Design And Analysis of Algorithms

Devam Desai- IIT2022035

Assignment

**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).**

```
// Devam Desai
// IIT2022035 DAA Assignment

void lisdp(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]);
                lisdp(result, temp, arr, index + 1);
                temp.pop_back();
        }
        lisdp(result, temp, arr, index + 1);
}
```

Time Complexity: O(n^2)
Space Complexity: O(n)

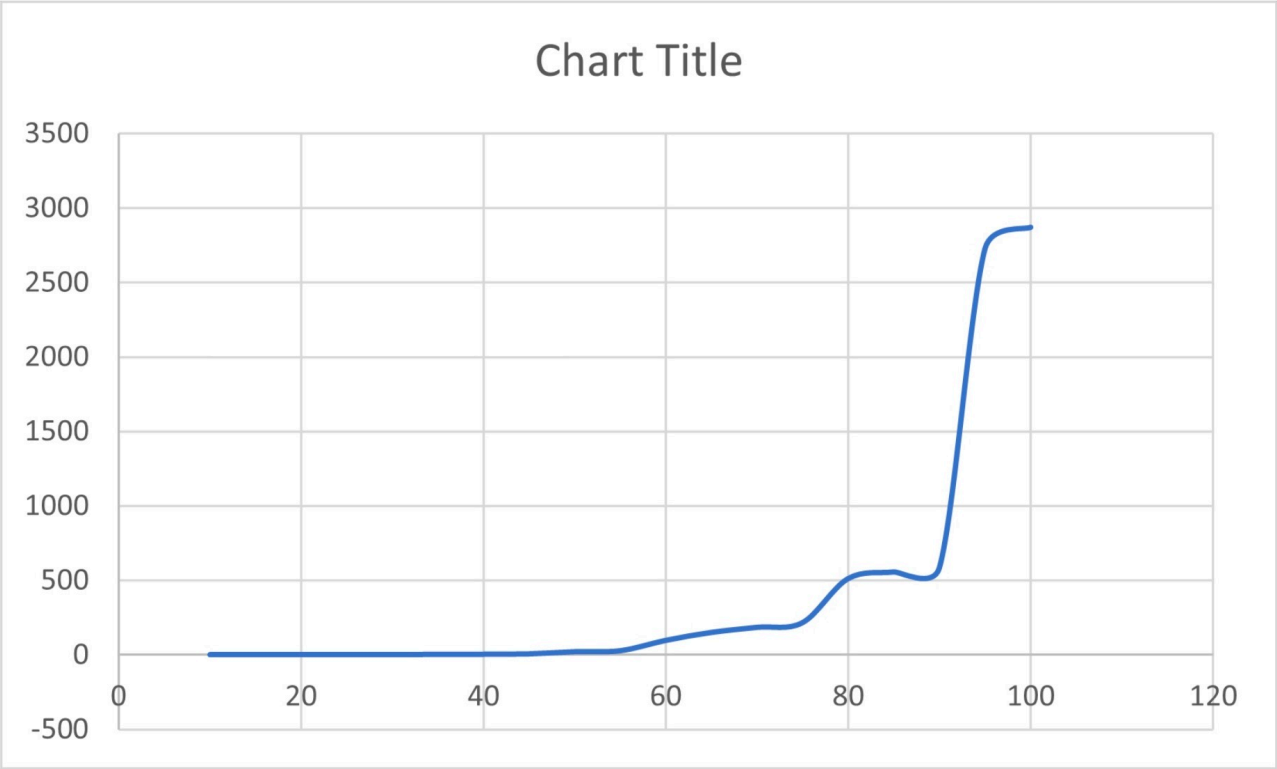
```
vector<vector<int>> solve(vector<int>& arr) {
        vector<vector<int>> result;
        vector<int> temp;
        lisdp(result, temp, arr, 0);
        int max_length = 0;
        for (auto& lis : result) {
                max_length = max(max_length, (int)(lis.size()));
        }
        vector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

Time Complexity: O(2^n)
Space Complexity: O(n^2)

```
int main(){
        vector<int> arr = {6,3,62,2,8,12,512,621};
        vector<vector<int>> LIS = solve(arr);
        printLIS(LIS);
        return 0;
}
```

Overall Time Complexity: O(2^n)
Overall Space Complexity: O(n^2)

Analysis:
Since this is a recursive solution, I have elected to omit the line by line complexity analysis.
