CS/ECE 374 Sec A ✧ Spring 2023

Homework 5

Due Wednesday, March 1, 2023 at 10am

---

1. Consider n intervals $I_1, I_2, \ldots, I_n$. Each interval I_i is specified by its two end points a_i and b_i with $a_i \leq b_i$. Two intervals I_i and I_j overlap if there is a number x such that $x \in [a_i, b_i]$ and $x \in [a_j, b_j]$. The overlap length between I_i and I_j is the geometrically natural one — the length of the longest interval shared between I_i and I_j. We can express this overlap length formally as the quantity:

$$\max\{0, \min(b_i, b_j) - \max(a_i, a_j)\}$$

   You may want to draw a picture to see the meaning of the formula. Given the n intervals we wish to find the two intervals I_i and I_j that have the maximum overlap length. You can assume that the intervals are specified in two arrays A and B of length n where $A[i] = a_i$ and $B[i] = b_i$. Describe an efficient algorithm for this problem. An $O(n^2)$ algorithm is straight forward. You should aim to beat this easy bound. You may want to first think of the conceptually easier setting where the a_i and b_i values are distinct. *Hint:* you can try Mergesort like divide and conquer.

2. Recall the Selection problem: given an unsorted array A of n integers and an index k between 1 and n, output the kth ranked number in the array. We saw a linear time algorithm for it in lecture. In this problem we see two variants of Selection.

   (a) Let A be an unsorted arrary of n elements. Suppose we are given h indices $k_1 < k_2 \ldots < k_h$. Describe an $O(n \log h)$ algorithm to find elements of ranks $k_1, k_2, \ldots, k_h$ in $O(n \log h)$ time. Note that one can use Selection h times to solve this problem in $O(nh)$ time. We can also do this via sorting in $O(n \log n)$ time which is advantageous when h is large. Here we are interested in the intermediate range when h is not too small but smaller than $\log n$. For istance consider $h = \Theta(\log \log n)$. The $O(nh)$-time algorithm will take $O(n \log \log n)$ time while the sorting based algorithm will take $O(n \log n)$ time while the $O(n \log h)$ time algorithm will achieve a running time of $O(n \log \log \log n)$ which is better.

   (b) Given 4 *sorted* arrarys A_1, A_2, A_3, A_4 with a total of n elements, and an index k between 1 and n, describe an $O(\log n)$ time algorithm to find the k'th ranked element in the union of the four arrays.

   (c) **Not to submit:** Instead of 4 sorted arrays as in the previous problem, suppose we had h sorted arrays. What running time can you achieve as a function of of h and n?

   You do not need to formally prove the correctness of the algorithms but they should be clear and high-level. You need to justify the running time of your algorithms.

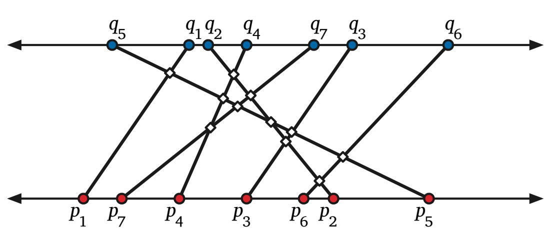
3. **Not to submit:** A **two-dimensional** Turing machine (2D TM for short) uses an infinite two-dimensional grid of cells as the tape. For simplicity assume that the tape cells corresponds to integers (i, j) with $i, j \geq 0$; in other words the tape corresponds to the positive quadrant of the two dimensional plane. The machine crashes if it tries to move below the $x = 0$ line or to the left of the $y = 0$ line. The transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, U, D, S\}$ where L, R, U, D stand for "left", "right", "up" and "down" respectively, and S stands for "stay put". You can assume that the input to the 2D TM is written on the first row and that its head is initially at location $(0, 0)$. Argue that a 2D TM can be simulated by an ordinary TM (1D TM); it may help you to use a multi-tape TM for simulation. In particular address the following points.

   - How does your TM store the grid cells of a 2D TM on a one dimensional tape?
   - How does your TM keep track of the head position of the 2D TM?
   - How does your 1D TM simulate one step of the 2D TM?

   If a 2D TM takes t steps on some input how many steps (asymptotically) does your simulating 1D TM take on the same input? Give an asymptotic estimate. Note that it is quite difficult to give a formal proof of the simulation argument, hence we are looking for high-level arguments similar to those we gave in lecture for various simulations.

