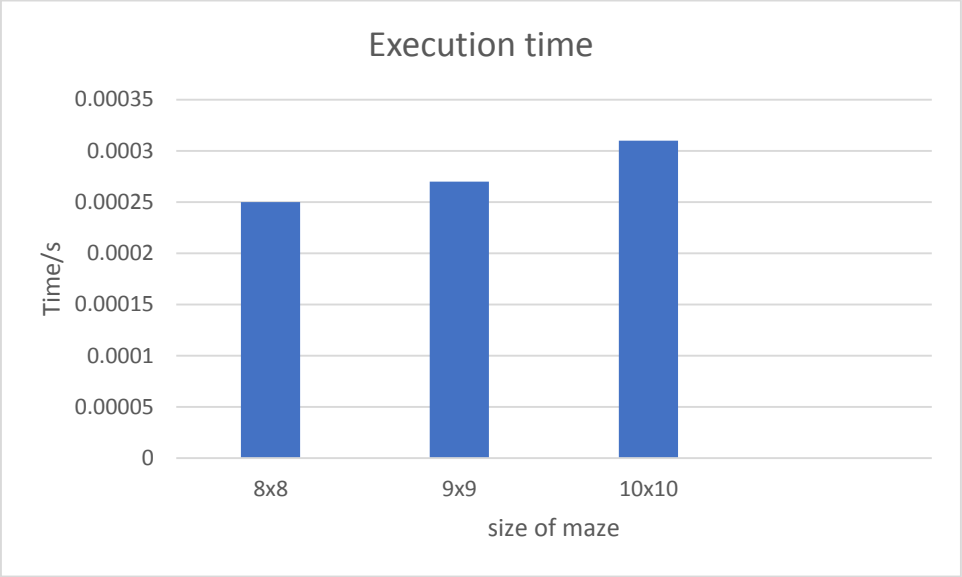
A-star

Maze solving algorithm:

