Difference between this class and CS1

Where as in CS I, we simply asked you to come up with solutions to problems and did not care about the quality of those solutions, in this class we will put more emphasis on the efficiency of solutions. To that end, we will study a number of different problem solving techniques and try to show the situations in which these techniques yield efficient solutions.

In CS1, we briefly touched on several data structures including linked lists and binary trees. In this class we will reinforce those ideas from CS1 and add some detail. Generally, most students coming out of CS1 are not proficient enough to code up programs using linked lists and binary trees on their own. Hopefully, by the end of this class you will have the confidence to do so. Furthermore, we will learn about some new data structures, such as heaps and hash tables.

In CS1 we very briefly introduced the idea of Big-Oh notation. In this class you will be given the actual definition of Big-Oh as well as some other complementary definitions. Furthermore, you will be asked to use this notation in the analysis of algorithms. Mathematically, the analysis will be more involved than the algorithms presented in CS1.

Putting this together, here is a list of the three basic goals of this class:

1) Coming up with efficient solutions to problems, utilizing several standard algorithmic techniques.
2) A more detail understanding of several data structures to allow writing code using these data structures
3) Proficient use of mathematics and Big-Oh notation so that the time efficiency of different algorithms can be compared.

Algorithm Analysis

Determining the exact amount of time an implementation of an algorithm will take to run is impossible. Different processors run and different speeds, and different architectures support different types of instruction sets, which may affect running time. These are just a couple of the many factors which could affect the actual running time of the implementation of an algorithm.

Thus, rather than try to determine the exact running time of an algorithm, we will be happy to simply approximate the running time of an algorithm to within a constant factor. In particular, our assumption will be that the running time of an algorithm depends on the input size, and that roughly speaking, one could construct a function t(n) that is the running time of an algorithm when given an input of size n.

Naturally, one of the difficulties you might notice here is that in reality, if you run an algorithm with two different data sets of the same size, it's quite possible that the algorithm will take different amounts of time running on each. Thus, we will make a distinction between worst case running time and average case running time. I'll explain this difference in a bit. For example, it's possible that an algorithm with input size n could take 2n+1 steps sometimes, but 5n+5 steps other times.

Since t(n), the running time of an algorithm with input of size n, can not be calculated exactly, we want to simply determine the function's Big-Oh value, or its largest possible value within a constant. Thus, we would like to make claims such as t(n) = O(f(n)), where f(n) is some standard function. Once we determine this, then we'll say that the algorithm runs in O(f(n)) time and use this as a basis of comparison with other algorithms.

Another way to think about algorithm analysis is trying to determine the approximate number of simple statements a program will execute. The hardware executes many simple instructions for a program to run. Relatively speaking, each of these instructions takes a small constant amount of time to run. From processor to processor, this constant amount of time is different, which accounts for the fact that the identical code run on two different machines may result in different running times.

Often times, the number of small instructions a program executes is based upon the size of the input to the program. We can think of the following as simple instructions:

Memory references
Comparisons
Arithmetic operations
Function calls (just the calls themselves, not necessarily the body of the function)

Assuming that these are simple constant time operations, we can also show that the following are constant time operations:

assignment statement, (as long as there are no non-constant time functions computed on the RHS.)
if statement, (as long as the body takes a constant amt. of time)

So, we can think of the goal of algorithm analysis as counting the number of simple operations a program implementing an algorithm will execute if run on an input of size n. When doing this, we will find that even this will be quite tedious. Thus, rather than finding the exact number of simple statements a program will execute, we will be happy finding a big Oh approximation for the total number of simple statements an algorithm will execute.

Examples of Counting the Number of Simple Statements

1) Consider the following code segment:

```
for (int i=1; i<= 2*n; i++)
  x = x + 1;
```

In this segment of code, the simple statements/expressions

i <= 2*n x = x + 1; and i++

each run 2n times. Thus, the total number of simple statements run is approximately 6n + 1. Since we don't care about multiplicative constants, we can safely say that the running time of this segment of code is O(n).

2) Consider the following segment of code:

```
for (int i=1; i <= 3*n; i++) {
  for (int j=1; j <= n/2; j=j+3) {
    x = x + 1;
  }
}
```

In this segment of code, the outer loop runs 3n times. The inner loop runs approximately n/6 times because only ever third value of j occurs in the iterations. Note that the value is approximate because of integer division and off by one issues. Also note that the number of times the inner loop runs is constant and not based upon the value of any variables. Thus, the statement x = x+1; runs approximately 3n(n/6) = $n^2/6$ times. All of the other statements run this many times or less. Since we are ignoring multiplicative constants, the run time of this segment of code is O(n^2).

