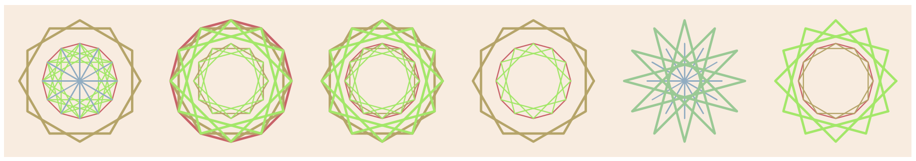
Exercises

Professor Leon Tabak

30 May 2022



Complexity

We want to compare algorithms. We want some measure that will allow us to select the "best" algorithm with which to solve some problem.

One measure counts the operations that the computer executes. We will call this measure the *complexity* of the algorithm. We will use a notation called "Big-Oh" to denote the complexity.

For example, we might want to know how much work the computer has to do to find the smallest value in a list of integers when we use the sequential search algorithm. Of course, the amount of work depends upon the length of the list.

For that reason, instead of looking for one number we will search for a relationship between the size of the input (in this case, how many numbers are in the list) and the number of operations executed.

We might choose not to count all instructions that the computer executes, not only the most significant instructions. In our analysis of searching and sorting algorithms, we are most interested in operations that compare one value to another. We want to know how many times the computer touches each element of the list that we are sorting or through which we are searching.

In the following table, N means the length of the list. The second column contains "Big-Oh" notation.

algorithm	**complexity**
sequential search	$O(N)$
binary search	$O(\log N)$
selection sort	$O(N^2)$
insertion sort	$O(N^2)$
merge sort	$O(N \log N)$

The selection and insertion sorting algorithms are both $O(N^2)$ algorithms. This means that, for sufficiently large values of N, the number of operations required to sort a list is no greater than some fixed multiple of N^2.

Let's put aside the part of the definition that says "for sufficiently large values of N. Let's also suppose that the constant of proportionality is one.

Then we can say this in different way: the time required to sort a list using either of these two algorithms is roughly proportional to the square of the length of the list.

- Doubling the length of the list increases the time required to sort by list by a factor of four.
- Multiplying the length of the list by ten increases the time required to sort by a factor of one hundred.

Look at the expressions in the table:

- A graph of $O(N)$ is a straight line. The line rises at a constant rate.
- A graph of $O(N^2)$ is a parabolic arc. The curve climbs upward at an ever steeper rate.

You are familiar with linear and quadratic functions. You have seen logarithms before, but might be less sure logarithms of how they work and what they mean in an analysis of complexity..

Let's review.

Properties of logarithms

Let $\log_2 x$ be the logarithm base 2 of x.

The logarithm of a number is the power to which we must raise the base to produce that number:

- $\log_2 1 = 0$ because $1 = 2^0$
- $\log_2 2 = 1$ because $2 = 2^1$
- $\log_2 4 = 2$ because $4 = 2^2$.

What are the values of the logarithms base 2 of the next few powers of 2?

- $\log_2 8 =?$
- $\log_2 16 =?$
- $\log_2 32 =?$

The *domain* of a function $f(x)$ is the set of values of x for which the function is defined.

- For which values of x can we compute $\log_2 x$?

Logarithms allow us to substitute addition for multiplication.

$$\log_2 2^1 = \log_2 2 = 1$$
$$\log_2 2^2 = \log_2 4 = 2$$

$$2^1 \cdot 2^2 = 2 \cdot 2 \cdot 2 = 2^{1+2} = 2^3$$

$$\log_2(2^1 \cdot 2^2) = \log_2 2^3 = \log_2 2 + \log_2 4 = 3$$

In general, $\log_2(a \cdot b) = \log_2 a + \log_2 b$.

- Who invented (or, if you prefer, discovered) logarithms? When?
  Find the answer online.
- Search on the Internet for an explanation of how slide rules work.

