CS 5050: Homework 1

Andrew Pound

September 10, 2014

I worked on this assignement with Chad Cummings.

1

(a) Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n \log n$ steps. For which values of n does insertion sort beats merge sort? **(5 points)**

There are a few different ways to do this. The first way that we attacked it was that we solved the inequality equation to easiest way to solve this is to tabulate the answers for each side of the equatoin and find the point that they cross. (This would actually get a bit long for this particular problem). So instead a plot of the two functions can be used to see where it crosses.

Equations:

$$\begin{aligned} 8n^2 &\leq 64n \log n \\ \frac{1}{8} &\leq \frac{1}{n} \log n \\ 2^{\frac{1}{8}} &\leq n^{\frac{1}{n}} \end{aligned} \tag{1}$$

Plotting both sides of this, produces the graph below.

From this we can see that the cross over point is around 43. Tabulating the values of $n^{1/n}$ in Table 1, the neighborhood of 43 and comparing them with the value of $2^{1/8} = 1.0905$, we can see that any number greater than 43 satisfies the inequality.

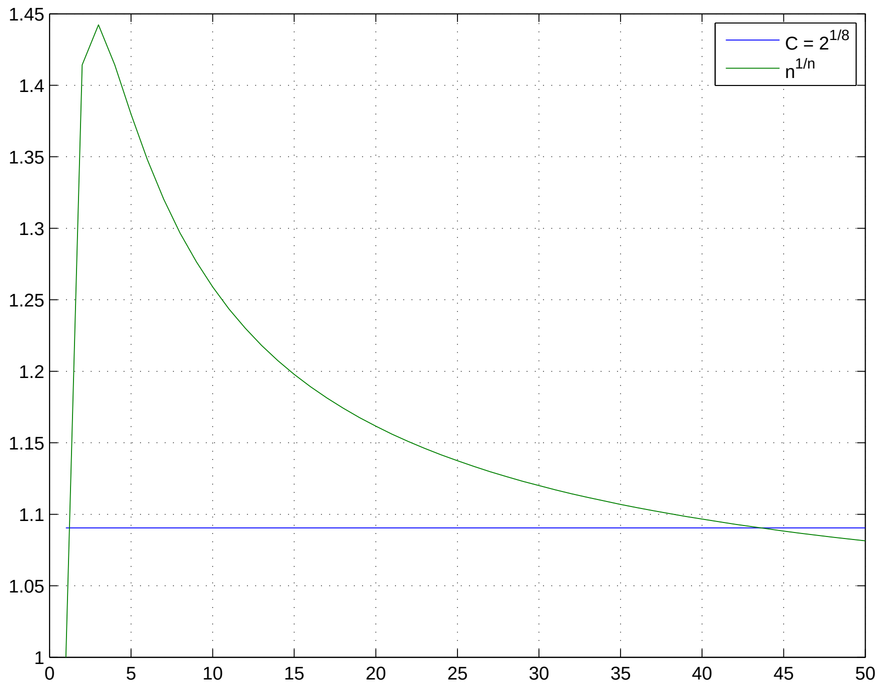

Figure 1: Plot of the inequality in equation 1.

Table 1: Tabulation of values around $n = 43$.

n	$n^{1/n}$
40	1.09661
41	1.0948
42	1.09307
43	1.09141
44	1.08981
45	1.08827

(b) What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine? **(5 points)**

This can be solved in much the same manner as the previous problem. First, Let's examine the inequality.

$$\begin{aligned}
100n^2 &\leq 2^n \\
\log_2(100n^2) &\leq log_2(2^n) = n \\
\frac{2}{n}\log(10n) &\leq 1 \\
\frac{1}{n}\log(10n) &\leq \frac{1}{2} \\
\log\left(n^{\frac{1}{n}}\right) &\leq \frac{1}{2} \\
(10n)^{\frac{1}{n}} &\leq \sqrt{2}
\end{aligned} \tag{2}$$

The plot of this function is seen in Figure 2, and the tabulation around the cross over is in Table 2. This time we ar comparing with the value $\sqrt{2} = 1.4142$, and we can see that the cross over happens at $n = 15$.