In terms of time complexity, the algorithm for finding the Longest Increasing Subsequence (LIS) operates as follows:
The longest length of LIS is determined in O(n*n) time.
Subsequently, all possible valid LISs are identified through recursion, which takes O(2^lenOfLIS) time, simplifying to O(2^n) in the worst case.
The overall time complexity is then O(n*n) + O(2^n) = O(2^n).
Regarding space complexity, the algorithm utilizes an additional array of size n to store the LIS, and the solve() function consumes O(n*n) space. Thus, the overall space complexity is O(n*n*).
For a more detailed analysis, the lisdp() function calculates the longest possible length of the LIS for the given array in O(n*n) time. The solve() function then recursively explores all valid combinations of LIS generated from the array. Finally, it constructs the LIS deque through backtracking, reverses it, and returns the result.
To summarize, after determining the longest length of the LIS in O(n*n) time, the algorithm checks each possible valid combination for this longest length through recursive operations in O(2^n)



time.

Q2. Print the number of swapping operations while sorting an array of 1000 positive integers using

bubble sort, insertion sort, and selection sort.

Code	Complexity	Frequency
#include<bits/stdc++.h>		
using namespace std;		
int bubbleSort(vector<int> arr, int n)	1	1
{		
int ct=0;	1	1
int i, j;		
bool swapped;	1	1
for (i = 0; i < n - 1; i++) {	1	n
swapped = false;	1	n
for (j = 0; j < n - i - 1; j++) {	1	n^2
if (arr[j] > arr[j + 1]) {	1	n^2
swap(arr[j], arr[j + 1]);	1	n^2
ct++;	1	n^2
swapped = true;	1	n^2
}		
}		
if (swapped == false)	1	n
break;	1	1
}		
return ct;	1	1
}		
		=**5n^2+3n+5**

Overall Time Complexity: O(n^2), Omega(n^2), Theta(n^2)
Overall Space Complexity: O(n)

Code	Complexity	Frequency
int insertionSort(vector<int> arr, int n)	1	1
{		
int ct=0;	1	1
int i, key, j;		
for (i = 1; i < n; i++) {	1	n
key = arr[i];	1	n
j = i - 1;	1	n
while (j >= 0 && arr[j] > key) {	1	n^2
arr[j + 1] = arr[j];	1	n^2
j = j - 1;	1	n^2
}		
arr[j + 1] = key;	1	n
ct++;	1	n
}		
return ct;	1	1
}		
		=2n^2+4n+3

Overall Time Complexity: O(n^2), Omega(n)
Overall Space Complexity: O(n)

Code	Complexity	Frequency

int selectionSort(vector<int> arr, int n)	1	1
{		
int i, j, min_idx;	1	1
int ct=0;	1	1
for (i = 0; i < n - 1; i++) {	1	n
min_idx = i;	1	n
for (j = i + 1; j < n; j++) {	1	n^2
if (arr[j] < arr[min_idx])	1	n^2
min_idx = j;	1	n^2
}		
if (min_idx != i) {	1	n
swap(arr[min_idx], arr[i]);	1	n
ct++;	1	n
}		
}		
return ct;	1	1
}		

=3n^2+5n+4

Overall Time Complexity: O(n^2), Omega(n^2), Theta(n^2)
Overall Space Complexity: O(n)

Code	**Complexity**	**Frequency**
int main(){		
int n=1000;	1	1
vector<int> arr1,arr2,arr3;	1	1
int maxi = 1e5;	1	1
for (int i = 0;i < n;i++){	1	n
int val=rand()%(maxi);	1	n
arr1.push_back(val);	1	n
arr2.push_back(val);	1	n
arr3.push_back(val);	1	n
}		
int selsw=selectionSort(arr1,1000);	n^2	1
int inssw=insertionSort(arr2,1000);	n^2	1
int bubbsw=bubbleSort(arr3,1000);	n^2	1
cout<<selsw<<" "<<inssw<<" "<<bubbsw<<endl;	1	1
return 0 ;	1	1
}		

Final Complexity	**=3n^2+5n+5**
Final Time Complexity	**=O(n^2)**
Final Space Complexity	**=O(n)**

Analysis:

Selection Sort:

- In selection sort, for each iteration, the algorithm finds the minimum element in the unsorted part of the array and swaps it with the first element of the unsorted part.

