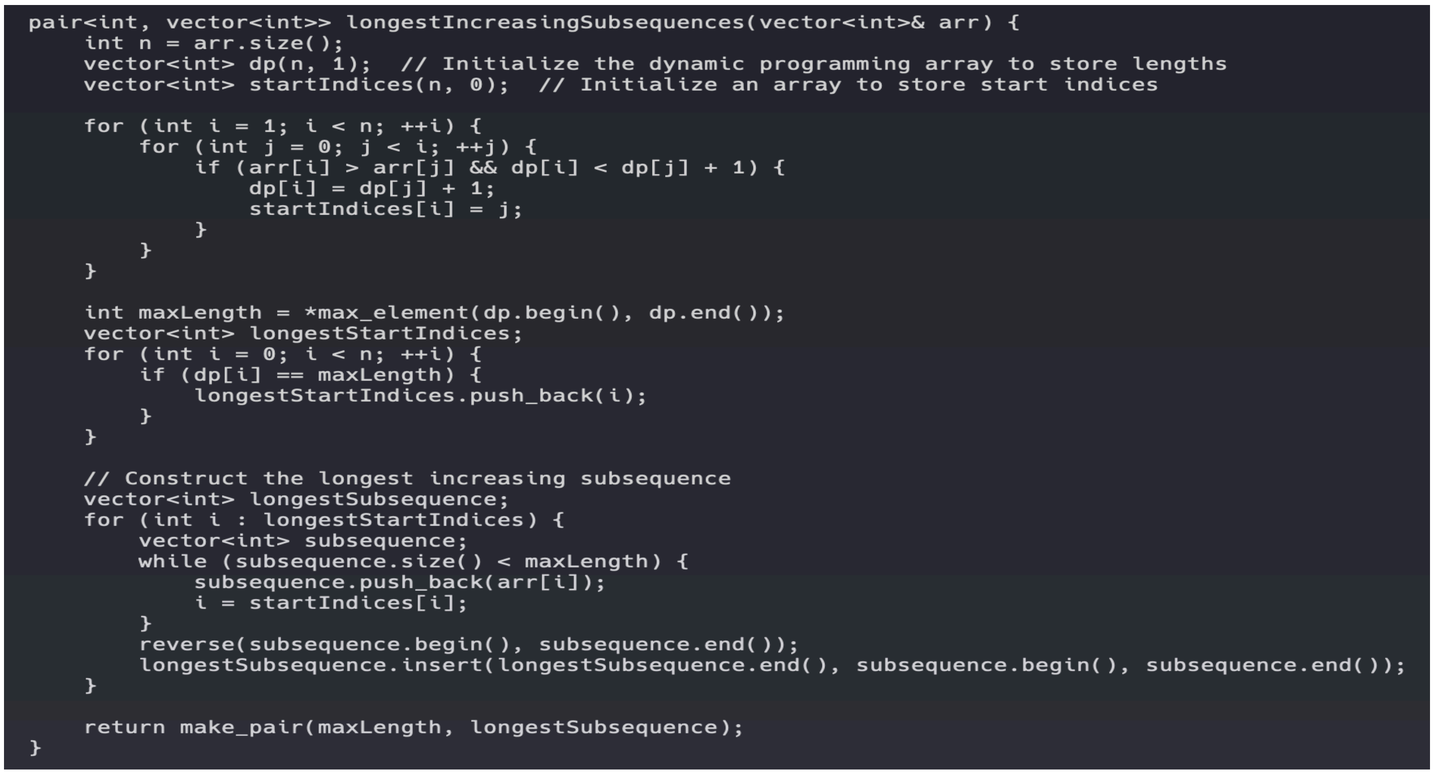
Mid Semester Lab Assignment

Name: Samuel E George
Roll no. : IIT2022007

Q1)



```cpp
pair<int, vector<int>> longestIncreasingSubsequences(vector<int>& arr) {
    int n = arr.size();
    vector<int> dp(n, 1);  // Initialize the dynamic programming array to store lengths
    vector<int> startIndices(n, 0);  // Initialize an array to store start indices

    for (int i = 1; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            if (arr[i] > arr[j] && dp[i] < dp[j] + 1) {
                dp[i] = dp[j] + 1;
                startIndices[i] = j;
            }
        }
    }

    int maxLength = *max_element(dp.begin(), dp.end());
    vector<int> longestStartIndices;
    for (int i = 0; i < n; ++i) {
        if (dp[i] == maxLength) {
            longestStartIndices.push_back(i);
        }
    }

    // Construct the longest increasing subsequence
    vector<int> longestSubsequence;
    for (int i : longestStartIndices) {
        vector<int> subsequence;
        while (subsequence.size() < maxLength) {
            subsequence.push_back(arr[i]);
            i = startIndices[i];
        }
        reverse(subsequence.begin(), subsequence.end());
        longestSubsequence.insert(longestSubsequence.end(), subsequence.begin(), subsequence.end());
    }

    return make_pair(maxLength, longestSubsequence);
}
```

Computation time analysis

Operation	**Number of Operations**	**Frequency**	**Time complexity**
Initializing ‘dp’ and startIndices	2	1	O(1)

Dynamic programming loop	O(n^2)	1	O(n^2)
Finding maximum length	O(n)	1	O(n)
Total time complexity			O(n^2+n+1)

Therefore the time complexity of this algorithm is O(N^2).

Best case time complexity: O(n)

Best case time complexity occurs when the array is already sorted in descending order.
For every iteration of the outer loop, the inner loop would stop after the first iteration.
Therefore the best case time complexity is O(n).

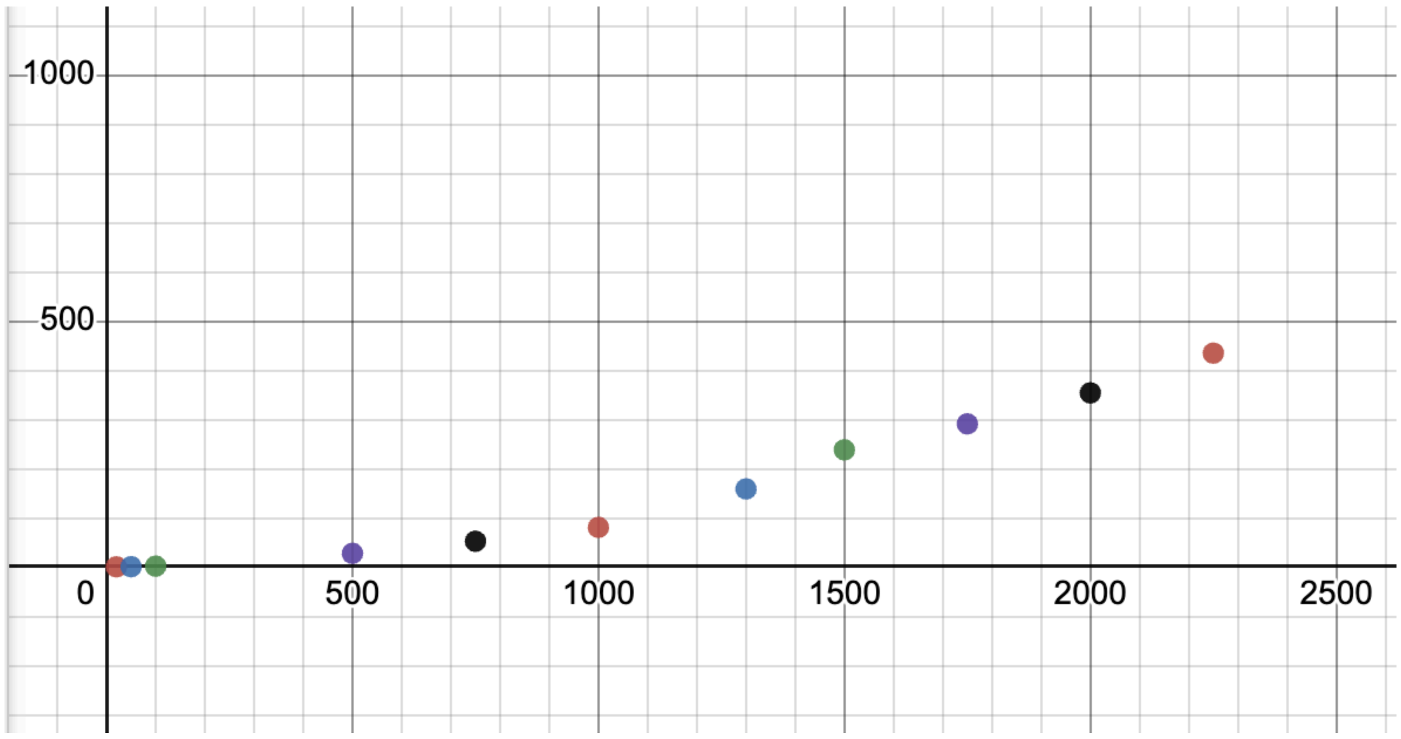
Average case time complexity:O(n^2)

