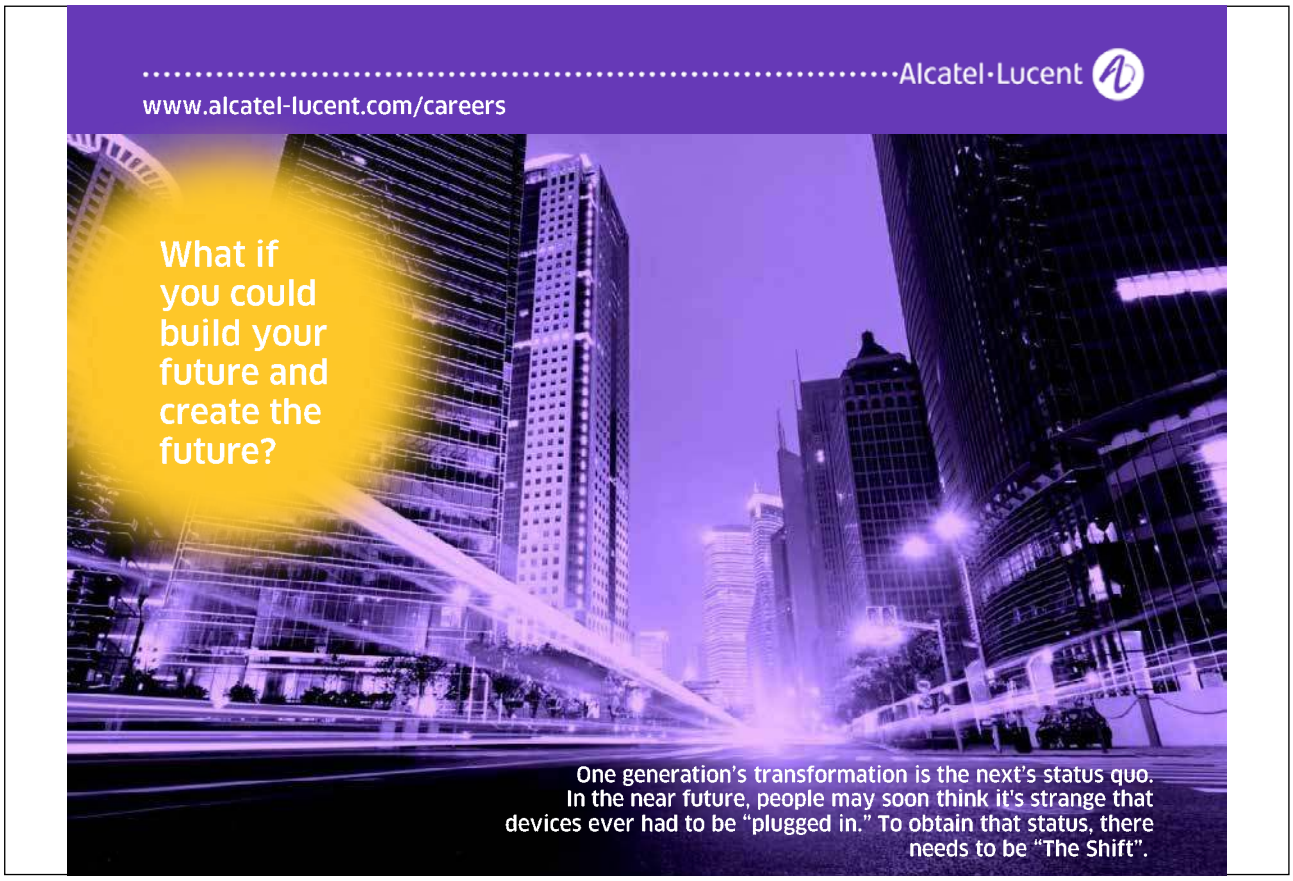
11 Analyzing Algorithms

11.1 Introduction

We have so far been developing algorithms in implementing ADTs without worrying too much about how good the algorithms are, except perhaps to point out in a vague way that certain algorithms will be more or less efficient than others. We have not considered in any rigorous and careful way how efficient our algorithms are in terms of how much work they need to do and how much memory they consume; we have not done a careful algorithm analysis.

> **Algorithm analysis**: The process of determining, as precisely as possible, how much of various resources (such as time and memory) an algorithm consumes when it executes.

