3 Algorithms

Introduction

In this chapter, we supplement the discussion of algorithms presented in the text with their implementation in Maple. In Section 3.1, we discuss the process of turning step-by-step instructions describing a procedure and pseudocode for a procedure into Maple code. In the second section, we make use of Maple's graphing capabilities to visualize functions related by the big-O notation. In Section 3.3, we explore the average-case complexity of algorithms by considering the performance of a procedure on input.

In this chapter, please keep in mind the difference between an algorithm and its implementation (referred to as a procedure in Maple parlance). An algorithm refers to an approach to solving a particular problem, while a procedure is the implementation of the algorithm within Maple. In this manual, we will also distinguish between complexity and performance. Complexity is a measure of an algorithm and is generally measured by counting basic operations such as comparisons, while performance takes into account additional factors related to the specifics of an implementation and can be measured by recording the time it takes for the procedure to complete. Some of the factors affecting performance may include: choice of data structures and how the system implements those structures, the kinds of loops employed and how those are implemented by the computer language, and various improvements to efficiency handled by the system (for example, many computer languages, when evaluating a Boolean expression such as $p \wedge q$, will not bother checking q if p is found to be false thereby decreasing the number of operations that need to be performed).

3.1 Algorithms

It is impossible to overemphasize the importance and utility of writing either pseudocode or step-by-step instructions for an algorithm before you write the actual code for the procedure. Doing so helps you organize your ideas about how to solve the problem without the rigid constraints of the particular programming language. The textbook serves as an excellent model for you as you learn how to turn mathematical concepts and solutions to problems into algorithms. Those algorithms can then be turned into procedures in any programming language you choose. This manual will help you turn your pseudocode or step-by-step instructions for algorithms into procedures written in Maple.

Section 3.1 of the textbook describes several algorithms, with an emphasis on how you can describe these algorithms using both English descriptions and pseudocode. Here, we will see how to implement several of these algorithms in Maple. In this chapter, it will be especially important for you to have the text alongside you, as we will not reproduce the descriptions of the algorithms from the text.

Finding the Maximum

The first algorithm we will implement is the algorithm for finding the maximum in an unsorted sequence. This is described in Section 3.1 in the solution to Example 1 as step-by-step instructions and as pseudocode in Algorithm 1. We will build the procedure according to the step-by-step instructions. In order to make the connection between the instructions and the code as clear as possible, we begin with a procedure with no statements and successively revise it to show the addition of each step. Be warned that the incomplete versions of the procedure will produce errors if you attempt to execute them. In addition, provided that you have not turned off this feature in the system settings, Maple will highlight syntax errors in the code edit regions.

We begin with the basic elements of the procedure definition without code. Specifically, we include the assignment to the name of the procedure, the input with its type, and, since we will have local variables, the local keyword.

```
FindMax := proc(L::list(integer))
   local ;

end proc:
```

```
Error, ';' unexpected
```

Note that the parameter declaration indicates that **L** is the name we will use to refer to the parameter, which must be a list and that all of the elements of the list must be integers.

Step 1 in the step-by-step instructions is to “set the temporary maximum equal to the first integer” in the list. We declare a local variable to store the temporary maximum and add a statement in the body of the procedure to assign the first integer in the list to this value.

```
FindMax := proc(L::list(integer))
   local tempMax;
   tempMax := L[1];

end proc:
```

Step 2, according to Example 1 of the main text, is to compare the next integer to the temporary maximum and update the temporary maximum if necessary. This requires an **if** statement to make the comparison.

```
FindMax := proc(L::list(integer))
   local tempMax;
   tempMax := L[1];
   if tempMax < L[2] then
      tempMax := L[2];
   end if;

end proc:
```

(We are intentionally following the step-by-step instructions to the letter, so in step 2, “the next integer” is **L[2]**.)

Step 3 tells us to repeat step 2 for all of the integers in the list. We need to revise our code as follows. The fact that we are supposed to repeat step 2 means that we need a loop. Since this loop is supposed to consider all of the elements of the list beyond the first means that we use a **for** loop over the indices of the list starting at 2. We put the code we wrote for step 2 inside this loop, since that is the instruction being repeated, and replace the specific index “2” with the loop variable. Finally, the loop variable needs to be added to the **local** list.

```
FindMax := proc(L::list(integer))
   local tempMax, i;
   tempMax := L[1];
   for i from 2 to nops(L) do
      if tempMax < L[i] then
         tempMax := L[i];
      end if;
   end do;
end proc:
```

Finally, step 4 tells us that once the loop is completed, the value of the temporary maximum is the maximum, so we return that value.

```
FindMax := proc(L::list(integer))
   local tempMax, i;
   tempMax := L[1];
   for i from 2 to nops(L) do
      if tempMax < L[i] then
         tempMax := L[i];
      end if;
   end do;
   return tempMax;
end proc:
```

```
> FindMax([3, 18, −5, 72, 6, 0])
     72                                                    (3.1)
```

Admittedly, this example may be a bit too simple to warrant such an elaborate process. However, it illustrates an essential point, namely, that a well-written set of step-by-step instructions describing a procedure can be easily turned into working and correct code. Moreover, for nontrivial algorithms, the two-step process of writing instructions for the procedure and then implementing the procedure based on those instructions typically results in the production of a correct implementation more quickly than attempting the implementation without writing instructions.

Take a moment to compare the procedure above with the pseudocode given in Algorithm 1. You will notice that, in this example, they are extremely similar. This is one of the benefits of pseudocode in comparison to step-by-step instructions. However, step-by-step instructions are often easier to write as a first step toward a working procedure. Step-by-step instructions are also typically easier to understand, especially for novice programmers, as they can more easily accommodate explanation and other information that is out of place in pseudocode.

Binary Search

The second example is the binary search algorithm, presented as Algorithm 3 in Section 3.1. The previous example showed how you can use step-by-step instructions to build up the procedure. Starting from pseudocode, writing the final implementation involves translating statements in the pseudocode into legal statements in the programming language and filling in the missing details.

As before, we will go through several iterations as we translate the pseudocode in the text to actual Maple code. Initially, we need to make sure that the input is specified in a way appropriate for Maple; declare the local variables that are indicated in the pseudocode; and make basic syntax adjustments, specifically the **while** loops, **if** statements, and mathematical expressions such as $\lfloor i/2+j/2 \rfloor$ must be translated into their Maple counterparts. In this case, we turn the sequence $a_1,\ a_2,\ ..,\ a_n$ of integers specified as input in the text into a list **A**.

```
binarysearch := proc(x::integer, A::list(integer))
   local i, j, m, location;
   i := 1;
   j := n;
   while i < j do
      m := floor((i+j)/2);
      if x > A[m] then
         i := m + 1;
      else
         j := m;
      end if;
   end do;
   if x = A[i] then
      location := i;
   else
      location := 0;
   end if;
   return location;
end proc:
```

It is too much to expect this first version to properly execute.

```
> binarysearch(19, [1, 2, 3, 5, 6, 7, 8, 10, 12, 13, 15, 16, 18, 19, 20, 22])
```

```
Error, (in binarysearch) cannot determine if this expression is true or false: 1 < n
```

The problem is a simple one to correct. The pseudocode used n as the last index of the sequence of integers, so we need to make that assignment in our code.

```
binarysearch := proc(x::integer, A::list(integer))
   local n, i, j, m, location;
   n := nops(A);
   i := 1;
   j := n;
   while i < j do
```

```
      m := floor((i+j)/2);
      if x > A[m] then
        i := m + 1;
      else
        j := m;
      end if;
    end do;
    if x = A[i] then
      location := i;
    else
      location := 0;
    end if;
    return location;
end proc:
```

Now we try running it again:

```
> binarysearch(19, [1, 2, 3, 5, 6, 7, 8, 10, 12, 13, 15, 16, 18, 19, 20, 22])
    14                                                              (3.2)
```

Observe that the only commands that we added were the declaration of local variables, the assignment of *n*, and ending the loop and if statements. We also added additional line breaks to be consistent with the style of code used in this manual. Otherwise, the Maple code and pseudocode match very closely. It is, of course, not always quite this straightforward, but well-written pseudocode should contain all the essential elements. Like with proofs, as you become more familiar with pseudocode, you will find yourself more comfortable with leaving some details out.

Bubble Sort

We will next implement the bubble sort, presented in the text as Algorithm 4 of Section 3.1.

For our first attempt at implementing Algorithm 4, we need to specify the input, declare the local variables (which can, in part, be gleaned from the pseudocode), and correct the syntax for the **for** loops and **if** statements.

```
bubblesort := proc(A::list(realcons))
    local i, j;
    for i from 1 to n - 1 do
      for j from 1 to n - i do
        if A[j] > A[j+1] then
          # interchange A[j] and A[j+1]
        end if;
      end do;
    end do;
end proc:
```

(The **realcons** type is Maple's type for real numbers.)

The implementation above gets us close to a correct implementation of the bubble sort algorithm, but there are some problems. The simplest to fix is that **n** is not defined. You can correct this two ways. Either declare **n** as a local variable and set it equal to the number of elements of **A** or replace the two occurrences of **n** with **nops(A)**. There is little difference between these solutions.

```
bubblesort := proc(A::list(realcons))
   local i, j, n;
   n := nops(A);
   for i from 1 to n - 1 do
      for j from 1 to n - i do
         if A[j] > A[j+1] then
            # interchange  A[j] and A[j+1]
         end if;
      end do;
   end do;
end proc:
```

The next problem is the instruction in the pseudocode to interchange a_j with a_{j+1}. Maple contains no such command. Therefore, we will either have to make an **interchange** procedure that can be used within the **bubblesort** procedure or flesh out the code to interchange the two list elements within **bubblesort** itself. We take the first approach here so as to preserve as close a connection as possible between our implementation and the pseudocode.

Before creating an **interchange** procedure, however, there is another issue to take into consideration. Namely, in Maple, within a procedure, you cannot make assignments to an argument. Instead, we need to copy the parameter to a local variable, both in **bubblesort** and in the **interchange** procedure we are about to write.

