Assignment 2 – Big Oh

This document is based on: Active Learning Approach to Data Structures using C++ by Dr. Timothy Budd

Due: Sep 25, 2014, 8:00 AM

Late penalties will be as described in the syllabus.
Points may be deducted for failure to follow these instructions.

General Objectives:

Learn how to use and apply Asymptotic Analysis with Big-Oh notation.
Introduce a simple search routine: SelectionSort
Demonstrate ability to think and communicate logically

Graded Part of the Assignment:

After the background material you will find a "walk through" worksheet.

1. Fill in all the blanks in the walk through portion.
Use that information to **complete the Big-Oh table at its beginning.**

After the walk-through:

2. Complete the Tracing Code Examples table.
3. Complete the Polynomial Classifications table.
4. Answer the questions on the "answer the following questions" page(s).
5. Type your name into the header blank (above right corner)

What to turn in:

Save a copy of this Word Document named: **A02_BigOh_*yourlastname*.docx**
Be certain the items noted above are completed.
Submit the completed document to the appropriate D2L dropbox.

Background Material:

Why do dictionaries list words in alphabetical order? The answer may seem obvious, but it nevertheless can lead to a better understanding of the concept of algorithms. Perform the following mind experiment. Suppose somebody were to ask you to look up the telephone number for "Chris Smith" in the directory for a large city. How long would it take? Now suppose they asked you to find the name of the person with number 564-8734. Would you do it? How long do you think it would take?

Is there a way to quantify this intuitive feeling that searching an ordered list is faster than searching an unordered list? Suppose *n* represents the number of words in a collection. In an unordered list you must compare the target word to each list word in turn. This is called a ***linear search***. If the search is futile; that is, the word is not on the list, you will end up performing *n* comparisons. Assume that the amount of time it takes to do a comparison is a constant. You don't need to know what the constant is; in fact it really doesn't matter what the constant is. What is important is that the total amount of work you will perform is *proportional* to *n*. That is to say, if you were to search a list of 2000 words it would take twice as long as it would to search a list of 1000 words. Now suppose the search is successful; that is, the word is found on the list. We have no idea where it might be found, so on average we would expect to search half the list. So the expectation is still that you will have to perform about *n/2* comparisons. Again, this value is proportional to the length of the list – if you double the length of the list, you would expect to do twice as much work.

What about searching an ordered list? Searching a dictionary or a phonebook can be informally described as follows: you divide the list in half after each comparison. That is, after you compare to the middle word you toss away either the first half or the last half of the list, and continue searching the remaining, and so on each step. This is termed a *binary search*. It is similar to the way you guess a number if somebody says "I'm thinking of a value between 0 and 100. Can you find my number?" To determine the speed of this algorithm, we have to ask how many times you can divide a collection of *n* elements in half.

To find out, you need to remember some basic facts about two functions, the *exponential* and the *logarithm*. The exponential is the function you get by repeated multiplication. In computer science we almost always use powers of two, and so the exponential sequence is 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, and so on. The logarithm is the inverse of the exponential. It is the number that a base (generally 2) must be raised to in order to find a value. If we want the log (base 2) of 1000, for example, we know that it must be a value between 9 and 10. This is because 2^9 is 512, and 2^{10} is 1024. Since 1000 is between these two values, the log of 1000 must be between 9 and 10. The log is a very slow growing function. The log of one million is less than 20, and the log of one billion is less than 30.

It is the log that provides the answer to our question. Suppose you start with 1000 words. After one comparison you have 500, after the second you have 250, after the third 125, then 63, then 32, then 16, 8, 4, 2 and finally 1. Compare this to the earlier exponential sequence. The values are listed in reverse order, but can never be larger than the corresponding value in the exponential series. The log function is an approximation to the number of times that a group can be divided. We say *approximation* because the log function returns a fractional value, and we want an integer. But the integer we seek is never larger than 1 plus the ceiling (that is, next larger integer) of the log.

