Kaleb Kim
9/21/24

CSDS 310 Assignment 2 (rev)

Note: Arrays are zero-indexed.

Problem 3

a) With $T(n)$ in Master Theorem form, $T(n) = aT(n/b) + f(n)$, given $a = b$ and $b = a$.

We have $a > b$ by reversing our given statement, meaning that $\log_b a > 1$. Raising both sides to the n, we have $n^{\log_b a} > n^1$. This means that $n^{\log_b a}$ is an upper bound of $f(n)$, which means $f(n) = O(n^{\log_b a})$, which is Case 1. Thus, $T(n) = n^{\log_b a}$.

b) With $T(n)$ in Master Theorem form, $T(n) = aT(n/b) + f(n)$, given $a = a^2$ and $b = a$. We have

$$\log_b a = \frac{\log a}{\log b} = \frac{\log a}{\log(a^2)} = \frac{\log a}{2\log a} = \frac{1}{2}$$

Given that $f(n) = \Theta(n^2)$, comparing it to $n^{\log_b a} = n^{\frac{1}{2}}$, $f(n)$ grows much faster than $n^{\log_b a}$. We have $f(n) = \Omega(n^{\log_b a})$, which is Case 3.

Thus, $T(n) = O(n^2)$.

c) With $T(n)$ in Master Theorem form, $T(n) = aT(n/b) + f(n)$, given $a = 1$ and $b = \lambda$. We have $\log_b a = \log_\lambda(1) = 0$. This means that $n^{\log_b a} = n^0 = 1$, which is constant. We have $f(n) = n^\lambda$, which grows more than a constant function. In other words, $f(n) = \Omega(n^{\log_b a})$, which is Case 3. Thus, $T(n) = O(n^\lambda)$.

Problem 4

Pseudocode:

procedure MERGE(A, n, k):
- **if** k == 1:
  - **return** A
- A' ⟵ Array of $\lceil \frac{k}{2} \rceil$ elements
- **for** $0 \le i < \lfloor \frac{k}{2} \rfloor$:
  - A'[i] ⟵ SUBMERGE(A[2i], A[2i+1])
- **if** k is odd:
  - A'[$\frac{k}{2}$] ⟵ A[$k - 1$] // Put last element in A'
- **return** MERGE(A', n, $\lceil \frac{k}{2} \rceil$)

The merging of two subarrays, cleverly named:

procedure SUBMERGE(A, B):
- i ⟵ 0
- j ⟵ 0
- C ⟵ Array of $n_A + n_B$ elements

```
5   while i + j < len(A) + len(B):
6       if i == len(A):
7           C[i+j] ⟵ B[j]
8           j ⟵ j + 1
9       else if j == len(B) or A[i] < B[j]:
10          C[i+j] ⟵ A[i]
11          i ⟵ i + 1
12      else:
13          C[i+j] ⟵ B[j]
14          j ⟵ j + 1
15  endwhile
16  return C
```

Proof:

`SUBMERGE` procedure proof by loop invariance:

- **Loop invariant:** At the start of each iteration, C is an array sorted in nondecreasing order, such that $C[0...i + j - 1]$ contains the same elements from subarrays $A[0...i - 1]$ and $B[0...j - 1]$.
- **Initialization:**: Before the first iteration, $i = 0$ and $j = 0$. We have $i + j = 0$. By the loop invariant, $C[0... - 1]$ is a null (empty) array, and so as $A[0... - 1]$ and $B[0... - 1]$. Trivially, C holds the same elements of subarrays A and B, thus the loop invariant holds at this step.
- **Maintenance**: (1) If at the start of an iteration $i \geq$ the elements in A, then increment j by one and append $B[j]$ to C. Likewise, (2) if $j \geq$ the elements in B, then increment i by one and append $A[i]$ to C. These cases occur when we have iterated through all the elements in either subarray. (3) If $A[i] < B[j]$, then the smaller element, append $A[i]$ to C and increment i by one. (4) Else, when $A[i] \geq B[j]$, append $B[j]$ to C and increment j by one.