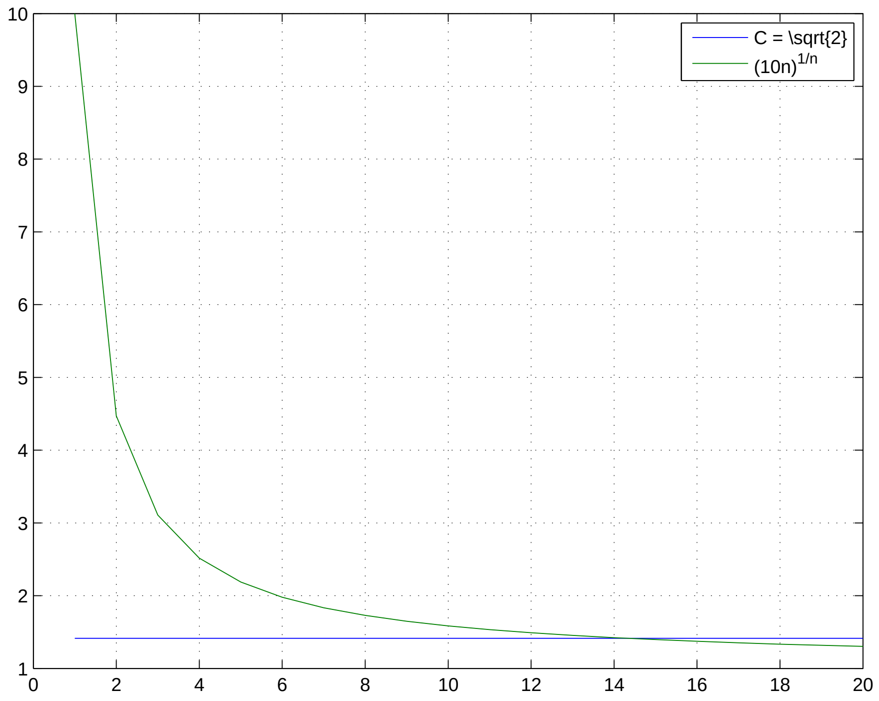

Figure 2: Plot of the inequality in equation 2.

Table 2: Tabulation of values around $n = 15$.

n	$(10n)^{1/n}$
12	1.23008
13	1.21811
14	1.20744
15	1.19786
16	1.18921
17	1.18135

2

We have discussed the insertion sort algorithm in class and we showed that in the worst case the algorithm runs in $\Theta(n^2)$ time. You may wonder whether the algorithm can run asymptotically faster for certain "average" cases. In this exercise, we will examine the running time of the insertion sort algorithm on some average cases. Let A be the input array of n elements.

Note: For each of the questions, please give your answers in the big-Theta notation and also briefly explain how you obtain your answer.

(a) Consider the case where the first $n/2$ elements of A have already been sorted *increasingly* and the last $n/2$ elements have been sorted *decreasingly*. For example, $A = \{1, 3, 5, 7, 8, 6, 4, 2\}$. Suppose we want to sort all elements of A in increasing order by using the insertion sort algorithm. What would be the running time of the algorithm? What if the first $n/2$ elements of A have already been sorted decreasingly and the last $n/2$ elements have been sorted increasingly? For example, $A = \{8, 6, 4, 2, 1, 3, 5, 7\}$. What would be the running time in this case? **(10 points)**

The insertion sort algorithm that we discussed in class is given as

```
for $i = 2 \ldots n$ do
    $x \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $A[j-1] < x$ & $j \geq 1$ do
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    end while
    $A[j+1] \leftarrow x$
end for
```

The outer for loop consists of $4c$ operations, and the inner loop (when run) consists of $2c$ operations. Now suppose that the elements of our array are ordered in such a way that the first half of the array are already in ascending order, and the second half is the worst case (descending order) and need to be merged into the first half. If that is the case, then the first half of the algorithm is a best case scenario, and only uses $4c\frac{n}{2}$ operations. The second half, though will utilize the inner for-loop to move some of the elements in the first half of the array in order to insert in the correct place. By considering the first elements in the second half, we can determine a relation between the number of times in the inner loop and the iteration index. If the array is in the form specified, then the first element of the second half will not require any moving to place it. The second will require 2 moves (the first element of the 2nd half and the last element of the first half). The third element in the 2nd half will require 4 moves. We see that for $i > \frac{n}{2}$, the number of times in the inner loop will be $2(i - \frac{n}{2})$. Thus, we can stick this into an expression for the total number of operations

$$\underbrace{4cn}_{\text{outer loop}} + \underbrace{\sum_{i=\frac{n}{2}}^{n} 2\left(i - \frac{n}{2}\right)(2c)}_{\text{inner loop}}. \tag{3}$$

