Ques 1:

Prior Analysis:

The Longest Increasing Subsequence (LIS) problem involves finding the longest subsequence within an array where each element is greater than the previous one. This seemingly simple problem has a rich history and various approaches, necessitating prior analysis before choosing the most suitable solution.

Existing Solutions:

Several algorithms address the LIS problem, each with its own time and space complexity trade-offs:

1. **Naive Recursive Approach:**
   - Explores all possible subsequence's recursively, leading to exponential time complexity (O(2^n)).
   - Impractical for large arrays due to its slow performance.
2. **Dynamic Programming (DP) Memoization:**
   - Stores computed solutions for subproblems using memoization to avoid redundant calculations.
   - Reduces time complexity to O(n^2) while requiring additional space for the memoization table.
3. **Binary Search:**
   - Utilizes binary search to efficiently find the insertion point for each element in a DP table, potentially maintaining a sorted structure.
   - Offers further time optimization to O(n log n) at the cost of increased space complexity for the DP table and binary search operations.

Here in given question we need to find longest increasing subsequence from each index, so for this could be done efficiently by traversing back in array and finding LIS of index ahead of it.

Here in this solution going with dp solution of O(n^2).

Algorithm:

We will make a dp array which will tell us what is LIS at particular index to end of array

So dp[i] = LIS from ith node to end of array.

Step 1: Initialization

1. Create an array dp of size n, where n is the length of the input array arr.
2. Initialize all elements of dp to 1. This implies that the LIS for a single element is itself.

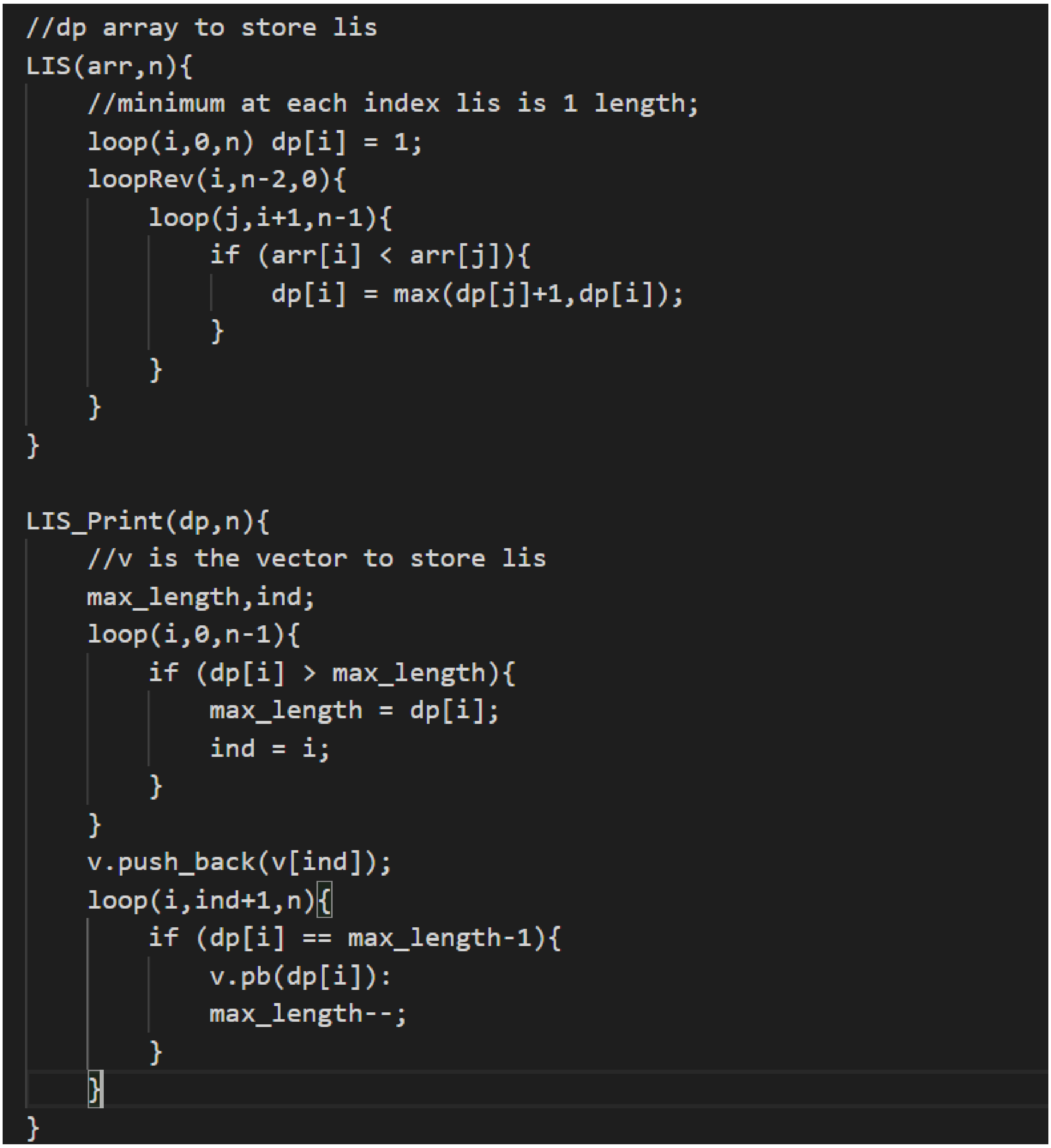
Step 2: DP Iteration

1. Iterate through the input array arr from index n-1 to 1:
   - For each element arr[i]:
     - Iterate through the dp array from index i+1 to n:
       - If arr[j] < arr[i]:
         - Check if dp[j] < dp[i] + 1. This compares the current LIS ending at i with the LIS ending at j extended by 1 using element arr[i].
         - If the inequality holds, update dp[j] to dp[i] + 1, indicating that the LIS starting at index j becomes longer.

Step 3: Finding the Actual LIS (Optional)

1. To find the actual elements of the LIS, you can start from the last element in the dp array (dp[n-1]) and backtrack using the following steps:
   - Create an empty stack or list to store the LIS elements.
   - Push the element arr[n-1] onto the stack.
   - Iterate backward from n-2 to 0:
     - If dp[i] == dp[i+1] + 1 and arr[i] < arr[i+1]:
       - Push arr[i] onto the stack.
2. The stack, when reversed, will contain the elements of the LIS.