Average case time complexity is when it is randomly ordered. So the average time complexity is closer to the worst case time complexity. So the average time complexity is O(n^2).

Worst case time complexity:O(n^2)

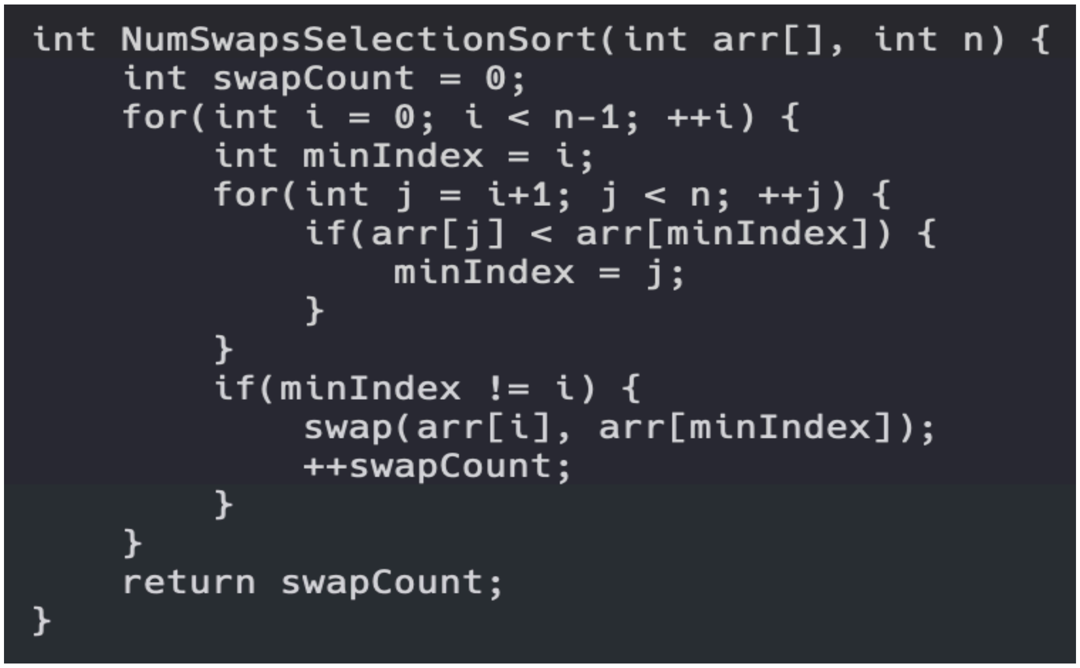
The worst case complexity occurs when the array is sorted in ascending order. For each iteration of the outer loop the inner loop iterates n-1 times.
Therefore the worst case time complexity is O(n^2).

Practical time complexity analysis:

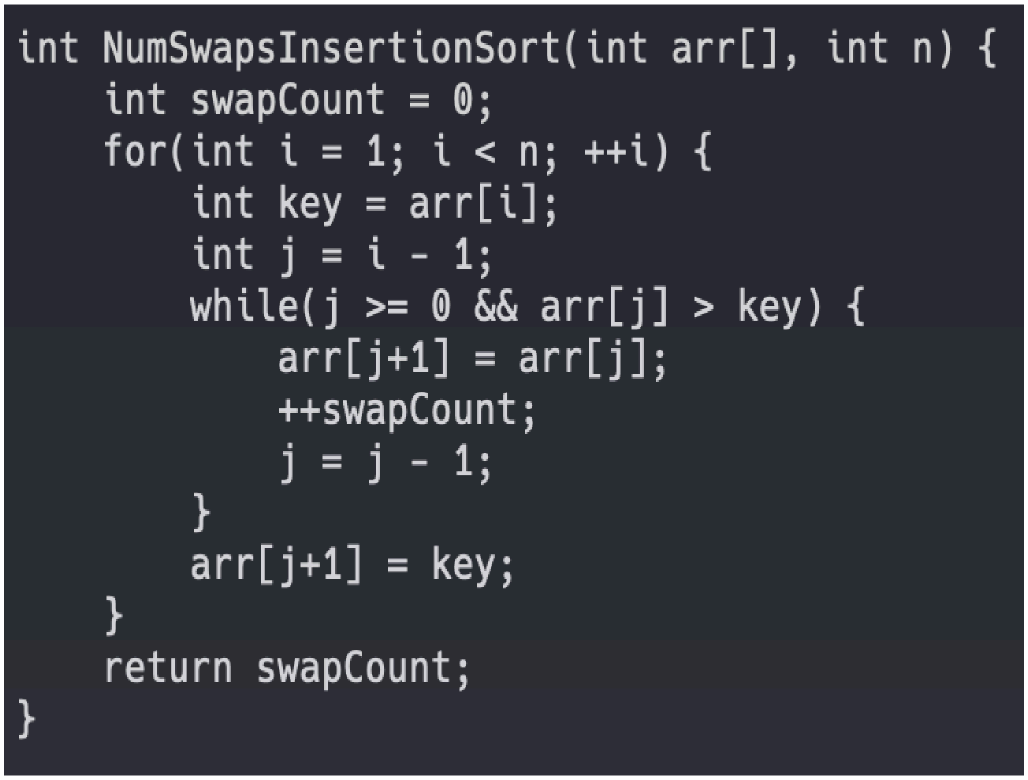


As you can see initially the computation times were pretty low, but as the input size increased, the computation times also increased at a rate which is very similar to the the graph of O(N^2).