Now expanding this, we get a total of

$$\begin{aligned}(4cn) + 4c\sum_{i=0}^{n/2} i &= 4cn + 4c\frac{\frac{n}{2}\left(\frac{n}{2}+1\right)}{2} \\ &= 4cn + 2c\left(\frac{1}{4}n^2 + \frac{1}{2}n\right) \\ &= 4cn + \frac{1}{2}n^2c + cn \\ &= \frac{1}{2}cn^2 + 5cn \\ &= O(n^2)\end{aligned} \tag{4}$$

(b) Consider the case where the elements of A with odd indices have already been sorted increasingly and elements with even indices have been sorted decreasingly (we assume the index of A starts from 1). For example, $A = \{1, 8, 2, 7, 3, 6, 4, 5\}$. Suppose we want to sort all elements of A in increasing order by using the insertion sort algorithm. What would be the running time of the algorithm?

What if the elements of A with odd indices have already been sorted *decreasingly* and elements with even indices have been sorted *increasingly*. For example, $A = \{8, 1, 7, 2, 6, 3, 5, 4\}$. What would be the running time in this case? **(10 points)**

Ok, we'll begin by looking at how the algorithm works on the example sequence. The results of the steps of the algorithm are shown below. The bolded numbers identify values that needed to be moved (i.e. the values for which the inner loop was used). The first line is the original sequence.

$$\begin{array}{cccccccc}
1 & 8 & 2 & 7 & 3 & 6 & 4 & 5 \\
1 & & & & & & & \\
1 & 8 & & & & & & \\
1 & 2 & \mathbf{8} & & & & & \\
1 & 2 & 7 & \mathbf{8} & & & & \\
1 & 2 & 3 & \mathbf{7} & \mathbf{8} & & & \\
1 & 2 & 3 & 6 & \mathbf{7} & \mathbf{8} & & \\
1 & 2 & 3 & 4 & \mathbf{6} & \mathbf{7} & \mathbf{8} & \\
1 & 2 & 3 & 4 & 5 & \mathbf{6} & \mathbf{7} & \mathbf{8}
\end{array} \tag{5}$$

From this we can see that for each pair of iterations, the number of moves ends up being predictable. In order to count the time, we then start with the equation

$$\underbrace{4cn}_{\text{outer loop}} + \underbrace{2\sum_{i=1}^{n/2}(i-1)(2c)}_{\text{inner loop}}. \tag{6}$$

The outer loop has $4c$ operations and happens for each element in the array, thus we get $4cn$. Each pair of iterations is hitting the inner loop for the same number of times, namely $i-1$, for the ith pair. The inner loop consists of $2c$ operations, thus we get the summation over all the pairs, and multiply by 2 to get the full effect of the inner loop. This is

$$\begin{aligned}
2\sum_{i=1}^{n/2}(i-1)(2c) &= 4c\sum_{i=1}^{n/2}(i-1) \\
&= 4c\sum_{i=1}^{n/2} i - 4c\frac{n}{2} \\
&= 4c\frac{\frac{n}{2}\left(\frac{n}{2}+1\right)}{2} - 4c\frac{n}{2} \\
&= 2c\left(\frac{n^2}{4}+\frac{n}{2}-n\right) \\
&= 2c\left(\frac{1}{4}n^2-\frac{1}{2}n\right)
\end{aligned} \tag{7}$$

The entire operations count together then would be

$$\begin{aligned}
4cn + 2c\left(\frac{1}{4}n^2-\frac{1}{2}n\right) &= 4cn + \frac{1}{2}cn^2 - cn \\
&= \frac{1}{2}cn^2 + 3cn \\
&= O(n^2)
\end{aligned} \tag{8}$$