Look what happens when we apply this rule repeatedly:

$$\begin{aligned} \log_2(a \cdot a) &= \log_2 a + \log_2 a = 2\log_2 a \\ \log_2(a \cdot a \cdot a) &= \log_2 a + \log_2 a + \log_2 a = 3\log_2 a \\ &\vdots \\ \log_2 a^n &= n\log_2 a \end{aligned}$$

Use these properties of logarithms to compute more values:

- $\log_2 64 = \log_2(8 \cdot 8) = ?$
- The logarithm base 2 of 16 is 4.
  What is the logarithm base 2 of $(16 \cdot 16) = 256$?
- $\log_2 32 = 5$ and $32 \cdot 32 = 1024$.
  What is the value of $\log_2 1024$?
- $1024 \cdot 1024 \approx 1,000,000$
  What is the approximate value of $\log_2(1,000,000)$?

We can define logarithms with bases other than 2:

$$\begin{aligned}
\log_{10} 1 &= \log_{10} 10^0 = 0 \\
\log_{10} 10 &= \log_{10} 10^1 = 1 \\
\log_{10} 100 &= \log_{10} 10^2 = 2 \\
\log_{10} 1000 &= \log_{10} 10^3 = 3
\end{aligned}$$

Your calculator probably does not have a key for computing logarithms base 2. It might have a key for computing logarithms base 10.

Here's how we can compute the value of $\log_2(x)$ given the value of $\log_{10}(x)$.

$$\begin{aligned}
10^x &= a \\
\log_{10} a &= x \\
\\
2^y &= a \\
\log_2 a &= y \\
\\
10^z &= 2 \\
\log_2 10 &= z \\
\\
(10^z)^y &= 2^y \\
&= a \\
\\
\log_{10} a &= z \cdot y \\
&= \log_{10} 2 \cdot \log_2 a \\
\\
\log_2 a &= \log_{10} a / log_{10} 2
\end{aligned}$$

When we characterize the complexity of the binary search algorithm with the expression $O(\log N)$, we mean that the computer executes a number of instructions that is no more than some multiple of $\log N$. Changing the base of the logarithms changes that multiple. However, with any base the amount of work remains proportional to the logarithm of the length of the list.

In this sense, the base of the logarithms does not matter in the characterization of an algorithm's complexity.

However, logarithms base 2 will arise naturally in our study of how these algorithms work.

The binary search and merge sort algorithms repeatedly divide a list in half. The logarithm of the list's length tells us how many times we can divide by two before we get down to one.

Before, we found the value of $\log_2 32$ by counting multiplications:

$$32 = 1 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$$

There are five twos in the product and so $\log_2 32 = 5$.

We can go in the opposite direction. If we count divisions, we also get 5.

$$\begin{aligned} 32/2 &= 16 \\ 16/2 &= 8 \\ 8/2 &= 4 \\ 4/2 &= 2 \\ 2/1 &= 1 \end{aligned}$$

- Approximate the values of N^2 and $N \log_2 N$ for $N = 1000$ and $N = 1,000,000$.

  How much advantage do we get by choosing a $O(N \log N)$ algorithm rather than a $O(N^2)$ algorithm?

- $\log x$ grows much more slowly than x. The derivative of a function $f(x)$ tells us how quickly $f(x)$ increases as we increase x.

  Find the derivative of the natural logarithm. (That's the logarithm whose base is e.)

  $\frac{d}{dx} \log x = ?$

- There are problems for which the best known solutions have an exponential time complexity: $O(e^x)$.

  What is the relationship between the exponential function and the logarithm function?

