Algorithms and Data — Problem Set 4

Nick Ippoliti

November 8, 2016

1 Master Method

1.1

$T(n) = 2T(n/4) + 1$
$f(n) = 1, a = 2, b = 4$
$c < \log_b a$
$c < \log_4 2$
$c < 0.5$
$f(n) \in O(n^{0.5}) \implies T(n) \in \Theta(n^{0.5}) = \Theta(\sqrt{n})$

1.2

$T(n) = 5T(n/4) + n$
$f(n) = n, a = 5, b = 4$
$c < \log_4 5$
$f(n) \in O(n^{\log_4 5}) \implies T(n) \in \Theta(n^{\log_4 5})$

1.3

$T(n) = 2T(n/4) + \sqrt{n}$
$f(n) = \sqrt{n}, a = 2, b = 4$
$c = 0.5$
$f(n) \in \Theta(\sqrt{n}) \implies T(n) \in \Theta(\sqrt{n} \log n)$

1.4

$T(n) = 7T(n/2) + n^3$
$f(n) = n^3, a = 7, b = 2$
$c < \log_2 7 \approx 2.807$
$f(n) \notin O(n^{2.807})$
$f(n) \notin \Theta(n^{2.807})$
$f(n) \in \Omega(n^{2.807}), 7(n^3/2) \leq 0.5n^3 \implies T(n) \in \Theta(n^3)$

2 Binary Search Recurrence

2.1

$T(n) = T(n/2) + \Theta(1)$

2.2

$f(n) \in \Theta(n^c \log^k n)$ where $c = \log_b a$
$c = \log_2 1 = 0$
$\Theta(1) \in \Theta(\log n)$
$\implies T(n) \in \Theta(\log n)$

3 A "Hole" in the Master Theorem

3.1

Yes, the Master Theorem can work here since it satisfies case 2.

3.2

$n = 2^k$
$2n = 2^{k+1}$
$T(2n) = 4T(2n/2) + (2n)^2 \log(2n)$
$T(2n) = 4T(n) + (2n)^2 \log(2n)$
$T(2n) = 4(n^2(\log n)^2) + (2n)^2 \log(2n)$
$T(2n) = 4n^2(\log n)^2 + (2n)^2 \log(2n)$
$T(2n) = (2n)^2(\log n)^2 + (2n)^2 \log(2n)$
$T(2n) = (2n)^2(\log 2n + \log 2n)$
$T(2n) = (2n)^2(2(\log n)^2)$
$T(2n) = 2(n^2 \log n)^2$
Thus, $T(n) \leq (n^2 \log n)^2$

3.3

$n = 2^k$
$2n = 2^{k+1}$
$T(2n) = 4T(2n/2) + (2n)^2 \log(2n)$
$T(2n) = 4T(n) + (2n)^2 \log(2n)$
$T(2n) = 4(\frac{1}{3}n^2(\log n)^2) + (2n)^2 \log(2n)$
$T(2n) = (\frac{1}{3})4n^2(\log n)^2) + 4n^2 \log(2n)$
$T(2n) = (\frac{1}{3})4n^2(\log(n)^2 + \log(n)^2)$
$T(2n) = \frac{4}{3}n^2(\log(n)^2 + \log(n)^2)$
$T(2n) = \frac{8}{3}n^2 \log(n)^2$
Thus, $T(n) \geq \frac{1}{3}n^2(\log n)^2$

4 A Maximum

4.1

$O(n^2)$, since you have to check every element and sum it with every element that comes after it in the list, keeping track of the maximum sum.

4.2

A more efficient algorithm is as follows. Split the list in half until it cannot be split any more. Then, at every level, find the maximum of the maximum value on the left, the maximum value on the right, or a sum spanning the middle. To find the sum that spans the middle, on both sides start closest to the middle and keep a running sum for every element up until the end, and then use the maximum value. Once both sides are calculated like that, add them together.

By doing this, you can find the resulting maximum sum in $O(n \log n)$ time. This is because finding the value spanning the middle can be done in $O(n)$ with calculating the maximum of the three values being $O(k)$. This is then done $O(\log n)$ times, for a total of $O(n \log n)$.

5 Duplicates

To remove all duplicates from an array in $O(n \log n)$ time, first we must sort the array ($O(n \log n)$). Then, for every element in the list (except the first), check if the previous element is the same as it, and if it is, remove it from the list. This is $O(n)$. This means that the total running time is $O(n \log n + n) = O(n \log n)$.

6 Squaring Numbers

To square a number n is the same as multiplying n by n. Because of this, the big-O runtime is the same. However, if you represent n as $(a + b)$, we can see that $(a + b)(a + b) = a^2 + ab + ab + b^2$, or $a^2 + 2ab + b^2$. This requires less subproblems to multiply than $(a+b)(c+d) = ac+ad+bc+bd$, so in that case, it could be said that squaring a number is faster.

7 Fixed Point

This could be solved by using a modified binary search that kept track of the current index of the element being checked. For example, if you picked element

a in E at index i where $i = |E|/2$, then on the next call you would check the elements at $i - i/2$ and $i + i/2$. This would continue until you found (or didn't find) a number n where $n = i$.