Pseudo-Code:



```
//dp array to store lis
LIS(arr,n){
    //minimum at each index lis is 1 length;
    loop(i,0,n) dp[i] = 1;
    loopRev(i,n-2,0){
        loop(j,i+1,n-1){
            if (arr[i] < arr[j]){
                dp[i] = max(dp[j]+1,dp[i]);
            }
        }
    }
}

LIS_Print(dp,n){
    //v is the vector to store lis
    max_length,ind;
    loop(i,0,n-1){
        if (dp[i] > max_length){
            max_length = dp[i];
            ind = i;
        }
    }
    v.push_back(v[ind]);
    loop(i,ind+1,n){
        if (dp[i] == max_length-1){
            v.pb(dp[i]):
            max_length--;
        }
    }
}
```

Time Complexity:

The DP solution for LIS involves two nested loops:

1. Outer loop: Iterates through each element in the array arr from index n-1 to 0. This loop executes n times.
2. Inner loop: For each element, compares it with all previous elements (arr[0]...arr[i-1]). In the worst case, this loop runs n-i times for the last element.

However, the inner loop's average behavior isn't simply n/2 comparisons per element. We can analyze it more closely:

- In the first iteration, there's only one element to compare, so the inner loop runs once.
- In the second iteration, there are two elements to compare, so the inner loop runs twice.
- Continuing this pattern, for the ith iteration, the inner loop runs i times on average.

Summation Formula:

To calculate the total number of comparisons across all iterations, we can use the summation formula:

Σ(i = 1 to n) i = n(n + 1) / 2

This formula represents the sum of an arithmetic series, where the first term is 1 (one comparison for the first element), the last term is n (comparisons for the last element), and there are n terms in total.

Therefore, the total number of comparisons in the nested loops is:

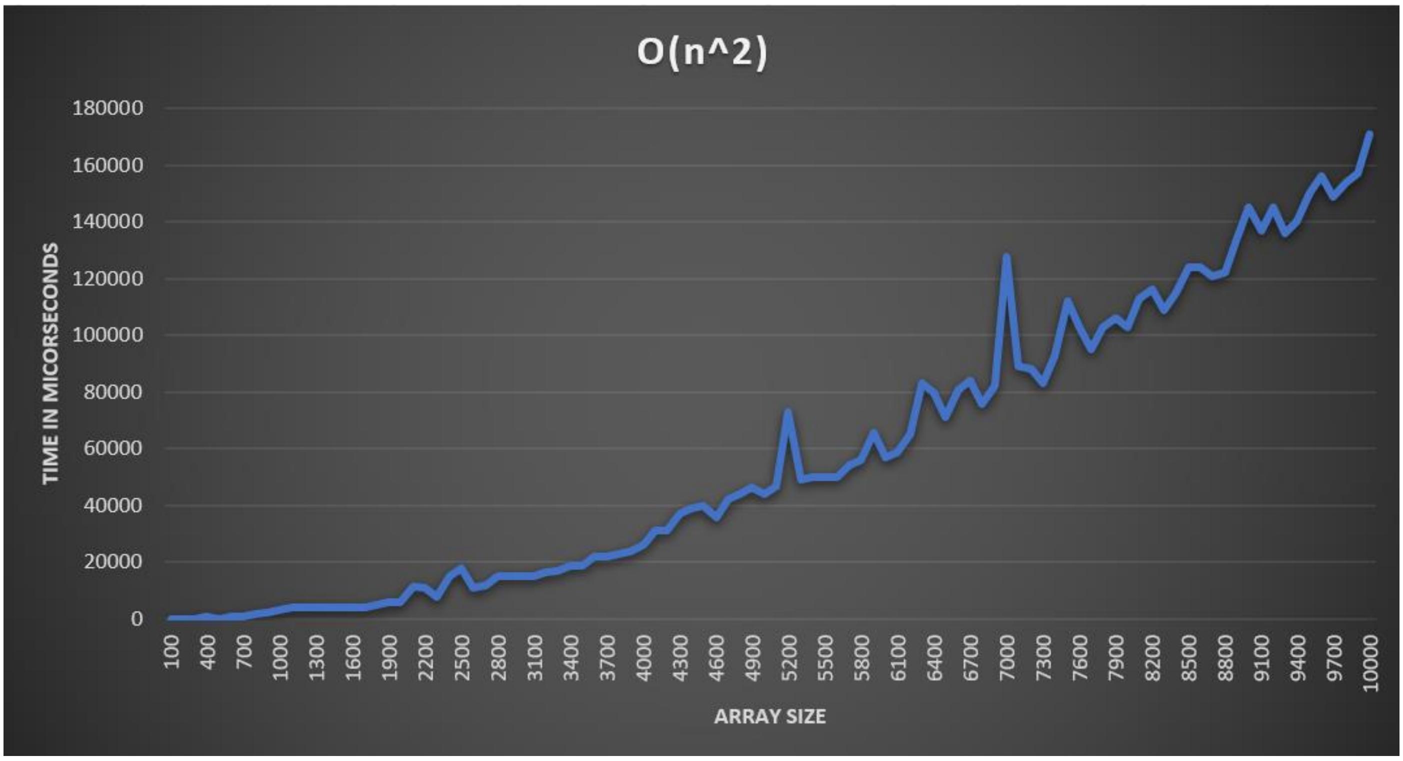
n + 2n + 3n + ... + (n-1)n + n^2 = n(n + 1) / 2

Simplifying this expression, we get:

n(n + 1) / 2 = n^2 / 2 + n/2 = O(n^2)

Conclusion:

The time complexity of the DP solution for LIS is O(n^2) due to the nested loops and the summation of comparisons within the inner loop. While the inner loop's average behavior is lower than its worst-case n comparisons, it still results in a quadratic dependency on the input size n.



Post Analysis:

- Optimizations like binary search within the inner loop can reduce the time complexity to O(n log n) but require additional space and complexity.
- For specific scenarios like positive elements, modified Kadane's Algorithm might offer space-efficient solutions with linear time complexity.
- The choice of algorithm depends on your specific needs, input size, and performance constraints.

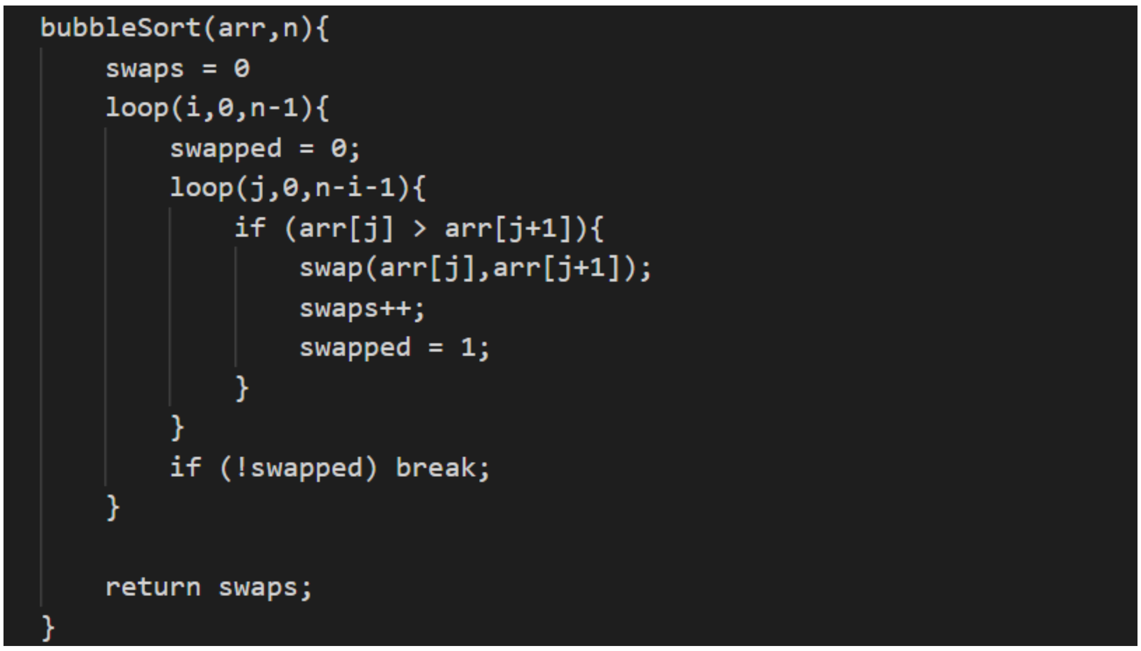
Ques 2:

