Quiz #1 Solutions
Introduction to Algorithms/Algorithms 1
600.363/463

March 4, 2014

1 Problem 1 (10 points: each subproblem is 2 points)

For each statement below, indicate whether it is true or false. You do not need to provide proofs or counterexamples.

1. Let f, g, h be three positive functions. If $f = \Theta(g)$ and $h = \Theta(g)$ then $f + h = \Theta(g)$.

   true false

2. Let f, g be two positive functions. If $f = \Omega(g)$ then $g = o(f)$.

   true **false**

3. $n^2 = O(n^{\log(n)})$

   true false

4. $n \log n = \omega\left((\log(n))^{10}\right)$

   true false

5. $n^{\log n} = \Theta\left(\log(n^n)\right)$

   true **false**

2 Problem 2 (20 points; each subproblem is 10 points)

Give asymptotic upper bounds for the following recurrences. You may use the Master theorem when it is applicable. You may assume that $n = a^k$ for some a, k. Assume that $T(0) = T(1) = 1$.

1. $T(n) = 25T(n/5) + n + 1$

   Applying the master theorem, we have $a = 25, b = 5$ and $f(n) = n + 1$. $\log_b a = 2$, and $f(n) = O(n^{2-\epsilon})$) for $0 < \epsilon < 1$.

   Thus, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

2. $T(n) = T(n-3) + 5$ (assume that $T(2) = 1$.)

   Substituting, we have

$$\begin{aligned} T(n) &= T(n-3) + 5 \\ &= T(n-6) + 10 \\ &= \dots \\ &= T(n-3k) + 5k. \end{aligned}$$

   This recurrence bottoms out when $0 \leq n - 3k \leq 2$, i.e., when $k = \lfloor n/3 \rfloor$. Thus, $T(n) = 5\lfloor n/3 \rfloor + 1$, which is $O(n)$.

3 Problem 3 (60 points)

Given k arrays $S_1, S_2, \ldots, S_k$, each a sorted array of n numbers, devise a divide-and-conquer algorithm which combines these k sorted arrays into a single sorted kn-length array in $O(kn \log k)$ time.

30 points will be given if (1) your algorithm works correctly, (2) your algorithm solves the problem in $O(k^2 n)$ time, (3) your analysis is correct and (4) your explanations and proofs are clear and with enough details.

Full credit will be given if (1) your algorithm works correctly, (2) your algorithm solves the problem in $O(kn \log k)$ time, (3) your analysis is correct and (4) your explanations and proofs are clear and with enough details. If you cannot prove your claims formally, give your best intuition.

If you present more than one solution, we will grade the best of your solutions. You may use and assume the correctness of any algorithms from class.

Solution

First, note that there were several valid solutions to this problem (including solutions that run faster than the $O(kn \log k)$ that we asked you for), so this is of course not the only way that one could have solved the problem.

Assume we have an algorithm $\mathrm{MERGE}(n, k)$, which merges k sorted lists of n elements each. Our divide and conquer algorithm will divide the k lists into two groups and call $\mathrm{MERGE}(n, k/2)$ on half of the lists and $\mathrm{MERGE}(n, k/2)$ on the other half, and then call $\mathrm{MERGE}(kn/2, 2)$ to merge the two sorted lists each of length $kn/2$ that result from the two $\mathrm{MERGE}(n, k/2)$ calls.

$\mathrm{MERGE}(m, 2)$ for any m is our base case– we can merge 2 lists of length m in $O(m)$ time using the merge step from mergesort, as seen in class. That is, $T(m, 2) = Cm$ for some constant $C > 0$. Thus, we have a recurrence of the form

$$T(n, k) = 2T(n, \frac{k}{2}) + T(\frac{kn}{2}, 2),$$

which we can rewrite as

$$T(n, k) = 2T(n, \frac{k}{2}) + C\frac{kn}{2}$$

by the fact that $T(m, 2) = Cm$. We can resolve this recurrence as follows:

$$\begin{aligned}
T(n,k) &= 2T(n, \frac{k}{2}) + C\frac{kn}{2} \\
&= 2(2T(n, \frac{k}{4}) + C\frac{kn}{4}) + C\frac{kn}{2} \\
&= 4T(n, \frac{k}{4}) + 2C\frac{kn}{2} \\
&= 4(2T(n, \frac{k}{8}) + C\frac{kn}{8}) + 2C\frac{kn}{2} \\
&= 8T(n, \frac{k}{16}) + 3C\frac{kn}{2} \\
&= \dots \\
&= 2^i T(n, \frac{k}{2^i}) + iC\frac{kn}{2}.
\end{aligned}$$

This recurrence bottoms out when $2^{i+1} = k$ (the base case where we are merging two lists), i.e., when $i = \log(k) - 1$. Thus we have

$$\begin{aligned}
T(n,k) &= 2^{\log(k)-1} T(n, \frac{k}{2^{\log(k)-1}}) + (\log(k) - 1)\frac{Ckn}{2} \\
&= \frac{Ckn}{2} + \frac{Ckn}{2}\log k - \frac{Ckn}{2} \\
&= \frac{Ckn}{2}\log k = O(kn \log k).
\end{aligned}$$

