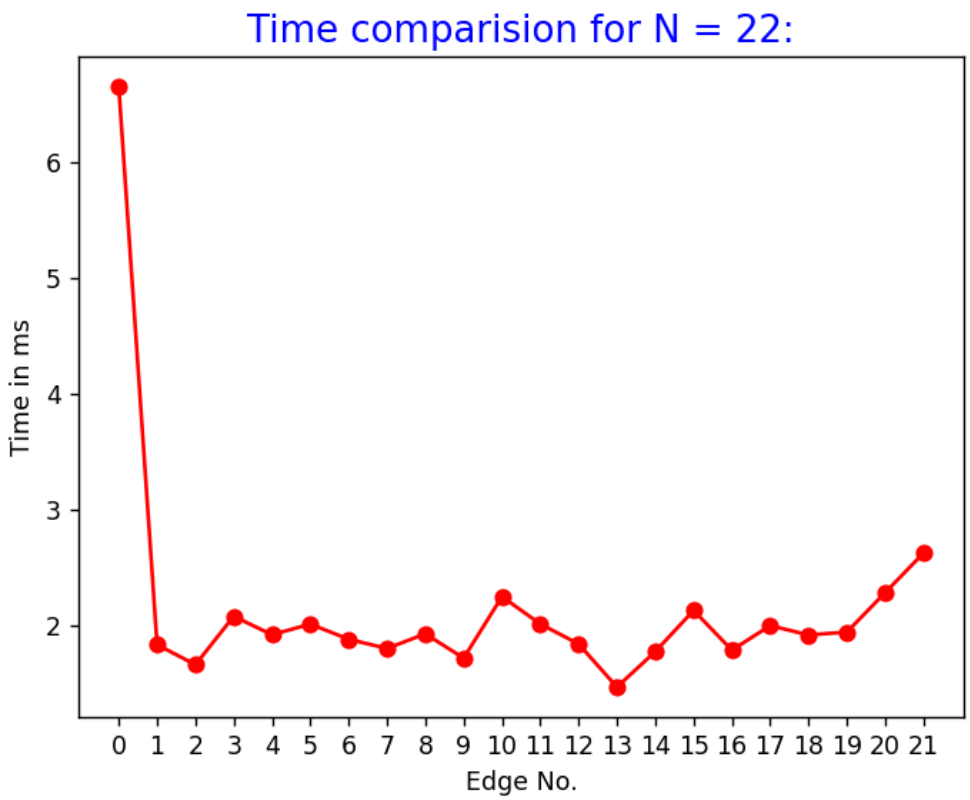
CHOOSING DIFFERENT STARTING POINTS FOR A GIVEN INPUT :

- Upon various experimentations, we found out that the starting point of the algorithm also determines how the polygon is partitioned, and hence also affects the running time of the algorithm, though not predictably.

