Chapter 1

Asymptotic Analysis

The fundamental object of study in computer science is the **algorithm**. An algorithm is a definite, step-by-step procedure for performing some task. In general, an algorithm takes some sort of **input** and produces some sort of **output**. Usually, it is required that an algorithm is guaranteed to terminate after a finite amount of time. All algorithms that we study in this course will have this property. Note that an algorithm is not the same as a program. A program is just a specific implementation of an algorithm. Most often, we will think of an algorithm as being a **function**, such as a function in C++ or Java, where the input to the algorithm is passed as one or more parameters to the function. The output can be given either as the return value of the function or as modifications made to the input parameters.

Here, for example is the well known algorithm *selection sort*, written as a C++ function. The purpose of this algorithm is to sort a list of integers into non-decreasing order:

```
 1.  void selection_sort( int[] list, int n ) {
 2.     for ( int top = n - 1; top > 0; top-- ) {
 3.        int max = 0;
 4.        for ( int i = 1; i <= top; i++ ) {
 5.            if ( list[i] > list[max] )
 6.               max = i;
 7.        }
 8.        int temp = list[top];
 9.        list[top] = list[max];
10.        list[max] = temp;
11.     }
12.  }
```

In this algorithm, the input is a list of integers. The second parameter to the function gives the number of items in the list. The output is the same list, with its elements rearranged into sorted order. The body of the function specifies the procedure performed by the algorithm. Note that the algorithm is this detailed procedure, not just the general idea of "sorting a list of integers."

There are many different algorithms that can be used to sort a list of integers. Selection sort is the algorithm that sorts the list in this particular way.

When we discuss algorithms, we often need a way of talking about the efficiency of the algorithm, that is, what sort of resources will be required to execute the algorithm. The resource in question can be *time*—how long will the algorithm run—or *space*—how much memory will the algorithm require. In practice, we will mostly be concerned with the **run-time efficiency** of the algorithm, but many of the same ideas apply to **space efficiency**. In general, of course, the running time of an algorithm will depend on a lot of things. Different inputs will almost always require different processing times. Implementing an algorithm in a slightly different way or in a different programming language will change the running time. And, of course, running the program on a faster computer will decrease the amount of time it takes for the algorithm to run. We need some way of talking about run-time efficiency without getting bogged down in these details.

First of all, instead of looking at the run time for a specific input, we concentrate on the *size* of the input. In the case of selection sort, for example, we could ask, How long does selection sort take to sort a list of n items? That is, the size of the input is the number of items in the list. In general, we expect that the larger the input, the longer the running time will be. In fact, the time will depend on the actual input, not just the size, but there are at least three specific times that we could consider: The **worst-case running time**, which refers to the longest running time of the algorithm for any input of size n; the **best-case running time**, which refers to the shortest running time for any input of size n; and the **average-case running time**, which refers to the average running time where the average is taken over all possible inputs of size n.

These various running times still depend on a specific implementation of the algorithm and on the specific computer on which the implementation is executed. Once these detailes are specified, the worst-case, best-case, and average-case running times become **functions** of the input size, n. It turns out that it is useful to look at a property of these functions that doesn't depend on the details, the so-called **growth rate** of the functions. The growth rate of a function has to do with the rate at which the value of the function changes as the size of its input increases. The growth rate gives the general shape of the function, rather than its specific value. The growth rate is most useful for comparing two functions. It will give us a way to compare the efficiency of two algorithms without worrying about the specific implementations or computers that we are using. We begin by looking at the growth rates of functions in the abstract; later, we apply what we learn to algorithms.

1.1 Growth Rates of Functions

Suppose that $f(n)$ and $g(n)$ are two functions, defined for non-negative integers n and with values in the set of real numbers. What would it mean to compare the "rates of growth" of the two functions? First of all, we are interested in what happens to the values of the functions as the input, n, gets larger and larger. So, for any given integer n_0, we can ignore the values of $f(n)$ and $g(n)$ for $n < n_0$. Furthermore, we want to ignore constant multiples; the functions $f(n)$, $1000 * f(n)$, and $0.0001 * f(n)$ are all considered to have the same rate of growth. (If the function gives the

running time of an algorithm, multiplying the function by a constant corresponds to running the algorithm on a faster or slower computer—a detail that we want to ignore.)

