CS 330: Homework 1

Michael Lin

September 15, 2015

Problem 1

(a) Since logarithmic transformation retains relative growth rate between functions, we will first transform the functions so that we have:

$$n\log(5/4), 4\log n, \log(n!), \sqrt{2\log n}\log 2, \log(\log n), (\log n)[\log(\log n)], \log(n)\log\sqrt{2}, 2^n\log 2$$

Given that $\log(n!) = O(n\log n)$ and $O(1) < O(\log(\log n)) < O(\log n) < O(n) < O(n\log n) < O(2^n)$, we have the functions in increase order of growth:

$$\log n < (2)^{\sqrt{2\log n}} < (\sqrt{2})^{\log n} < n^4 < (\log n)^{\log n} < (5/4)^n < n! < 2^{2^n}$$

(b) First, we prove $n\log n$ is an upper bound.

$$\begin{aligned}\log(n!) &= \log n + \log(n-1) + \cdots + \log 1\\ &\leq \log n + \log n + \cdots + \log n\\ &= n\log n\end{aligned}$$

Hence, $\log(n!) = O(n\log n)$. Now, we prove $n\log n$ is also a lower bound.

$$\begin{aligned}\log(n!) &= \log n + \log(n-1) + \cdots + \log(n/2) + \cdots + \log 1\\ &\geq \log n + \log(n-1) + \cdots + \log(n/2)\\ &\geq \log(n/2) + \log(n/2) + \cdots + \log(n/2)\\ &= (n/2)\log(n/2)\\ &= (n/2)\log n - (n/2)\end{aligned}$$

where the last step assumed log with base 2. If we choose $c = 0.1$ and $n_0 = 4$, then for all $n \geq n_0$:

$$cn\log n \leq (n/2)\log n - (n/2)$$

Thus, $\log(n!) = \Omega(n\log n)$. Since $\log(n!) = O(n\log n) = \Omega(n\log n)$, therefore

$$\log(n!) = \Theta(n\log n)$$

Problem 2

- $T(n) = 5T(n/4) + n$
  First, solve the recurrence relation in closed form:

$$\begin{aligned} T(n) &= 5T(n/4) + n \\ &= 5(5T(n/4^2) + n/4) + n = 5^2T(n/4^2) + (5/4)n + n \\ &= 5^2(5T(n/4^3) + n/4^2) + (5/4)n + n = 5^3T(n/4^3) + (5/4)^2n + (5/4)n + n \\ &\vdots \\ &= 5^kT(n/4^k) + n\sum_{i=0}^{k-1}(5/4)^i \\ &= 5^kT(n/4^k) + n\frac{(5/4)^k - 1}{(5/4) - 1} \\ &= 5^kT(n/4^k) + 4n(5/4)^k - 4n \end{aligned}$$

To find $T(0)$, let $n/4^k = 0 \implies k = \log_4 n$. Using the fact that $x^{\log_4 n} = n^{\log_4 x}$ (see proof):

Proof: $x^{\log_4 n} = n^{\log_4 x}$.

$$\begin{aligned} a &= x^{\log_4 n} \\ \log_4 a &= \log_4(x^{\log_4 n}) = (\log_4 n)(\log_4 x) = \log_4 n^{\log_4 x} \\ a &= n^{\log_4 x} \end{aligned}$$

□

and suppose $T(0) = 1$, substitute into the recurrence relation to get:

$$\begin{aligned} T(n) &= 5^{\log_4 n} \cdot T(0) + 4n(5/4)^{\log_4 n} - 4n \\ &= n^{\log_4 5} + 4n(n^{\log_4(5/4)}) - 4n \\ &= n^{\log_4 5} + 4n(n^{(\log_4 5)-1}) - 4n \\ &= n^{\log_4 5} + 4n^{\log_4 5} - 4n \\ &= 5n^{\log_4 5} - 4n \end{aligned}$$

Since $\log_4 5 > 1$, we have $n^{\log_4 5} > n$. Therefore,

$$T(n) = \Theta(n^{\log_4 5})$$

Using $n^{\log_4 5}$ as a "guess", prove through induction that $T(n) = \Theta(n^{\log_4 5})$.

Proof: $T(n) = \Omega(n^{\log_4 5})$*.* Equivalently, we want to prove that $T(n) \geq cn^{\log_4 5}$ for some real constant c.

Base case: $T(1) = 1 \geq c$ if we let $c = 1$.

Hypothesis: Suppose $T(m) \geq cm^{\log_4 5}$ for all $m < n$.

