Workshop 2

A search function takes as inputs a list $(a_0, \dots, a_{n-1})$ and an element x. The task is to find the element x in the list if it is a member. More precisely, if there is an index i such that $a_i = x$ then a search algorithm should return the index i. If there is no such index we want a return value of -1. We will restrict ourselves to lists of integers. If we denote by A the set of all finite lists of integers then a search function is a map

$$f : A \times \mathbb{Z} \to \mathbb{Z}_{\geq -1}$$
$$f\left((a_0, \dots, a_{n-1}), x\right) = \begin{cases} i & \text{if } a_i = x \\ -1 & \text{otherwise.} \end{cases}$$

Exercise 1

a) Explain why f is not a function using the list $(3, 4, 4, 5)$ as an example. (Note the domain and codomain. Try and find inputs that lead to ambiguous outputs.)

$$f : A \times \mathbb{Z} \to \mathbb{Z}_{\geq -1}$$
$$f((3,4,4,5),4) = \{1,2\}$$

A function by definition must return only one value. A function from A to B is an assignment of EXACTLY ONE element in B to each element of A.

b) If we require the lists we use to not have repeated elements, then f can be seen as a function. Decide whether this function is injective, surjective or bijective.

Recall the definitions of injective, surjective and bijective functions

Injective property

A function $f : A \to B$ is injective if and only if for all $x, y \in A$ it holds that $f(x) = f(y)$ implies that $x = y$. This effectively means that each image can only have one preimage. No two elements in A can map to the same element in B. Equivalently, a function $f : A \to B$ is injective if

$$\forall x, y \in A, f(x) = f(y) \implies x = y.$$

f is injective, as each element in the list has its own unique index.

Surjective property

A function $f : A \to B$ is **surjective** if the image of f is the whole codomain. This effectively means that every element of B is the image of an element in A. Equivalently, a function $f : A \to B$ is surjective if

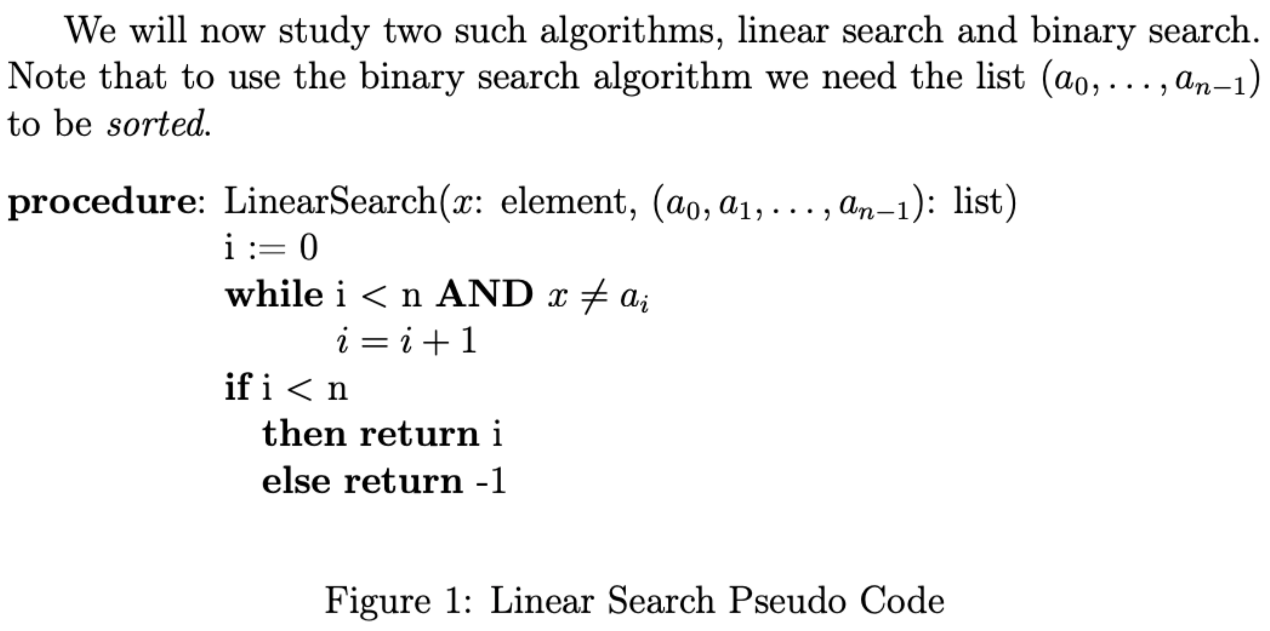
$$\forall y \in B, \exists x \in A, f(x) = y.$$

