Design and Analysis of Algorithms (DAA)
Mid-Sem Assignment

Name: Vatsal Bhuva
Roll No.: IIT2022004
Section: A

Q1: Find the largest sorted subsequence starting with different positions in the array (find all occurrences of the largest sorted subsequence along with the start location).

Ans. The `findLIS()` function computes the length of the Longest Increasing Subsequence (LIS) for a given array in O(n^2) time complexity.

The `get_LIS()` function then recursively explores all valid combinations of LIS that can be generated from the given array.

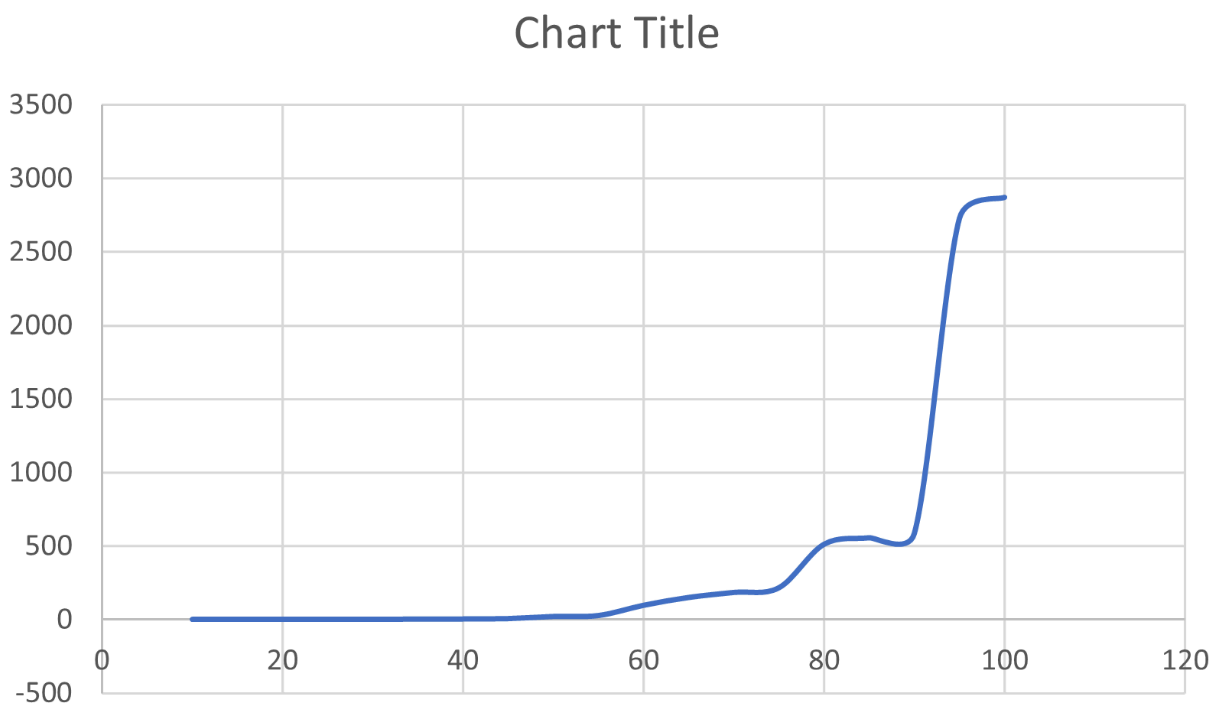
Afterward, it constructs the LIS deque by backtracking and reverses it before returning.

Now, we proceed to evaluate each possible valid combination for this longest length of LIS. This evaluation is carried out recursively in O(2^n) time complexity.

```cpp
#include <bits/stdc++.h>
using namespace std;
void findLIS(vector<vector<int>> &result, vector<int> &temp, vector<int> &arr,
             int index)
{
    if (index == arr.size())
    {
        if (temp.size() > 0)
        {
            result.push_back(temp);
        }
        return;
    }
    if (temp.empty() || arr[index] > temp.back())
    {
        temp.push_back(arr[index]);
        findLIS(result, temp, arr, index + 1);
        temp.pop_back();
    }
    findLIS(result, temp, arr, index + 1);
}
```

```cpp
vector<vector<int>> get_LIS(vector<int> &arr)
{
    vector<vector<int>> result;
    vector<int> temp;
    findLIS(result, temp, arr, 0);
    int max_length = 0;
    for (auto &lis : result)
    {
        max_length = max(max_length, (int)(lis.size()));
    }
    vector<vector<int>> longest_lis;
    for (auto &lis : result)
    {
        if (lis.size() == max_length)
        {
            longest_lis.push_back(lis);
        }
    }
    return longest_lis;
}
```

```cpp
void printLIS(vector<vector<int>> &arr)
{
    for (auto vec : arr)
    {

        for (auto x : vec)
            cout << x << " ";
        cout << endl;
    }
}
```



