106119029 Lab 2

Question 1

Write a program to implement BSearch. Find the time required for Best case, worst case and a random case.

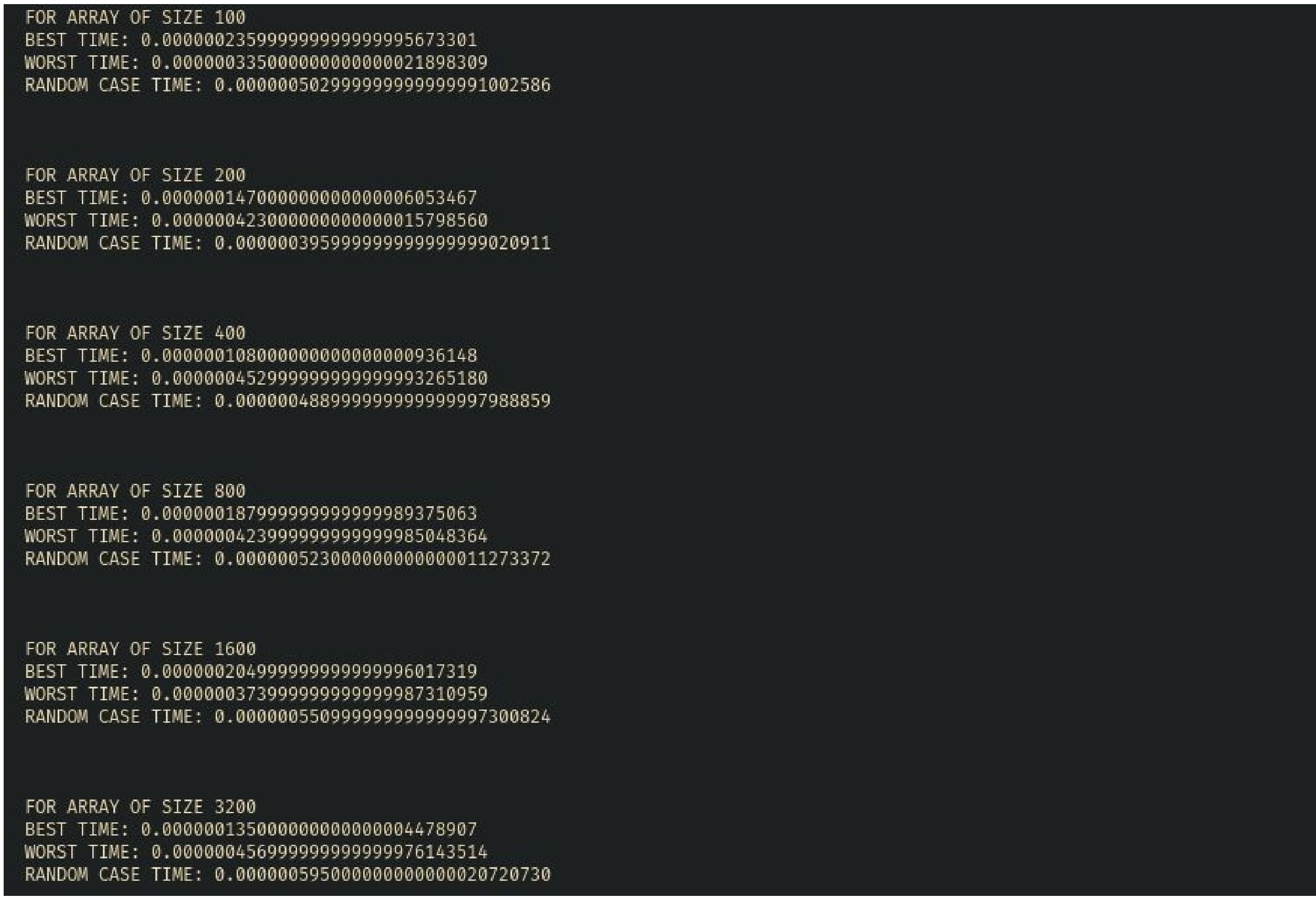
Code

Created arrays of different sizes 100,200,400,800,1600,3200,6400,12800,25600,51200,102400. I search for Middle element first(which is the best possible case for binary search), then i search for element that doesnt exist in the array(worst case) and then i search for a random element from the record. This way I get times for best, worst and random case. I have stored the times in files(to make plotting easier) and also printed the time to the console.

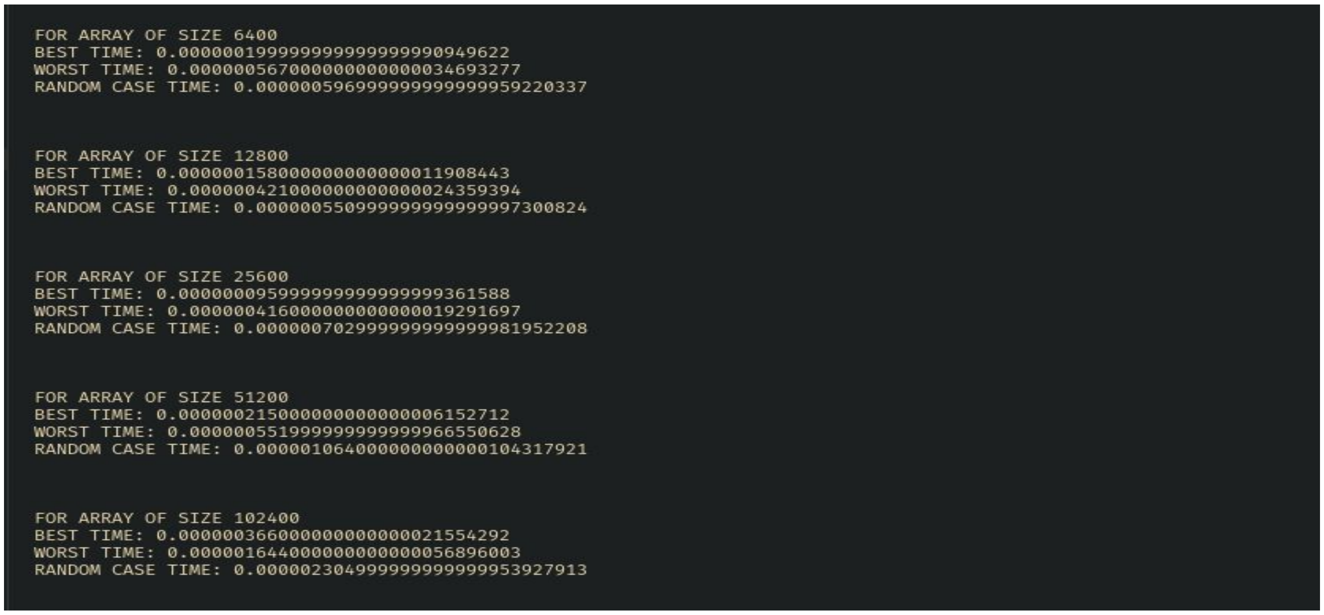
```
#include <iostream>
#include <vector>
#include <chrono>
#include <algorithm>
#include <fstream>
#include <numeric>
#include <iomanip>
#include <cstdlib>

using namespace std;

int binarySearch(vector<int> &arr, int key)
{ // iterative binary search
    int l = 0;
    int r = arr.size();
    int mid;
    while (l < r)
    {
        mid = (l + r) / 2;
        if (arr[mid] == key)
        {
            return mid;
        }
        else if (arr[mid] < key)
        {
            l = mid + 1;
        }
        else
            r = mid;
    }
    return -1;
}
int main()
{
    srand(time(0));
    vector<int> size_array{100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, 51200,
102400};
    ofstream best_case("BestCaseQ1.txt");
    ofstream worst_case("WorstCaseQ1.txt");
    ofstream random_case("RandomCaseQ1.txt");
    vector<int> arr;
    int key;   // this will store the value to be searched
    int index; // the returned index will be stored
    chrono::duration<double> elapsed_time;
    for (auto size : size_array)
    {
        cout << "FOR ARRAY OF SIZE " << size << '\n';
        arr.clear();
        // allocates size for array
        arr.resize(size);
        // this fills the array  with 1,2,3... size
        iota(arr.begin(), arr.end(), 1);

        // for best time , the value should be in the middle for binary search
        // so i'll assign key that value
        // so itll find the answer in first attempt itself
        key = arr[arr.size() / 2];
        cout << "BEST TIME: ";
        // stores start time
        auto start_time = chrono::high_resolution_clock::now();
        index = binarySearch(arr, key);
        //stores end point
        auto end_time = chrono::high_resolution_clock::now();

        // gets the elapsed time
        elapsed_time = chrono::duration_cast<chrono::duration<double>>(end_time -
start_time);
```

```
        cout << fixed << setprecision(30) << elapsed_time.count() << '\n';
        best_case << size << ':' << fixed << setprecision(30) << elapsed_time.count()
<< '\n';

        // for worst case
        // the key shuldnt be there
        // this value will not be present in arr because it only
        // has 1..size
        key = size + 1;
        cout << "WORST TIME: ";
        start_time = chrono::high_resolution_clock::now();
        index = binarySearch(arr, key);
        end_time = chrono::high_resolution_clock::now();

        elapsed_time = chrono::duration_cast<chrono::duration<double>>(end_time -
start_time);
        cout << fixed << setprecision(30) << elapsed_time.count() << '\n';
        worst_case << size << ':' << fixed << setprecision(30) << elapsed_time.count()
<< '\n';

        // for random case
        //
        key = arr[rand() % size];
        cout << "RANDOM CASE TIME: ";
        start_time = chrono::high_resolution_clock::now();
        index = binarySearch(arr, key);
        end_time = chrono::high_resolution_clock::now();

        elapsed_time = chrono::duration_cast<chrono::duration<double>>(end_time -
start_time);
        cout << fixed << setprecision(30) << elapsed_time.count() << '\n';
        random_case << size << ':' << fixed << setprecision(30) << elapsed_time.count()
<< '\n';
        cout << "\n\n\n";
    }
}
```

