4.2. Problems, Algorithms, and Programs

4.2.1. Problems, Algorithms, and Programs

4.2.1.1. Problems

Programmers commonly deal with problems, algorithms, and computer programs. These are three distinct concepts.

As your intuition would suggest, a ***problem*** is a task to be performed. It is best thought of in terms of inputs and matching outputs. A problem definition should not include any constraints on *how* the problem is to be solved. The solution method should be developed only after the problem is precisely defined and thoroughly understood. However, a problem definition should include constraints on the resources that may be consumed by any acceptable solution. For any problem to be solved by a computer, there are always such constraints, whether stated or implied. For example, any computer program may use only the main memory and disk space available, and it must run in a "reasonable" amount of time.

Problems can be viewed as functions in the mathematical sense. A ***function*** is a matching between inputs (the ***domain***) and outputs (the ***range***). An input to a function might be a single value or a collection of information. The values making up an input are called the ***parameters*** of the function. A specific selection of values for the parameters is called an ***instance*** of the problem. For example, the input parameter to a sorting function might be an array of integers. A particular array of integers, with a given size and specific values for each position in the array, would be an instance of the sorting problem. Different instances might generate the same output. However, any problem instance must always result in the same output every time the function is computed using that particular input.

This concept of all problems behaving like mathematical functions might not match your intuition for the behavior of computer programs. You might know of programs to which you can give the same input value on two separate occasions, and two different outputs will result. For example, if you type `date` to a typical Linux command line prompt, you will get the current date. Naturally the date will be different on different days, even though the same command is given. However, there is obviously more to the input for the date program than the command that you type to run the program. The date program computes a function. In other words, on any particular day there can only be a single answer returned by a properly running date program on a completely specified input. For all computer programs, the output is completely determined by the program's full set of inputs. Even a "random number generator" is completely determined by its inputs (although some random number generating systems appear to get around this by accepting a random input from a physical process beyond the user's control). The limits to what functions can be implemented by programs is part of the domain of ***Computability***.

4.2.1.2. Algorithms

An ***algorithm*** is a method or a process followed to solve a problem. If the problem is viewed as a function, then an algorithm is an implementation for the function that transforms an input to the corresponding output. A problem can be solved by many different algorithms. A given algorithm solves only one problem (i.e., computes a particular function).

The advantage of knowing several solutions to a problem is that solution **A** might be more efficient than solution B for a specific variation of the problem, or for a specific class of inputs to the problem, while solution B might be more efficient than **A** for another variation or class of inputs. For example, one sorting algorithm might be the best for sorting a small collection of integers (which is important if you need to do this many times). Another might be the best for sorting a large collection of integers. A third might be the best for sorting a collection of variable-length strings.

By definition, something can only be called an algorithm if it has all of the following properties.

1. It must be *correct*. In other words, it must compute the desired function, converting each input to the correct output. Note that every algorithm implements some function, because every algorithm maps every input to some output (even if that output is a program crash). At issue here is whether a given algorithm implements the *intended* function.
2. It is composed of a series of *concrete steps*. Concrete means that the action described by that step is completely understood --- and doable --- by the person or machine that must perform the algorithm. Each step must also be doable in a finite amount of time. Thus, the algorithm gives us a "recipe" for solving the problem by performing a series of steps, where each such step is within our capacity to perform. The ability to perform a step can depend on who or what is intended to execute the recipe. For example, the steps of a cookie recipe in a cookbook might be considered sufficiently concrete for instructing a human cook, but not for programming an automated cookie-making factory.
3. There can be *no ambiguity* as to which step will be performed next. Often it is the next step of the algorithm description. Selection (e.g., the `if` statement) is normally a part of any language for describing algorithms. Selection allows a choice for which step will be performed next, but the selection process is unambiguous at the time when the choice is made.
4. It must be composed of a *finite* number of steps. If the description for the algorithm were made up of an infinite number of steps, we could never hope to write it down, nor implement it as a computer program. Most languages for describing algorithms (including English and "pseudocode") provide some way to perform repeated actions, known as iteration. Examples of iteration in programming languages include the `while` and `for` loop constructs. Iteration allows for short descriptions, with the number of steps actually performed controlled by the input.
5. It must *terminate*. In other words, it may not go into an infinite loop.

