CSci 243 Homework 5

Due: 10:00 am, Wednesday, Oct 7
Alexander Powell

1. (10 points) Arrange the functions

$$3^n, 2^n, n2^n, n^{30}, (\log n)^3, \sqrt{n}\log^2 n, n\log n, \sqrt{n!}, n^{29}+n^{27}, n^{2\sqrt{n}}$$

into increasing order of growth rates.

Arranging the functions in increasing order of growth we get:

$$(\log n)^3 < \sqrt{n}\log^2 n < n\log n < n^{29}+n^{27} < n^{30} < n^{2\sqrt{n}} < 2^n < n2^n < 3^n < \sqrt{n!}$$

2. (10 points) To solve a particular problem you have access to two algorithms. The execution time of the first algorithm can be given as a function of the input size n as $f(n) = n^{1.5}\log^2 n$. The execution time of the second algorithm is similarly: $g(n) = n^2$. Which algorithm is faster asymptotically? Is this algorithm faster for small n? Find the minimum problem size n needed so that the fastest asymptotic algorithm becomes faster than the other one. Hint: limit your search in powers of 2. You may use calculators to help you but the answer must be self contained.

By plugging in some small values for n we see that initially $f(n)$ appears to grow faster. However, since $g(x)$ has the highest power, it seems likely that for an big enought input, $g(x)$ will eventually overtake $f(n)$. To see if this is true, we can look for an itersection of the two functions. We do this by setting them equal and taking the logarithm of both sides:

$$n^{1.5}\log^2 n = n^2$$

$$\log n^{1.5}\log^2 n = \log n^2$$

$$1.5\log n + \log(\log^2 n) = 2\log n$$

$$\log(\log^2 n) = \frac{1}{2}\log n$$

$$\log(\log^2 n) = \log(\sqrt{n})$$

$$\log^2 n = \sqrt{n}$$

$$\log n = n^{.5\times .5} = n^{.25}$$

$$2^{n^{.25}} = n$$

Now, we have gotten our n in a form of a power of 2 so we can limit our search to that. If we start our search with $n = 2^{14}$ and go from there we get the following results.

n	$f(n)$	$g(n)$
$2^{14} = 16384$	411041792	268435456
$2^{15} = 32768$	1334619360	1073741824
$2^{16} = 65536$	4294967296	4294967296
$2^{17} = 131072$	13713955382	17179869184

From these points, we can see that the two functions cross exactly when $n = 2^{16}$ or 65536. For points smaller than 65536, $f(n)$ takes more time but for inputs larger than 65536, $g(n)$ takes more time. Therefore, we can say that $g(n)$ has a greater asymptotic complexity which means that $f(n)$ is asymptotically faster, even though it is not faster for small n, specifically those smaller than 65536.

Because the two algorithms have the same speed when operating on a problem size of $n = 2^{16}$ or 65536, then the minimum problem size n needed so that $f(n)$ becomes faster is one more than that, or 65537.

3. (10 points) What is the largest problem size n that we can solve in no more than **one hour** using an algorithm that requires $f(n)$ operations, where each operation takes 10^{-9} seconds (this is close to a today's computer), with the following $f(n)$?

   First, it will be easier to deal with everything in terms of seconds, so there are 3600 seconds in one hour. Now, we just need to multiply every $f(n)$ by 10^{-9}, set it equal to 3600 and solve for n.

   (a) $\log_2 n$
   Solution:

   $$3600 = \log_2 n \times 10^{-9}$$

   $$\frac{3600}{10^{-9}} = \log_2 n \longrightarrow n = 2^{\frac{3600}{10^{-9}}}$$

   (b) $\log_2^4 n$
   Solution:
   Similarly to the above problem, we get

   $$n = 2^{\sqrt[4]{\frac{3600}{10^{-9}}}}$$

   (c) $3n$
   Solution:

   $$3600 = 3n \times 10^{-9} \longrightarrow n = \frac{3600}{3 \times 10^{-9}}$$

   (d) $n \log_2 n$
   Solution:
   $3600 = n \log_2 n \times 10^{-9}$
   $\frac{3600}{10^{-9}} = \log_2 n$

   $$2^{\frac{3600}{10^{-9}}} = n$$

   From here it is very difficult to solve for n, however we can conclude that it is an extremely large number