Taking this into consideration, we say that f has the same growth rate as g if there is a positive integer n_0 and two positive real constants c_1 and c_2 such that for any integer $n \geq n_0$, $c_1 * |g(n)| \leq |f(n)| \leq c_2 * |g(n)|$. Note that we don't require $f(n)$ to be a constant multiple of $g(n)$; we just require that it be bounded above and below by constant multiples of $g(n)$. In practice, all the functions that we will be interested in have positive values, at least for large values of n, so we can drop the absolute values from the above condition.

Symbolically, we express the fact that $f(n)$ and $g(n)$ have the same rate of growth by saying that $f(n)$ is $\Theta(g(n))$. The notation $\Theta(g(n))$ is read "Big-Theta of $g(n)$." Sometimes we will abuse notation, as is the custom, and write this as $f(n) = \Theta(g(n))$. The equal sign here does not represent a true equality, since the thing on the right-hand side is not a function; it is merely a notation that represents a certain property of $f(n)$.[1]

Looking at growth rates in this way is sometimes called **asymptotic analysis**, where the term "asymptotic" carries the connotation of "for large values of n." In the same way, we say that a property of a function $f(n)$ is asymptotically true if there is some number n_0 such that the property holds for all $n \geq n_0$. For example, we say that the function $f(n)$ is **asymptotically positive** if there is an integer n_0 such that $f(n) > 0$ for all $n \geq n_0$. All the functions that we will be looking at will be asymptotically positive. So, we have the following formal definition:

Definition 1.1. Let $f(n)$ and $g(n)$ be two asymptotically positive real-valued functions. We say that $f(n)$ is $\Theta(g(n))$ if there is an integer n_0 and positive real constants c_1 and c_2 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$.

For example, suppose that $f(n) = 3 * n^2 + 7n - 5$. Intuitively, for large values of n, $7n - 5$ is insignificant compared to $3n^2$, so we would expect that $f(n)$ is $\Theta(3n^2)$. In fact, since constant multiples are not significant, we expect that $f(n)$ is $\Theta(n^2)$. Proving this is not terribly informative: We could note that for $n \geq 1$, $7n - 5 \geq 0$ and so $3n^2 + 7n - 5 \geq 3n^2$. And we could note that for $n \geq 7$, $3n^2 + 7n - 5 \leq 3n^2 + 7n \leq 3n^2 + n^2 \leq 4n^2$. So we have, taking $g(n) = n^2$, that $3 * g(n) \leq f(n) \leq 4 * g(n)$ for all $n \geq 7$. Taking $n_0 = 7$, $c_1 = 3$, and $c_2 = 4$ in the definition, we have proved that $f(n)$ is $\Theta(n^2)$.

In practice, we will almost always work informally, using the two rules that a constant multiple can be ignored and that lower order terms in a summation can be ignored. Using these rules, we can say immediately, for example, that $7n^5 - 2n^3 + 18n^2$ is $\Theta(n^5)$.

1.2 Application to Algorithms

As a more practical example, we can do an asymptotic analysis of the running time of the selection sort algorithm. The total time it takes to run this algorithm can be computed by considering how many times each line in the algorithm is executed and how long it takes to run each line. For

[1] To be mathematically rigorous, we would define $\Theta(g(n))$ to be the set of *all* functions that satisfy the property. That is $\Theta(g(n)) = \{f(n) \mid \exists n_0\, \exists c_1\, \exists c_2\, (\forall n \geq n_0,\ c_1 * |g(n)| \leq |f(n)| \leq c_2 * |g(n)|)\}$. In that case, it would be proper to write $f(n) \in \Theta(g(n))$.

example, note that line 3 in the algorithm is in a loop that is executed $n-1$ times, where n is the size of the input list. So this line is executed $n-1$ times. If it takes c_3 seconds to execute the line, then it contributes a total of $c_3 * (n-1)$ seconds to the running time of the algorithm. Here, c_3 is a constant that depends, for example, on the speed of the computer on which the algorithm is run. Since c_3 is a constant multiple, its value won't, in the end, make any difference to the asymptotic analysis. Similarly, lines 8, 9, and 10, as well as the decrement and test operations in the for loop in line 2, are executed $n-1$ times and contribute terms of the form $c*(n-1)$ to the run time. We see that the run time for these lines is $\Theta(n)$.