Now for the other example, let's see what it looks like:

$$\begin{array}{cccccccc}
8 & 1 & 7 & 2 & 6 & 3 & 5 & 4 \\
8 & & & & & & & \\
1 & \mathbf{8} & & & & & & \\
1 & 7 & \mathbf{8} & & & & & \\
1 & 2 & \mathbf{7} & \mathbf{8} & & & & \\
1 & 2 & 6 & \mathbf{7} & \mathbf{8} & & & \\
1 & 2 & 3 & \mathbf{6} & \mathbf{7} & \mathbf{8} & & \\
1 & 2 & 3 & 5 & \mathbf{6} & \mathbf{7} & \mathbf{8} & \\
1 & 2 & 3 & 4 & \mathbf{5} & \mathbf{6} & \mathbf{7} & \mathbf{8}
\end{array} \tag{9}$$

Essentially, the idea is the same, in that the iterations can be split into pairs such that a pair will use the inner loop (to move larger numbers) the same number of times. Thus the analysis is (almost) exactly the same: $O(n^2)$.

3

For each of the following pairs of functions, indicate whether it is one of the three cases: $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, or

$f(n) = \Theta(g(n))$. (30 points)

(a) $f(n) = 100n + \log n$ and $g(n) = 6n + \log^2 n$.

$f(n) = \Omega(g(n))$. My reasoning is this:

$$\begin{aligned} f(n) &\; ?\; g(n) \\ 100n + \log n &\; ?\; 6n +]log^2 n \\ 94n &\; ?\; \log^2 n - \log n \end{aligned} \tag{10}$$

Now, $\log^2 n - \log n < \log^2 n$, and comparing this as above, we see that

$$94n > \log^2 n > \log^2 n - \log n. \tag{11}$$

Thus, we can say that $f(n) = \Omega(g(n))$.

(b) $f(n) = 20 \log n + 4$ and $g(n) = \log n^2 - 100$.

$f(n) = \Theta(g(n))$. Playing with the equations a bit, we can see

$$\begin{aligned} f(n) &\; ?\; g(n) \\ 20 \log n + 4 &\; ?\; \log n^2 - 100 \\ 20 \log n + 4 &\; ?\; 2 \log n - 100. \end{aligned} \tag{12}$$

Now, if we choose $c_1 = 20$ and $c_2 = 10$, then we get an asymptotic upper and lower bound, respectively, by forming the equations $c_i g(n)$. Thus, we know that $f(n) = \Theta(g(n))$.

(c) $f(n) = \frac{n^2}{\log n}$ and $g(n) = n \log^2 n$.

$f(n) = \Omega(g(n))$. Let's play with the equations:

$$\begin{aligned} \frac{n^2}{\log n} &\; ?\; n \log^2 n \\ n &\; ?\; \log^3 n. \end{aligned} \tag{13}$$

Then using a nifty formula:

$$\log^k n = O(n^d) \quad \forall\, k > 0,\ d > 0, \tag{14}$$

we can see that $d = 1$, and $k = 3$, and thus, $g(n) = O(f(n))$. Or in other words, $f(n) = \Omega(g(n))$.

(d) $f(n) = \sqrt{n}$ and $g(n) = \log^5 n$.

$f(n) = \Omega(f(n))$. Using the same nifty equation from above, we can see that $g(n) = O(f(n))$, as $k = 5$, and $d = 1/2$. Thus flipping this around, we see that $f(n) = \Omega(f(n))$.

(e) $f(n) = n2^n$ and $g(n) = 3^n$.

$f(n) = O(g(n))$. We will make use of our other nifty formula

$$n^d = o(e^n), \quad \forall\, e > 1. \tag{15}$$

First, let's play a bit with our equation

$$\begin{aligned} n2^n &\,?\, 3^n \\ n &\,?\, \left(\frac{3}{2}\right)^n . \end{aligned} \tag{16}$$

This helps us to see that when $d = 1$ and $e = 3/2$, then we know that $f(n) = o(g(n))$ and therfore, $f(n) = O(g(n))$.

(f) $f(n) = 4n \log n$ and $g(n) = n \log_3 n$.

