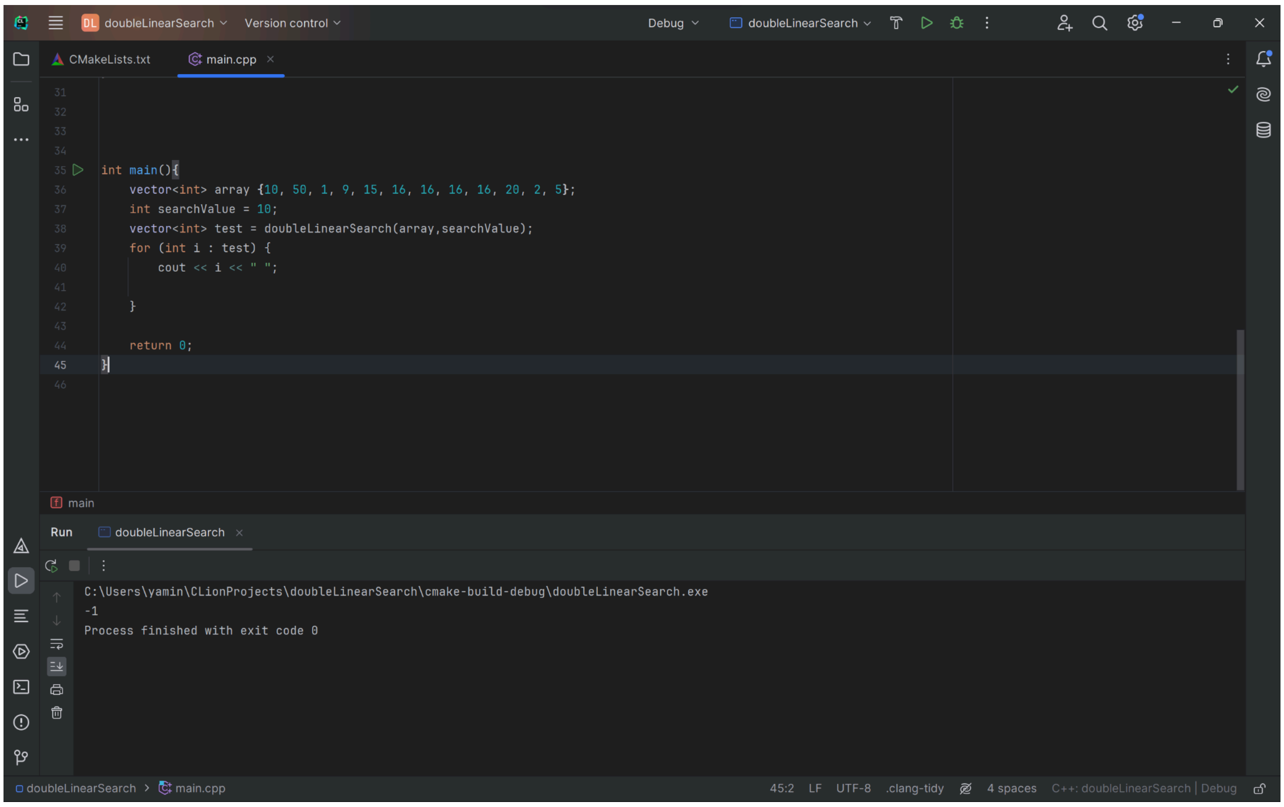
Deliverable 1:
Pseudo code:

```
Function doubleLinearSearch(array, searchValue){
    //need a vector array to store the first two elements
    Initialize an empty vector variable named search_findings
    //need a variable to store the number of elements found
    Initialize a int variable named index_found to 0
    //need to loop through over each element. Stop the loop when 2 elements are found.
And add these 2 elements to the search_findings vector.
    //need a if statement where if 2 elements are not found, it will add -1 to
search_findings.
    Initialize a loop to iterate over each elements, index pair in the array.
        If the current element value is equal to search value
            Add the index of the current element to search_findings.
            And increment the index found by 1.
            If index_found = 2
            Break out of the loop.
        If index_found is less than 2:
        Clear the search_findings array and
        Add -1 to search_finding
    Return search_findings.
}
```

