

Algorithms for Search

Searching and sorting are two of the most fundamental and widely encountered problems in computer science. In this handout, we describe four algorithms for search.

Given a collection of objects, the goal of search is to find a particular object in this collection or to recognize that the object does not exist in the collection. Often the objects have key values on which one searches and data values which correspond to the information one wishes to retrieve once an object is found. For example, a telephone book is a collection of names (on which one searches) and telephone numbers (which correspond to the data being sought). For the purposes of this handout, we shall consider only searching for key values (e.g., names) with the understanding that in reality, one often wants the data associated with these key values.

The collection of objects is often stored in a list or an array. Given a collection of n objects in an array $A[1 \dots n]$, the i -th element $A[i]$ corresponds to the key value of the i -th object in the collection. Often, the objects are sorted by key value (e.g., a phone book), but this need not be the case. Different algorithms for search are required if the data is sorted or not.

The input to a search algorithm is an array of objects A , the number of objects n , and the key value being sought x . In what follows, we describe four algorithms for search.

1. Unordered Linear Search

Suppose that the given array was not necessarily sorted. This might correspond, for example, to a collection exams which have not yet been sorted alphabetically. If a student wanted to obtain her exam score, how could she do so? She would have to search through the entire collection of exams, one-by-one, until her exam was found. This corresponds to the unordered linear search algorithm.

Unordered Linear Search

Input: objects array A , the number of objects n , key value being sought x .

Output: if found, return position i , if not, return message “ x not found”

- Compare x with **EACH** element in array A from the every beginning.
- If $x =$ the i -th element in A . Terminate the search and return position i .
- If not, keep searching for the next element until the end of the array.
- If no proper element was found in the array, return “ x not found”.

Note that in order to determine that an object does not exist in the collection, one needs to search through the entire collection.

Now consider the following array:

i	1	2	3	4	5	6	7	8
A	34	16	25	33	7	29	48	14

If we want to search for $x = 33$ in this array. We have to compare x with (34, 16, 25, 33), each element once. Then we find 33 in position 4. Therefore, we return 4. The total number of comparisons we have executed is **4**.

If we want to search for $x = 18$ in this array. We have to compare x with (34, 16, 25, 33, 7, 29, 48, 14), each element once. After searching all the elements in this array, we don't find 18. So we return "56 not found". The total number of comparisons we have executed is **8**.

In General, we want to search for x in an **unordered** object array A with n elements. In the worst case, we have to search through the entire array to get the final answer. That means we have to execute n comparisons. Now let's use an equation to represent the number of comparisons we have executed. That should be $T(n) = n$.

2. Ordered Linear Search

Now suppose that the given array is sorted. In this case, one need not necessarily search through the entire list to find a particular object or determine that it does not exist in the collection. For example, if the collection of exams were sorted by name, one need not search beyond the "P"s to determine that the exam for "Peterson" does or does not exist in the collection. A simple modification of the above algorithm yields the ordered linear search algorithm.

Ordered Linear Search

Input: ordered objects array A , the number of objects n , key value being sought x .

Output: if found, return position i , if not, return message " x not found"

- From the every beginning of the array A , compare x with the element, say $A[i]$, in A . See if they are **EQUAL**, " $=$ ".
- If $x = A[i]$, terminate the search and return position i .
- If not, compare x with that element **AGAIN**. See if x is **GREATER** than, " $>$ " $A[i]$.
- If $x > A[i]$, go on searching for the next element in array A .
- If not, which means $x < A[i]$, terminate the search and return " x not found".

Now consider the following array, the sorted version of the array used in our previous example:

i	1	2	3	4	5	6	7	8
A	7	14	16	25	29	33	34	48

This time, if we still search for $x = 33$ in this array. We have to compare x with (7, 14, 16, 25, 29), each element **twice**, one for " $=$ ", another one for " $>$ ". Then we will compare x with (33) only once, for " $=$ ". And we will find 33 in position 6. Therefore, we return position 6. Total comparisons = $2*5 + 1 = 11$.

If we search for $x = 18$ in this array. We have to compare x with (7, 14, 16, 25), each element **twice**, one for " $=$ ", another one for " $>$ ". And in then last comparison (if $x > 25$), we got an answer "NO"! That means all the elements after 25 in this array are greater than x . Therefore, we return " x not found". Total comparisons = $2*4 = 8$.