On the other hand, line 5 is nested inside a second for loop and so is executed more often. The inner for loop is executed *top* times, where the value of the variable *top* is $n-1$ the first time the loop is executed, $n-2$ the second time, $n-3$ the third time, and so on. It follows that the total number of times that line 5 is executed is $(n-1)+(n-2)+\cdots+2+1$. The value of this summation is $\frac{n(n-1)}{2}$, or $\frac{1}{2}n^2 - \frac{1}{2}n$, and so the total running time contributed by line 5 is $c_2 * n^2 - c_2 * n$, where c_2 is some constant. Since we can ignore constant multiples and lower order terms, we can say that the total running time for line 5 is $\Theta(n^2)$. Similarly, the for loop operations in line 4 contribute a term of the form $c * n^2$, plus lower order terms, to the running time.

Line 6 is a little more interesting since it might or might not be executed in a particular case, depending on the result of the test in line 5. However, it cannot be executed any more times than line 5 is executed. In the best case, its contribution to running time is 0, and in the worst case, its contribution is $\Theta(n^2)$. Adding this to the run time for the rest of the nested for loop gives a total that is $\Theta(n^2)$ in both the best and worse case. Since the rest of the algorithm has a run time that is $\Theta(n)$ and n is of lower order than n^2, we see that the running time of selection sort is $\Theta(n^2)$. This is true for the best case running time, the worst case running time, and also the average case running time.

We can do the same analysis much more simply by noting that line 5 in the selection sort algorithm is executed $\Theta(n^2)$ times and that other lines are executed an equal or smaller number of times. Also, the time it takes to execute this line once is a constant. Since we can ignore lower order terms and constant multiples, it follows immediately that the total running time is $\Theta(n^2)$.

As a second example, we consider another common sorting method, the *insertion sort* algorithm. This algorithm can be expressed by the following function:

```
 1.  void insertion_sort( int[] list, int n ) {
 2.     for ( int next = 1; next < n; next++ ) {
 3.        int value = list[next];
 4.        int pos = next;
 5.        while ( pos > 0 && list[pos-1] > value ) {
 6.           list[pos] = list[pos-1];
 7.           pos--;
 8.        }
 9.        list[pos] = value;
10.     }
11.   }
```

Here, as in selection sort, the main for loop is executed $n-1$ times, so that lines 2, 3, 4, and 9 contribute terms of order $\Theta(n)$ to the running time. However, the while loop in lines 5 through 8 can be executed anywhere between zero and *next* times, depending on the particular numbers in the list. We see that the best case running time for insertion sort occurs when the while loop is executed zero times, giving a total running time that is $\Theta(n)$. You can check that this happens when the algorithm is applied to a list that is already sorted into non-decreasing order. In the worst case, the condition "`list[pos-1] > value`" in the while loop is always true. You can check that this will happen if the list is originally in *strictly decreasing* order. In this worst case, the while loop is executed $(n-1)+(n-2)+\cdots+1$, or $\frac{n(n-1)}{2}$, times. This is $\Theta(n^2)$.

So for insertion sort, the best case and worst case running times have different orders, $\Theta(n)$ and $\Theta(n^2)$ respectively. What about the average case running time? It is **not** necessarily true in a given case that the average case running time for an algorithm is the simple average of the best case and the worst case. The average must be taken over all possible inputs, and this requires a separate analysis which is often more difficult than the best and worst cases. For example, if running time for the large majority of cases is close to the best case running time, then the average running time will be very close to the best case running time. For insertion sort, it turns out that the average case running time is $\Theta(n^2)$. The proof of this is not trivial, but it is based on the intuitive fact that on the average, the while loop test "`list[pos-1] > value`" is true about half the time, so that on average, the number of executions of the while loop is one-half of the maximum number. Since the maximum number is $\frac{n(n-1)}{2}$, the average number of executions of the while loop is $\frac{n(n-1)}{4}$, which is still $\Theta(n^2)$.

We turn next to the problem of *searching* a list for a specified item. The obvious way to do this is to examine each item in the list in turn until the desired item is found. This is the *linear search* algorithm, which can be expressed as the function:

```
1.   int linear_search( int item, int[] list, int n ) {
2.      for ( int index = 0; index < n; index++ ) {
3.         if ( list[index] == item )
4.            return index;
5.      }
6.      return -1;
7.   }
```

In this function, the first parameter, *item*, is the item that we want to find. The number of items in the list is n. If the item is found in the list, then the return value of the function is its position in the list; if the item does not occur in the list, then the return value is -1. We take the size of the input to be n, the length of the list, and we investigate the best, worst, and average case running time as functions of n.