Program 0

```
#include <stdio.h>
#include <stdlib.h>

// TO-DO: What does f() do?
//  Give this function a sensible name.
int f( int data[], int length, int start ) {
    int bestGuessSoFar = start;

    for( int i = start + 1; i < length; i++ ) {
        if( data[i] < data[bestGuessSoFar] ) {
            bestGuessSoFar = i;
        } // if
    } // for

    return bestGuessSoFar;
} // f( int [], int, int )

// TO-DO: What does g() do?
//  Give this function a sensible name.
void g( int data[], int i, int j ) {
    int temp = data[i];
    data[i] = data[j];
    data[j] = temp;
} // g( int [], int, int )

int main( int argc, char** argv ) {

    // length of the data array
    int length = 8;

    // the data array
    // the first 4 elements are the 4 smallest
    // values in the array
    // the first 4 elements are in order
    int data[] = { 2, 5, 7, 13, 31, 23, 17, 29 };

    // print array before calling f() and g()
    printf( "\n\n data = {" );
    for( int i = 0; i < length; i++ ) {
       printf( "%4d", data[i] );
    } // for
    printf( "  }\n\n" );
    // index of the first element in the unsorted
    // part of the data array
    int start = 4;

    int index = f( data, length, start );

    // f0() returns an index of an element in data
    // print the index of that element
    printf( "index = %2d\n", index );

    // print the value of the element at that index
    printf( "data[%2d] = %2d\n", index, data[index] );

    g( data, start, index );

    // print array after calling f() and g()
    // TO-DO: The code that follows is repeated code.
    //    It would be better to define a function once
    //    and call it twice rather than write the
    //    same code twice.
    //
    //    Can you define a function that prints the
    //    contents of an array of integers?
    printf( "\n\n data = {" );
    for( int i = 0; i < length; i++ ) {
       printf( "%4d", data[i] );
    } // for
    printf( "  }\n\n" );

    // TO-DO: This program shows one step in a sorting
    // algorithm. Which sorting algorithm?

    // TO-DO: Change the values of length, data, and start.
    //    * The value of length must equal the size of the data array
    //    * The first n elements in data must be in order and
    //      must be the smallest n elements in data..
    //    * The value of start must equal n.
    // Run the program again. Do you see what you expected to see?
} // main( int, char** )
```

Output of Program 0

```
 data = {    2    5    7   13   31   23   17   29   }
```

```
index =  6
data[ 6] = 17


 data = {    2    5    7   13   17   23   31   29   }
```

Program 1

```
#include <stdio.h>
#include <stdlib.h>

// TO-DO:
//    * Run this program.
//    * Study the function f().
//    * Give the function f() a sensible name.

// the first k elements of data are in order
//
// the first k elements of data need not be
// the smallest k elements in data
void f( int data[], int k ) {

    // TO-DO:
    //    * What is the greatest number of times that
    //      the body of the loop will be executed?
    //    * What is the least number of times that
    //      the body of the loop will be executed?
    while( k > 0 && data[k] < data[k - 1] ) {

        // TO-DO: Define a function that does
        // what these next 3 lines of code do.
        // Replace these 3 lines of code with
        // a call to the new function.
        int temp = data[k];
        data[k] = data[k - 1];
        data[k - 1] = temp;

        k--;
    } // while

} // f( int[], int )

int main( int argc, char** argv ) {

    // number of elements in array
    int length = 8;

    // partly sorted array of integers
    int data[] = {2, 3, 5, 11, 13, 7, 19, 17 };

    // first 5 elements of data are already in order
    int lengthOfSortedPartOfList = 5;

    // look at data before calling f()
    printf( "\n\n data = { " );
    for( int i = 0; i < length; i++ ) {
        printf( "%4d", data[i] );
    } // for
    printf( " }\n\n" );

    f( data, lengthOfSortedPartOfList );

    // Look at data after calling f().
    printf( "\n\n data = { " );
    for( int i = 0; i < length; i++ ) {
        printf( "%4d", data[i] );
    } // for
    printf( " }\n\n" );

    // TO-DO:
    //    * Increment the value of lengthOfSortedPartOfList.
    //    * Call f() a second time.
    //    * Look at the data after calling f() the second time.
    //    * What do you see?

    // TO-DO: The function f() describes one step in
    // which sorting algorithm?
} // main( int, char** )
```

Output of Program 1

```


 data = {    2   3   5  11  13   7  19  17 }



 data = {    2   3   5   7  11  13  19  17 }

```