Prior Analysis:

To count number of swaps in given sorting algorithm whenever we swap elements we will increment a variable to keep track of number of swaps in each of sorting algorithm.

Algorithm & Pseudo-Code:

- **Bubble Sort:**



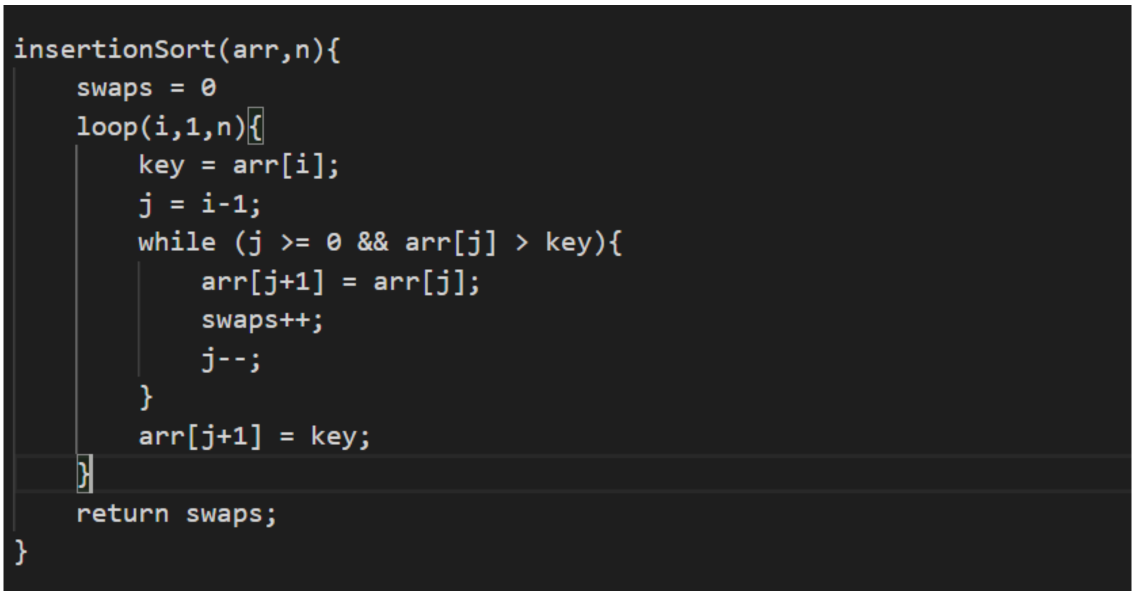
```
bubbleSort(arr,n){
    swaps = 0
    loop(i,0,n-1){
        swapped = 0;
        loop(j,0,n-i-1){
            if (arr[j] > arr[j+1]){
                swap(arr[j],arr[j+1]);
                swaps++;
                swapped = 1;
            }
        }
        if (!swapped) break;
    }

    return swaps;
}
```

Time complexity:

- Loop to run iterations: O(n-1)
- Loop to compare elements: O(n) inside, nested within the first loop: O(n*(n-1))
- Swapping happens within the inner loop, contributing to O(n*(n-1))
- Optimization reduces iterations slightly in best case

Total: O(n^2) (worst and average case),O(n) (best case)

- **Insertion Sort:**



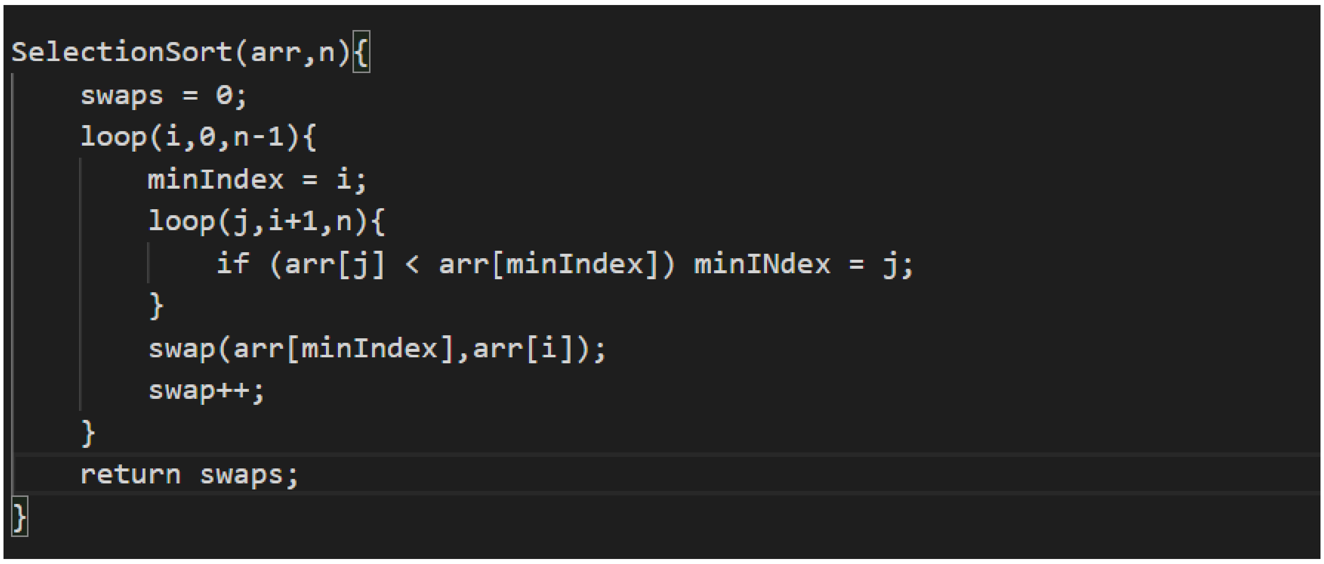
```
insertionSort(arr,n){
    swaps = 0
    loop(i,1,n){
        key = arr[i];
        j = i-1;
        while (j >= 0 && arr[j] > key){
            arr[j+1] = arr[j];
            swaps++;
            j--;
        }
        arr[j+1] = key;
    }
    return swaps;
}
```