In General, we want to search for x in an **ordered** object array A with n elements. In the worst case (x is greater than the last element in the array), we have to search through the entire array to get the final answer. That means we have to execute $n*2$ comparisons, n for " $=$ ", another n for " $>$ ". Now let's use an equation to represent the number of comparisons we have executed. That should be $T(n) = 2n$.

3. Chunk Search

Given an ordered list, one need not (and one typically does not) search through the entire collection one-by-one. Consider searching for a name in a phone book or looking for a particular exam in a sorted pile: one might naturally grab 50 or more pages at a time from the phone book or 10 or more exams at a time from the pile to quickly determine the 50 page (or 10 exam) “chunk” in which the desired data lies. One could then carefully search through this chunk using an ordered linear search. Let c be the chunk size used (e.g., 50 pages or 10 exams), and assume that we have access to a slightly generalized algorithm for ordered linear search, Encoding the above ideas; we have the chunk search algorithm.

Chunk Search

Input: ordered objects array A , the number of objects n , chunk size c , key value being sought x .

Output: if found, return position i , if not, return message “ x not found”

- Cut array A into chunks of size c .
- Compare x with the last elements of each chunk, except the last chunk! See if x is **GREATER** than that element.
- If yes, check the next chunk
- If no, that means x should be in that chunk
- Execute Ordered Linear Search inside the chunk

Again consider the following array:

i	1	2	3	4	5	6	7	8
A	7	14	16	25	29	33	34	48

Let’s choose the chunk size of 2. And we still search for $x = 33$.

First, we cut this array into 4 chunks of size 2.

Second, we compare x with the last element of each chunk, (14, 25, 33), once. See if x is GREATER than that element. We got an answer “NO” when we meet 33. That means x should be in the third chunk.

At last, we execute ordered liner search in the third chunk. And we will find 33 in position 6.

In this case, we execute 3 comparisons to find the proper chunk, and then execute 3 comparisons inside the chunk (2 for 29 and 1 for 33). In Total, we performed 6 comparisons.

In general, we want to search for x in an **ordered** object array A with n elements and chunk size of c . In the worst case, we will perform $n/c - 1$ comparisons to find the proper chunk and $2c$ comparisons to perform the linear search. In total $T(n) = n/c + 2c - 1$. Usually, we just omit the constant number, because when n grows big, the constant number will have no effect. Finally, we have $T(n) = n/c + 2c$.

Question: How to choose a good chunk size?

Answer: when $n/c = 2c$, $T(n)$ is optimized. $c = \sqrt{n/2}$

So, $T(n) = 2\sqrt{2n}$

4. Binary Search

Now consider the following idea for a search algorithm using our phone book example. Select a page roughly in the middle of the phone book. If the name being sought is on this page, you're done. If the name being sought occurs alphabetically before this page, repeat the process on the "first half" of the phone book; otherwise, repeat the process on the "second half" of the phone book. Note that in each iteration, the size of the remaining portion of the phone book to be searched is divided in half; the algorithm applying such a strategy is referred to as binary search. While this may not seem like the most "natural" algorithm for searching a phone book (or any ordered list), it is probably the fastest. This is true of many algorithms in computer science: the most natural algorithm is not necessarily the best!

Binary Search

Input: ordered objects array A , the number of objects n , key value being sought x .

Output: if found, return position i , if not, return message " x not found"

- a. Cut the array into two halves.
- b. Compare x with last element of the first half. See if x **EQUALS** that element.
- c. If yes, terminate the search, return the position.
- d. If no, compare x with last element of the first half **AGAIN**. See if x is **GREATER** than, " $>$ " that element.
- e. If yes, that means x should be in the second half. Now treat the second half as a new array, perform Binary Search on this array.
- f. If no, that means x should be in the first half. Treat the first half as a new array. Then perform Binary Search on this array.
- g. If no x is found, return " x not found".

Again consider the following array:

i	1	2	3	4	5	6	7	8
A	7	14	16	25	29	33	34	48

If we want to search for $x = 29$. We will compare x with elements (25, 33), each twice, one for "=", another for ">". And then element (29) once, for "=". And we find x in position 5. Total comparison is $2*2 + 1 = 5$.

In general, in the worst case, $T(n) = 2\log_2 n$.