Text Output:

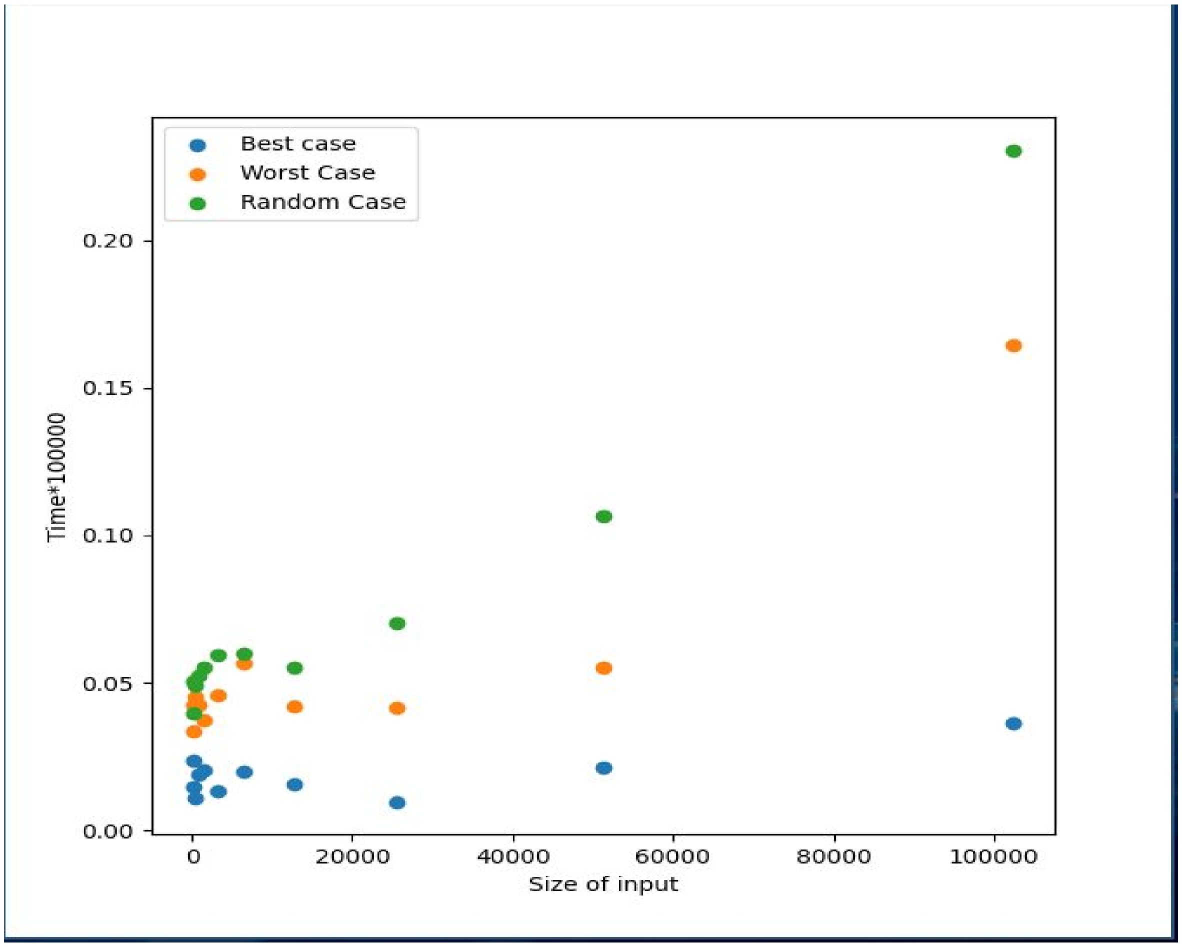


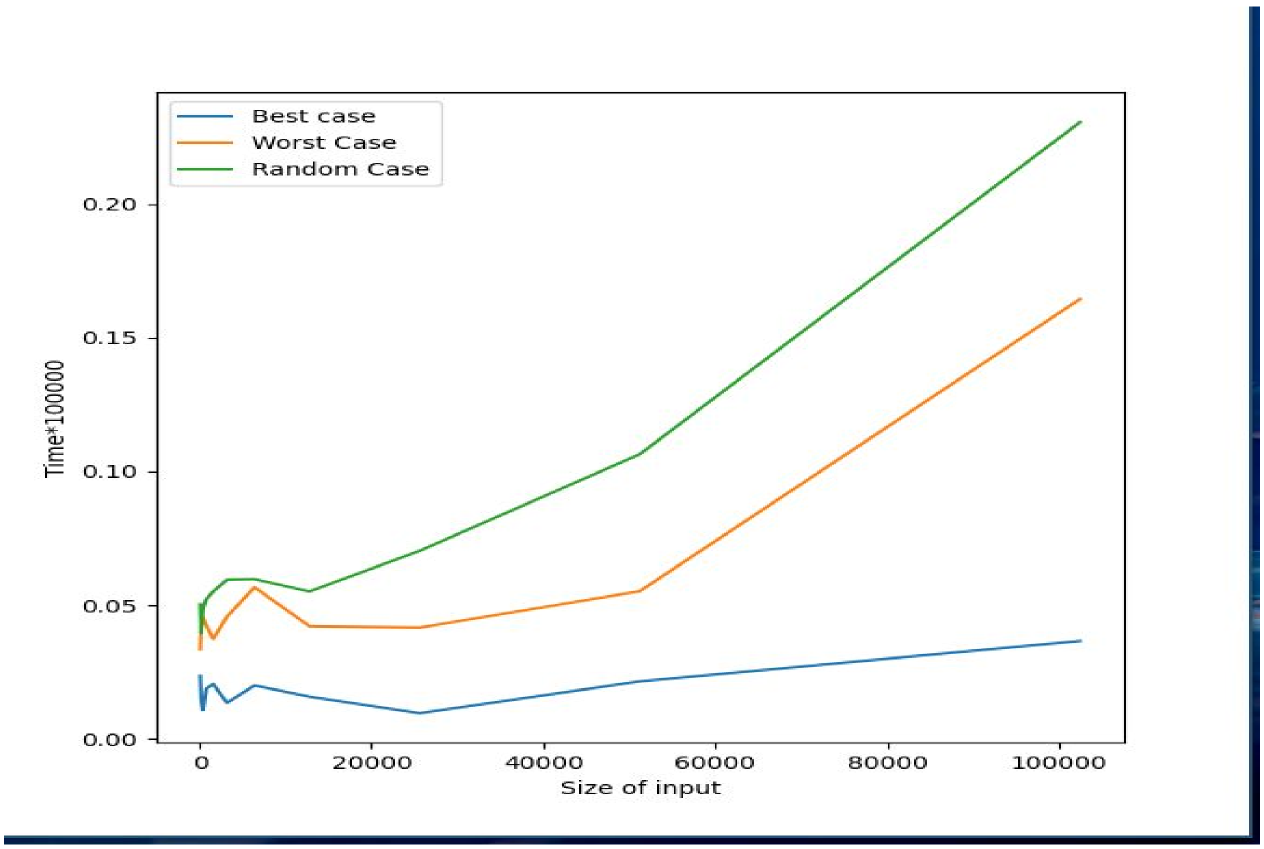
```
FOR ARRAY OF SIZE 100
BEST TIME: 0.0000002359999999999999995673301
WORST TIME: 0.0000003350000000000000021898309
RANDOM CASE TIME: 0.0000005029999999999999991002586



FOR ARRAY OF SIZE 200
BEST TIME: 0.0000001470000000000000006053467
WORST TIME: 0.0000004230000000000000015798560
RANDOM CASE TIME: 0.0000003959999999999999999020911



FOR ARRAY OF SIZE 400
BEST TIME: 0.0000001080000000000000000936148
WORST TIME: 0.0000004529999999999999993265180
RANDOM CASE TIME: 0.0000004889999999999999997988859



FOR ARRAY OF SIZE 800
BEST TIME: 0.0000001879999999999999989375063
WORST TIME: 0.0000004239999999999999985048364
RANDOM CASE TIME: 0.0000005230000000000000011273372



FOR ARRAY OF SIZE 1600
BEST TIME: 0.0000002049999999999999996017319
WORST TIME: 0.0000003739999999999999987310959
RANDOM CASE TIME: 0.0000005509999999999999997300824



FOR ARRAY OF SIZE 3200
BEST TIME: 0.0000001350000000000000004478907
WORST TIME: 0.0000004569999999999999976143514
RANDOM CASE TIME: 0.0000005950000000000000020720730
```



```
FOR ARRAY OF SIZE 6400
BEST TIME: 0.0000001999999999999999990949622
WORST TIME: 0.0000005670000000000000034693277
RANDOM CASE TIME: 0.0000005969999999999999959220337



FOR ARRAY OF SIZE 12800
BEST TIME: 0.0000001580000000000000011908443
WORST TIME: 0.0000004210000000000000024359394
RANDOM CASE TIME: 0.0000005509999999999999997300824



FOR ARRAY OF SIZE 25600
BEST TIME: 0.0000000959999999999999999361588
WORST TIME: 0.0000004160000000000000019291697
RANDOM CASE TIME: 0.0000007029999999999999981952208



FOR ARRAY OF SIZE 51200
BEST TIME: 0.0000002150000000000000006152712
WORST TIME: 0.0000005519999999999999966550628
RANDOM CASE TIME: 0.0000010640000000000000104317921



FOR ARRAY OF SIZE 102400
BEST TIME: 0.0000003660000000000000021554292
WORST TIME: 0.0000016440000000000000056896003
RANDOM CASE TIME: 0.0000023049999999999999953927913
```

Graph Output




Best case
Worst Case
Random Case
0.20
0.15
0.10
0.05
0.00
Time*100000
0
20000
40000
60000
80000
100000
Size of input

Question 2

Compare Sequential search algorithm (SSearch) with BSearch and find when SSearch will perform better than BSearch.