Our **interchange** procedure will take two arguments: the list of numbers and the index of the smaller of the two positions to be swapped. It will proceed as follows:
1. Set a temporary variable equal to the first value to be swapped.
2. Set the value of the first position equal to the second value.
3. Set the value of the second position equal to the value stored in the temporary variable.

```
interchange := proc(L::list, i::posint)
   local M, temp;
   M := L;
   temp := M[i];
   M[i] := M[i+1];
   M[i+1] := temp;
   return M;
end proc:
```

Now, we finish our implementation of bubble sort by copying the list to a local name and replacing all the occurrences of the parameter with the local variable; applying the **interchange** procedure; and ending the procedure by explicitly returning the local copy of the list.

```
bubblesort := proc(A::list(realcons))
   local B, i, j, n;
   B := A;
   n := nops(B);
   for i from 1 to n - 1 do
      for j from 1 to n - i do
         if B[j] > B[j+1] then
            B := interchange(B,j);
         end if;
      end do;
   end do;
   return B;
end proc:
```

> $bubblesort([3, 18, -5, 72, 6, 0])$

$$[-5, 0, 3, 6, 18, 72] \qquad \textbf{(3.3)}$$

3.2 The Growth of Functions

In this section, we will use Maple to computationally explore the growth of functions. In particular, we will graph functions in order to visually convince ourselves that the big-O relationship is satisfied. We will also see how to use graphs to determine possible witnesses for the constants C and k in the definition of big-O notation. Since, as the textbook mentions, $f(x)$ is $O(g(x))$ if and only if $g(x)$ is $\Omega(f(x))$, the techniques we explore in this section apply also to big-Omega and big-Theta notation.

We begin by considering the function $f(x) = 5x^3 + 4x^2 + 3x + 9$. Theorem 1 from Section 3.2 tells us that this is $O\left(x^3\right)$, but we will use this function as an example of using Maple to find values for C and k such that $|f(x)| \leq C|g(x)|$ for all $k \leq x$.

The plot Command

We first look at options to the **plot** command that will be useful in this context. Start by giving names to the formulas.

> $f1 := 5x^3 + 4x^2 + 3x + 9$

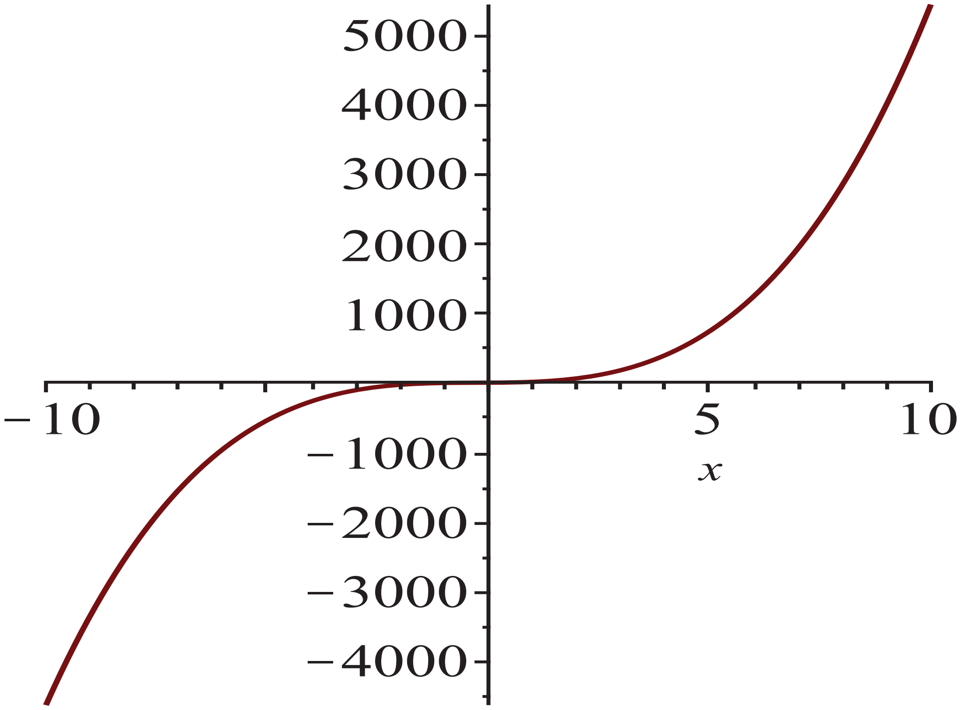
$$f1 := 5x^3 + 4x^2 + 3x + 9 \qquad \textbf{(3.4)}$$

> $g1 := x^3$

$$g1 := x^3 \qquad \textbf{(3.5)}$$

Graphing $f(x)$ can be as simple as the following.

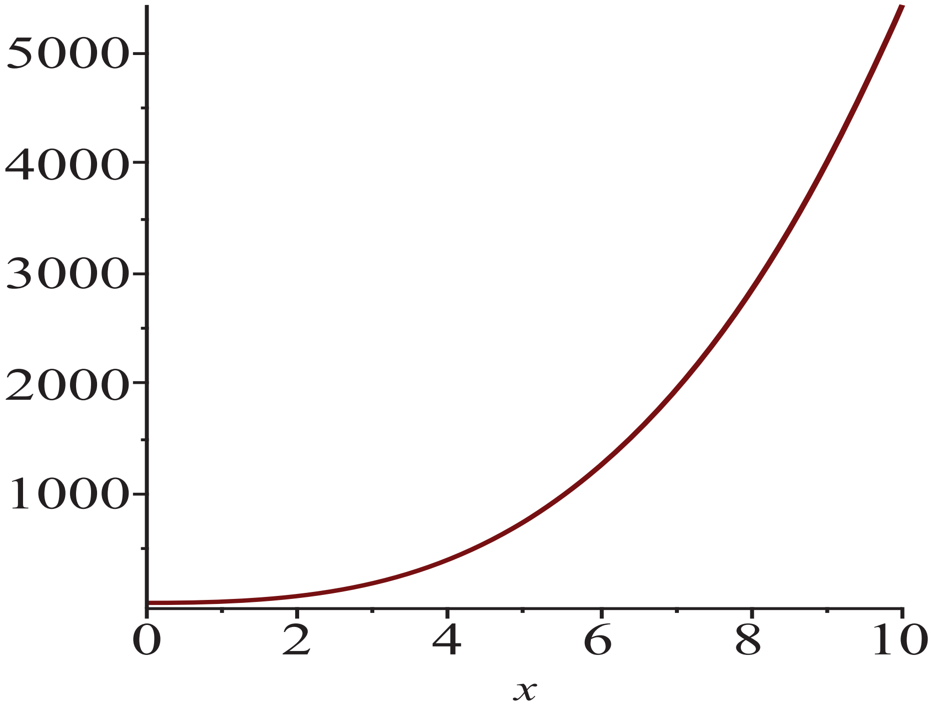
> $plot(f1, x)$



The first argument is the function in terms of an independent variable and the second argument is the name of the variable.

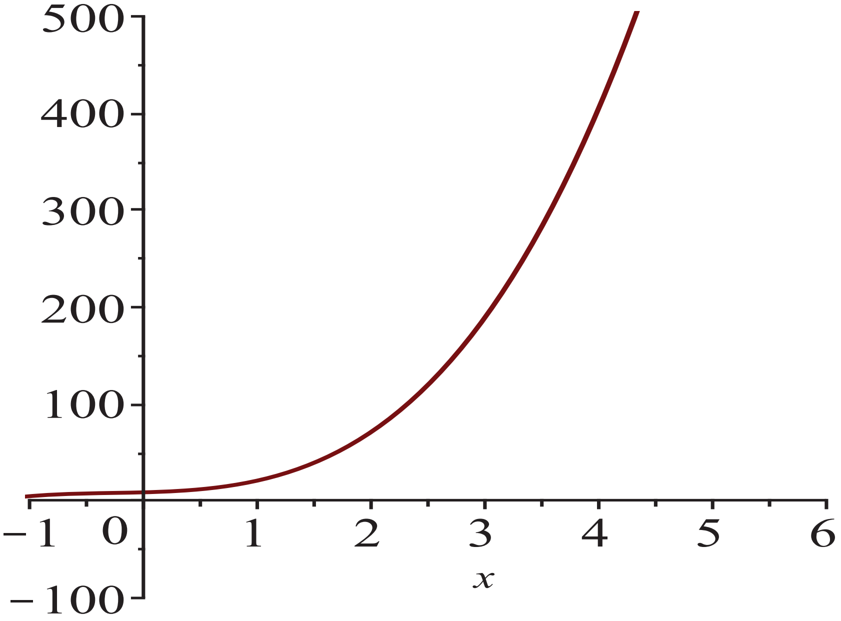
By default, Maple displays the graph with the horizontal axis ranging from −10 to 10. You can specify a different range of *x* values as follows:

```
> plot(f1, x = 0 ..10)
```



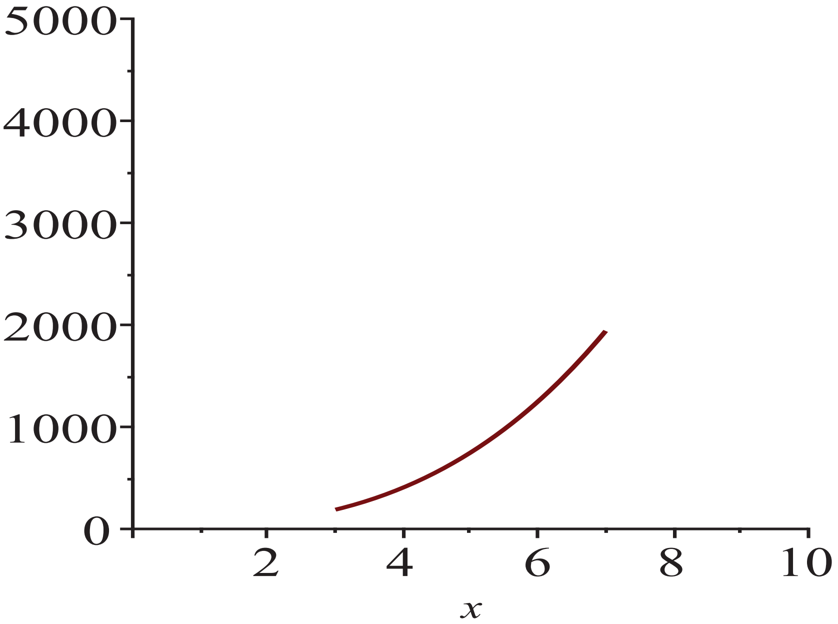
Note that Maple automatically selects the vertical range of the graph. You can control this with the **view** option. You use the view option by setting the keyword **view** equal to a two-element list. The first element of the list consists of the horizontal range to be displayed and the second member of the list is the vertical range to be displayed. For example, to display our graph with x ranging from −1 to 6 and y restricted between −100 and 500, enter the following command.

```
> plot(f1, x, view = [−1 ..6, −100 ..500])
```



It should be observed that, despite a somewhat similar effect, the **view** option is quite different from setting the range of the independent variable. We illustrate the difference with the following example.