Time complexity:

- Loop to run iterations: O(n)
- Inner loop (while) depends on element position: O(i) in worst case (nearly sorted array), O(1) in best case (already sorted)
- Swapping happens in the inner loop, contributing to O(n * max(i))
- Worst case inner loop dominates: O(n^2)
- Best case inner loop reduces overall complexity to O(n)

Total: O(n^2) (worst case and average case), O(n) (best case)

Selection Sort:



```
SelectionSort(arr,n){
    swaps = 0;
    loop(i,0,n-1){
        minIndex = i;
        loop(j,i+1,n){
            if (arr[j] < arr[minIndex]) minINdex = j;
        }
        swap(arr[minIndex],arr[i]);
        swap++;
    }
    return swaps;
}
```

Time complexity:

- Loop to run iterations: O(n-1)
- Loop to find minimum element: O(n) inside, nested within the first loop: O(n*(n-1))
- Swapping happens once per iteration: O(n-1)

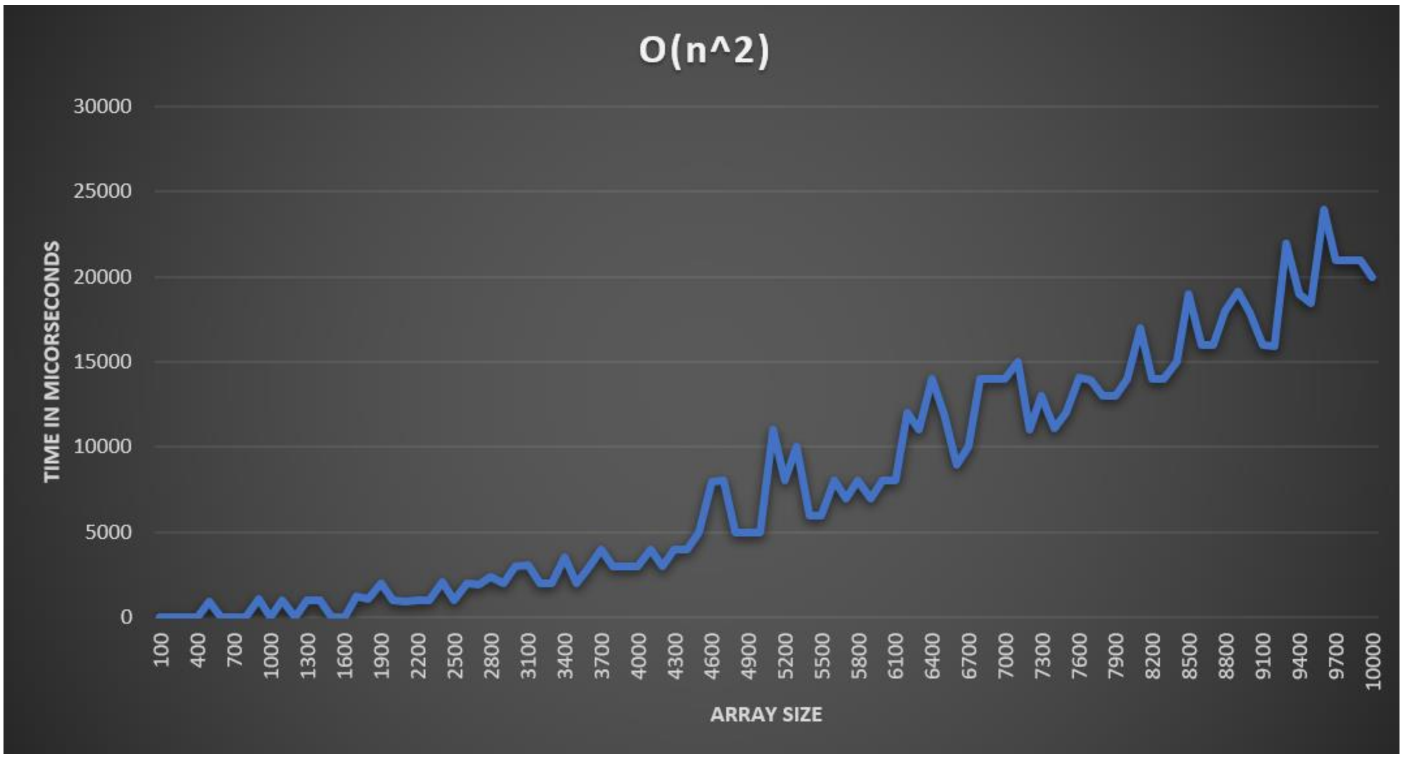
Total: O(n^2) (worst and average case)

Post Analysis:

In Context of number of swap operation, in worst cases and average cases

Selection Sort > Insertion Sort == Bubble Sort

(n-1) > n*(n-1) > n*(n-1)



But in Time Complexity all of these Sorting algorithm are not much efficient because there worst time complexity goes to O(n^2) far worse than many other sorting algorithm like merge sort, heap sort that can work in O(n logn).

Conclusion:

- Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets that are already partially sorted
- Bubble Sort is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.
- Selection Sort does not preserve the relative order of items with equal keys which means it is not stable.

Ques 3:

Prior Analysis:

For Finding collinear points which are parallel to X-Axis we need to consider all those points with same y-coordinate

And for points which are parallel to Y-Axis we need to consider all those points which have same x -coordinate

Algorithm:

1. X-axis:

a) Store all points in a vector<pair<int, int>> where the first element represents the X-coordinate and the second represents the Y-coordinate.

b) Iterate through the vector.

c) For each point, compare its Y-coordinate to the Y-coordinate of the next point.

d) If the Y-coordinates are equal, consider both points collinear with the X-axis. Store them in a separate list or data structure for collinear points.

e) Continue iterating and comparing until reaching the end of the vector.