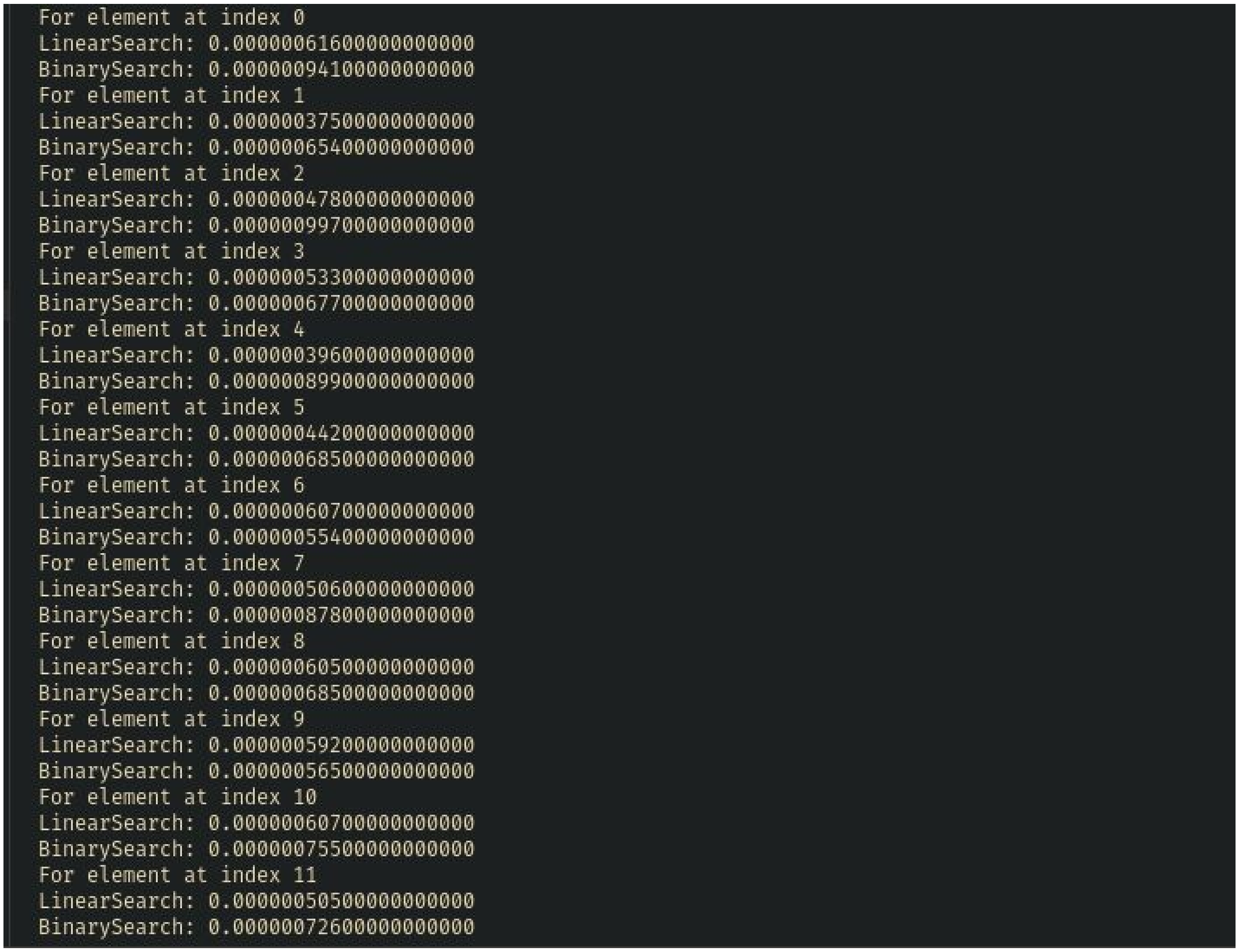
CODE:

What I did in this is, I created an array of 200 elements and filled it with values 1 to 200 in sorted order of course. After that, I performed linear search and binary search on each of the element at indices 0 to 199 and timed them. The times are stored in files from within the code and I took that and plotted the graph with python.

```
#include <iostream>
#include <vector>
#include <chrono>
#include <algorithm>
#include <fstream>
#include <numeric>
#include <iomanip>
#include <cstdlib>

using namespace std;

int binarySearch(vector<int> &arr, int key)
{ // iterative binary search
    int l = 0;
    int r = arr.size();
    int mid;
    while (l < r)
    {
        mid = (l + r) / 2;
        if (arr[mid] == key)
        {
            return mid;
        }
        else if (arr[mid] < key)
        {
            l = mid + 1;
        }
        else
            r = mid;
    }
    return -1;
}
```

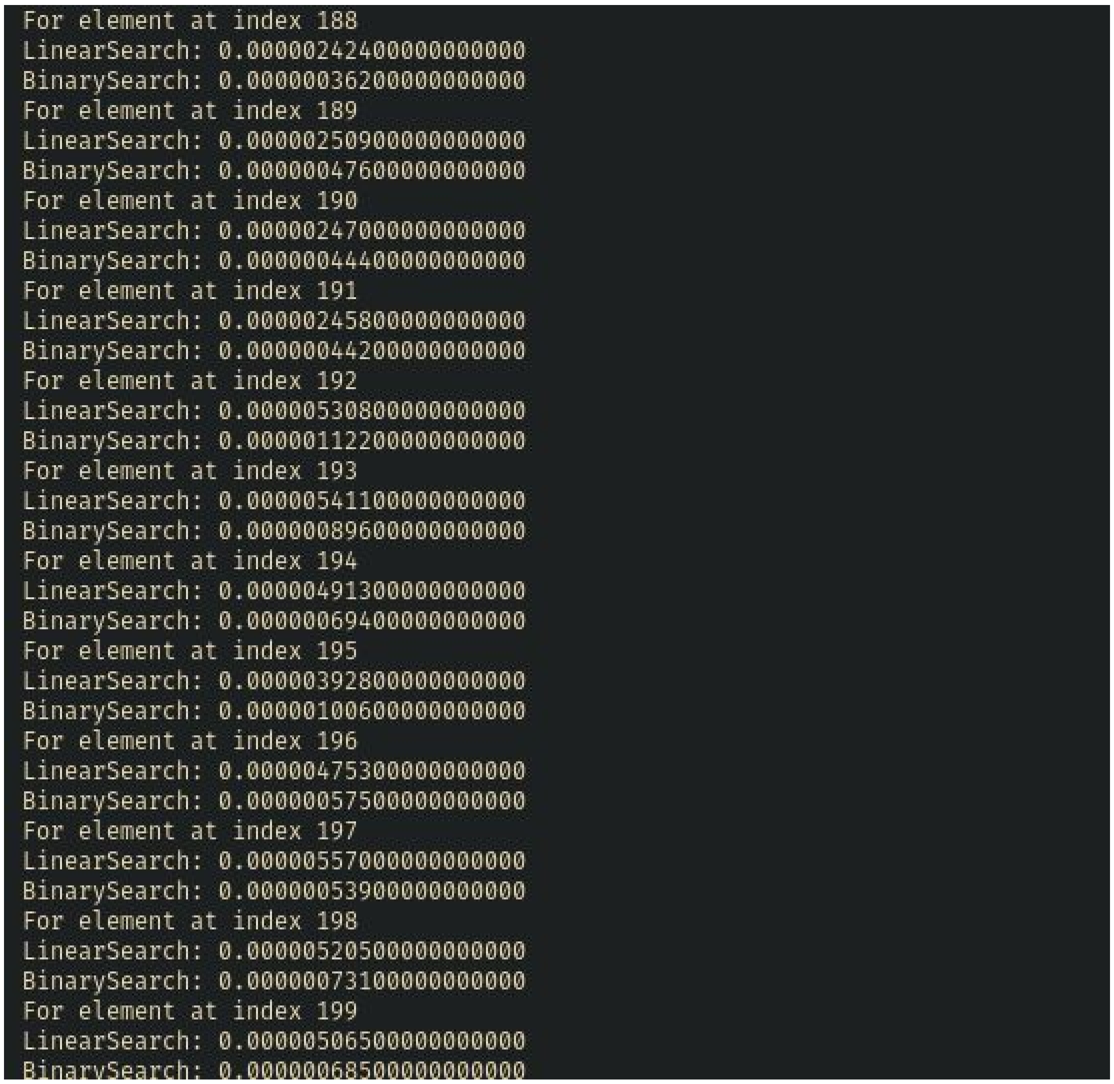
```cpp
int linearsearch(vector<int> &arr, int key)
{
    int len = arr.size();
    for (int i = 0; i < len; i++)
    {
        if (arr[i] == key)
            return i;
    }
    return -1;
}

int main()
{
    srand(time(0));
    ofstream SSearch("SSearch.txt");
