DAA ASSIGNMENT

(AADITYA GOYAL IIB2022038 SEC – C)

Q1.

Code :

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

int len ;
vector<vector<int>> res;

void func (vector<int> &nums , int index ,vector<int> &v){
    if(index>=nums.size()){
        if(v.size()==len+1){
            res.push_back(v);
        }
        return;
    }

    if(v[v.size()-1]<=nums[index]){
        v.push_back(nums[index]);
        func(nums,index+1,v);
        v.pop_back();
    }
    func(nums,index+1,v);
}

int main(){
    vector<int> v={3, 4, 1, 2, 5, 6, 7, 8, 1, 2, 3};

    vector<int> dp(v.size());
    dp[0]=1;

    for(int i=1;i<v.size();i++){
        int m = 1;
        for(int j=0;j<i;j++){
            if(v[j]<=v[i]){
                m=max(1+dp[j],m);
            }
        }
        dp[i]=m;
    }

    // maximum sorted subsequence length possible till the particular index
    // for(int i=0;i<v.size();i++){
```

```
        // cout<<dp[i]<<" ";
    // }
    // cout<<endl;

    len = *max_element(dp.begin(),dp.end());

    vector<int> temp;
    temp.push_back(INT_MIN);
    func(v,0,temp);

    cout<<"Total longest sorted subsequences "<< res.size()<<endl;

    for(auto i:res){
        for(int j=1;j<i.size();j++){
            cout<<i[j]<<" ";
        }
        cout<<endl;
    }

}
```

INPUT: (given in code itself)

```
{3, 4, 1, 2, 5, 6, 7, 8, 1, 2, 3}
```

OUTPUT:

Total longest sorted subsequences 2
3 4 5 6 7 8
1 2 5 6 7 8

ALGORITHM:

Firstly , Used Dynamic programming(DP) to find the length of the longest increasing subsequence (LIS) by constructing a list of all subsequences with this length.
This approach has a runtime complexity of O(n^2) to find the length of the LIS using dynamic programming.
Generating all subsequences with the LIS length could potentially take O(2^n) time, where each element is part of the LIS (similar to taking one and not taking).
However, by efficiently generating subsequences while constructing the LIS, the overall runtime complexity can be reduced.
Therefore, the total runtime complexity would be dominated by the LIS calculation and getting the actual longest subsequences , O(2^n).

If any one of the sequence need to be printed , then in that case we just needed to use dp to find maximum length and then taking O(n) time to compute any one of the possible subsequences , thus in that case complexity should be order of (n^2) .
Here we have printed all possible sequence , there is no other way than the actual traversing and using the take-not take condition.

TIME COMPLEXITY : O(2^n)
SPACE COMPLEXITY : O(N)

Q2 .
CODE :

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


int bubble(vector<int> v){
    int n=v.size();
    int count =0 ;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (v[j] > v[j + 1]){count++; swap(v[j], v[j + 1]);}
        }
    }
    return count;
}

int insertion(vector<int> v){
    int n=v.size();
    int count=0;

    int k;
    for (int i = 1; i < n; i++) {
        k = v[i];
        int j = i - 1;
        while (j >= 0 && v[j] > k) {
            count++;
            v[j + 1] = v[j];
            j = j - 1;
        }
        v[j + 1] = k;
    }
    return count;
}

int selection(vector<int> v){
```

```
    int n=v.size();
    int count=0;

    for (int i = 0; i < n - 1; i++) {
        int k = i;
        for (int j = i + 1; j < n; j++) {
            if (v[j] < v[k])
                k = j;
        }
        if (k != i) {
            count++;
            swap(v[k], v[i]);
        }
    }
    return count;
}


int main(){
    vector<int > v;
    srand(time(0));
    for(int i=0;i<1000;i++){
        v.push_back(rand()%1000);
    }

    cout<<"Bubble sort : "<<bubble(v)<<endl;
    cout<<"Insertion sort : "<<insertion(v)<<endl;
    cout<<"Selection sort : "<<selection(v)<<endl;

}
```