- The number of swaps in selection sort is typically proportional to the number of iterations performed.
- The worst-case, average-case, and best-case time complexities are all O(n^2), where n is the number of elements in the array.

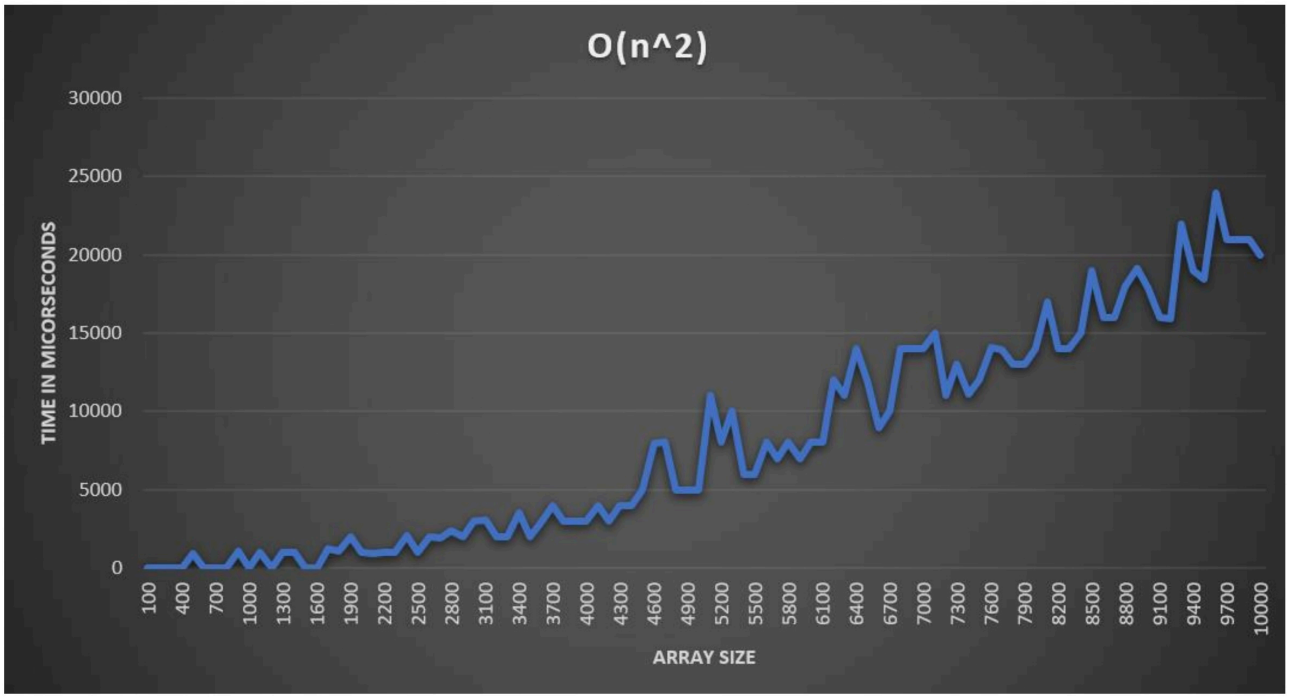
Insertion Sort:

- In insertion sort, the algorithm builds the sorted sequence one element at a time by repeatedly taking an element from the unsorted part and inserting it into its correct position in the sorted part.
- The number of swaps in insertion sort depends on how far each element needs to move to reach its correct position.
- The worst-case and average-case time complexities are O(n^2), and the best-case time complexity is O(n) when the array is nearly sorted.

Bubble Sort:

- In bubble sort, the algorithm repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
- The number of swaps in bubble sort depends on the number of inversions in the array (pairs of elements in reverse order).
