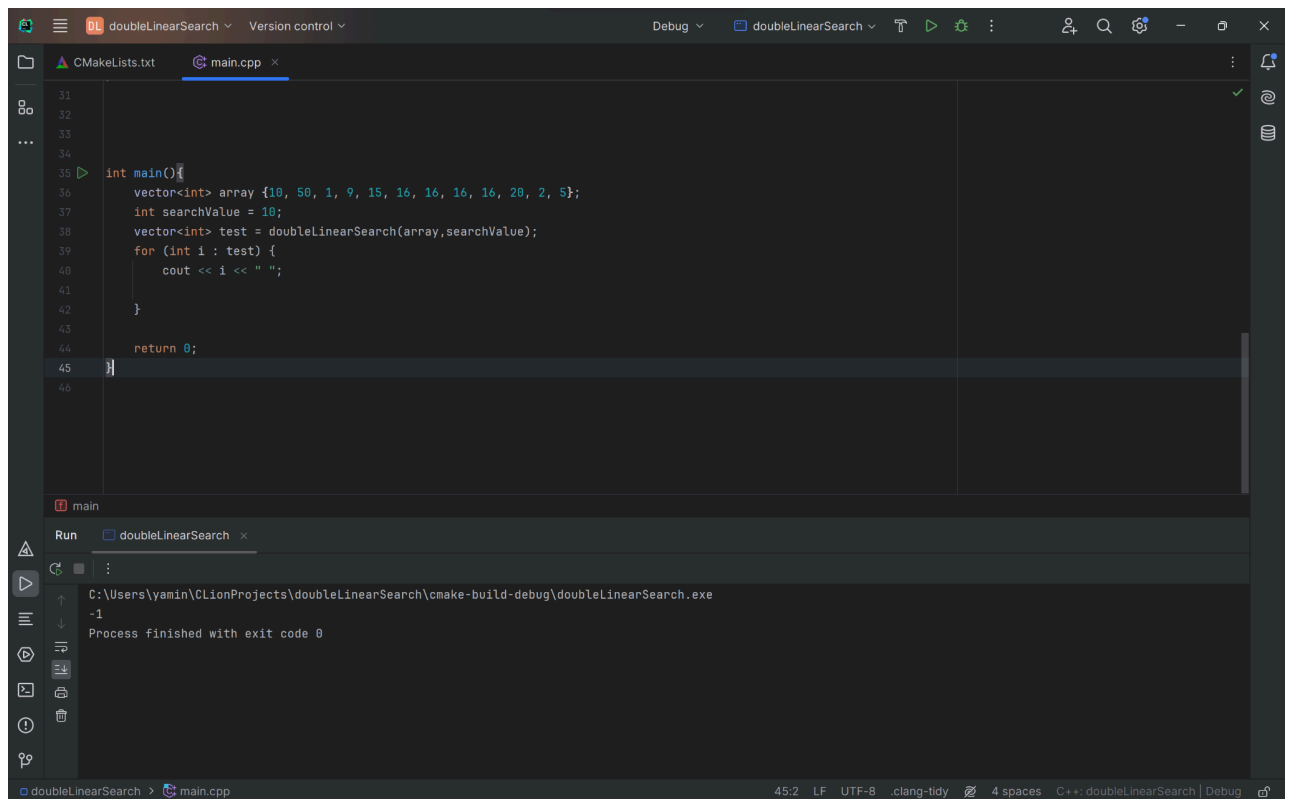


Deliverable 1:

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:



The screenshot shows a C++ IDE with a file named `main.cpp` open. The code defines a `doubleLinearSearch` function that takes a `vector<int>` and an `int` search value. It uses a loop to find the first two occurrences of the search value, storing their indices in a `vector<int>` named `search_findings`. If two elements are found, it returns the vector; otherwise, it returns a vector containing -1. The `main` function tests this with an array `{10, 50, 1, 9, 15, 16, 16, 16, 16, 20, 2, 5}` and a search value of 10. The output window shows the execution path and the result `-1`.

```
31
32
33
34
35 int main(){
36     vector<int> array {10, 50, 1, 9, 15, 16, 16, 16, 16, 20, 2, 5};
37     int searchValue = 10;
38     vector<int> test = doubleLinearSearch(array, searchValue);
39     for (int i : test) {
40         cout << i << " ";
41     }
42
43
44     return 0;
45 }
```

Run doubleLinearSearch

C:\Users\yamin\CLionProjects\doubleLinearSearch\cmake-build-debug\doubleLinearSearch.exe

-1

Process finished with exit code 0

```
31
32
33
34
35 int main(){
36     vector<int> array {10, 50, 1, 9, 15, 16, 16, 16, 16, 20, 2, 5};
37     int searchValue = 16;
38     vector<int> test = doubleLinearSearch(array, searchValue);
39     for (int i : test) {
40         cout << i << " ";
41     }
42
43
44     return 0;
45 }
46
```

Run doubleLinearSearch

C:\Users\yamin\CLionProjects\doubleLinearSearch\cmake-build-debug\doubleLinearSearch.exe