Time Complexity:

1. Longest len of LIS is found in O(n*n).
2. Then all possible valid LISs are found using recursion which takes time of O(2^lenOfLIS) which reduces to O(2^n) in the worst case.
3. Thus overall time complexity will be O(n*n) + O(2^n) = O(2^n).

Space Complexity:

The algorithm uses additional array of size n to store the LIS, and O(n*n) is used by get_LIS() function. Therefore, the space complexity is O(n*n).

Q2: Print the number of swapping operations while sorting an array of 1000 positive integers using bubble sort, insertion sort, and selection sort.

Ans. We are required to find the number of swapping operations in three commonly yet not-so-efficient swapping algorithms, bubble sort, insertion sort, and selection sort.
All of these sorting techniques work by swapping elements to sort a part of the array in each step, till the entire array is sorted.

Bubble Sort:

```
int bubble_sort(vector<int> arr, int n)
{
    int swaps = 0;
    for (int i = 0; i < n; i++)
    {
        bool isSwapped = false;
        for (int j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                swap(arr[j], arr[j + 1]);
                swaps++;
                isSwapped = true;
            }
        }
        if (!isSwapped)
            return swaps;
    }
    return swaps;
}
```

The bubble sort algorithm works by placing the largest element at the end of the array for each iteration, and iterating over the unsorted part for each loop.

It achieves this by swapping the jth element with the (j+1)th element when each comparison is made, to ensure that the (j+1)th element is always >= jth element in each iteration of the inner loop.

Apriori Analysis:

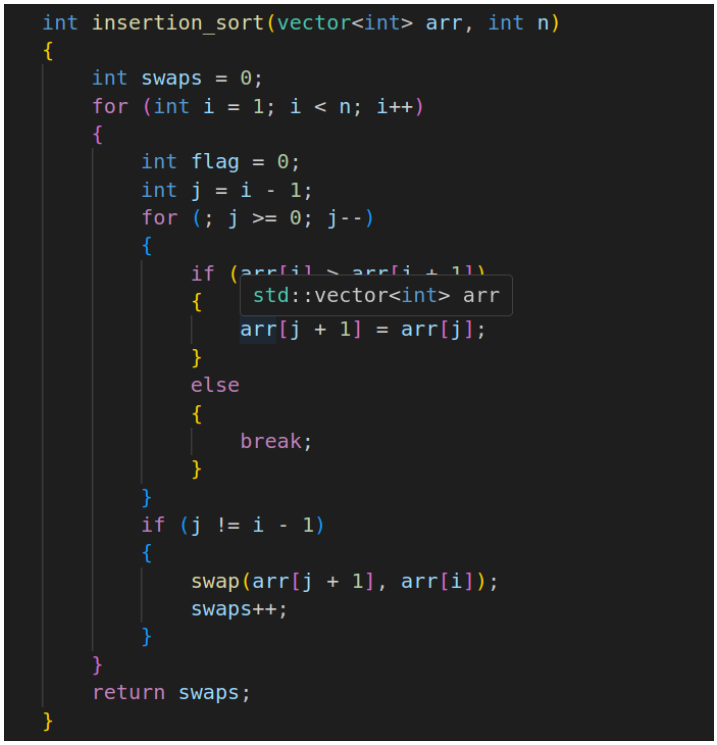
Code	Operations	Frequency	Total
int swaps = 0;	1	1	1
for (int i = 0; i < n; i++)	3	n+1	3(n+1)
for (int j = 0; j < n - i - 1;j++)	3	n(n-i)	3n(n-i)
if (arr[j] > arr[j + 1]);	1	n(n-i)	n(n-i)
swap(arr[j], arr[j + 1]);	1	n(n-i)	n(n-i)
swaps++;	1	n(n-i)	n(n-i)