- Spiral Graph

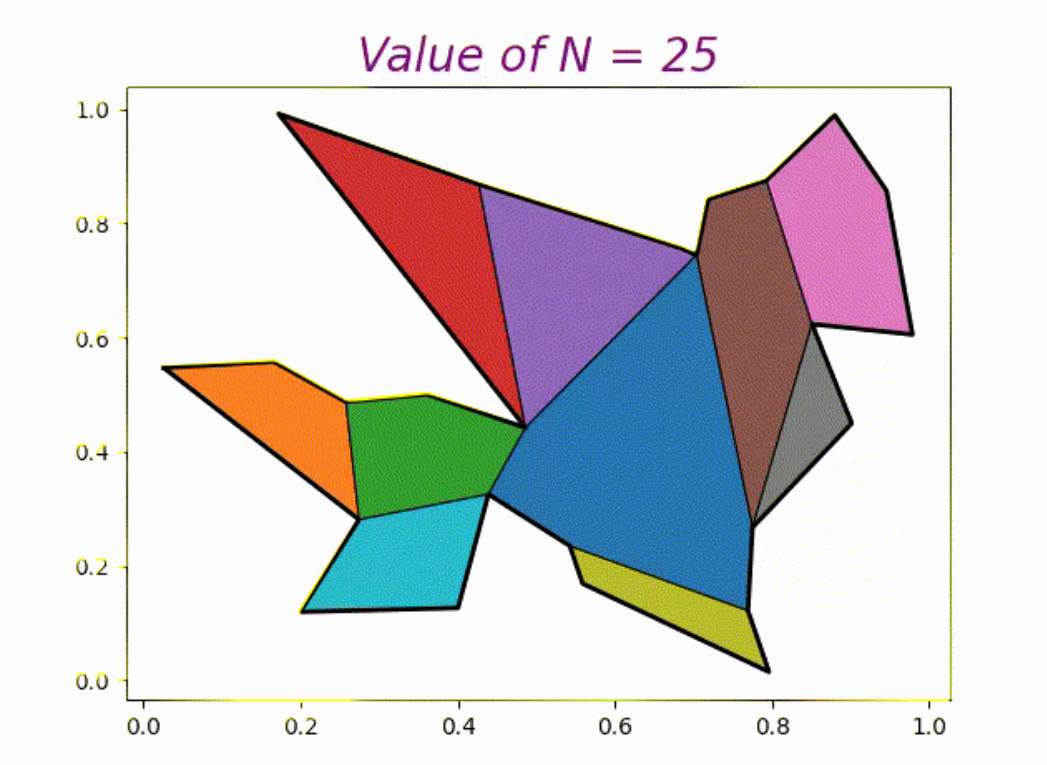


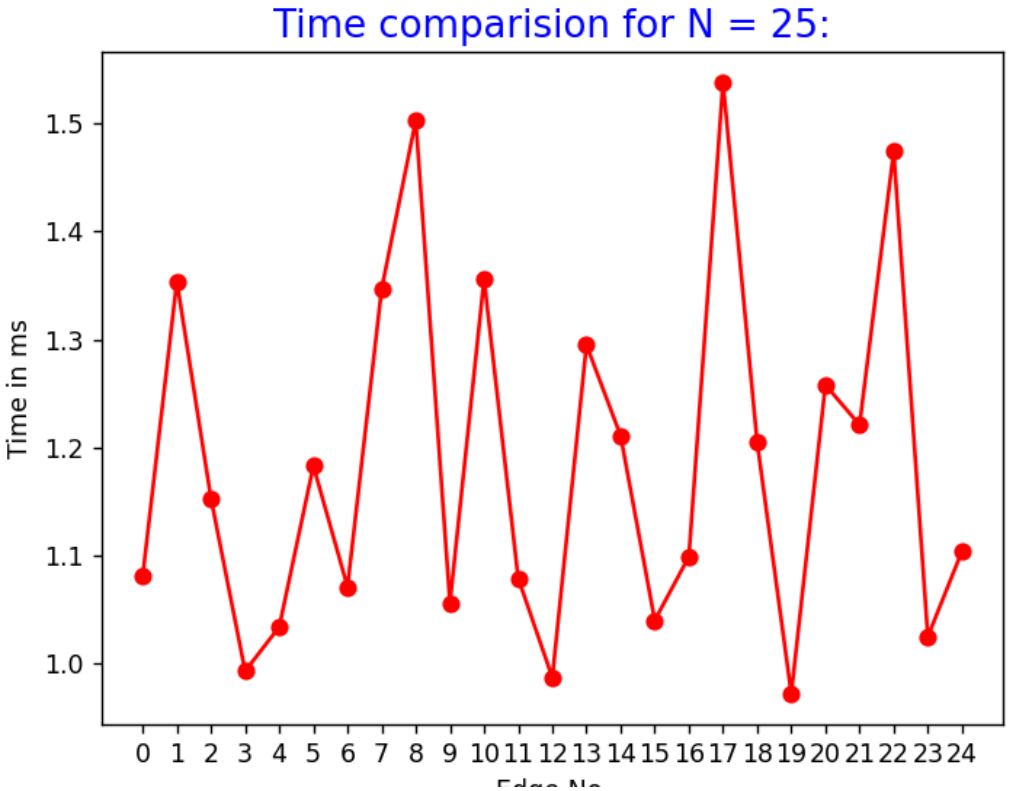
  - The following has 22 vertices, and the timing analysis is shown considering each of these edges as the starting edge.



- We notice that for edge 0, that is when the program is initially started, it shows a high peak, which can be a consequence of the program just starting and the CPU allocating its resources, and hence leading to a higher activity.
- The following starting edges, show more or less the same run time complexity of about 1.8 ms on an average, which makes sense as we can see through the GIF above, that on changing starting edges, the partitioned polygons don't vary much, and hence there is no reason for the runtime to vary indiscriminately.
- But we further see that this is not always the case with all test cases.