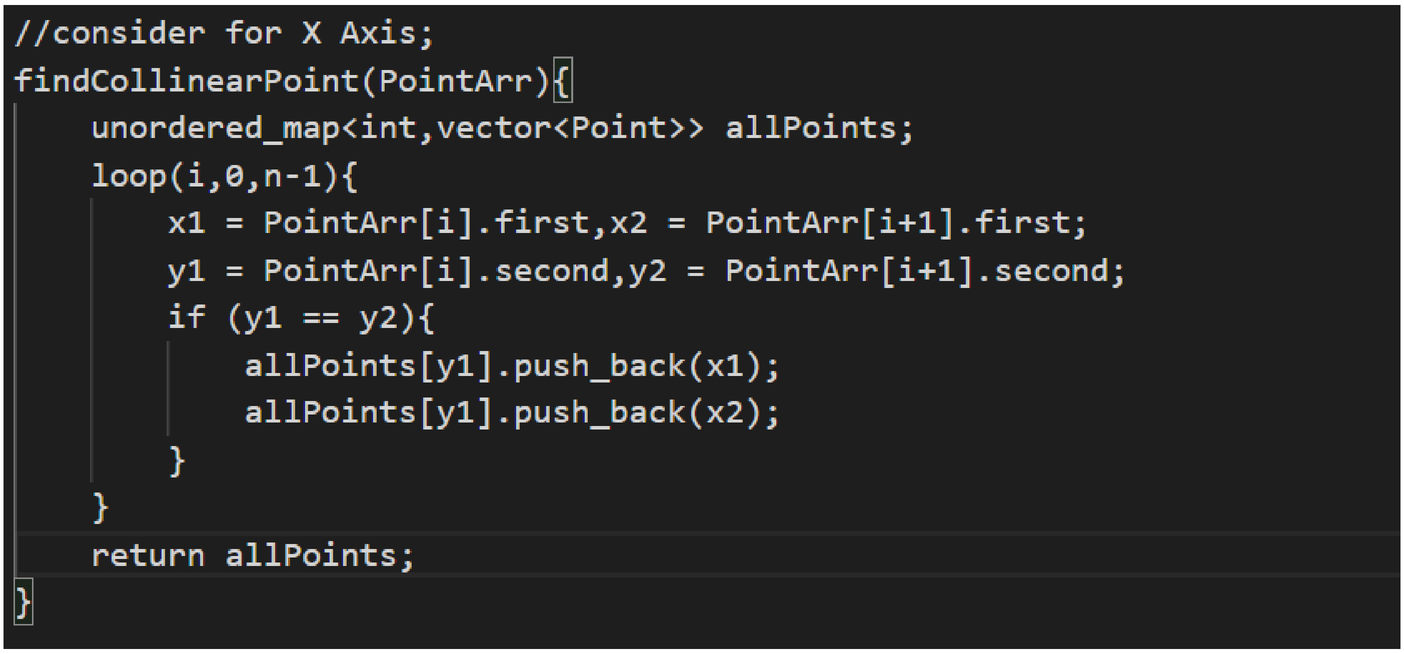
2. Y-axis:

a) Perform steps (a) and (b) from the X-axis approach.

b) Instead of comparing Y-coordinates, compare X-coordinates for each point and its next.

c) If the X-coordinates are equal, consider both points collinear with the Y-axis. Store them in a separate list or data structure for collinear points.

Pseudo-Code:



```
//consider for X Axis;
findCollinearPoint(PointArr){
    unordered_map<int,vector<Point>> allPoints;
    loop(i,0,n-1){
        x1 = PointArr[i].first,x2 = PointArr[i+1].first;
        y1 = PointArr[i].second,y2 = PointArr[i+1].second;
        if (y1 == y2){
            allPoints[y1].push_back(x1);
            allPoints[y1].push_back(x2);
        }
    }
    return allPoints;
}
```

Time Complexity:

Both X and Y axes:

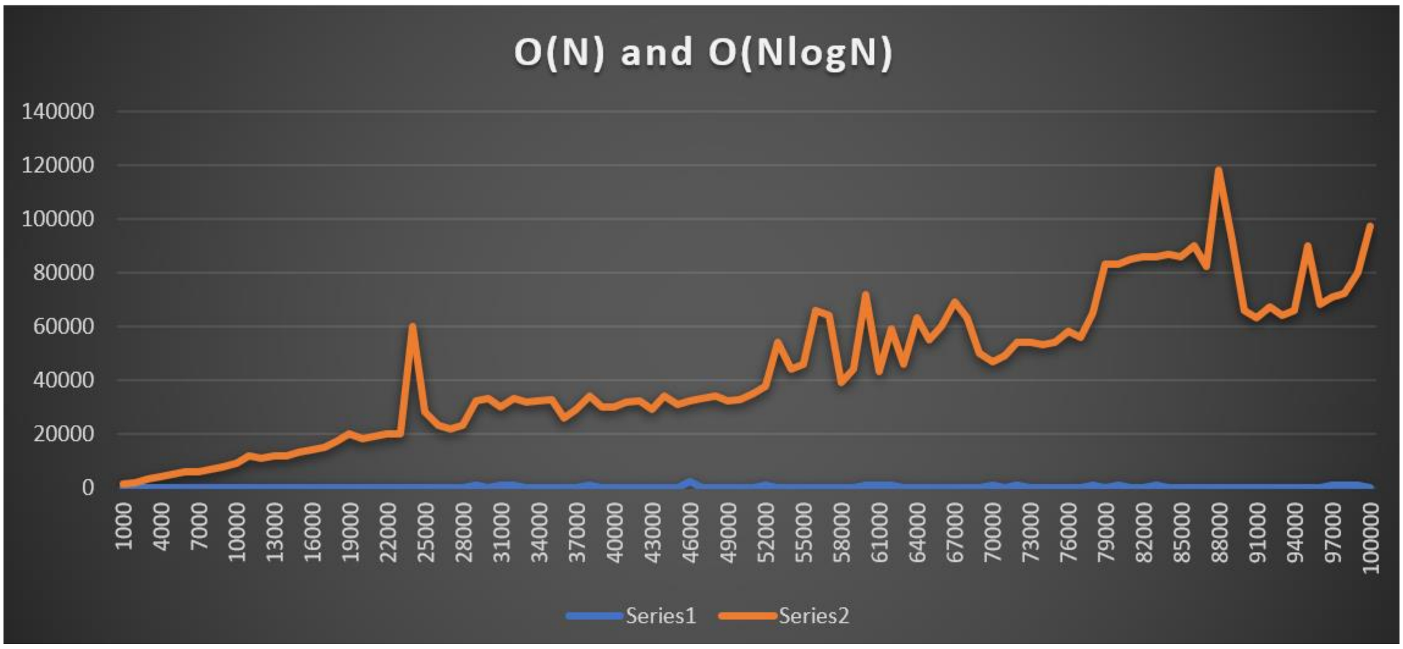
- Iteration through the vector: O(n), where n is the number of points (5000 in this case).
- Comparison inside the loop: O(1), constant operation.
- If matched push in O(2),constant operation
- Total: O(n)

Additional considerations:

- Storing collinear points can be done in an unordered map for efficient membership checks with constant amortized time complexity.
- For large datasets, consider optimizations like spatial hashing to potentially avoid unnecessary comparisons if points are far apart.

Post Analysis:

- There could be duplicate entries in the vector. Two ways to handle that
  - (a) Remove all duplicate entries from the vector TC : O(n)
  - (b) Can use set instead of vector to store element TC : O(nlogn)



Ques 4:

Prior Analysis:

To find ith Fibonacci number we simply need (i-1)th and (i-2)th Fibonacci number.