The best case occurs when the item occurs as the first item in the list. In that case, only a small, constant amount of work is done. We can say that the best case running time is $\Theta(1)$.[2] The

[2]To say that a function $f(n)$ is $\Theta(1)$ means technically that there are constants c_1 and c_2 and an integer n_0 such that $c_1 \leq f(n) \leq c_2$ for all $n \geq n_0$. That is, the function is bounded above and below by constants. In this case, the best case running time is actually constant, which is a stronger than saying that it is $\Theta(1)$.

worst case occurs when the item is not in the list. In that case, the for loop executes n times, so the worst case running time is $\Theta(n)$. If we assume that each item in the list has an equal chance of being searched for, then the for loop is executed an average of $n/2$ times for items in the list, giving an average case running time of $\Theta(n)$ for items in the list. Allowing for items that are not in the list would raise the average running time, but the average would still be $\Theta(n)$.

If we assume that the list is sorted into non-decreasing order, we can improve on this running time by using the *binary search* algorithm instead of linear search. The idea of binary search is this: Examine the item in the middle of the list. If that item is the item that you are searching for, return the position of the item. Otherwise, if the item in the middle of the list is greater than the item you are searching for, then apply the same search procedure to the first half of the list. And if the the item in the middle of the list is less than item you are searching for, then apply the same search procedure to the second half of the list. Continue in this way until you have found the item or reduced the size of the list to zero. Although the description of this algorithm is recursive, it can be implemented in an iterative version:

```
 1.   int binary_search( int item, int[] sorted_list, int n ) {
 2.      int low = 0;    // minimum possible position of item
 3.      int high = n-1; // maximum possible position of item
 4.      while ( high >= lo ) {
 5.         int middle = (high + lo) / 2;
 6.         if ( sorted_list[middle] == item )
 7.            return middle;
 8.         else if ( sorted_list[middle] > item )
 9.            high = middle - 1;
10.         else
11.            low = middle + 1;
12.      }
13.      return -1;
14.   }
```

In the best case, of course, the item is found immediately, and the amount of work that is done is constant. So, the best case running time is $\Theta(1)$. The worst case occurs when the item is not in the list. How many times is the while loop executed in this worst case? Each time through the loop, the difference between *high* and *low* is reduced to a little less than half its previous value. (The exact new value is $(int)((high - low - 1)/2)$.) When this difference is reduced to zero, the loop ends. Initially, $high - low$ is n. The question is, how many times do we divide this value by 2 before the answer becomes zero? Mathematically, the answer to this question is given by the integer part of the function $\log_2(n)$, the logarithm of n to the base 2. So, in the worst case, the while loop is executed about $\log_2(n)$ times, and the worst case running time is $\Theta(\log_2(n))$. For all but very small values of n, $\log_2(n)$ is much smaller than n, which means that binary search is much more efficient than linear search at least for sufficiently large lists.

1.3 Growth Rates of Common Functions

I've just said that an algorithm with run time $\Theta(\log_2(n))$ is "more efficient" than one with run time $\Theta(n)$, at least for large values of the input size, n. Remember that for a given function $f(n)$, there are many different functions that are $\Theta(f(n))$. These functions can differ by constant multiples and by lower order terms, for example. However, if $r(n)$ is **any** function that is $\Theta(\log_2(n))$ and $s(n)$ is **any** function that is $\Theta(n)$, $s(n)$ will be bigger than $r(n)$ for sufficiently large values of n. This is true because n grows so much faster than $\log_2(n)$ as n gets bigger and bigger.[3] We have to be a little careful: When we find that one algorithm is more efficient than another, at least for large values of n, the result could theoretically hold only for **very** large values of n. So, it's conceivable that an algorithm with $\Theta(n)$ run time is actually faster than one with $\Theta(\log_2(n))$ run time for all **practical** values of n, even though the $\Theta(\log_2(n))$ algorithm would be faster for large enough n. Nevertheless, as a general rule, a $\Theta(\log_2(n))$ algorithm is generally preferable to a $\Theta(n)$ algorithm.

In order to make this kind of judgment, we need to have some idea of the relative growth rates of the functions involved. Fortunately, we will encounter only a few general classes of functions.