Since the codomain $\mathbb{Z}_{\geq -1}$ is an infinite set, and the codomain $A \times \mathbb{Z}$ is finite, then f is definitionally not surjective.

Bijective property

A function $f : A \to B$ is **bijective** if it is both injective and surjective. We then say f is a **bijection**. This effectively means that every element in B is the image of exactly one element in A.

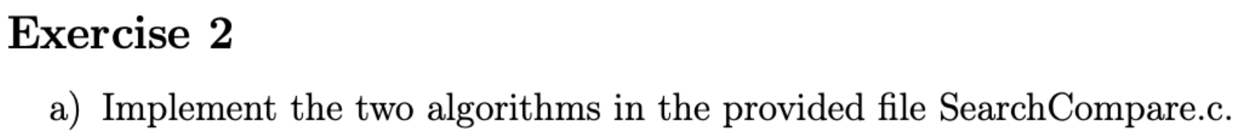
f is obviously not bijective, since it is not surjective.



We will now study two such algorithms, linear search and binary search. Note that to use the binary search algorithm we need the list $(a_0, \ldots, a_{n-1})$ to be *sorted*.

```
procedure: LinearSearch($x$: element, $(a_0, a_1, \ldots, a_{n-1})$: list)
           i := 0
           while i < n AND $x \neq a_i$
                 $i = i + 1$
           if i < n
              then return i
              else return -1
```

Figure 1: Linear Search Pseudo Code



Exercise 2

a) Implement the two algorithms in the provided file SearchCompare.c.

Done!

```
procedure: BinarySearh($x$: element, $(a_0, a_1, \ldots, a_{n-1})$: list)
           i := 0
           j := $n - 1$
           while i < j
                m=$\lfloor \frac{i+j}{2} \rfloor$
                if x $>a_m$
                   then i:=m+1
                   else j:=m
           if x=$a_i$
              then return i
              else return -1
```

Figure 2: Binary Search Pseudo Code

b) Check if the element 7000 appears in the lists List500.txt, List1000.txt, List2000.txt, List6000.txt, and List9000.txt.

The code provided runs each algorithm 10^6 times and measures the time they take.

7000 does not appear in any of the files.

c) The elements 7, 899, 3000 and 8991 are all contained in the list List9000.txt. Compare how long the different algorithms need to find them.