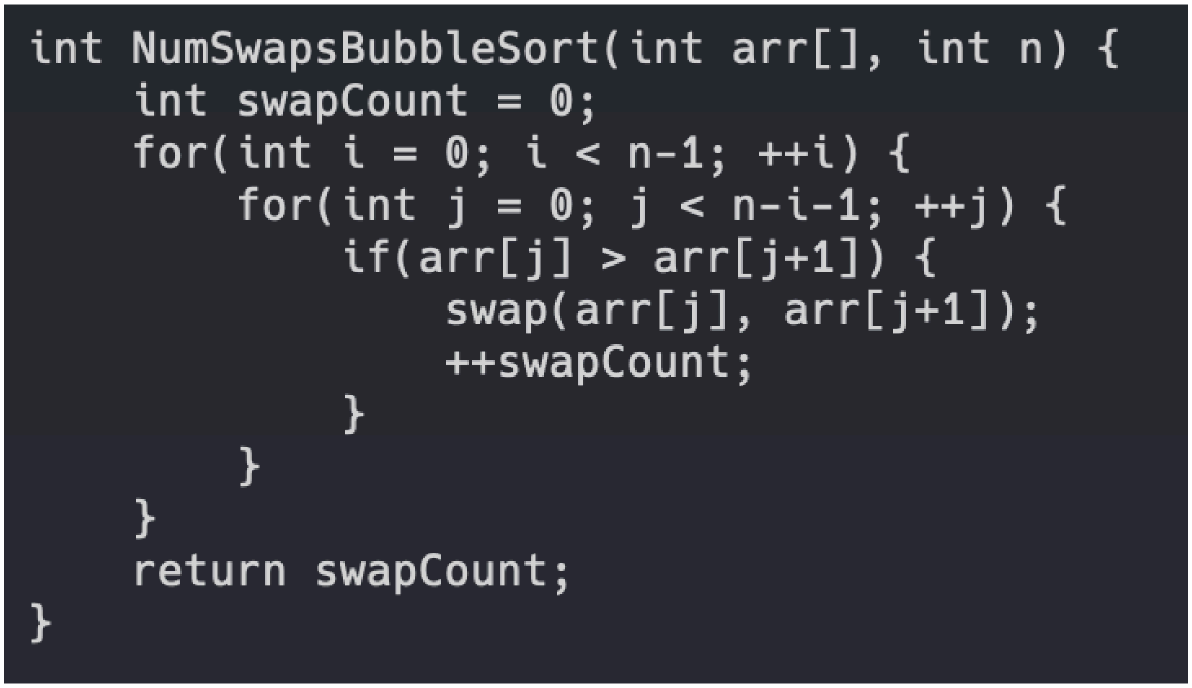
Q2)



```
int NumSwapsSelectionSort(int arr[], int n) {
    int swapCount = 0;
    for(int i = 0; i < n-1; ++i) {
        int minIndex = i;
        for(int j = i+1; j < n; ++j) {
            if(arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        if(minIndex != i) {
            swap(arr[i], arr[minIndex]);
            ++swapCount;
        }
    }
    return swapCount;
}
```



```
int NumSwapsInsertionSort(int arr[], int n) {
    int swapCount = 0;
    for(int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > key) {
            arr[j+1] = arr[j];
            ++swapCount;
            j = j - 1;
        }
        arr[j+1] = key;
    }
    return swapCount;
}
```



```
int NumSwapsBubbleSort(int arr[], int n) {
    int swapCount = 0;
    for(int i = 0; i < n-1; ++i) {
        for(int j = 0; j < n-i-1; ++j) {
            if(arr[j] > arr[j+1]) {
                swap(arr[j], arr[j+1]);
                ++swapCount;
            }
        }
    }
    return swapCount;
}
```

Computation time analysis:

Bubble Sort:

Operation	**Number of Operations**	**Frequency**	**Time complexity**
Outer loop	O(n)	1	O(n)
Inner loop	O(n)	1	O(n)
Total time complexity			O(n^2)

The outer loop will run n times and for each iteration it will compare the following adjacent elements. This will run n-1 times. Therefore the overall time complexity is O(n^2).

For bubble sort, best case, average case and worst case have O(n^2) because even if it is sorted it will iterate through the entire array and compare.

<u>**Insertion Sort:**</u>

Operation	**Number of Operations**	**Frequency**	**Time complexity**
Initializing variables	2	1	O(2)=O(1)
Outer loop	O(n)	1	
Inner loop	O(n)	1	O(n)
Total time complexity			O(n^2)

The outer loop iterates n-1 times and the inner loop performs at worst case O(n) operations in every iteration. Therefore the overall time complexity of this algorithm is O(n^2).

The best case occurs when the array is already sorted. As it iterates through the array , it doesn't have to go back and swap elements so the best case time complexity is O(n).

The average case complexity is when the array is sorted, hence the time complexity is O(n^2).

In the worst-case scenario, every element needs to be moved to the beginning of the array, resulting in n^2 comparisons and swaps. This leads to a quadratic time complexity. Hence time complexity is O(n^2).

Selection sort:

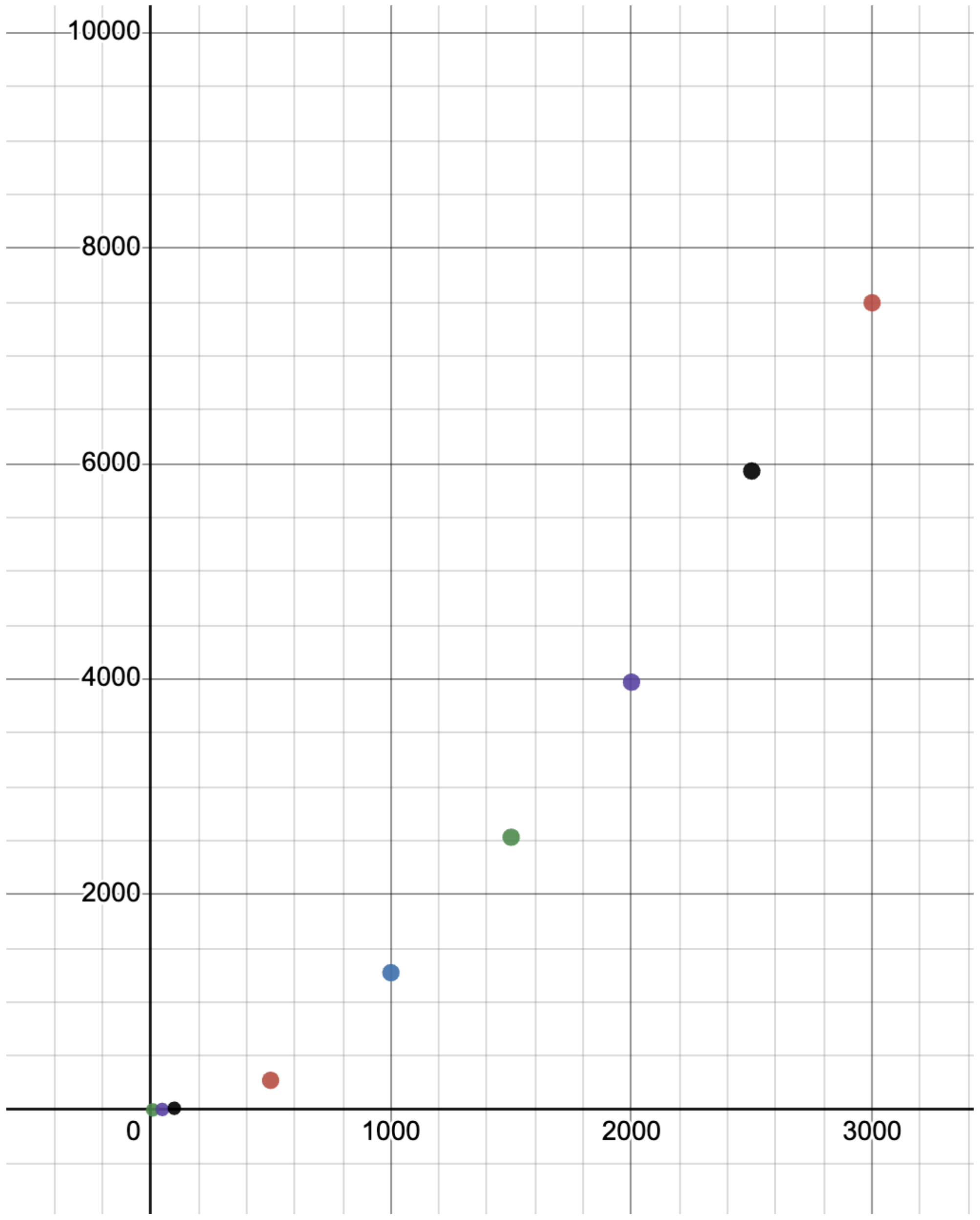
Operation	**Number of Operations**	**Frequency**	**Time complexity**
Initializing variables	2	1	O(2)=O(1)
Outer loop	O(n)	1	
Inner loop	O(n)	1	O(n)
Total time complexity			O(n^2)

The outer loop iterates through the entire array. In every iteration it goes through the rest of the array to find the smallest element and swap with the element in index i. Therefore the overall time complexity of this algorithm is O(n^2).

