CSC373 Fall'19
Tutorial 1

Mon. Sept. 16, 2019

Recall: Master Theorem For constants $a \geq 1$ and $b > 1$, and an asymptotically positive function $f(n)$, the recurrence relation $T(n) = a \cdot T(n/b) + f(n)$ has the following solution.

1. If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$ for some constant $k \geq 0$, then $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$.
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$ and f satisfies the regularity condition*, then $T(n) = \Theta(f(n))$.
   (*Regularity condition: For some constant $c < 1$ and all sufficiently large n, $af(n/b) \leq cf(n)$.)

Note: There are recurrence relations which do not fall under any of these three cases (e.g. the recurrence relation $T(n) \leq T(n/5) + T(7n/10) + O(n)$ from QuickSelect where the smaller instances are not of uniform size, or the recurrence relation $T(n) = \sqrt{n}T(\sqrt{n}) + n$ where a and b are not constants). If you're interested in how more general recurrences can be solved, there are some excellent resources available online.[1,2]

Q1 Practicing Recurrence Relations

Find the best possible asymptotic upper bound for $T(n)$ under the following recurrence relations.[3]

(a) $T(n) \leq 3T(n/2) + O(n \log^3 n)$

(b) $T(n) \leq 4T(n/2) + O(n^2)$

(c) $T(n) \leq 2T(n/2) + O(n \log^2 n)$

(d) $T(n) \leq 2T(n/4) + O(n^{0.5001})$

Q2 Circularly Shifted Sorted Array

Suppose you are given an array $A[1 \dots n]$ of sorted distinct integers that has been circularly shifted k positions to the right. For example, $[35, 42, 5, 15, 27, 29]$ is a sorted array that has been circularly shifted $k = 2$ positions to the right, while $[27, 29, 35, 42, 5, 15]$ has been shifted $k = 4$ positions.

We can obviously find the largest element in A in $O(n)$ time. Describe an algorithm which runs in time $O(\log n)$. Prove correctness of your algorithm and the fact that it runs in $O(\log n)$ time.

[1] `http://jeffe.cs.illinois.edu/teaching/algorithms/notes/99-recurrences.pdf`

[2] `http://web.csulb.edu/~tebert/teaching/lectures/528/recurrence/recurrence.pdf`

[3] Note that when proving an upper bound on the worst-case running time of an algorithm, you would encounter equations of the form $T(n) \leq \dots$ rather than $T(n) = \dots$. For a lower bound, you would often need to explicitly construct an instance on which the algorithm takes the desired amount of time.

Q3 Majority Element

Given an array A of size n that is known to contain a majority element, your goal is to find this majority element. The majority element is an element that appears strictly more than $\lfloor n/2 \rfloor$ times.

Your algorithm is only allowed to use equality comparisons (i.e. "Is $A[i] = A[j]$?"), and the running time of the algorithm is measured by the number of equality comparisons it needs to make in the worst case. Try to design an algorithm with as little worst-case running time as you can.

Q4 Monotonic Function Evaluation

Consider a monotonously decreasing function $f : \mathbb{N} \to \mathbb{Z}$ (that is, a function defined on natural numbers which takes taking integer values and satisfies $f(i) > f(i+1)$ for all $i \in \mathbb{N}$). Assuming we can evaluate f at any point i in constant time, we want to find $n = \min\{i \in N | f(i) <= 0\}$ (that is, we want to find the first point where f becomes non-positive). Note that n is not given to us, but we are allowed to express the running time of our algorithm in terms of n.

We can obviously solve the problem in $O(n)$ time by simply evaluating $f(1), f(2), f(3), \ldots, f(n)$. Describe an $O(\log n)$ time algorithm.

[Hint: Try to quickly get an estimate of n, and then precisely pinpoint the exact value of n in the range you estimated.]