

## 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  $A[1..n]$ , 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[ $A, n, x$ ]

```
1 for  $i \leftarrow 1$  to  $n$ 
2     do if  $A[i] = x$ 
3         then return  $i$ 
4         else  $i \leftarrow i + 1$ 
5 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	12	11	54	10	65	37

Consider executing the pseudocode UNORDERED-LINEAR-SEARCH[ $A, 8, 54$ ]. The variable  $i$  would initially be set to 1, and since  $A[1]$  (i.e., 34) is not equal to  $x$  (i.e., 54),  $i$  would be incremented by 1 in Line 4. Since  $A[2] \neq x$ ,  $i$  would again be incremented, and this would continue until  $i = 5$ . At this point,  $A[5] = 54 = x$ , so the loop would terminate and 5 would be returned in Line 3.

Now consider executing the pseudocode UNORDERED-LINEAR-SEARCH[ $A, 8, 53$ ]. Since 53 does not exist in the array,  $i$  would continue to be incremented until it exceeded the bounds of the **for** loop, at which point the **for** loop would terminate. In this case, “ $x$  not found” would be returned in Line 5.

## 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[ $A, n, x$ ]
1  for  $i \leftarrow 1$  to  $n$ 
2      do if  $A[i] = x$ 
3          then return  $i$ 
4      elseif  $A[i] < x$ 
5          then  $i \leftarrow i + 1$ 
6      else return “ $x$  not found”
7  return “ $x$  not found”
```

Note that the search can now be terminated early if and when it is determined that  $A[i] > x$  in Line 6.

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
$A$	10	11	12	16	34	37	54	65

Consider executing the pseudocode ORDERED-LINEAR-SEARCH[ $A, 8, 54$ ]. The variable  $i$  would initially be set to 1, and since  $A[1]$  (i.e., 10) is not equal to  $x$  (i.e., 54), a test would be performed in Line 4 to see if  $A[1]$  is less than  $x$ . Since  $A[1]$  is less than  $x$ ,  $i$  would be incremented by 1 in Line 5. Since  $A[2] < x$ ,  $i$  would again be incremented, and this would continue until  $i = 7$ . At this point,  $A[7] = 54 = x$ , so the loop would terminate and 7 would be returned in Line 3.

Now consider executing the pseudocode ORDERED-LINEAR-SEARCH[ $A, 8, 53$ ]. Since 53 does not exist in the array,  $i$  would continue to be incremented until  $i = 7$ , at which point  $A[i]$  (i.e., 54) is no longer less than  $x$  (i.e., 53) so the loop would terminate in Line 6 returning “ $x$  not found.”

## 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, ORDERED-LINEAR-SEARCH[ $A, low, high, x$ ],

which searches for  $x$  in the array range  $A[low \dots high]$ . Encoding the above ideas, we have the *chunk search* algorithm.

```

CHUNK-SEARCH[ $A, n, c, x$ ]
1   $high \leftarrow c$ 
2  while  $high < n$  and  $A[high] < x$ 
3      do  $high \leftarrow high + c$ 
4   $low \leftarrow \max\{high - c + 1, 1\}$ 
5   $high \leftarrow \min\{high, n\}$ 
6  return ORDERED-LINEAR-SEARCH[ $A, low, high, x$ ]

```

Again consider the following array:

$i$	1	2	3	4	5	6	7	8
$A$	10	11	12	16	34	37	54	65

Consider executing the pseudocode  $\text{CHUNK-SEARCH}[A, 8, 3, 34]$ . The variable  $high$  would initially be set to 3, and since  $high$  is less than  $n$  (i.e., 8) and  $A[high]$  (i.e., 12) is less than  $x$  (i.e., 34),  $high$  would be incremented by  $c$  (i.e., 3) in Line 3. Now  $high = 6$ , and since  $A[high]$  (i.e., 37) is no longer less than  $x$ , the **while** loop is terminated. The variable  $low$  is then set to 4 in Line 4, and  $\text{ORDERED-LINEAR-SEARCH}[A, 4, 6, 34]$  is called, resulting in  $i = 5$  being returned.

Note the purpose of Lines 4 and 5: Line 4 properly sets the lower array bound of the current chunk, typically one greater than the upper array bound of the *previous* chunk unless that would result in a value less than 1 (think about how this could happen). Line 5 ensures that  $\text{ORDERED-LINEAR-SEARCH}$  is called with an upper array bound no greater than the length of the array.

Now consider executing the pseudocode  $\text{CHUNK-SEARCH}[A, 8, 33]$ .  $\text{CHUNK-SEARCH}$  would behave exactly as described above, except that the call to  $\text{ORDERED-LINEAR-SEARCH}$  would return “ $x$  not found.”

## 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 is 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 provably the fastest. This is true of many algorithms in computer science: the most natural algorithm is not necessarily the best!

```

BINARY-SEARCH[ $A, n, x$ ]
1   $low \leftarrow 1$ 
2   $high \leftarrow n$ 
3  while  $low \leq high$ 
4      do  $mid \leftarrow \lfloor (low + high)/2 \rfloor$ 
5          if  $A[mid] = x$ 
6              then return  $mid$ 
7          elseif  $A[mid] < x$ 
8              then  $low \leftarrow mid + 1$ 
9          else  $high \leftarrow mid - 1$ 
10 return “ $x$  not found”

```

Again consider the following array:

$i$	1	2	3	4	5	6	7	8
$A$	10	11	12	16	34	37	54	65

Consider executing the pseudocode BINARY-SEARCH[ $A, 8, 34$ ]. The variable  $high$  is initially set to 8, and since  $low$  (i.e., 1) is less than or equal to  $high$ ,  $mid$  is set to

$$\lfloor (low + high)/2 \rfloor = \lfloor 9/2 \rfloor = 4.$$

Since  $A[mid]$  (i.e., 16) is not  $x$  (i.e., 34), and since  $A[mid] < x$ ,  $low$  is reset to  $mid + 1$  (i.e., 5) in Line 8. Since  $low \leq high$ , a new  $mid = \lfloor (5 + 8)/2 \rfloor = 6$  is calculated, and since  $A[mid]$  (i.e., 37) is greater than  $x$ ,  $high$  is now reset to  $mid - 1$  (i.e., 5) in Line 9. It is now the case that  $low$  and  $high$  are both 5;  $mid$  will then be set to 5 as well, and since  $A[mid]$  (i.e., 34) is equal to  $x$ ,  $mid = 5$  will be returned in Line 6.

Now consider executing the pseudocode BINARY-SEARCH[ $A, 8, 33$ ]. BINARY-SEARCH will behave exactly as described above until the point when  $mid = low = high$ . Since  $A[mid]$  (i.e., 34) is greater than  $x$  (i.e., 33),  $high$  will be reset to  $mid - 1 = 4$  in Line 9. Since it is no longer the case that  $low \leq high$ , the **while** loop terminates, returning “ $x$  not found” in Line 10.