    ofstream BSearch("BSearch.txt");
    chrono::duration<double> elapsed_time;
    // an array of size 200
    vector<int> arr(200, 0);
    // this will put 1,2,3..200 in the array
    iota(arr.begin(), arr.end(), 1);
    int index;
    for (int i = 0; i < arr.size(); i++)
    {
        cout << "For element at index " << i << '\n';
        // for ssearch
        cout << "LinearSearch: ";
        auto start_time = chrono::high_resolution_clock::now();
        index = linearsearch(arr, arr[i]);
        auto end_time = chrono::high_resolution_clock::now();
        elapsed_time = chrono::duration_cast<chrono::duration<double>>(end_time - start_time);
        cout << fixed << setprecision(20) << elapsed_time.count() << '\n';
        SSearch << fixed << setprecision(20) << i << ':' << elapsed_time.count() << '\n';

        cout << "BinarySearch: ";
        start_time = chrono::high_resolution_clock::now();
        index = binarySearch(arr, arr[i]);
        end_time = chrono::high_resolution_clock::now();
        elapsed_time = chrono::duration_cast<chrono::duration<double>>(end_time - start_time);
        cout << fixed << setprecision(20) << elapsed_time.count() << '\n';
        BSearch << fixed << setprecision(20) << i << ':' << elapsed_time.count() << '\n';
    }
}
```