Finding 7 took BinSearch 0.0507s and LinSearch 0.0048s

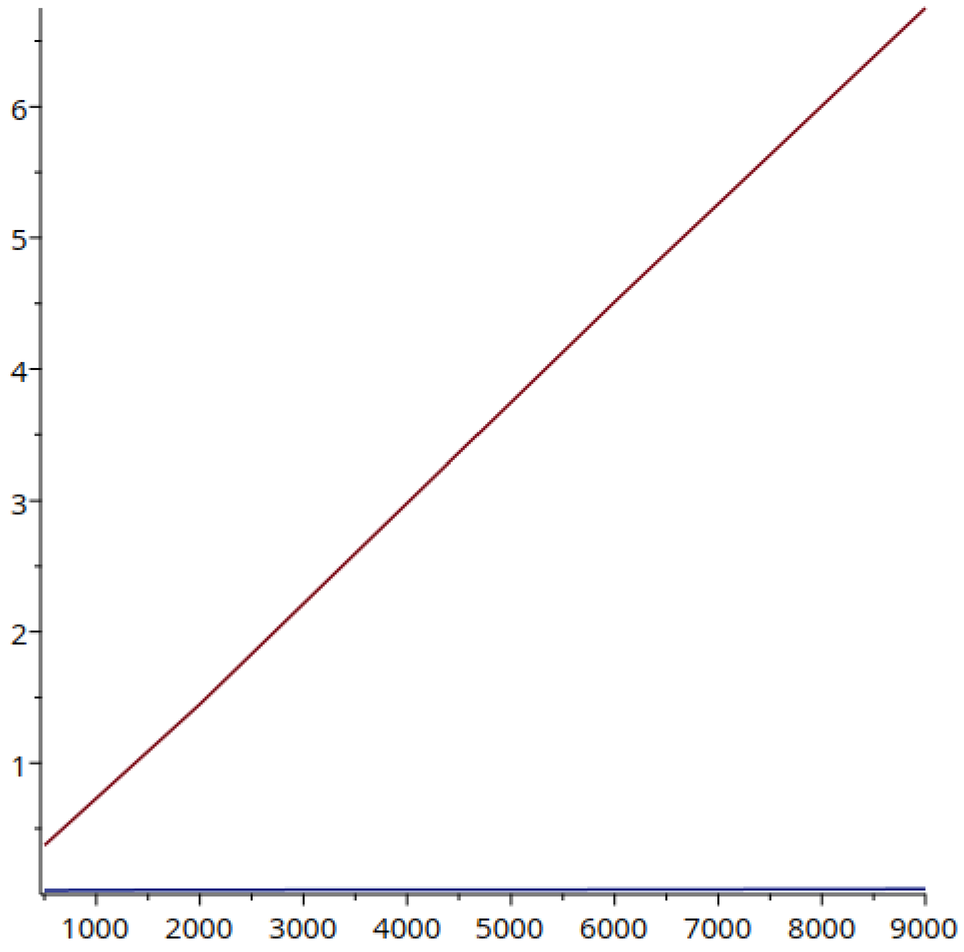
Finding 899 took BinSearch 0.0578s and LinSearch 0.7070s

Finding 3000 took BinSearch 0.0594s and LinSearch 2.349s

Finding 8991 took BinSearch 0.0578s and LinSearch 6.9593s

d) Take note of how long the algorithms take for the different lists and plot list size on the x-axis and the time it took on the y-axis.

List size	Linear search time	Binary search time
500	0.3725640s	0.0281510s
1000	0.7308030s	0.0302280s
2000	1.4481610s	0.0323800s
6000	4.5116670s	0.0349930s
9000	6.7518970s	0.0391740s