4.2.1.3. Programs

We often think of a computer ***program*** as an instance, or concrete representation, of an algorithm in some programming language. Algorithms are usually presented in terms of programs, or parts of programs. Naturally, there are many programs that are instances of the same algorithm,

because any modern computer programming language can be used to implement the same collection of algorithms (although some programming languages can make life easier for the programmer). To simplify presentation, people often use the terms "algorithm" and "program" interchangeably, despite the fact that they are really separate concepts. By definition, an algorithm must provide sufficient detail that it can be converted into a program when needed.

The requirement that an algorithm must terminate means that not all computer programs meet the technical definition of an algorithm. Your operating system is one such program. However, you can think of the various tasks for an operating system (each with associated inputs and outputs) as individual problems, each solved by specific algorithms implemented by a part of the operating system program, and each one of which terminates once its output is produced.

4.3. Comparing Algorithms

4.3.1. Comparing Algorithms

4.3.1.1. Introduction

How do you compare two algorithms for solving some problem in terms of efficiency? We could implement both algorithms as computer programs and then run them on a suitable range of inputs, measuring how much of the resources in question each program uses. This approach is often unsatisfactory for four reasons. First, there is the effort involved in programming and testing two algorithms when at best you want to keep only one. Second, when empirically comparing two algorithms there is always the chance that one of the programs was "better written" than the other, and therefore the relative qualities of the underlying algorithms are not truly represented by their implementations. This can easily occur when the programmer has a bias regarding the algorithms. Third, the choice of empirical test cases might unfairly favor one algorithm. Fourth, you could find that even the better of the two algorithms does not fall within your resource budget. In that case you must begin the entire process again with yet another program implementing a new algorithm. But, how would you know if any algorithm can meet the resource budget? Perhaps the problem is simply too difficult for any implementation to be within budget.

These problems can often be avoided by using asymptotic analysis. Asymptotic analysis measures the efficiency of an algorithm, or its implementation as a program, as the input size becomes large. It is actually an estimating technique and does not tell us anything about the relative merits of two programs where one is always "slightly faster" than the other. However, asymptotic analysis has proved useful to computer scientists who must determine if a particular algorithm is worth considering for implementation.

The critical resource for a program is most often its running time. However, you cannot pay attention to running time alone. You must also be concerned with other factors such as the space required to run the program (both main memory and disk space). Typically you will analyze the *time* required for an *algorithm* (or the instantiation of an algorithm in the form of a program), and the *space* required for a *data structure*.

Many factors affect the running time of a program. Some relate to the environment in which the program is compiled and run. Such factors include the speed of the computer's CPU, bus, and peripheral hardware. Competition with other users for the computer's (or the network's) resources can make a program slow to a crawl. The programming language and the quality of code generated by a particular compiler can have a significant effect. The "coding efficiency" of the programmer who converts the algorithm to a program can have a tremendous impact as well.

If you need to get a program working within time and space constraints on a particular computer, all of these factors can be relevant. Yet, none of these factors address the differences between two algorithms or data structures. To be fair, if you want to compare two programs derived from two algorithms for solving the same problem, they should both be compiled with the same compiler and run on the same computer under the same conditions. As much as possible, the same amount of care should be taken in the programming effort devoted to each program to make the implementations "equally efficient". In this sense, all of the factors mentioned above should cancel out of the comparison because they apply to both algorithms equally.

If you truly wish to understand the running time of an algorithm, there are other factors that are more appropriate to consider than machine speed, programming language, compiler, and so forth. Ideally we would measure the running time of the algorithm under standard benchmark conditions. However, we have no way to calculate the running time reliably other than to run an implementation of the algorithm on some computer. The only alternative is to use some other measure as a surrogate for running time.

4.3.1.2. Basic Operations and Input Size

Of primary consideration when estimating an algorithm's performance is the number of ***basic operations*** required by the algorithm to process an input of a certain size. The terms "basic operations" and "size" are both rather vague and depend on the algorithm being analyzed. Size is often the number of inputs processed. For example, when comparing sorting algorithms the size of the problem is typically measured by the number of records to be sorted. A basic operation must have the property that its time to complete does not depend on the particular values of its operands. Adding or comparing two integer variables are examples of basic operations in most programming languages. Summing the contents of an array containing n integers is not, because the cost depends on the value of n (i.e., the size of the input).