The correctness of our algorithm follows from the correctness of mergesort. Our divide step takes ℓ sorted lists of length m and recursively calls $\mathrm{MERGE}()$ on two disjoint sets of $\ell/2$ lists each. The conquer step of our algorithm merges two lists via mergesort. The correctness of mergesort shows that our conquer step works as required– we get back two sorted lists from our two calls to mergesort, and we merge them into a single sorted list. It remains only to show that the base case of the divide step is handled correctly. But our algorithm's base case, in which we call $\mathrm{MERGE}(m, 2)$, simply merges two sorted lists using mergesort, which we know to work correctly. Thus, the base case is handled correctly, and correctness of the algorithm follows.

4 Problem 4 (60 points)

Given two sorted arrays A and B, each containing n numbers (with all $2n$ numbers in the two arrays distinct), we call a pair of integers (i, j) with $1 \leq i \leq n$ and $1 \leq j \leq n$, an inversion if $A[i] > B[j]$. Devise an algorithm for counting the number of inversions for a pair of sorted arrays A and B in $O(n)$ time. Prove the correctness of your algorithm and give and prove its runtime.

25 points will be given if (1) your algorithm works correctly, (2) your algorithm solves the problem in $O(n \log n)$ time, (3) your analysis is correct and (4) your explanations and proofs are clear and with enough details.

Full credit will be given if (1) your algorithm works correctly, (2) your algorithm solves the problem in $O(n)$ time, (3) your analysis is correct and (4) your explanations and proofs are clear and with enough details. If you cannot prove your claims formally, give your best intuition.

If you present more than one solution, we will grade the best of your solutions.

Solution

We are given arrays A and B, each of length n, with all numbers distinct. The following algorithm will suffice:

```
i ← 1
j ← 1
c ← 0
while i ≤ n and j ≤ n do
  if A[i] > B[j] then
    while A[i] > B[j] do
      j ← j + 1
    end while
    c ← c + j − 1
    i ← i + 1
  else if A[i] < B[j] then
    c ← c + j − 1
    i ← i + 1.
  end if
end while
if i ≤ n then
  c ← c + (n − i + 1)
end if
return c
```

Runtime: The algorithm only ever increments i and j, and never resets either of them. Both are initialized to 1, and thus get incremented at most $2n$ times in total before the condition of the outer while loop is violated. Between increments to i and j, we do at most constant work, so the total runtime is $O(n)$.

Correctness: We maintain the invariant that at any time we have counted all the inversions of the form $A[k] > B[\ell]$ for $k < i$ and $\ell < j$. This invariant is trivially true when the algorithm starts ($c = 0, i = 1, j = 1$). Within each iteration of the outer while loop, the invariant is maintained by handling the two possible cases:

(1) If $A[i] > B[j]$, then we know that index i contributes *at least* j inversions (but possibly more) because both arrays are sorted (so $A[i] > B[j] > B[j-1] > \ldots B[1]$). We increment j until this condition is no longer true, at which point we have $B[j] > A[i] > B[j-1] > \ldots B[1]$, i.e., $A[i]$ contributes to $j-1$ inversions, which we add to the count. Thus, we have accounted for all inversions involving $A[i]$ and any element below $B[j]$. We increment i (because we've now accounted for $A[i]$) and the invariant is maintained.

(2) If $A[i] < B[j]$, then we know that $A[i]$ contributes to *at most* $j-1$ inversions. Our algorithm adds $j-1$ to the count in this condition and increments i, so we must show that indeed under this condition we have $B[j-1] < A[i] < B[j]$ (assume that $B[0] = -\infty$ but does not contribute to inversions, since it isn't actually in the array B), from which it will follow that we are indeed counting the correct number of inversions for $A[i]$, since B is sorted. Let k and ℓ be the current values of indices i and j, respectively and suppose by way of contradiction that $A[k] < B[\ell-1]$. Then we have $A[k-1] < A[k] < B[\ell-1]$. But if $A[k-1] < B[\ell-1]$, then in the previous iteration of the outer while loop, we would have had $i = k-1$ and $j = \ell - 1$, and by construction of the algorithm we would have incremented index i but not j because (at the time) $A[i] < B[j]$. Thus, index j would not have the value it currently has, contradicting the assumption that the index into B does indeed currently point to $B[j]$. Thus, by contradiction, $B[j-1] < A[i] < B[j]$.

Termination of the algorithm is handled by ensuring that we don't accidentally fail to count remaining elements in A when we reach the end of array B before reaching the end of A. If we endthe while loop with $i < n$, then elements $A[i]$ through $A[n]$ each contribute n inversions, because each one must have been larger than every element of B.