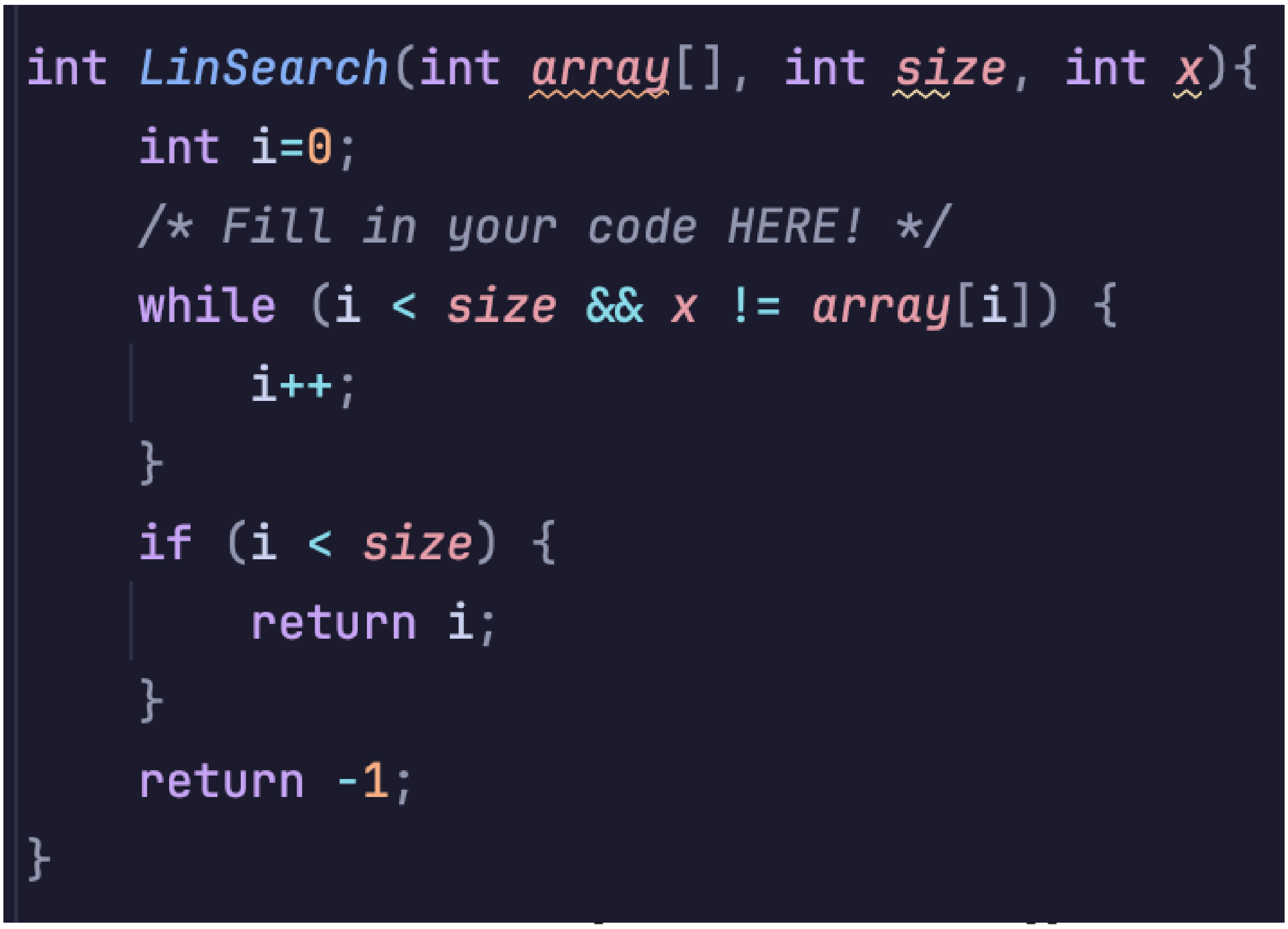
As we can see, the blue line (BinSearch) is slightly better than the red line (LinSearch). This is because binary search has an average case time complexity of $\log n$, while linear search's average case time complexity is n.

We will now study the theoretical complexity of the two algorithms. We will count comparisons made in the course of the algorithm.
Use the fact that computers optimize the AND operation, i.e. if an expression of the form $P \wedge Q$ is evaluated and P is determined to be false the machine will skip the evaluation of Q since it has no influence on the result in that case.

Exercise 3

a) Argue that the linear search algorithm will use $2n + 2$ comparisons in the worst case. Describe the inputs for which this will happen.

In the while loop, we make 2 comparisons for every element up to n in the worst case. We then, when the element is reached, make the `i < size` comparison and the other `i < size` comparison, totaling to $2n + 2$ comparisons.



```
int LinSearch(int array[], int size, int x){
    int i=0;
    /* Fill in your code HERE! */
    while (i < size && x != array[i]) {
        i++;
    }
    if (i < size) {
        return i;
    }
    return -1;
}
```

b) Show that the number of comparisons in the worst case for linear search is in $\Theta(n)$.

If we see the witnesses as $C = 3, k = 2$, then our big O case is $|f(x)| \leq C|g(x)| \Leftrightarrow |2n+2| \leq 3 \cdot n$ where $x > 2$. Our big Omega case, with witnesses $C = 2, k = 0$ is $|f(x)| \geq C|g(x)| \Leftrightarrow |2n+2| \geq 2 \cdot n$ where $x > 0$. With this we have shown that the worst case for linear search is in big Theta of n.

c) Consider the binary search algorithm and assume that the list has length $n = 2^k$.

i) Check that every full iteration of the while loop uses two comparisons.

So true bestie

ii) Show that in the first iteration of the while loop $m = 2^{k-1} - 1$.

