1 Linear Search

To solve the problem of deciding whether $x \in S$, we must determine first how we are going to organize the elements of set Sm so they can be stored in the memory of the computer. The simplest approach might be to store the elements of S in an array L. Let us assume that the positions of the array are indexed from 0 to $n-1$, as it the convention in Java, where n is the number of elements in S. For simplicity let us assume that the elements of S are all integers.

The simplest way of looking for a value x in L is to scan sequentially the array. Thus, we first compare element $L[0]$ with x; if $x = L[0]$ then we are done, otherwise we compare x with $L[1]$ and so on until either we find x or we scan the entire array L without finding x. This way of looking for a value in an array is called *linear search.*

Algorithm LinearSearch(x, L, n)
Input: Array L, value x, and number n of elements in S
Output: Position i, $0 \leq i < n$ such that $L[i] = x$, or -1 if $x \notin S$

$i \leftarrow 0$
while $(i < n)$ **and** $(L[i] \neq x)$ **do** $i \leftarrow i + 1$
if $i < n$ **then return** i
else return -1

2 Binary Search

There are many algorithms for solving a given problem, and as designers, we always look for the best one. We will say a few words later about criteria for comparing algorithms. Right now, let us assume that we want to solve the search problem as fast as we can. Is linear search the best solution for the problem at hand?

Let us assume that the elements in the array L are sorted in non-decreasing order of value. Making that assumption, can we come up with a different way of solving the problem? Just think about the way in which we look up a name in the phone book. We do not (at least I hope that you do not do this) open the phone book in the first page, compare all the names there against the name that we are looking for, then turn to the second page, do the same thing, and so on until we find the desired name.

Instead what we do is to open the phone book at some page near the middle of the book, compare the name that we want against any name in the page and based on this comparison decide whether we should look for the name in the succeeding or in the preceding pages. The search is continued in the same fashion, namely, we take the pages where the name might be, select a page and compare the name against any name from that page. When there is only one page that might contain the name of interest, then we look for it in basically the same manner.

We can apply the same idea when looking for a value x in array L. We compare x with the

element $L[mid]$ located in the middle of the array, and: (i) if $x = L[mid]$ then the search ends, (ii) if $x < L[mid]$ then we look for x in the left half of the array, (iii) if $x > L[mid]$ then we look for x in the right half of the array. To look for x in either half of the array we proceed in the same manner as above. This way of searching a value in a sorted array is called *binary search*

Algorithm BinarySearch($L, x, first, last$
Input: Array $L[first, last]$ and value x.
Output: -1 if $x \notin L$, or i, $0 \leq i < n$ if $L[i] = x$.

if $first > last$ **then return** -1
else {
    $middle \leftarrow \lfloor \frac{first+last}{2} \rfloor$
    if $L[middle] = x$ **then return** $middle$
    else if $L[middle] < x$ **then return** BinarySearch($L, x, middle + 1, last$)
    else return BinarySearch($L, x, first, middle - 1$)
}

3 Comparing Algorithms

Given 2 or more algorithms that solve the same problem, how do we select the best one? To answer this question we need to define the criteria that we are going to use to compare the algorithms. We might be interested in choosing the algorithm that is simpler, easier to understand, implement, and modify. Or we might be interested in choosing the algorithm that makes better use of the computer resources.

The first kind of criteria, qualitative criteria, is used in Software Engineering courses, and we will not use it here. Instead we will say that an algorithm is better than other if it uses less computer resources. We are interested in 2 computer resources: processing time and memory. We define the *time complexity* of an algorithm as the amount of time that the algorithm needs to complete. The *space complexity* of an algorithm is the amount of memory that it needs to execute properly.

In this course we are mainly interested in computing the time complexity of algorithms. The techniques that we describe for computing the time complexity can be used to determine the space complexity of an algorithm. So when we talk about complexity of an algorithm we are referring to time complexity, unless we specifically state otherwise.

The time that an algorithm needs to find a solution depends on the instance of the problem that it is given as input. Consider for example the linear search algorithm. If the input is an array L in which the value x is in the first position of the array, then the algorithm will find the solution very quickly, while if the value x is not in L then the algorithm will require a larger amount of time.

To consider this dependency of the time complexity on the particular instance that the algorithm has to solve, we define:

- The time complexity in the best case. Measures the least amount of time that the algorithm needs to solve an instance of the problem.
- The time complexity in the average case. Measures the average time that the algorithm needs for solving all the instances of the problem.
- The time complexity in the worst case. Measures the maximum amount of time that the algorithm needs to solve an instance of the problem.

The time complexity in the average case gives the most information about the efficiency of an algorithm, but it is very difficult to compute. To determine the complexity of an algorithm in the average case we should compute the time that the algorithm needs to solve every instance of the problem. Since the number of possible instances of a problem is usually very large, computing the average complexity is very difficult.

The complexity in the best case gives little information about the efficiency of an algorithm since usually we are not interested in knowing how fast the algorithm could be. However, we are interested in knowing how slow and algorithm might be. So in this course we will focus on computing the complexity of algorithms in the worst case. If in the worst case an algorithm takes too much time to solve a problem, we might decide that the algorithm is not good enough and then we would try to design a faster one.