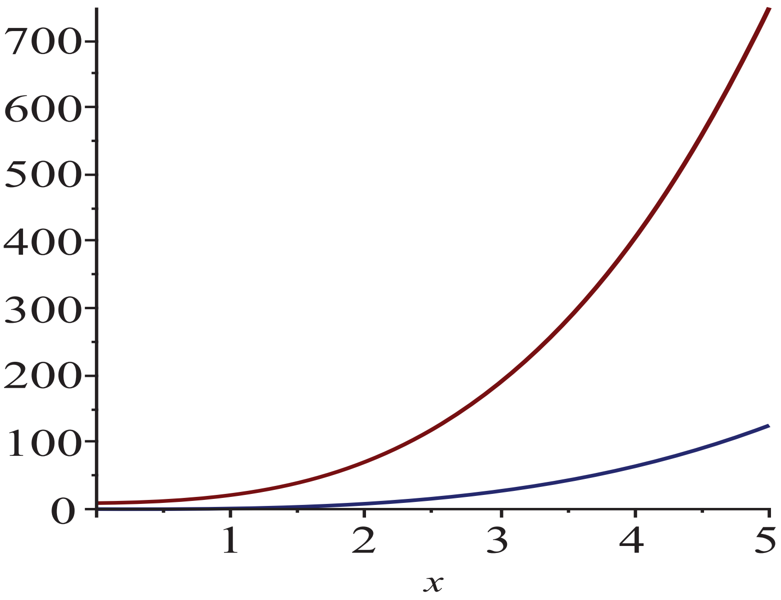
```
> plot(f1, x = 3 ..7, view = [0 ..10, 0 ..5000])
```



You see above that the **view** option is specifying the extent of the graph. On the other hand, setting the range **x=3..7** is actually a restriction of the domain of the function.

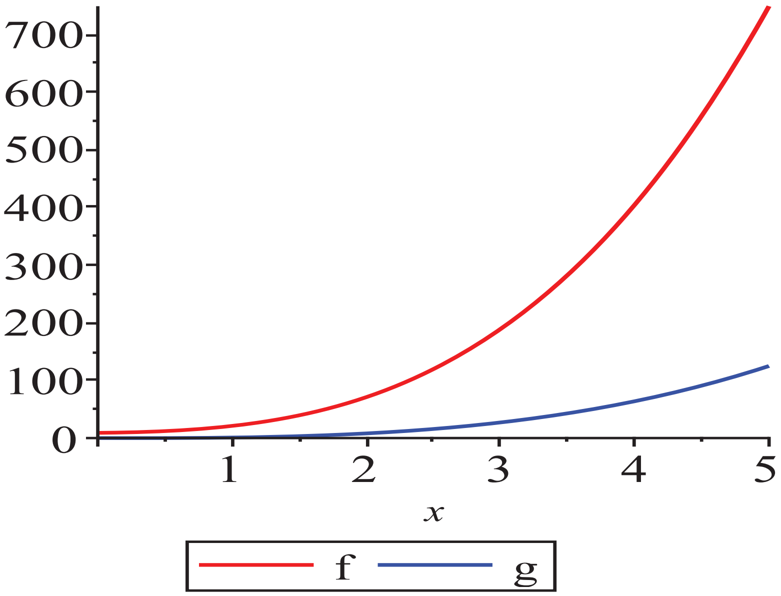
To plot multiple functions in the same graph, we merely issue the **plot** command with a list of the functions as the first argument.

> $plot([f1, g1], x = 0..5)$



Note that Maple automatically selects colors for the two functions. You can manually select the colors you want with the **color** option. If you set the **color** keyword equal to a list of color names, the first color is assigned to the first function, the second color to the second function, and so on. You can also create a legend by setting the **legend** keyword equal to a list of strings identifying the functions.

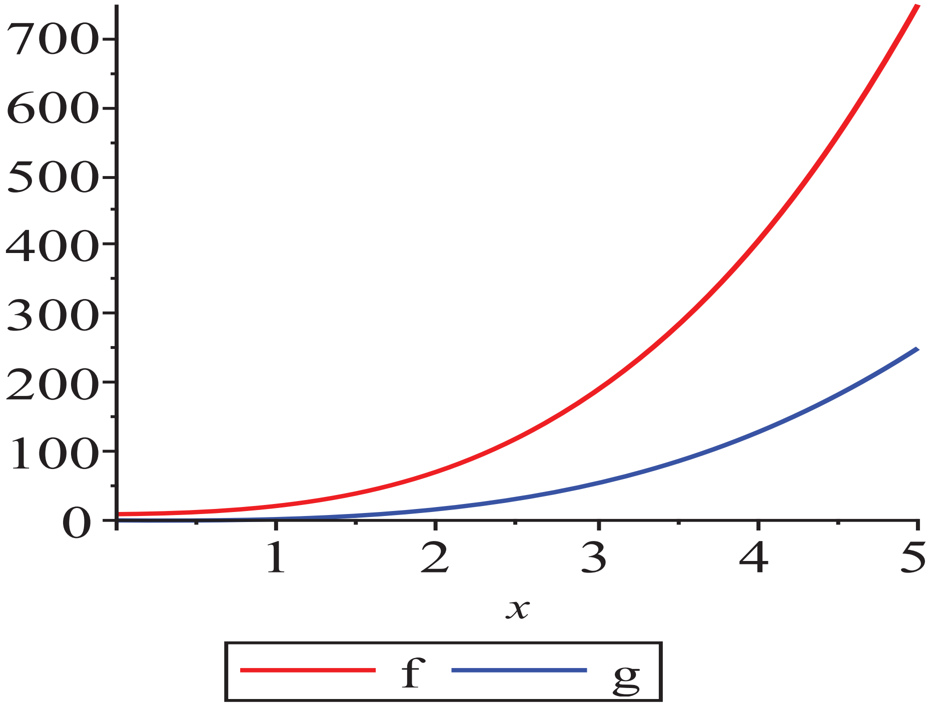
> $plot([f1, g1], x = 0..5, color = [red, blue], legend = [\text{“}f\text{”}, \text{“}g\text{”}])$



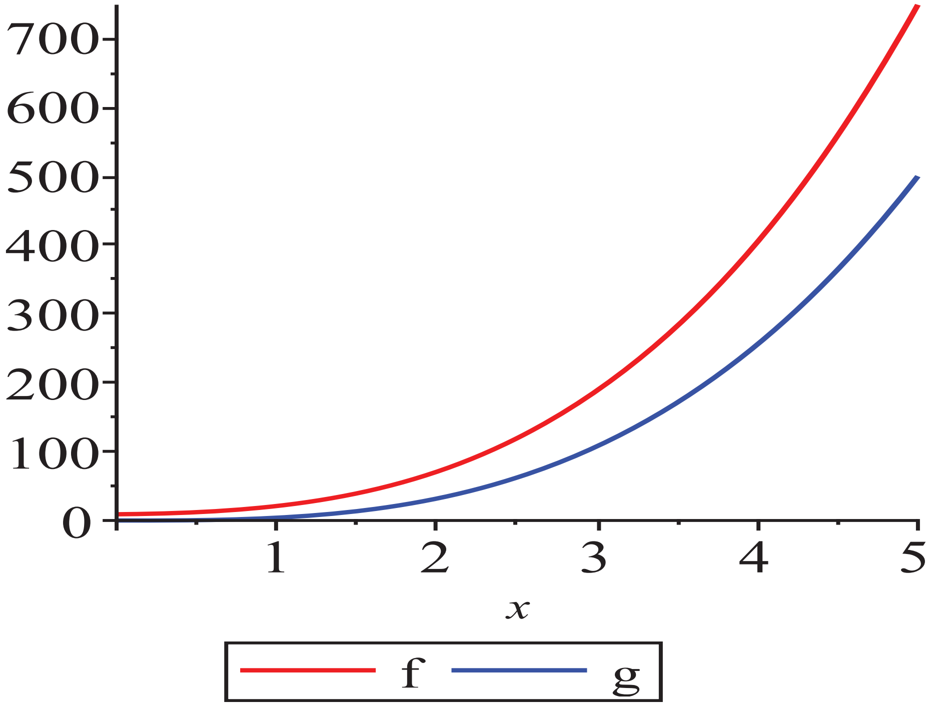
Finding Values for C and k

Now, we can start exploring different values of C for which the equation $f(x) \leq Cg(x)$ is satisfied. To do this, we just have to multiply **g1(x)** by different values within the list of functions. We will choose several values until we see a clear crossing.