A *power function* is one of the form n^a, where a is a positive constant. For example, n^2, n^{17}, and $n^{1.87}$ are power functions. The function n is a power function since $n = n^1$. The square root function, $\sqrt{n}$, is a power function since $\sqrt{n} = n^{1/2}$. Similarly, $n\sqrt{n}$ is the power function $n^{3/2}$. The rule for power functions is that whenever $a > b > 0$, the function n^a has a higher growth rate than n^b. For example, an algorithm with run time $\Theta(\sqrt{n})$ will run more slowly than one with run time $\Theta(n)$, for all sufficiently large values of n. Also, $\sqrt{n}$ is a "lower order term" when compared to n, so that the function $3n + 17\sqrt{n}$ can immediately be seen to be $\Theta(n)$.

An *exponential function* is one of the form b^n, where b is a positive constant. We will only be interested in exponential functions where $b > 1$. For example, 2^n, 10^n, and 1.7^n are exponential functions. The rule for exponential functions is that whenever $a > b > 0$, the function a^n has higher growth rate than b^n. Furthermore, if $b > 1$, then the function b^n has a higher growth rate than any power function. For example, an algorithm with a run time that is $\Theta(n^3)$ is faster than an algorithm with run time $\Theta(1.001^n)$ for all sufficiently large values of n. And we can say that the function $2^n + 3^n + n^2 + n^3$ is $\Theta(3^n)$.

A *logarithmic function* is one of the form $\log_b(n)$, where b is a positive constant. We will only be interested in logarithmic functions where $b > 1$. As it turns out, for any numbers a and b, both greater than 1, the functions $\log_a(n)$ and $\log_b(n)$ have exactly the same rate of growth. This follows from the mathematical identity $\log_a(n) = \frac{1}{\log_b(a)} \cdot \log_b(n)$. That is, the function $\log_a(n)$ is just a constant multiple of the function $\log_b(n)$. So, for the purposes of analyzing growth rates, we only need one logarithmic functions. In computer science, it is natural to use $\log_2(n)$, the logarithm to the base 2, which is defined by the fact that $\log_2(2^x) = x$ for any $x > 0$. In the future, the notation $\log(n)$ will always mean $\log_2(n)$. However, remember that any other logarithm function could be used instead. As for rates of growth, the logarithm function has a smaller rate of growth than any power function (and hence also of course a smaller rater of growth than any exponential function).

[3] More technically, for any positive constants a and b, there exists an integer n_0 such that $a * n > b * \log_2(n)$ for any $n \geq n_0$. (This is true even if a is very small and b is very large.) To prove this, you need to know some math; it follows from the fact that $\lim_{n\to\infty} \frac{n}{\log_2(n)} = 0$.

For example, $\log(n)$ grows more slowly than $\sqrt{n}$. Even though $\log(n)$ grows very slowly, it has a higher growth rate than the constant function. Occasionally, we might encounter the *iterated logarithm function*, $\log(\log(n))$, which has an even smaller rate of growth than $\log(n)$.

In addition to these basic functions, we can consider products and quotients of these functions. For example, the function $n\log(n)$ often turns up in the analysis of algorithms. This function has a higher growth rate than n but a much lower growth rate than n^2. In fact, it has a lower growth rate than $n^{1+\varepsilon}$ for any positive number ε. Similarly, $n/\log(n)$ has a lower growth rate than n and a higher growth rate than $n^{1-\varepsilon}$ for any positive number ε.

In summary, here is a list of some typical common functions that we might encounter in this course, arranged in order of increasing growth rate:

$$1 \quad \log(\log(n)) \quad \log(n) \quad \sqrt{n} \quad n \quad n\log(n) \quad n^2 \quad n^2\log(n) \quad n^3 \quad 2^n \quad n2^n \quad 3^n$$

1.4 Upper and Lower Bounds

To say that a function, $f(n)$, is $\Theta(g(n))$ is to say that for sufficiently large values of n, $f(n)$ is bounded above and below by constant multiples of $g(n)$. Sometime, it's useful to work separately with upper and lower bounds. For example, we might be able to show that the running time of a certain algorithm is less than some constant times n^2, for large enough values of n, without being able to show that it is greater than some other constant times n^2. In this case, we will say that the running time is $\mathcal{O}(n^2)$. The notation $\mathcal{O}$ is pronounced "Big-Oh." It indicates that we have found an upper bound but not necessarily a lower bound. It leaves open the possibility that the growth rate of the running time is actually **strictly less** than n^2. For example, the function $n\log(n)$ is $\mathcal{O}(n^2)$ but is not $\Theta(n^2)$.[4] Formally, we have the following definition:

Definition 1.2. Let $f(n)$ and $g(n)$ be two asymptotically positive real-valued functions. We say that $f(n)$ is $\mathcal{O}(g(n))$ if there is an integer n_0 and a positive real constant c such that $f(n) \leq c*g(n)$ for all $n \geq n_0$.

