CSC373 Summer '22
Tutorial 1 Solutions

Thursday, May 19, 2022

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

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

Note: There are recurrence relations which do not fall under any of these three cases (e.g. the recurrence relation $T(n) \leqslant T(n/5) + T(7n/10) + O(n)$ from QuickSelect where the smaller instances are not of uniform size, or the recurrence relation $T(n) \leqslant \sqrt{n} \cdot T(\sqrt{n}) + O(n)$ 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) $T(n) \leqslant 3 \cdot T(n/2) + O(n \log^3 n)$

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

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

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

Solution to Q1

(a) For $T(n) \leqslant 3T(n/2) + O(n \log^3 n)$, we have:

- $a = 3$ and $b = 2$; thus, $n^{\log_b a} = n^{\log_2 3}$.
- $f(n) = n \log^3 n$.

Hence, by case 1 of the Master theorem, $T(n) = O(n^{\log_2 3})$.

[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 $T(n) \leqslant \ldots$ rather than $T(n) = \ldots$, yielding $T(n) = O(\cdot)$ rather than $T(n) = \Theta(\cdot)$. To derive a lower bound, you need to explicitly construct instances on which the algorithm takes at least the claimed amount of time.

(b) For $T(n) \leqslant 4T(n/2) + O(n^2)$, we have:

- $a = 4$ and $b = 2$; thus, $n^{\log_b a} = n^{\log_2 4} = n^2$.
- $f(n) = n^2$.

Hence, by case 2 of the Master theorem, $T(n) = O(n^2 \log n)$.

(c) For $T(n) \leqslant 2T(n/2) + O(n \log^2 n)$, we have

- $a = 2$ and $b = 2$; thus, $n^{\log_b a} = n^{\log_2 2} = n$.
- $f(n) = n \log^2 n$.

Hence, again by case 2 of the Master theorem, $T(n) = O(n \log^3 n)$.

(d) For $T(n) \leqslant 2T(n/4) + O(n^{0.5001})$, we have

- $a = 2$ and $b = 4$; thus, $n^{\log_b a} = n^{\log_4 2} = n^{0.5}$.
- $f(n) = n^{0.5001}$.

Hence, by case 3 of the Master theorem, $T(n) = O(n^{0.5001})$.

Q2 Monotonic Function Evaluation

Consider a monotonously decreasing function $f : \mathbb{N} \to \mathbb{Z}$ (that is, a function defined on natural numbers which takes 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 $n = \min\{i \in N | f(i) \leqslant 0\}$ (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 told that some point i with $f(i) \leqslant 0$ exists (i.e. n is well-defined), and 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.]

Solution to Q2

Let $k = 1$, and while $f(k) > 0$, double k. In at most $\lceil \log n \rceil$ iterations, this will terminate as we will have $k \geqslant n$. Let k^* be the value at which it terminates. Then, we know that $k^*/2 < n \leqslant k^*$. We can binary-search n in this range (or for simplicity, in the range $1 \ldots k^*$), as described by the function FindFirstNonPositive below.

The running time for finding k^* is $O(\log n)$, and the running time for the subsequent binary search is also $O(\log n)$. Hence, the overall running time is $O(\log n)$.

```
Function FindFirstNonPositive($A[1 \ldots r]$):
    if $r = 1$ then
        return $A[1]$
    $m \leftarrow \lfloor r/2 \rfloor$
    if $A[m] \leqslant 0$ then
        return FindFirstNonPositive($A[1 \ldots m]$)
    else
        return FindFirstNonPositive($A[(m+1) \ldots r]$)
```

Q3 Maximum Subarray Sum

You are given an array $A[1 \ldots n]$, and you are asked to find the *maximum subarray sum*, that is, the maximum value of $\sum_{t=i}^{j} A[t]$ over all possible (i, j) with $1 \leqslant i \leqslant j \leqslant n$. Design an $O(n)$ time divide and conquer algorithm for the problem.

[Hint: Once you divide the array into two equal halves, say $A[1 \ldots \text{mid}]$ and $A[\text{mid}+1 \ldots n]$, you will get the maximum subarray sum within each half. What extra information do you need from each half?

If you spend $O(n)$ time in the merge step to calculate this extra information, you will get $O(n \log n)$ running time. Can you get your recursive algorithm to return this information instead?]

Solution to Q3

Suppose we have already calculated the maximum subarray sums in the two halves, $A[1 \ldots \text{mid}]$ and $A[\text{mid}+1 \ldots n]$. To find the overall maximum subarray sum, we need a third value: the maximum subarray sum where the subarray overlaps both halves.

Note that this quantity is the sum of two quantities: the maximum suffix sum in the left half (i.e. maximum sum of any $A[i \ldots \text{mid}]$) and the maximum prefix sum in the right half (i.e. maximum sum of any $A[\text{mid}+1 \ldots j]$).

After recursively calling our algorithm on the two halves, we could spend $O(n)$ time calculating the maximum suffix sum in the left half and the maximum prefix sum in the right half. But as the hint suggests, this will give us $T(n) = 2 \cdot T(n/2) + O(n)$, i.e., $T(n) = O(n \log n)$.

Instead, we want our recursive algorithm to return the maximum suffix sum in addition to the maximum subarray sum, when called on the left half, and return the maximum prefix sum in addition to the maximum subarray sum, when called on the right half.

That means, our algorithm must return three quantities: maximum subarray sum, maximum prefix sum, and maximum suffix sum. Note that the "parent" call must also return these quantities once getting them from the two recursive calls, otherwise it is not a legitimate recursive algorithm.

When thinking along these lines, it becomes clear that to compute the maximum prefix and suffix sums in the entire array, we will also need to know the sum of the left and right halves. Also, in prefix and suffix sums, we need to allow empty prefix and suffix, which plays a key role in the first step of the algorithm below.

```
Function Subarray-Prefix-Suffix-Sum($A[\ell \dots r]$):
    if $r = \ell$ then
        return $(A[\ell], \max(A[\ell], 0), \max(A[\ell], 0), A[\ell])$
    mid $\leftarrow \lfloor (\ell + r)/2 \rfloor$
    [subarrayLeft,prefixLeft,suffixLeft,sumLeft] $\leftarrow$ Subarray-Prefix-Suffix-Sum($A[\ell \dots \mathrm{mid}]$)
    [subarrayRight,prefixRight,suffixRight,sumRight] $\leftarrow$
      Subarray-Prefix-Suffix-Sum($A[\mathrm{mid}+1 \dots r]$)
    subarray $\leftarrow \max$(subarrayLeft,subarrayRight,suffixLeft+prefixRight)
    prefix $\leftarrow \max$(prefixLeft,sumLeft+prefixRight)
    suffix $\leftarrow \max$(suffixRight,suffixLeft+sumRight)
    sum $\leftarrow$ sumLeft+sumRight
    return [subarray,prefix,suffix,sum]
```

Note that the worst-case running time of this algorithm is given by $T(n) \leqslant 2 \cdot T(n/2) + O(1)$, which yields $T(n) = O(n)$.