CSCI 270 Homework 4

Qilin Ye

September 25, 2022

1. Master Theorem

Theorem: Master Theorem

Suppose the runtime of a recursive algorithm satisfies

$$T(n) = aT(n/b) + f(n).$$

- (MT1) If $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
- (MT2) If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
- (MT3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ with $\limsup_{n\to\infty} \frac{af(n/b)}{f(n)} < \infty$, then $T(n) = \Theta(f(n))$.

Solution. Throughout this exercise we keep our notations of a, b, and f consistent with the lecture's style.

(1.a) Since $c_1 n^2 = \Theta(n^2) = \Theta(n^{\log_3(9)})$, we invoke (MT2) and obtain $T(n) = \Theta(n^2 \log n)$.

(1.b) Since $5^{1.5} = \sqrt{125} > 10$, implying $f(n) = c_2 n^{1.5} = \Omega(n^{\log_5(10)+\epsilon})$ for small $\epsilon > 0$, and

$$\limsup_{n\to\infty} \frac{10f(n/5)}{f(n)} = \limsup_{n\to\infty} \frac{10n^{1.5}}{5^{1.5}n^{1.5}} = \frac{10}{5^{1.5}} < \infty,$$

we invoke (MT3) and obtain $T(n) = \Theta(f(n)) = \Theta(n^{1.5})$.

(1.c) Since $\log_4 3 > \log_4 2 = 1/2$, $f(n) = c_3 n^{1/2} = \mathcal{O}(n^{\log_4(3)-\epsilon})$ for small $\epsilon > 0$. We therefore invoke (MT1) and obtain $T(n) = \Theta(n^{\log_4(3)})$.

(1.d) The root (0^{th} level) involves $c_4 n = \Theta(n)$ amount of work. The next (1^{st}) level involves $c_4(n/3) + c_4(2n/3) = \Theta(n)$ work as well. Using this fact recursively, we see that if levels $k, k+1$ are both full, then $\Theta(n)$ work on level k implies $\Theta(n)$ work on level $k+1$ as each node in level k corresponds to precisely two nodes in the next level, and the amount of works add up.

Now, clearly $\log_3(n) < \log_{3/2}(n)$, so a valid lower bound for the number of full levels is $\log_3(n)$. We therefore have $T(n) \geqslant \log_3(n)\Theta(n)$. On the other hand, the tree has height $\log_{3/2}(n)$, forcing an upper bound $T(n) \leqslant \log_{3/2}\Theta(n)$. Combining these two inequalities give $T(n) = \Theta(n \log n)$.

(1.e) The root has work $c_5 n$. The next (1^{st}) level has work $c_5(n/2) + c_5(n/4) = c_5(3/4)n$. Applying this fact iteratively, the k^{th} level has work at most $c_5(3/4)^k n$. The root also guarantees a trivial lower bound of $c_5 n$

total work. Since the tree has a finite height ($\log_4(n)$), we obtain

$$c_5 n \leqslant T(n) \leqslant c_5 n \sum_{i=j}^{\log_4(n)} (3/4)^j < c_5 n \sum_{j=1}^{\infty} (3/4)^j < Cn \qquad \text{for some } C > 0.$$

Therefore $T(n) = \Theta(n)$.

2. Increasing Index Value

Proof. (a) Suppose not, i.e., $a_i \geqslant a_{i+1}$ for $1 \leqslant i \leqslant n-1$. Then $a_1 \geqslant a_2 \geqslant \ldots \geqslant a_n$, contradicting $a_1 < a_n$.

(b) The algorithm goes as follows.

```
1  function find_index(A,i,j)
2    if i == j: return null # bad array, i.e., n = 1
3    else if A[i] < A[i+1]: return pair (A[i], A[i+1])
4    else:
5      if A[i] < A[(i+j)//2]: return find_index(A,i,(i+j)//2) # see if there exists such (i,i+1) in the
           first half of this array
6      else: return find_index(A,(i+j)//2, j) # if not, such (i,i+1) must be the second half; start with
           (i+j) // 2 not (i+j) // 2 + 1 because these two indices may actually be the only adjacent pair
           satisfying such property
```

Since in each iteration the difference between the parameters i, j decay by half, the recursion has at most $\log_2(n)$ calls and terminate either because $i = j$ at line 2 or return a pair of indices before that. In each iteration, all other operations are primitive. Therefore the total runtime is $\mathcal{O}(\log_2(n) + 1) = \mathcal{O}(\log n)$ and its correctness is guaranteed by (a). □

3. Median of Two Sorted Arrays

Proof. The algorithm goes as follows. As a good coding practice I tried to make it cover all edge cases of a, b; when doing runtime analysis, however, I will assume n is a power of 2. Also, we assume all arrays used in the following algorithm start with index 1 (why??).

```
function find_median(a,b)
  total_length = len(a) + len(b)
  return find_kth(a,b,total_length/2) # c'mon why are we starting at index 1??

function find_kth(a,b,k)
  if not a: return ['b', k]
  if not b: return ['a', k]

  mid_a, mid_b = len(a) // 2 + 1, len(b) // 2 + 1 # again, issues with counting from 1: 1 // 2 = 0, no good
  median_a, median_b = a[mid_a], b[mid_b]

  # if k > mid_a + mid_b, skip either first half of a or b
  if mid_a + mid_b < k:
    if median_a > median_b: # b[:mid_b] cannot contain candidate
      return find_kth(a, b[mid_b+1:], k - mid_b)
    else:
      return find_kth(a[mid_a+1:], b, k - mid_a)
  # if not, then the true median is in a[:mid_a] or b[:mid_b]
  else:
    if mid_a > mid_b: # a[mid_a+1:] cannot contain candidate
```

```
21          return find_kth(a[:mid_a], b, k)
22      else:
23          return find_kth(a, b[:mid_a], k)
```

The comments above have pretty much explained the correctness of this algorithm already. The if-else cases guarantee that whatever we rule out cannot possibly eradicate all valid candidates (there might be more than 1 medians and the algorithm excludes a few of them, but the algorithm will never exclude them all). On the other hand, the algorithm does not terminate until one array is empty, whereas one array *will* become empty in finite iterations since the total length of a and b strictly decreases. Therefore, the value eventually returned is what we desire.

A rough upper bound of the runtime uses the fact that it takes (at most) n iterations to bring the size of a to 1 and another n for b. (This is the worst case where neither a nor b becomes empty early.) One more iteration, and either a will become empty in line 20 or b will in 22, and we will return the other remaining value. In this case, the algorithm loops a total of $2\log_2(n) + 1$ times so the worst case complexity is $\mathcal{O}(\log n)$. □