$f(n) = \Theta(g(n))$. Playing with the equation, we get

$$\begin{aligned} 4n \log n &\,?\, n \log_3 n \\ 4 \log n &\,?\, \frac{\log n}{\log 3} \\ 4 \log n &\,?\, \frac{1}{\log 3} \log n \\ 4 &\,?\, \frac{1}{\log 3} \end{aligned} \tag{17}$$

Now we see that $f(n) = \Theta(g(n))$.

Note: For each question, you only need to give your answer and the proof is not required.

4

The *knapsack problem* is defined as follows: Given as input a knapsack of size K and n items whose sizes are $k_1, k_2, \ldots, kn$, where K and $k_1, k_2, \ldots, k_n$ are all real numbers, find a full "packing" of the knapsack (i.e., choose a subset of the n items such that the total sum of the sizes of the items in the chosen subset is exactly K).

It is well known that the knapsack problem is NP-complete, which implies that it is very likely that efficient algorithms (i.e., those with a polynomial running time) for this problem do not exist. Thus, people tend to look for good **approximation algorithms** for solving this problem. In this exercise, we relax the constraint of the knapsack problem as follows. We still seek a packing of the knapsack, but we need not look for a "full" packing of the knapsack; instead, we look for a packing of the knapsack (i.e., a subset of the n input items) such that the total sum of the sizes of the items in the chosen subset is at least $K/2$ (but no more than K). This is called a *factor of 2 approximation solution* for the knapsack problem. To simplify the problem, we assume that a factor of 2 approximation solution for the knapsack problem always exists, i.e, there always exists a subset of items whose total size is at least $K/2$ and at most K.

Design a polynomial time algorithm for computing a factor of 2 approximation solution for this problem, and analyze the running time of your algorithm (in the big-O notation). If your algorithm runs in $O(n)$ time and is correct, then you get **5 extra points**. **(20 points)**

Note: You are required to clearly describe the main idea of your algorithm. Although the pseudo-code is not required, you may also give the pseudo-code if you feel it is helpful for you to describe your algorithm. (The reason I want to see the algorithm description instead of only the code or pseudo-code is that it would be difficult to understand another person's code without any explanation.) You also need to briefly explain why your algorithm works, i.e., why your algorithm can produce a factor of 2 approximation solution. Finally, please analyze the running time of your algorithm.

My algorithm In order to find a factor of 2 approximation, we will make use of the assumptions that there definitely exists such an approximation, and that all of the k_i sizes are positive.

The algorithm is as follows:

Iterate through the list by pairs, finding the largest of each pair and moving it to the lower position in the pair. Then iterate and add the largest of each pair (unless it is too large (i.e. makes the knapsack over full). Then, if at the end of this loop, that knapsack is not at least 1/2 full, then iterate through the smaller of the pairs, adding them, unless they overfill the knapsack.

Because there exists a factor of 2 approximation, this algorithm will terminate when either the solution is found or the entire list is added during the 3rd loop, which will then constitute a solution (because one has to exist).

Pseudocode for the algorithm can be laid out as

```
$cap \leftarrow$ Knapsack size/capacity
**for** $i = 1, 3, \ldots, n$ **do**
    **if** $k[i] \leq k[i+1]$ **then**
        Swap($k[i], k[i+1]$)
    **end if**
**end for**
$Sz = 0$
// Let's fill in the larger sizes first
**for** $i = 1, 3, \ldots, n$ **do**
    **if** $Sz + k[i] \leq cap$ **then**
        $Sz \leftarrow Sz + k[i]$
    **end if**
    **if** $Sz \geq \frac{cap}{2}$ **then**
        return
    **end if**
**end for**
// Now, start filling in the smaller sizes
**for** $i = 2, 4, \ldots, n$ **do**
    **if** $Sz + k[i] \leq cap$ **then**
        $Sz \leftarrow Sz + k[i]$
    **end if**
    **if** $Sz \geq \frac{cap}{2}$ **then**
        return
    **end if**
**end for**
```

The runtime for this algorithm is $O(\frac{3}{2}n)$, because at the most it will utilize all three for-loops which are each over only half the length of the array.