$i = 0$
$j = 2^k - 1$
$m = \lfloor \frac{2^k - 1}{2} \rfloor$
It is a rule that $\frac{2^x}{2} = 2^{x-1}$. We can split m such that
$m = \lfloor 2^{k-1} \rfloor + \lfloor -\frac{1}{2} \rfloor$
Here -1/2 floored becomes -1, so
$m = 2^{k-1} - 1$

iii) Argue that after the first time the while loop has run we have excluded half the elements of the list.

Either the starting point i or the ending point j becomes m after the first iteration, and m is at the middle point of the list. This of course means the area of the list we are searching within becomes halved.

iv) Check that we have to half the list $\log_2(n)$ times to end up with a list of just one element. (Remember that we assume $n = 2^k$. What happens if n is not a power of 2?)

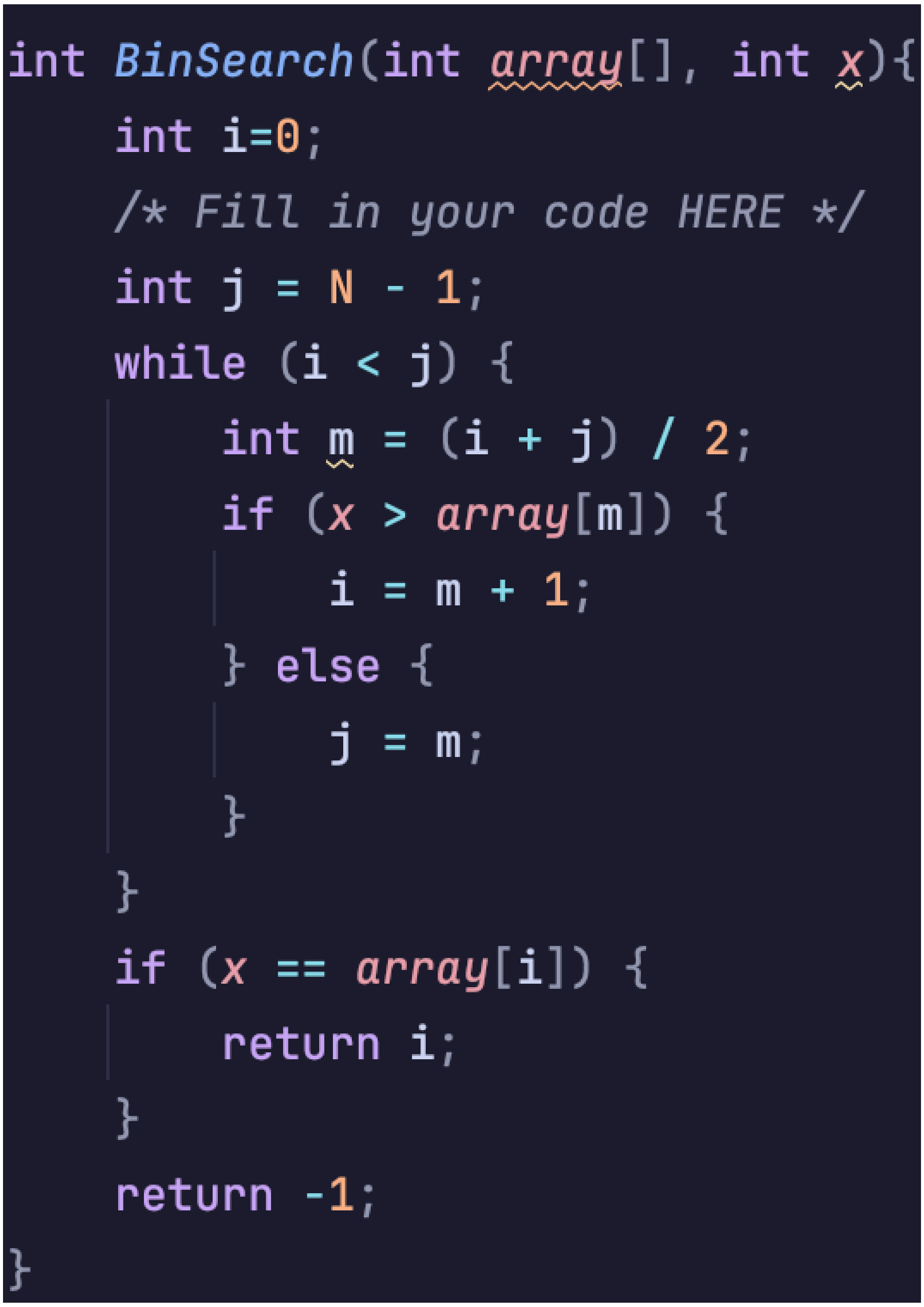
$$\frac{2^k}{2} = 2^{k-1}$$

This means if we want to turn 2^k into 1, we must halve 2^k enough times to reach the case where $k = 0$, so k times. $log_2(n)$ when $n = 2^k$ gives $log_2(2^k) = k$. So when n is not of the formula 2^k, we can safely assume that $log_2(n)$ halves of n should still give us 1, when using floor.