Example 4.3.1

Consider a simple algorithm to solve the problem of finding the largest value in an array of nn integers. The algorithm looks at each integer in turn, saving the position of the largest value seen so far. This algorithm is called the *largest-value sequential search* and is illustrated by the following function:

```
// Return position of largest value in integer array A
```

```
static int largest(int[] A) {
  int currlarge = 0;              // Position of largest element
seen
  for (int i=1; i<A.length; i++) // For each element
    if (A[currlarge] < A[i])     //   if A[i] is larger
       currlarge = i;            //     remember its position
  return currlarge;              // Return largest position
}
```

Here, the size of the problem is `A.length`, the number of integers stored in array `A`. The basic operation is to compare an integer's value to that of the largest value seen so far. It is reasonable to assume that it takes a fixed amount of time to do one such comparison, regardless of the value of the two integers or their positions in the array.

Because the most important factor affecting running time is normally size of the input, for a given input size n we often express the time $\mathbf{T}$ to run the algorithm as a function of n, written as $\mathbf{T}(n)T(n)$. We will always assume $\mathbf{T}(n)T(n)$ is a non-negative value.

Let us call c the amount of time required to compare two integers in function `largest`. We do not care right now what the precise value of c might be. Nor are we concerned with the time required to increment variable i because this must be done for each value in the array, or the time for the actual assignment when a larger value is found, or the little bit of extra time taken to initialize `currlarge`. We just want a reasonable approximation for the time taken to execute the algorithm. The total time to run `largest` is therefore approximately cn, because we must make n comparisons, with each comparison costing c time. We say that function `largest` (and by extension, the largest-value sequential search algorithm for any typical implementation) has a running time expressed by the equation

$$\mathbf{T}(n)=cn.$$

This equation describes the growth rate for the running time of the largest-value sequential search algorithm.

Example 4.3.2

The running time of a statement that assigns the first value of an integer array to a variable is simply the time required to copy the value of the first array value. We can assume this assignment takes a constant amount of time regardless of the value. Let us call c_1 the amount of time necessary to copy an integer. No matter how large the array on a typical computer (given reasonable conditions for memory and array size), the time to copy the value from the first position of the array is always c_1. Thus, the equation for this algorithm is simply

$\mathbf{T}(n)=c_1,$

indicating that the size of the input n has no effect on the running time. This is called a ***constant running time***.

Example 4.3.3

Consider the following code:

```
sum = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    sum++;
```

What is the running time for this code fragment? Clearly it takes longer to run when n is larger. The basic operation in this example is the increment operation for variable `sum`. We can assume that incrementing takes constant time; call this time c_2. (We can ignore the time required to initialize `sum`, and to increment the loop counters `i` and `j`. In practice, these costs can safely be bundled into time c_2) The total number of increment operations is n_2n2. Thus, we say that the running time is $\mathbf{T}(n)=c_2n_2$.

4.3.1.3. Growth Rates

The ***growth rate*** for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows. The following figure shows a graph for six equations, each meant to describe the running time for a particular program or algorithm. A variety of growth rates that are representative of typical algorithms are shown.