Output



```
For element at index 0
LinearSearch: 0.00000061600000000000
BinarySearch: 0.00000094100000000000
For element at index 1
LinearSearch: 0.00000037500000000000
BinarySearch: 0.00000065400000000000
For element at index 2
LinearSearch: 0.00000047800000000000
BinarySearch: 0.00000099700000000000
For element at index 3
LinearSearch: 0.00000053300000000000
BinarySearch: 0.00000067700000000000
For element at index 4
LinearSearch: 0.00000039600000000000
BinarySearch: 0.00000089900000000000
For element at index 5
LinearSearch: 0.00000044200000000000
BinarySearch: 0.00000068500000000000
For element at index 6
LinearSearch: 0.00000060700000000000
BinarySearch: 0.00000055400000000000
For element at index 7
LinearSearch: 0.00000050600000000000
BinarySearch: 0.00000087800000000000
For element at index 8
LinearSearch: 0.00000060500000000000
BinarySearch: 0.00000068500000000000
For element at index 9
LinearSearch: 0.00000059200000000000
BinarySearch: 0.00000056500000000000
For element at index 10
LinearSearch: 0.00000060700000000000
BinarySearch: 0.00000075500000000000
For element at index 11
LinearSearch: 0.00000050500000000000
BinarySearch: 0.00000072600000000000
```

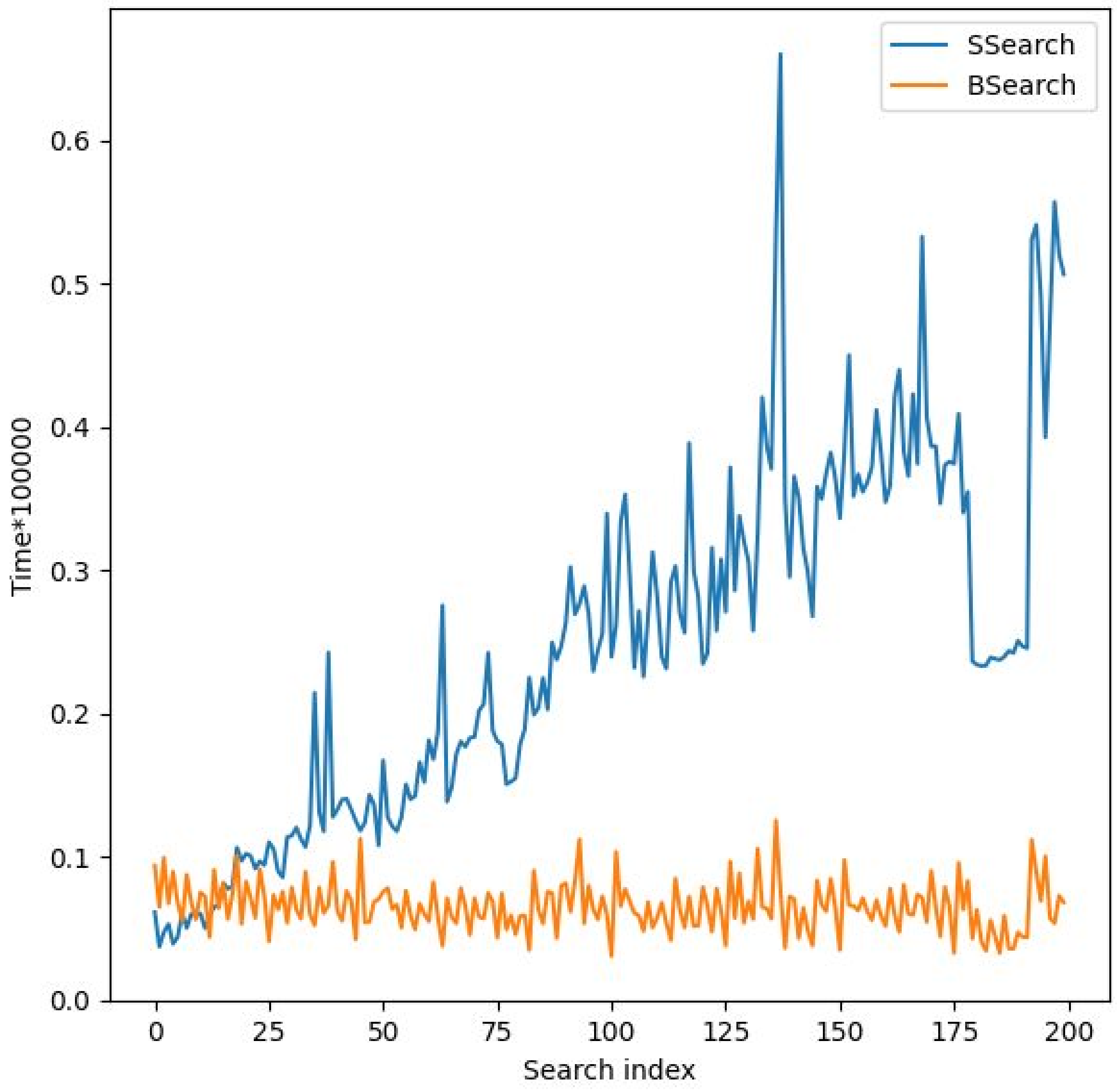
I did this for an array of size 200, so there is more output , but I'm just going to show for the end parts of array because it'll be very large. In the lower indices, we see clearly that the SSearch(Linear Search) is faster than BSearch(Binary Search). But in the higher indices, binary search is much faster, which kind of verifies our claim that linear search can be faster when the key we are searching for is in index 0..log(N) , where N is the length of array or the record we are searching in. In this case, it seems to be faster from 0 to 9 deducing from the plot and output.



```
For element at index 188
LinearSearch: 0.00000242400000000000
BinarySearch: 0.00000036200000000000
For element at index 189
LinearSearch: 0.00000250900000000000
BinarySearch: 0.00000047600000000000
For element at index 190
LinearSearch: 0.00000247000000000000
BinarySearch: 0.00000044400000000000
For element at index 191
LinearSearch: 0.00000245800000000000
BinarySearch: 0.00000044200000000000
For element at index 192
LinearSearch: 0.00000530800000000000
BinarySearch: 0.00000112200000000000
For element at index 193
LinearSearch: 0.00000541100000000000
BinarySearch: 0.00000089600000000000
For element at index 194
LinearSearch: 0.00000491300000000000
BinarySearch: 0.00000069400000000000
For element at index 195
LinearSearch: 0.00000392800000000000
BinarySearch: 0.00000100600000000000
For element at index 196
LinearSearch: 0.00000475300000000000
BinarySearch: 0.00000057500000000000
For element at index 197
LinearSearch: 0.00000557000000000000
BinarySearch: 0.00000053900000000000
For element at index 198
LinearSearch: 0.00000520500000000000
BinarySearch: 0.00000073100000000000
For element at index 199
LinearSearch: 0.00000506500000000000
BinarySearch: 0.00000068500000000000
```

