CHAPTER 9

Algorithms for Searching and Sorting: Description and Analyses

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

9.1 Algorithms for Searching

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.

9.1.1 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. 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 UNORDERED-LINEAR-SEARCH algorithm on this array while searching for the number 11. The first four elements would need to be examined until the fourth element containing the value 11 is found. In analyzing the performance of search algorithms, we will consider these "examination counts" as a measure of the performance of such algorithms.

Now consider executing the UNORDERED-LINEAR-SEARCH on this array while searching for the number 13. Since 13 does not exist in the array, one must examine all eight of the array elements until one could definitively return "13 not found."

9.1.2 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. Note that while scanning the array from left-to-right (smallest-to-largest values), a search can now be terminated early if and when it is determined that the number being sought (and as yet not found) is less than the element currently being examined.

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 ORDERED-LINEAR-SEARCH on this array while searching for the number 11. In this case, the first two elements would need to be examined until the second element containing the value 11 is found.

Now consider executing the ORDERED-LINEAR-SEARCH on this array while searching for the number 13. Note that only the first four elements need be examined until the value 16 is encountered and one can definitively return "13 not found."

9.1.3 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). We shall refer to the algorithm encoding the above ideas as *chunk search*.

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 CHUNK-SEARCH on the above array while searching for the number 34 and using $c = 3$ (i.e., chunks of size 3). To determine if 34 is in the first chunk, the third element (at the end of the first "chunk") with value 12 must be examined. Since $34 > 12$, we next examine the sixth element (at the end of the second "chunk") with value 37. Since $34 < 37$, we conclude that the value 34, if it exists in the array, must be contained in the second chunk. We then simply execute ORDERED-LINEAR-SEARCH on the subarray $A[4\,..\,6]$ consisting of the three elements in the second chunk, eventually finding 34 in the fifth position.

Now consider executing CHUNK-SEARCH on the above array while searching for the number 33. CHUNK-SEARCH would behave exactly as described above, except that the call to ORDERED-LINEAR-SEARCH would return "33 not found" when searching the subarray $A[4\,..\,6]$.

9.1.4 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 many 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!

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 Binary-Search on this array while searching for the number 34. Two determine if 34 is in the first or second half of the array, we split the array in half, considering the fourth element (at the end of the "first half") whose value is 16. Since $34 > 16$, we conclude that the number 34, if it exists in the array, must be contained in the second half, i.e., the subarray $A[5\,..\,8]$. We then repeat on the second half of the array, splitting it in half and considering the sixth element whose value is 37. Since $34 < 37$, we continue with the subarray $A[5\,..\,6]$, finding the element whose value is 34 in the next iteration.

Now consider executing Binary-Search on this array while searching for the number 33. Binary-Search will behave exactly as described above until the last subarray of size 1 containing only the element whose value is 34 is considered. At this point, one can definitively return "33 not found" since there are no subarrays yet to be searched which could possibly contain the number 34.

9.2 Analysis of Algorithms

One of the major goals of computer science is to understand how to solve problems with computers. Developing a solution to some problem typically involves at least four steps: (1) designing an *algorithm* or step-by-step procedure for solving the problem, (2) analyzing the correctness and efficiency of the procedure, (3) implementing the procedure in some programming language, and (4) testing the implementation. One of the goals of CSU200 and CSU211 is to provide you with the tools and techniques necessary to accomplish these steps. In this handout, we consider the problem of analyzing the efficiency of algorithms by first considering the algorithms for search that we developed earlier.

How can one describe the efficiency of a given procedure for solving some problem? Informally, one often speaks of "fast" or "slow" programs, but the absolute execution time of an algorithm depends on many factors:

- the size of the input (searching through a list of length 1,000 takes longer than searching through a list of length 10),
- the algorithm used to solve the problem (Unordered-Linear-Search is inherently slower than Binary-Search),

- the programming language used to implement the algorithm (interpreted languages such as Basic are typically slower than compiled languages such as C++),
- the quality of the actual implementation (good, tight code can be much faster than poor, sloppy code), and
- the machine on which the code is run (a supercomputer is faster than a laptop).