Inductive step: Since $n/4 < n$, we have:

$$\begin{aligned} T(n) &= 5T(n/4) + n \\ &\geq 5c(n/4)^{\log_4 5} + n \\ &\geq 5c(n/4)^{\log_4 5} = 5c \cdot \frac{n^{\log_4 5}}{4^{\log_4 5}} = 5c \cdot \frac{n^{\log_4 5}}{5} \\ &= cn^{\log_4 5} \end{aligned}$$

Since $T(n) \geq cn^{\log_4 5}$ for all n and c (let $c = 1$ so that it's consistent with the base case), $T(n) = \Omega(n^{\log_4 5})$. $\square$

Proof: $T(n) = O(n^{\log_4 5})$. Equivalently, we want to prove that $T(n) \leq cn^{\log_4 5} - 4n$ for some real constant c.
Base case: $T(1) = 1 \leq c - 4$ if we let $c = 6$.

Hypothesis: Suppose $T(m) \leq cm^{\log_4 5} - 4m$ for all $m < n$.

Inductive step: Since $n/4 < n$, we have:

$$\begin{aligned} T(n) &= 5T(n/4) + n \\ &\leq 5[c(n/4)^{\log_4 5} - 4(n/4)] + n \\ &= 5(c \cdot \frac{n^{\log_4 5}}{5} - n) + n \\ &= cn^{\log_4 5} - 5n + n \\ &= cn^{\log_4 5} - 4n \end{aligned}$$

Since $T(n) \leq cn^{\log_4 5} - 4n$ for all n and c (let $c = 6$ so that it's consistent with the base case), $T(n) = O(n^{\log_4 5})$. $\square$

Proof: $T(n) = \Theta(n^{\log_4 5})$. Since $T(n) = \Omega(n^{\log_4 5})$ and $T(n) = O(n^{\log_4 5})$, therefore $T(n) = \Theta(n^{\log_4 5})$. $\square$

- $T(n) = \sqrt{n}T(\sqrt{n}) + n \log n$
Suppose $\log n := \lg n$. First, solve the recurrence relation in closed form:

$$\begin{aligned} T(n) &= n^{1/2}T(n^{1/2}) + n \log n \\ &= n^{1/2}[n^{(1/2)^2}T(n^{(1/2)^2}) + n^{1/2} \lg n^{1/2}] + n \lg n \\ &= n^{1/2+(1/2)^2}T(n^{(1/2)^2}) + (1 + 1/2)n \lg n \\ &= n^{1/2+(1/2)^2}[n^{(1/2)^3}T(n^{(1/2)^3}) + n^{(1/2)^2} \lg n^{(1/2)^2}] + (1 + 1/2)n \lg n \\ &= n^{1/2+(1/2)^2+(1/2)^3}T(n^{(1/2)^3}) + (1 + 1/2 + (1/2)^2)n \lg n \\ &\vdots \\ &= n^{\sum_{i=1}^{k}(1/2)^i}T(n^{(1/2)^k}) + \Big(\sum_{j=0}^{k-1}(1/2)^j\Big) \\ &= n^{\frac{(1/2)[1-(1/2)^k]}{1/2}}T(n^{(1/2)^k}) + (\frac{1-(1/2)^k}{1/2})n \lg n \\ &= n^{1-(1/2)^k}T(n^{(1/2)^k}) + 2(1 - (1/2)^k)n \lg n \end{aligned}$$

To find $T(2)$, let $n^{(1/2)^k} = 2 \implies (1/2)^k \lg n = 1 \implies \lg n = 2^k \implies k = \lg(\lg n)$. Using the fact that $(1/2)^{\lg(\lg n)} = 1/(\lg n)$ (see proof):

Proof: $(1/2)^{\lg(\lg n)} = 1/(\lg n)$.

$$\begin{aligned} a &= (1/2)^{\lg(\lg n)} \\ \lg a &= \lg(\lg n)\lg(1/2) = \lg[(\lg n)^{-1}] \\ a &= (\lg n)^{-1} \end{aligned}$$

$\square$

and suppose $T(2) = 2$, substitute into the recurrence relation to get:

$$\begin{aligned}T(n) &= n^{1-(\lg n)^{-1}}T(2) + 2(1-(\lg n)^{-1})n\lg n \\ &= \Big(\frac{n}{n^{(\lg n)^{-1}}}\Big)T(2) + 2n\lg n - 2n \\ &= (n/2)\cdot 2 + 2n\lg n - 2n \\ &= 2n\lg n - n \\ &= 2n\log n - n\end{aligned}$$

since we defined that $\log n := \lg n$. Therefore, we have

$$T(n) = \Theta(n\log n)$$

Using $n\log n$ as a "guess", prove through induction that $T(n) = \Theta(n\log n)$.