In this chapter we will lay out an approach for analyzing algorithms and demonstrate how to use it on several simple algorithms. We will mainly be concerned with analyzing the amount of work done by algorithms; occasionally we will consider how much memory they consume as well.



Download free eBooks at bookboon.com

Click on the ad to read more

11.2 Measuring the Amount of Work Done

An obvious measure of the amount of work done by an algorithm is the amount of time the algorithm takes to do some task. Before we get out our stopwatches, however, we need to consider several problems with this approach.

To measure how much time an algorithm takes to run, we must code it up in a program. This introduces the following difficulties:

- A program must be written in a programming language. How can we know that the language or its compiler or interpreter have not introduced some factors that artificially increase or decrease the running time of the algorithm?
- The program must run on a machine under the control of an operating system. Machines differ in their speed and capacity, and operating systems may introduce delays; other processes running on the machine may also interfere with program timings.
- Programs must be written by programmers; some programmers write very fast code and others write slower code.

Without finding some way to eliminate these confounding factors, we cannot have trustworthy measurements of the amount of work done by algorithms—we will only have measurements of the running times of various programs written by particular programmers in particular languages run on certain machines with certain operating systems supporting particular loads.

In response to these difficulties, we begin by abandoning direct time measurements of algorithms altogether, instead focussing on algorithms in abstraction from their realization in programs written by programmers to run on particular machines running certain operating systems. This immediately eliminates most of the problems we have considered, but it leads to the question: if we can't measure time, what can we measure?

Another way to think about the amount of work done by an algorithm is to consider how many operations the algorithm executes. For example, consider the subtraction algorithm that elementary children learn. The input comes in the form of two numbers written one above the other. The algorithm begins by checking whether the value in the units column of the bottom number is greater than the value in the units column of the top number (a comparison operation). If the bottom number is greater, a borrow is made from the tens column of the top number (a borrow operation). Then the values are subtracted and the result written down beneath the bottom number (a subtraction operation). These steps are repeated for the tens column, then the hundreds column, and so forth, until the entire top number has been processed. For example, subtracting 284 from 305 requires three comparisons, one borrow, and three subtractions, for a total of seven operations.

Download free eBooks at bookboon.com

In counting the number of operations required to do this task, you probably noticed that the number of operations is related to the size of the problem: subtracting three digit numbers requires between six and eight operations (three comparison, three subtractions, and zero to two borrows), while subtracting nine digit numbers requires between 18 and 26 operations (nine comparisons, nine subtractions, and zero to eight borrows). In general, for n digit numbers, between $2n$ and $3n$-1 operations are required.

How did the algorithm analysis we just did work? We simply figured out how many operations were done in terms of the size of the input to the algorithm. We will adopt this general approach for deriving measure of work done by an algorithm:

> *To analyze the amount of work done by an algorithm, produce measures that express a count of the operations done by an algorithm as a function of the size of the input to the algorithm.*

11.3 The Size of the Input

How to specify the size of the input to an algorithm is usually fairly obvious. For example, the size of the input to an algorithm that searches a list will be the size of the list, because it is obvious that the size of the list, as opposed to the type of its contents, or some other characteristic, is what determines how much work an algorithm to search it will do. Likewise for algorithms to sort a list. An algorithm to raise b to the power k (for some constant b) obviously depends on k for the amount of work it will do.

11.4 Which Operations to Count

In most cases, certain operations are done far more often than others by an algorithm. For example, in searching and sorting algorithms, although some initial assignment and arithmetic operations are done, the operations that are done by far the most often are loop control variable increments, loop control variable comparisons, and key comparisons. These are (usually) each done approximately the same number of times, so we can simply count key comparisons as a stand-in for the others. Thus counts of key comparisons are traditionally used as the measure of work done by searching and sorting algorithms.

This technique is also part of the standard approach to analyzing algorithms: one or perhaps two *basic operations* are identified and counted as a measure of the amount of work done.

> **Basic operation**: An operation fundamental to an algorithm used to measure the amount of work done by the algorithm.

As we will see when we consider function growth rates, not counting initialization and bookkeeping operations (like loop control variable incrementing and comparison operations), does not affect the overall efficiency classification of an algorithm.