- The worst-case, average-case, and best-case time complexities are all O(n^2).

In summary, the number of swaps for these sorting algorithms tends to be higher for arrays that are more unsorted. Insertion sort has a better best-case scenario compared to selection sort and bubble sort, but all three algorithms have quadratic time complexities in the worst and average cases. Keep in mind that these complexities are theoretical, and actual performance may vary based on implementation details and specific cases.



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.

//IIT2022035 Devam Desai
//DAA Assignment

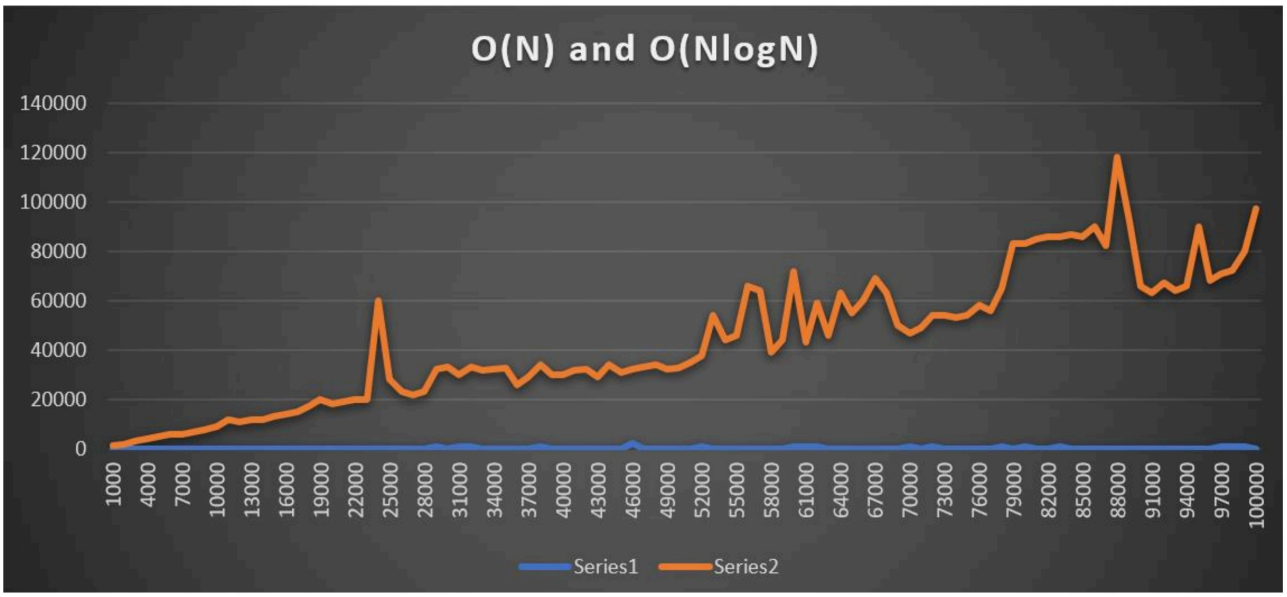
Code	Complexity	Frequency
void collinearPoints(vector<pair<int,int>> &points){	1	1
map<int,unordered_set<int>> xs,ys;	1	1
for(auto p: points){	1	n
int xi=p.first;	1	n
int yi=p.second;	1	n
xs[xi].insert(yi);	(logn)	n
ys[yi].insert(xi);	(logn)	n
}		
// resulting map contains all the points		
for(auto yi:ys){	1	n
if(yi.second.size()<2) continue;	1	n
for(auto x:yi.second){	1	n
cout<<x<<" "<<yi.first<<endl;	1	n
}		
}		
for(auto xi:xs){	1	n
if(xi.second.size()<2) continue;	1	n
for(auto y:xi.second){	1	n
cout<<y<<" "<<xi.first<<endl;	1	n
}		
}		
cout<<"COMPLETE"<<endl;	1	1
}		

Final Time Complexity **2n*(logn)+11n+3**

=O(nlogn)
=Omega(nlogn)
=Theta(nlogn)

Final Space Complexity= O(n)

Analysis:
The above code stores the x and y axis in the coordinates in a map of integer and unordered set to remove duplicate points.