v) Consider the final steps of the algorithm. Once $i = j$ the while loop checks its run condition once more. Then the algorithm continues to check if we have found the element in question.

Show that the algorithm will need $2\log_2(n) + 2$ comparisons.

In the below code, we can see that each iteration has 2 comparisons: `i < j` and `x > array[m]`. After the iterations, we have the final comparison in while `i < j` which will return false, and then at the end we have `x == array[i]`. We have $log_2(n)$ because we in the worst case will run the loop $log_2(n)$ times.



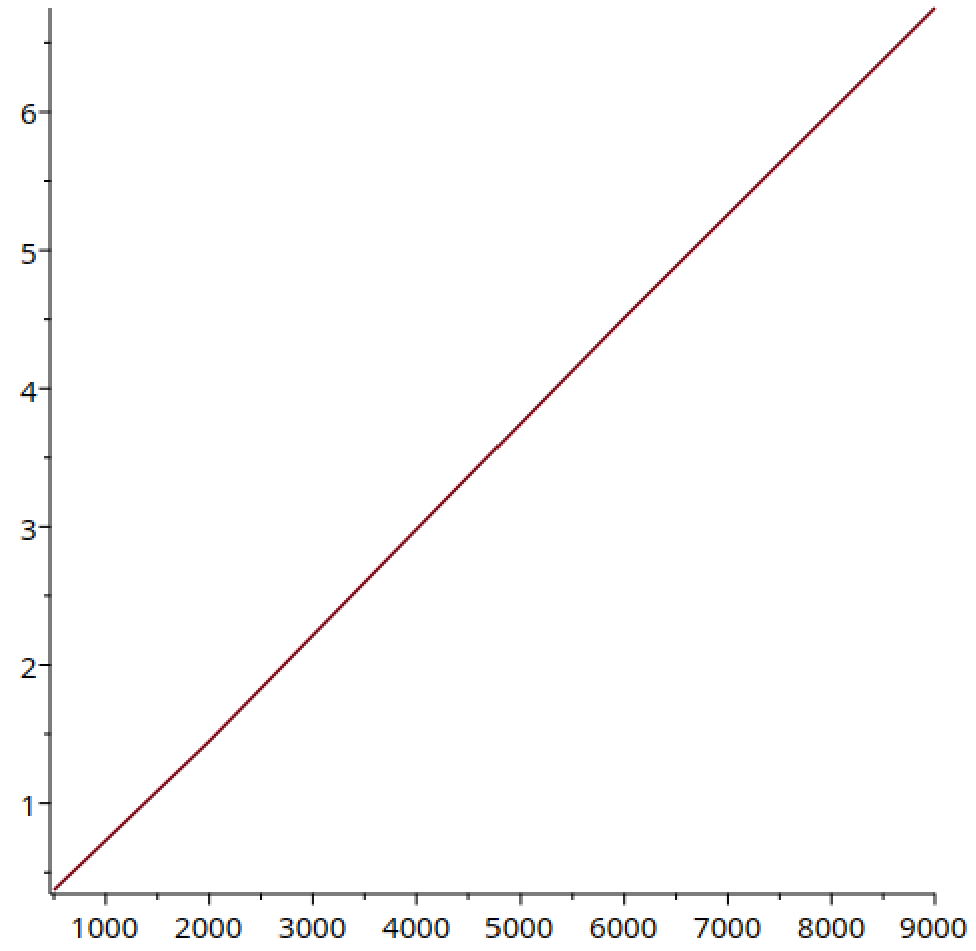
```
int BinSearch(int array[], int x){
    int i=0;
    /* Fill in your code HERE */
    int j = N - 1;
    while (i < j) {
        int m = (i + j) / 2;
        if (x > array[m]) {
            i = m + 1;
        } else {
            j = m;
        }
    }
    if (x == array[i]) {
        return i;
    }
    return -1;
}
```