4 Experimental Evaluation of the Time Complexity

How do we compute the time complexity of an algorithm? Since the time complexity measures the amount of time that an algorithm needs to complete, it seems that the only way of computing it is by implementing the algorithm and running it on some computer. Then we measure the time that the program needs to find the solution for different inputs.

There are several drawbacks that make this method inadequate for computing the time complexity of an algorithm. First, this method is very costly. We need to implement the algorithm and debug it to make sure that the implementation is correct. This takes a lot of time if the algorithm is complex.

Then we need to carefully select the set of inputs that we will use to determine the efficiency of the algorithm. We must select enough instance of the problem to make sure that our measure of the complexity accurately reflects the efficiency of the algorithm.

Third, the results that we obtain are valid only for the particular implementation that we made. If we change the implementation, either by selecting a different computer to do the tests or a different programming language/compiler for implementing the algorithm, then we need to repeat the tests and recompute the time complexity.

5 Computing the Time Complexity

If we do not actually implement the algorithm and measure the time that it needs to solve a problem, then how can we determine its time complexity? One possibility count the number of *basic* or *primitive* operations that the algorithm needs to perform, where a basic or primitive operation requires constant time (independent of the size of the input) in any implementation. Such operation might include: assigning an integer value to a variable, adding two integer values, comparing to values, and so on.

If an algorithm needs to perform a large number of primitive operations, then when implemented such an algorithm will have a large execution time. Similarly, if an algorithm needs to perform a small number of primitive operations, then when that algorithm is run on a computer it will only need a small amount of time to complete. Hence, the number of primitive operations of an algorithm will give us a good idea of how fast the algorithm is.

Given a high level description of the algorithm (in pseudo-code) we can estimate the amount of time that it takes for the algorithm to solve an instance of the problem by counting the number of primitive operations that the algorithm needs to perform. This time can easily be translated into real time if we know the actual execution time that each primitive operation takes on a real computer.

6 Time Complexity of LinearSearch in the Worst Case

The basic operations that the algorithm performs are: assignment, comparison, return, boolean and arithmetic operations.

a. Outside the **while** loop, the algorithm performs one assignment ($i \leftarrow 0$).

b. At each iteration of the **while** loop, the algorithm performs two comparisons ($i < n$) and ($L[i] \neq x$), one boolean **and** operation, one addition and one assignment $i \leftarrow i + 1$.

c. After the last iteration, i has value n. So the next time that the condition of the **while** loop is tested, the condition $i < n$ is false, so the body of the loop is not executed.

d. Finally, one comparison and one return operation are performed.

Since the for loop is repeated n times, the total number of operations is

assignments:	$n + 1$
additions:	n
comparisons ($<$):	$n + 1$
comparisons ($\neq$):	n
comparisons ($=$):	1
boolean and:	n
return:	1

Let $t_{\leftarrow}$, $t_{=}$, t_{+}, $t_{\neq}$, $t_{<}$, t_{and}, and t_{ret} denote the execution times of the primitive operations. Then, the total running time of the algorithm is

$$\begin{aligned} t(n) &= (n+1)t_{\leftarrow} + nt_{+} + (n+1)t_{<} + nt_{\neq} + nt_{\text{and}} + t_{=} + t_{\text{ret}} \\ &= (t_{\leftarrow} + t_{+} + t_{<} + t_{\neq} + t_{\text{and}})n + (t_{\leftarrow} + t_{<} + t_{=} + t_{\text{ret}}) \\ &= t_1 n + t_2 \end{aligned}$$

where $t_1 = (t_{\leftarrow} + t_{+} + t_{<} + t_{\neq} + t_{\text{and}}$ and $t_2 = t_{\leftarrow} + t_{<} + t_{=} + t_{\text{ret}}$. Since $t_1 n + t_2$ is $O(n)$, then the time complexity of the algorithm is $O(n)$.

6.1 A Faster Method

When we express the time complexity of an algorithm we are interested in knowing the order of the complexity. Thus, implementation dependent constants are not important to us. This observation allows us to compute the time complexity of an algorithm without having to keep track of the number of times that each individual operation needs to be performed. Thus, a slightly faster method for computing the time complexity of an algorithm is to count the total number of basic operations performed by the algorithm, regardless of their type. Let us use this method with algorithm LinearSearch:

1. The first instruction involves one operation.
2. Each iteration of the while loop perform 5 operations.
3. After the last iteration i has value $n-1$, so one additional operation is needed to determine that the body of the for loop does not need to be executed again.
4. At the end two more operations are performed.

Since the number of iterations of the for loop is n, the number of operations that are performed is 5n+4, which is also $O(n)$.

6.2 An Even Simpler Method

An even faster way of computing the time complexity would be not to count the exact number of operations that each part of the algorithm performs, but to identify those blocks of instructions where the total number of operations performed is a constant. For example, we know that each iteration of the while loop performs a constant number of operations, say k. We do not need to know the exact value of k, we just need to know that k is constant. Since the number of iterations of the loop is n, then the total number of operations performed by the while loop is kn. Outside the loop, a constant number k' of additional operations are performed, so the total number of operations is $kn + k'$. Since $kn + k'$ is $O(n)$, using this method we still get the right order for the time complexity.