Time complexity:
As can be seen, the total number of operations is 6n^2-6n(n+1)/2+4. Hence, the time complexity of this algorithm is O(n^2).
The worst case when this algorithm is used, is when the array is sorted in a decreasing order. In that case, the number of swaps would be n(n+1)/2, as for each iteration of the inner loop, there will always be one swap (since the larger elements from the left have to be moved to the right)
The best case is when the array is already sorted, resulting in no swaps and an O(n) algorithm.

Insertion Sort:

Insertion sort works by moving the current element to its corrected position in the sorted array. This algorithm assumes that the array up

to the current index is sorted, and moves the element at the current index (part of the unsorted array) to its correct place towards the left in the sorted array iteratively.
If x (0<=x<i) is the correct position of the element in the sorted part of the array (i is the position of the current element), then all the elements from x to i are shifted to the right by one (or copied to the right by one), and finally the element at the xth index is replaced by the current element, effectively reducing the number of swaps.
This results in only one swap whenever an element is to be inserted in its correct position.



Apriori Analysis:

Code	Operations	Frequency	Total

int swaps = 0;	1	1	1
for (int i = 1; i < n; i++)	3	n	3n
for (int j = i; j >= 0; j--)	3	n(i+1)	3n(i+1)
if (arr[j] < arr[j - 1])	1	m*i	m*i
arr[j] = arr[j - 1];	1	m*i	m*i
else {	1	(n-m)*i	(n-m)*i
swap(arr[j], arr[i]);	1	(n-m)*i	(n-m)*i
swaps++;	1	(n-m)*i	(n-m)*i
break;}	1	(n-m)*i	(n-m)*i

Time complexity:

The total number of operations is approximately: 6n + 8n(n+1)/2 + 2, which is of the order of n^2 (O(n^2)).

However, the number of swaps is much lesser than that in bubble sort, as this algorithm swaps only once per each iteration of i (unlike bubble sort, which swaps for each iteration of j if the elements are not ordered)

The worst case for insertion sort is when the array is in descending order, and the best case is when the array is already sorted, minimizing the number of swaps done. In both cases however, the algorithm runs with an O(n^2) time complexity, as we still have to check whether the element can be inserted anywhere towards the left or not.

Selection Sort:

This sorting algorithm selects the required element from the unsorted array on the right to be inserted at the current index of the iteration.

It assumes that the entire array is sorted up to the current index, and now we select the element that is just bigger than the last element of the sorted part to be placed at the current index.
Hence, selection sort also requires only 1 swap per iteration to place the correct element at the current position.

```
int selection_sort(vector<int> arr, int n)
{
    int swaps = 0;

    for (int i = 0; i < n; i++)
    {
        int mini = arr[i];
        int miniIndex = i;
        for (int j = i + 1; j < n; j++)
        {
            if (arr[j] < mini)
            {
                mini = arr[j];
                miniIndex = j;
            }
        }
        if (miniIndex != i)
        {
            swap(arr[i], arr[miniIndex]);
            swaps++;
        }
    }

    return swaps;
}
```

Code	Operations	Frequency	Total
int swaps = 0;	1	1	1
for (int i = 0; i < n; i++){	3	n+1	3(n+1)
int mini = arr[i];	1	n	n
int miniIndex = i;	1	n	n
int flag = 0;	1	n	n
for (int j = i + 1; j < n; j++){	3	(n)(n-i)	3(n)(n-i)

flag = 1;	1	(n)(n-i-1)	(n)(n-i-1)
if (arr[j] < mini){	1	(m)(n-i-1)	(m)(n-i-1)
mini = arr[j];	1	(m)(n-i-1)	(m)(n-i-1)
miniIndex = j;}}	1	(m)(n-i-1)	(m)(n-i-1)
if (flag){	1	(a)(n-i-1)	(a)(n-i-1)
swap(arr[i], arr[miniIndex]);	1	(a)(n-i-1)	(a)(n-i-1)
swaps++;}}	1	(a)(n-i-1)	(a)(n-i-1)

Time complexity:

Total number of operations is approximately: 6n + 4 + 4n^2 + 3mn + 3an ~ Time complexity is O(n^2)

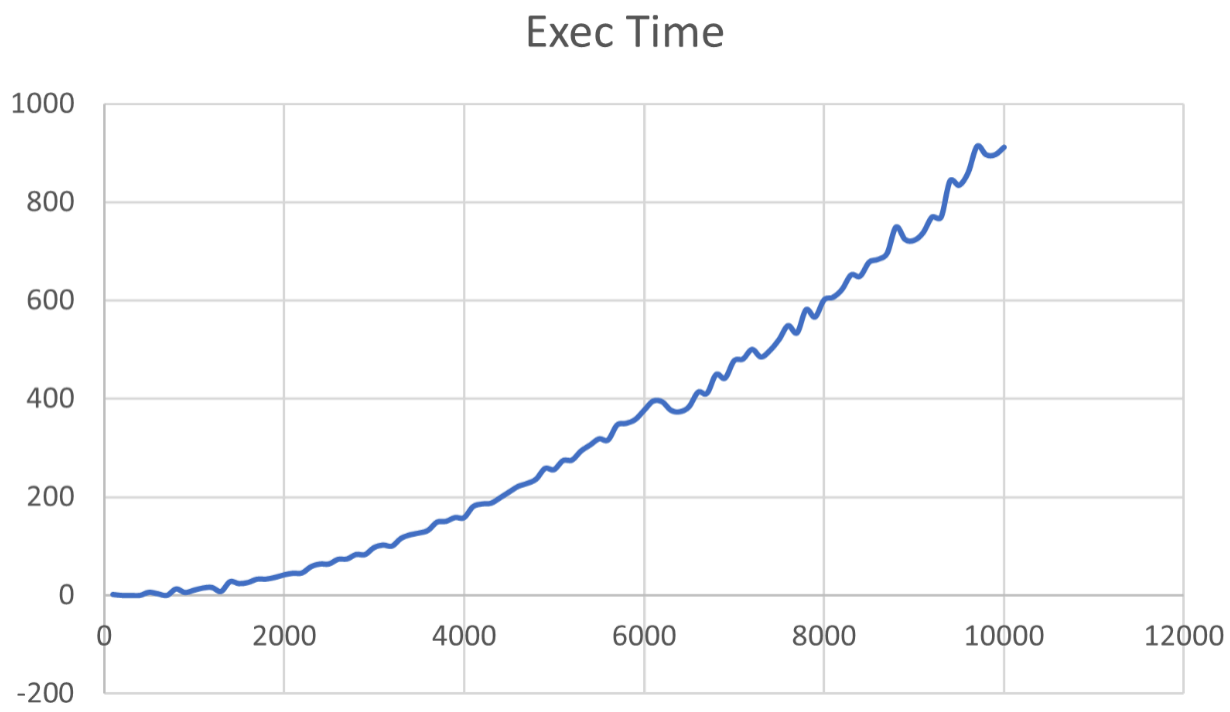
Because of the way this algorithm works, there are at max n-1 swaps in this algorithm depending on how the vector is sorted. There are no swaps if the array is already sorted, and n-1 swaps if no element is in it's correct position.
Hence, in terms of the number of swaps, insertion sort and selection sort perform better than bubble sort.

Practical Results:

No. of swaps for the algorithms depending upon the array:

Algorithm	Sorted Array	Random Array	Descending
Bubble Sort	0	237958	499019
Insertion Sort	0	994	622
Selection Sort	0	991	998

As can be seen, bubble sort has a high number of swaps always as compared to insertion sort and selection sort.



Time Complexity:

- All three sorting algorithms have a worst-case time complexity of O(n^2) due to nested loops iterating over the entire array.
- Insertion sort can achieve best-case time complexity of O(n) when the array is already sorted.
- Selection sort has no specific best-case scenario.

Space Complexity:

All three algorithms have a space complexity of O(1) as they use constant extra space besides the input array.

Q3: Given a set of 5000 randomly generated 2-D points, find out collinear points parallel to the (a) X-axis, and (b) Y-axis.

Ans. This problem requires us to firstly generate 5000 random points (achieved by generating 2 random numbers for each iteration, one becomes the x coordinate and the other y) using the srand(time(0)) function to randomize the points every time the code is run.