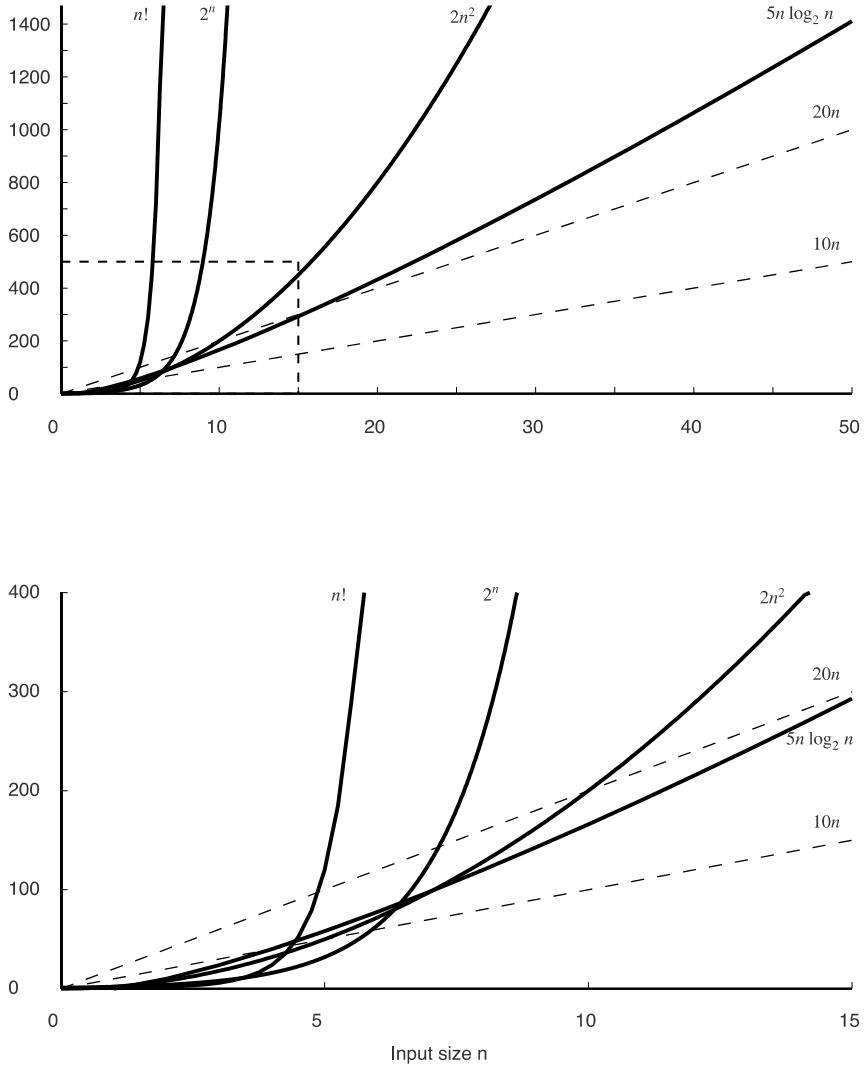

Figure 4.3.2: Two views of a graph illustrating the growth rates for six equations. The bottom view shows in detail the lower-left portion of the top view. The horizontal axis represents input size. The vertical axis can represent time, space, or any other measure of cost.

The two equations labeled $10n$ and $20n$ are graphed by straight lines. A growth rate of cn (for c any positive constant) is often referred to as a ***linear growth rate*** or running time. This means that as the value of n grows, the running time of the algorithm grows in the same proportion. Doubling the value of n roughly doubles the running time. An algorithm whose running-time equation has a highest-order term containing a factor of n^2 is said to have a ***quadratic growth rate***. In the figure, the line labeled $2n^2$ represents a quadratic growth rate. The line labeled 2^n represents an ***exponential growth rate***. This name comes from the fact that n appears in the exponent. The line labeled $n!$ also grows exponentially.

As you can see from the figure, the difference between an algorithm whose running time has cost $\mathbf{T}(n) = 10n$ and another with cost $\mathbf{T}(n) = 2n^2$ becomes tremendous as n grows. For $n > 5$, the algorithm with running time $\mathbf{T}(n) = 2n^2$ is already much slower. This is despite the fact that $10n$ has a greater constant factor than $2n^2$. Comparing the two curves marked $20n$ and $2n^2$ shows that changing the constant factor for one of the equations only shifts the point at which the two curves cross. For $n > 10$, the algorithm with cost $\mathbf{T}(n) = 2n^2$ is slower than the algorithm with cost $\mathbf{T}(n) = 20n$. This graph also shows that the equation $\mathbf{T}(n) = 5n \log n$ grows somewhat more quickly than both $\mathbf{T}(n) = 10n$ and $\mathbf{T}(n) = 20n$, but not nearly so quickly as the equation $\mathbf{T}(n) = 2n^2$. For constants $a, b > 1, n^a$ grows faster than either $\log^b n$ or $\log n^b$. Finally, algorithms with cost $\mathbf{T}(n) = 2^n$ or $\mathbf{T}(n) = n!$ are prohibitively expensive for even modest values of n. Note that for constants $a, b \geq 1, a^n$ grows faster than n^b.

We can get some further insight into relative growth rates for various algorithms from the following table. Most of the growth rates that appear in typical algorithms are shown, along with some representative input sizes. Once again, we see that the growth rate has a tremendous effect on the resources consumed by an algorithm.

Table 4.3.1

Costs for representative growth rates.

n	log log n	log n	n	n log n	n^2	n^3	2^n
16	2	4	2^4	$4 \cdot 2^4 = 2^6$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8 \cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10 \cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16 \cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	2^{64K}
1M	≈ 4.3	20	2^{20}	$20 \cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	2^{1M}
1G	≈ 4.9	30	2^{30}	$30 \cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	2^{1G}

4.4. Best, Worst, and Average Cases¶

When analyzing an algorithm, should we study the best, worst, or average case? Normally we are not interested in the best case, because this might happen only rarely and generally is too optimistic for a fair characterization of the algorithm's running time. In other words, analysis based on the best case is not likely to be representative of the behavior of the algorithm. However, there are rare instances where a best-case analysis is useful—in particular, when the best case has high probability of occurring. The ***Shellsort*** and ***Quicksort*** algorithms both can take advantage of the best-case running time of ***Insertion Sort*** to become more efficient.