In analyzing the efficiency of an algorithm, one typically focuses on the first two of these factors (i.e., the "speed" of the algorithm as a function of the size of the input on which it is run), and one typically determines the number of program steps (or some count of other interesting computer operations) as a function of the input size—the actual "wall clock" time will depend on the programming language used, the quality of the code produced, and the machine on which the code is run.

The latter three factors are important, but they typically induce a constant factor speedup or slowdown in the "wall clock" execution time of an algorithm: a 2GHz PC will be twice as fast as a 1GHz PC, a compiled language may run 10 times faster than an interpreted one, "tight" code may be 30% faster than "sloppy" code, etc. However, a more efficient algorithm may induce a speedup which is proportional to the size of the input itself: the larger the input, the greater the speedup, as compared to an inefficient algorithm.

Finally, when analyzing the efficiency of an algorithm, one often performs a *worst case* and/or an *average case* analysis. A worst case analysis aims to determine the slowest possible execution time for an algorithm. For example, if one were searching through a list, then in the worst case, one might have to go through the entire list to find (or not find) the object in question. A worst case analysis is useful because it tells you that no matter what, the running time of the algorithm cannot be slower than the bound derived. An algorithm with a "good" worst case running time will always be "fast." On the other hand, an average case analysis aims to determine how fast an algorithm is "on average" for a "typical" input. It may be the case that the worst case running time of an algorithm is quite slow, but in reality, for "typical" inputs, the algorithm is much faster: in this case, the "average case" running time of the algorithm may be much better than the "worst case" running time, and it may better reflect "typical" performance.

Average case analyses are usually much more difficult than worst case analyses—one must define what are "typical" inputs and then "average" the actual running times over these typical inputs—and in actual practice, the average case running time of an algorithm is usually only a constant factor (often just 2) faster than the worst case running time. Since worst case analyses are (1) interesting in their own right, (2) easier to perform than average case analyses, and (3)

often indicative of average case performance, worst case analyses tend to be performed most often.

With this as motivation, we now analyze the efficiency of the various algorithms for search described above.

9.2.1 Linear Search

Consider the UNORDERED-LINEAR-SEARCH algorithm discussed above. This algorithm simply iterates through the array, examining its elements one-by-one. Note that each iteration of this algorithm takes some constant amount of time to execute, dependent on the programming language used, the actual implementation, the machine on which the code is run, etc. Since we do not know this constant, we may simply count how many times the algorithm iterates or how many array elements must be examined; the running time of the algorithm will be proportional to this count. For searching and sorting algorithms, we shall consider the number of array elements which must be examined as an indicator of the performance of the algorithm.

In the worst case, on an input of size n, the number being sought will be compared to each of the n elements in the array for a total of n array examinations. Let $T(n)$ be the function of n which describes the running time of an algorithm. We then have

$$T(n) = n$$

in the worst case. This is a *linear* function—if one where to plot $T(n)$ vs. n, it would be a straight line—and this explains why this algorithm is referred to as a *linear* search algorithm.

Now consider the ORDERED-LINEAR-SEARCH algorithm described above.

In the worst case, on an input of size n, if the number being sought is at least as large as every element in the array A, then each array element must be examined. Thus, we again have

$$T(n) = n$$

in the worst case, which is a linear function.

9.2.2 Chunk Search

Now consider the CHUNK-SEARCH algorithm described above.

One element must be examined for each chunk considered, for a maximum of n/c such examinations on an array of size n using chunks of size c. ORDERED-LINEAR-SEARCH will then be performed on a chunk of size c (at most), engendering c further element examinations in the

worst case. We therefore have

$$T(n) = n/c + c. \tag{9.1}$$

Note that the running time of CHUNK-SEARCH depends on both n and c. What does this analysis tell us? We can use this analysis, and specifically Equation 9.1, in order to determine the *optimal* chunk size c; i.e., the chunk size which would *minimize* the overall running time of CHUNK-SEARCH (in the worst case).