Proof: $T(n) = \Theta(n\log n)$. Equivalently, we want to prove that $T(n) \geq cn\log n$ for some real constant c. Suppose $T(1) = 1$.
Base case: $T(1) = 1 \geq 0$.

Hypothesis: Suppose $T(m) \geq cm\log m$ for all $m < n$.

Inductive step: Since $\sqrt{n} < n$, we have:

$$\begin{aligned}T(n) &= \sqrt{n}T(\sqrt{n}) + n\log n \\ &\geq \sqrt{n}\cdot c\sqrt{n}\log(\sqrt{n}) + n\log n \\ &= (1/2)cn\log n + n\log n \\ &= [(1/2)c+1]n\log n \\ &\geq cn\log n\end{aligned}$$

if we choose $c = 2$. Since $T(n) \geq cn\log n$ for all n, $T(n) = \Omega(n\log n)$. □

Proof: $T(n) = O(n\log n)$. Equivalently, we want to prove that $T(n) \leq cn\log n + n$ for some real constant c.

Base case: $T(1) = 1 \leq 1$

Hypothesis: Suppose $T(m) \leq cm\log m + m$ for all $m < n$.

Inductive step: Since $\sqrt{n} < n$, we have:

$$\begin{aligned}T(n) &= \sqrt{n}T(\sqrt{n}) + n\log n \\ &\leq \sqrt{n}[c\sqrt{n}\log(\sqrt{n}) + \sqrt{n}] + n\log n \\ &= (1/2)cn\log n + n + n\log n \\ &= ((1/2)c+1)n\log n + n \\ &\leq cn\log n + n\end{aligned}$$

if we choose $c = 3$. Since $T(n) \leq cn\log n + n$ for all n, $T(n) = O(n\log n)$. □

Proof: $T(n) = \Theta(n\log n)$. Since $T(n) = \Omega(n\log n)$ and $T(n) = O(n\log n)$, therefore $T(n) = \Theta(n\log n)$. □

Problem 3

Algorithm for finding kth smallest value (see Algorithm 1).

Algorithm 1 Find kth smallest value

```
function kFind(list $A$, int $m$, list $B$, int $n$, int $k$)
    $i = \lfloor (m-1)/2 \rfloor$
    $j = \lfloor (n-1)/2 \rfloor$
    $q = i + j + 2$

    BASE CASES
    if $k = 1$ then
        return $\min(A[0], B[0])$
    end if

    if $k = m + n$ then
        return $\max(A[m], B[n])$
    end if

    RECURSIVE CASES
    if $A[i] < B[j]$ then
        if $q = k$ then
            return kFind($A[i+1 : m-1]$, $m-(i+1)$, $B[0 : j]$, $j+1$, $k-(i+1)$)
        else if $q > k$ then
            return kFind($A$, $m$, $B[0 : j-1]$, $j$, $k$)
        else if $q < k$ then
            return kFind($A[i+1 : m-1]$, $m-(i+1)$, $B$, $n$, $k-(i+1)$)
        end if
    end if

    if $A[i] > B[j]$ then
        if $q = k$ then
            return kFind($A[0 : i]$, $i+1$, $B[j+1 : n-1]$, $n-(j+1)$, $k-(j+1)$)
        else if $q > k$ then
            return kFind($A[0 : i-1]$, $i$, $B$, $n$, $k$)
        else if $q < k$ then
            return kFind($A$, $m$, $B[j+1 : n-1]$, $n-(j+1)$, $k-(j+1)$)
        end if
    end if
end function
```

Proof of correctness. Algorithm 1 recursively finds the median of the list and retains only useful part. Suppose i is the index of median of A and j is the index of median of B, and $A[i] < B[j]$. If the number of elements on the left sides of lists A and B is greater than k (i.e. $i+j+2 > k$), then the kth smallest element cannot be on right side of B since the lists are sorted (see this by contradiction: suppose kth smallest element is on the right side of B. Since $B[j] > A[i]$, then there are at least $i+j+2$ elements left of the kth smallest element. However, this cannot be by the invariant $i+j+2 > k$).