   (e) $n \log_2^2 n$
   Solution:
   $3600 = n \log_2^2 n \times 10^{-9}$

Again, we can rewrite this equation to look like

$$2^{\sqrt{\frac{3600}{10^{-9}}}} = n$$

Which can't easily be solved by hand but again, we get a large number, meaning this is a very good algorithm to use when working with large problem sizes.

(f) n^2

Solution:

$$3600 = n^2 \times 10^{-9} \longrightarrow n = \sqrt{\frac{3600}{10^{-9}}}$$

(g) $(3n)^3$

Solution:

$$3600 = (3n)^3 \times 10^{-9} \longrightarrow n = \frac{\sqrt[3]{\frac{3600}{10^{-9}}}}{3}$$

(h) 2^n

Solution:

$3600 = 2^n \times 10^{-9}$

$\log_2 3600 = \log_2 (2^n \times 10^{-9})$

$\log_2 3600 = \log_2 (2^n + \log_2 10^{-9})$

$\log_2 3600 = n + \log_2 10^{-9}$

$n = \log_2 3600 - \log_2 10^{-9}$

$n = \log_2 \left(\frac{3600}{10^{-9}}\right)$

(i) $n!$

Solution:

Because $n!$ is so fast growing, it's not necessary to compute this algebraicly. By inputting a few numbers into the equation:

$$3600 = n! \times 10^{-9}$$

we can quickly see what the largest problem size possible is before going over an hour. An input of size 14 will take $14! \times 10^{-9} \approx 87$ seconds. Input size 15 gives us $15! \times 10^{-9} \approx 1307$ seconds. The next largest input size, 16, gives us the following: $16! \times 10^{-9} \approx 20922$ seconds. Clearly 20922 is greater than 3600 so the largest problem size n that takes no longer than one hour is $n = 15$.

(j) n^n

Solution:

We will use a process similar to the previous problem to determine the max problem size. We know that n^n grows faster than $n!$ so we can start guessing at numbers a little lower than before. Let's begin with a guess of $n = 10$, then $10^{10} \times 10^{-9} = 10$. When $n = 11$ we have $11^{11} \times 10^{-9} \approx 285$. When $n = 12$ we have $12^{12} \times 10^{-9} \approx 8916$, which is clearly over 3600, so the largest problem size n that takes no longer than one hour is $n = 11$.

4. (10 points) Use pseudocode to describe an algorithm that determines whether a given function from a finite set to another finite set is one-to-one.

   Hint: You may assume that the domain is $A = \{a_1, \ldots, a_m\}$ and the co-domain is $B = \{b_1, \ldots, b_n\}$. The function $f : A \to B$ is given as a set of pairs $\{(a_i, f(a_i)) | \forall a_i \in A\}$.

   To determine if the given function is one-to-one, we just need to determine if there are two or more inputs that map to the same output. In other words, we need to examine our input elements in $f(a_i)$ and determine if any duplicates exist. If there are no duplicates in $f(a_i) \forall a_i \in A$, we can conclude the function is one-to-one. If at least one duplicate does exist, the function is not one-to-one. The pseudocode for the algorithm is shown below.

```
Inputs: Two arrays, one of a_i and one of f(a_i), where the input and output
mappings are related through their index position.

var length = lengthOfArray(a_i)
for (int x = 0; x < length; x++)
{
    for (int y = x; y < length; y++)
    {
        if (f(a_i)[x] == f(a_i)[y])
        {
            return false;
        }
    }
}
return true;
```

   Also, it's important to note this algorithm assumes it's given a valid function as input and there are no redundant inputs in the set of pairs (meaning two identical ordered pairs are not listed twice).