Purdue University — S. Hambrusch - C. Guerra

CS 381 — Fall 1999

Solution Sketches to Assignment 3

1) (Graded by Tian Luan) For each of the following recurrence equations determine whether the Master Theorem can be applied. If it can, give the asymptotic bound and show which one of the three cases holds and why. If it cannot, show why all cases fail. Assume that $T(1) = 1$.

(i) $T(n) = 4T(n/4) + 6$
In this equation, $b = 4, c = 4, E = \frac{\log 4}{\log 4} = 1$, $f(n) = 6 = O(n^0)$. Hence, $f(n)$ is $O(n^{1-\epsilon})$ where $\epsilon = 1$. Case (1) of the master theorem applies and $T(n) = \Theta(n^E) = \Theta(n)$.

(ii) $T(n) = 4T(n/3) + n^2$
In this equation, $b = 4$, $c = 3, E = \frac{\log 4}{\log 3}$. We have $f(n) = n^2 = \Omega(n^{E+\epsilon})$ where $\epsilon \leq 2 - \log_3 4$ and $f(n) = n^2 = O(n^{E+\delta})$ where $\delta \geq 2 - \log_3 4$.
Case (3) of the master theorem applies and $T(n) = \Theta(f(n)) = \Theta(n^2)$.

(iii) $T(n) = 2T(n/2) + \frac{n}{\log n}$
In this equation, $b = 2, c = 2$, $E = 1$, and $f(n) = \frac{n}{\log n}$. None of the three cases applies. For case (1) we cannot find an ϵ that satisfies $f(n) \in O(n^{1-\epsilon})$ because there exists no ϵ such that $\log n \geq n^\epsilon$. For case (2), $f(n)$ is not $\Theta(n)$. For case (3), we can not find ϵ and δ that satisfy $f(n) \in \Omega(n^{1+\epsilon})$ and $f(n) \in O(n^{1+\delta})$ - the reason is the same as for Case (1).
Hence, all three cases fail and the master theorem does not give a solution.

(iv) $T(n) = 6T(n/6) + 3n - 5$
In this equation, $b = 6$, $c = 6$, $E = \frac{\log 6}{\log 6} = 1$, and $f(n) = 3n - 5 = \Theta(n)$. Therefore, $f(n) \in \Theta(n^E)$.
Case (2) applies and $T(n) = \Theta(n \log n)$.

(v) $T(n) = T(\sqrt{n}) + \log n$.
This equation is not of the form $T(n) = bT(\frac{n}{c}) + f(n)$, where b abd c are constants. The master theorem cannot be used to generate a solution to the recurrence.

2) (Graded by Tian Luan) Consider the recurrence relation $T(n) = 2T(n/2) + 5n \log n$ with $T(2) = 1$ and $n = 2^k$, $k \geq 0$. Show by induction that $T(n) = O(n \log^2 n)$.

Claim: $T(n) \leq cn\log^2 n$ where $c \geq 6$.
Proof (by induction):
Base case: for $n = 2$,

$$1 = T(2) \leq 2c$$

The base case will hold as long as $c \geq 1/2$.
Induction Hypothesis: Assume for all $m < n$ and m a power of 2, we have $T(m) \leq cm\log^2 m$.
Consider $n = 2^k$:

$$T(n) = 2T(\frac{n}{2}) + 5n\log n$$

(by definition)

$$\leq 2(c\frac{n}{2}(\log^2 \frac{n}{2})) + 5n\log n$$

(by induction hypothesis)

$$= cn(\log n - 1)^2 + 5n\log n$$

$$= cn\log^2 n - 2cn\log n + cn + 5n\log n$$

Observe that $\log^2 n = (\log n)^2$. If we want to prove $T(n) \leq cn\log^2 n$, we must show

$$-2cn\log n + cn + 5n\log n \leq 0$$

Because $c \geq 6$,

$$(2c - 5)n\log n \geq cn$$

for all $n \geq 2$. So we have $-2cn\log n + cn + 5n\log n \leq 0$ and $T(n) \leq cn\log^2 n$. The claim holds thus for n.

3) (Graded by Biana Babinsky)
i) Assume you are given an Array A of size n containing real numbers. Describe and analyze an efficient algorithm to determine n/4 numbers not in the array A.

Find the maximum element in the array, call it max. This will take O(n) time. Since max is the biggest number in the array, max+1 is not in the array, and neither are max+2 or max+3, etc. Thus $n/4$ numbers that are not in the array are max+1, max+2, ..., max+n/4. Finding max takes O(n) time and finding n/4 numbers greater then it will also take O(n) time. Thus overall the algorithm will take O(n) time. Note: it's also ok to find the minimum of the array and then find n/4 numbers that are smaller then the minimum.

ii) You are given an array A of size n and array B of size m. The elements of each array are in arbitrary order. Describe and analyze an efficient algorithm to determine whether the elements in the two arrays are disjoint. State your running time in terms of m and n. Make sure you consider the case when m is substantially smaller then n.

Sort array B using your favorite $O(m \log m)$ time sorting algorithm. Now search for each element of A in sorted array B (using binary search). If you can find an element that is both in A and B, the arrays are not disjoint. Otherwise, they are.

Time Analysis: sorting of B takes $O(m\ logm)$ time. Searching for one element of A in B takes O(logm). Thus, to search for n elements in B will take O(nlogm). Altogether, this will take $O(m \log m + n \log m) = O((m+n) \log m)$.
Note: sorting array A and then searching in it for elements of B will take O((m+n)logn) time, which is not as efficient when m is significantly smaller than n.