In this algorithm the time complexity of best case, average and worst case is O(n^2) because even if the array is sorted, at every iteration of the outer loop, the inner loop iterates the rest of the loop but does not execute the swap function. So the time complexity of best, average and worst cases for this algorithm is O(n^2).

Graph for bubble sort algorithm

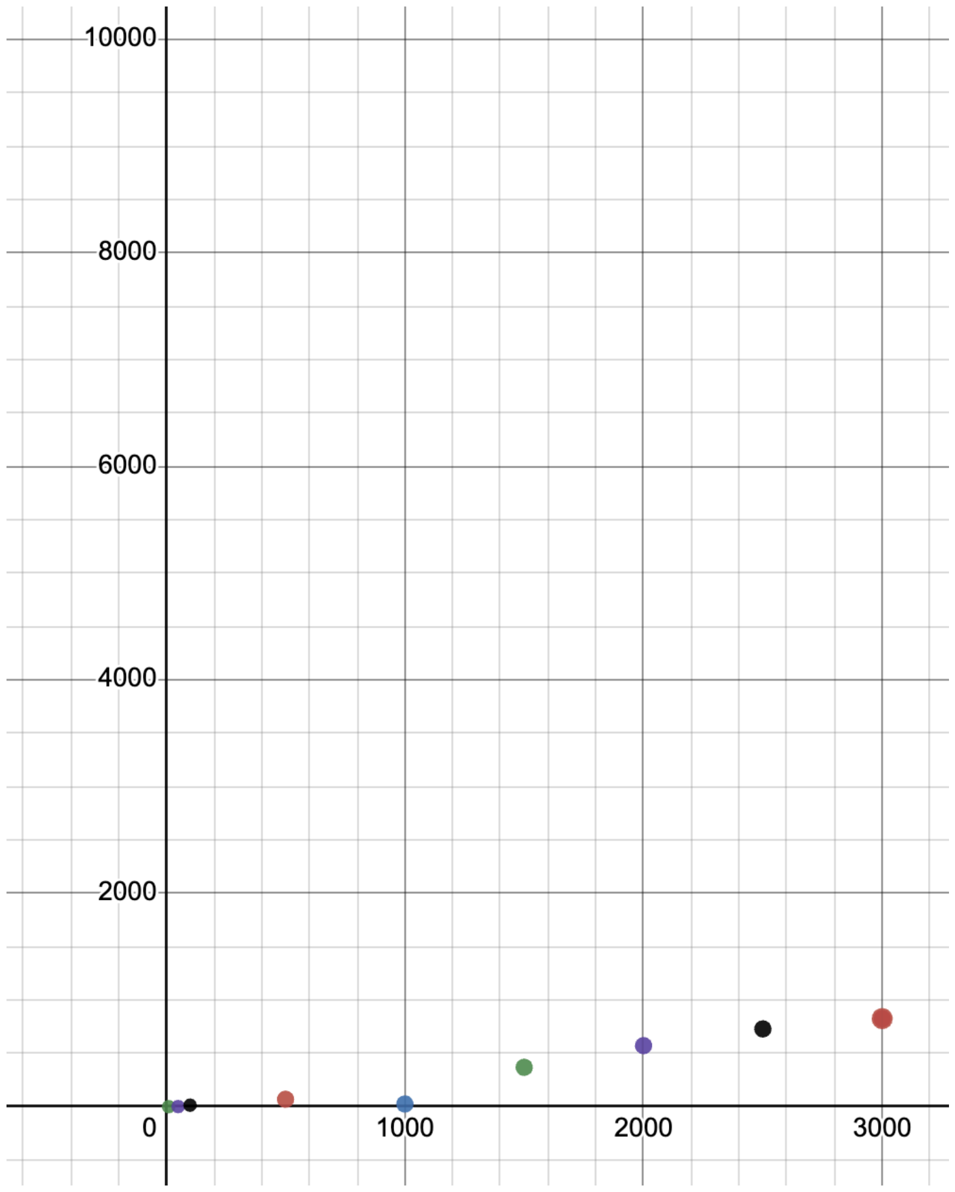
X-axis : number of elements in list(n)
Y-axis: computation time(10^-5 seconds)



This is the practical computation time of the algorithm of different input sizes.
This graph is very similar to the graph of O(n^2).

Graph for insertion sort algorithm

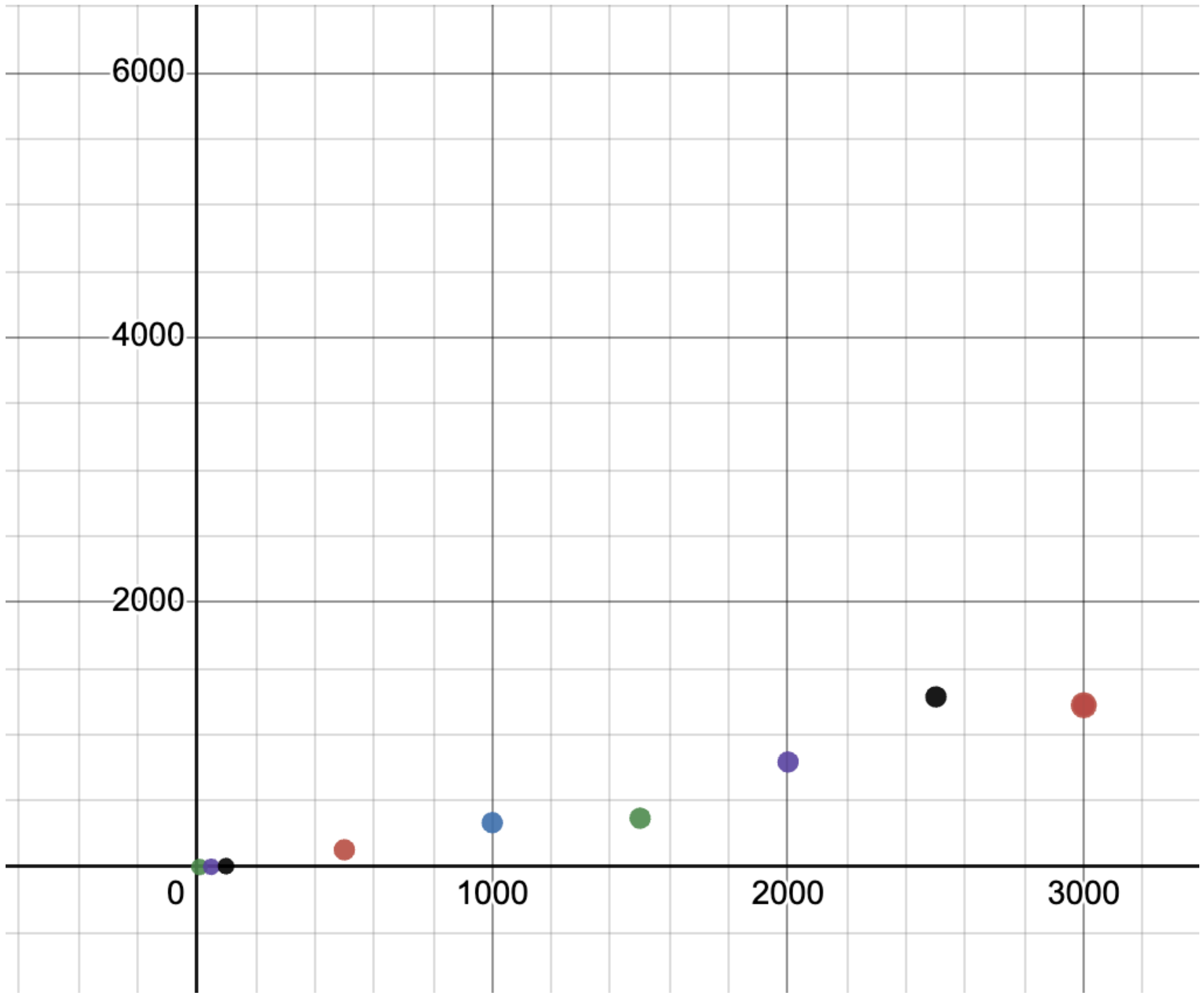
X-axis : number of elements in list(n)
Y-axis: computation time(10^-5 seconds)



The time complexity of the algorithm is also O(n^2), but the practical computation times of arrays of the same are much lower than computation time when bubble sort is used.