Insertion in map is logn operation and in unordered set is O(1). Thus the overall Complexity is O(nlogn) and space complexity is O(n) as the points are stored in 2 different maps.



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

Code	Complexity	Frequency

bool isPerfectSquare(int x)
{
  int s = sqrt(x);
  return (s * s == x);
}		

bool isFibonacci(int n)
{
  return isPerfectSquare(5 * n * n + 4) ||
  isPerfectSquare(5 * n * n - 4);
}
	1
	logn
		1
		2

	2logn +1	
Time Complexity=O(logn) **Space Complexity: O(1)**		

void nearestFibonacci(int num)
{
  if (num == 0) {
  cout << 0;
  return;
  }
  int first = 0, second = 1;
  int third = first + second;
  while (third <= num) {
  first = second;
  second = third;
  third = first + second;
  }
  int ans = ((abs(third - num)>=
  abs(second - num)))? second: third;
  cout << ans;
}
	1
	1
	2
	1
	1
	1
	1
	1
	1
		1
		1
		1
		1
		logn
		logn
		logn
		1
		1

Final Complexity	**3logn+7**	
=O(logn) **=Omega(logn)** **Space Complexity: O(1)**		

Explanation:
This code consists of three functions: `isPerfectSquare`, `isFibonacci`, and `nearestFibonacci`.

isPerfectSquare(int x):

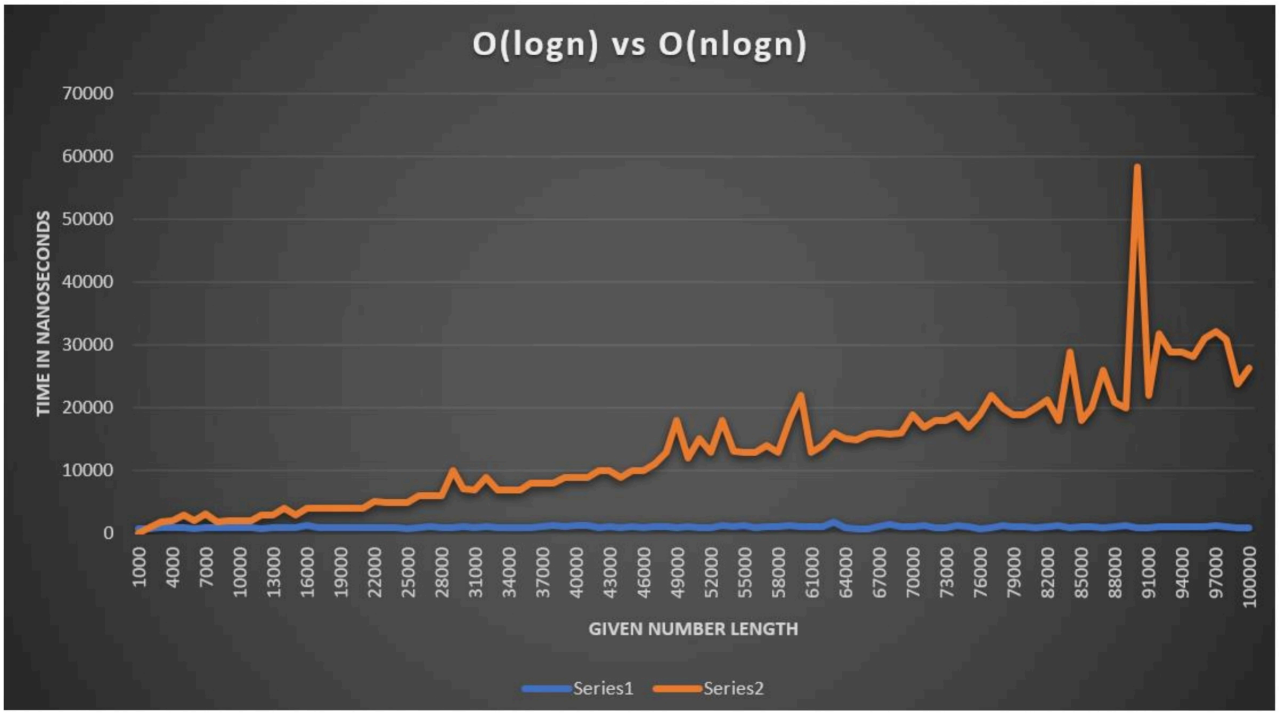
- The function calculates the square root of the input x using the sqrt function.