Download free eBooks at bookboon.com

11.5 Best, Worst, and Average Case Complexity

Algorithms don't always do the same number of operations on every input of a certain size. For example, consider the following algorithm to search an array for a certain value.

```
def find(key, array)
  array.each { |e| return true if key == e }
  return false
end
```

Figure 1: An Array Searching Algorithm

The measure of the size of the input is the array size, which we will label n. Let us count the number of comparisons between the key and the array elements made in the body of the loop. If the key is the very first element of the array, then the number of comparisons is only one; this is the *best case complexity*. We use $B(n)$ to designate the best case complexity of an algorithm on input of size n, so in this case $B(n) = 1$.

In contrast, suppose that the key is not present in the array at all, or is the last element in the array. Then exactly n comparisons will be made; this is the *worst case complexity*, which we designate $W(n)$, so for this algorithm, $W(n) = n$.

Download free eBooks at bookboon.com

Click on the ad to read more

Sometimes the key will be in the array, and sometimes it will not. When it is in the array, it may be at any of its n locations. The number of operations done by the algorithm depends on which of these possibilities obtains. Often we would like to characterize the behavior of an algorithm over a wide range of possible inputs, thus producing a measure of its *average case complexity*, which we designate $A(n)$. The difficulty is that it is often not clear what constitutes an "average" case. Generally an algorithm analyst makes some reasonable assumptions and then goes on to derive a measure for the average case complexity. For example, suppose we assume that the key is in the array, and that it is equally likely to be at any of the n array locations. Then the probability that it is in position i, for $0 \leq i < n$, is $1/n$. If the key is at location zero, then the number of comparisons is one; if it is at location one, then the number of comparisons is two; in general, if the key is at position i, then the number of comparisons is i+1. Hence the average number of comparisons is given by the following equation.

$$A(n) = \sum_{i=0 \text{ to } n-1} 1/n \cdot (i+1) = 1/n \cdot \sum_{i=1 \text{ to } n} i$$

You may recall from discrete mathematics that the sum of the first n natural numbers is $n(n+1)/2$, so $A(n) = (n+1)/2$. In other words, if the key is in the array and is equally likely to be in any location, then on average the algorithm looks at about half the array elements before finding it, which makes sense.

Lets consider what happens when we alter our assumptions about the average case. Suppose that the key is not in the array half the time, but when it is in the array, it is equally likely to be at any location. Then the probability that the key is at location i is $1/2 \cdot 1/n = 1/2n$. In this case, our equation for $A(n)$ is the sum of the probability that the key is not in the list (1/2) times the number of comparisons made when the key is not in the list (n), and the sum of the product of the probability that the key is in location i times the number of comparisons made when it is in location i:

$$A(n) = n/2 + \sum_{i=0 \text{ to } n-1} 1/2n \cdot (i+1) = n/2 + 1/2n \cdot \sum_{i=1 \text{ to } n} i = n/2 + (n+1)/4 = (3n+1)/4$$

In other words, if the key is not in the array half the time, but when it is in the array it is equally likely to be in any location, then the algorithm looks about three-quarters of the way through the array on average. Said another way, it looks all the way through the array half the time (when the key is absent), and half way through the array half the time (when the key is present), so overall it looks about three quarters of the way through the array. This makes sense too.

We have now completed an analysis of the algorithm above, which is called sequential search.

> **Sequential search**: An algorithm that looks through a list from beginning to end for a key, stopping when it finds the key.

Sometimes a sequential search returns the index of the key in the list, and -1 or nil if the key is not present—the `index()` operation in our List interface is intended to embody such a version of the sequential search algorithm.

Download free eBooks at bookboon.com

Not every algorithm has behavior that differs based on the content of its inputs—some algorithms behave the same on inputs of size *n* in all cases. For example, consider the algorithm in Figure 2.

```
def max(array)
  return nil if array.empty?
  result = array[0]
  1.upto(array.size-1).each do | index |
    result = array[index] if result < array[index]
  end
  return result
end
```

Figure 2: Maximum-Finding Algorithm

This algorithm, the *maximum-finding algorithm*, always examines every element of the array after the first (as it must, because the maximum value could be in any location). Hence on an input of size *n* (the array size), it always makes *n*-1 comparisons (the basic operation we are counting). The worst, best, and average case complexity of this algorithm are all the same. The *every-case complexity* of an algorithm is a the number of basic operations performed by the algorithm when it does the same number of basic operations on all inputs of size *n*. We will use $C(n)$ to designate every-case complexity, so for the maximum-finding algorithm, $C(n) = n\text{-}1$.