- RANDOM POLYGON WITH N = 25 vertices :



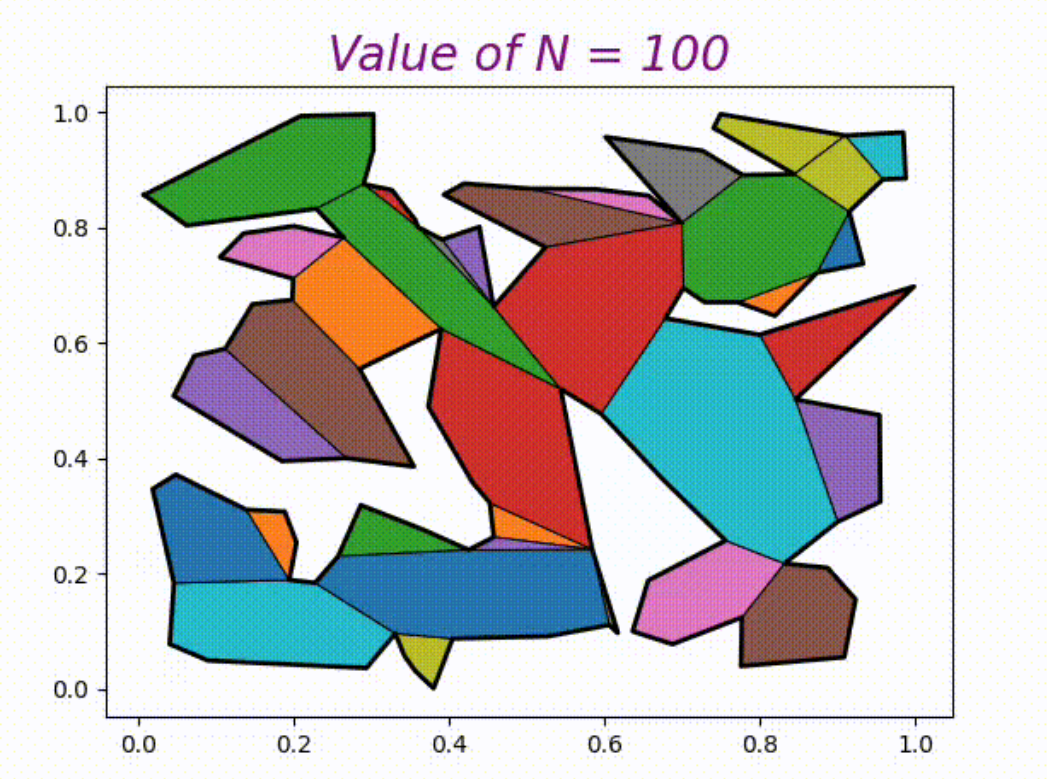


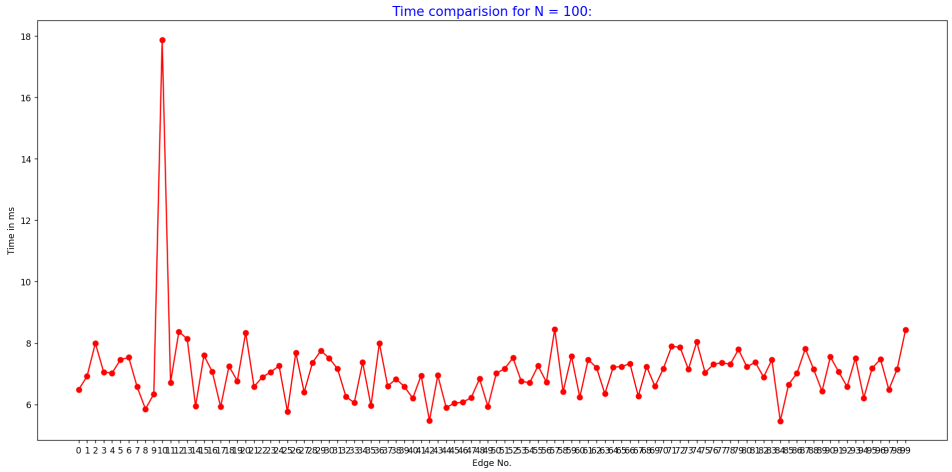
- We notice that there is no specific reason for the spikes, and since these are only run time analysis, it also depends on

various other factors like background processes running and the state of the computer at that point of time.

- But more or less, the run time remains constant at an average of about 1.3 ms for all possibilities of starting edges.