5 6

Process finished with exit code 0

```
31
32
33
34
35 int main(){
36     vector<int> array {10, 50, 1, 9, 15, 16, 16, 16, 16, 20, 2, 5};
37     int searchValue = 1;
38     vector<int> test = doubleLinearSearch(array, searchValue);
39     for (int i : test) {
40         cout << i << " ";
41     }
42
43
44     return 0;
45 }
46
```

Run doubleLinearSearch

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. Initializing 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 time $O(n)$, 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:

The screenshot shows a C++ IDE with a file named `simulDoubleLinearSearch.cpp`. The code implements a function `doubleLinearSearch` that takes a vector of integers, a search value, and a reference to a counter for steps. It iterates through the vector, counting steps for each element checked. If the search value is found, it pushes the index to a `searchFindings` vector. If two indices are found, it breaks the loop. The `main` function runs the simulation for two input sizes: 75000 and 100000. The output shows the average number of steps, hits, misses, and minimum steps for each input size.

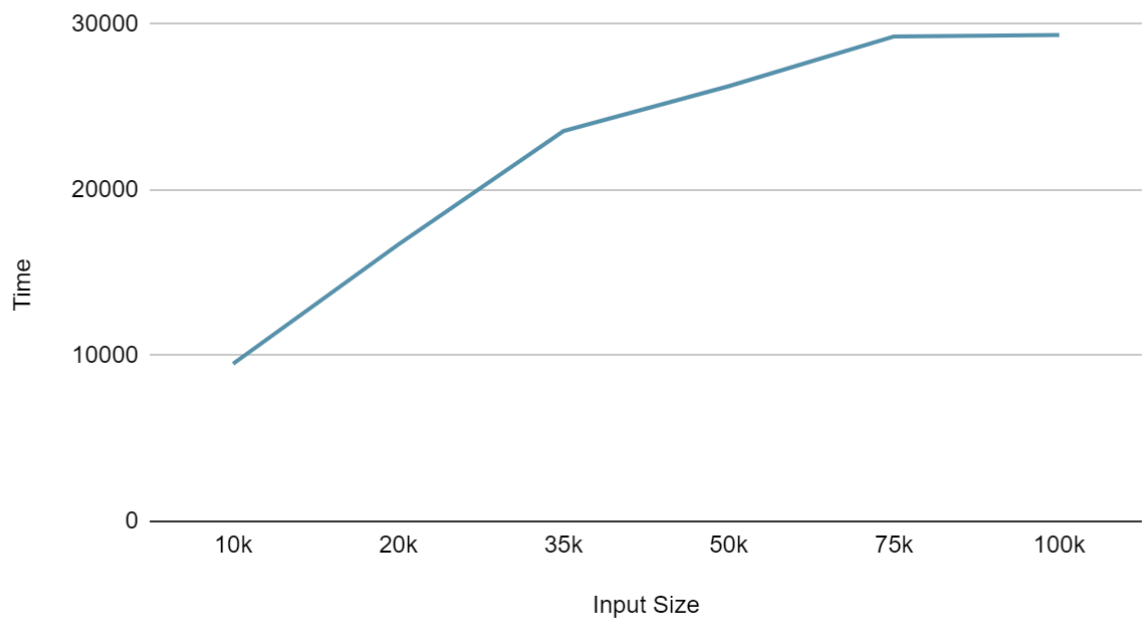
```
111 }
112
113 //takes a vector and an int and returns two indexes max if found. Otherwise returns -1
114 vector<int> doubleLinearSearch(const vector<int> &vec, int searchValue, int &countSteps){
115     int vecSize;
116     vecSize = vec.size();
117     countSteps = 0;
118     vector<int> searchFindings;
119
120     for (int i = 0; i < vecSize; ++i)
121     {
122         countSteps++; //Incrementing steps taken to find index in the beginning of every loop.
123         if (vec.at(i) == searchValue)
124         {
125             searchFindings.push_back(i);
126         }
127         else if (searchFindings.size() == 2)
128         {
129             break; // when the second index is found, the loop is broken.
130         }
131     }
132     return searchFindings;
133 }
```

Run doubleLinearSearchSimulation

```
Input Size: 75000
Average number of steps taken: 29192.6
Hits: 955
Misses: 45
Minimum Steps: 176
-----
Input Size: 100000
Average number of steps taken: 30421.8
Hits: 989
Misses: 11
Minimum Steps: 486
-----
```

Input Size	Hits	Misses	Min Steps	Average Steps
10k	147	853	356	9508.39
20k	370	630	621	16708.6
35k	702	298	230	23564.9
50k	844	156	948	26260.8
75k	955	45	176	29269.6
100k	989	11	486	29347.5

Time vs Input Size



Deliverable 5:

The graph illustrating the correlation between input size and time exhibits a behavior similar to logarithmic growth. With an increase in input size, the corresponding time taken also increases, but at a slower rate. This resemblance to a logarithmic trend suggests that the algorithm's time complexity adheres to $O(\log n)$, where n denotes the input size.

In the realm of Big O notation, $O(\log n)$ denotes logarithmic time complexity, indicating that the algorithm's runtime grows logarithmically with the size of the input. Consequently, if the input size doubles, the time required for the algorithm's execution increases only logarithmically.

The notable similarity between the depicted graph and the expected behavior of $O(\log n)$ time complexity implies that the algorithm's efficiency conforms to $O(\log n)$.