Similarly, we might have only a lower bound on the rate of growth. The is indicated by the notation $\Omega(g(n))$. The notation Ω is pronounced "Big-Omega". We have the definition:

Definition 1.3. Let $f(n)$ and $g(n)$ be two asymptotically positive real-valued functions. We say that $f(n)$ is $\Omega(g(n))$ if there is an integer n_0 and a positive real constant c such that $f(n) \geq c*g(n)$ for all $n \geq n_0$.

For example, if we say that the running time of an algorithm is $\Omega(n\log(n))$, we are claiming that the running time is at least as big as a constant times $n\log(n)$ for all sufficiently large values of n. This leaves open the possibility that it is, in fact, **much** bigger.

Note that a function $f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is **both** $\mathcal{O}(g(n))$ and $\Omega(g(n))$. A Big-Theta bound gives more information than either a Big-Oh or a Big-Omega bound, but $\mathcal{O}$ and Ω are still useful. Informally, a $\mathcal{O}$ bound on the running time of an algorithm gives you some idea about the maximum amount of time that you will need to run the algorithm while a Ω bound gives you some idea about the minimum amount of time needed.

[4] You should be aware that the notation $\mathcal{O}(f(n))$ is used by some people to mean what I am calling $\Theta(f(n))$.

1.5 Other Approaches

Asymptotic analysis (Θ, $\mathcal{O}$, and Ω) gives some idea about the running time of an algorithm, but there are other approaches to the analysis of algorithms. For example, aside from some bookkeeping, sorting and search algorithms are often built from two basic operations: comparing two items and copying an item from one location to another. Counting the number of comparisons and/or copies used by the algorithm can often give a better idea of the efficiency of the algorithm than a simple Big-Theta analysis. And sometimes the best approach is *empirical*. That is, test the algorithm by executing it on a variety of inputs and measuring its performance. Throughout the course, we will use other approaches whenever they are appropriate, but asymptotic analysis will remain our primary tool.

Exercises

1. We have looked at selection sort and insertion sort. Another common sorting algorithm is *bubble sort*, which can be expressed as follows:

```
1.   void bubble_sort( int[] list, int n ) {
2.      for ( int top = n-1; top > 0; top-- ) {
3.         boolean swapped = false;
4.         for ( int i = 0; i < top; i++ ) {
5.            if ( list[i] > list[i+1] ) {
6.               swapped = true;
7.               int temp = list[i];
8.               list[i] = list[i+1];
9.               list[i+1] = temp;
10.           }
11.        }
12.        if ( swapped == false )
13.           break;
14.     }
15.  }
```

Give a Big-Theta analysis of the best case, worst case, and average case running times of the bubble sort algorithm. You should explain your reasoning, but you do not have to give formal proofs of your answers.

2. In mathematics, a *matrix* is a two-dimensional array of numbers. An $n \times n$ matrix is one that has n rows and n columns. There is an algorithm for multiplying two $n \times n$ matrices, giving another $n \times n$ matrix as output. It's not easy to express this algorithm in C++, because of the way C++ handles two-dimensional arrays, but it can be expressed in Java as follows:

```
1.   int[][]  multiply( int[][] A, int[][] B, int n ) {
2.      int[][] product = new int[n][n];
3.      for ( int row = 0; row < n; row++ ) {
4.         for (int col = 0; col < n; col++ ) {
5.            int sum = 0;
6.            for ( int k = 0; k < n; k++ ) {
```

```
 7.                sum = sum + A[row][k] * B[k][col];
 8.             }
 9.             product[row][col] = sum;
10.          }
11.       }
12.       return product;
13.    }
```

Because there are no if statements in this algorithm, its best case, worst case, and average case running times are identical. Give a Big-Theta analysis of the running time of this algorithm as a function of the array size n. Explain your reasoning.