Midterm Corrections

Robert George Phillips (rogphill@ucsc.edu)

February 11, 2018

Q1 (1 point): Consider the following sorting algorithm. If the input length is even, run insertion sort. Otherwise, run Mergesort. Suppose $T(n)$ denotes the worst-case running time of this algorithm. Which is true?

- (A) $T(n) = O(n^2)$ but not $\Omega(n^2)$.
- (B) $T(n) = \Theta(n^2)$.
- (C) $T(n) = O(nlog_2 n)$.

Answer:

$$f(x) = \begin{cases} O(n^2) & if even, using Insertion Sort. \\ O(nlogn) & if odd, using Merge Sort. \end{cases}$$

The worst case running time is when the algorithm is even, thus using Insertion Sort, which worst case runs in $O(n^2)$.

$F(n) \in O(n^2)$.

$G(n) \in O(n^2)$.

$\lim_{n\to\infty} \frac{F(n)}{G(n)} = \frac{n^2}{n^2} = 1$

Since 1 is a constant, $F(n) \in O(n^2)$.

However, since there can be a worst case run time where the input length is odd instead of even, thusly running Merge Sort, which $\in \Theta(nlogn)$, so therefore $F(n) \notin \Omega(n^2)$ and thus the answer is (A), $T(n) = O(n^2)$ but not $\Omega(n^2)$.

■

Q2 (1 point): Answer True or False. $\sum_{i=0}^{n} 2^i = \Theta(\sum_{i=0}^{n} 3^i)$

Answer: False by asymptotic analysis. Consider:

$F(n) = \sum_{i=0}^{n} 2^i = O(2^n)$.

$G(n) = \sum_{i=0}^{n} 3^i = O(3^n)$.

$\lim_{n\to\infty} \frac{F(n)}{G(n)} = \frac{2^n}{3^n} = (\frac{2}{3})^n = 0$.

Since the result of the limit is 0, $\lim_{n\to\infty} \frac{F(n)}{G(n)} < \infty$ and thus $\sum_{i=0}^{n} 2^i \in O(\sum_{i=0}^{n} 3^i)$ but $\notin \Omega(\sum_{i=0}^{n} 3^i)$ and therefore $\notin \Theta(\sum_{i=0}^{n} 3^i)$.

■

Q3 (1 point): Consider an array A of positive integers. You need to reorder the array such that all the odd numbers appear first, and then the even numbers. You can only use O(1) additional storage, so no hash tables allowed.

Give a short description of your algorithm, and the worst case running time. You should not need more than five lines to explain your answer.

Answer:

For a given array A, set i equal to 0 and j equal to $n-1$.

Increment i until $A[i] \mod 2 \neq 0$. Decrement j until $A[j] \mod 2 = 0$.

Then, while $i < j$, $swap(A[i], A[j])$.

The worst case running time is $O(n)$.

■

Q4 (2.33 points): We discussed the algorithm to merge two sorted arrays. Give an algorithm to merge k sorted arrays. For convenience, assume that every array has size exactly n. Your running time will depend on both k and n.

Answer:

One solution is where we can merge arrays A_0 and A_1 together, then merge arrays A_0 and $A_2 \dots A_0$ and A_n. Here's some pseudocode:

```
A0, A1, ... Ak are k sorted arrays.
for i = 1 to k-1 {
     merge(A0, Ai)
}
mergeSort(A0)
```

Using asymptotic analysis, $merge() \in O(n)$, and will $merge()$ for every value $0 \dots k$. Thus, this will run in $O(n*k = n^2)$ time. Since Merge Sort runs after the for loop, and is known to be $O(nlogn)$, it is a lower order than $O(n^2)$.

There is also a solution where we have n arrays, each of size k, the trivial brute-force solution would to make a new array of size $n * k$ and copy each element to the new array, running Merge Sort afterwards. Here's some pseudocode:

```
A = a 2d array with n columns (for n elements per array)
    and k rows (for k different arrays)
answer = array of size n*k
for (i = 0; i < k; i++) {
     for (j = 0; j < n; j++) {
          answer[(n*i)+j] = A[i][k]
     }
}
mergeSort(answer)
```

This algorithm's worst-case running time is $O(n * k = n^2)$, as the inner loop iterates n times for every k iterations of the outer loop.

As far as solving it recursively, we can use the divide and conquer method. The concept is similar to a Merge Sort, except we will be using an array of arrays instead. For our base case, we will return if there is only one array in the array. We will split the left side from arrays $A_0 \dots A_{k/2}$ and run it recursively through our function. Then, we split the right side from arrays $A_{k/2+1} \dots A_{k-1}$ and run it recursively through our function. Finally, we will run $merge()$ on arrays A_{left} and A_{right}. Here's some very basic pseudocode:

```
kArrayMerge(a list of k arrays) {
     if array size = 1, return
     left = {A0 ... A k/2}
     kArrayMerge{left}
     right = {Ak/2+1 ... Ak-1}
     kArrayMerge{right}
     merge(Aleft, Aright)
}
```

This solution has a recurrence of $T(n) \leq T(\frac{1}{2}) + O(n)$ and thus the algorithm $\in O(nlog_2 n)$.