Deliverable 2:

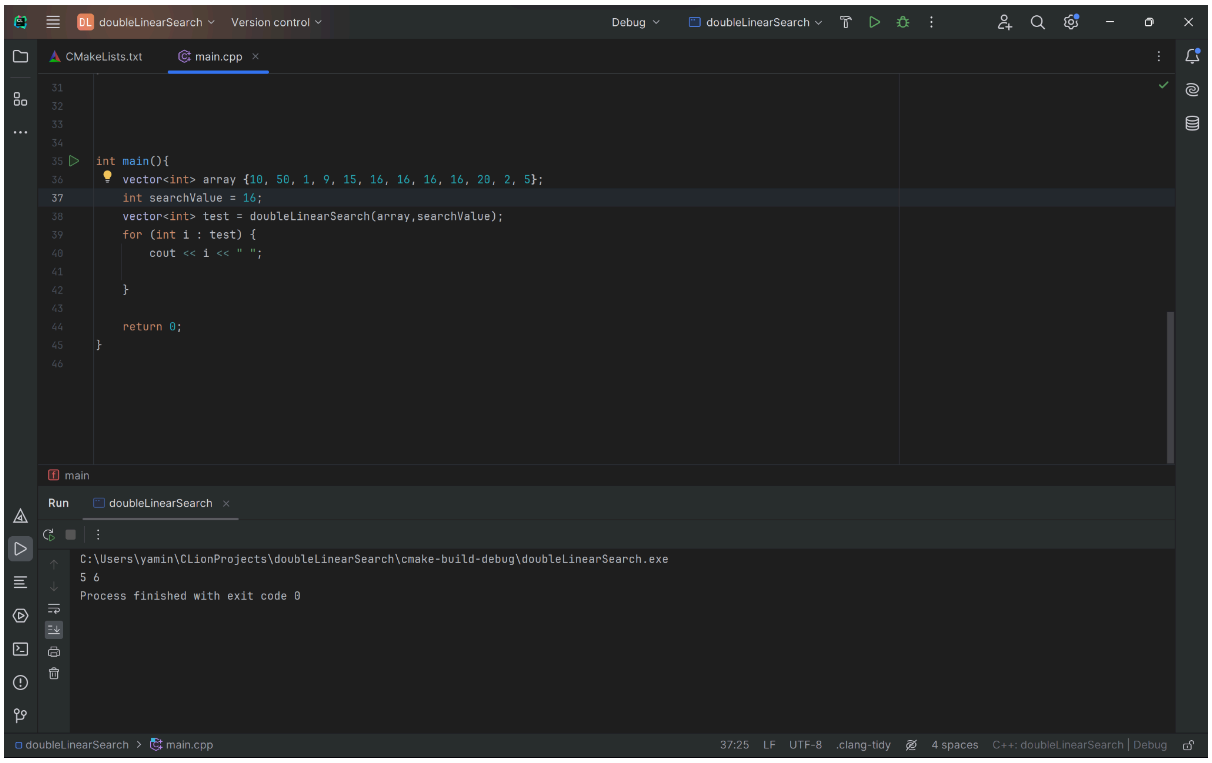


```cpp
int main(){
    vector<int> array {10, 50, 1, 9, 15, 16, 16, 16, 16, 20, 2, 5};
    int searchValue = 10;
    vector<int> test = doubleLinearSearch(array,searchValue);
    for (int i : test) {
        cout << i << " ";

    }

    return 0;
}
```

```
C:\Users\yamin\CLionProjects\doubleLinearSearch\cmake-build-debug\doubleLinearSearch.exe
-1
Process finished with exit code 0
```