Solved Problem

4. Suppose we are given two sets of n points, one set $\{p_1, p_2, \ldots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \ldots, q_n\}$ on the line $y = 1$. Consider the n line segments connecting each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time. See the example below.



Seven segments with endpoints on parallel lines, with 11 intersecting pairs.

   Your input consists of two arrays $P[1..n]$ and $Q[1..n]$ of x-coordinates; you may assume that all $2n$ of these numbers are distinct. No proof of correctness is necessary, but you should justify the running time.

   Solution: We begin by sorting the array $P[1..n]$ and permuting the array $Q[1..n]$ to maintain correspondence between endpoints, in $O(n \log n)$ time. Then for any indices

$i < j$, segments i and j intersect if and only if $Q[i] > Q[j]$. Thus, our goal is to compute the number of pairs of indices $i < j$ such that $Q[i] > Q[j]$. Such a pair is called an ***inversion***.

We count the number of inversions in Q using the following extension of mergesort; as a side effect, this algorithm also sorts Q. If $n < 100$, we use brute force in $O(1)$ time. Otherwise:

- Recursively count inversions in (and sort) $Q[1 \,..\, \lfloor n/2 \rfloor]$.
- Recursively count inversions in (and sort) $Q[\lfloor n/2 \rfloor + 1 \,..\, n]$.
- Count inversions $Q[i] > Q[j]$ where $i \le \lfloor n/2 \rfloor$ and $j > \lfloor n/2 \rfloor$ as follows:
  - Color the elements in the Left half $Q[1 \,..\, n/2]$ bLue.
  - Color the elements in the Right half $Q[n/2 + 1 \,..\, n]$ Red.
  - Merge $Q[1 \,..\, n/2]$ and $Q[n/2 + 1 \,..\, n]$, maintaining their colors.
  - For each blue element $Q[i]$, count the number of smaller red elements $Q[j]$.

The last substep can be performed in $O(n)$ time using a simple for-loop:

```
CountRedBlue(A[1..n]):
  count ← 0
  total ← 0
  for i ← 1 to n
      if A[i] is red
          count ← count + 1
      else
          total ← total + count
  return total
```

In fact, we can execute the third merge-and-count step directly by modifying the Merge algorithm, without any need for "colors". Here changes to the standard Merge algorithm are indicated in red.

```
MergeAndCount(A[1..n], m):
  i ← 1; j ← m + 1; count ← 0; total ← 0
  for k ← 1 to n
      if j > n
          B[k] ← A[i]; i ← i + 1; total ← total + count
      else if i > m
          B[k] ← A[j]; j ← j + 1; count ← count + 1
      else if A[i] < A[j]
          B[k] ← A[i]; i ← i + 1; total ← total + count
      else
          B[k] ← A[j]; j ← j + 1; count ← count + 1
  for k ← 1 to n
      A[k] ← B[k]
  return total
```

We can further optimize this algorithm by observing that *count* is always equal to $j - m - 1$. (Proof: Initially, $j = m + 1$ and $count = 0$, and we always increment j and *count* together.)

```
MergeAndCount2(A[1..n], m):
  i ← 1; j ← m + 1; total ← 0
  for k ← 1 to n
      if j > n
          B[k] ← A[i]; i ← i + 1; total ← total + j − m − 1
      else if i > m
          B[k] ← A[j]; j ← j + 1
      else if A[i] < A[j]
          B[k] ← A[i]; i ← i + 1; total ← total + j − m − 1
      else
          B[k] ← A[j]; j ← j + 1
  for k ← 1 to n
      A[k] ← B[k]
  return total
```

The modified MERGE algorithm still runs in $O(n)$ time, so the running time of the resulting modified mergesort still obeys the recurrence $T(n) = 2T(n/2) + O(n)$. We conclude that the overall running time is $O(n \log n)$, as required. ■

Rubric: 10 points = 2 for base case + 3 for divide (split and recurse) + 3 for conquer (merge and count) + 2 for time analysis. Max 3 points for a correct $O(n^2)$-time algorithm. This is neither the only way to correctly describe this algorithm nor the only correct $O(n \log n)$-time algorithm. No proof of correctness is required.