Performing a ***binary search*** of an ordered list containing *n* words you will examine approximately *log n* words. You don't need to know the exact amount of time it takes to perform a single comparison, as it doesn't really matter. Represent this by some unknown quantity *c,* so that the time it takes to search a list of *n* words is represented by *c * log n*. This analysis tells us is the amount of time you expect the search process to change if, for example, you double the size of the list. If you next search a collection of *2n* words, you would expect the search to require *c * log (2n)* steps. But this is *c * (log 2 + log n*). The log 2 is just 1, and so this is nothing more than *c + c * log n*. What this is saying is that if you double the size of the list, you will expect to perform just one more comparison. This is considerably better than in the unordered list, where if you doubled the size of the list you doubled the number of comparisons that you would expect to perform.

Big Oh notation

In order to simplify the comparison between two or more algorithms there is a standard notation. We say that ***a linear search is a O(n) algorithm*** (read "big-Oh of n"),
while ***a binary search is a O(log n) algorithm*** ("big-Oh of log n").

The idea captured by big-Oh notation is like the concept of the derivative in calculus. It represents the rate of growth of the execution time as the number of elements increases, or ∂-time versus ∂-size. Saying that an algorithm is O(n) means that the execution time is bounded by some constant times n. Write this as c*n. If the size of the collection doubles, then the execution time is c*(2n). But this is 2*(c*n), and so you expect that the execution time will double as well. On the other hand, if a search algorithm is O(log n) and you double the size of the collection, you go from c*(log n) to c*(log 2n), which is simply c + c*log n. This means that O(log n) algorithms are much faster than O(n) algorithms, and this difference only increases as the value of n increases.

A task that can be performed in constant time is described as O(1), or "constant time" regardless of the amount of time it takes. A task that is O(n) is termed a *linear* time task. One that is O(log n) is called *logarithmic*. Other terms that are used include *quadratic* for O(n^2) tasks, and *cubic* for O(n^3) algorithms.

What a big-oh characterization of an algorithm does is to abstract away unimportant distinctions caused by factors such as different machines or different compilers. Instead, it goes to the heart of the key differences between algorithms. The discovery of a big-oh characterization of an algorithm is an important aspect of algorithmic analysis. Due to the connection to calculus and the fact that the big-oh characterization becomes more relevant as *n* grows larger this is also often termed *asymptotic analysis*.

In this worksheet we will be concentrating on algorithms involving loops. In this case the **question** to ask is: **how many times the inner statements in a loop are being executed, and how this quantity changes if the input size is changed.** As you look at each example, fill in the values in the table below.

So, complete the following table. *The blanks in the walk-through that follows should help you.*

Copy the completed table to the word document you submit to D2L.

The first two entries are from the above background material and are included in the table for reference.

Linear search	***O(n)***
Binary search	***O(log n)***
countOccurrences	*O(____)*
isPrime	*O(____)*
printPrimes	*O(____)*
matrixMult	*O(____)*
SelectionSort	*O(____)*

Big-Oh Table

log n here reads:
log base 2 of n

You may also see it written as: *lg n*

COMPLETE THIS PAGE in the document you turn in to D2L --- indicate your answers in the above table

Walk Through Begins:

```
int countOccurrences (double data [ ], double
testValue)
{
   int count = 0;
   for (int i = 0; i < data.length; i++)
   {
      if (data[i] == testValue)
         count++;
   }
   return count;
}
```

C++ arrays of type double
do NOT have a member variable: *length*

So this code will NOT compile in C++

Treat this as pseudo-code

Assume data.length returns the correct value, which is the size of the input array

Our first function, ***countOccurences***, counts the number of occurrences of a given value in an array of data.
If the execution time is proportional to the size of the input array (data.length = n),

it is ***O(__________).***

```
int isPrime (int n)
{
   for (int i = 2; i * i <= n; i++)
   {
      if (0 == n % i) return 0;
   }
   return 1;
}
```

Our second function, ***isPrime***, determines if an integer variable is prime. A prime, you will recall, is a value that is divisible only by itself and 1. Note that the input here is not an array, but a single integer value. What can you say about the worst-case execution time for this function in relation to the value of this variable?

It is ***O(__________)***

```
void printPrimes (int n)
{
   for (int i = 2; i < n; i++)
   {
      if (isPrime(i))
         printf("Value %d is prime", i);
   }
}
```

printf is a C style function, you may again Consider this pseudo-code if it helps.