Code Analysis that Shows the Necessity of Some Math

3)

```
int j = n;
while (j >= 1) {
  for (int i=1; i<=j; i++)
    x = x + 1;
  j = j/3;
}
```

The inner loop runs j times, where j is first equal to n, then n/3, then n/9, etc. Technically, since integer division is done, the real number division values of n/3 and n/9 could be slight overestimates. Thus, the following sum represents an upper bound on the total number of times the statement x = x + 1; is executed:

$$\sum_{i=0}^{\infty} \frac{n}{3^i}$$

Of course, with integer division, we know that the sum will not go to infinity. (It will end when i is equal to $\log_3 n$, approximately.)

As we see from this example, in order for us to determine the run time of this segment of code, ultimately, we are forced to deal with a summation.

Thus, the major reason we will briefly cover mathematics at the beginning of the course is to assist us in algorithm analysis: counting the number of simple statements an algorithm will take and/or counting the amount of memory an algorithm will use if implemented.

Handling the Sum in the Example Above

Let S = $\sum_{i=0}^{\infty} \frac{n}{3^i}$

S = 1 + 1/3 + 1/9 + 1/27 + …
S/3 = 1/3 + 1/9 + 1/27 + …

S – S/3 = 1, which is obtained by subtracting the bottom equation from the top.
S(2/3) = 1
S = 3/2

This summation above is an infinite geometric sum. The derivation for solving such a sum in general is very similar to the technique shown above.

A general infinite geometric sum has a first term, let this be a_1, and a common ratio between successive terms, let this be r. (Note that |r| < 1 to ensure that the series converges.)

S = a_1 + a_1r + a_1r^2 + a_1r^3 + …
rS = a_1r + a_1r^2 + a_1r^3 + …

S – rS = a_1, which is obtained by subtracting the bottom equation from the top.
S(1 - r) = a_1
S = a_1/(1-r)

(Here note that to obtain a subsequent term from a previous term in the sequence we just multiply be r. Thus, the third term is a_1r^2, and the fourth term is a_1r^3, etc.)

Note that we can handle the sum of a finite geometric series of n terms similarly:

$$S = a_1 + a_1r + a_1r^2 + a_1r^3 + \ldots + a_1r^{n-1}$$
$$rS = a_1r + a_1r^2 + a_1r^3 + \ldots + a_1r^{n-1} + a_1r^n$$

$$S - rS = a_1 - a_1r^n$$
$$S(1 - r) = a_1(1 - r^n)$$
$$S = a_1(1 - r^n)/(1-r)$$

(Here note that since the series is finite, there is one "extra" term on the second row, which is the result of multiplying a_1r^{n-1} by r. This term appears in the difference shown above.)

Now that we've established the necessity of some mathematics, let's review some mathematical rules and proof techniques that will be necessary for us to use throughout the semester in our algorithm analysis.

Mathematical Preliminaries

Logs

The log function is the inverse of an exponent. Thus, if we have

$b^a = c$, then it follows by definition that $\log_b c = a$.

Since a positive number to any exponent can never be negative, and only logs with positive bases (except for 1) are computed, it follows that you can never take the log of 0 or any negative value.

Let's review several rules that apply to logarithms and exponents.

$\log_b a + \log_b c = \log_b ac$
$\log_b a - \log_b c = \log_b a/c$
$\log_b a^c = c\log_b a$
$\log_b a = \log_c a / \log_c b$
$b^{(\log_c a)} = a^{(\log_c b)}$
$b^a b^c = b^{a+c}$
$b^a / b^c = b^{a-c}$
$(b^a)^c = b^{ac}$

One key observation here: *Logarithms grow slowly.* $2^{10} = 1024$ ($\log_2 1024 = 10$). $2^{20} = 1048576 \approx 1 \times 10^6$ ($\log_2 1 \times 10^6 \approx 20$). $2^{30} = 1073741824 \approx 1 \times 10^9$ ($\log_2 1 \times 10^9 \approx 30$). This means that the performance of an O(N log N) algorithm is much closer to that of an O(N) algorithm than an $O(N^2)$ algorithm, even for moderately large amounts of input.

In general, any time we repeatedly halve or multiply a quantity in some way, a log is involved in the analysis.

Summations

By definition of a summation we have the following general form:

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \ldots + f(b)$$

Here are a couple standard summations formulas we will use often:

$$\sum_{i=0}^{n-1} a_1 r^i = \frac{a_1(1-r^n)}{1-r}$$

and

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Also, here is an example of a technique that works with some sums:

Find $1(2^0) + 2(2^1) + 3(2^2) + \ldots + n(2^{n-1})$

Let $S = 1(2^0) + 2(2^1) + 3(2^2) + \ldots + n(2^{n-1})$. Then
$2S = \quad 1(2^1) + 2(2^2) + 3(2^3) + \ldots + n(2^n)$

Now, subtract the bottom equation from the top:

$-S = 1(2^0) + (2^1) + (2^2) + \ldots + (2^{n-1}) - n(2^n)$
$S = n(2^n) - ((2^0) + (2^1) + (2^2) + \ldots + (2^{n-1}))$
$S = n(2^n) - \sum_{i=0}^{n-1} 2^i$
$S = n(2^n) - (2^n - 1)/(2-1) = (n-1)2^n + 1$