Question-1

**1st iteration:**

1st step: **7 2** 13 2 11 4 -> 2 7 13 2 11 4

2nd step: 2 **7 13** 2 11 4 -> 2 7 13 211 4

3rd step: 2 7 **13 2** 11 4 -> 2 7 2 13 11 4

4th step: 2 7 2 **13 11** 4 -> 2 7 2 11 13 4

5th Step: 2 7 2 11 **13 4** -> 2 7 2 11 4 13

**2nd iteration:**

1st step: **2 7** 2 11 4 13 -> 2 7 2 11 4 13

2nd step: 2 **7 2** 11 4 13 -> 2 2 7 11 4 13

3rd step: 2 2 **7 11** 4 13 -> 2 2 7 11 4 13

4th step: 2 2 7 **11 4** 13 -> 2 2 7 4 11 13

5th Step: 2 2 7 4 **11 13** -> 2 2 7 4 11 13

**3rd iteration:**

1st step: **2 2** 7 4 11 13 -> 2 2 7 4 11 13

2nd step: 2 **2 7** 4 11 13 -> 2 2 7 4 11 13

3rd step: 2 2 **7 4** 11 13 -> 2 2 4 7 11 13

4th step: 2 2 4 **7 11** 13 -> 2 2 4 7 11 13

5th Step: 2 2 4 7 **11 13** ->2 2 4 7 11 13

**Since the array is already sorted but the algorithm doesn't know if it's done yet. So the algorithm needs a full iteration without swapping to know it's sorted.**

**4th iteration:**

1st step: **2 2** 4 7 11 13-> 2 2 4 7 11 13

2nd step: 2 **2 4** 7 11 13 -> 2 2 4 7 11 13

3rd step: 2 2 **4 7** 11 13 -> 2 2 4 7 11 13

4th step: 2 2 4 **7 11** 13 -> 2 2 4 7 11 13

5th Step: 2 2 4 7 **11 13 ->** 2 2 4 7 11 13

Question-2

An **array** is a collection of items stored in contiguous memory locations This is storing multiple items of the same type together. This makes it easy to access the elements stored in each element by location. On the other hand, **vectors** are known as dynamic arrays that have the ability to automatically resize when an element is inserted or deleted, their storage is automatically managed by the container itself.

The differences between vector and array are given below:

|  |  |
| --- | --- |
| **Vector** | **Array** |
| A vector is a sequential container for storing elements and is not index oriented. | Arrays store a fixed-size sequential collection of elements of the same type and are index oriented. |
| Vectors are dynamic in nature so the size increases with the insertion of elements. | Since arrays are fixed size, they cannot be resized once initialized. |

Question-3

**Condition inner loop loops total loop Loops**

i=1 , j=1 to n, ----🡪 False 0 1

i=2,j=1 to n; ----🡪 False 0 2

i=3,j=1 to n;-----🡪 True n 3\*5

i=4,j=1 to n;-----🡪 False 0 4

i=5,j=1 to n;-----🡪 True n 5\*5

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

i=n,j=1 to n;-----🡪 if False 0 n

i=n,j=1 to n;-----🡪 if True n n\*n

That is, if the condition is true for the outer n loop, then the inner loop will loop n times.

The worst case is (n\*n) or n2.

The time complexity with proper explanation of the following code segment is **O(n2)**.

Question-4

Yes. There are two flaws in this code. Here the for(int i=0 ; **i<=n** ; i++) loop has flaws in the **i<=n** conditionand **if(a[i]!=a[i-1])** conditions. Because when i=0 then a[i-1]=a[0-1]=a[-1]. But from location a[0] in memory we will start giving the input value. That is, we will start getting the first value from the place where the input value is at a[0]. So a[-1] location will contain a **garbage** value and since the a[0]=a[-1] condition is true, after sorting the inputs that **garbage** value will be compared with the zeroth (0) position or first value of the vector and the value of ans 1 will increase. Then for the condition i<=n the value of ans will be incremented for as many distinct values as there are in the input. That is, if distinct value is 4 four, then value of ans will show **5**. That is, five distinct values will be shown. Which is the **wrong** answer.