How about the worst case? The advantage to analyzing the worst case is that you know for certain that the algorithm must perform at least that well. This is especially important for real-time applications, such as for the computers that monitor an air traffic control system. Here, it would not be acceptable to use an algorithm that can handle n airplanes quickly enough *most of the time*, but which fails to perform quickly enough when all n airplanes are coming from the same direction.

For other applications—particularly when we wish to aggregate the cost of running the program many times on many different inputs—worst-case analysis might not be a representative measure of the algorithm's performance. Often we prefer to know the average-case running time. This means that we would like to know the *typical* behavior of the algorithm on inputs of size n. Unfortunately, average-case analysis is not always possible. Average-case analysis first requires that we understand how the actual inputs to the program (and their costs) are distributed with respect to the set of all possible inputs to the program. For example, it was stated previously that the sequential search algorithm on average examines half of the array values. This is only true if the element with value K is equally likely to appear in any position in the array. If this assumption is not correct, then the algorithm does *not* necessarily examine half of the array values in the average case.

The characteristics of a data distribution have a significant effect on many search algorithms, such as those based on ***hashing*** and search trees such as the ***BST***. Incorrect assumptions about data distribution can have disastrous consequences on a program's space or time performance. Unusual data distributions can also be used to advantage, such as is done by ***self-organizing lists***.

In summary, for real-time applications we are likely to prefer a worst-case analysis of an algorithm. Otherwise, we often desire an average-case analysis if we know enough about the distribution of our input to compute the average case. If not, then we must resort to worst-case analysis.