INPUT : (already taking randomly)

OUTPUT:
Bubble sort : 248516
Insertion sort : 248516
Selection sort : 994

ALGORITHM:

Bubble Sort:
Bubble sort works by repeatedly swapping adjacent elements if they are in the wrong order.
In the worst-case scenario, when the array is in reverse sorted order, bubble sort will perform n*(n-1)/2 swaps, where n is the number of elements in the array.
For an array of 1000 elements, the worst-case number of swaps would be 1000*(1000-1)/2 = 499,500 swaps.
The runtime complexity of bubble sort is O(n^2) in the worst and average cases.

Insertion Sort:
Insertion sort builds the sorted array one element at a time by repeatedly moving the current element into its correct position relative to the already sorted elements.
The number of swaps depends on the initial order of the elements. In the worst-case scenario, when the array is in reverse sorted order, insertion sort would perform the same number of swaps as bubble sort.
Insertion sort can achieve best-case time complexity of O(n) when the array is already sorted
For an array of 1000 elements, the worst-case number of swaps would be 499,500 swaps.
The runtime complexity of insertion sort is also O(n^2) in the worst and average cases.
Selection Sort:
Selection sort works by repeatedly selecting the smallest (or largest, depending on the sorting order) unsorted element and moving it to its correct position.
The number of swaps in selection sort is fixed regardless of the input order. For each pass, only one swap is performed.
For an array of 1000 elements, there will be exactly 999 swaps in the worst-case scenario.
The runtime complexity of selection sort is O(n^2) in the worst and average cases.

OBSERVATION FROM OUTPUT:

- Bubble sort and insertion sort have the same worst-case number of swaps for an array of 1000 elements (499,500 swaps).
- Selection sort has a fixed number of swaps (999 swaps) regardless of the input order.
- All three algorithms have a worst-case runtime complexity of O(n^2), making them inefficient for large number of elements.

Space complexity : O(1) constant space (INPLACE SORTING)
Time complexity : O(n^2)

RUNTIME OF SORTING TECHNIQUE VS NUMBER OF ELEMENTS

	Bubble	Insertion	Selection
10000	141	35	54
20000	707	142	218
30000	1786	322	494
40000	3084	548	825
50000	4806	854	1292
60000	6997	1230	1860
70000	9835	1715	2650
80000	12920	2198	3303
90000	16135	2776	4178

Q3.
CODE:

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


int main(){
```

```
    vector<pair<int,int>> v;

    map<int,vector<pair<int,int>>> mapx;
    map<int,vector<pair<int,int>>> mapy;

    srand(time(0));

    for(int i=0;i<5000;i++){
        int a = rand()%1000;
        int b = rand()%1000;
        pair<int,int> p = {a,b};
        v.push_back(p);
        mapx[a].push_back(p);
        mapy[b].push_back(p);
    }

    for(auto i:mapx){
        cout<< "collinear with x = ";
        cout<<i.first<<" -> ";
        for(auto j:i.second){
            cout<<"{"<<j.first<< ","<<j.second<<"}  ";
        }
        cout<<endl;
    }

    cout<<endl<<endl;

    for(auto i:mapy){
        cout<< "collinear with y = ";
        cout<<i.first<<" -> ";
        for(auto j:i.second){
            cout<<"{"<<j.first<< ","<<j.second<<"}  ";