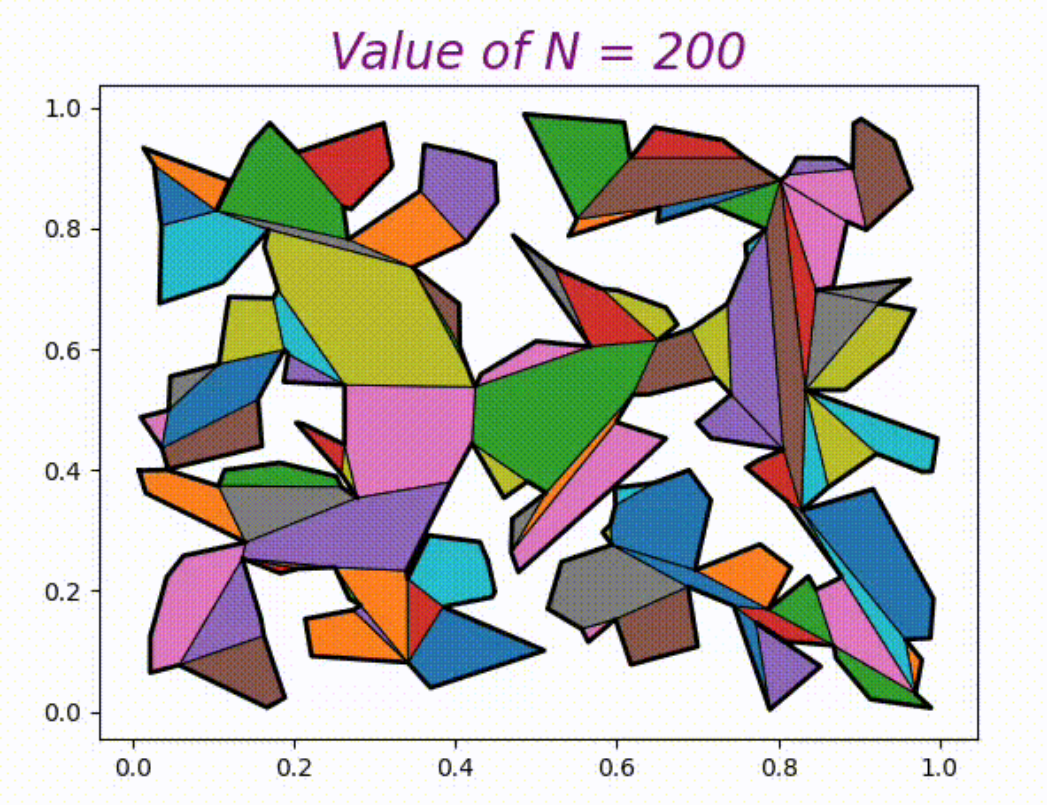
- RANDOM POLYGON WITH 100 VERTICES :

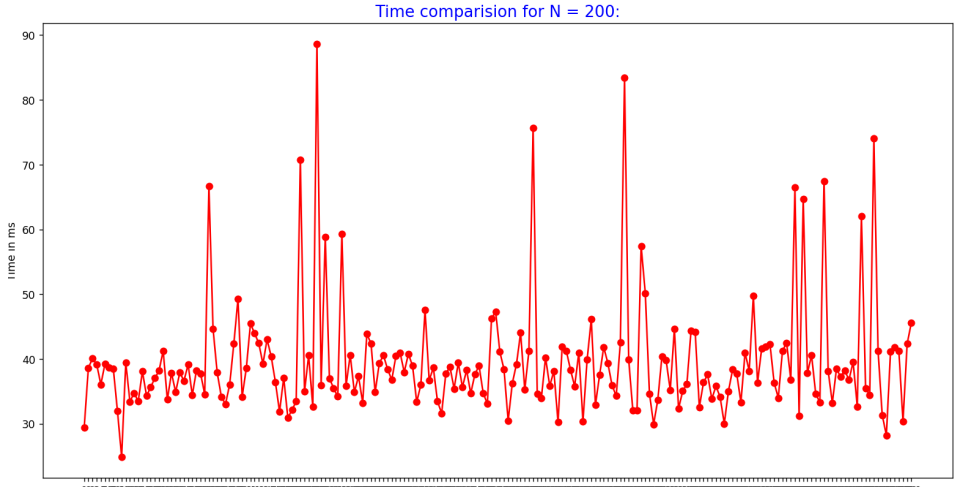




- Here we notice the same, that more or less the run time remains same for all possible edges, except for one that is at edge 10, which is an exceptional case in this randomly generated polygon, where the overhead of continuous notch checking occurs and increases the run time drastically. This could also be added to by an increased background process in the computer while testing.