4.6. Asymptotic Analysis and Upper Bounds¶

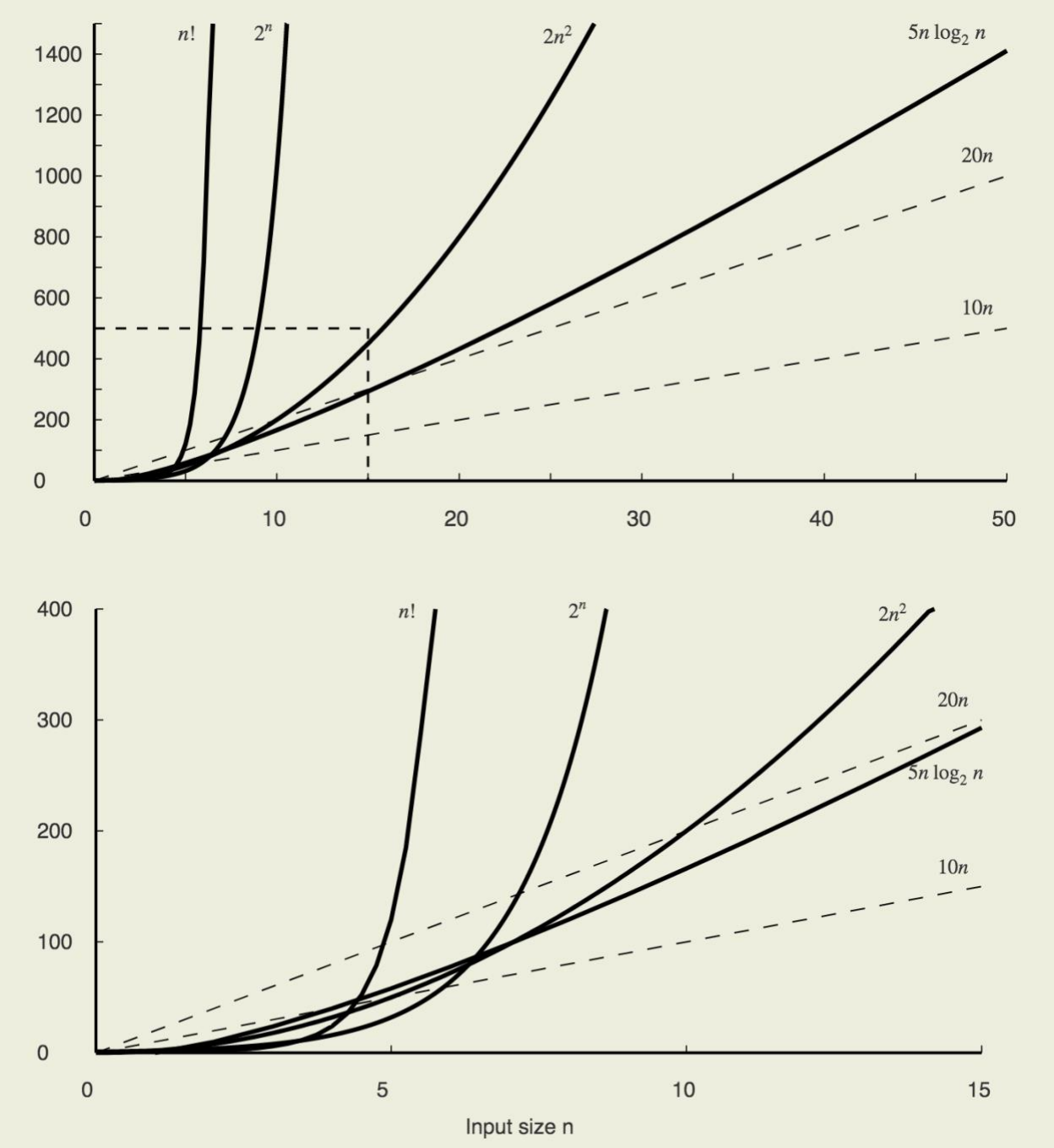

Figure 4.6.2: Two views of a graph illustrating the growth rates for six equations. The bottom view shows in detail the lower-left portion of the top view. The horizontal axis represents input size. The vertical axis can represent time, space, or any other measure of cost.

Despite the larger constant for the curve labeled $10n$ in the figure above, $2n^2$ crosses it at the relatively small value of $n=5$. What if we double the value of the constant in front of the linear

equation? As shown in the graph, $20n$ is surpassed by $2n^2$ once $n=10$. The additional factor of two for the linear ***growth rate*** does not much matter. It only doubles the x-coordinate for the intersection point. In general, changes to a constant factor in either equation only shift *where* the two curves cross, not *whether* the two curves cross.

When you buy a faster computer or a faster compiler, the new problem size that can be run in a given amount of time for a given growth rate is larger by the same factor, regardless of the constant on the running-time equation. The time curves for two algorithms with different growth rates still cross, regardless of their running-time equation constants. For these reasons, we usually ignore the constants when we want an estimate of the growth rate for the running time or other resource requirements of an algorithm. This simplifies the analysis and keeps us thinking about the most important aspect: the growth rate. This is called ***asymptotic algorithm analysis***. To be precise, asymptotic analysis refers to the study of an algorithm as the input size "gets big" or reaches a limit (in the calculus sense). However, it has proved to be so useful to ignore all constant factors that asymptotic analysis is used for most algorithm comparisons.

In rare situations, it is not reasonable to ignore the constants. When comparing algorithms meant to run on small values of nn, the constant can have a large effect. For example, if the problem requires you to sort many collections of exactly five records, then a sorting algorithm designed for sorting thousands of records is probably not appropriate, even if its asymptotic analysis indicates good performance. There are rare cases where the constants for two algorithms under comparison can differ by a factor of 1000 or more, making the one with lower growth rate impractical for typical problem sizes due to its large constant. Asymptotic analysis is a form of "back of the envelope" ***estimation*** for algorithm resource consumption. It provides a simplified model of the running time or other resource needs of an algorithm. This simplification usually helps you understand the behavior of your algorithms. Just be aware of the limitations to asymptotic analysis in the rare situation where the constant is important.

4.6.1.1. Upper Bounds

Several terms are used to describe the running-time equation for an algorithm. These terms—and their associated symbols—indicate precisely what aspect of the algorithm's behavior is being described. One is the ***upper bound*** for the growth of the algorithm's running time. It indicates the upper or highest growth rate that the algorithm can have.

Because the phrase "has an upper bound to its growth rate of $f(n)$ is long and often used when discussing algorithms, we adopt a special notation, called ***big-Oh notation***. If the upper bound for an algorithm's growth rate (for, say, the worst case) is (f(n)), then we would write that this algorithm is "in the set $O(f(n))$ in the worst case (or just "in $O(f(n))$ in the worst case"). For example, if n^2 grows as fast as $\mathbf{T}(n)$ (the running time of our algorithm) for the worst-case input, we would say the algorithm is in $O(n^2)$ in the worst case.

The following is a precise definition for an upper bound. $\mathbf{T}(n)$ represents the true running time of the algorithm. $f(n)$ is some expression for the upper bound.

For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in set $O(f(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \leq cf(n)$ for all $n > n_0$.