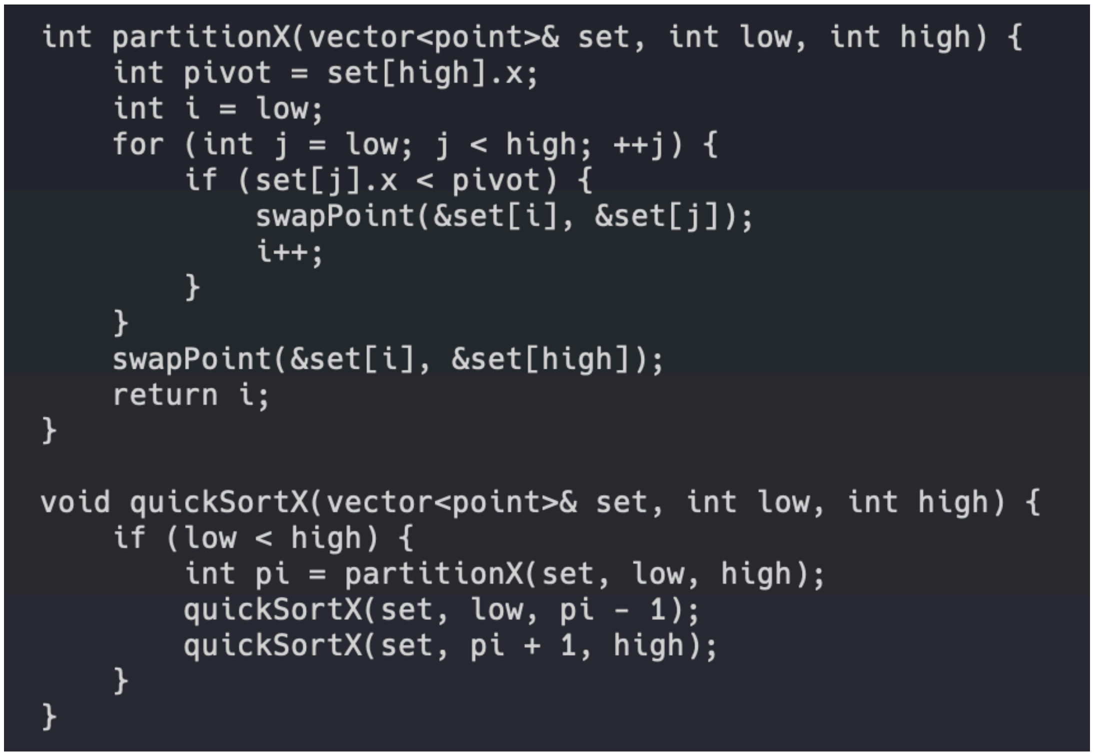
Graph for selection sort algorithm

X-axis : number of elements in list(n)
Y-axis: computation time(10^-5 seconds)



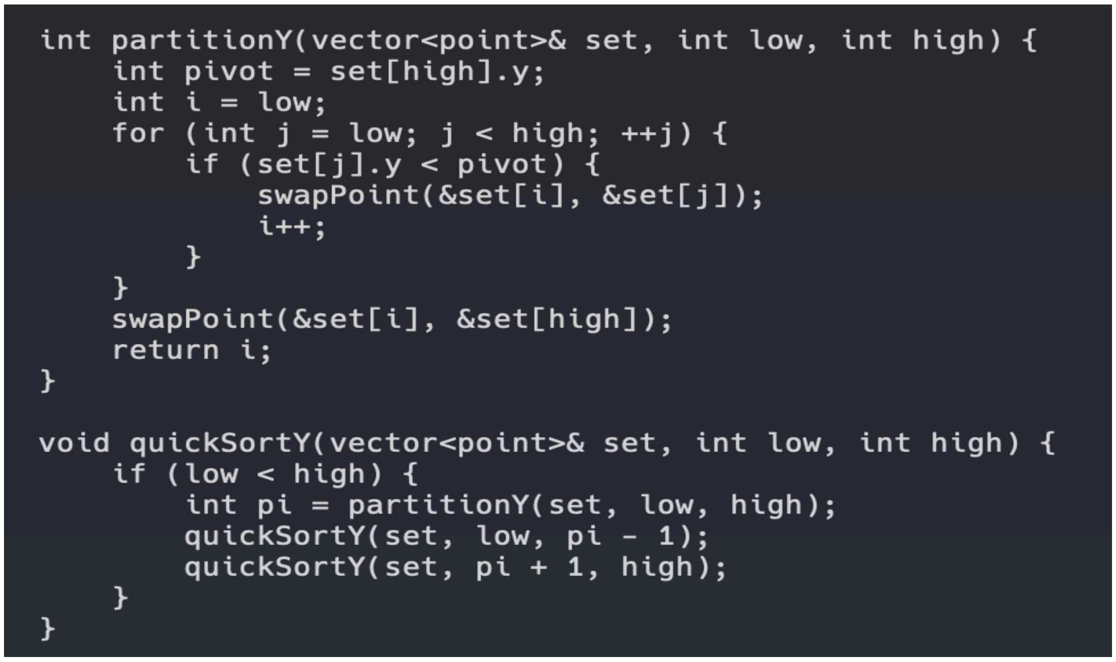
Here is the practical computation times. As you can see this is very similar to the graph of O(n^2).

Q3)



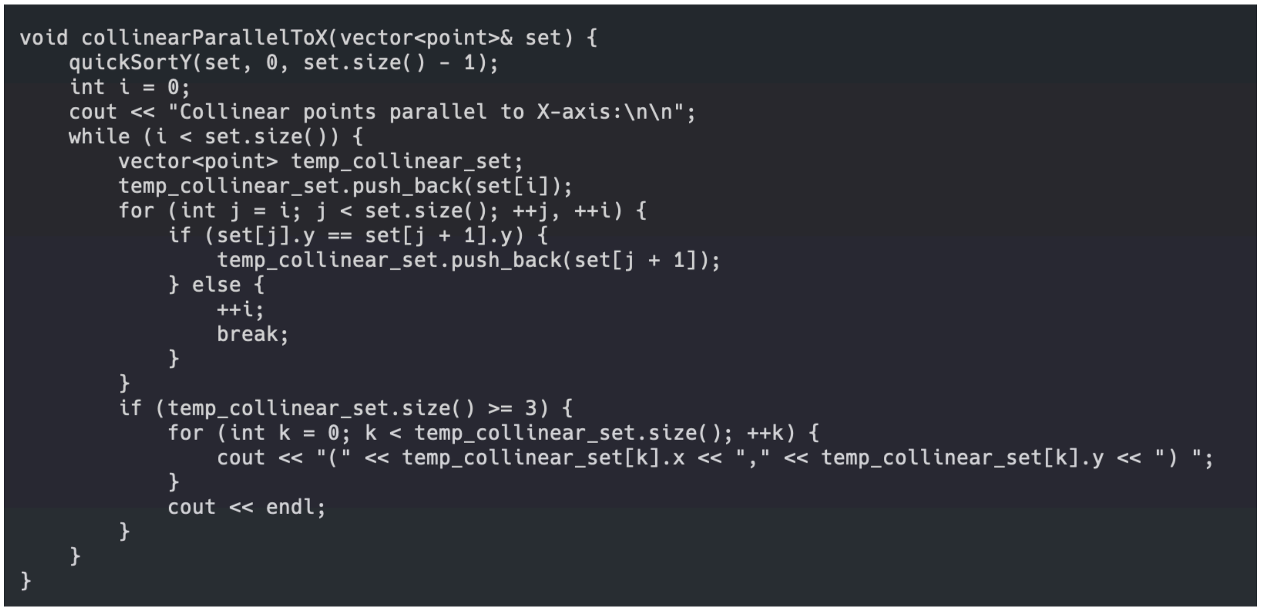
```
int partitionX(vector<point>& set, int low, int high) {
    int pivot = set[high].x;
    int i = low;
    for (int j = low; j < high; ++j) {
        if (set[j].x < pivot) {
            swapPoint(&set[i], &set[j]);
            i++;
        }
    }
    swapPoint(&set[i], &set[high]);
    return i;
}

void quickSortX(vector<point>& set, int low, int high) {
    if (low < high) {
        int pi = partitionX(set, low, high);
        quickSortX(set, low, pi - 1);
        quickSortX(set, pi + 1, high);
    }
}
```