size	Time/s
8x8	0.00025
9x9	0.00027
10x10	0.00031

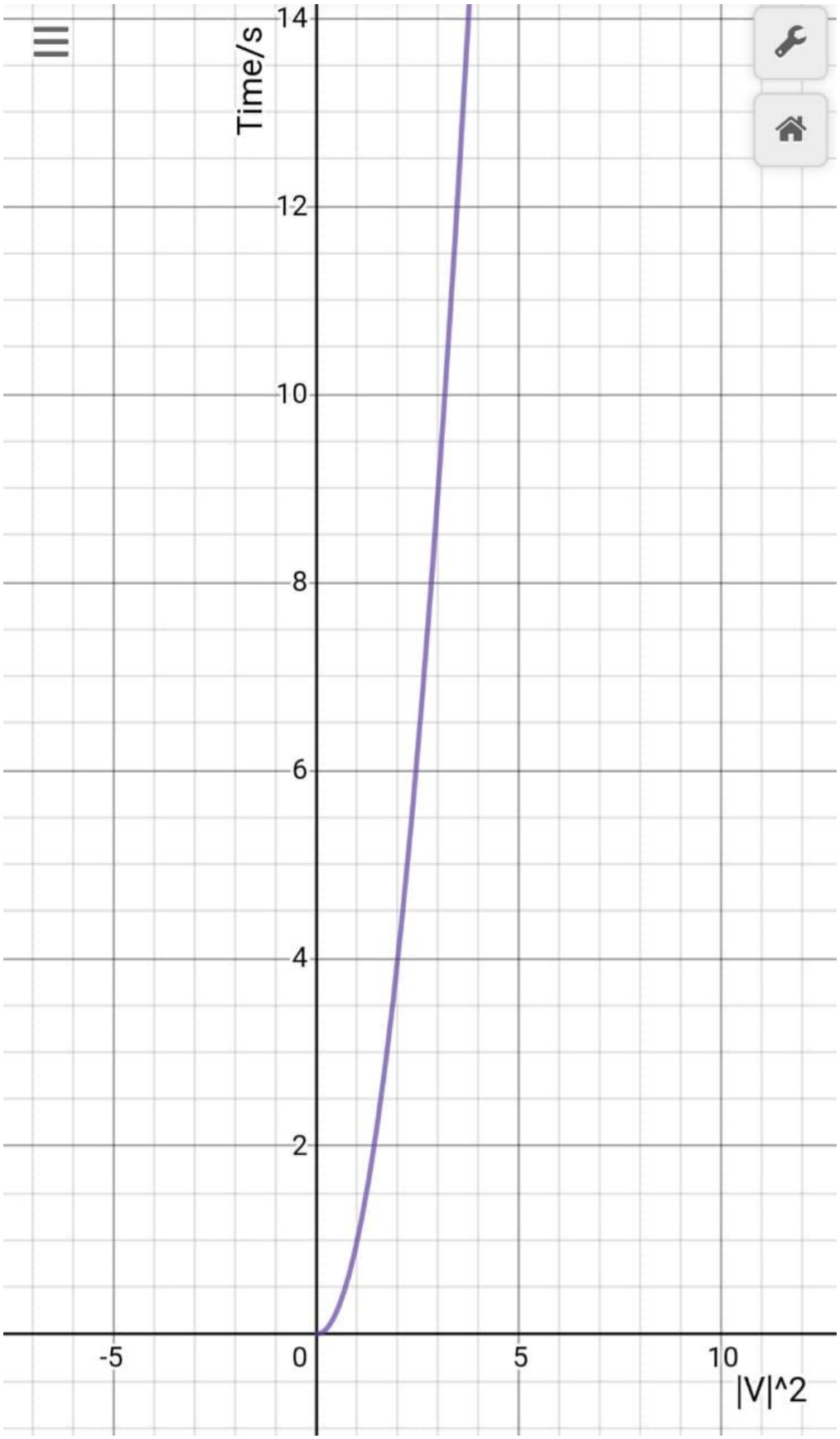


- This implementation has worst-case time complexity of $O(|V|^2)$ where 'V' represent vertices.

```
while len(open_list) > 0:
        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index
```

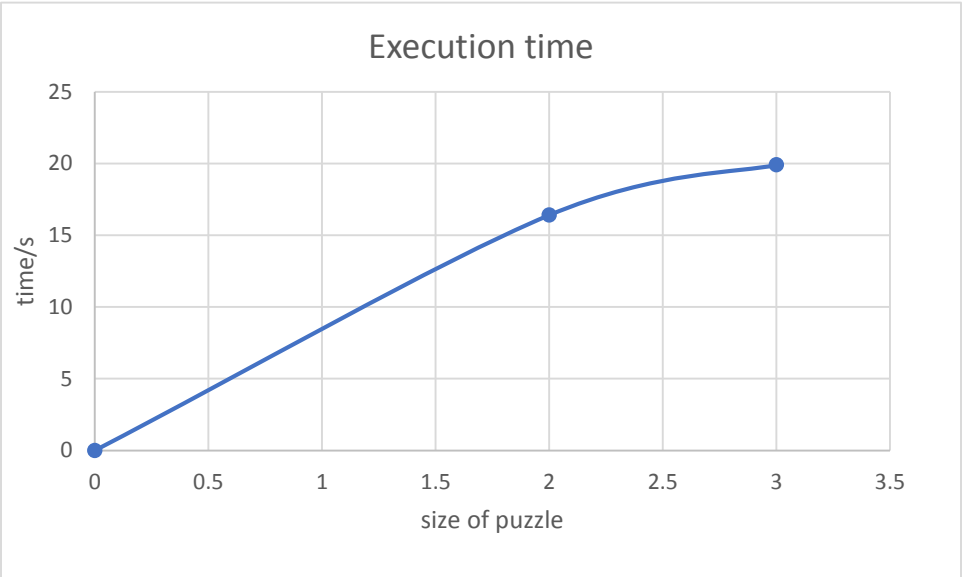
$$f(|V|) = \frac{|V|(|V|+1)}{2}$$
$$f(|V|) = |V|^2$$

In this code, while loop runs over |V| times in worst case, and we have a "for" loop inside it which also runs for |V| times. Therefore, it can be calculated that the time complexity is polynomial (quadratic), and thus has a parabolic graph as shown below.


Time/s
14
12
10
8
6
4
2
0
-5
5
10
|V|^2

Puzzle solving algorithm:

size	Time/s
0	0.00015
2	16.4
3	19.9

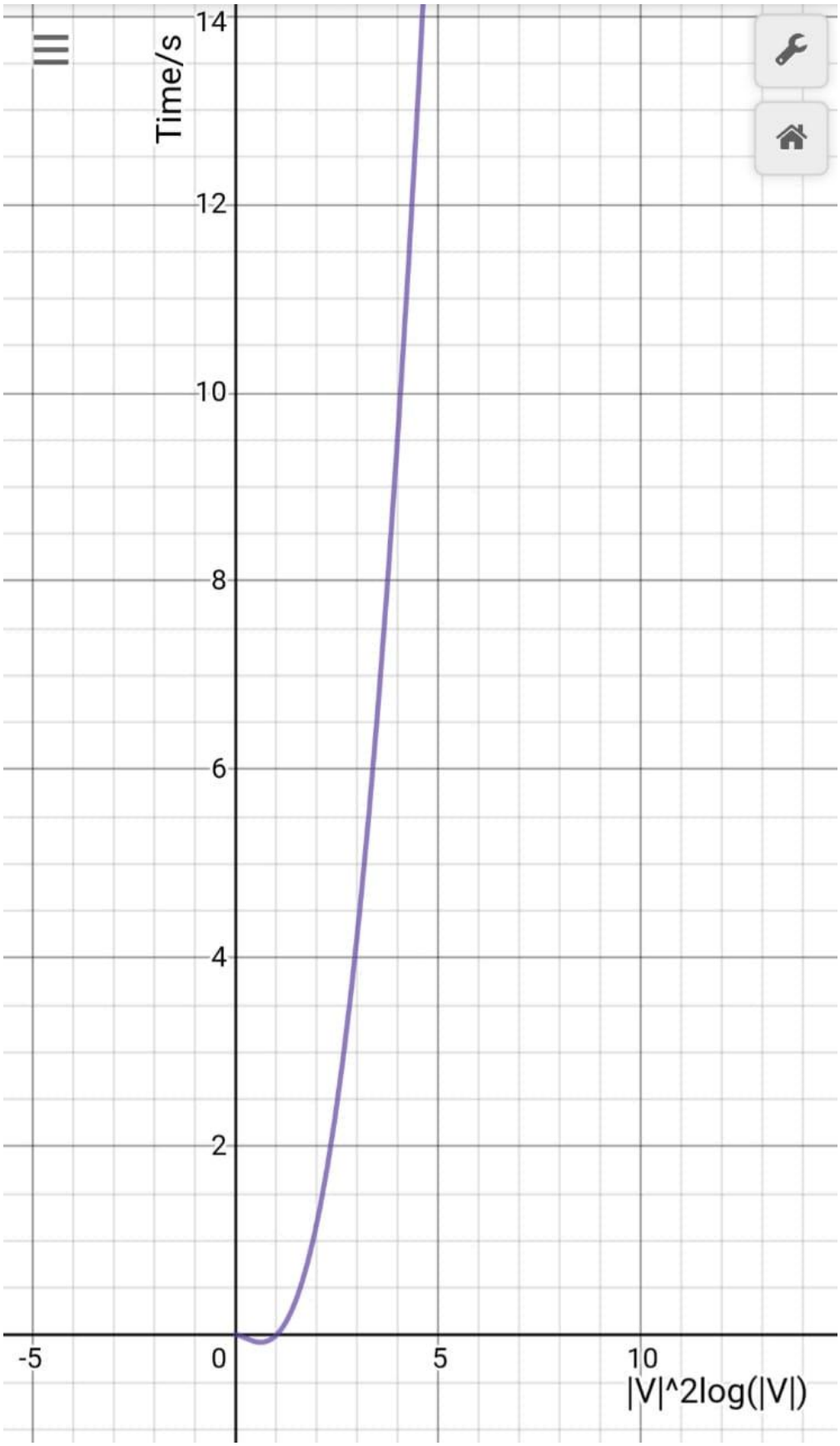



```
while len(self.open) > 0:
            cur = self.open[0]
            self.open.sort(key = lambda x:x.fval,reverse=False)
```

- This implementation has time complexity of $O(|V|^2.\log_2|V|)$ where 'n' is length of input while 'V' represents vertices.