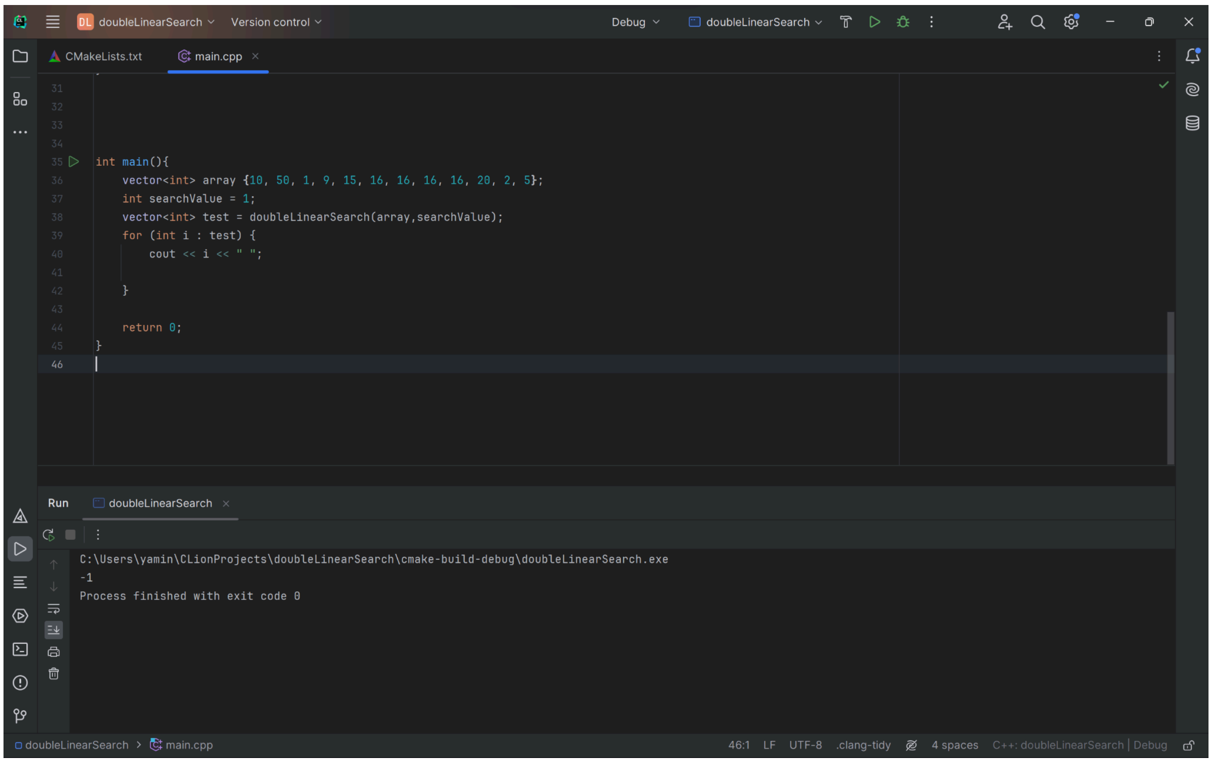


```cpp
int main(){
    vector<int> array {10, 50, 1, 9, 15, 16, 16, 16, 16, 20, 2, 5};
    int searchValue = 16;
    vector<int> test = doubleLinearSearch(array,searchValue);
    for (int i : test) {
        cout << i << " ";

    }

    return 0;
}
```

```
C:\Users\yamin\CLionProjects\doubleLinearSearch\cmake-build-debug\doubleLinearSearch.exe
5 6
Process finished with exit code 0
```



```cpp
int main(){
    vector<int> array {10, 50, 1, 9, 15, 16, 16, 16, 16, 20, 2, 5};
    int searchValue = 1;
    vector<int> test = doubleLinearSearch(array,searchValue);
    for (int i : test) {
        cout << i << " ";

    }

    return 0;
}
```

```
C:\Users\yamin\CLionProjects\doubleLinearSearch\cmake-build-debug\doubleLinearSearch.exe
-1
Process finished with exit code 0
```