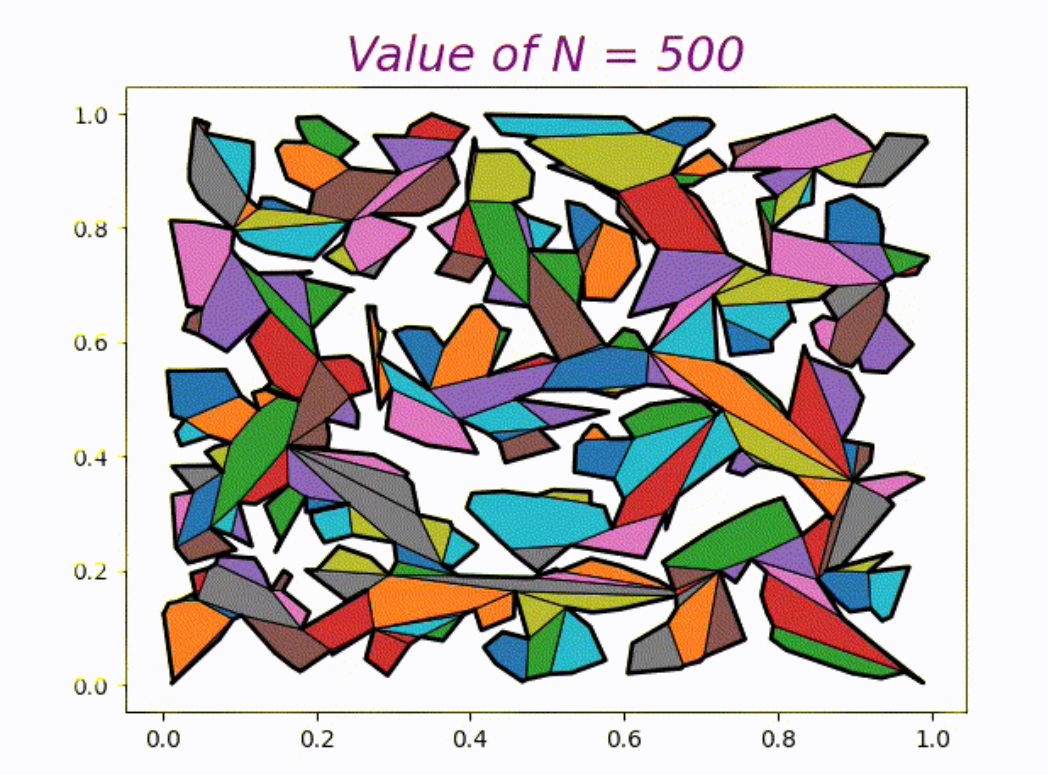
- RANDOM POLYGON WITH N = 200 vertices :

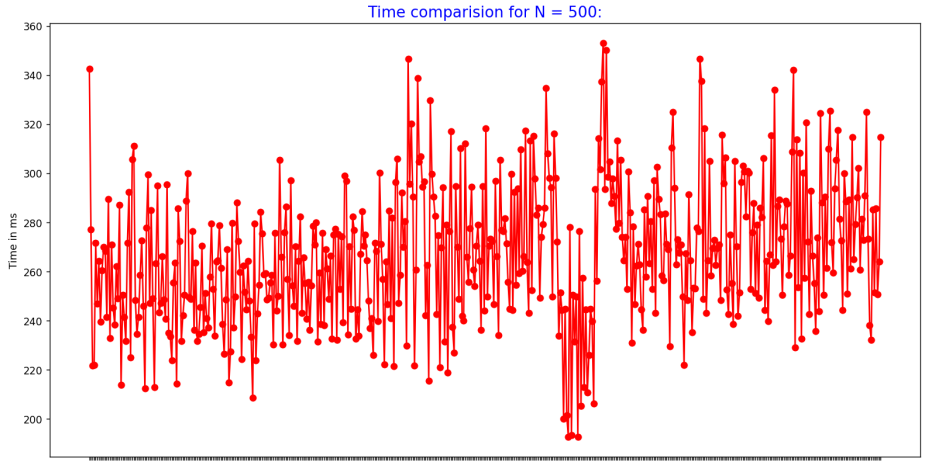




- The runtime has drastically increased here, and this means that the computer has reached a saturation of its resources, though the starting edge doesn't change the runtime significantly, we get to observe random spikes every now and then.

- RANDOM POLYGON WITH 500 vertices :



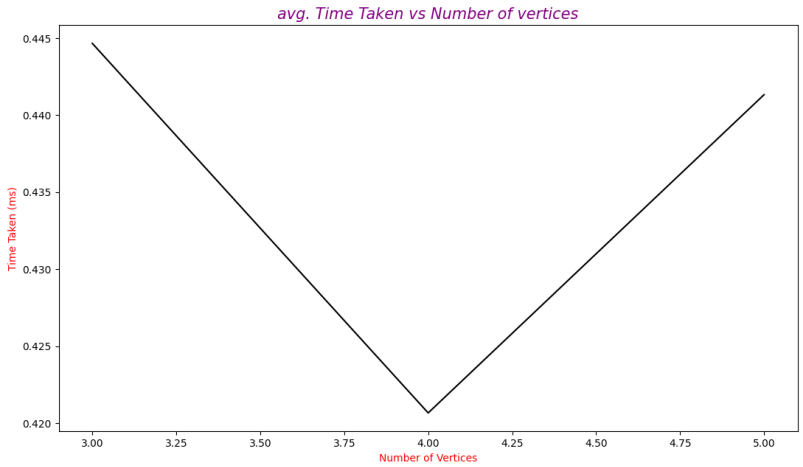


- Here the graph is even more cluttered, and we notice an even higher increase in the run time, following the reason stated above.