■

Q5 (2.33 points): A sorting algorithm is called stable if the relative ordering of equal elements does not change. Thus, for $i < j$, if $A[i] = A[j]$, after a stable sort, the element initially at $A[i]$ ends up before the element initially at $A[j]$.

For example, given A = [4 3 10 8 4], the sorted version is B = [3 4 4 8 10]. In a stable sort, the element $A[0]$ (which is 4) will end up as $B[1]$, and the element $A[4]$ (which is also 4) will end up at $B[2]$. In a stable sort, the two 4s will not switch their order.

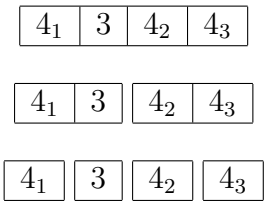
(This is relevant when sorting an array A of objects, by an integer field (say $A[i].key$). In a stable sort, the following is always true: if $i < j$ and $A[i].key = A[j].key$, then after sorting, the object initially at $A[i]$ will precede the object initially at $A[j]$.)

Show that Mergesort (with a very simple modification/clarification) is stable. Show that Quicksort is not stable, by providing an example.

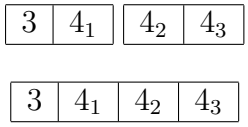
Answer:

The change we would need to make to Merge Sort would be at the end of the function – when running the $merge(A[left], A[right])$ step, we will additionally compare $A[left]$ to $A[right]$. If they are equal to each other, we merge in A[left] first.

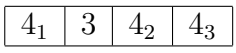
If we take a look at a recursive tree for this modified Merge Sort, we can see that it is stable:



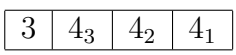
When the algorithm merges the two 4's on the right side of the tree, it checks to see if the values are equal to each other, and then if they are, it selects $A[left]$:



Quick Sort, however, is not stable. We can see this very clearly with an example:



In the common flavor of Quick Sort where we use the first element as a pivot, where the index is $i = 0$. Therefore, the pivot element is 4_1 and thus when Quick Sort makes a $\leq$ comparison to place the values either left or right of the pivot, the left side of the tree will have 3, 4_2, and 4_3 and the right side of the tree will have nothing. When the algorithm recursively sorts everything back to one array, we get a sorted array back, however it is not stable:



■

Q6 (2.33 points): Given an array A, give pseudocode and big-Oh running time analysis for an algorithm that determines which permutation of the stable sorted order is A. Meaning, output an array P, such that $P[i]$ is the index of the element $A[i]$ after a stable sort of A. For example, given $A = [4\ 3\ 10\ 8\ 4]$, the output should be $P = [1\ 0\ 4\ 3\ 2]$. So $A[0] = 4$ and it is at index 1 in the sorted order. Thus, $P[0] = 1$, etc. Note that because of the stable sorting, the last 4 goes to index 2 (and thus $P[4] = 2$). Furthermore, $A[1] = 3$, which ends up at index 0 in the sorted order. Thus, $P[1] = 0$. So on and so forth. (Yeah, my permutations start from 0, instead of 1, just to keep indexing easier.) You may assume access to a stable sorting algorithm, like the version of Mergesort you designed in Q5. If the whole discussion about stability is confusing you, solve this problem assuming all elements are distinct. (In this case, the stability is irrelevant.) You will get partial credit.

Answer: Assuming we use the version of Merge Sort from the previous question where it checks if the values are equal and then pulls A[left] first if so, we can additionally store a key value containing the original index for each element in the array. Once sorted using this special version of Merge Sort, we can iterate through the array and change the value of each element to the value of their key (original index). Here's some pseudocode:

```
MergeSortThenIndex(array A) {
     if array size = 1, return
     left = {A[0] ... A[k/2]}
     MergeSortThenIndex{left}
     right = {A[k/2+1] ... A[k-1]}
    MergeSortThenIndex{right}
    merge(A[left], A[right])
}
for i = 0 to k-1 {
    A[i] = A[i].key
}
```

Using asymptotic analysis, MergeSortThenIndex() runs similarly to Merge Sort except with an extra comparison, so it runs in $O(nlogn)$. The bottom

for loop that changes elements to their original index value runs in linear time and thus runs in $O(n)$. Thus, the overall runtime of this version of Merge Sort is $O(nlogn)$.

■