        }
        cout<<endl;
    }


}
```

INPUT: (already taking random values)

OUTPUT:
collinear with x = 0 -> {0,448} {0,482} {0,365} {0,346} {0,424} {0,715}
collinear with x = 1 -> {1,306} {1,672} {1,599} {1,961} {1,283} {1,883} {1,239} {1,414}
collinear with x = 2 -> {2,380} {2,148} {2,769} {2,955} {2,123} {2,155}
collinear with x = 3 -> {3,355} {3,225} {3,945} {3,751} {3,354} {3,92} {3,824} {3,734} {3,944}

collinear with x = 4 -> {4,322} {4,907} {4,602} {4,551} {4,379}
collinear with x = 5 -> {5,243} {5,283} {5,590} {5,126} {5,530} {5,553}
collinear with x = 6 -> {6,46} …
(large output can't be displayed here)

ALGORITHM:

Using standard template library(STL) object map in this algorithm.
Here as soon the random x ,y coordinates are created in the loop , by using map<int , vector<pair<int ,int>> , making a vector with respect to same x coordinates and similarly with y coordinates . inserting in the map take log(n) operation . and we are using it for all the number in the loop that is n numbers , thus time taken in this step is 2*n*log(n) , here twice because inserting in two map x and y map .
Thus , finally in the map , we have the collinear points having x same and in other map all having y coordinates same .
This finally printing the element of the map give the required output.

TIME COMPLEXITY: O(N*log(N)) map used within the loop.
SPACE COMPLEXITY : O(N) space occupied by the two maps and vector<pair< int, int > > .

Q4:
CODE:

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

int main(){
    int n;
    cin>>n;

    int a = 0 ;
    int b = 1 ;

    while(b <= n ){
        int c = a + b;
        a=b;
        b=c;
    }

    if(a==n || b==n){
        cout<<"YES , it is fibonacci ";
        return 0;

    }
    else{
        cout<<"NO , it is not fibonacci"<< endl;
        cout<<"The nearest fibonacci is ";
        if(abs(n-a) > abs(n-b)){
            cout<< b <<endl;
```

```
        }
        else{
            cout<< a <<endl;
        }
    }
}
```

INPUT :
36

OUTPUT:
NO , it is not fibonacci
The nearest fibonacci is 34

INPUT :
55

OUTPUT:
YES , it is Fibonacci

ALGORITHM:

1. The first two Fibonacci numbers are 0 and 1.
2. The subsequent Fibonacci numbers are obtained by adding the previous two numbers: (F(n) = F(n-1) + F(n-2)).

Now, let's explore the runtime complexities of different approaches to compute Fibonacci numbers:

1. Recursive Approach:
    - The recursive equation for Fibonacci is (F(n) = F(n-1) + F(n-2)).
    - The time taken to calculate (F(n)) is equal to the sum of the time taken to calculate (F(n-1)) and (F(n-2)).
    - This approach results in exponential time complexity.
    - Specifically, the running time is (O(2^n)) due to the branching nature of the recursive calls .
2. Iterative Approach (Can also be treated as part of dynamic programming):
    - The iterative approach computes Fibonacci numbers iteratively using a loop.
    - The running time of the iterative approach is (O(n)), where (n) is the number of elements in the sequence.

    Thus using the iterative approach is the most practical for computing Fibonacci numbers in linear time.

TIME COMPLEXITY : O(n) where n is the number of iteration taken to reach the input number.
This n is not at all similar to n of input . for large number of inputs also , the iteration inside the loop remain much less.

SPACE COMPLEXITY : O(1) constant space just using two variables in the whole code.

Q5 .

CODE:

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

bool check(int a , int b, int c){
    if(a+b <= c){
        return false;
    }
    return true;
}

void func(int a,int b, int c , string &s){
    string x= "";
    for(int i=0;i<a;i++){
        x.push_back(s[i]);
    }

    int p = stoi(x);
    x="";
    for(int i=a;i<b;i++){
        x.push_back(s[i]);
    }
    int q = stoi(x);
    x="";
    for(int i=b;i<c;i++){
        x.push_back(s[i]);
    }

    int r = stoi(x);
    cout<<p<< " "<<q<<" "<<r<<endl;
    vector<int> v;
    v.push_back(p);
    v.push_back(q);
    v.push_back(r);

    sort(v.begin(),v.end());

    p = v[0],q=v[1],r=v[2];

    if(check(p,q,r)==false){
        cout<<"The side length doesnot form any triangle"<<endl;
        cout<<endl;
        return;
    }
```