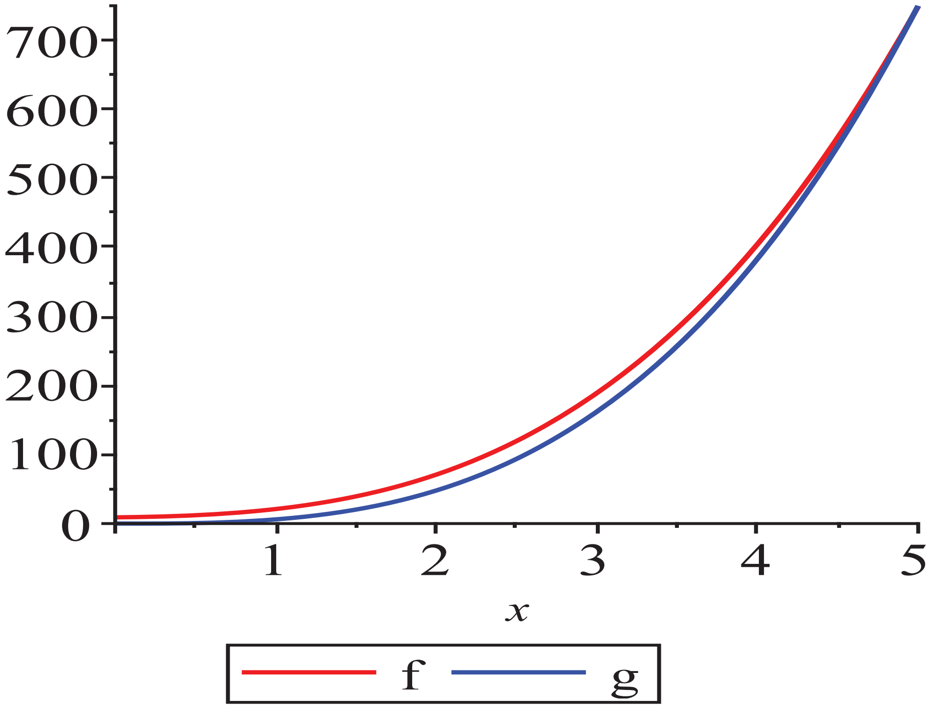
```
> plot([f1, 2 · g1], x = 0 ..5, color = [red, blue], legend = [“f”, “g”])
```



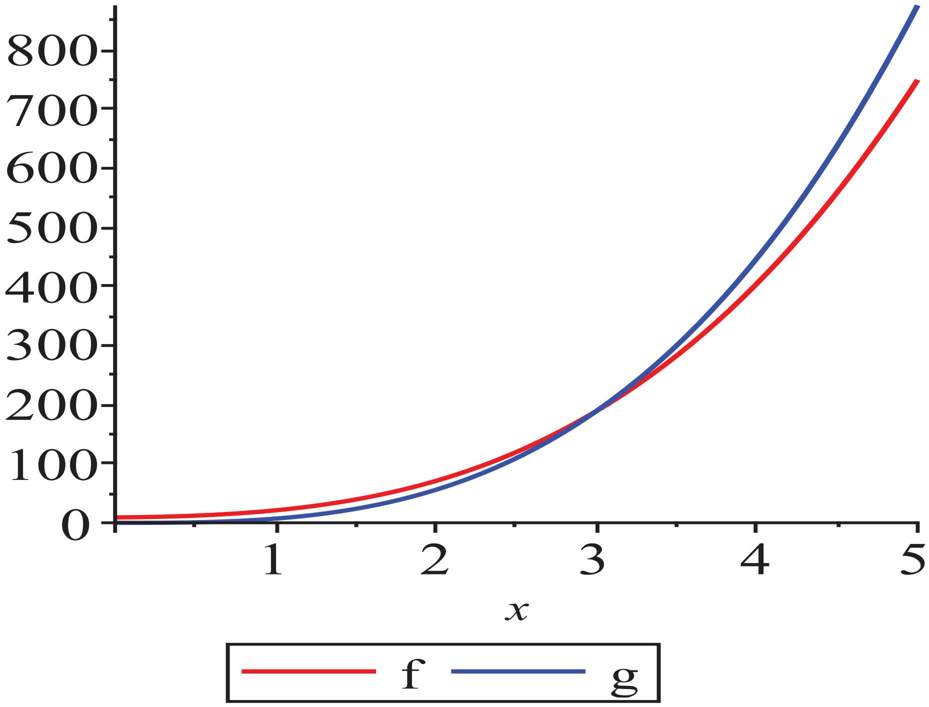
```
> plot([f1, 4 · g1], x = 0 ..5, color = [red, blue], legend = [“f”, “g”])
```



> $plot([f1, 6 \cdot g1], x = 0..5, color = [red, blue], legend = [“f”, “g”])$

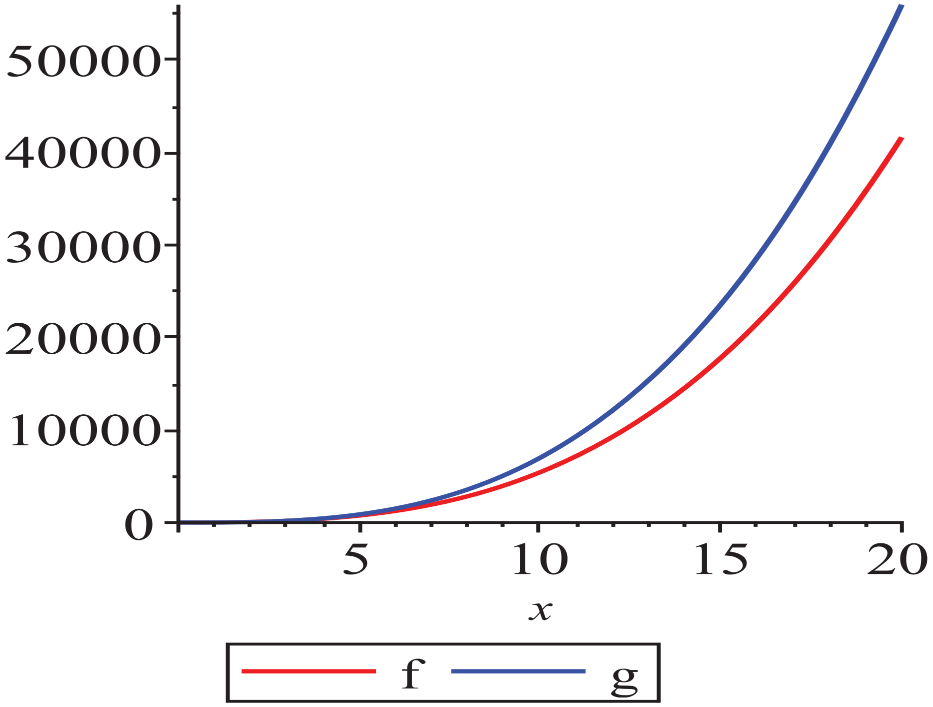


> $plot([f1, 7 \cdot g1], x = 0..5, color = [red, blue], legend = [“f”, “g”])$



By expanding the range of x-values, we can produce a graph that provides fairly convincing evidence that $C = 7$ and $k = 3$ witness for the assertion that $f(x)$ is $O\left(x^3\right)$.

> $plot([f1, 7 \cdot g1], x = 0..20, color = [red, blue], legend = [“f”, “g”])$



It is important to note that the graph above is not *proof* that $5x^3 + 4x^2 + 3x + 9$ is $O\left(x^3\right)$. A formal proof must follow the model provided by the examples in the text.

A Second Example

As a second example, consider $f(x) = 3x^5 + x^3 \ln\left(x^2 + 2x + 1\right)$. We claim that this is $O(x^n)$ for some value of n. We need to first determine the smallest value of n and then find witnesses for C and k. We assign a name for the formula for $f(x)$.

> $f2 := 3x^5 + x^3 log\left(x^2 + 2x + 1\right)$

$$f2 := 3x^5 + x^3 \ln\left(x^2 + 2x + 1\right) \tag{3.6}$$

We could proceed as in the previous example and display a selection of graphs comparing **f2** and x^n for different exponents in order to find a likely choice of n and then explore the coefficients. However, Maple's Exploration Assistant allows for a more interactive approach. There are two ways to create interactive elements in Maple. One is to enter a command that produces the output you wish to explore, but in terms of variables that have not been specified. For example:

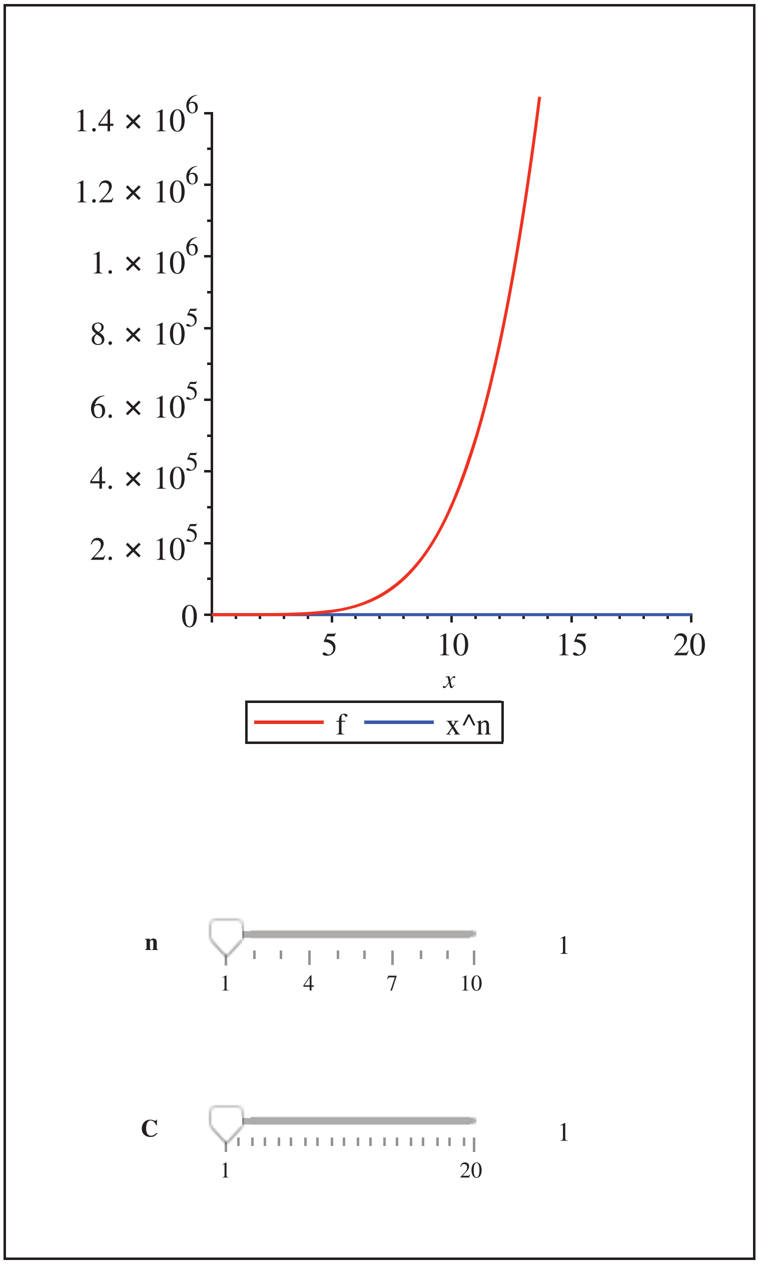
> $plot([f2, Cx^n], x = 0..20, color = [red, blue], legend = [\text{“}f\text{”}, \text{“}g\text{”}])$

Executing this command would produce an error (in the Maple version of this manual, it has been set to be not executable). However, if you right-click anywhere on the line above, one of the options in the menu that pops up should be “Explore.” Likewise, if you have the Context Panel open and the cursor is anywhere in the code for the plot, one of the options will be “Explore.” Clicking on “Explore” in either menu will cause a dialog window to open allowing you to specify certain options. Clicking on the “Explore” button at the bottom of that window will result in an interactive application with which you can explore the effects of the parameters.