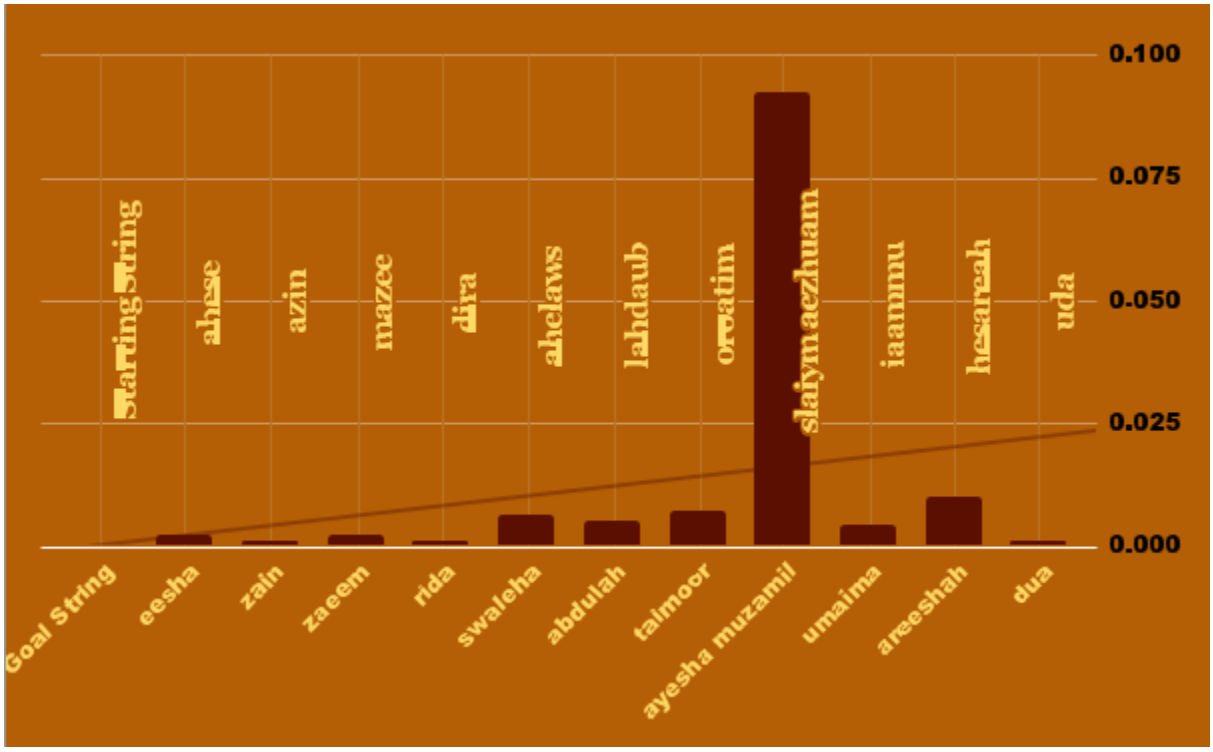
Here, it should be noted that the while loop iterates over |V| times in the worst case which implies that it will visit every node in the input. Furthermore, the built-in Python sort function has the complexity of $O(n.\log_2 n)$. Therefore, the total time complexity of the code becomes $O(|V|^2.\log_2|V|)$.

Its time complexity is linearithmic and the graph of the time complexity is shown below.


Time/s
14
12
10
8
6
4
2
0
-5
5
10
|V|^2log(|V|)

String Combination:

No. of times.	Starting String	Goal String	Time
1	ahese	eesha	0.002000331879
2	azin	zain	0.000999927520
3	mazee	zaeem	0.001999855042
4	dira	rida	0.001000165939
5	ahelaws	swaleha	0.006000518799
6	lahdaub	abdulah	0.005000114441
7	oroatim	taimoor	0.00700044632
8	slaiym aezhuam	ayesha muzamil	0.09200525284
9	iaammu	umaima	0.00400018692
10	hesareah	areeshah	0.0100004673
11	uda	dua	0.001000165939



