Lecture I. (Re)-Introduction to Algorithms

Welcome to CSC 373 – Design of Algorithms. In this class you will learn about several standard techniques for designing efficient algorithms, enabling you to design elegant, optimal algorithms for many problems. Additionally, we will try to teach you about *practical* algorithms. The goal is that after taking this class, when you are given a particular problem, you should be able not only to solve it optimally, but also to write a *fast* algorithm that solves it *well enough*. The algorithm may not be the fastest one; it just has to be fast. And the solution may not be perfect. When you are out in the real world there are many other constraints that will determine what algorithm should be used. For example, consider that you have graduated, and are working as a programmer for a large bank: your boss wants to send a monthly letter to select account holders, who have > $100,000 in their bank accounts, informing them of new investment opportunities. You consider two possible solutions: write a simple program that goes through the accounts, checks the amount in the bank, and sends this letter to the owners, or, alternatively, build a complex data structure that would allow you to immediately identify these accounts much faster. While the first program will have a relatively slow running time – O(n), where n is the number of accounts, it will take you a very short time to write, and since it will only run once a month, this will not be a significant amount of time, even for millions of accounts. In this case you should go with the simpler, rather than the fastest solution – by finishing your task faster you will make your boss happy, earning you a promotion.

Once you are promoted, your boss asks you to write a function that given two numbers, a real number a and an integer n returns the value of a^n. This function will be used to compute the cumulative interest on some savings accounts, and will be called millions of times every day. You write the following simple algorithm (we will use a C-like language for most of our algorithmic examples, though they should be simple enough to understand even if you do not know C):

```
double pow (double a, int n) {
   if (n == 0)
      return 1;
   else
      return a * pow (a, n-1);
}
```

The complexity of this program is also O(n). While this may be good enough if n is small, it will become too slow if n is large, and the algorithm is called millions of times everyday. So when given a problem, before coding anything, try to estimate how long a trivial solution will run and how often it will be called. Ease of implementation is an important consideration when programming in the real world, as if the actual running time of the program, rather than it's big-O complexity. Your boss will not care if you have a theoretically faster algorithm if it is slow in practice. One of the things we will try to show in this class is how to use your knowledge of big-O to decide when it is safe to implement an algorithm that is sub-optimal.

If your boss asks you to rewrite the exponentiation algorithm because it is too slow, you may recall something you learned in middle school and Algebra 1: $a^{2n} = a^n * a^n$. This allows us to halve n at every step: When evaluating a^n for even n we just use the formula above. Otherwise we multiply the result by a, and then use the formula. The following is our resulting code:

```
double qpow(double a, int n) {
   double res;
   if (n == 0)
      return 1;
   res = qpow (a, n/2); // n/2 will round down for odd n
   res = res * res;
   if (n%2 == 1)
      res = res * a;
   return res;
 }
```

Because n is halved at every level of recursion, this implementation will take O(log n) time, rather than the linear time of the previous solution. If n is small this doesn't really matter, but for large numbers the difference will be quite noticeable.

Exponentiation is one example where it is easy to build a very fast, optimal algorithm. While many of the other problems we will see in this class will have similarly elegant solutions, most problems, alas, can not be solved both quickly and optimally. Consider the factorial function: *n! = n * (n-1)!* While the simple, linear algorithm is obvious:

```
double fact(int n) {
   if (n == 0)
      return 1;
   return n * fact(n-1);
}
```

If one needs to compute this value for large *n* this will again be too slow, just like exponentiation. And unlike exponentiation, there is no simple O(log n) algorithm. However there is a well known approximation for the factorial function, known as Stirling's formula: $n! = \sqrt{2\pi n} * (n/e)^n$ Stirling's formula is remarkably close to n!, especially for very large values of n (you can read more about this approximation at http://en.wikipedia.org/wiki/Stirling's_approximation). Using the formula, we can compute a good approximation to the factorial, using the quick exponentiation routine determined above (we would also need a fast sqrt function, but that is beyond the scope of this lecture ☺). The answer will not be exact, but as we will see during this course perfect is not always possible (or, in fact, necessary).

What about for smaller n? In these cases the formula is less accurate, but you can pre-compute these, and store the results in an array. In fact, with modern computers having 2GB or more RAM, if you store (pre-computed) a few thousand factorial values (1000! is thousands of digits long on its own) no one will ever notice. This brings us to another important principle of algorithm design: time-space tradeoffs. In this case, we can *pre-compute* some answers, and while using some space allow for faster computation of factorial functions at runtime. During the course we will see the opposite examples, where by sacrificing some amount of running time we will get an algorithm that uses a practical amount of space.

As an algorithm designer (and implementer) you have to trade-off four critical features: time efficiency, space efficiency, accuracy of the answer, and implementation time. While there is no magic formula that will give you the right tradeoff, we hope that during the course you will learn some of the tricks to doing this effectively.