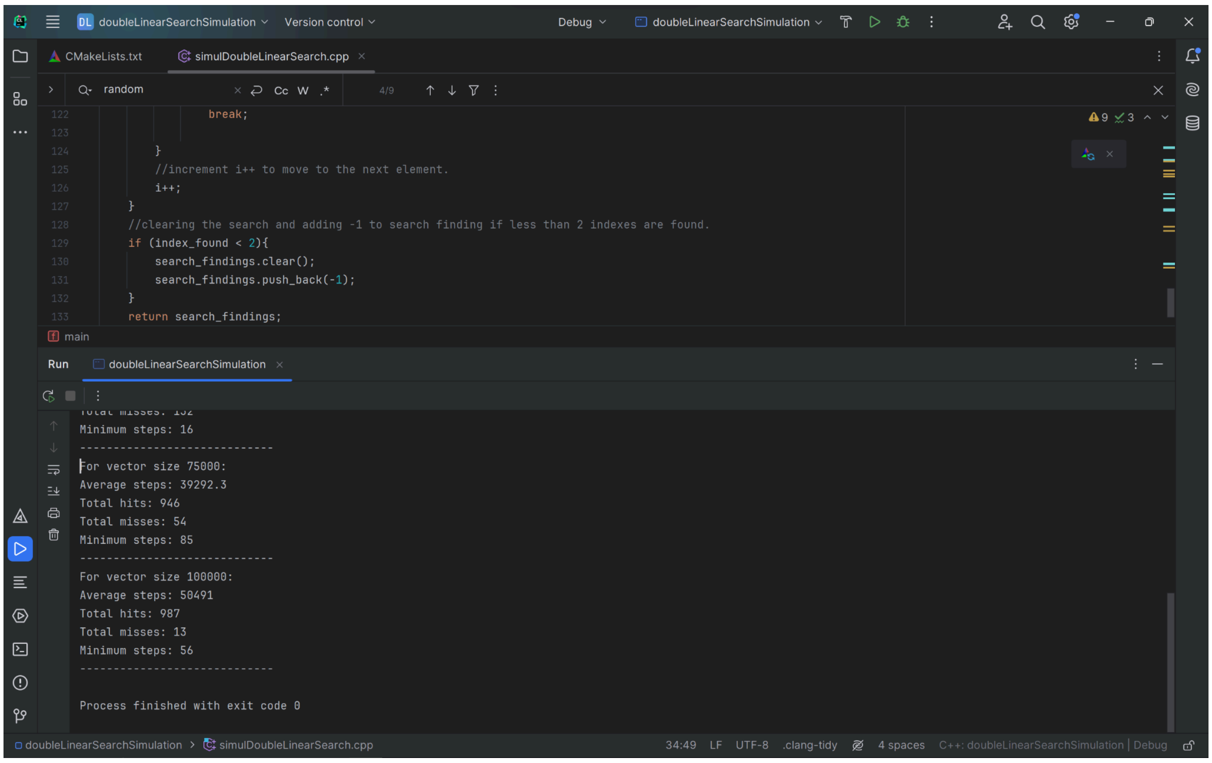
Deliverable 3:

1. Creating an empty vector takes constant time O(1).
2. Inititalizing a variable takes constant time O(1).
3. The while loop iterates through each element of the array once, so its complexity depends on the size of the input array, denoted by n. The worst case scenario is that

the search value was not found or only one value was found. In this case, the while loop runs through all elements of the array, giving it a complexity of O(N).