Time Analysis of Varying the input size of vertices :

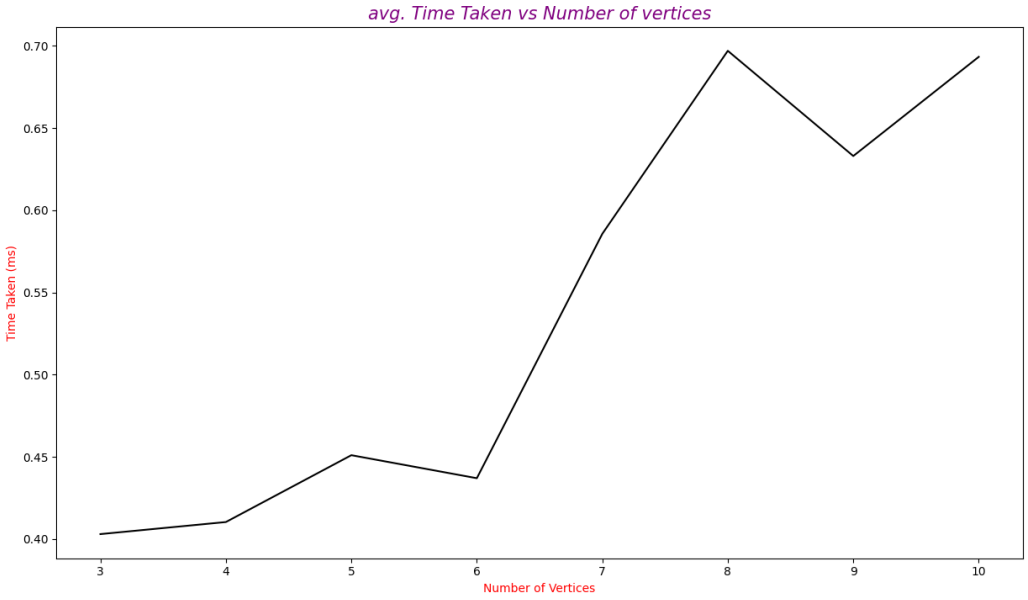
- In here, we have written a wrapper python script, which on given an input X, calculates the average run time of all inputs in the range from 3 to X vertices.
- The average of each y in interval [3, X] is calculated by generating 3 random samples of each y and then calculating the average time to generate those 3 samples.
- We have followed the above process for various inputs X and recorded the timing analysis :

- X = 5 vertices :

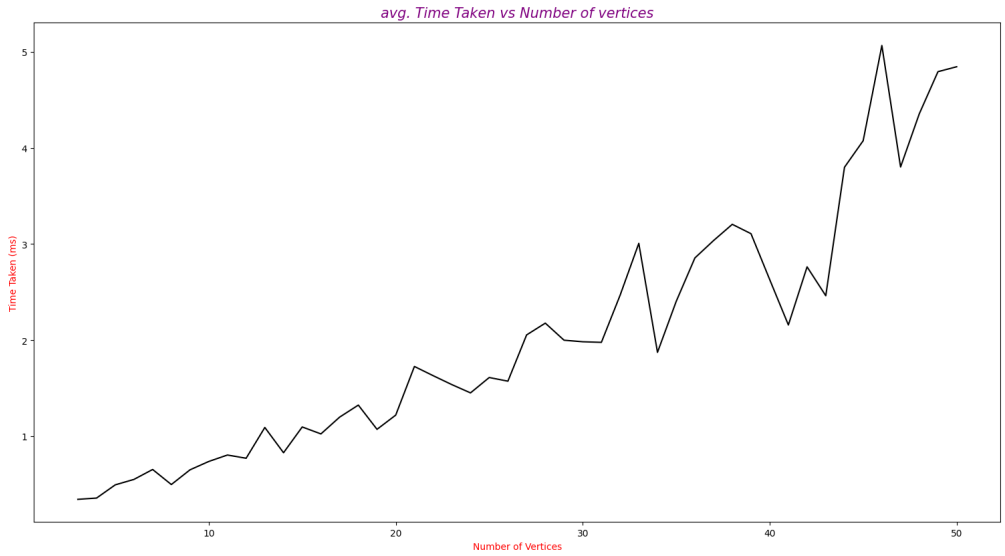


  - We see that for input size 3, the average time taken is at a spike, this is only due to the program just starting and CPU allocating its resources to the program to execute, we notice this graph getting much more general as and when we increase our input size

