College of Computer & Information Science
Northeastern University
Fall 2018

CS5800
MS Algorithms
12 October 2018

Sample Solution to Problem Set 3

1. (4 + 4 = 8 points) Multi-Merge

Suppose you would like to merge k sorted arrays, each of length n, into one sorted array of length nk.

(a) One strategy is to use the merge operation we studied in class to merge the first two arrays to obtain a new sorted array of size $2n$, then merge in the third to obtain a new sorted array of size $3n$, then merge in the fourth, and so on until you obtain the desired sorted array of length nk.

Formalize the above algorithm in pseudocode and analyze its time complexity, in terms of n and k.

Answer: Here is one way to write the algorithm. Let the arrays be labeled A_1, A_2, …, A_k.

```
1. Copy $A_1$ into array $B$.
2. For $i$ from 2 to $k$:
    (a) Merge $B$ with $A_i$ to obtain array $C$.
    (b) Copy $C$ into $B$.
```

Ignoring copying times, for the first merge, the running time is $2n$, for the second merge it is $3n$, and so on. So the total running time is

$$2n + 3n + 4n + \ldots + kn = n(2 + 3 + \ldots + k) = n(k^2/2 + k/2 - 1) = \Theta(nk^2).$$

Copying only doubles the total running time, so we have a tight $\Theta(nk^2)$ bound.

(b) Give a more efficient algorithm to this problem, using divide and conquer. Analyze its time complexity, in terms of n and k.

Answer: Here is a more efficient divide-and-conquer algorithm. We split the k lists into two parts, one containing $\lfloor k/2 \rfloor$ lists and the other containing $\lceil k/2 \rceil$ lists. Then, we recursively call the algorithm on the two parts and obtain two sorted lists, the total number of elements being nk. Finally, we merge the two lists in $O(nk)$ time using the merge algorithm covered in class to obtain a single sorted list with nk elements. The correctness of the algorithm directly follows from the correctness of the merge algorithm.

The recurrence for the worst-case running time is calculated as follows. Let $T(k)$ denote the worst-case running time of merging together k sorted lists, each of length n. Clearly, $T(1) = \Theta(1)$. And the recursion yields:

$$T(k) = 2T(k/2) + nk.$$

By unrolling the recurrence, we get

$$T(k) = nk + nk + \ldots(\lg k \text{ times }) = nk \lg k.$$

Problem 2. (8 points) Finding widgets of majority type using pairwise testing

You have n widgets, each of which is of a certain *type*. You want to determine whether there is a *majority type*; i.e., if there is a type t such that the number of widgets of type t is *greater than* $n/2$. For instance, the set of 7 widgets with types A, A, B, C, A, C, A, respectively has a majority type (A) since there are 4 widgets of that type. On the other hand, the set of 6 widgets with types A, A, B, C, D, A has no majority type.

Unfortunately, you are unable to determine the type of any given widget. Instead, the only operation available to you is to call a subroutine EqualityTest that takes two widgets as input and returns *True* if both are of the same type and *False* otherwise.

Give a divide-and-conquer algorithm for determining if there is a majority type in a given set of n widgets. The running time of your algorithm is the number of calls made by your algorithm to EqualityTest. Analyze the running time of your algorithm.

Ideally, the running time of your algorithm should be $O(n \log n)$. You will receive extra credit if the running time of your algorithm is $O(n)$.

Answer: The following algorithm returns a majority type widget, if one exists, and $\perp$ otherwise. The basic idea is to split the widgets into two halves and compute a majority type of each half. If any of these types is a majority in the whole array, then return it; otherwise, return $\perp$. Let A denote the array consisting of the n widgets.

```
1. Majority($A, p, q$)
2. if $q - p < 1$
3. return $A[p]$
4. else
5.    $r \leftarrow \lfloor (q+p)/2 \rfloor$
6.    $m_1 \leftarrow$ Majority($A, p, r$)
7.    $m_2 \leftarrow$ Majority($A, r+1, q$)
8.    if EqualityTest($m_1, m_2$)
9.    return $m_1$
10.   else
11.      if $m_1 \neq \perp$ and $m_1$ passes EqualityTest for more than $(q-p+1)/2$ widgets
12.      return $m_1$
13.      if $m_2 \neq \perp$ and $m_2$ passes EqualityTest for more than $(q-p+1)/2$ widgets
14.      return $m_2$
15.      return $\perp$
```

For completeness, we give a detailed proof of correctness. The algorithm can be proved correct by induction on the size n of the list A. The base case $n = 1$ is clearly correct since the lone widget is of majority type of the list, and the algorithm does return this widget (line 3).