When one function calls another function (isPrime) inside of the loop, the execution time of the called function must be counted in determining the time for the loop. Here there is a simple O(n) loop that calls isPrime. So the total execution time is in the worst case, for ***printPrimes*** is ***O(_________)***

```
void matrixMult (int **a, int **b, int **c, int n)
{
    for (int i = 0; i < n; i++)
    {
     for (int j = 0; j < n; j++)
     {
           c[i][j] = 0;
           for (k = 0; k < n; k++)
           {
                c[i][j] += a[i][k] * b[k][j];
           }
     }
}
```

Don't worry about the function parameter types so much as the for-loops of the function body. Take it that a, b, and c are all 2d-arrays (matrices) of size *n* x *n*

The classic matrix multiplication routine is a good example of nested loops. Again the question to ask is how many times the statements in the innermost loop are executed as a function of the data size (in this case, the number of rows and columns of the input arrays). Trace the code and variables. The execution time for ***matrixMult*** is ***O(______________)***

Finally, consider the ***selectionSort*** method shown. You may want to:
Trace the code by hand (track all variables) and complete an example of sorting an array of 4 (n = 4) elements with the values 3.3, 2.2, 4.4 and 1.1.

```
void selectionSort (double storage[], int n)
{
     for (int p = n – 1;   p > 0; p--)
     {
        int indexLargest = 0;

        for (int i = 0; i <= p; i++)
        {
                if (storage[i] > storage[indexLargest])
                        indexLargest = i;
        }

        if (indexlargest != p)
                swap(storage, indexLargest, p);
     }
}
```

Here the inner loop is more subtle. Trace the code for the various values of *p* and determine the number of steps in the inner loop.

How is the inner loop related to the outer loop? ____________________

How many steps will it take the first time the outer loop executes? __________

The last? _____________

Can you identify the pattern? ______________________

What does this tell you about the
number of times the innermost if statement will be executed? ___________________________________

If you are unable to determine the Big-Oh for the function ***selectionSort*** *defined above, then you may optionally include this page's work as the LAST page of the document you submit to D2L. Indicate this in the appropriate table entry. Partial credit may or may not be awarded at the discretion of the instructor at time of grading.*

Tracing code examples:

Assume that the … represents constant time operations. Describe in Big-Oh notation the running time of each of the following loops. As n grows larger only the fastest growing portion of the polynomial is included in Big-Oh notation so do not include any unnecessary constants in your answers.

Methods or Loops	Big-Oh
for (int i = 0; i < n; i++) …	O(______________________)
for (int i = n; i > 0; i--) …	O(______________________)
for (int i = 0; i * i < n; i++) …	O(______________________)
for (int i = n; i > 0; i = i / 2) …	O(______________________)

for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)…
	O(______________________)

for (int i = 0; i < n; i++)
  for (int j = 0; j < i; j++)…
	O(______________________)

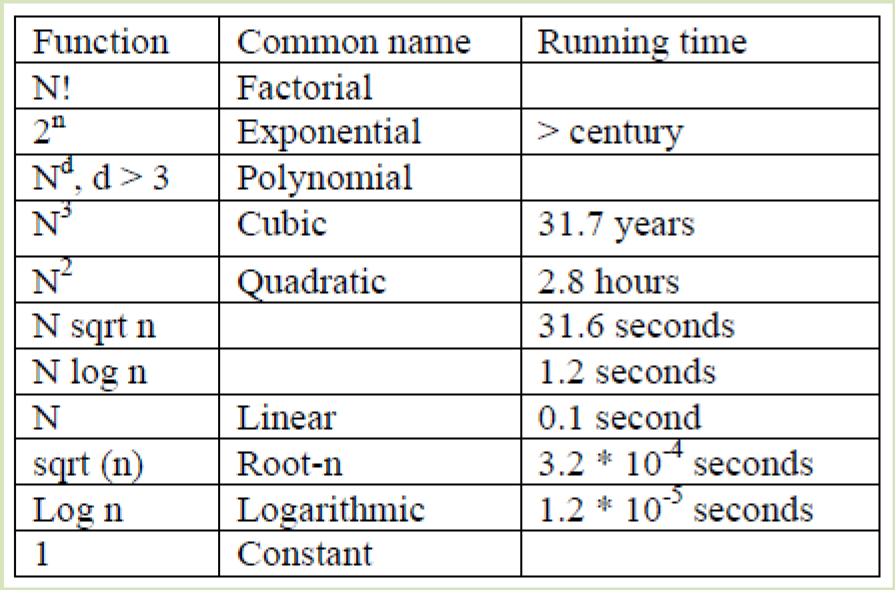
for (int i = 0; i < n; i++)
  for (int j = 0; j < 13; j++)..
	O(______________________)

for (int i = n; i > 0; i = i / 2)
  for (int j = 0; j < n; j++)…
	O(______________________)

for (int i = 0; i < n; i++)
  for (int j = n; j > 0; j = j / 2)…
	O(______________________)