d) Show that the number of comparisons needed for binary search is $\Theta(\log_2(n))$.

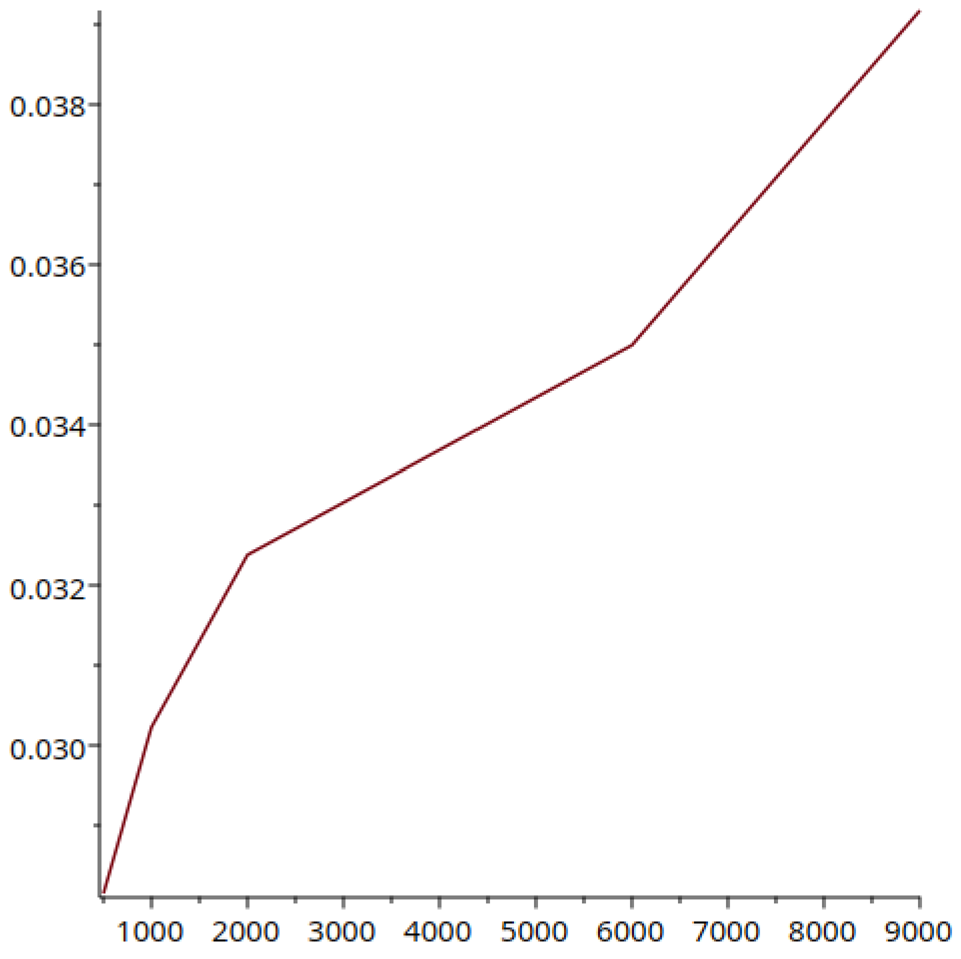
If we see the witnesses as $C = 3, k = 3$, then our big O case is $|f(x)| \leq C|g(x)| \Leftrightarrow |2log_2(n) + 2| \leq 3 \cdot log_2(n)$ where $x > 3$. Our big Omega case, with witnesses $C = 2, k = 0$ is $|f(x)| \geq C|g(x)| \Leftrightarrow |2log_2(n) + 2| \geq 2 \cdot log_2(n)$ where $x > 0$. With this we have shown that the average case for binary search is in big Theta of $log_2(n)$.

e) Compare these theoretical results with the results you got from your implementation.