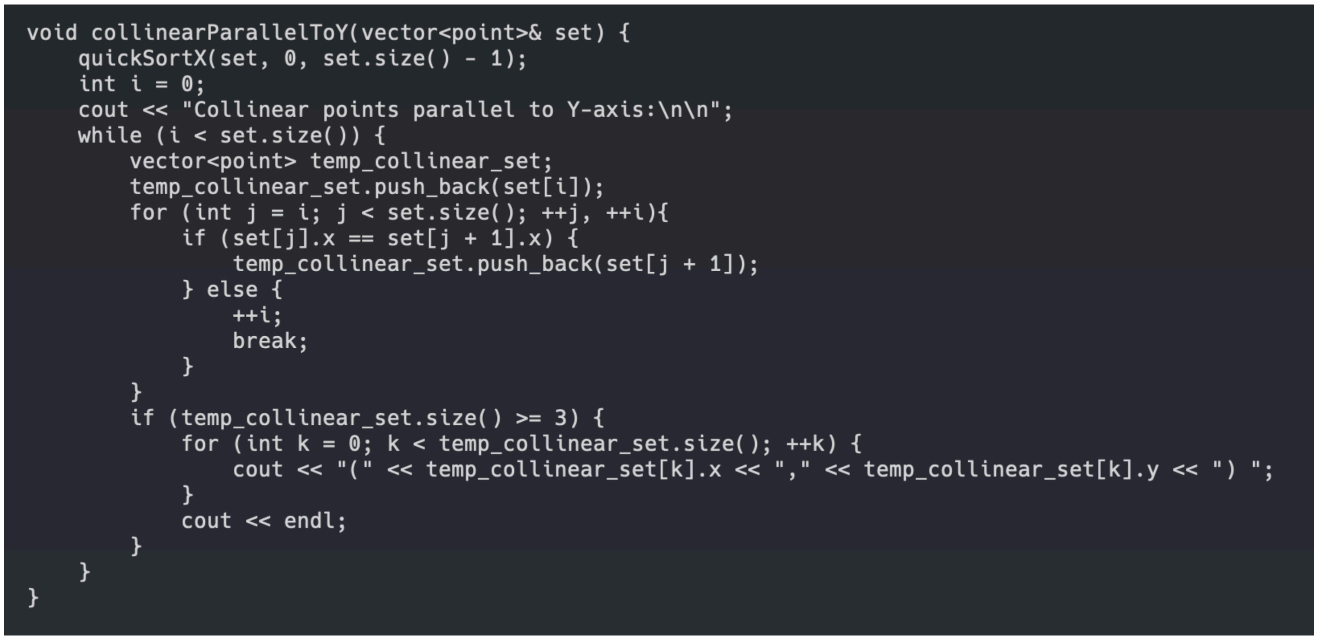


```
int partitionY(vector<point>& set, int low, int high) {
    int pivot = set[high].y;
    int i = low;
    for (int j = low; j < high; ++j) {
        if (set[j].y < pivot) {
            swapPoint(&set[i], &set[j]);
            i++;
        }
    }
    swapPoint(&set[i], &set[high]);
    return i;
}

void quickSortY(vector<point>& set, int low, int high) {
    if (low < high) {
        int pi = partitionY(set, low, high);
        quickSortY(set, low, pi - 1);
        quickSortY(set, pi + 1, high);
    }
}
```



```cpp
void collinearParallelToX(vector<point>& set) {
    quickSortY(set, 0, set.size() - 1);
    int i = 0;
    cout << "Collinear points parallel to X-axis:\n\n";
    while (i < set.size()) {
        vector<point> temp_collinear_set;
        temp_collinear_set.push_back(set[i]);
        for (int j = i; j < set.size(); ++j, ++i) {
            if (set[j].y == set[j + 1].y) {
                temp_collinear_set.push_back(set[j + 1]);
            } else {
                ++i;
                break;
            }
        }
        if (temp_collinear_set.size() >= 3) {
            for (int k = 0; k < temp_collinear_set.size(); ++k) {
                cout << "(" << temp_collinear_set[k].x << "," << temp_collinear_set[k].y << ") ";
            }
            cout << endl;
        }
    }
}
```



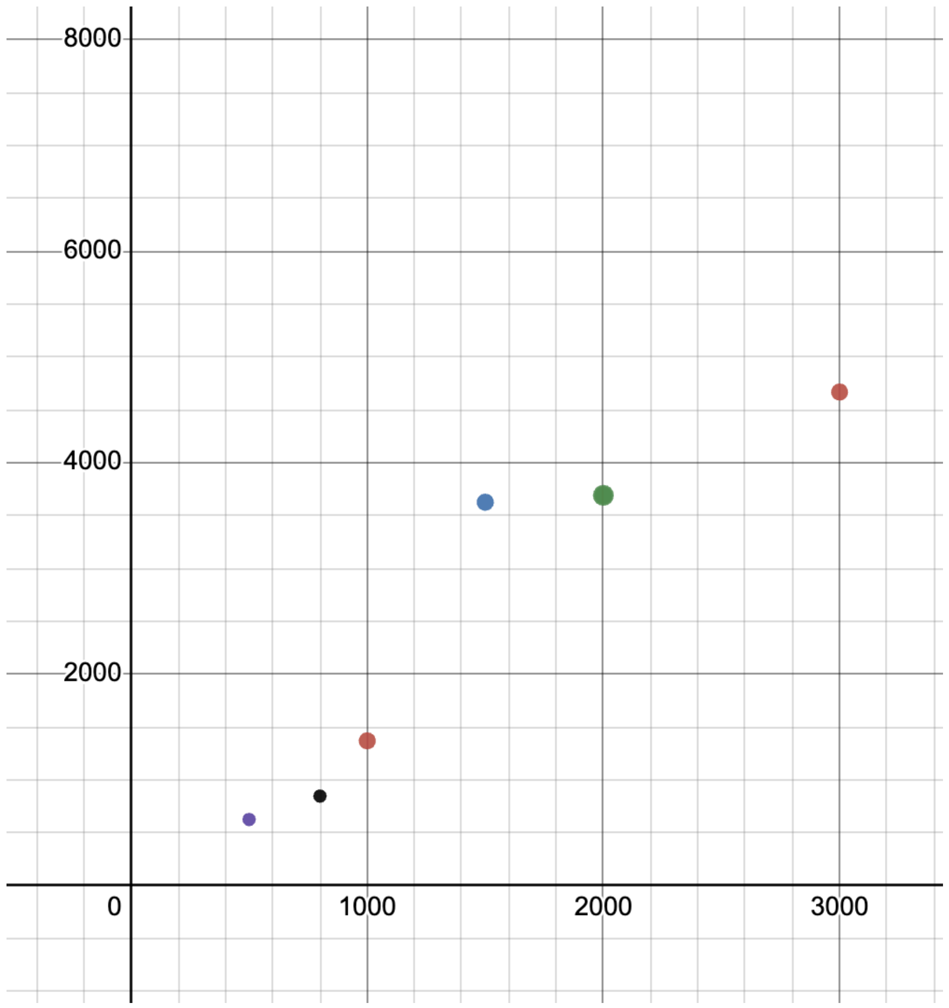
```cpp
void collinearParallelToY(vector<point>& set) {
    quickSortX(set, 0, set.size() - 1);
    int i = 0;
    cout << "Collinear points parallel to Y-axis:\n\n";
    while (i < set.size()) {
        vector<point> temp_collinear_set;
        temp_collinear_set.push_back(set[i]);
        for (int j = i; j < set.size(); ++j, ++i){
            if (set[j].x == set[j + 1].x) {
                temp_collinear_set.push_back(set[j + 1]);
            } else {
                ++i;
                break;
            }
        }
        if (temp_collinear_set.size() >= 3) {
            for (int k = 0; k < temp_collinear_set.size(); ++k) {
                cout << "(" << temp_collinear_set[k].x << "," << temp_collinear_set[k].y << ") ";
            }
            cout << endl;
        }
    }
}
```