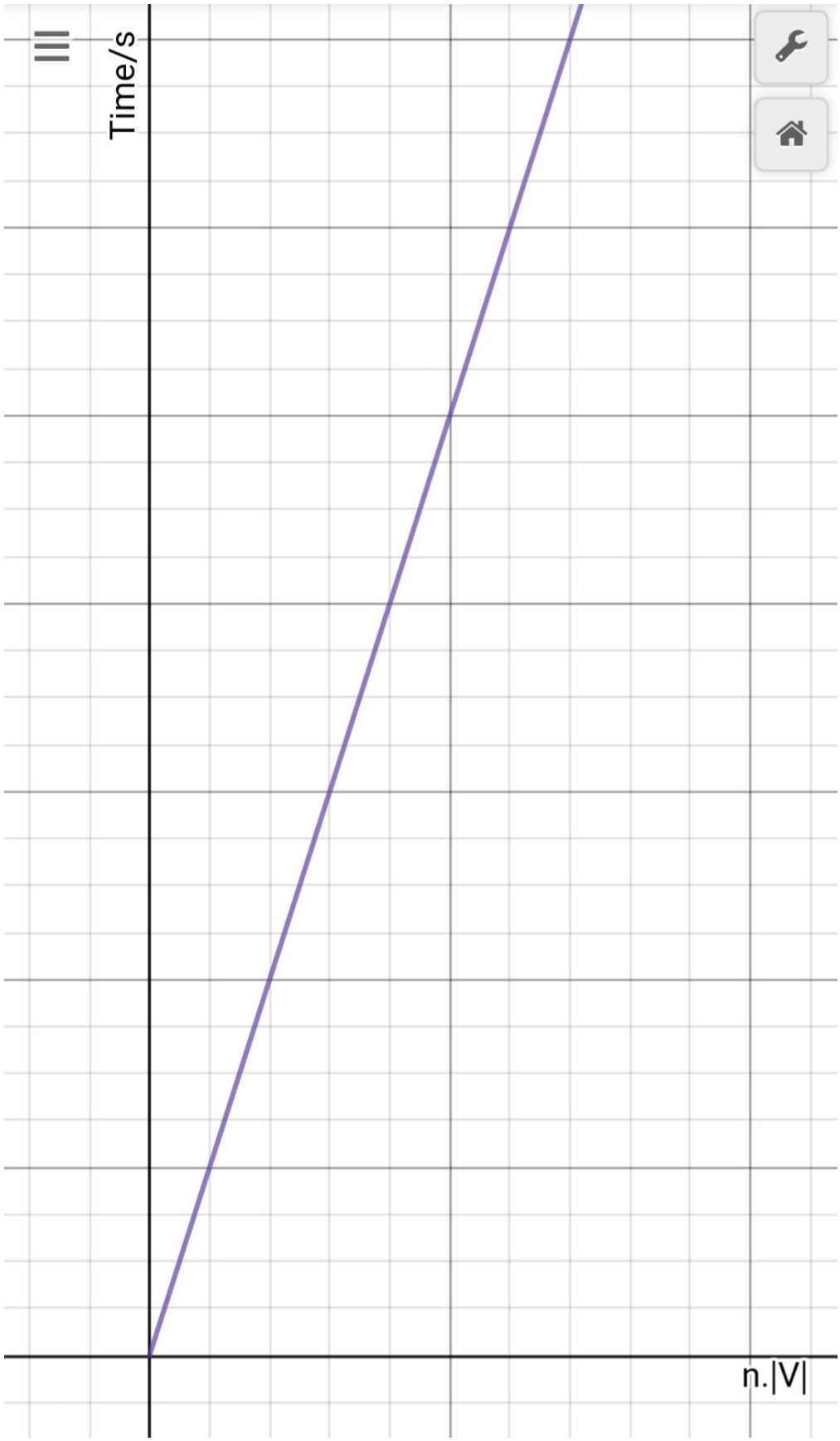
```
while(not self.path and self.priorityQueue.qsize()):
            closestChild = self.priorityQueue.get()[2]
            closestChild.CreateChildren()
            self.visitedQueue.append(closestChild.value)

            for child in closestChild.children:
                if child.value not in self.visitedQueue:
                    count +=1
```

- This implementation has worst case time complexity of $O(n.|V|)$ where "n" is the length of goal string, and |V| is the no. of vertices or nodes.

In String Combination application of A*, we have used a unique data structure which is the Priority Queue. It is a special form of Queue which stores elements in the ascending order of their values. Therefore, when we need to extract the node with the lowest value, we just need to *get* it from the Priority Queue, whose time complexity is $O(1)$. Therefore, it has lower time complexity as compared to the other two applications.

From the above snippet of the code, we can see that Priority Queue would run over |V| times in the worst case. The "for" loop inside the while loop runs for n times where n is the length of the goal string. Hence, the total time complexity of the code is $O(n.|V|)$.

For the sake of simplicity, if we consider "n" as a constant value, the time complexity becomes "linear" in this case. The graph of its complexity can be shown as below:


Time/s
n.|V|

Comparison:

In all the three application of A* algorithm, we came across different time complexities. Maze solving has $O(|V|^2)$ time complexity, 8-puzzle has $O(|V|^2.\log_2|V|)$ and string combination has $O(n.|V|)$ time complexity. From these results, we can compare that 8-puzzle has the highest time complexity, and string combination has the lowest.

The reason for such result is that we are using matrices in the 8-puzzle code which has certain time complexity of its own, and merged with A* time complexity, it becomes $O(|V|^2.\log_2|V|)$. In the case of string combination, we are using Priority Queue which reduces our time complexity and returns the lowest value node in O(1) time. Therefore, its time complexity is the lowest in all the three applications.

One final conclusion can be made that the A* time complexity depends upon the heuristics and the data structure that we use. Better the approximation of heuristics, better the time complexity. More efficient and cleverly we use the data structure in the application, better the time complexity of the algorithm.