CS/ECE 374 Spring 2023 William Cheng (shihuac2@illinois.edu)

Homework 5 Problem 1

Solution: First, sort the arrays A, B using A as the key, in ascending order. In other words, sort the intervals according to their starting point. This can be done in $O(n \log n)$ time using mergesort or quicksort.

After sorting, divide the intervals into two halves A_l, B_l and A_r, B_r. Suppose we already know the maximum overlap length of these two halves. Then the maximum overlap length of A, B is the maximum of:

i. The maximum overlap length of (A_l, B_l).

ii. The maximum overlap length of (A_r, B_r).

iii. The maximum overlap length of two intervals such that one is taken from (A_l, B_l) and the other is taken from (A_r, B_r).

For case (iii), let $(a_l, b_l) \in (A_l, B_l)$ and $(a_r, b_r) \in (A_r, B_r)$ be the two intervals with maximum overlap length. Then $a_l < a_r$ because the array is sorted. Therefore, the overlap interval must start with a_r. It ends with $min(b_l, b_r)$. If b_l is maximized, then we only need to search once through (A_r, B_r) to find a maximum $min(b_l, b_r) - a_r$, which is the maximum overlap length indicated in (iii). Finding a maximum b_l requires $O(|B_l|)$ time, and searching through (A_r, B_r) requires $O(|B_r|)$ time. The sum is $O(|B_l|) + O(|B_r|) \leq O(|B|) = O(n)$ time. Finding the maximum of the three cases above takes $O(1)$ time, so the total cost for the algorithm without recursion is $O(n)$. The base case is when $n = 1$, there is only one interval and no overlap so the algorithm returns 0. The algorithm is on the next page.

```
MaxOverlapIntervals(A, B):
  sort A,B in ascending order using A as key
  n ← A.length
  _, (a_i, b_i), (a_j, b_j) ← Recurse(A, B, n)
  return (a_i, b_i), (a_j, b_j)

Recurse(A, B, n):
  if n ≤ 1
      return 0, nil, nil
  m ← ⌈n/2⌉
  lmax,(a_{i,l}, b_{i,l}),(a_{j,l}, b_{j,l}) ← Recurse(A[0:m-1], B[0:m-1], m)
  rmax,(a_{i,r}, b_{i,r}),(a_{j,r}, b_{j,r}) ← Recurse(A[m:n-1], B[m:n-1], n-m)
  (a_i, b_i) ← (0, −∞)
  for k ← 0 to m-1
      if B[k] > b_i
          (a_i, b_i) ← (A[k], B[k])
  crossmax ← 0
  for k ← m to n-1
      overlap ← max{0, min(b_i, B[k]) - A[k]}
      if overlap > crossmax
          crossmax ← overlap
          (a_j, b_j) ← (A[k], B[k])
  if rmax ≥ crossmax and rmax ≥ lmax
      return rmax, (a_{i,r}, b_{i,r}), (a_{j,r}, b_{j,r})
  else if lmax ≥ crossmax and lmax ≥ rmax
      return lmax, (a_{i,l}, b_{i,l}), (a_{j,l}, b_{j,l})
  else
      return crossmax, (a_i, b_i), (a_j, b_j)
```

The algorithm returns two intervals (a_i, b_i) and (a_j, b_j). If no intervals overlap, the algorithm returns *nil, nil*. The recurrence relation for Recurse is $T(n) = 2T(n/2) + O(n)$, which gives $T(n) = O(n \log n)$. The time complexity of sorting is $O(n \log n)$. Therefore, the total time complexity of MaxOverlapIntervals is $O(1) + O(n \log n) + O(n \log n) = O(n \log n)$. ■

CS/ECE 374 Spring 2023 William Cheng (shihuac2@illinois.edu)
Homework 5 Problem 2