If the number of elements on the left sides of lists A and B is less than k (i.e. $i+j+2 < k$), then the kth smallest element cannot be on the left side of A since the lists are sorted (again, this can be seen by the same logic as above).

If the number of elements on the left sides of lists A and B is equal to k (i.e. $i+j+2 = k$), then the kth smallest element cannot be on the left side of A (see this by contraction: suppose k smallest element is on the left side of A, say at index $i* < i$. This can only be if the first $j*$ elements in list B are all less than

$A[i*]$. By the invariant $i + j + 2 = k$, $j*$ must then be greater than j, which implies $B[j*] < A[i]$. However, this is a contradiction since $A[i] < B[j] < B[j*]$). By similar logic, the kth smallest element cannot be on the right side of B.

The side(s) on which the kth smallest element does not reside is discarded, and k is updated when the left side of the lists are discarded to reflect the fact that the first i elements of A or the first j elements of B are discarded. When both lists only have 1 element each, then k will be 1 or 2, and the base cases will ensure the correct result is returned. $\square$

Proof of runtime. Each base case runs in constant time. Each recursive call discards half of at least one of the lists, which also runs in constant time. Since list A will discard by half up to $\log m$ times and list B will discard by half up to $\log n$ times, the runtime of algorithm 1 is $O(\log m + \log n)$. $\square$

Problem 4

Algorithm for counting number of inversions (see Algorithm 2). This algorithm is essentially the merge sort procedure with a counter that counts the number of inversions.

Recursively split list into halves and merge sort. In the base case, where only two numbers are compared, increment number of inversions by 1 if the first number if larger than the second number. When comparing two sublists A with index i and B with index j, begin pointer i at $A[0]$ and pointer j at $B[0]$. Besides this addition, the algorithm is exactly the same as standard merge sort algorithm (see lecture notes and [Er] from class reading).

Proof of correctness. Consider the first case where we have a list A with only 2 numbers. Performing the merge sort procedure on A would simply be a comparison between the two numbers. If $a_0 < a_1$, then i increments by 1, and counter increments by j, which is 0. So counter is 0, which is correct since there is no inversion. If $a_0 > a_1$, then j increments by 1. After that, i increments by 1, and counter increments by j, which is 1. So counter is 1, which is correct since there is 1 inversion.

Now suppose the second case where we have a list A with n elements which has been divided into two lists, each of which has been sorted by merge sort. Call the left list B and the right list C. If $b_i < c_j$, then i increments by 1, and counter increments by j. This is a correct procedure because the j elements less than c_j, i.e. $c_0, c_1, \ldots, c_{j-1}$, are all less than b_i, which gives j inversions. If $b_i > c_j$, then j increments by 1, and counter stays the same because otherwise we would be double counting the number of inversions.

Observe that the total number of inversions is the sum of number of inversions in the unsorted left list, number of inversions in the the unsorted right list, and number of inversions between the left and right lists. Since the merge sort eventually divides the array into just two elements, the first case will ensure the number of inversions in the base case is correct. Once sublists are sorted and merged, the second case will ensure the number of inversions between sublists is correct. However, this is essentially the same as number of inversions within the list. Thus, we have considered all instances are correct. Therefore, the algorithm is correct. □

Proof of runtime. The algorithm uses the same steps as merge sort, so the recurrence relation is:

$$T(n) = 2T(n/2) + n$$

Need to show $T(n) = O(n \log n)$, i.e., $T(n) \leq cn \log n + n$. Suppose $c = 1$ and $T(1) = 1$.
Base: $T(1) = 1 \cdot 1 \cdot \log(1) + 1 = 1$
Hypothesis: Suppose $T(m) \leq cm \log m + m$ for all $m < n$.
Inductive step:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq 2[c(n/2)\log(n/2) + (n/2)] + n \\ &= cn(\log n - 1) + 2n \\ &= cn \log n - cn + 2n \\ &= cn \log n + n \end{aligned}$$