Operation	**Number of Operations**	**Frequency**	**Time complexity**
Sorting the set of points based on X or Y coordinate	O(n log n)	1	O(n log n)
Outer loop	O(n)	1	O(n)
Inner loop	O(n)	1	O(n)

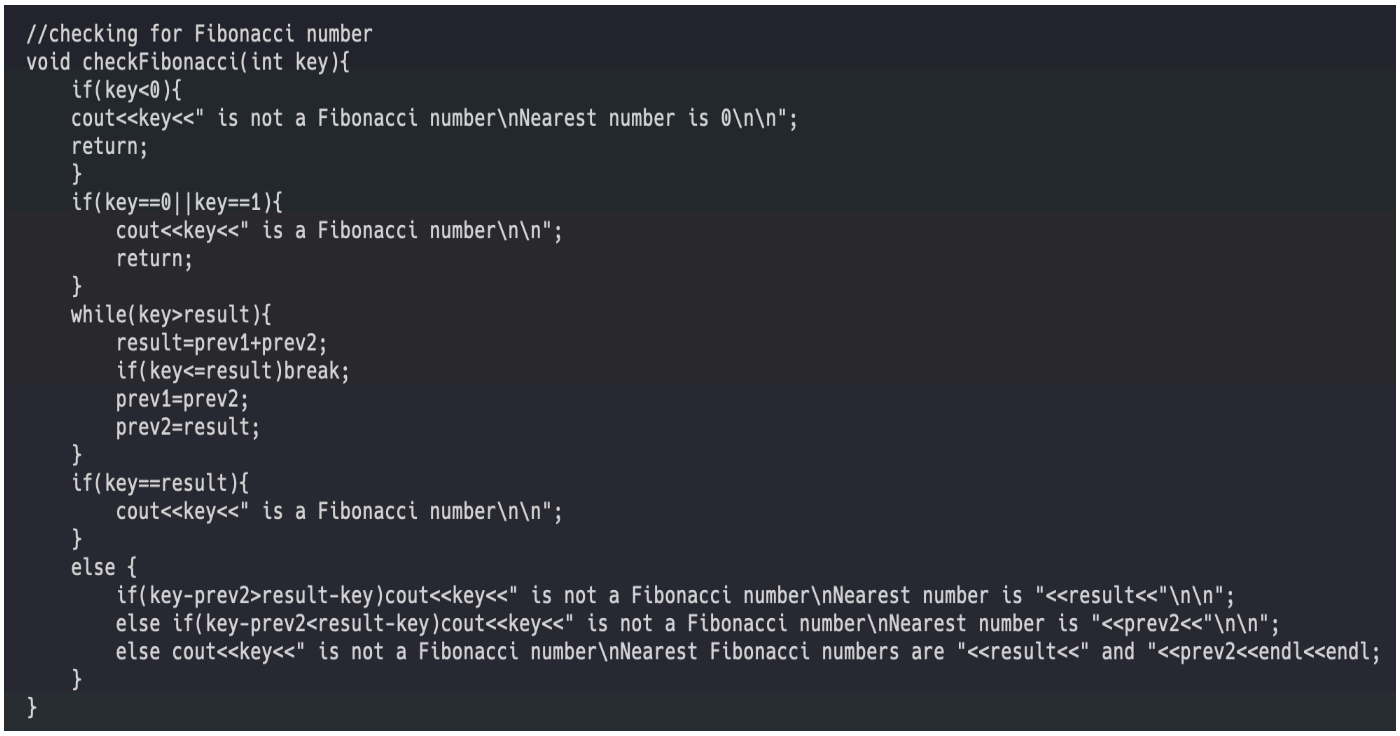
In this algorithm, we first sort the points based on X or Y coordinate using the quick sort algorithm. So the time complexity of that operation is O(n log n). After that the two nested loops have time complexity of O(n). But the overall time complexity of the two loops together is O(n), because after the outer loop iterates through a few elements, the inner loop starts from that position itself. Then as the inner loop iterates through, the value of ‘i’ also increases so when it breaks from the inner loop the outer loop continues from where the inner loop left off.

Graph for algorithm

X-axis : number of elements in list(n)
Y-axis: computation time(microseconds)



Q4)



```
//checking for Fibonacci number
void checkFibonacci(int key){
    if(key<0){
    cout<<key<<" is not a Fibonacci number\nNearest number is 0\n\n";
    return;
    }
    if(key==0||key==1){
        cout<<key<<" is a Fibonacci number\n\n";
        return;
    }
    while(key>result){
        result=prev1+prev2;
        if(key<=result)break;
        prev1=prev2;
        prev2=result;
    }
    if(key==result){
        cout<<key<<" is a Fibonacci number\n\n";
    }
    else {
        if(key-prev2>result-key)cout<<key<<" is not a Fibonacci number\nNearest number is "<<result<<"\n\n";
        else if(key-prev2<result-key)cout<<key<<" is not a Fibonacci number\nNearest number is "<<prev2<<"\n\n";
        else cout<<key<<" is not a Fibonacci number\nNearest Fibonacci numbers are "<<result<<" and "<<prev2<<endl<<endl;
    }
}
```