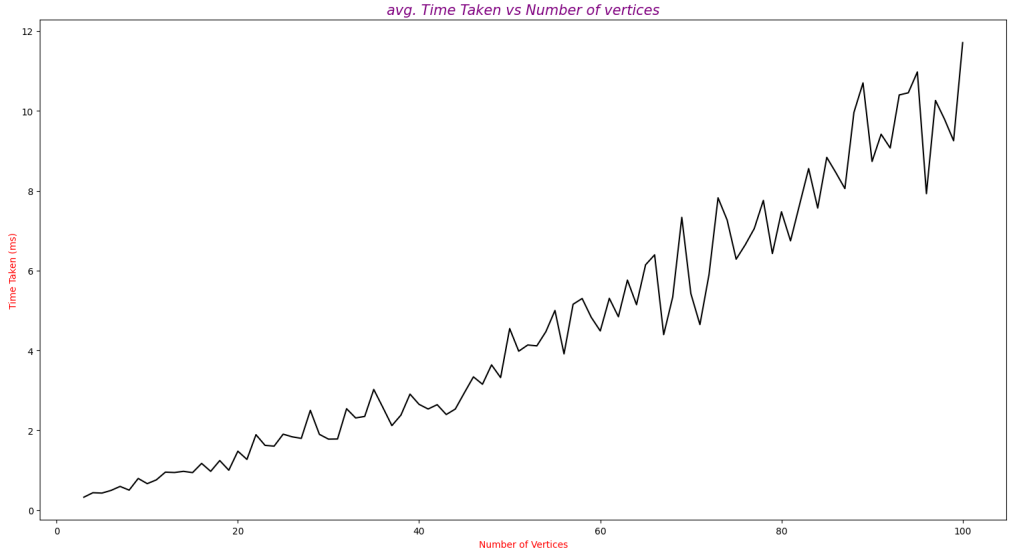
- X = 10 Vertices :



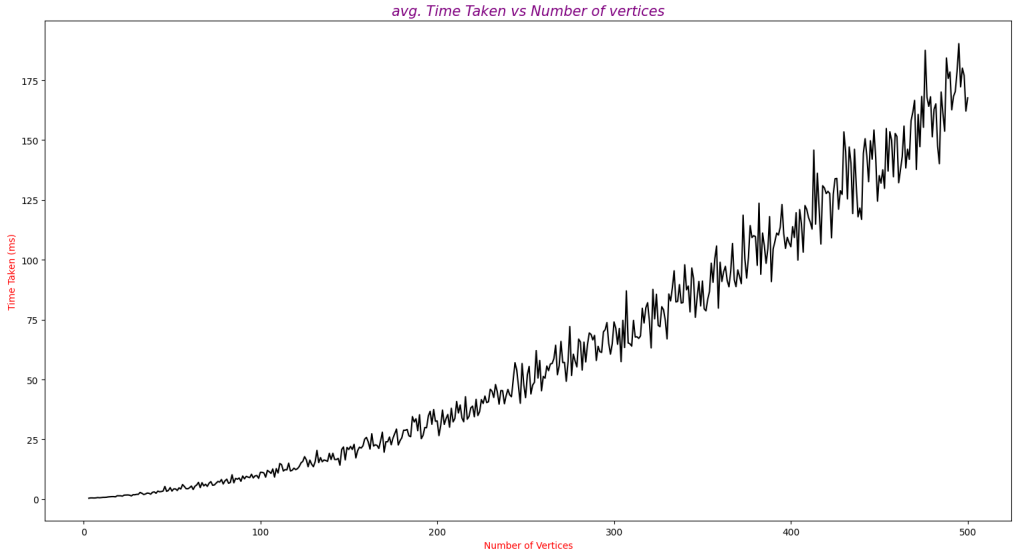
- X = 50 Vertices :


avg. Time Taken vs Number of vertices
Time Taken (ms)
Number of Vertices
1
2
3
4
5
10
20
30
40
50

- X = 100 vertices :