The other approach is to type the commands to create the interactive exploration. This is the approach we will take in this manual, since it is more explicit and is typically easier to replicate. The name of the command used to create interactive explorations is **Explore**. The first argument is always a function call in terms of one or more names that will be altered by sliders or other controls, such as the call to **plot** above. If that is the only argument, then the result of executing it is to open the dialog window described in the previous paragraph.

If you don't want to use the dialog window, the simplest way to use **Explore** is to follow the command you wish to explore with equations setting the names of the parameters to ranges specifying possible values. For example, to allow the exponent to be integers from 1 to 10 and the constant between 1 and 20, we enter the following.

```
> Explore(plot([f2, Cx^n], x = 0..20, color = [red, blue],
    legend = ["f", "x^n"]), n = 1..10, C = 1..20)
```

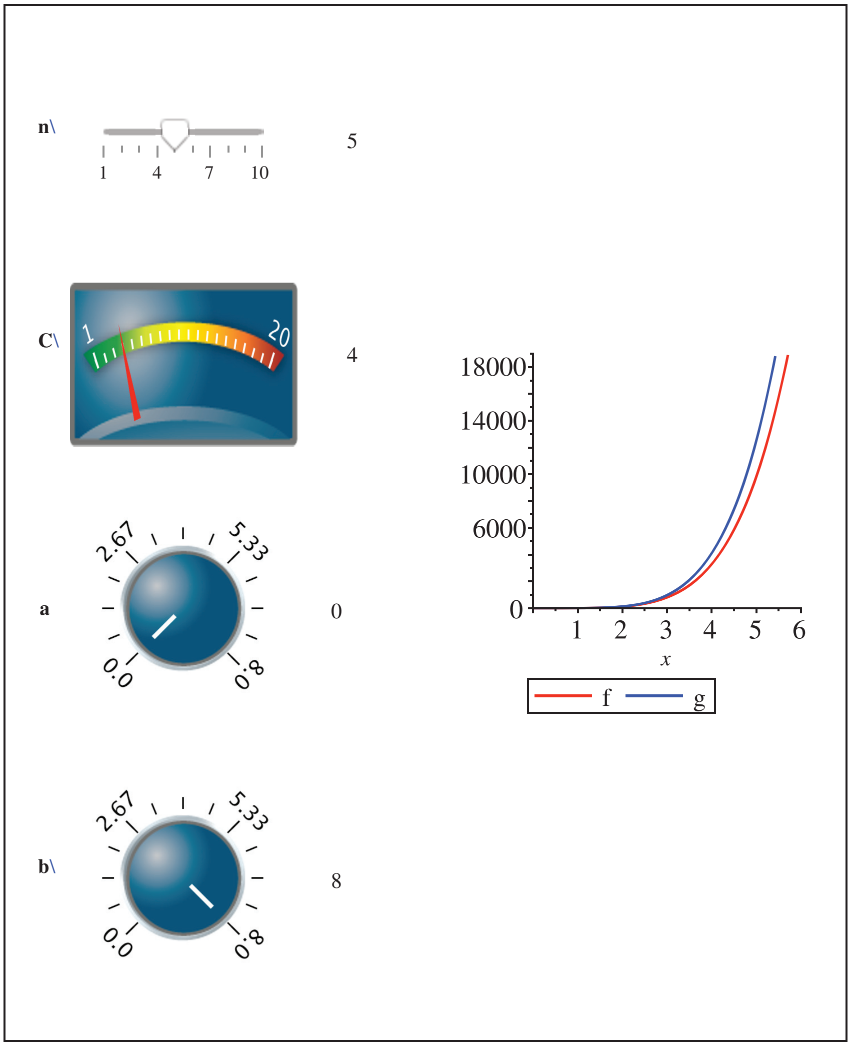


This is sufficient to find n for which **f2** is $O(x^n)$. Simply start dragging the slider for n. For $n \le 3$, x^n grows so slowly relative to $f(x)$ that it is virtually invisible on the graph. For $n = 4$, the graph of x^4 is visible, but clearly growing at a much slower rate than $f(x)$. For $n = 5$, however, there is some hope that multiplying by a constant may enable x^5 to catch up to $f(x)$. If you then start changing the value of C, you see that $C = 4$ appears sufficient for $4\,x^5$ to dominate $f(x)$.

To find a value for k, we will add parameters, a and b, for the endpoints of the range of x-values being plotted, which will effectively allow us to dynamically zoom in. Note that all that is required for a parameter to be allowed to take on noninteger values within its range is to give at least one of the endpoints as a noninteger. Keep in mind that for Maple, the number **8.** (with a decimal point) is a noninteger.

We will take this opportunity to also explore some of the options associated to an **Explore**. We first provide the command and then explain the options.

```
> Explore(plot([f2, Cx^n], x = a..b, color = [red, blue], legend = [“f”, “g”]),
    parameters = [n = 1..10, [C = 1..20, controller = meter],
    [a = 0..8.0, controller = dial], [b = 0..8.0, controller = dial]],
    initialvalues = [n = 5, C = 4, b = 8], placement = left,
     widthmode = pixels, width = 500)
```



First, rather than just providing the parameters and ranges, we use the **parameters** keyword in order to make some choices about how we interact with the parameters. The **parameters** keyword is set to a list. Each element in the list is the specification for a parameter. That specification can be a simple “parameter equals range,” or, in place of the range, a list. In the above, that is what was done for the parameter *n*, resulting in a slider. The other parameters are each specified by a sublist within the main **parameters** list. The first entry in each sublist is an equation identifying a parameter with a range or list of values. In addition, we have specified the **controller** for the parameter. The possible controllers are **slider**, **volumegauge**, **dial**, **meter**, **rotarygauge**, **checkbox**, **combobox**, **listbox**, and **textarea**. Note that some of these require the parameter be specified as having a list of values, rather than a range. For example, if you wish to have the exponent displayed as a combobox, also known as a drop-down menu, it would need to be specified as

```
[n=[$1..10], controller=combobox]
```

The **$** creates the sequence of integers and the brackets form the list of those values.

Second, after the list of lists specifying the parameters, we use the **initialvalues** option to specify values to which the parameters should be initially set. This option is set to a list of equations specifying the initial values, with any omitted parameters defaulting to the lowest value in their range or the first value in their list. This was an important option to use in this example; without it, both *a* and *b* would have started at 0, making the plot empty.

Third, we used the **placement** option to place the controllers to the left of the plot. Possible values are **left**, **right**, **bottom**, and **top**. Note that **placement** can also be used within the parameter specifications to position the controllers for different parameters separately.

Finally, the **width** option is used to specify the width of the exploration. The value is always a nonnegative real number, but the interpretation of the value depends on the **widthmode** option, which can be set to either **percentage** to specify the width as a percentage of the worksheet’s width or **pixels** to specify an absolute width.

Using the interactive application and by tightening the range of x values, you can see that $k = 1.5$ appears to be sufficient.

3.3 Complexity of Algorithms

Section 3.3 of the textbook emphasizes worst-case complexity and shows you how it can be deductively determined. The textbook also mentions average-case complexity and shows how to compute the average-case complexity of the linear search algorithm (Example 4).

Average-case complexity is typically more difficult to analyze deductively, but is still very important. From a practical standpoint, average-case complexity can help differentiate algorithms whose worst-case complexities are of the same order. Moreover, algorithms that have very poor worst-case complexity may have reasonable average-case complexity, provided that the “bad” inputs that produce the worst case are rare.

While average-case complexity is difficult to analyze, average-case performance can be computed fairly directly. Recall from the introduction to this chapter that we distinguish complexity of an algorithm from performance of a procedure. In this section, we will see how Maple can be used to analyze the average-case performance of procedures experimentally. We will use the **bubblesort** procedure developed in Section 3.1 of this manual as an example. Our goal will be to produce a graph displaying the empirically determined average-case time performance of the procedure.

It is essential to note that performance depends on many factors besides the algorithm, including the programming language used. In particular, different languages are designed with different purposes, leading to different efficiencies. This means, for example, that in comparing two algorithms, it is possible that one will outperform the other when implemented in one language and the reverse could be true if they are implemented in a different language.

We begin by explaining the standard approach to timing procedures in Maple. The **time** command returns the total CPU time that has been used by Maple since the start of the Maple session. By computing this value before running a procedure and again after the procedure has been completed, the difference in the values will be a very good estimate of the time taken to run the procedure.

Here is an example of the use of the **time** command.

```
> st := time() :
  FindMax([$1 ..100 000]) :
  time () − st
     0.075                                                        (3.7)
```

The name **st** stands for "start time." Recall that **[$1..100000]** produces the list consisting of the integers 1 through 100 000. Note that colons are used to suppress all the output except the final command that reports the elapsed time. This is important because if the procedure produces output, the display of the output takes time. We do not want the time it takes for Maple to display the results included as part of the time performance of the procedure.

Average Input

By average-case performance, we mean the average performance of a procedure on a random input selected from all possible inputs of the given size. The particulars of how the random input is selected is a necessary component in the analysis. It is natural to assume that each possible input will appear with the same likelihood as every other, but it is important to recognize that this may not always be the case. It may be that, in the circumstances under which the algorithm is intended to be used, some inputs may appear with relatively higher or lower frequency.