11.6 Summary and Conclusion

We define the various kinds of complexity we have discussed as follows.

Computational complexity: The time (and perhaps the space) requirements of an algorithm.

Every-case complexity $C(n)$: The number of basic operations performed by an algorithm as a function of the size of its input *n* when this value is the same for any input of size *n*.

Worst case complexity $W(n)$: The maximum number of basic operations performed by an algorithm for any input of size *n*.

Best case complexity $B(n)$: The minimum number of basic operations performed by an algorithm for any input of size *n*.

Average case complexity $A(n)$: The average number of basic operations performed by an algorithm for all inputs of size *n*, given assumptions about the characteristics of inputs of size *n*.

Download free eBooks at bookboon.com

We can summarize the process for analyzing an algorithm as follows:

1. Choose a measure for the size of the input.
2. Choose a basic operation to count.
3. Determine whether the algorithm has different complexity for various inputs of size n; if so, then derive measures for $B(n)$, $W(n)$, and $A(n)$ as functions of the size of the input; if not, then derive a measure for $C(n)$ as a function of the size of the input.

We will consider how to do step 3 in more detail later.

11.7 Review Questions

1. Give three reasons why timing programs is insufficient to determine how much work an algorithm does.
2. How is a measure of the size of the input to an algorithm determined?
3. How are basic operations chosen?
4. Why is it sometimes necessary to distinguish the best, worst and average case complexities of algorithms?
5. Does best case complexity have anything to do with applying an algorithm to smaller inputs?

Download free eBooks at bookboon.com

Click on the ad to read more

11.8 Exercises

1. Determine measures of the size of the input and suggest basic operations for analyzing algorithms to do the following tasks.
   a) Finding the average value in a list of numbers.
   b) Finding the number of 0s in a matrix.
   c) Searching a text for a string.
   d) Finding the shortest path between two nodes in a network.
   e) Finding a way to color the countries in a map so that no adjacent countries are the same color.
2. Write a Ruby sequential search method that finds the index of a key in an array.
3. Consider the Ruby code below.

```
def max_char_sequence(string)
  return 0 if string.empty?
  max_len = 0
  this_len = 1
  last_char = nil
  string.each_char do | this_char |
    if this_char == last_char
      this_len += 1
    else
      max_len = this_len if max_len < this_len
      this_len = 1
    end
    last_char = this_char
  end
  return (max_len < this_len) ? this_len : max_len
end
```

   a) What does this algorithm do?
   b) In analyzing this algorithm, what would be a good measure of input size?
   c) What would be a good choice of basic operation?
   d) Does this algorithm behave differently for different inputs of size n?
   e) What are the best and worst case complexities of this algorithm?

4. Compute the average case complexity of sequential search under the assumption that the likelihood that the key is in the list is p (and hence the likelihood that it is not in the list is 1-p), and that if in the list, the key is equally likely to be at any location.

Download free eBooks at bookboon.com

11.9 Review Question Answers

1. Timing depends on actual programs running on actual machines. The speed of a real program depends on the skill of the programmer, the language the program is written in, the efficiency of the code generated by the compiler or the efficiency of program interpretation, the speed of the hardware, and the ability of the operating system to accurately measure the CPU time consumed by the program. All of these are confounding factors that make it very difficult to evaluate algorithms by timing real programs.
2. The algorithm analyst must choose a measure that reflects aspects of the input that most influence the behavior of the algorithm. Fortunately, this is usually not hard to do.
3. The algorithm analyst must choose one or more operations that are done most often during execution of the algorithm. Generally, basic operations will be those used repeatedly in inner loops. Often several operations will be done roughly the same number of times; in such cases, only one operation need be counted (for reasons to be explained in the next chapter about function growth rates).
4. Algorithms that behave differently depending on the composition of inputs of size n can do dramatically different amounts of work, as we saw in the example of sequential search. In such cases, a single value is not sufficient to characterize an algorithm's behavior, and so we distinguish best, worst, and average case complexities to reflect these differences.
5. Best case complexity has to do with the behavior of an algorithm for inputs of a given size, not with behavior as the size of the input varies. The complexity functions we produce to count basic operations are already functions of the size of the input. Best, worst, average, and every-case behavior are about differences in behavior given input of a certain size.

Download free eBooks at bookboon.com