- The time complexity of calculating the square root is typically O(log(n)).
- The rest of the operations (multiplication and comparison) are constant time.
- Therefore, the overall time complexity is O(log(n)).

isFibonacci(int n):

- This function calls `isPerfectSquare` twice and performs some basic arithmetic operations.
- The time complexity is dominated by the calls to `isPerfectSquare`.
- Therefore, the overall time complexity is O(log(n)).

nearestFibonacci(int num):

- This function calculates the nearest Fibonacci number to the input num.
- It uses a while loop to find the Fibonacci numbers until it exceeds num.
- The loop runs until the Fibonacci numbers become greater than num, so the number of iterations is determined by the Fibonacci sequence.
- The Fibonacci sequence has an exponential growth rate, but the loop will terminate before reaching the actual Fibonacci number greater than num.
- Therefore, the time complexity of this function is O(log(num)).



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.

Code **Complexity** **Frequency**

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

```
bool isPythagorean(int a, int b, int c) {                    1          1
    int d=max(a,max(b,c));                                   1          1
    if(d==a) return (b*b + c*c == a*a);                      1          1
    if(d==b) return (c*c + a*a == b*b);                      1          1
    if(d==c) return (a*a + b*b == c*c);                      1          1

}
```

=5

=O(1)

Code	**Complexity**	**Frequency**
bool isNearlyPythagorean(int a, int b, int c) {	1	1
int d=max(a,max(b,c));	1	1
if(d==a) return abs(b*b + c*c - a*a) <=1;	1	1
if(d==b) return abs(c*c + a*a - b*b) <=1;	1	1
if(d==c) return abs(a*a + b*b - c*c) <=1;	1	1
}		

=5

=O(1)

Code	**Complexity**	**Frequency**
int main(){		
int nums=81517;	1	1
bool flag=false;	1	1
for (int i = 1; i <= 3; i++) {	1	1
for (int j = 1; j <= 3; j++) {	1	1
int a = nums / int(pow(10, 5 - i));	1	1
int b = (nums / int(pow(10, 5 - i - j))) % (int(pow(10,j)));	1	1
int c = nums % int(pow(10, 5 - i - j));	1	1
if(a==0 || b==0 || c==0) continue;	1	1
cout<<"The Partition Formed Is"<<endl;	1	1
cout << a << " " << b << " " << c << endl;	1	1
if (isPythagorean(a, b, c)) {	5	1
cout << "Pythagorean triplet found: "	1	1
<< a << ", " << b << ", " << c << endl;	1	1
cout<<endl;	1	1
flag=true;	1	1
}		
else if (isNearlyPythagorean(a, b, c)) {	5	1
cout << "Nearly Pythagorean triplet found: "	1	1
<< a << ", " << b << ", " << c << endl;	1	1
cout<<endl;	1	1
flag=true;	1	1
}		
}		
}		
if(!flag) cout << "No Pythagorean OR Nearly Pythagorean triplets found" << endl;	1	1

return 0;	1	1
}		

Final Complexity **=30**

=O(1)
=Omega(1)
Space Complexity: O(1)

Time Complexity:
The time complexity of the code is determined by the nested loops, each running from 1 to 3. Therefore, the time complexity is O(30) or O(1), which is a constant time complexity.
Logic:

- The code uses two functions isPythagorean and isNearlyPythagorean to check if a triplet is a Pythagorean triplet or a Nearly Pythagorean triplet, respectively.
- The main function uses nested loops to iterate through all possible partitions of the given number nums.
- It extracts the three integers (a, b, c) from each partition and checks if it forms a Pythagorean or Nearly Pythagorean triplet using the defined functions.
- If a triplet is found, it prints the triplet type and sets the flag variable to true.
- If no triplets are found, it prints a message indicating that no Pythagorean or Nearly Pythagorean triplets are present.