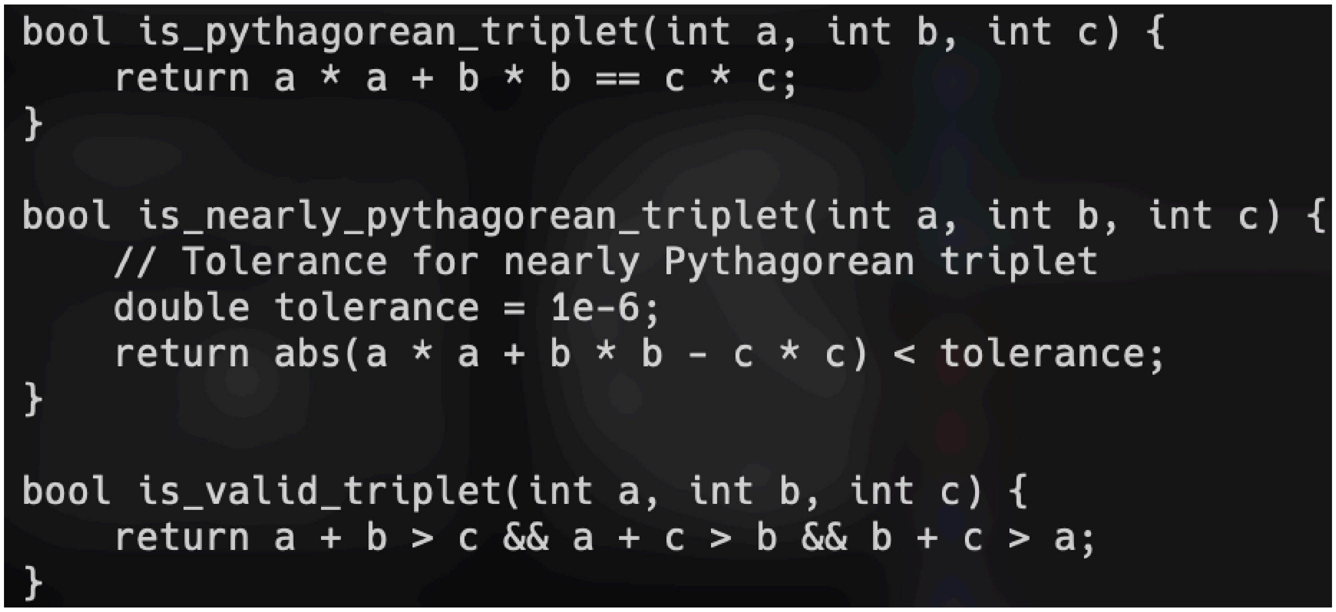
<u>Time complexity analysis:</u>

Since we are only inputting a single integer. Time complexity depends on how large the number is. The greater the number, then the algorithm has to run that much longer.

Let F(n) be a function which shows the largest fibonacci number less than n. Therefore larger the value of F(n), greater the time complexity.

Therefore the time complexity of this algorithm is O(F(n)).

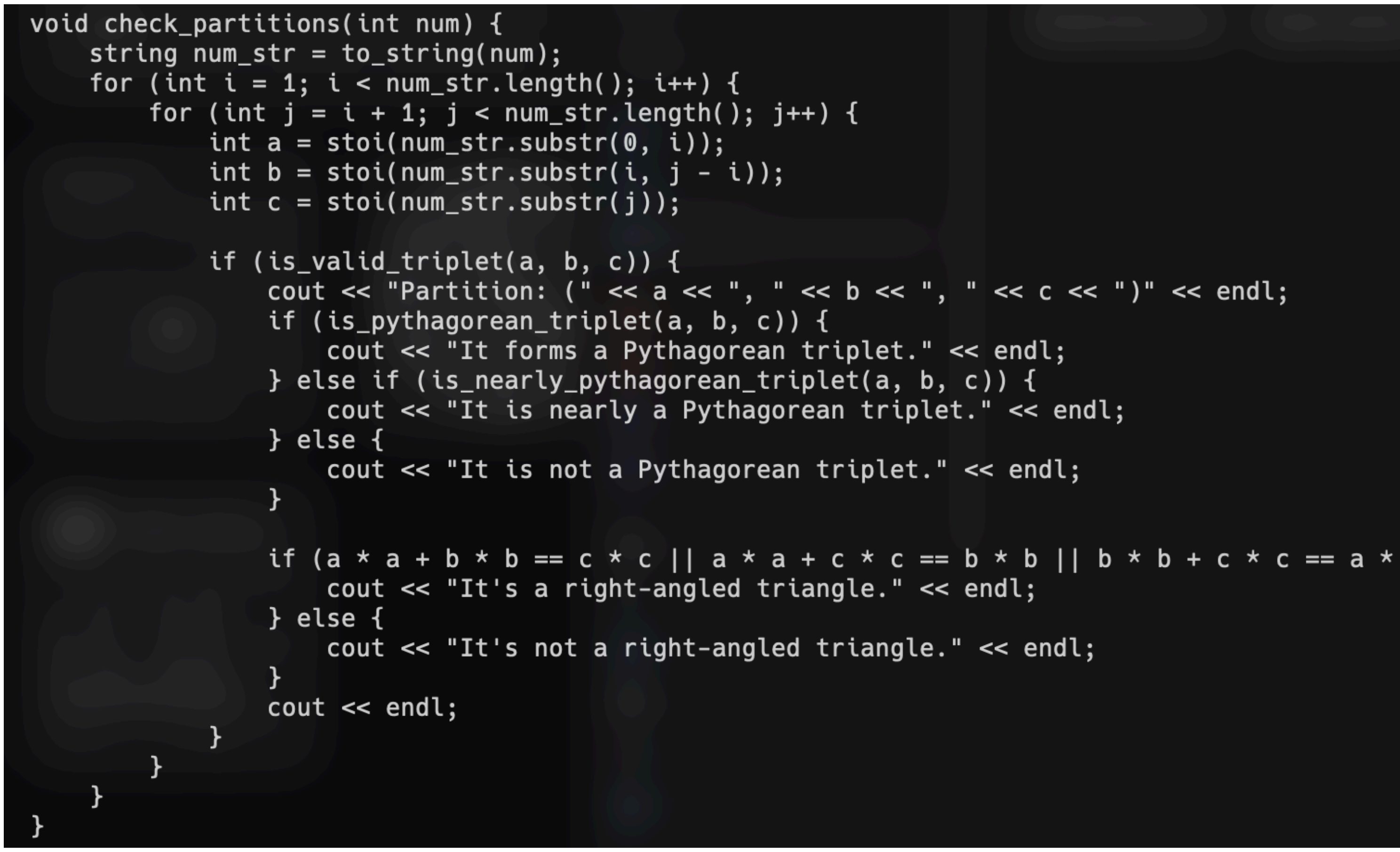
Q5)



```
bool is_pythagorean_triplet(int a, int b, int c) {
    return a * a + b * b == c * c;
}

bool is_nearly_pythagorean_triplet(int a, int b, int c) {
    // Tolerance for nearly Pythagorean triplet
    double tolerance = 1e-6;
    return abs(a * a + b * b - c * c) < tolerance;
}

bool is_valid_triplet(int a, int b, int c) {
    return a + b > c && a + c > b && b + c > a;
}
```



```
void check_partitions(int num) {
    string num_str = to_string(num);
    for (int i = 1; i < num_str.length(); i++) {
        for (int j = i + 1; j < num_str.length(); j++) {
            int a = stoi(num_str.substr(0, i));
            int b = stoi(num_str.substr(i, j - i));
            int c = stoi(num_str.substr(j));

            if (is_valid_triplet(a, b, c)) {
                cout << "Partition: (" << a << ", " << b << ", " << c << ")" << endl;
                if (is_pythagorean_triplet(a, b, c)) {
                    cout << "It forms a Pythagorean triplet." << endl;
                } else if (is_nearly_pythagorean_triplet(a, b, c)) {
                    cout << "It is nearly a Pythagorean triplet." << endl;
                } else {
                    cout << "It is not a Pythagorean triplet." << endl;
                }

                if (a * a + b * b == c * c || a * a + c * c == b * b || b * b + c * c == a *
                    cout << "It's a right-angled triangle." << endl;
                } else {
                    cout << "It's not a right-angled triangle." << endl;
                }
                cout << endl;
            }
        }
    }
}
```

Time complexity analysis:

In this algorithm,we are using two for loops to find the number of partitions. The size of the number that we have to partition is 5.

Operation	**Computation time**	**frequency**	**Time complexity**
Outer loop	O(n)	1	O(n)
Inner loop	O(n)	1	O(n)

Therefore the time complexity of this algorithm is O(n^2).