Suppose that one were to run CHUNK-SEARCH using a very small value of c. Our chunks would be small, so there would be lots of chunks. Much of the time would be spent trying to find the right chunk, and very little time would be spent searching for the element in question within a chunk. Consider the extreme case of $c = 1$: in the worst case, $n/c = n/1 = n$ element examinations would be spent trying to find the right chunk while only $c = 1$ examinations would be spent searching within a chunk for a total of $n + 1$ examinations (in the worst case). This is worse than UNORDERED-LINEAR-SEARCH or ORDERED-LINEAR-SEARCH (though it is still linear).

Now consider using a very large value of c. Our chunks would be big, so there would be few of them, and very few element examinations would be spent finding the right chunk. However, searching for the element in question within a very large chunk would require many such examinations. Consider the extreme case of $c = n$: in the worst case, $n/c = n/n = 1$ element examinations would be spent "finding" the right chunk (our chunk is the *entire* list) while $c = n$ examinations would be spent searching within a chunk for a total of $n + 1$ compares (in the worst case). This is again worse than either UNORDERED-LINEAR-SEARCH or ORDERED-LINEAR-SEARCH (though, again, it is still linear).

Is CHUNK-SEARCH doomed to be no faster than linear search? No! One must *optimize* the value of c in order to *minimize* the total number of comparisons, and this can be accomplished by choosing a value of c which *balances* the time (number of examinations) spent *finding* the right chunk and the time spent *searching* within that chunk.

Suppose that we wish to spend precisely *equal* amounts of time searching for the correct chunk and then searching within that chunk; what value of c should we pick? Our goal is then to find a c such that n/c (the time spent searching for a chunk) is equal to c (the time spent searching within a chunk). We thus have

$$\begin{aligned} & n/c &=& \ c \\ \Leftrightarrow \quad & n &=& \ c^2 \\ \Leftrightarrow \quad & \sqrt{n} &=& \ c. \end{aligned}$$

Thus, the desired chunk size is $c = \sqrt{n}$, and using this chunk size, we have

$$\begin{aligned} T(n) &= n/c + c \\ &= n/\sqrt{n} + \sqrt{n} \\ &= \sqrt{n} + \sqrt{n} \\ &= 2\sqrt{n}. \end{aligned}$$

Note that for sufficiently large n, this is much faster than a linear search. For example, if $n = 1,000,000$, Ordered-Linear-Search would require 1,000,000 element examinations in the worst case, while Chunk-Search would require approximately 2,000 examinations in the worst case—Chunk-Search would be 500 times faster (in the worst case).

Do even better values of c exist? No. One can show through the use of calculus that $c = \sqrt{n}$ is optimal. We essentially have a function $(n/c + c)$ which we wish to minimize with respect to c. Taking the derivative with respect to c, setting this derivative to zero, and solving for c yields $c = \sqrt{n}$.

9.2.3 Binary Search

Finally, consider the Binary-Search algorithm discussed above. In each iteration of binary search, one element is examined and the procedure either returns (if the number being sought is found or the subarray being considered cannot further be cut in half) or the (sub)array being considered is cut in half. The question now becomes, how many times can one cut an array of size n in half until there is only one element left? This is the maximum number of array elements that will be examined. Consider: Cutting an array of size n in half yields $n/2$. Cutting this array in half again yields $(n/2)/2 = n/2^2 = n/4$. Cutting the array in half a third time yields $((n/2)/2)/2 = (n/2^2)/2 = n/2^3 = n/8$. In general, cutting an array in half k times yields an array of size $n/2^k$. How large can k be until $n/2^k$ is one? We have

$$\begin{aligned} & n/2^k &=& \ 1 \\ \Leftrightarrow \quad & n &=& \ 2^k \\ \Leftrightarrow \quad & \log_2 n &=& \ k. \end{aligned}$$

Therefore, at most $\log_2 n$ iterations will be performed until the array can no longer be cut in half, and thus the worst case running time of Binary-Search is

$$T(n) = \log_2 n.$$

This is faster still than even Chunk-Search: on an array of size 1,000,000 Binary-Search would perform 20 comparisons (in the worst case), as compared to 2,000 for Chunk-Search and 1,000,000 for Ordered-Linear-Search! This is the power of fast algorithms.