since $c = 1$. Thus $T(n) = O(n \log n)$. □

Algorithm 2 Count number of inversions

```
counter= 0 //counter keeps track of the number of inversions, which is returned after all recursive steps

function MERGESORT(list $A$)
    $n = |A|$
    if $n = 1$ then
        return $A$
    end if
    B = MERGESORT($A[0 : n/2]$)
    C = MERGESORT($A[n/2 + 1 : n - 1]$)
    return MERGE($B$, $C$)
end function

function MERGE(list $A$, list $B$)
    create empty array $C$
    $m = |A|$, $n = |B|$
    add $\infty$ as $m + 1$st element of $A$ and $n + 1$st element of $B$
    $i = j = k = 0$
    while $A[i] \neq \infty$ and $B[j] \neq \infty$ do
        if $A[i] < B[j]$ then
            $C[k] = A[i]$
            $i = i + 1$
            counter=counter+$j$
        else if $A[i] > B[j]$ then
            $C[k] = B[j]$
            $j = j + 1$
        end if
        $k = k + 1$
    end while

    while $A[i] \neq \infty$ do
        $C[k] = A[i]$
        $i = i + 1$
        counter=counter+$j$
        $k = k + 1$
    end while

    while $B[j] \neq \infty$ do
        $C[k] = B[j]$
        $j = j + 1$
        $k = k + 1$
    end while
    return $C$
end function

return counter //usually returned in the main method
```

Problem 5

Algorithm for finding the optimal location of the main pipeline in linear time. The optimal location is the median of the y coordinates of the wells. Essentially, we are proving that the median η minimizes the sum of the absolute loss $|y - \eta|$:

Proof that median minimizes the sum of the absolute loss. Denote the absolute loss as $|y - \eta|$. Derivative of the sum of absolute loss:

$$\begin{aligned}\frac{d}{d\eta}\sum_{i=0}^{n-1}|y_i - \eta| &= \sum_{i=0}^{n-1}\frac{d}{d\eta}|y_i - \eta| \\ &= 1 \cdot n_1 + 0 \cdot n_2 + (-1) \cdot n_3 \\ &= n_1 - n_3\end{aligned}$$

where n_1 is number of elements less than η, n_2 is number of elements equal to η, n_3 is number of elements greater than η, and because:

$$\frac{d}{d\eta}|y_i - \eta| = \begin{cases} -1 & \text{if } y_i > \eta \\ 0 & \text{if } y_i = \eta \\ 1 & \text{if } y_i < \eta \end{cases}$$

To minimize the sum of absolute loss, set derivative to zero:

$$\begin{aligned}\frac{d}{d\eta}\sum_{i=0}^{n}|y_i - \eta| &= 0 \\ n_1 - n_3 &= 0 \\ n_1 &= n_3\end{aligned}$$

which means the number of elements less than η is the same as number of elements greater than η. This is true if and only if η is the median. □

The algorithm described below to find the median of a list using median of medians is from lecture and from Fall 2014 lecture 3 notes:

In the case of even number of wells, the median could be either one of the middle two values in a sorted list of y coordinates. For consistency, choose the smaller of the middle two values.

Storing the y coordinates in an array takes $O(n)$ time. Selecting the median (essentially the $n/2$th element) of the array will only take $O(n)$ time if we choose the pivot using median of medians method. There are 2 components to finding the median using this method:

- Choosing the pivot using median and medians
- Selecting the median based on the pivot

Choosing the pivot requires first dividing the array into $n/5$ blocks each of size 5. Sort each of these blocks using any sorting method (which will be in constant time since they all have size 5) and select the median in each of the blocks. Then, store these medians in a new array and find the median of this new array.

Use this new pivot to find the $n/2$th, i.e. the median, of the original array of y coordinates. First, divide the array into 3 blocks: A_1 is a block where all values are less than the pivot, A_2 is a block where all values are equal to the pivot, and A_3 is a block where all values are greater than the pivot (this takes $O(n)$ time). After that, if $n/2 < |A_1|$, then recursively select pivot and find the median in block A_1. If $n/2 > |A_1| + |A_2|$, then recursively select pivot and find the median in block A_3. Otherwise, the median is the pivot.

Picking the pivot this way ensures $\max\{|A_1|, |A_3|\} \leq 7n/10$, therefore each recursive selection of median takes $T(7n/10)$ time. Since each array of medians has size $n/5$, finding the median of the medians will take $T(n/5)$ time. Finally, restructuring the array around the pivot takes linear time simply by stepping through the array, thus $O(n)$. Altogether, the recurrence relation is:

$$T(n) = T(7n/10) + T(n/5) + O(n)$$

Suppose $T(n) \leq cn$ for all $n < N$, and let $d \in \mathbb{R}$ be given.

$$\begin{aligned} T(n) &= T(7n/10) + T(n/5) + dn \\ &\leq c(7n/10) + c(n/5) + dn \\ &= c(9n/10) + dn = (9c/10 + d)n \\ &= cn \end{aligned}$$

if we let $c = 10d$. Thus, $T(n) = O(n)$.