Now for a given number we start generating Fibonacci from start: 1,1,2,3,5,8 …..

When we reach close to a number greater than or equal to given number we would stop..

and check among all traversed number so far which is the nearest one to the given N.

Proof that this is optimal because Fibonacci number increase rapidly this can be seen by

Looking fib[50] = 12586269025 approx(10^10) => approx.(2^35)

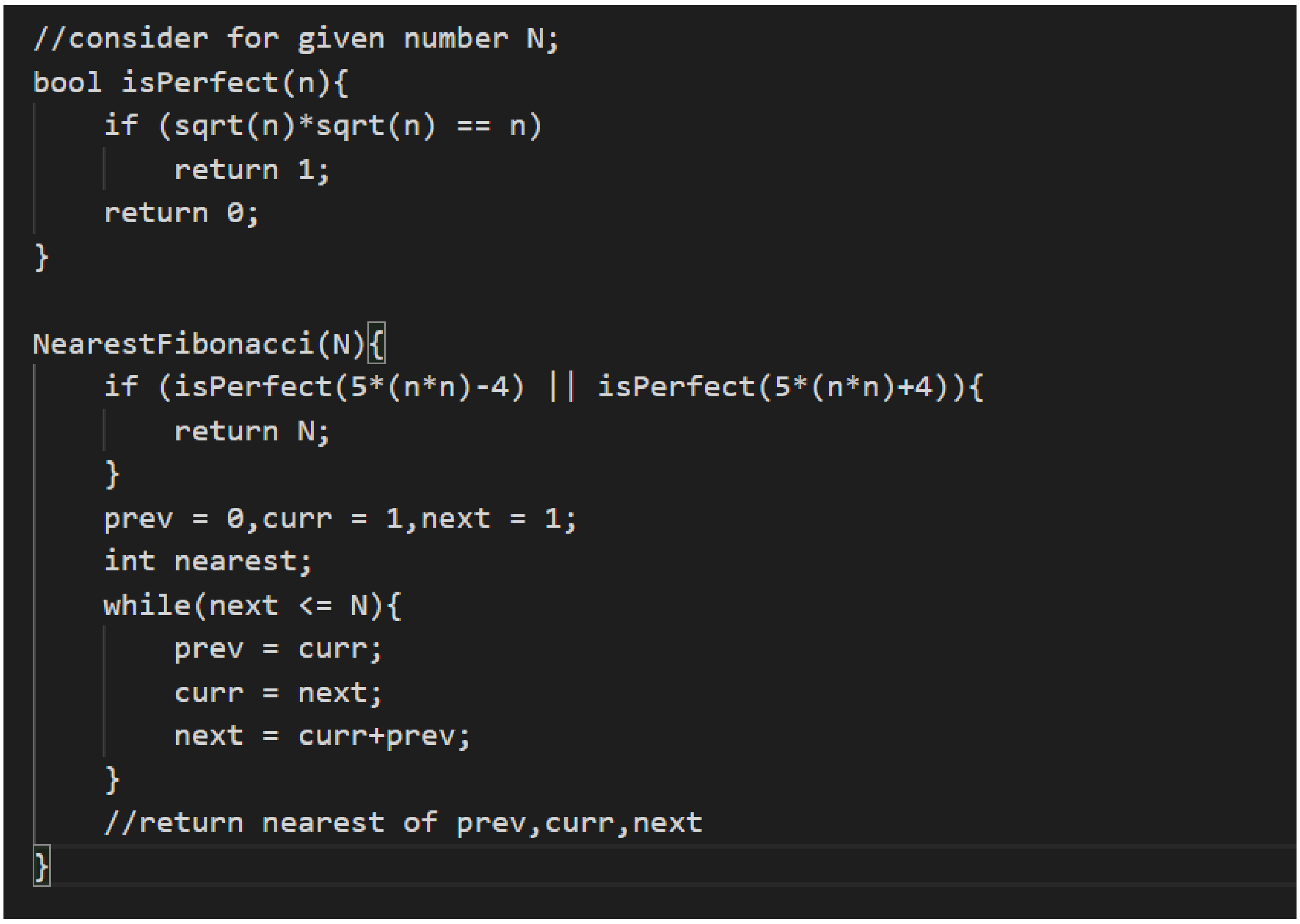
So in nearly log(N) we could find number is Fibonacci or not and its nearest Fibonacci.

This process can be further optimised by the property of Fibonacci number i.e., if number N is Fibonacci, then $5N^2 + 4$ or $5N^2 – 4$ is perfect square.

Algorithm:

a) Input number
b) Check $5N^2 + 4$ or $5N^2 – 4$ is perfect square.
c) If perfect square done
d) Start from 0th Fibonacci and go till it is greater than given number

Pseudo-Code:



```
//consider for given number N;
bool isPerfect(n){
    if (sqrt(n)*sqrt(n) == n)
        return 1;
    return 0;
}

NearestFibonacci(N){
    if (isPerfect(5*(n*n)-4) || isPerfect(5*(n*n)+4)){
        return N;
    }
    prev = 0,curr = 1,next = 1;
    int nearest;
    while(next <= N){
        prev = curr;
        curr = next;
        next = curr+prev;
    }
    //return nearest of prev,curr,next
}
```

Time Complexity Analysis:

- Each iteration generates a new Fibonacci number, and the loop continues until c exceeds n.
- In the worst case, n is very close to a Fibonacci number, requiring approximately log2(n) Fibonacci numbers to be generated before exceeding n.
- Therefore, the time complexity of this algorithm is O(log N).

Example:

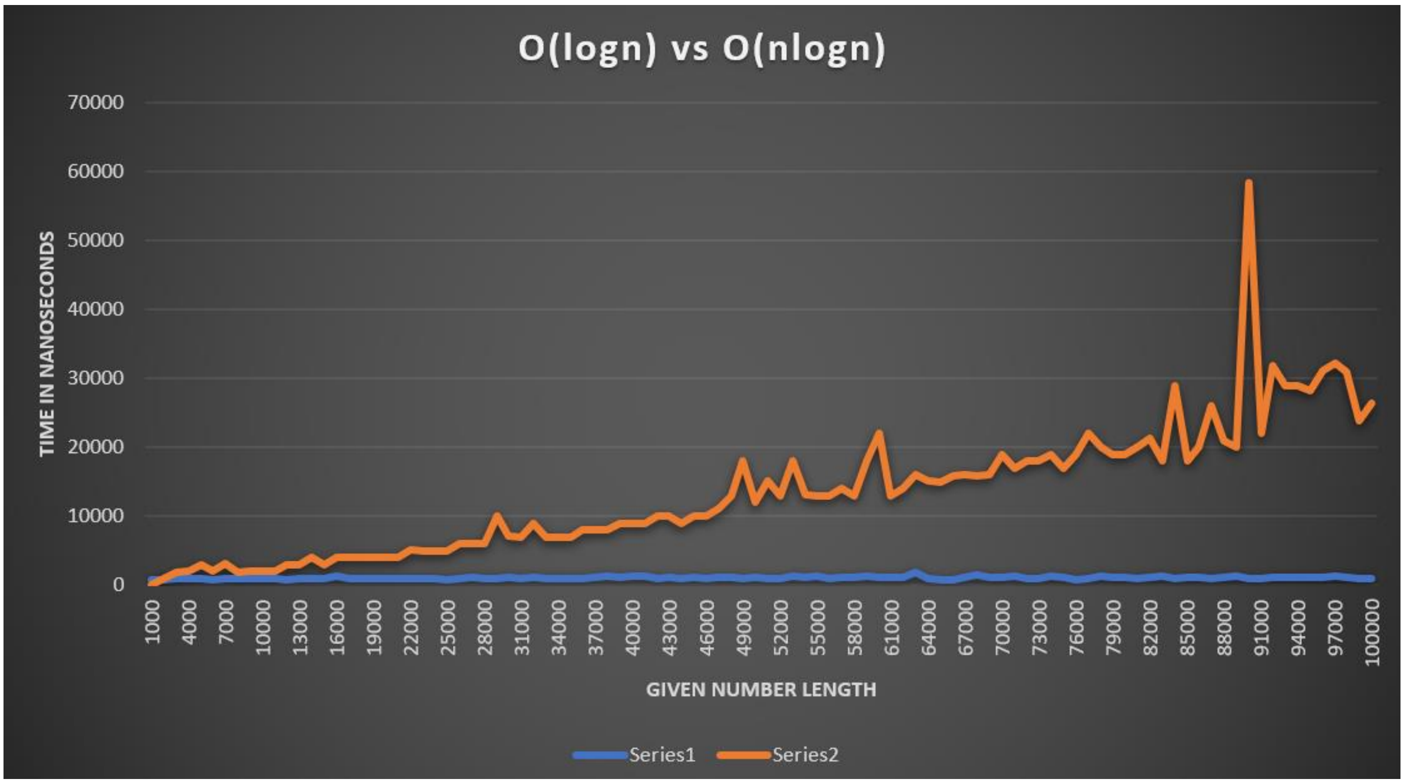
- Let's find the nearest Fibonacci number to n = 23.
- The Fibonacci numbers generated are: 0, 1, 1, 2, 3, 5, 8, 13, 21.
- At c = 21, the loop terminates as c becomes greater than n.
- a = 13 is the largest Fibonacci number smaller than or equal to n, and b = 21 is the smallest Fibonacci number greater than or equal to n.
- Comparing abs(n - a) = 10 and abs(n - b) = 2, we find that 21 is closer to n as it has the smaller absolute difference.

Post Analysis:

If number is given in long long or integer format then maximum allowed value is 1e18, in that case it could easily be checked in log(n)

But if number is given in a string format, then for calculating fib(n) we need fib(n-1) and fib(n-2) to be stored in string format and also have to do string addition to calculate new number. So O(n) is required for string addition.

So Overall time complexity jumps to O(nlogn) if input is given in string format.



Ques 5:

Prior Analysis:

We are given a 5-digit number so at max number out of three could be of three digit. So the partition can be done using two pointer placing them in all possible places to make 3 numbers and furthermore can be checked.

Checking given triplet is a Pythagorean or not can be done easily with its property of $a^2+b^2=c^2$

Algorithm:

Here's an algorithm to find Pythagorean triplets among 5-digit numbers with three partitions and analyse its time complexity:

1. Input and Validation:

- Take a 5-digit number as input and convert it to an integer.
- Validate the input to ensure it has five digits.

2. Generating Partitions:

- Use nested loops to iterate through all possible combinations of three partitions across the five digits.
  - The outer loop iterates from 1 to 3 for the first partition size.
  - The middle loop iterates from the previous first partition size + 1 to the remaining length - 1 for the second partition size.
  - The inner loop iterates from the previous second partition size + 1 to the remaining length for the third partition size.

3. Pythagorean Check:

- For each partition, construct three corresponding integers (a, b, c) from the partitioned digits.
- Check if $a^2 + b^2 = c^2$ using an efficient square root function (e.g., optimized sqrt or integer-based comparisons).

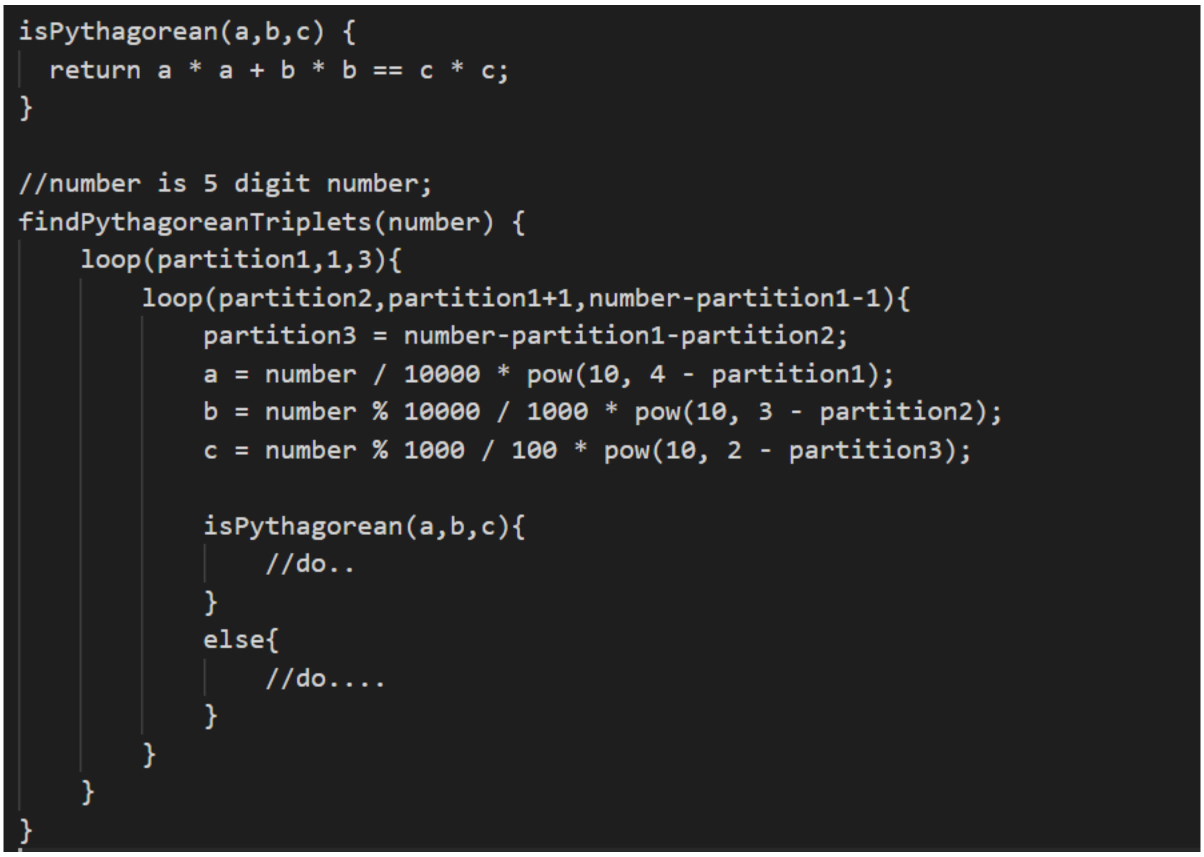
4. Nearly Pythagorean (Optional):