Question-5

**Time complexity for the external & internal for loop :**

**Let n=5**

**inner loop loops total loop Loops**

i=1 , j=1, ----🡪 5(n) 1\*n

i=2,j=2 ; ----🡪 2 2\*n

i=3,j=3 to n;-----🡪 1 3\*1

i=4,j=1 to n;-----🡪 1 n\*1

i=5,j=5;-----🡪 1 n\*1

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

i=n,j=1 to n;-----🡪 1 n\*1 = n

The worst case of time complexity for the external & internal for loopis =(n\*n)= **O(n2)**.

Space Complexity :

vector<int>d[n+1]; it is a two dimensional array.

If we need to create an array of size n, this will take O(n) space. If we need to build a two-dimensional array of size n\*n, this will need O(n2) space.

So, The space Complexity = O((n+1)\*(n+1))

= O(n\*n+n+n+1)

= O(n\*n)

= **O(n2)**.

Total space complexity of the code = **O(n2)**.

Question-6

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Accessibility from own class | Accessibility from derived class | Accessibility from world |
| Public | **YES** | **YES** | **YES** |
| Private | **YES** | **NO** | **NO** |
| Protected | **YES** | **YES** | **NO** |

Question-7

**New**: A **new** is an operator in C++ that indicates a request to allocate memory on the heap. If enough memory is available, it initializes the memory and returns its address in the pointer variable.

The new operator should only be used if the data object remains in memory until it is deleted. Otherwise, if no new operator is used in the program, the object is automatically destroyed when it goes out of scope. In other words, newly used objects are cleaned up manually while other objects are cleaned up automatically when they go out of scope.

The syntax of the new operator is as follows:

**pointer\_variable = new DataType;**

**Delete**: A **delete** is an operator in C++ which is used to deallocate memory space that is created dynamically during the execution of a program in the C++ language using the new operator, callc and malloc() functions etc. Moreover, a delete operator is used to delete arrays and non-array (pointer) objects from the heap, which in turn are dynamically allocated by the new operator to keep variables in the heap memory. We can use delete operator or delete [ ] operator in our program to delete deallocated space. A deletion operator has a void return type, and therefore, it does not return a value.

Syntax of delete operator:

**delete pointer\_variable;**

**delete [ ] pointer\_variable;**

Question-8

**Yes**. I agree with Bob that Alice’s computer will take years to finish in the worst case.

**Provident:**

Here,

N=106

Since Alice’s algorithm works in O(n3)

Total Number of Operation = (106)3

= 1018

Given,

Alice’s computer can perform 109 operations in 1 second

Alice’s computer can perform 1 operation in seconds

Alice’s computer can perform 1018 operations in seconds

= 109 seconds.

we know that,

3.15360000×107 seconds = 1 year

1 second = years

109 seconds = years

= 31.70979198 years

= 31.71 years (approximately).

Therefore, Alice’s computer will take approximately 31.71 years to complete the 1018 operations in the worst case that is **O(n3).**

**(Proved).**

Question-9

A **linear search** is also known as a sequential search which scans each element one at a time. Suppose we want to search for an element in an array or list; We simply calculate its length and do not jump to any item. On the other hand, a **binary search** is a search in which the intermediate element is calculated to check if it is smaller or larger than the element being searched for. The main advantage of using binary search is that it does not scan every element of the list. Instead of scanning each element, it searches half of the list.

The differences between linear Search and binary search are given below:

|  |  |
| --- | --- |
| **Linear Search** | **Binary Search** |
| In a linear search, elements do not have to be sorted in sorted order. | The pre-condition of binary search is that the elements must be sorted in a sorted order. |
| In a linear search, the worst case is O(n) to find the element. | In a binary search, the worst case to find the element is O(logn). |

Question-10

Yes. There are a flaw in the function.Because in the code mentioned in the question, the memory is not **deallocated** after it is dynamically allocated. As a result, the possibility of **overflow** remains.

Dynamic memory allocation is done when extra memory is needed somewhere in the code. After storing the address value of the memory address that is dynamically allocated, the allocated memory is usually deallocated to avoid overflow and memory wastage.

**The modified code:**

void func()

{

int\* p = new int;

delete p;

return ;

}