In our test of **bubblesort**, we have no particular application in mind and so will assume that all inputs are equally likely. In order to generate a random input, we will use the **randperm** command from the **combinat** package. Applied to a positive integer *n*, **randperm** will produce a list of the integers from 1 to *n* in random order. (We describe the commands of the **combinat** package in more detail in Chapter 6.)

```
> combinat[randperm] (20)
     [13, 4, 5, 20, 17, 7, 16, 10, 15, 14, 6, 3, 12, 8, 2, 11, 9, 19, 1, 18]     (3.8)
```

We can apply the **bubblesort** algorithm directly to the result and time how long it takes to execute.

```
> st := time() :
  bubblesort(combinat[randperm](100)) :
  time () − st
     0.022                                                          (3.9)
```

Since we are after average-case performance, we will need to execute **bubblesort** on some number, say 100, of different random inputs and average the time taken by each execution. To collect the 100 times, we can use a for loop to build a list in which the times are stored.

```
> timings := [ ] :
  for i to 100 do
     st := time( );
     bubblesort(combinat[randperm](100));
     et := time( ) − st;
     timings := [op(timings), et]
  end do :
```

```
> timings
     [0.078, 0.013, 0.014, 0.015, 0.072, 0.016, 0.016, 0.014, 0.070, 0.014,
        0.014, 0.015, 0.070, 0.015, 0.015, 0.014, 0.070, 0.014, 0.012,
        0.015, 0.070, 0.016, 0.012, 0.014, 0.070, 0.016, 0.014, 0.013,
        0.066, 0.015, 0.014, 0.014, 0.072, 0.016, 0.015, 0.013, 0.069,
        0.016, 0.015, 0.019, 0.076, 0.016, 0.014, 0.014, 0.015, 0.070,
        0.014, 0.014, 0.014, 0.070, 0.014, 0.014, 0.014, 0.069, 0.013,
        0.013, 0.014, 0.070, 0.014, 0.013, 0.014, 0.066, 0.012, 0.014,
        0.013, 0.068, 0.016, 0.015, 0.014, 0.071, 0.013, 0.014, 0.014,
        0.069, 0.014, 0.013, 0.015, 0.067, 0.014, 0.013, 0.013, 0.014,
        0.066, 0.015, 0.012, 0.015, 0.069, 0.012, 0.013, 0.015, 0.072,
        0.013, 0.013, 0.015, 0.069, 0.015, 0.013, 0.016, 0.067, 0.014]          (3.10)
```

(Depending on the speed of your computer, you may need to increase or decrease the size of the input list.)

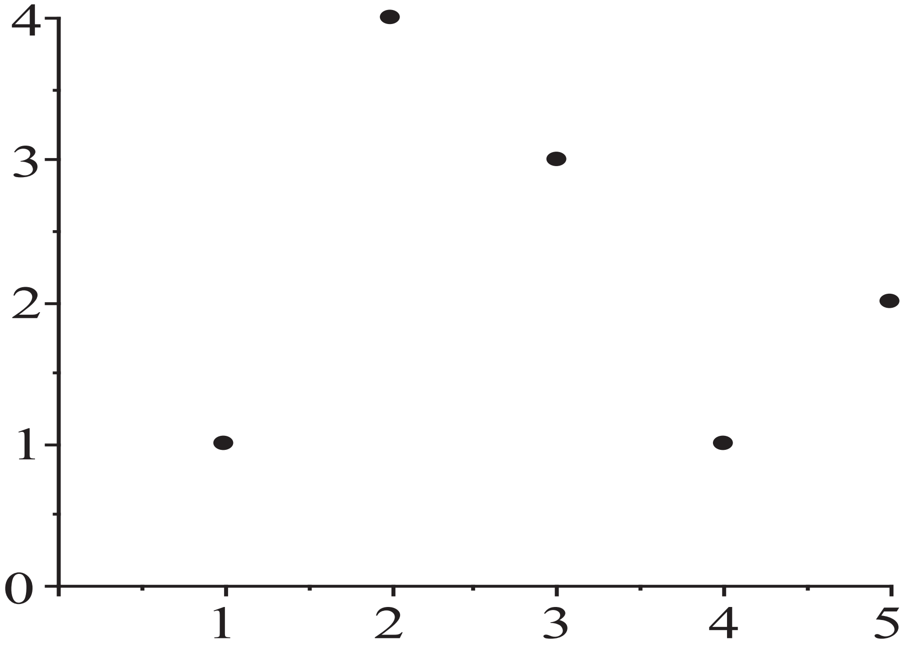
To average the times, we apply the **Mean** command from the **Statistics** package to the list of values.

```
> Statistics[Mean] (timings)
     0.0281000000000000                                             (3.11)
```

Graphing the Empirically Calculated Average-Case Complexity

To graph the average time data, we use the **plot** command. We saw above how to use **plot** to graph functions defined by formulas. We can also use **plot** to draw graphs by giving it specific values for the *x* and *y* coordinates. The coordinates must be given as two lists—the first list consisting of the *x* values and the second list the corresponding *y* values.

```
> plot ([1, 2, 3, 4, 5], [1, 4, 3, 1, 2], view = [0 ..5, 0 ..4], style = point,
     symbol = solidcircle, symbolsize = 15)
```



We have already seen the **view** option. The option **style=point** causes Maple to display only the points specified in the lists. Omitting **style=point** results in a graph in which the data are connected by straight line segments. The **symbol=solidcircle** determines the symbol used to plot the points. Other options include **asterisk**, **box**, **solidbox**, and **circle**. Finally, **symbolsize=15** increases the size of the dots to 15 points.

We will now write a procedure that produces the two lists required by **plot**. This procedure will accept no arguments, but will compute the average, over 100 trials, of the time taken to execute **bubblesort** on randomly generated lists of size 10, 20, 30, 40, and 50.

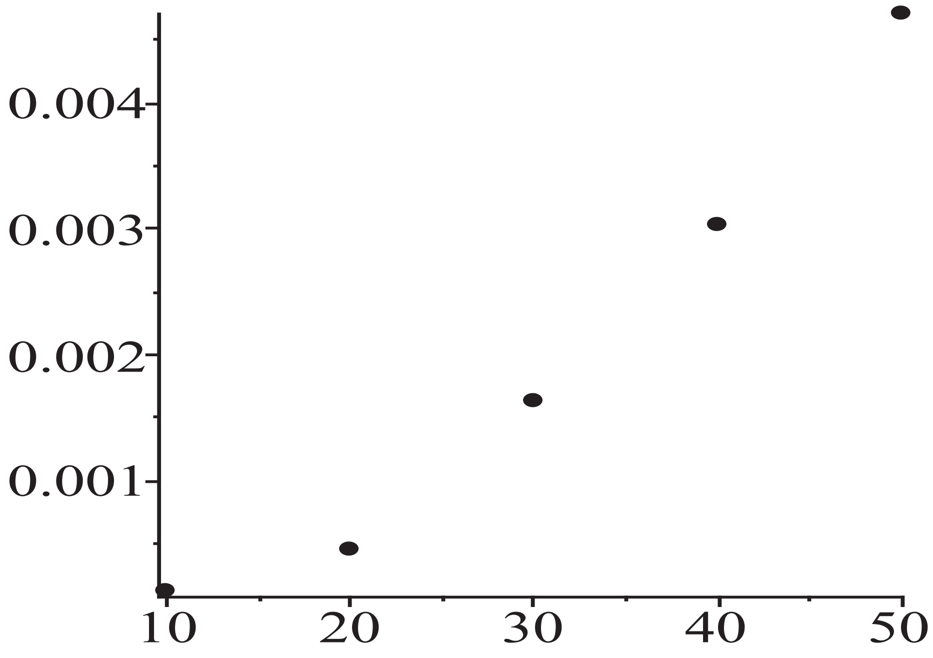
```
getTimes := proc()
   local sizes, s, avgTimes, times, trials, data, st, t, i;
   sizes := [seq(10*i,i=1..5)];
   avgTimes := [];
   for s in sizes do
      times := [];
      for trials from 1 to 100 do
         data := combinat[randperm](s);
         st := time();
         bubblesort(data);
         t := time() - st;
         times := [op(times),t];
      end do;
      avgTimes := [op(avgTimes),Statistics[Mean](times)];
   end do;
   return [sizes,avgTimes];
end proc:
```

The procedure begins by generating the list **sizes**. These values indicate the sizes of the lists to which **bubblesort** will be applied. The **avgTimes** list will hold the averages of the times for the corresponding input size. For each possible size, the procedure empties the **times** list and then runs 100 trials in which a randomly ordered list of the appropriate size is generated and the time it takes for **bubblesort** to sort the list is recorded and added to the **times** list. After the 100 trials are complete, the average of the times is computed and added to the **avgTimes** list. The procedure returns a two-element list consisting of the **sizes** list and the **avgTimes** list. Note that we moved the call to **randperm** to before the assignment of **st**, rather than calling **randperm** within the argument to **bubblesort**. This is so that the time it takes Maple to generate the random permutation is not counted as part of the time it takes **bubblesort** to execute.

We now apply the **getTimes** procedure and use its output to create a graph.

```
> getTimesOut := getTimes()
    getTimesOut := [[10, 20, 30, 40, 50], [0.000120000000000000,
       0.000450000000000000, 0.00163000000000000,
       0.00303000000000000, 0.00471000000000000]]                    (3.12)
```

```
> plot(getTimesOut[1], getTimesOut[2], style = point,
     symbol = solidcircle, symbolsize = 15)
```



From the shape of the graph, it appears that the average-case performance of **bubblesort** is polynomial. This suggests that the complexity of the algorithm is also polynomial. Of course, a proof of that fact would require an analysis of the kind given in Example 4 of Section 3.3.

The reader can modify the definition of the **sizes** list in order to produce finer detail (by decreasing the step between the input sizes) and to obtain data for larger input lists (by increasing the maximum value of **i**). However, note that for a list of length greater than 100, attempting to modify an element will produce an error. This is because lists in Maple are immutable meaning that changing a single element actually creates a new modified copy of the list, which can quickly become memory intensive. Therefore, the maximum possible list size that **bubblesort**, as currently written, can handle is 100. For larger sized inputs, you would need to rewrite **bubblesort** and **interchange** to uses **Array**s instead of lists.

We conclude with a caveat. The empirical testing we have done in this section is an example of a way to get an idea of the average-case performance of a procedure. It can be used to compare two or more algorithms with each other and can indicate major differences in worst-case and average-case complexity (for instance, in an algorithm with exponential worst-case complexity and polynomial average-case complexity). However, beyond generalities, the implementation of the algorithm, the computer running it, the computer language it is written in, and a host of other factors can play a sufficiently significant role that this approach is generally not helpful for making finer distinctions (e.g., between quadratic and cubic complexity).

The reader should refer to the solution of Computer Project 9 for a method of analyzing average case complexity that modifies the procedure in order to count the number of operations used with the input values.