Plotted Comparision

- The plot also shows for lower region, that linear search(SSearch) is much faster than BSearch. Everywhere else though, BSearch is faster by a significant amount.


SSearch
BSearch
Time*100000
Search index
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
25
50
75
100
125
150
175
200

Question 3

Take f(n)=c1.n and g(n) =c2.log(n). Plot the graph for the worst case of SSearch and BSearch along with these two functions. Find some constants for f(n) and g(n) such that the plot for SSearch is bounded by g(n)

CODE

Code for getting worst times for binary and linear search. I have done it for array of sizes 100,200,400,800,1600,3200,6400,12800,25600,51200,102400. All the time I'm searching for element that doesnt exist in the record, so its guaranteed we'll get worst possible time.

```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <algorithm>
#include <fstream>
#include <numeric>
#include <iomanip>
#include <cstdlib>

using namespace std;

int binarySearch(vector<int> &arr, int key)
{ // iterative binary search
    int l = 0;
    int r = arr.size();
    int mid;
    while (l < r)
    {
        mid = (l + r) / 2;
        if (arr[mid] == key)
        {
            return mid;
        }
        else if (arr[mid] < key)
        {
            l = mid + 1;
        }
        else
            r = mid;
    }
    return -1;
}
```

```cpp
int linearsearch(vector<int> &arr, int key)
{
    int len = arr.size();
    for (int i = 0; i < len; i++)
    {
        if (arr[i] == key)
            return i;
    }
    return -1;
}

int main()
{
    srand(time(0));
    vector<int> size_array{100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, 51200, 102400};
    ofstream SSearch("WorstCaseSSearch.txt");
    ofstream BSearch("WorstCaseBSearch.txt");
    chrono::duration<double> elapsed_time;
    vector<int> arr;
    int index, key;
    for (int size : size_array)
    {
        cout << "FOR SIZE=" << size << '\n';
        arr.clear();
        arr.resize(size);
        // puts 1,2,3,4..size in the array
        iota(arr.begin(), arr.end(), 1);
        // the value size+1 will never be in the record, so this will always yield worst time
        key = size + 1;
        // for ssearch
        cout << "LinearSearch: ";
        auto start_time = chrono::high_resolution_clock::now();
        index = linearsearch(arr, key);
        auto end_time = chrono::high_resolution_clock::now();
        elapsed_time = chrono::duration_cast<chrono::duration<double>>(end_time - start_time);
        cout << fixed << setprecision(20) << elapsed_time.count() << '\n';
        SSearch << fixed << setprecision(20) << size << ':' << elapsed_time.count() << '\n';

        cout << "BinarySearch: ";
        start_time = chrono::high_resolution_clock::now();
        index = binarySearch(arr, key);
        end_time = chrono::high_resolution_clock::now();
        elapsed_time = chrono::duration_cast<chrono::duration<double>>(end_time - start_time);
        cout << fixed << setprecision(20) << elapsed_time.count() << '\n';
        BSearch << fixed << setprecision(20) << size << ':' << elapsed_time.count() << '\n';
    }
}
```

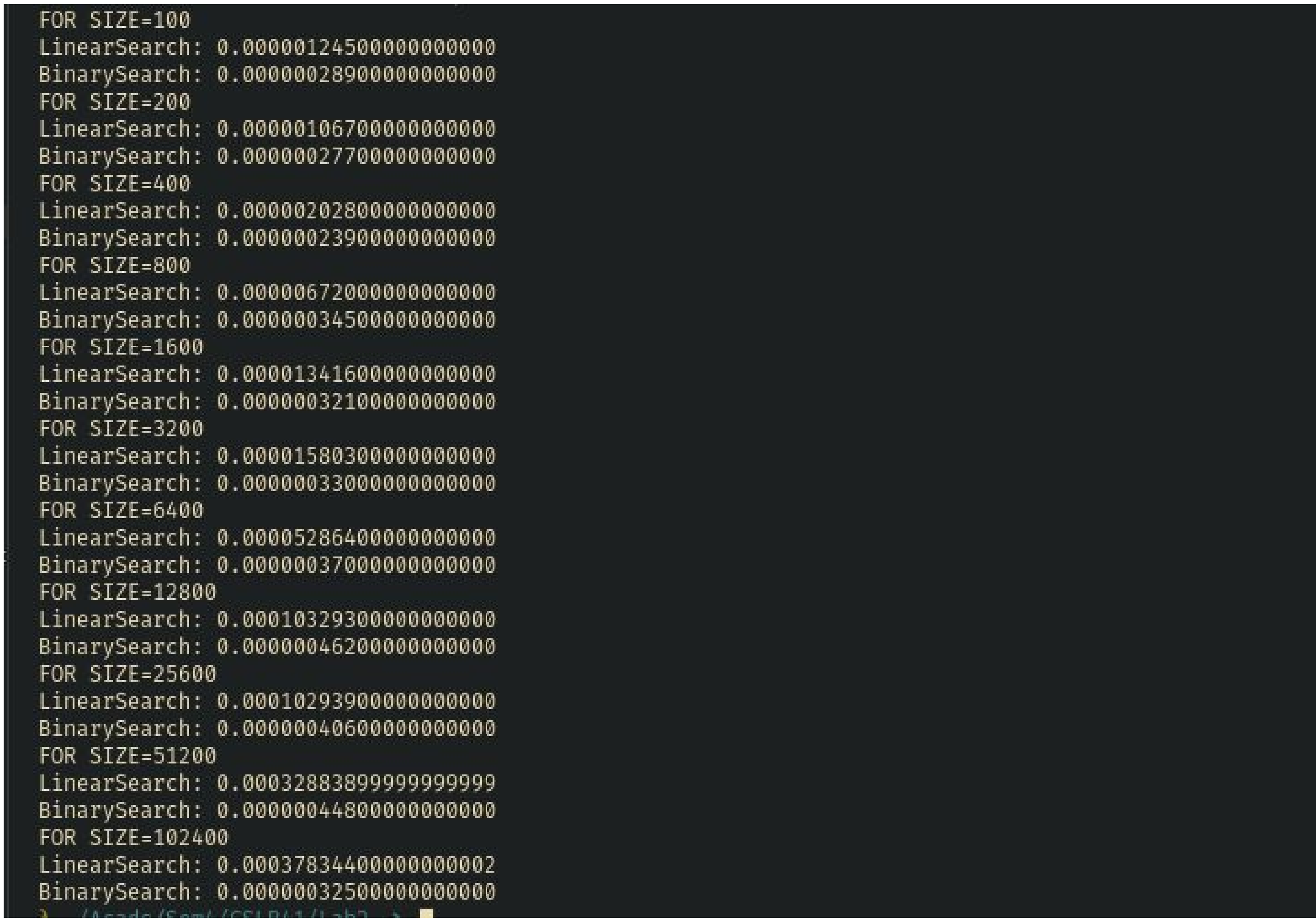
Code for finding the constant. To do this, I did the following.
Let
 ssearchtimes=[x1,x2,x3,x4,x5]
 linearTime = [y1,y2,y3,y4,y5]
Constants = [x1/y1,x2/y2,x3/y3,x4/y4,x5/y5]
BoundConstant = max(Constants)