For the induction hypothesis, suppose the algorithm is correct on lists of size smaller than n. Consider the algorithm on a list of length n. Let A_1 and A_2 be the two halves. Let m_1 (resp., m_2) be the majority widget of A_1 (resp., A_2), if it exists, and $\perp$ otherwise. By the induction hypothesis, the values m_1 and m_2 computed in lines 6 and 7 are correct. We now claim that no widget type other than that of m_1 or m_2 can be the majority type of A. This is because any other widget types occurs at most $|A_1|/2 + |A_2|/2 = n/2$ times, and thus cannot be a majority element of A.

Therefore, the majority type of A can now be computed by counting the number of other widgets of the same type as m_1 and m_2, respectively, in A and returning one, if it occurs more than $n/2$ times, and $\perp$ otherwise. This is exactly what the computations in lines 11 through 15 do.

For the running time, we have a recurrence:

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

which we already know is $\Theta(n \log n)$.

A linear-time algorithm

If A has only one widget, then return the lone widgets. Otherwise: (a) pair up the widgets of A arbitrarily, to get $\lfloor n/2 \rfloor$ pairs; (b) if two widgets of a pair are different, then discard both of them, else keep just one of them; (c) recursively call the procedure on the remaining widgets.

We will argue its correctness and analyze its running time. Let A be a given list of n widgets. Furthermore, let A have a majority widget and let this widget be m. If A has only one widget, then the claim is trivial. Otherwise, let B be the list of widgets obtained after step (b). We will show that m is also of majority type in B, from which the correctness follows using induction. We know that the number of occurences of m in A is greater than the total number of occurences of all other widgets in A. In obtaining B from A, whenever we discard m in step (b), we discard one widget of different type than m as well! So the number of occurences of m in B is also greater than the total number of occurences of all other widgets in B. So m is of the majority type of B as well, completing the proof.

For the running time, we get the recurrence $T(1) = 1$ and for $n > 1$

$$T(n) = T(n/2) + n,$$

which, on unraveling and applying the formula for geometric progressions, yields

$$T(n) = n + n/2 + \ldots = \Theta(n).$$

Problem 3. (4 points) Finding the end of an infinite array?

Suppose you are given a very long array A, whose first n elements are integers (in arbitrary order) and the remaining elements are the special symbol ∞. You can access any position i in A by referring to $A[i]$. However, you do not know n.

Give an algorithm for determining n. Analyze the number of accesses to A made by your algorithm, in terms of n. For full credit, your algorithm must make $O(\log n)$ accesses to A.

Answer: Here is the algorithm.

```
1. set $i = 1$;
2. while $A[i] < \infty$: $i = 2 \cdot i$;
3. $\ell = i/2; h = i$;
4. while $h > \ell + 1$:
    (a) $m = \lfloor (\ell + h)/2 \rfloor$;
    (b) if $A[m] = \infty$ then $h = m$ else $\ell = m$;
5. return $\ell$.
```

The first while loop determines an i such that $A[i/2] < \infty$ and $A[i] = \infty$. Clearly, $i/2 \leq n < i$. So $i \leq 2n$. Since the while loop doubles i in every iteration, the running time of the while loop is $O(\log n)$.

The second while loop conducts a sort of binary search on an sub-array of size $i/2 = \Theta(n)$ and on completion, satisfies $A[\ell] \neq \infty$ and $A[h] = \infty$. So this also takes time $\Theta(\log n)$. The algorithm returns ℓ as desired.