COMPLETE THIS PAGE in the document you turn in to D2L --- indicate your answers in the above table

Polynomial Classifications:



Function	Common name	Running time
N!	Factorial	
2^n	Exponential	> century
N^d, d > 3	Polynomial	
N^3	Cubic	31.7 years
N^2	Quadratic	2.8 hours
N sqrt n		31.6 seconds
N log n		1.2 seconds
N	Linear	0.1 second
sqrt (n)	Root-n	$3.2 * 10^{-4}$ seconds
Log n	Logarithmic	$1.2 * 10^{-5}$ seconds
1	Constant	

The table at left lists functions in order from most costly to least. The middle column is the common name for the function.

Suppose that by careful analysis you have discovered a function that describes the precise running time of various algorithms as a function of the input size as shown in the table below. For each of the following, describe the running time using Big-Oh notation.

$2n^3 + 5n + 7$	O(___________________________)
(4 * n) * (3 + log n)	O(___________________________)
(5 n + 4) / 6	O(___________________________)
1 + 2 + 3 + … + n *(be careful with this one, it is NOT written in a polynomial form)*	O(___________________________)
$(n + 1)^4$	O(___________________________)
n^2 + n log n	O(___________________________)
$n^3 – 1000 n^2$	O(___________________________)
$2^n + n^2$	O(___________________________)
n + log n^2	O(___________________________)

COMPLETE THIS PAGE in the document you turn in to D2L --- indicate your answers in the above table

Additional Background Material:

As you have seen, a big-Oh description of an algorithm is a characterization of the change in execution time as the input size changes. If you have actual execution timings ("wall clock time") for an algorithm with one input size, you can use the big-Oh to estimate the execution time for a different input size. The fundamental equation says that the ratio of the big-Oh's is equal to the ratio of the execution times. If an algorithm is O(f(n)), and you know that on input n1 it takes time t1, and you want to find the time t2 it will take to process an input of size n2, you create the equation

$$f(n1) / f(n2) = t1 / t2$$

To illustrate, suppose you perform the following experiment:
You ask a friend to search for the phone number for the **name**: "Chris Smith" in 8 pages of a phone book (i.e. a binary search). Your friend does this in 10 seconds. From this, you can estimate how long it would take to search a 256 page phone book. Remembering that binary search is O(log n), you set up the following equation:

$$\log(8)/\log(256), \text{ which is } 3 / 8 = 10 / X$$

Solving for X gives you the answer that your friend should be able to find the number in about 24 seconds.

Now you time your friend performing a search for the name attached to a given **number** in the same 8 pages (i.e. a linear search). This time your friend takes 2 minutes. Recalling that a linear search is O(n), this tells you that to search a 256 page phone could require:

$$8/256 = 2 / X$$

Solving for X tells you that your friend would need about 64 minutes, or about an hour. Note this also serves to illustrate a binary search is really faster than a linear search.

Continue to the next page to answer questions

Name: ___________________________

Answer the following questions:

1. What is a linear search and how is it different from a binary search?

2. Can a linear search be performed on an unordered list? Can a binary search?

3. If you start out with *n* items and repeatedly divide the collection in half, how many steps will you need before you have just a single element? (Express as function of *n*)

Explain how – show work/equations used

4. Suppose an algorithm is *O(n)*, where *n* is the input size. If the size of the input grows to 16 times larger, how will the execution time change? (1000 to 16000)

5. Suppose an algorithm is *O(log n)*, where *n* is the input size. If the size of the input grows to 16 times larger, how will the execution time change?

log n here reads: log base 2 of n

6. Suppose an algorithm is $O(n^2)$, where n is the input size. If the size of the input grows to 16 times larger, how will the execution time change?

COMPLETE THIS PAGE in the document you turn in to D2L --- with your answers clearly indicated
-=-=-=