

Analysis of Algorithms: Running Time

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 in an earlier handout.

1 Linear Search

Consider the UNORDERED-LINEAR-SEARCH algorithm below.

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”
```

This algorithm consist of a **for** loop (Lines 1–4) within which a block of statements (Lines 2–4) is executed. Note that this block of statements 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 this block of statements is executed. For searching and sorting algorithms, one often takes this even one step further: the basic operation in an algorithm for searching (or sorting) is a *comparison*; i.e., once checks if an array element has a particular value or if an array element is greater or less than some value. In the UNORDERED-LINEAR-SEARCH algorithm, the relevant¹ comparison takes place in Line 2, and one compare is performed per statement block. Thus, counting the execution of statements or statement blocks is equivalent (at least within constant factors) to simply counting comparisons.

In the worst case, on an input of size n , x will be compared to each of the n elements in the list for a total of n comparisons. 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 given below.

¹Note: Other comparisons take place as well; for example, the index i is implicitly compared to the upper limit n in the **for** loop of Line 1. However, for searching (and sorting) algorithms, one typically only counts comparisons with *array* elements as this is indicative of overall performance.

```

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”

```

In the worst case, on an input of size n , if x is larger than every element in the array A , then $2n$ total comparisons will be performed: for each i , $A[i]$ will be compared to x once (for equality) in Line 2 and once (for less than) in Line 4. Thus, we have

$$T(n) = 2n$$

in the worst case, which is still a linear function. (If one were to plot this function, it would be a straight line, but now with a slope of two.)

2 Chunk Search

Now consider the CHUNK-SEARCH algorithm given below.

```

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   $high \leftarrow \min\{high, n\}$ 
5   $low \leftarrow \max\{high - c + 1, 1\}$ 
6  return ORDERED-LINEAR-SEARCH[ $A, low, high, x$ ]

```

One relevant² comparison is performed in each pass through the **while** loop in Line 2, and no more than n/c passes through the **while** loop can be performed before $high$ is guaranteed to exceed n . Thus, at most n/c compares will be induced by Line 2. The call to ORDERED-LINEAR-SEARCH will be performed on a list whose size is at most c , and thus at most $2c$ additional comparisons will be performed (as described above). We therefore have

$$T(n) = n/c + 2c. \tag{1}$$

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 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, $n/c = n/1 = n$ comparisons would be spent trying to find the right chunk while only $2c = 2$ compares would be spent searching within a chunk

²Again, we count only comparisons involving array elements, by convention.

for a total of $n + 2$ compares (in the worst case). This is worse than 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 comparisons would be spent finding the right chunk. However, searching for the element in question within a very large chunk would require many comparisons. Consider the extreme case of $c = n$: in the worst case, $n/c = n/n = 1$ comparison would be spent “finding” the right chunk (our chunk is the *entire* list) while $2c = 2n$ compares would be spent searching within a chunk for a total of $2n + 1$ compares (in the worst case). This is 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 comparisons) 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 $2c$ (the time spent searching within a chunk). We thus have

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

Thus, the desired chunk size is $c = \sqrt{n/2}$, and using this chunk size, we have

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

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 2,000,000 comparisons in the worst case, while CHUNK-SEARCH would require approximately 2,828 comparisons in the worst case—CHUNK-SEARCH would be approximately 707 times faster (in the worst case).

Do even better values of c exist? No. One can show through the use of calculus that $c = \sqrt{n/2}$ is optimal. We essentially have a function $(n/c + 2c)$ 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 $c = \sqrt{n/2}$.

3 Binary Search

Finally, consider the BINARY-SEARCH algorithm given below.

```

BINARY-SEARCH[A, n, x]
1  low ← 1
2  high ← n
3  while low ≤ high
4      do mid ← ⌊(low + high)/2⌋
5          if A[mid] = x
6              then return mid
7          elseif A[mid] < x
8              then low ← mid + 1
9          else high ← mid - 1
10 return “x not found”

```

In each pass through the **while** loop, at most two relevant³ comparisons are made, one each in Lines 5 and 7. The question now becomes, how many passes through the **while** loop can be made? In each pass through the **while** loop, *mid* is set halfway between *low* and *high*, and the subsequent pass through the **while** loop occurs in either the lower “half” (*low* to *mid* - 1) or upper “half” (*mid* + 1 to *high*) of the remaining portion of the array. (In reality, subsequent passes through the **while** loop consider slightly less than half of the remaining array, due to the missing middle element.) How many times can one cut an array of size *n* in half until there is only one element left? This will correspond to the maximum number of passes through the **while** loop. 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

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

Therefore, at most $\log_2 n$ passes through the **while** can be performed, and thus the worst case running time of BINARY-SEARCH is

$$T(n) = 2 \log_2 n.$$

This is faster still than even CHUNK-SEARCH: on an array of size 1,000,000 BINARY-SEARCH would perform approximately 40 comparisons (in the worst case), as compared to 2,828 for CHUNK-SEARCH and 2,000,000 for ORDERED-LINEAR-SEARCH! This is the power of fast algorithms.

³As always, we count only those comparisons with array elements.