- If not a perfect triplet, calculate the absolute difference between $a^2 + b^2$ and c^2 .
- This difference represents how **far** the triplet is from being right-angled.

5. Output:

- Print all partitions that form perfect Pythagorean triplets.
- Optionally, report nearly Pythagorean triplets with their "nearness" value.

Pseudo-Code:



```
isPythagorean(a,b,c) {
  return a * a + b * b == c * c;
}

//number is 5 digit number;
findPythagoreanTriplets(number) {
    loop(partition1,1,3){
        loop(partition2,partition1+1,number-partition1-1){
            partition3 = number-partition1-partition2;
            a = number / 10000 * pow(10, 4 - partition1);
            b = number % 10000 / 1000 * pow(10, 3 - partition2);
            c = number % 1000 / 100 * pow(10, 2 - partition3);

            isPythagorean(a,b,c){
                //do..
            }
            else{
                //do....
            }
        }
    }
}
```

Time Complexity:

- Input/Validation: O(1), constant time operations.
- Partition Generation: O(n^2), where n is the number of digits (5), due to the nested loops.
- Pythagorean Check: O(1) per check, negligible compared to generation.
- Nearly Pythagorean (optional): O(1) per calculation, also negligible.
- Output: O(k), where k is the number of found triplets (either perfect or nearly), negligible compared to generation.

Overall Time Complexity:

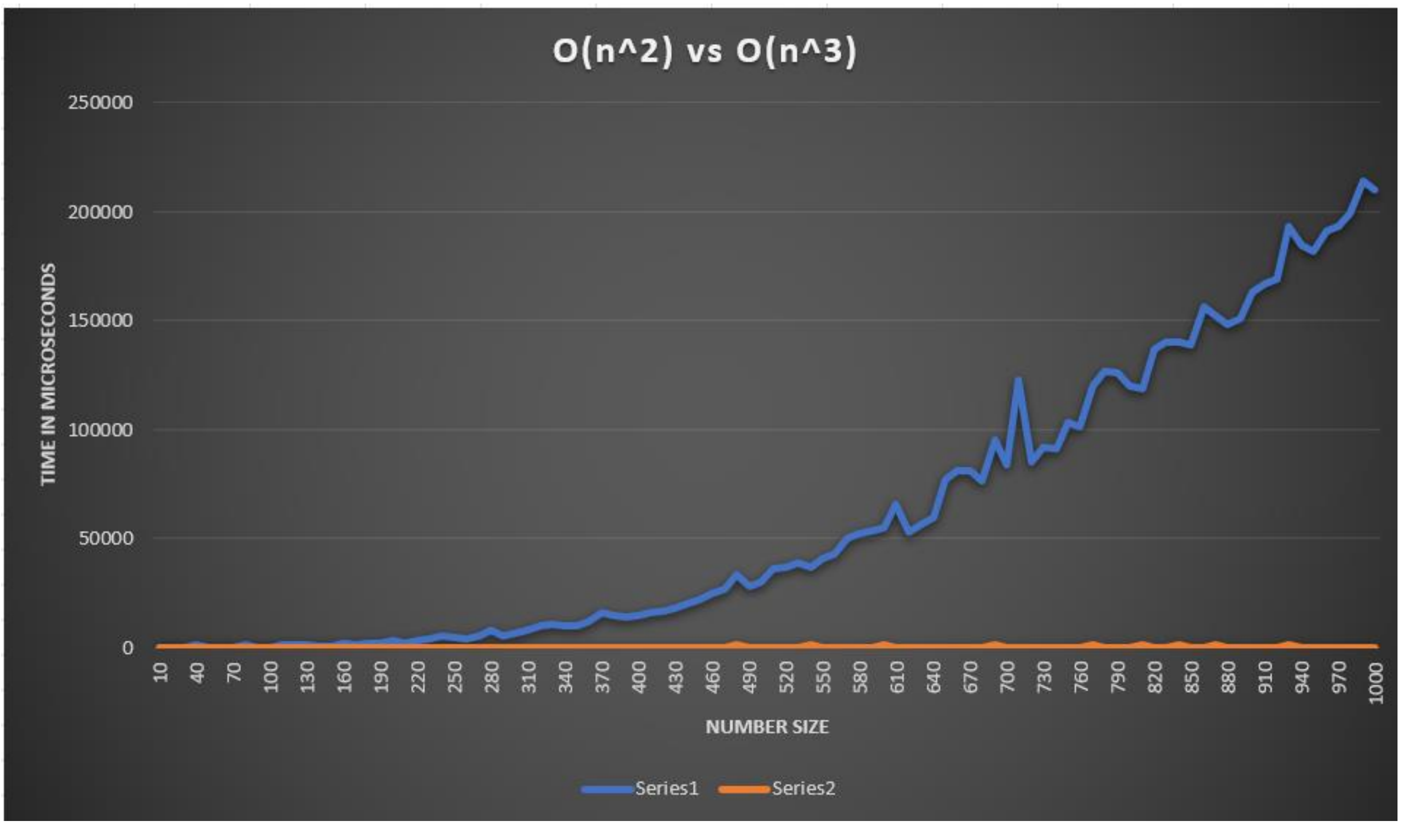
T(n) = O(n^2), dominated by the partition generation, making it cubic in the number of digits (n = 5 in this case).

Post Analysis:

If number is given in long long or integer format then maximum allowed value is 1e18. In that case n = 18 and n^2 algorithm can work easily

But if number is given in a string format, then n^2 algorithm can work only if number $n <= 10^4$ Also in that case Pythagorean triplet can't be checked as done here. In that case we have to do string multiplication to calculate a^2 , b^2 and c^2 and also string addition to check Pythagorean equality.

So, in that case Time Complexity would be $O(n^3)$ because in $O(n^2)$ we would partition array and O(n) each partition would be checked.


O(n^2) vs O(n^3)
TIME IN MICROSECONDS
NUMBER SIZE
Series1
Series2