These points are not directly stored in a data structure, but rather segregated when input is taken into two maps: one which stores the points collinear to it and parallel to x-axis, and the other map stores the points collinear to it and parallel to y-axis.

The numbers are scaled between 0 to 99 for ease of readability and output, though it works perfectly well for any range as well.

```
using namespace std;

int main()
{
    srand(time(0));
    vector<pair<int, int>> points;
    /*-*/
    unordered_map<int, vector<int>> xCollinear, yCollinear;
    int n = 5000;

    for (int i = 0; i < n; i++)
    {
        int x, y;
        x = rand() % n;
        y = rand() % n;
        yCollinear[x].push_back(y);
        xCollinear[y].push_back(x);
    }

    // i) collinear to x axis means y same.
    cout << "Points collinear to x-axis:" << endl;
    for (auto pr : xCollinear)
    {
        cout << "At y=" << pr.first << ":\n";
        if (pr.second.size() > 0)
        {
            for (auto xCoord : pr.second)
            {
                cout << "(" << pr.first << "," << xCoord << "), ";
            }
            cout << endl;
        }
    }
    cout << endl;
    cout << endl;
    cout << endl;
    cout << "Points collinear to y-axis:" << endl;
    for (auto pr : yCollinear)
    {
        cout << "At x=" << pr.first << ":\n";
        if (pr.second.size() > 0)
        {

            for (auto yCoord : pr.second)
            {
                cout << "(" << pr.first << "," << yCoord << "), ";
            }
            cout << endl;
        }
    }

    return 0;
}
```

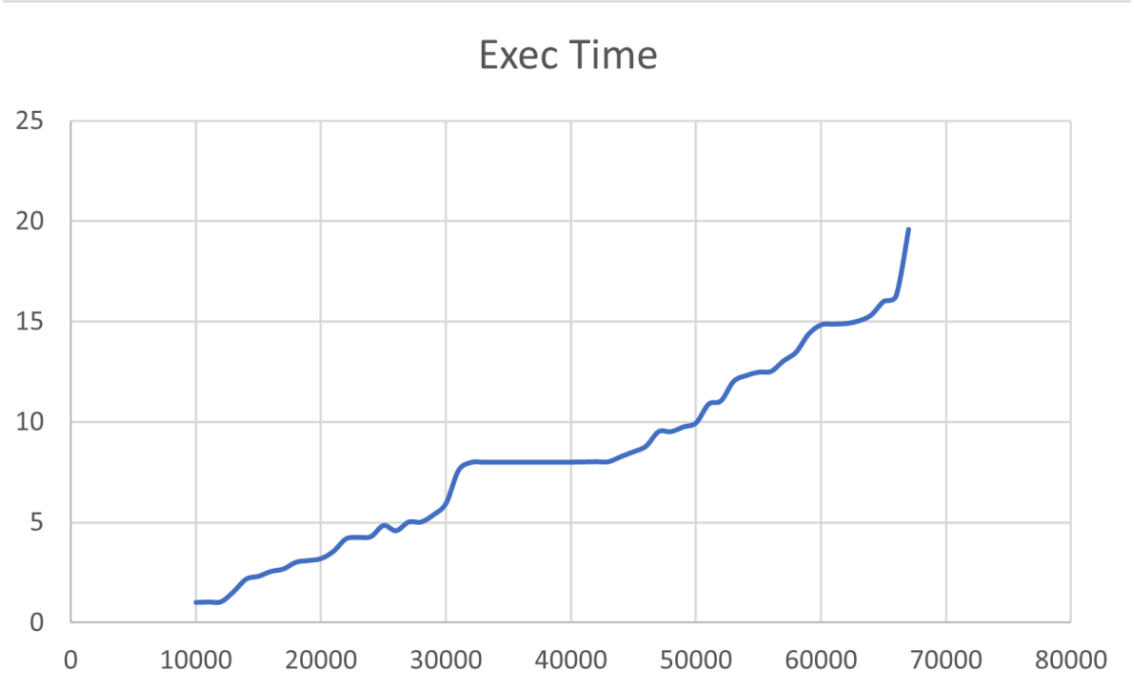
Time Complexity:

Since the number of points (5000) are constant, the time complexity is Θ(nlogn). This is because inserting points in an ordered map is a log(n) operation.