9.3 Algorithms for Sorting

Ordered-Linear-Search, Chunk-Search, and Binary-Search each assume that the underlying array or list which is being searched is already in sorted order. How can one take an unordered array or list and sort it? Algorithms for sorting are fundamental to computer science and dozens have been developed. In this section, we shall consider three such algorithms: Insertion-Sort, Selection-Sort, and Merge-Sort.

9.3.1 Insertion Sort

Insertion-Sort corresponds to the method that many people use to sort cards as they are being dealt to them. The first card is placed in one's hand, the second card is compared to the first and placed either before or after it, and so on. In general, one has some i cards, in sorted order, in one's hand, and the $i+1$st card is compared to successive cards (starting from the left or right) until the proper location to "insert" the card is found.

Now consider the following array:

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

Let us consider how to sort this array using Insertion-Sort. Imagine the numbers 34, 16, 12, ..., being "dealt" to you, in this order, like cards. You would start with the number 34, a trivially sorted list. Then the number 16 would be inserted into this list, producing the sorted list "16 34." The number 12 would then be inserted into this list, producing "12 16 34," and so on. In general, in phase k of the algorithm, the kth element of the array is inserted into the sorted list formed from the first $k-1$ elements of the array. The process of insertion sort is shown below, where a diamond $\diamond$ is used to separate the processed and sorted array elements from the as yet unprocessed array elements.

Phase	Processed				◇	Unprocessed			
0	◇	34	16	12	11	54	10	65	37
1	34	◇	16	12	11	54	10	65	37
2	16	34	◇	12	11	54	10	65	37
3	12	16	34	◇	11	54	10	65	37
4	11	12	16	34	◇	54	10	65	37
5	11	12	16	34	54	◇	10	65	37
6	10	11	12	16	34	54	◇	65	37
7	10	11	12	16	34	54	65	◇	37
8	10	11	12	16	34	37	54	65	◇

Analysis: What is the running time of INSERTION-SORT on an array containing n elements? In phase 1 of the algorithm, the first array element must be examined. In phase 2 of the algorithm, the second element must be examined and compared to the first (which must therefore be examined). In general, in the kth phase of INSERTION-SORT, the kth element must be examined, and in the worst case, it may be compared to each of the previously processed $k-1$ elements, resulting in k total elements being examined. Thus, the total number of elements examined (in the worst case) is given by

$$1 + 2 + 3 + \cdots + (n-2) + (n-1) + n.$$

We shall consider mathematical techniques for determining the size of such sums in Chapter 10.

9.3.2 Selection Sort

While INSERTION-SORT can be used to process cards on-line, as they are being dealt, the next algorithm we shall consider cannot proceed until all the cards have been dealt. However, in sorting a list or array of elements that are all given in advance, both algorithms are equally applicable.

SELECTION-SORT begins by examining the list of elements, "selecting" the smallest one, and "swapping" it with the first element of the list. In phase 2, SELECTION-SORT selects the smallest element from the remaining $n-1$ elements and swaps it with the element in position 2. In general, in the kth phase, SELECTION-SORT selects the smallest element from the remaining $n-k+1$ elements and swaps it with the element in position k. The process of selection sort is shown below.

Phase	Processed				◇	Unprocessed			
0	◇	34	16	12	11	54	10	65	37
1	10	◇	16	12	11	54	34	65	37
2	10	11	◇	12	16	54	34	65	37
3	10	11	12	◇	16	54	34	65	37
4	10	11	12	16	◇	54	34	65	37
5	10	11	12	16	34	◇	54	65	37
6	10	11	12	16	34	37	◇	65	54
7	10	11	12	16	34	37	54	◇	65
8	10	11	12	16	34	37	54	65	◇

Analysis: What is the running time of Selection-Sort on an array containing n elements? In phase 1 of the algorithm, all n elements must be examined in order to find the smallest element. In phase 2 of the algorithm, the $n-1$ remaining elements must all be examined to find the second smallest element. In general, in phase k of the algorithm, the remaining $n-k+1$ elements must all be examined to determine the kth smallest element. Thus, the total number of elements examined is given by

$$n + (n-1) + (n-2) + \cdots + 3 + 2 + 1.$$

Note that this is the same sum as given by the analysis of Insertion-Sort, only written backwards. Again, we shall consider mathematical techniques for determining the size of such sums in Chapter 10.