The results from our implementation of linear search graph to this:



This looks like $f(x) = x$, which fits our mathematically derived asymptotic bounding of $\Theta(n)$. For our binary search implementation, the graph is this:



This doesn't look exactly like $f(x) = log_2(x)$, but this can be explained by small inaccuracies in the test environment. With a large enough data set, the graph should be asymptotically bounded by $\Theta(log_2(n))$.

f) Assume we have a list of 10000 elements and need to search for several elements in it. We want to figure out when it pays off to sort the list before searching. Use the Θ approximations for the following exercises when calculating the number of comparisons made, i.e., assume linear search uses n comparisons and binary search uses $\log_2(n)$ comparisons.

i) How many comparisons will be used when we search for m elements using linear search?

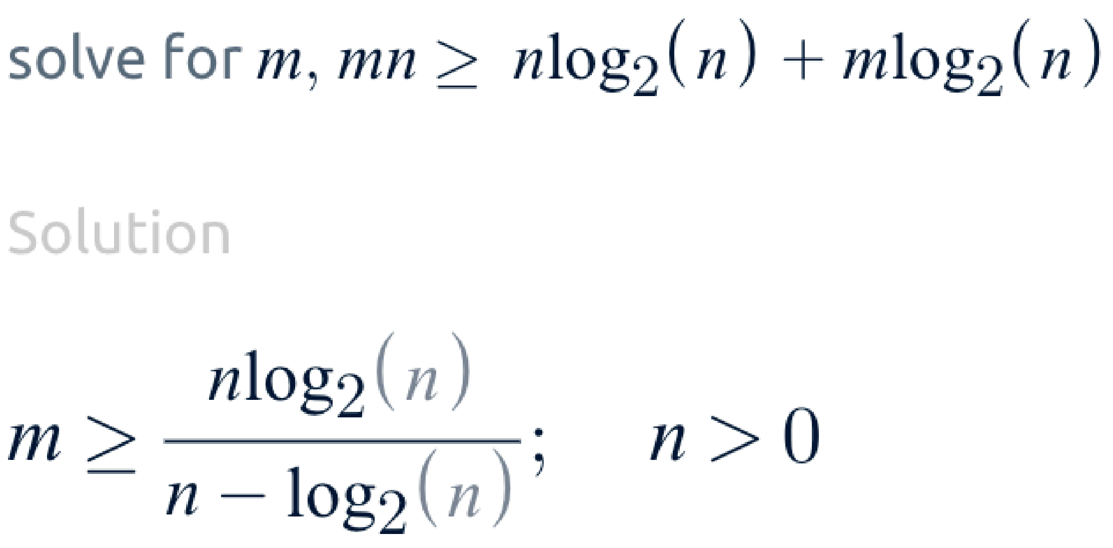
$$mn$$

ii) We will see that a list of n elements can be sorted with an effort of $\Theta(n \log_2(n))$. How many operations are needed if we sort the list first, then search for m elements using binary search?

$$n \log_2 n + m \log_2 n$$

iii) At which point, i.e., for which m, do these two approaches have similar complexity? Which one is better when?

We solve for the point at which the amount operations of linear search becomes greater than the amount of operations of binary search, and solve for m



solve for m, $mn \geq n\log_2(n) + m\log_2(n)$

Solution

$$m \geq \frac{n\log_2(n)}{n - \log_2(n)}; \quad n > 0$$

We get that once m is greater than or equal to $\frac{n \log_2 n}{n - \log_2 n}$, the same inequality is satisfied. So before this inequality is true, linear search is better, and after, binary search is better.