In this problem, it is better to use an unordered map over an ordered map as there are lower chances of collisions in the map, thus reducing the time complexity (insertion in unordered map takes O(1) time when there are less collisions, as compared to ordered map which takes O(logn)).

In the best case (no collinear points), the time complexity will hence be O(n), while in the worst case (all collinear points), the time complexity will be O(n^2). The alternative to this is to use an ordered map, which takes O(logn) for insertion in all cases and an overall time complexity of O(nlogn).

Also, the space complexity in this case is O(2*n), as we are using two maps to store two kinds of information about the points: for



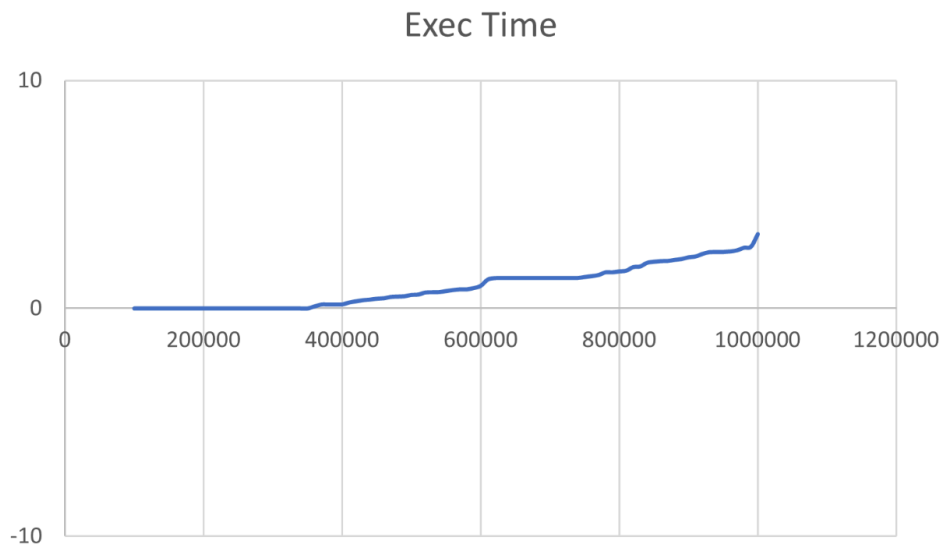
each point, we are storing the points that are collinear to it and parallel to the x-axis, and the y-axis, in the respective maps.

Q4: Check whether a given number is a Fibonacci number or not, and design an algorithm to find the nearest Fibonacci number.

Ans. This can be solved by using the concepts of precomputation. We precompute the Fibonacci numbers and store them in an array, so that the Fibonacci number can be accessed in O(logn) time using Binary Search.
If the element to search is not in that array, we are sure that the high and low pointers will point to two numbers: one which is just lower than the target, and one which is just higher than the target.
We then print that number which results in a minimum difference with the target element.

```cpp
/*
NAME: VATSAL BHUVA
ROLL NO.: IIT2022004
SECTION: A
QUESTION: 4
*/

#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> fib = {0, 1};
    int prev = fib[0], curr = fib[1];
    for (int i = 0; i < 45; i++)
    {
        int next = prev + curr;
        fib.push_back(next);
        prev = curr;
        curr = next;
    }
    int q;
    cin >> q;
```

```
int q;
cin >> q;
while (q--)
{
    int lo = 0, hi = 45;
    int target;
    cout << "Enter element to verify in Fib. : ";
    cin >> target;

    int flag = 0;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        if (fib[mid] == target)
        {
            cout << "Element belongs to fibonacci series, at position " << mid + 1 << endl;
            flag = 1;
            break;
        }
        else
        {
            if (fib[mid] < target)
                lo = mid + 1;
            else
                hi = mid - 1;
        }
    }
    if (!flag)
    {
        cout << "Element does not belong to fibonacci series.\n";
        if (target - fib[hi] < fib[lo] - target)
        {
            cout << "Closest element is: " << fib[hi] << endl;
        }
        else
        {
            cout << "Closest element is: " << fib[lo] << endl;
        }
    }
}

return 0;
```



Time Complexity:
Since there are only 48 Fibonacci numbers that are within the integer limit, the time complexity of storing these numbers is O(48).
Now, for executing the q queries, we apply binary search on these stored numbers, and output the result depending on whether the element is found in the array (output is the position of the fibonacci number), or not (in which case, we output the closest fibonacci number).
Hence, this results in an overall time complexity of O(q*log(48)).

