1 Complexity Analysis

1.1

```
my_func(some_nums)
    result = 0
    for (num in some_nums)
        result += num
    return result
```

All this function is doing taking in some container of numbers and finding the sum of all elements.

$f(n) = O(n)$ as $n \to \infty$ where n is the size of *some_nums*. We are only traversing a list with no exit condition other than getting to the end of said list.

$f(n) = \Omega(1)$ as $n \to \infty$. If we just receive and empty or size one list.

There exist no $\theta(g(n))$ for $f(n)$ as $O(h(n)) == \Omega(h(n))$

1.2

```
int temp = 0
for (i =0; i < 100000000; i++) {
    for (j = 0; j < N; j++) {
    # N is a value defined based on user input
        temp = 0
        while(temp < j) {
            temp++;
        }
    }
}
```

This function gets the number $N - 1$, where N is defined by the user, 100,000,000 times.

$f(n) = O(n^2)$ as $n \to \infty$. Our outer most loop is contingent on the inner for loop finishing all it's iterations which is contingent on ITS inner while loop to finish all its iterations all

just to step once making this a triple nested loop. However, the outer most loop is a constant value, so we do not consider this and it does not contribute to our time complexity.

$f(n) = \Omega(1)$ as $n \to \infty$. If the user inputs 0, we still have to iterate 100,000,000 but we're just not doing anything during all those cycles, however it is ALWAYS 100,000,000, so our time complexity is constant time.

There exist no $\theta(g(n))$ for $f(n)$ as $O(h(n)) == \Omega(h(n))$.

1.3

```
input: array A
for(i = A.length-1; i > 0; i--) {
    swapped = false
    for(j=0; j < i-1; j++) {
        If(A[j] > A[j+1]) {
            swapped = true
            temp = A[j+1]
            A[j+1] = A[j]
            A[j] = temp
        }
    }
    if(!swapped) {
        break
    }
}
```

$f(n) = O(n^2)$ as $n \to \infty$. As simple as we have a double nested loop that we must iterate through.

$f(n) = \Omega(n)$ as $n \to \infty$. In the case that the array we receive is already sorted, our break inside the second loop acts as an exit condition to that inner loop. If a list is already sorted, the for loop is always broken out of before any actual iteration, there for making the complexity of that particular loop constant.

This is a basic sorting algorithm where we check if the value at the current index is greater than the value at the next. If so, we swap them. This is done until we have a sorted list.

2 Algorithm Description

1. **Left/Right-Hand Rule**. A classic maze solving algorithm in which all we do as we traverse in a direction is make sure we stay to one side of the wall and follow that wall regardless of junctions. What is unique about this pathfinder compared the many popular ones of the modern computing era is that we are reliant on knowing the maze to solve it.

2. **Dijkstra's Algorithm** A very modern, powerful pathfinding algorithm that we've all likely used in our life. The gist of it is that between different nodes on maze, there are different costs to traverse between those particular nodes. To build out path, we evaluate the weights between particular nodes and build partial routes. For example, say we have partial two routes, (NodeA, NodeB) costing 9, and (NodeA, NodeC, NodeB) costing 4. Perhaps route one is more "direct", but is bares a heavier cost that route two.Therefore, we conclude that the most optimised route from NodeA to NodeB is the latter of the routes, regardless if there was an extra node we have to go through.

3. **A***. There is one issue with Dijkstra's in that, although we may have the "cheapest" effect route calculated, we have no concept of our actual endpoint aside for when it's visible in our graph. A* adds such heuristic in the cost to traverse each node in that now, part of the cost of each node is distance to our endpoint.

4. **Merge Sort** A sorting algorithm that uses this idea of "divide and conquer". We split the list in half for some n amount of sublists, and we sort the sublist and combine sublists. Do this over and over again until we have a single list again.

5. **Bogo Sort** A very terrible sorting algorithm where we just randomly shuffle the array elements until it is eventually sorted.