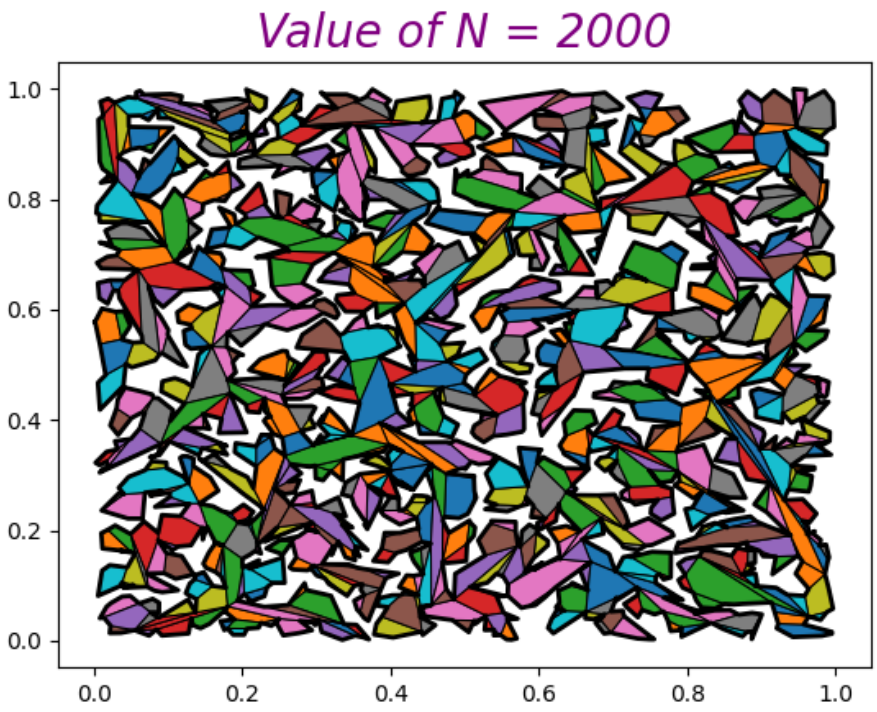
- X = 500 Vertices :



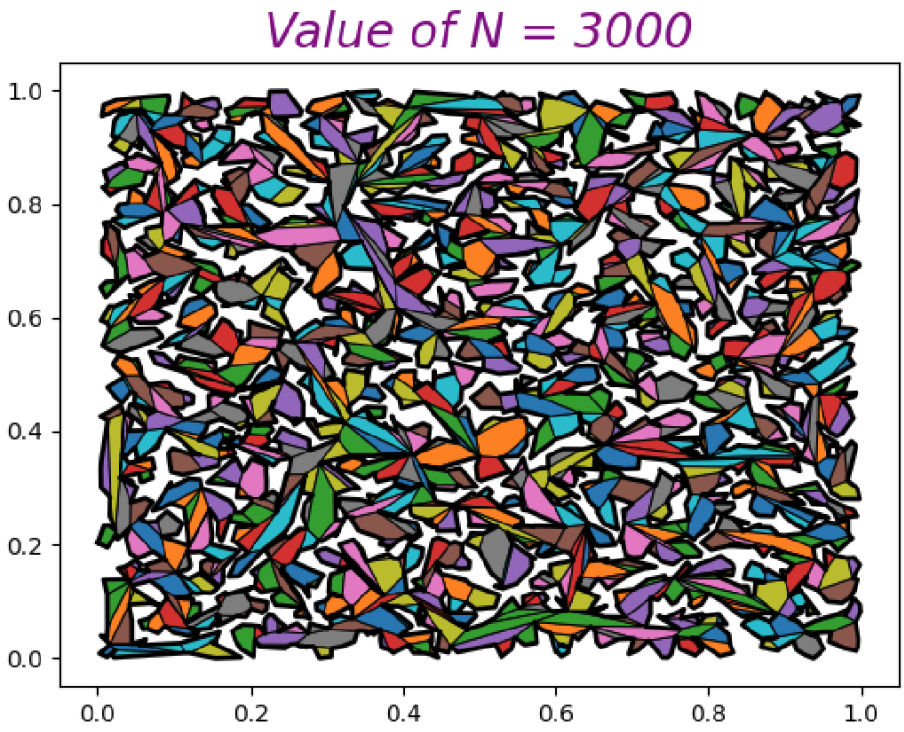
- From the above timing analysis graphs, we can see that the run time increases as and when we increase the input size, and it becomes more and more continuous and predictable as the size of the input increases, which is as expected.
- There is an explicit increase in the runtime with the input size, and after a point this increase is almost exponential, because this means that the resources are now at saturation, and hence causing the bottleneck.

● LIMITATIONS of the ALGORITHM :

- The following is the merged convex polygon generated for 2000 vertices, which took a run time of 3.18 seconds



- The algorithm also works efficiently for vertices of N = 3000, with an increased runtime of 8.7110 seconds, where the bottleneck is only generating a polygon with the given number of vertices.



- We notice that the algorithm works efficiently for even large cases, but the problem of generating a simple polygon in a restricted space with such huge number of vertices is very clumsy and impossible, which also is the bottleneck causing a limitation to the testing of this algorithm, which can obviously be improved with a better utilization of resources and computing power.