Solution: (a) Let K represent the ranks $k_1, k_2, ..., k_h$. We begin by splitting K into two halves, K_l and K_r. We then find the rank $K_r[0]$ element in A and call it m. Then we can divide A into A_l and A_g where $\max(A_l) < m$ and $\min(A_g) \geq m$. Then we subtract everything in K_r by $K_r[0]$ such that K_r starts with 0. Then we can recursively apply the algorithm on A_l, K_l and A_g, K_r and denote the return values as B_l and B_r. The concatenation $B_l B_r$ is the elements of rank K in A since every element is B_r is greater than any element in B_l. The base case is when $h = 1$, $K = \{k_1\}$, therefore the algorithm returns the set of the $k_1 th$ smallest element in A. The algorithm is shown below.
Note: Zero-based indices are used.

```
ElementsOfRank(A, K, h):
  if h = 0
      return nil
  if h = 1
      b ← K[0]th smallest element in A
      return [b]
  p ← ⌈h/2⌉
  K_l ← K[0 : p − 1]; K_r ← K[p : h − 1]
  q ← K_r[0]
  m ← qth smallest element in A
  A_l ← []; A_g ← []
  for each a ∈ A
      if a < m
          add a to A_l
      else
          add a to A_g
  for i ← 0 to h − p − 1
      K_r[i] ← K_r[i] − q
  B_l ← ElementsOfRank(A_l, K_l, p)
  B_r ← ElementsOfRank(A_g, K_r, h − p)
  return B_l B_r //concatenation
```

From the algorithm we can write the recurrence $T(h) = 2T(h/2) + O(n) + O(h)$ and $T(1) = O(n)$ since selection costs $O(n)$. However, there is a lower bound. Consider the recursion tree. The depth of the tree is $\log h$. Although the array A is split randomly during each recursion, it must be true that the sum of the work on each level is $O(n) + O(h) = O(n)$ because only linear operations are involved with A so no matter how A is split the sum is still $O(|A|)$. Therefore the time complexity $=$ *work on each level* $\times$ *number of levels* $= O(n \log h)$.

(b) We begin by calculating the mid indices of all four arrays, m_1, m_2, m_3, m_4.

- If $m_1 + m_2 + m_3 + m_4 < k$, then since all the arrays are sorted, at least one subarray $A_1[1 : m_1], A_2[1 : m_2], A_3[1 : m_3], A_4[1 : m_4]$ can be dismissed. The one with the smallest number at index m_i should be dismissed since it would be filled first. Then we adjust $k \leftarrow k - m_i$ and recurse on the adjusted arrays.
- If $m_1 + m_2 + m_3 + m_4 \geq k$, let n_i be the length of A_i, then similarly the subarray $A_i[m_i : n_i]$ with the largest number at index m_i should be dismissed since reaching it

would require $k > m_1 + m_2 + m_3 + m_4$ which is a contradiction. Then we recurse on k and the adjusted arrays.

The base case is when exactly one array is non-empty, the algorithm returns the *kth* element of that array.

```
FindKthRank(A_1, A_2, A_3, A_4, k):
  if exactly one A_i is non-empty
      return A_i[k]
  n_1 ← A_1.length; n_2 ← A_2.length; n_3 ← A_3.length; n_4 ← A_4.length
  m_1 ← ⌈n_1/2⌉; m_2 ← ⌈n_2/2⌉; m_3 ← ⌈n_3/2⌉; m_4 ← ⌈n_4/2⌉
  if m_1 + m_2 + m_3 + m_4 < k
      p ← min(A_1[m_1], A_2[m_2], A_3[m_3], A_4[m_4])
      for each A_i ∈ {A_1, A_2, A_3, A_4}
          if A_i is empty
              continue
          if A_i[m_i] = p
              if n_i > 1
                  A_i ← A_i[m_i + 1 : n_i]
              else
                  A_i ← [ ]
              k ← k − m_i
              break
      return FindKthRank(A_1, A_2, A_3, A_4, k)
  else
      p ← max(A_1[m_1], A_2[m_2], A_3[m_3], A_4[m_4])
      for each A_i ∈ {A_1, A_2, A_3, A_4}
          if A_i is empty
              continue
          if A_i[m_i] = p
              if m_i > 1
                  A_i ← A_i[1 : m_i − 1]
              else
                  A_i ← [ ]
              break
      return FindKthRank(A_1, A_2, A_3, A_4, k)
```

Time complexity analysis: $T(1) = O(1)$ since we only need to retrieve one element from an array. Only one branch of the if-statement is executed, so there is exactly one recursive call in each function call. Let's just assume that each array A_i has exactly n elements which will be strictly larger than the actual situation. Then it takes $\lceil \log n \rceil + 1$ iterations for each A_i to be empty, hence $3(\lceil \log n \rceil + 1) + \lceil \log n \rceil = 4\lceil \log n \rceil + 3$ iterations in total. Only $O(1)$ operations are involved during each iteration. Therefore, an upper bound for the time complexity is $O(\log n)$.

■