```
    if(p*p + q*q == r*r ){
        cout<<"They are a pythagorean triplet"<<endl<<endl;
        return;
    }

    cout<<"Angle in this traingle are ";

    double result = (p*p + q*q - r*r);
    double deno = 2*p*q;
    result = result/deno ;
    double max_angle_trianle = acos(result);
    max_angle_trianle = max_angle_trianle * 180 / 3.141592;
    double angle1 = max_angle_trianle;
    cout<<max_angle_trianle<<" ";


    result = (p*p + r*r - q*q);
    deno = 2*p*r;
    result = result/deno ;
    max_angle_trianle = acos(result);
    max_angle_trianle = max_angle_trianle * 180 / 3.141592;
    double angle2 = max_angle_trianle;
    cout<<max_angle_trianle<<" ";

    result = (q*q + r*r - p*p);
    deno = 2*q*r;
    result = result/deno ;
    max_angle_trianle = acos(result);
    max_angle_trianle = max_angle_trianle * 180 / 3.141592;
    double angle3 = max_angle_trianle;
    cout<<max_angle_trianle<<endl;

    if(abs(angle1-90)<=5 || abs(angle2-90)<=5 || abs(angle3-90)<=5){
        cout<<"It is nearly pythagorean "<<endl;
    }
    else if(p*p + q*q == r*r + 1){
        cout<<"It is nearly pythagorean "<<endl;
    }
    else{
        cout<<"It is not nearly pythoagorean"<<endl;
    }
    cout<<endl;

}

int main(){
    string s="";
    cin>>s;
```

```
    // 6 case
    // 1 3 1 -> 1 4 5
    // 3 1 1 -> 3 4 5
    // 1 1 3 -> 1 2 5
    // 2 2 1 -> 2 4 5
    // 1 2 2 -> 1 3 5
    // 2 1 2 -> 2 3 5

    // func(1,2,5,s);
    // func(1,3,5,s);
    // func(1,4,5,s);
    // func(2,4,5,s);
    // func(2,3,5,s);
    // func(3,4,5,s);

    for(int i=1;i<=3;i++){
        for(int j=i+1;j<=4;j++){
            func(i,j,5,s);
        }
    }


}
```

INPUT:
51213

OUTPUT:
5 121 3
The side length doesnot form any triangle

512 1 3
The side length doesnot form any triangle

5 1 213
The side length doesnot form any triangle

51 21 3
The side length doesnot form any triangle

5 12 13
They are a pythagorean triplet

51 2 13
The side length doesnot form any triangle

INPUT :
51113

OUPUT:
5 111 3
The side length doesnot form any triangle

511 1 3
The side length doesnot form any triangle

5 1 113
The side length doesnot form any triangle

51 11 3
The side length doesnot form any triangle

5 11 13
Angle in this traingle are 102.069 55.8378 22.0932
It is not nearly pythoagorean

51 1 13
The side length doesnot form any triangle

Nearly Pythagoras:
I have considered nearly Pythagoras triplet to be following either of the one conditions:

1. Any of the angle in triangle is lying between 85 degree to 95 degree .
2. $a^2 + b^2 = c^2 + 1$

ALGORITHMS:
Taking string as input , has made a function that has 4 parameter that make partition and then find the three length of triangle and then firstly checking if it is a triangle or not , and if it is then it is a right angles triangle or not , all of these operation are constant time just using the if else conditions.

Then if the triangle is not right angled then algorithm is finding all the angles of that triangle and checking checking for condition of angle between 85 and 95 and also checking condition of $a^2 + b^2 = c^2 + 1$. this is also constant time operations.

TIME COMPLEXITY : : O(1) all constant time operations. As there are only 5 digit number given .
If let say the length of string be s then complexity will be equal to all possible number of partition of three is O(nC2) nC2 is choosing as two partition are to be inserted among n places.
SPACE COMPLEXITY : O(1) constant space just using variables in the whole code.