Constant n_0 is the smallest value of n for which the claim of an upper bound holds true. Usually n_0 is small, such as 1, but does not need to be. You must also be able to pick some constant c, but it is irrelevant what the value for c actually is. In other words, the definition says that for *all* inputs of the type in question (such as the worst case for all inputs of size n) that are large enough (i.e., $n>n_0$), the algorithm *always* executes in less than or equal to $cf(n)$ steps for some constant c.

If someone asked you out of the blue "Who is the best?" your natural reaction should be to reply "Best at what?" In the same way, if you are asked "What is the growth rate of this algorithm", you would need to ask "When? Best case? Average case? Or worst case?" Some algorithms have the same behavior no matter which input instance of a given size that they receive. An example is finding the maximum in an array of integers. But for many algorithms, it makes a big difference which particular input of a given size is involved, such as when searching an unsorted array for a particular value. So any statement about the upper bound of an algorithm must be in the context of some specific class of inputs of size n. We measure this upper bound nearly always on the best-case, average-case, or worst-case inputs. Thus, we cannot say, "this algorithm has an upper bound to its growth rate of n^2 because that is an incomplete statement. We must say something like, "this algorithm has an upper bound to its growth rate of n^2 *in the average case*".

Knowing that something is in $O(f(n))$ says only how bad things can be. Perhaps things are not nearly so bad. Because sequential search is in $O(n)$ in the worst case, it is also true to say that sequential search is in $O(n^2)$. But sequential search is practical for large n in a way that is not true for some other algorithms in $O(n^2)$. We always seek to define the running time of an algorithm with the tightest (lowest) possible upper bound. Thus, we prefer to say that sequential search is in O(n). This also explains why the phrase is in $O(f(n))$ or the notation "$\in O(f(n))$" is used instead of "is $O(f(n))$" or "$=O(f(n))$". There is no strict equality to the use of big-Oh notation. $O(n)$ is in $O(n^2)$, but $O(n^2)$ is not in $O(n)$.

4.6.1.2. Simplifying Rules¶

Once you determine the running-time equation for an algorithm, it really is a simple matter to derive the big-Oh expressions from the equation. You do not need to resort to the formal definitions of asymptotic analysis. Instead, you can use the following rules to determine the simplest form.

1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n) + f_2(n)$ is in $O(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

The first rule says that if some function $g(n)$ is an upper bound for your cost function, then any upper bound for $g(n)$ is also an upper bound for your cost function.

The significance of rule (2) is that you can ignore any multiplicative constants in your equations when using big-Oh notation.

Rule (3) says that given two parts of a program run in sequence (whether two statements or two sections of code), you need consider only the more expensive part.

Rule (4) is used to analyze simple loops in programs. If some action is repeated some number of times, and each repetition has the same cost, then the total cost is the cost of the action multiplied by the number of times that the action takes place.

Taking the first three rules collectively, you can ignore all constants and all lower-order terms to determine the asymptotic growth rate for any cost function. The advantages and dangers of ignoring constants were discussed near the beginning of this section. Ignoring lower-order terms is reasonable when performing an asymptotic analysis. The higher-order terms soon swamp the lower-order terms in their contribution to the total cost as (n) becomes larger. Thus, if T(n)=3n4+5n2, then T(n) is in O(n4). The ***n2*** term contributes relatively little to the total cost for large ***n***.

4.7. Lower Bounds and Θ Notation

4.7.1. Lower Bounds and Theta Notation

4.7.1.1. Lower Bounds

Big-Oh notation describes an upper bound. In other words, big-Oh notation states a claim about the greatest amount of some resource (usually time) that is required by an algorithm for some class of inputs of size n (typically the worst such input, the average of all possible inputs, or the best such input).

Similar notation is used to describe the least amount of a resource that an algorithm needs for some class of input. Like big-Oh notation, this is a measure of the algorithm's growth rate. Like big-Oh notation, it works for any resource, but we most often measure the least amount of time required. And again, like big-Oh notation, we are measuring the resource required for some particular class of inputs: the worst-, average-, or best-case input of size n.

The ***lower bound*** for an algorithm (or a problem, as explained later) is denoted by the symbol Ω, pronounced "big-Omega" or just "Omega". The following definition for Ω is symmetric with the definition of big-Oh.

