CSC-421 Applied Algorithms and Structures
Spring 2021-22

Instructor: Iyad Kanj
Office: CDM 832
Phone: (312) 362-5558
Email: `ikanj@cs.depaul.edu`
Office Hours (Office/Zoom): Monday 4:40-5:40 & Wednesday 1:00-3:00
Course Website: https://d2l.depaul.edu/

Solution Key to the Sample Midterm

I. Using the iteration method, you can show that $T(n) = \Theta(n^2)$.

II. We only need to consider A[1..k] and B[1..k], i.e., a problem with $2k$ entries overall. Let $q = (k+1)/2$, and $p = k - q$.

```
1. If A[q] = B[p] then q - 1 elements in A and p - 1 elements in B are
no greater than A[q] or B[p], and hence, either A[q] or B[p] is the
k-th smallest element.

2. If A[q] < B[p] then the k-th smallest element must be either in A[q+1..k]
or in B[1..p]. In fact, the k-th smallest element of the original problem
will be the (k-q)-th smallest element in the arrays A[q+1..k] and B[1..p]
each of size k - q.

3. If A[q] > B[p] then the k-th smallest element must be either in A[1..q]
or in B[p + 1..k] and the k-th smallest element of the original problem
will be the (k -p)-th smallest element in the arrays A[1..q] and B[p+1, k]
each of size k-p.
```

The algorithm uses the above idea to either solve the problem (case 1) or recursively solve the problem by solving a subproblem of half the size. The recurrence relation is $T(2k) = T(k) + O(1)$, with a running time of running time $O(\lg k)$.

III. 1. The idea is to divide the array into two subarrrays each with n/2 elements. The algorithm would then find the maximum and minimum elements of each of the two halves, by recursive applications of the algorithm. The specifications for our algorithm are:

```
Input: An array A[1..n]  and indices 1 <= p <= r <= n.
Output: The minimum element m and the maximum element M in A.


Find-Min-Max(A, p, r, m, M)

if p = r - 1 then
  if A[p] < A[r]
    m = A[p];
    M = A[r];
  else
    m = A[r];
    M = A[p];
else
  q =(p+r)/2;
  Find-Min-Max(A, p, q, m1, M1);
  Find-Min-Max(A, q+1, r, m2, M2);
  if m1 < m2
    m = m1;
  else
    m = m2;
  if M1 < M2
    m = M2;
  else
    m = M1;
```

The number of comparisons is described by the recurrence $T(n) = 2T(n/2) + 2$ if $n > 2$, and $T(2) = 1$. Using the iteration method, we can show that $T(n) = 3n/2 - 2$.

2. Here is the idea. We arrange the n elements into pairs, then compare the two elements within each pair, marking the larger of the two. This costs $n/2$ comparisons. Then go through the $n/2$ larger elements, keeping track of the current maximum, and similarly through the $n/2$ smaller elements, keeping track of the

current minimum. This costs twice a total of $n/2 + 2(n/2 - 1) = 3n/2 - 2$ comparisons.

Here is a formal description of a more streamlined version that does not use any marking:

```
Iter-FMM(A, n)

if A[1] < A[2] then
  m = A[1];
  M=A[2];
else
  m=A[2];
  M=A[1];

for i = 2 to n/2 do
  if A[2i-1] < A[2i] then
    if A[2i-1] < m then
       m = A[2i-1];
    if M < A[2i] then
       M = A[2i];
  else
    if A[2i] < m then
       m = A[2i];
    if M < A[2i-1] then
       M = A[2i-1];

return (m, M);
```

IV. We sort (the smaller array) A using Merge Sort in time $O(m \lg m)$. Afterwards, we intialize an auxiliary array C to contain all elements in A; this takes $O(m)$ time. Then, for each of the n elements in B, we check if the element is in A using Binary Search, and add it to C only if the element is not in A; we return C at the end as $A \cup B$. Each call to Binary Search takes $O(\lg m)$ time, for a total of $O(n \lg m)$ time over all elements of B. The overall running time is $O(m \lg m) + O(m) + O(n \lg m) = O(n \lg m)$.

V. We pair up the m arrays and merge each pair using the subroutine Merge(). This takes $O(n \cdot m)$ time and results in $m/2$ arrays, each of size $2n$, on which we recurse. The depth of the recursion is $O(\lg m)$, and at each level merging the arrays takes $O(n \cdot m)$ time, resulting in an overall running time of $O(n \cdot m \cdot \lg m)$.