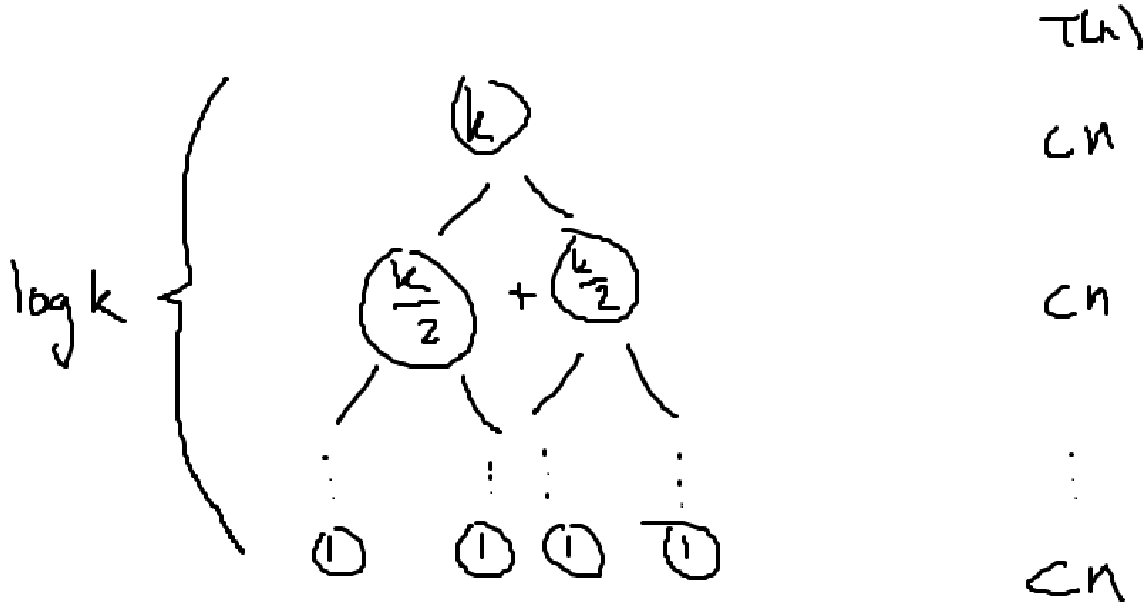
  All cases (1-4) result in incrementing either i or j by one, with each case appending the smaller element to $C[0...(i + j + 1) - 1]$ from either $A[0...(i + 1) - 1]$ or $B[0...(j + 1) - 1]$, maintaining the loop invariant.
- **Termination**: The loop terminates when $i + j$ meet the total number of elements from A and B. Since the loop invariant held over all elements of A and B, C is an array sorted in nondecreasing order, containing each element from A and B once.

`MERGE` procedure proof by induction:

- **Base case**: At $k = 1$, trivially array A is fully merged.
- **Inductive step**: By induction, it can be assumed that for $k > 1$, we can continue to merge subarrays. We create an output array A' that stores $\left\lceil \frac{k}{2} \right\rceil$ elements. Trivially, the for loop merges two subarrays $A[2i]$ and $A[2i + 1]$ into one (proven that `SUBMERGE` is correct), storing it in $A'[i]$. The loop terminates at the end of $\left\lfloor \frac{k}{2} \right\rfloor$, adding the leftover odd element when necessary.

Runtime analysis:

The `SUBMERGE` procedure contains a while loop that makes $n_A + n_B$ comparisons, with the number of elements of A and B being n_A and n_B respectively. In one iteration of the `MERGE` procedure, the `SUBMERGE` procedure is called $\frac{k}{2}$ times in the for loop of line (5), iterating over two subarrays each iteration. Hence, the worst-case runtime for one iteration of `MERGE` is $O\left(\frac{k}{2} * 2 * \frac{n}{k}\right) = O(n)$.



Drawing a recursion tree, we will `MERGE` about $O(\log k)$ times. Thus, $T(n) = O(n) * O(\log k) = O(n \log k)$.