For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \geq cg(n)$ for all $n > n_0$

Example 4.7.1

Assume $\mathbf{T}(n) = c_1 n^2 + c_2 n$ for c_1 and $c_2 > 0$. Then,

$$c_1 n^2 + c_2 n \geq c_1 n^2$$

for all $n > 1$. So, $\mathbf{T}(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

It is also true that the equation of the example above is in $\Omega(n)$. However, as with big-Oh notation, we wish to get the "tightest" (for Ω notation, the largest) bound possible. Thus, we prefer to say that this running time is in $\Omega(n2)$. Recall the sequential search algorithm to find a value K within an array of integers. In the average and worst cases this algorithm is in $\Omega(n)$, because in both the average and worst cases we must examine *at least* cn values (where c is 1/2 in the average case and 1 in the worst case).

[1] An alternate (non-equivalent) definition for Ω is

> $\mathbf{T}(n)$ is in the set $\Omega(g(n))$ if there exists a positive constant c such that $\mathbf{T}(n) \geq cg(n)$ for an infinite number of values for n.

This definition says that for an "interesting" number of cases, the algorithm takes at least $cg(n)$ time. Note that this definition is *not* symmetric with the definition of big-Oh. For $g(n)$ to be a lower bound, this definition *does not* require that $\mathbf{T}(n) \geq cg(n)$ for all values of n greater than some constant. It only requires that this happen often enough, in particular that it happen for an infinite number of values for n. Motivation for this alternate definition can be found in the following example.

Assume a particular algorithm has the following behavior:

$$\mathbf{T}(n) = \begin{cases} n & \text{for all odd } n \geq 1 \\ n^2/100 & \text{for all even } n \geq 0 \end{cases}$$

From this definition, $n^2/100 \geq \frac{1}{100} n^2$ for all even $n \geq 0$. So, $\mathbf{T}(n) \geq cn^2$ for an infinite number of values of n (i.e., for all even n) for $c = 1/100$. Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

For this equation for $\mathbf{T}(n)$, it is true that all inputs of size n take at least cn time. But an infinite number of inputs of size n take cn^2 time, so we would like to say that the algorithm is in $\Omega(n^2)$. Unfortunately, using our first definition will yield a lower bound of $\Omega(n)$ because it is not possible to pick constants c and n_0 such that $\mathbf{T}(n) \geq cn^2$ for all $n > n_0$. The alternative definition does result in a lower bound of $\Omega(n^2)$ for this algorithm, which seems to fit common sense more closely. Fortunately, few real algorithms or computer programs display the pathological behavior of this example. Our first definition for Ω generally yields the expected result.

As you can see from this discussion, asymptotic bounds notation is not a law of nature. It is merely a powerful modeling tool used to describe the behavior of algorithms.

4.7.1.2. Theta Notation

The definitions for big-Oh and Ω give us ways to describe the upper bound for an algorithm (if we can find an equation for the maximum cost of a particular class of inputs of size n) and the lower bound for an algorithm (if we can find an equation for the minimum cost for a particular class of inputs of size n). When the upper and lower bounds are the same within a constant factor, we indicate this by using Θ (big-Theta) notation. An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ *and* it is in $\Omega(h(n))$. Note that we drop the word "in" for Θ notation, because there is a

strict equality for two equations with the same Θ. In other words, if f(n) is Θ(g(n)), then g(n) is Θ(f(n)).

Because the sequential search algorithm is both in O(n) and in Ω(n) in the average case, we say it is Θ(n) in the average case.

Given an algebraic equation describing the time requirement for an algorithm, the upper and lower bounds always meet. That is because in some sense we have a perfect analysis for the algorithm, embodied by the running-time equation. For many algorithms (or their instantiations as programs), it is easy to come up with the equation that defines their runtime behavior. The analysis for most commonly used algorithms is well understood and we can almost always give a Θ analysis for them. However, the class of ***NP-Complete*** problems all have no definitive Θ analysis, just some unsatisfying big-Oh and Ω analyses. Even some "simple" programs are hard to analyze. Nobody currently knows the true upper or lower bounds for the following code fragment.

```
while (n > 1)
  if (ODD(n))
    n = 3 * n + 1;
   else
     n = n / 2;
```

While some textbooks and programmers will casually say that an algorithm is "order of" or "big-Oh" of some cost function, it is generally better to use Θ notation rather than big-Oh notation whenever we have sufficient knowledge about an algorithm to be sure that the upper and lower bounds indeed match. We prefer use Θ notation in preference to big-Oh notation whenever our state of knowledge makes that possible. Limitations on our ability to analyze certain algorithms may require use of big-Oh or Ω notations. In rare occasions when the discussion is explicitly about the upper or lower bound of a problem or algorithm, the corresponding notation will be used in preference to Θ notation.