4. Inside the loop, the operation of checking whether the array element is equal to the search value takes constant time O(1)
5. Pushing an element into search_findings, incrementation of index_found, checking the number of index found is equal to 2 or not also takes constant time O(1).
6. Finally clearing the search findings vector takes linear timeO(n), where n is the number of elements in the vector. However, only max 2 elements will be cleared. It can be considered constant time O(1)
   So, the overall time complexity: T(n) = O(n)

Deliverable 4:



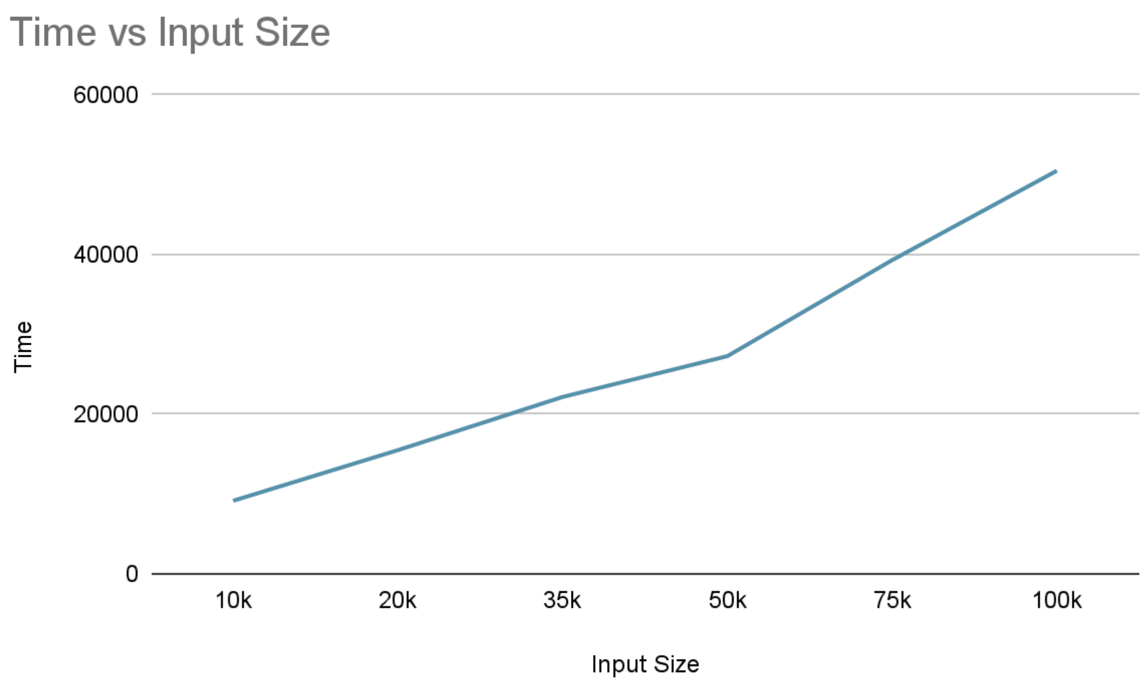
```
                break;

            }
            //increment i++ to move to the next element.
            i++;
        }
        //clearing the search and adding -1 to search finding if less than 2 indexes are found.
        if (index_found < 2){
            search_findings.clear();
            search_findings.push_back(-1);
        }
        return search_findings;
```

```
Total misses: 132
Minimum steps: 16
----------------------------
For vector size 75000:
Average steps: 39292.3
Total hits: 946
Total misses: 54
Minimum steps: 85
----------------------------
For vector size 100000:
Average steps: 50491
Total hits: 987
Total misses: 13
Minimum steps: 56
----------------------------

Process finished with exit code 0
```

Input Size	Hits	Misses	Min Steps	Average Steps
10k	144	856	13	9151.1
20k	400	600	19	15492.7
35k	697	303	5	22156.4
50k	868	132	16	27268.1
75k	946	54	85	39292.3
100k	987	13	56	50491



Deliverable 5:
The graph depicting the relationship between input size and time shows a pattern closely resembling a linear relationship. As the input size increases, the time taken also increases. This alignment with a linear trend suggests that the algorithm's time complexity is consistent with O(n), where n represents the input size.

In Big O notation, O(n) signifies a linear time complexity, indicating that the algorithm's execution time grows linearly with the size of the input. This means that if the input size doubles, the time taken for the algorithm to execute also approximately doubles.

The close match between the observed graph and the theoretical expectation of O(n) time complexity implies that the algorithm's performance scales at O(n).