Solutions to Computer Projects and Computations and Explorations

Computer Project 7

> Given two strings of characters use the naive string matching algorithm to determine whether the shorter string occurs in the longer string.

Solution: We begin, as usual, by following the pseudocode presented in the main text as Algorithm 6. As before, we will need to adjust basic syntax for the loops and other elements. However, the pseudocode indicates that the input should include two sequences of characters. That makes for a clunky user experience. We would much rather be able to enter

stringMatch[7, 3, “eceyeye”, “eye”]

rather than

stringMatch[7, 3, “e”, “c”, “e”, “y”, “e”, “y”, “e”, “e”, “y”, “e”]

We will implement the more user-friendly version by applying the **seq** command to transform a string into the sequence of its characters. We then wrap the sequence in brackets to produce the list of characters.

```
> [seq("eceyeye")]
    ["e", "c", "e", "y", "e", "y", "e"]                    (3.13)
```

Here is our first attempt.

```
stringMatch1 := proc(n::integer,m::integer,t::string,p::string)
   local T, P, s, j;
   T := [seq(t)];
   P := [seq(p)];
   for s from 0 to n-m do
      j := 1;
      while j <= m and T[s+j] = P[j] do
         j := j + 1;
```

```
        end do;
        if j > m then
            print(cat(s, " is a valid  shift "));
        end if;
    end do;
end proc:
```

Note that when **print** is applied to multiple arguments, it will print the comma-separated sequence. We apply **cat**, for concatenate, to merge the sequence into a single string.

Having written the function, we test it.

```
> stringMatch1(7, 3, “eceyeye”, “eye”)
      2 is a valid shift
      4 is a valid shift                                    (3.14)
```

It is often a good idea, having written a working procedure, to reflect on it and think about whether it could be made simpler. In this case, you might start wondering why the lengths of the text and pattern are included as arguments to the function. Remember that the syntax for executing this function suggested by the pseudocode is

stringMatch[7, 3, “e”, “c”, “e”, “y”, “e”, “y”, “e”, “e”, “y”, “e”]

With the text and pattern given as individual characters, the lengths are necessary in order to determine where the text stops and the pattern begins. In our implementation, the text and pattern are separate arguments, so the lengths can be computed by the function rather than entered by the user. Our final version of the naive string matcher is below.

```
stringMatch := proc(t::string, p::string)
    local T, n, P, m, s, j;
    T := [seq(t)]; n := numelems(t);
    P := [seq(p)]; m := numelems(p);
    for s from 0 to n-m do
        j := 1;
        while j <= m and T[s+j] = P[j] do
            j := j + 1;
        end do;
        if j > m then
            print(cat(s, " is a valid  shift "));
        end if;
    end do;
end proc:
```

```
> stringMatch(“eceyeye”, “eye”)
      2 is a valid shift
      4 is a valid shift                                    (3.15)
```

Maple, of course, includes extensive support for string manipulation, including a command that performs the same function as our naive string matcher called **SearchAll**, which is part of the

StringTools package. The arguments for this function are in the reverse order as ours, with the pattern first. The result is a sequence of the locations of the first characters of the matches or, if there is no match, the command returns 0.

```
> StringTools[SearchAll]("eye", "eceyeye")
      3, 5                                                        (3.16)
```

The difference in output is that the built-in function, rather than indicating valid shifts, returns the starting positions of where the pattern occurs in the text.

Computer Project 10

> Given an ordered list of n integers and an integer x in the list, find the number of comparisons used to determine the position of x in the list using a linear search and using a binary search.

Solution: There is no loss of generality to assume that the list of n integers is the list of integers from 1 to n.

For the linear search algorithm provided as Algorithm 2 in Section 3.1 of the text, each step in the search requires 2 comparisons, one tests whether the end of the list has been reached and one tests whether the current element is the element being searched for. These are both contained in the Boolean expression that controls the while loop. A final comparison is used after the while loop is completed to determine whether the element was found or not. In the list of n integers 1 through n, the integer x is therefore found after $2x + 1$ comparisons.

To determine the number of comparisons needed to find x via the binary search algorithm, we modify the procedure we wrote in Section 3.1 of this manual to count comparisons. For reference, here is the original **binarysearch** procedure.

```
binarysearch := proc(x::integer,A::list(integer))
   local n, i, j, m, location;
   n := nops(A);
   i := 1;
   j := n;
   while i < j do
      m := floor((i+j)/2);
      if x > A[m] then
         i := m + 1;
      else
         j := m;
      end if;
   end do;
   if x = A[i] then
      location := i;
   else
      location := 0;
   end if;
   return location;
end proc:
```

We will modify this procedure to count comparisons. Each time through the while loop accounts for two comparisons, the $i < j$ comparison that controls the loop and the $a_m < x$ comparison in the if statement. Thus, we add a line of code to increment the comparison count by two at the start of the while loop. In addition, we need to add one to the comparison count after the end of the loop to account for the comparison that terminates the loop. One final comparison is done to determine if the search has succeeded or not.

The modified procedure returns the count of comparisons instead of the position of the element.

```
binarysearchC := proc(x::integer, A::list(integer))
   local n, i, j, m, location, count;
   count := 0;
   n := nops(A);
   i := 1;
   j := n;
   while i < j do
      count := count + 2;
      m := floor((i+j)/2);
      if x > A[m] then
         i := m + 1;
      else
         j := m;
      end if;
   end do;
   count := count + 1;
   if x = A[i] then
      location := i;
   else
      location := 0;
   end if;
   count := count + 1;
   return count;
end proc:
```

For example, to find 15 in the list from 1 to 20, it takes

```
> binarysearchC(15, [$1 ..20])
    10                                                        (3.17)
```

comparisons.

We can use the information above to compare the average number of comparisons required in a list of n elements. We need to determine the number of comparisons needed to find each value from 1 to n in the list from 1 to n and average these numbers of comparisons. For the linear search, we know that it takes $2x + 1$ comparisons, so the average can be found from

$$\frac{\sum_{x=1}^{n} 2x + 1}{n}.$$

We can use Maple's symbolic summation capabilities (discussed in Section 2.4 of this manual).

$$> \frac{sum(2\cdot x+1, x=1\,..n)}{n}$$

$$\frac{(n+1)^2-1}{n} \qquad \textbf{(3.18)}$$

$$> simplify\,(\%)$$

$$n+2 \qquad \textbf{(3.19)}$$

(The **simplify** command forces Maple to simplify expressions.)

For the binary search procedure, we can find the average by applying our procedure above to each integer in turn and taking the average. The following procedure will produce the average number of comparisons required for a given value of *n*.

```
binaryAvg := proc(n::posint)
   local comps, L, x;
   comps := [];
   L := [$1..n];
   for x from 1 to n do
      comps := [op(comps),binarysearchC(x,L)];
   end do;
   return Statistics[Mean](comps);
end proc:
```

For example, in the list from 1 to 20, it requires an average of 20 + 2 = 22 comparisons using the linear search, and an average of 10.8 comparisons using the binary search.

$$> binaryAvg\,(20)$$

$$10.8000000000000 \qquad \textbf{(3.20)}$$

Next, we graph the average number of comparisons as *n* ranges from 1 to 100. To do this, we first create the necessary inputs to the **plot** command. Recall from Section 3.2 of this manual that plot requires a list of the *x* values and a list of the *y* values. The *x* values will be the values of *n*.

$$> nList\ := [\$\,1\,..100]:$$

For the linear search algorithm, the *y* values are $n+2$.

$$> linearAverages\ := [seq(n+2, n=1\,..100)]$$

For the binary search algorithm, the *y* values are obtained from the procedure **binaryAvg**.

$$> binaryAverages\ := [seq(binaryAvg(n), n=1\,..100)]:$$

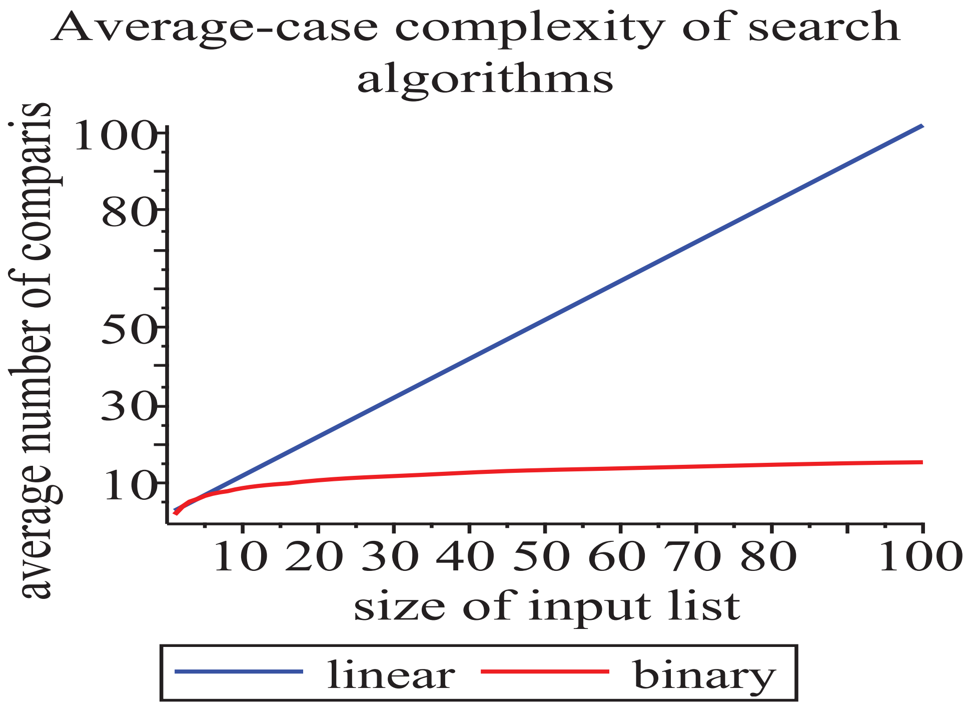
We want to overlay the graphs for the two algorithms. To do this, we assign the results of the **plot** command to a name instead of displaying them. We use different colors for the plots and ensure that the views are the same.

```
> linearPlot := itplot(nList, linearAverages, color = blue,
    view = [1..100, 1..102], legend = "linear") :
```

```
> binaryPlot := itplot(nList, binaryAverages, color = red,
    view = [1..100, 1..102], legend = "binary") :
```