We suppose constant is 1 initially and then find out what the actual constant for our times are , by doing element wise division. I did similarly for bsearch as well.

```
import matplotlib.pyplot as plt
import math

ssearch = [line.split(':')
           for line in open('WorstCaseSSearch.txt', 'r').readlines()]
bsearch = [line.split(':')
           for line in open('WorstCaseBSearch.txt', 'r').readlines()]

sizes = [int(i[0]) for i in ssearch]

ssearch_times = [float(i[1]) for i in ssearch]
bsearch_times = [float(i[1]) for i in bsearch]

linearWith_c_equal_to_1 = [1*i for i in sizes]

logWith_c_equal_to_1 = [math.log2(i) for i in sizes]

#### I'll find the ratio of our times with a linear slop for ssearch
###### The max of that ratio will bound our curve

linearConstants = list(
    map(lambda x: x[0]/x[1], list(zip(ssearch_times, linearWith_c_equal_to_1))))
linearBoundConstant = max(linearConstants)
boundingLinearCurve = [linearBoundConstant *
                       i for i in linearWith_c_equal_to_1]

logConstants = list(
    map(lambda x: x[0]/x[1], list(zip(bsearch_times, logWith_c_equal_to_1))))
logBoundConstant = max(logConstants)
boundingLogCurve = [logBoundConstant * i for i in logWith_c_equal_to_1]

# logWith_c_equal_to_2 = [2*math.log2(i) for i in sizes]
# logWith_c_equal_to_3 = [3*math.log2(i) for i in sizes]
```

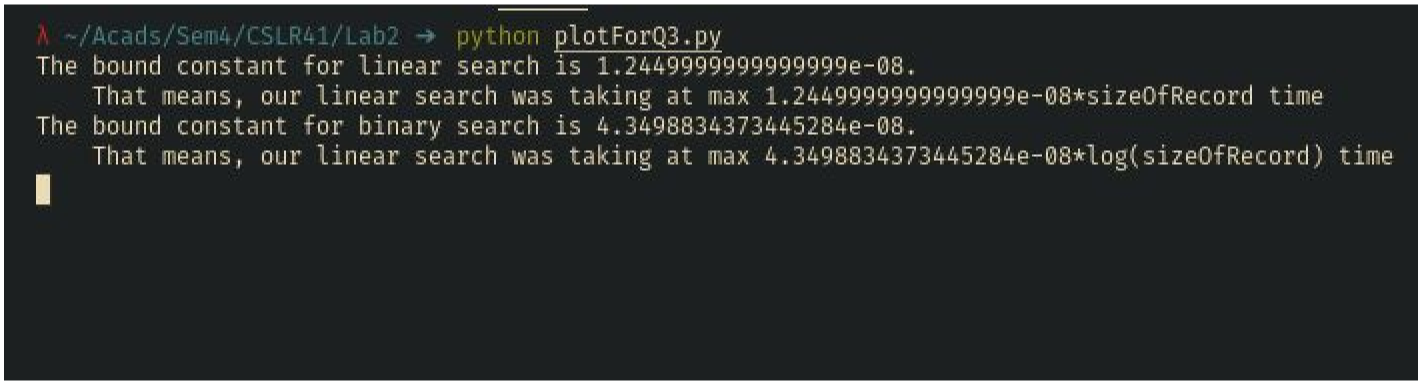
```
print(
    f'''The bound constant for linear search is {linearBoundConstant}.
    That means, our linear search was taking at max {linearBoundConstant}*sizeOfRecord time''')
print(
    f'''The bound constant for binary search is {logBoundConstant}.
    That means, our linear search was taking at max {logBoundConstant}*log(sizeOfRecord)
time''')

plt.plot(sizes, ssearch_times)
plt.plot(sizes, bsearch_times)
plt.plot(sizes, boundingLinearCurve)
plt.plot(sizes, boundingLogCurve)

plt.legend(["SSearch ", "BSearch ",
            f"SSearch bounding curve with c={linearBoundConstant}",
            f"BSearch bounding curve with c={logBoundConstant}"])
plt.xlabel("Search index")
plt.ylabel("Time")
```

Output for First Code



```
FOR SIZE=100
LinearSearch: 0.00000124500000000000
BinarySearch: 0.00000028900000000000
FOR SIZE=200
LinearSearch: 0.00000106700000000000
BinarySearch: 0.00000027700000000000
FOR SIZE=400
LinearSearch: 0.00000202800000000000
BinarySearch: 0.00000023900000000000
FOR SIZE=800
LinearSearch: 0.00000672000000000000
BinarySearch: 0.00000034500000000000
FOR SIZE=1600
LinearSearch: 0.00001341600000000000
BinarySearch: 0.00000032100000000000
FOR SIZE=3200
LinearSearch: 0.00001580300000000000
BinarySearch: 0.00000033000000000000
FOR SIZE=6400
LinearSearch: 0.00005286400000000000
BinarySearch: 0.00000037000000000000
FOR SIZE=12800
LinearSearch: 0.00010329300000000000
BinarySearch: 0.00000046200000000000
FOR SIZE=25600
LinearSearch: 0.00010293900000000000
BinarySearch: 0.00000040600000000000
FOR SIZE=51200
LinearSearch: 0.00032883899999999999
BinarySearch: 0.00000044800000000000
FOR SIZE=102400
LinearSearch: 0.00037834400000000002
BinarySearch: 0.00000032500000000000
λ ~/Acads/Sem4/CSLR41/Lab2 →
```