Space Complexity: The space complexity of the provided code can be broken down into several components. Firstly, a vector named 'fib' is initialized to store Fibonacci numbers, which grows in size as Fibonacci numbers are computed and stored within it. Since the maximum number of Fibonacci numbers computed is fixed at 45, the space required by this vector is O(1) as it's a constant size.

Q5: Take a 5-digit number, make 3 partitions, and check whether it is a Pythagorean triplet or not. If not, define nearly Pythagorean. Check how nearly it is right-angled. Check for all possible partitions.

Ans. This code requires us to first convert the given number to a string, with each digit as a character in the string. This makes it easier to make partitions in the string.

```cpp
#include <bits/stdc++.h>
using namespace std;

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

vector<vector<int>> triplets(int n)
{
    string s = to_string(n);
    vector<vector<int>> ans;
    for (int i = 1; i < 4; i++)
    {
        for (int j = i + 1; j < 5; j++)
        {
            if (i > 0 && j - i > 0 && 5 - j > 0)
            {
                int num1 = stoi(s.substr(0, i));
                int num2 = stoi(s.substr(i, j - i));
                int num3 = stoi(s.substr(j));
                if (isValidTriangle(num1, num2, num3))
                {
                    vector<int> triplet = {num1, num2, num3};
                    sort(triplet.begin(), triplet.end(), greater<int>());
                    ans.push_back(triplet);
                }
            }
        }
    }
    return ans;
}
```

The partitions are then split into three substrings
We then validate the partitions to see whether the sum of two sides is greater than the third side (for it to be a valid triangle).

```
void pythagorean(vector<vector<int>> &triplets)
{
    for (auto triplet : triplets)
    {
        int val = triplet[1] * triplet[1] + triplet[2] * triplet[2];
        if (triplet[0] * triplet[0] == val)
        {
            cout << "(" << triplet[0] << "," << triplet[1] << "," << triplet[2] << ") is a pythagorean triplet." <
            return;
        }
        else
        {
            if (abs(val - triplet[0] * triplet[0]) <= 1)
            {

                cout << "(" << triplet[0] << "," << triplet[1] << "," << triplet[2] << ") is nearly pythagorean" <
            }
        }
    }
}
```

```
int main()
{
    cout << "Enter the 5 digit number: ";
    int n;
    cin >> n;
    vector<vector<int>> validTriplets = triplets(n);
    if (validTriplets.size() == 0)
    {
        cout << "There are no valid triangle triplets.";
    }
    else
    {

        cout << "Valid triangle triplets are:\n";
        for (auto vec : validTriplets)
        {
            for (auto elem : vec)
            {
                cout << elem << " ";
            }
            cout << endl;
        }
```

As can be seen in the code, we generate all valid triplets (valid in terms of valid triangles, such that sum of any two sides is greater than the third side) and store them in a 2D vector.
Then for each triplet in that vector, we check whether it's a pythagorean triplet, or an almost pythagorean triplet.

Time Complexity:
The partition generation takes O(1) time nearly (since we have fixed 5 digit integers).
These are stored in a 2D vector.
Now, we traverse through these generated partitions to see which are valid pythagorean triplets, and which are not but are almost pythagorean triplets. The number of valid triangle triplets is very less for 5 digit integers, which makes the traversal through these partitions also close to O(1) time.
Hence, the overall time complexity of this code is O(1).

However, if we have a different number of digits, the time complexity would roughly be: O(log10(n)*n^part) (part = 3 here)

Space Complexity:
The function utilizes a result vector to store partitions, which may expand considerably based on the input values. The recursion depth is limited by the variable 'part', thus the maximum stack space needed is directly proportional to 'part'. Additionally, within the loop, the function generates a subPart vector to hold partitions for the remainder, further contributing to the space complexity. Consequently, the space complexity of the function can be approximated as O(n^part), where 'part' equals 3 in this scenario.