To overlay the two plots, we use the **display** command, which is part of the **plots** package. In its simplest form, this command accepts a list of plot structures (such as what you obtain by applying the **plot** command) and overlays the plots. It can also accept most of the options that are available for **plot**. Below, we demonstrate the **display** command along with the use of several options to provide an informative graph.

```
> plots[display] ([linearPlot, binaryPlot,
    title = "Average-case complexity of search algorithms",
    labels = ["size of input list", "average number of comparisons"],
    labeldirections = [horizontal, vertical])
```

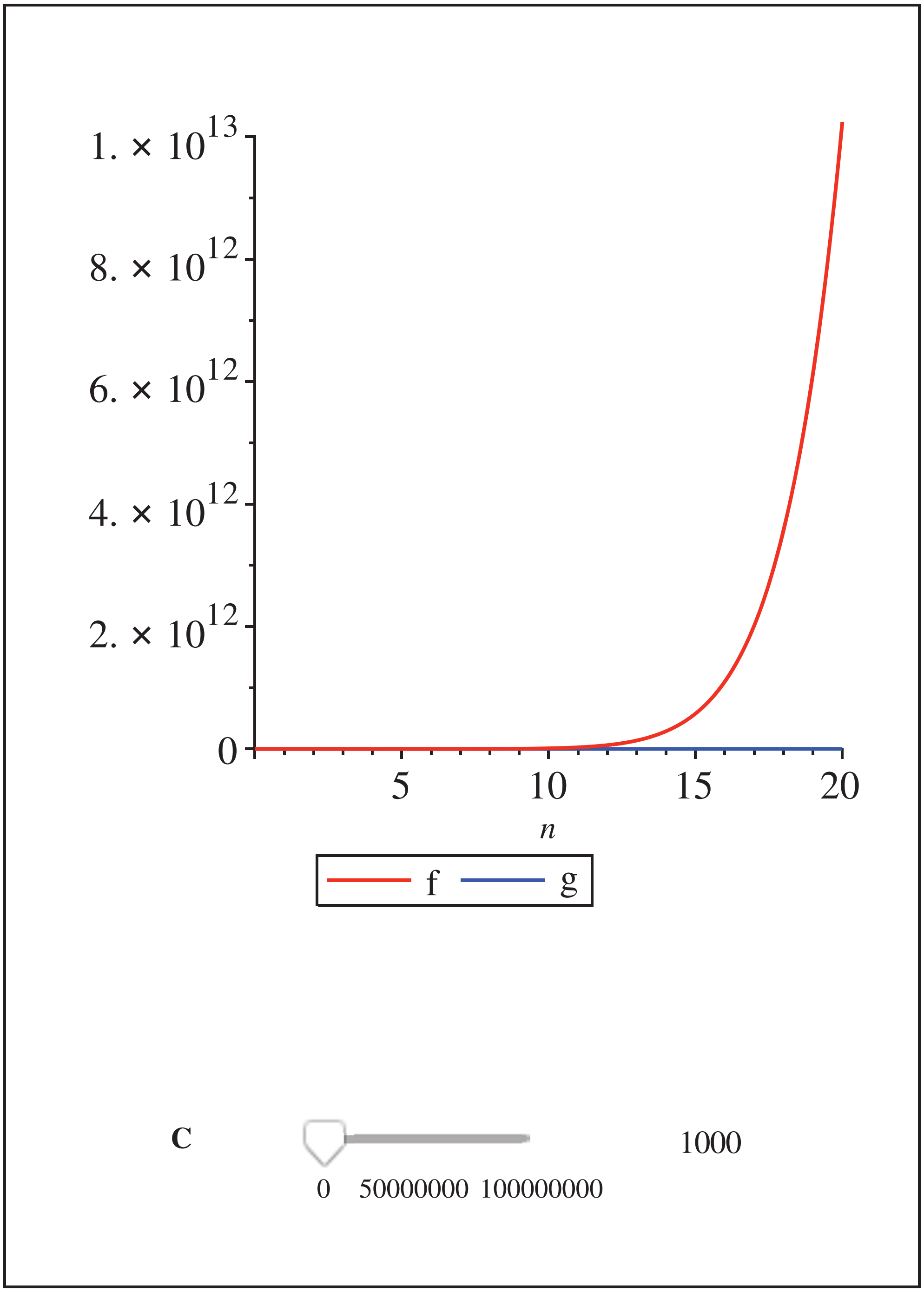


Computations and Explorations 1

We know that n^b is $O(d^n)$ when b and d are positive numbers with $2 \leq d$. Give values of the constants C and k such that $n^b \leq Cd^n$ whenever $k < n$ for each of these sets of values: .

Solution: For $b = 10$ and $d = 2$, we need to compare the functions $f(n) = n^{10}$ to $g(n) = 2^n$. Following the approach we took in Section 3.2, we will use Explore to graph the functions while dynamically changing the value of C. Note that the values of C that will suffice with $k < 20$ are extremely large. When exploring other values of b and d, you may have to modify the maximum values for C. It is also possible to find smaller values of C by increasing the horizontal extent of the graph.

> $\mathit{Explore}\left(\mathit{plot}\left([n^{10}, C \cdot 2^{n}], n = 0\,..20, \mathit{view} = [0\,..20, 0\,..10^{13}],\right.\right.$
$\left.\left.\mathit{color} = [\mathit{red}, \mathit{blue}], \mathit{legend} = [\text{“}f\text{”}, \text{“}g\text{”}]\right), C = 0\,..10^{8},\right.$
$\left.\mathit{initialvalues} = [C = 1000]\right)$



Once you find a potential value for C, you can confirm that it is correct and also find an exact value for k. For example, it appears that with $g(n) = C2^{n}$ dominates $f(n) = n^{10}$ for $n > 20$. We can confirm this by having Maple solve the inequality $n^{10} \leq 10^7 \cdot 2^n$ with the **solve** command. The **solve** command applied to an equation or inequality in one variable will find the solution to the equation or inequality. We apply **evalf** to force a floating-point result.

$$> \mathit{evalf}\left(\mathit{solve}\left(n^{10} \leq 10^7 \cdot 2^n\right)\right)$$

$$[-3.840511598, 10.07828296], [19.87621431, \infty) \tag{3.21}$$

This indicates that $C = 10^7$ and $k = 19.9$ witness for n^{10} being $O(2^n)$. Note that smaller values of C will work provided the value of k is made sufficiently large.

$$> \mathit{evalf}\left(\mathit{solve}\left(n^{10} \leq 10^5 \cdot 2^n\right)\right)$$

$$[-2.634476634, 4.243747408], [34.45805187, \infty) \tag{3.22}$$

$$> \mathit{evalf}\left(\mathit{solve}\left(n^{10} \leq 10^3 \cdot 2^n\right)\right)$$

$$[-1.765447237, 2.347895073], [44.93042315, \infty) \tag{3.23}$$

$$> \mathit{evalf}\left(\mathit{solve}\left(n^{10} \leq 1 \cdot 2^n\right)\right)$$

$$[-0.9371092012, 1.077550150), [58.77010593, \infty) \tag{3.24}$$

Exercises

Exercise 1. Write step-by-step instructions, then pseudocode, and then implement in Maple an algorithm to determine the k largest integers in a list of integers.

Exercise 2. Implement the linear search presented as Algorithm 2 in Section 3.1 of the text.

Exercise 3. Implement the insertion sort presented as Algorithm 5 in Section 3.1 of the text.

Exercise 4. Implement the cashier's algorithm presented as Algorithm 7 in Section 3.1 of the text.

Exercise 5. Implement the algorithm for scheduling talks presented as Algorithm 8 in Section 3.1 of the text.

Exercise 6. Implement the brute-force algorithm for finding the closest pair of points as presented in Algorithm 3 in Section 3.3 of the text.

Exercise 7. Modify the **bubblesort** procedure so that it terminates when no more interchanges are necessary. (See Exercise 39 from Section 3.1.)

Exercise 8. Implement the selection sort algorithm in Maple. (Refer to the preamble to Exercise 43 in Section 3.1 for information on selection sort.)

Exercise 9. Implement the binary insertion sort in Maple. (Refer to the preamble to Exercise 49 in Section 3.1 for information on the binary insertion sort.)

Exercise 10. Implement the deferred acceptance algorithm in Maple. (Refer to the preamble to Exercise 65 in Section 3.1 for information on the deferred acceptance algorithm.)

Exercise 11. Implement the Boyer–Moore majority vote algorithm in Maple. (Refer to the preamble to Exercise 68 in Section 3.1 for information on the majority vote algorithm.)

Exercise 12. Following the solution to Computations and Explorations 1, use Maple to determine values for C and k that witness for the fact that $f(x)$ is $O(g(x))$ for each of the pairs of functions given below.

a) $f(x) = 7\ln\left(3x^2 - 2x + 5\right)$; $g(x) = x$.
b) $f(x) = \dfrac{x^4}{x^2 - 4x - 4}$; $g(x) = x^2$.
c) $f(x) = \lfloor x \rfloor \lceil x \rceil$; $g(x) = x^2$.
d) $f(x) = n\ln(n)$; $g(x) = \ln(n!)$.

Exercise 13. Using the approach described in Section 3.3 of this manual, compare the average-case performance of the **bubblesort** procedure presented in Section 3.1 to Maple's **sort** command.

Exercise 14. Using the solution to Computer Project 10 as a model, compare the average-case complexity (as measured by the number of comparisons) of the **bubblesort** procedure with the modified procedure that you implemented as Exercise 7.

Exercise 15. Using the solution to Computer Project 10 as a model, compare the average-case complexity (as measured by the number of comparisons) of the **bubblesort** procedure with the other sort procedures you wrote (e.g., insertion sort, selection sort, or binary selection sort).

Exercise 16. Implement the two algorithms suggested by Exercise 31 of Section 3.1 for determining whether two strings are anagrams. Then, using the approach described in Section 3.3 of this manual, compare the average-case performance of the two algorithms. Compare the results of your performance analysis with the big-O estimates you found in Exercise 38 of Section 3.3.

Exercise 17. Implement the two algorithms suggested by Exercise 32 of Section 3.1 for finding the closest of n real numbers. Then, using the approach described in Section 3.3 of this manual, compare the average-case performance of the two algorithms. Compare the results of your performance analysis with the big-O estimates you found in Exercise 39 of Section 3.3.