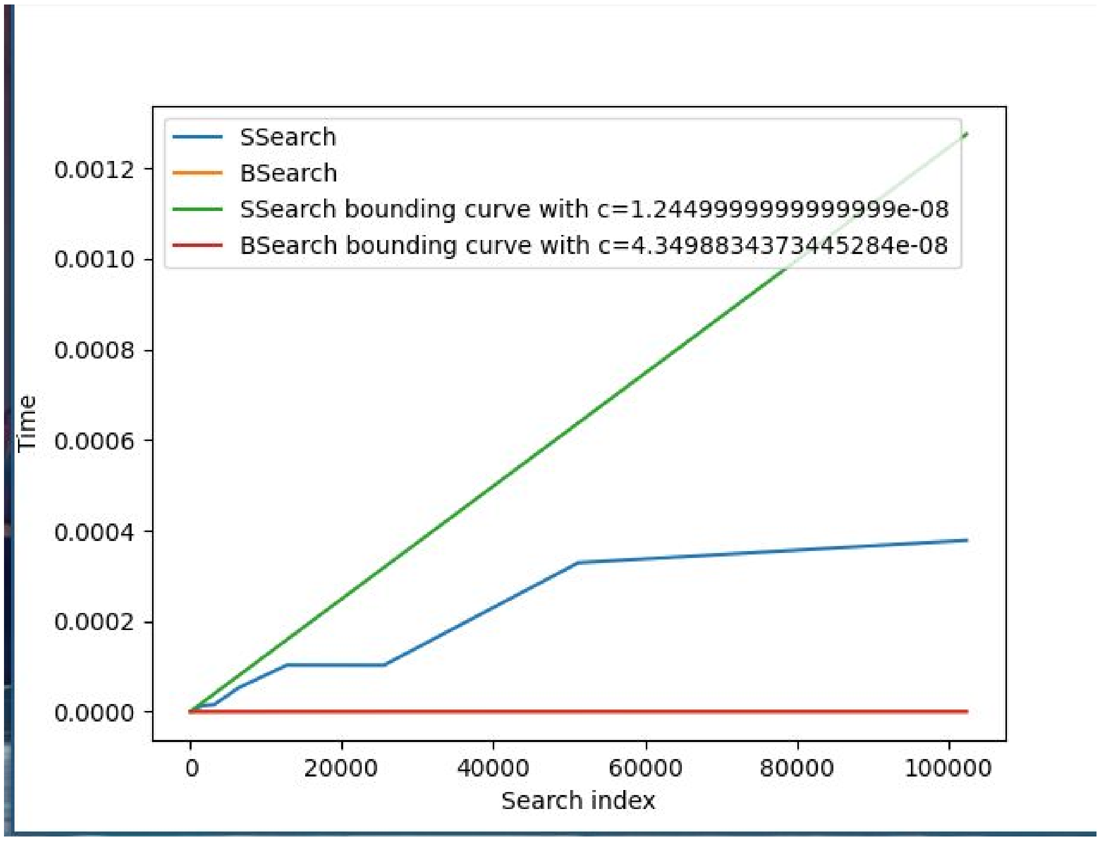
Output For Second Code

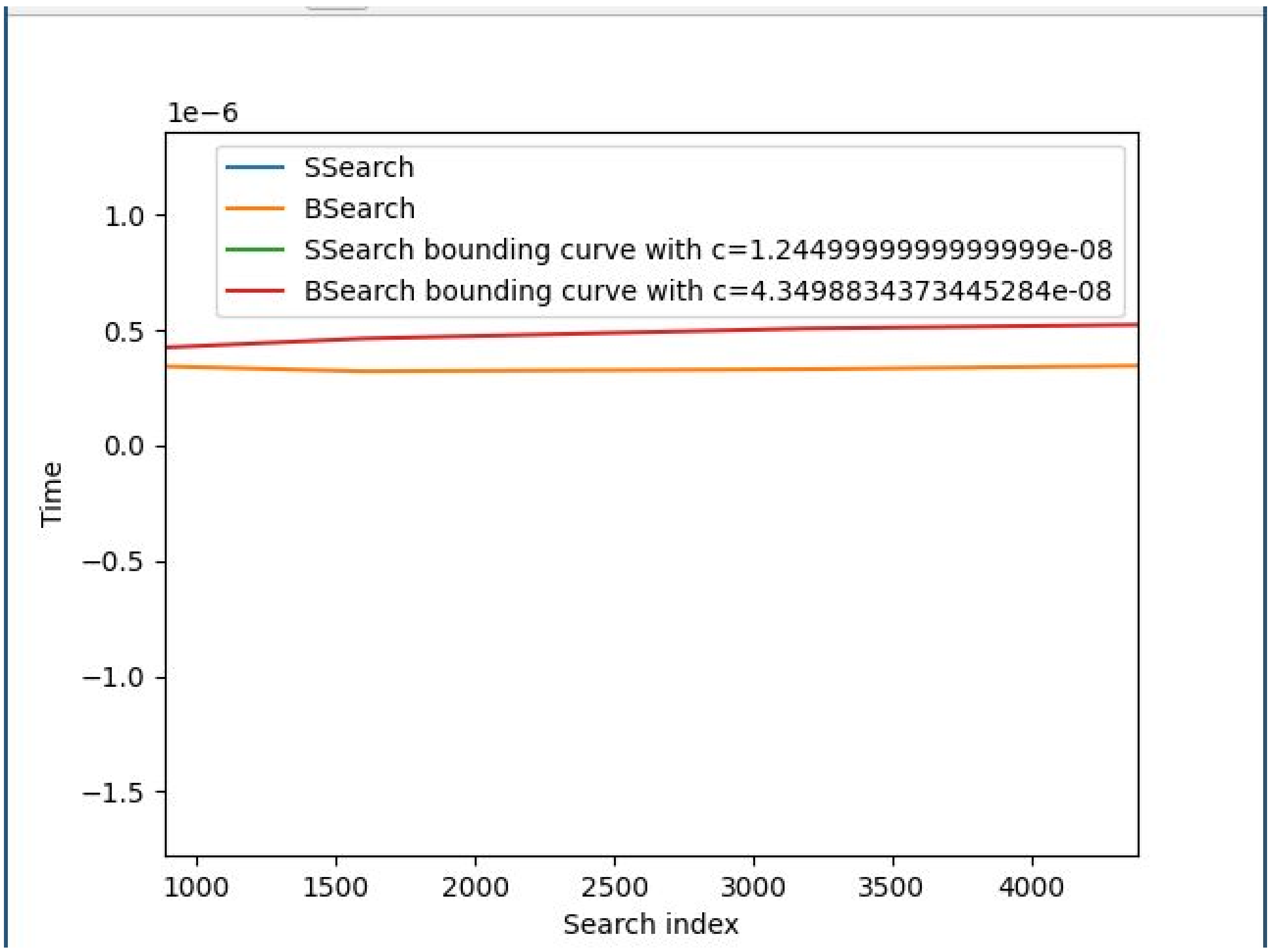


```
λ ~/Acads/Sem4/CSLR41/Lab2 → python plotForQ3.py
The bound constant for linear search is 1.2449999999999999e-08.
    That means, our linear search was taking at max 1.2449999999999999e-08*sizeOfRecord time
The bound constant for binary search is 4.3498834373445284e-08.
    That means, our linear search was taking at max 4.3498834373445284e-08*log(sizeOfRecord) time
```

Plot

Here, the BSearch curve is not visible because the BSearch and BSearch bounding curve are so close together, its not possible to see it at such a small scale. But it is there nonetheless bounded by the red curve. I have also attached a zoomed in plot and its clear from that.




1e−6
SSearch
BSearch
SSearch bounding curve with c=1.2449999999999999e-08
BSearch bounding curve with c=4.3498834373445284e-08
1.0
0.5
0.0
−0.5
−1.0
−1.5
Time
1000
1500
2000
2500
3000
3500
4000
Search index