9.3.3 Merge Sort

In Chapter 11 we shall consider a third algorithm for sorting, Merge-Sort, whose implementation is inherently recursive and whose analysis depends on the mathematical technique of *recurrences*. Merge-Sort is provably faster than either Insertion-Sort or Selection-Sort, in the worst case.

Exercises

In the problems that follow, you will consider three of the algorithms for search which we discussed in class: Ordered-Linear-Search, Chunk-Search, and Binary-Search. Let $T_1(n)$, $T_2(n)$, and $T_3(n)$, respectively, be the number of element examinations[1] required by

[1] worst-case...

these algorithms when run on a list whose length is n. We have

$$\begin{aligned} T_1(n) &= n \\ T_2(n) &= \sqrt{n} \\ T_3(n) &= \log_2(n). \end{aligned}$$

In the problems that follow, you will compare and contrast the growth rates of these functions.

Exercise 9.1

On a *single* sheet of graph paper, plot the number of element examinations required for each of the three algorithms when run on lists of length $n = 1$, 2, 4, 8, and 16. For each algorithm, connect the plot points with a smooth, hand-drawn curve. See the plots given in the "Exponentials and Logs" handout for examples of what you should do. You may print a piece of graph paper from the PDF located at the following URL:

`http://www.printfreegraphpaper.com/gp/c-i-14.pdf`

(If you view this assignment on-line, you may simply click on the above hyperlink.)

Exercise 9.2

i. Suppose that you were given a budget of 20 element examinations. For each of the three algorithms, determine the largest array length such that the number of examinations made is guaranteed to be at most 20.

ii. How many times larger is the array that Binary-Search can handle, as compared to the arrays that Chunk-Search and Ordered-Linear-Search can handle? How many times larger is the array that Chunk-Search can handle, as compared to the array that Ordered-Linear-Search can handle?

Exercise 9.3

Moe, Larry, and Curly have just purchased three new computers. Moe's computer is 10 times faster than Larry's and 50 times faster than Curly's.[2] However, Moe runs Ordered-Linear-Search on his computer, while Larry and Curly run Chunk-Search and Binary-Search, respectively. Moe, Larry, and Curly begin to perform searches over various data stored on their computers...

[2]In other words, Moe's computer can perform 50 operations (such as an element examination) in the time it takes Curly's computer to perform one operation, and Moe's computer can perform 10 operations in the time it takes Larry's computer to perform one operation.

i. How large must n (the size of the array) be so that Curly's computer begins to outperform Moe's?

ii. How large must n be so that Larry's computer begins to outperform Moe's?

iii. How large must n be so that Curly's computer begins to outperform Larry's?

Hint: For parts of this problem, you will have to solve an equation that involve n (or $\sqrt{n}$) and $\log_2 n$. Such equations typically cannot be solved *analytically*, i.e., by applying the rules of algebra to obtain a formula for n. Such equations are often solved *numerically* by using... binary search! Consider the equation

$$10 \log_2 n = n.$$

First, find two initial values of n, one of which causes the left-hand side of the equation to exceed the right, and the other of which causes the right-hand side of the equation to exceed the left. The solution to the equation lies somewhere between these two values. For example, when $n = 2$, the left-hand side of the equation is 10 while the right-hand side is 2. Conversely, when $n = 128$, the left-hand side is 70 while the right-hand side is 128. The solution to this equation lies somewhere between $n = 2$ and $n = 128$. One could then apply binary search in this range to find the solution in question. (Think about which half of the interval one should search and why.)

Exercise 9.4
Consider the setup described in the problem above.

i. Moe and Curly both run a search on the same data set. Despite the fact that Curly's machine is 50 times slower than Moe's, Curly's machine performs the search 100 times faster than Moe's machine. How large is the data set?

ii. Suppose that Moe switches to CHUNK-SEARCH. On this same data set, will Moe's machine now outperform Curly's? Explain.

Hint: Again, for part of this problem, you will need to solve an equation using binary search.