CS/ECE 374 ✧ Spring 2021
Homework 4

Due Thursday, March 11, 2021 at 10am

Groups of up to three people can submit joint solutions. Each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the Gradescope names and email addresses of each group member. In addition, whoever submits the homework must tell Gradescope who their other group members are.

The following unnumbered problems are not for submission or grading. No solutions for them will be provided but you can discuss them on Piazza.

- Problem 9 in Jeff's note on counting inversions. This is also a solved problem in Kleinberg-Tardos book. This is the simpler version of the solved problem at the end of this home work.
- We saw a linear time selection algorithm in class which is based on splitting the array into arrays of 5 elements each. Suppose we split the array into arrays of 7 elements each. Derive a recurrence for the running time.
- Suppose we are given n points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ in the plane. We say that a point (x_i, y_i) in the input is dominated if there is another point (x_j, y_j) such that $x_j > x_i$ and $y_j > y_i$. Describe an $O(n \log n)$ time algorithm to find all the *undominated* points in the given set of n points.
- Solve some recurrences in Jeff's notes.

1. Recall that a Turing Machine (TM) M decides a language L if on any input string w the machine M halts in an accept state if $w \in L$ and in a reject state if $w \notin L$. In other words M is an algorithm for deciding membership in L. Note that we do not have any upper bound on the running time of M. We say that L is decidable if there is a TM M that decides L. The purpose of this problem is to show that decidable languages are closed under basic operations.

    - Show that if L_1, L_2 are decidable then $L_1 \cap L_2$ and $L_1 \cup L_2$ are decidable.
    - Show that if L_1 and L_2 are decidable then $L_1 L_2$ is decidable (concatenation).
    - Show that if L_1 is decidable then L_1^* is decidable.
    - **Not to submit:** Show that if L_1 and L_2 are decidable then $(L_1 \cup L_2)^*$ is decidable.

    For each of the problems you should describe a simple TM that decides the given language in a high-level fashion assuming that L_1 has a TM M_1 and L_2 has a TM M_2. One way to think of the problem is to give a program for the given language using sub-routines for deciding L_1 and L_2. No proof is necessary but clarity of your algorithm is important; give a brief description if necessary. Note that in decidability we do not pay attention to the quality of the running time so brute-force algorithms are fine.

2. Suppose you are given k sorted arrays $A_1, A_2, \dots, A_k$ each of which has n numbers. Assume that all numbers in the arrays are distinct. You would like to merge them into single sorted array A of kn elements. Recall that you can merge two sorted arrays of sizes n_1 and n_2 into a sorted array in $O(n_1 + n_2)$ time.

   - Use a divide and conquer strategy to merge the sorted arrays in $O(nk \log k)$ time. To prove the correctness of the algorithm you can assume a routine to merge two sorted arrays.
   - In MergeSort we split the array of size N into two arrays each of size $N/2$, recursively sort them and merge the two sorted arrays. Suppose we instead split the array of size N into k arrays of size N/k each and use the merging algorithm in the preceding step to combine them into a sorted array. Describe the algorithm formally and analyze its running time via a recurrence. You do not need to prove the correctness of the recursive algorithm.
   - **Extra credit:** This is a generalization of the first part. Suppose the k arrays are of potentially different sizes $n_1, n_2, \dots, n_k$ where $N = \sum_{i=1}^{k} n_i$. Describe and analyze an $O(N \log k)$ algorithm to obtain a sorted array.

3. Sorting is a fundamental and heavily used routine and can be done in $O(n \log n)$ time for a list of n numbers. In the comparison tree model there is a lower bound of $\Omega(n \log n)$ for sorting. Selection can be done in $O(n)$ time. Although a faster Selection algorithm may not be as directly useful in practice as Sorting, the ideas behind a linear time algorithm for it are theoretically interesting and related ideas play an important role in other problems. For each of the problems below use Selection as a black box algorithm to derive an $O(n)$ time algorithm.

   - It is common these days to hear statistics about wealth inequality in the United States. A typical statement is that the the top 1% of earners together make more than ten times the total income of the bottom 70% of earners. You want to verify these statements on some data sets. Suppose you are given the income of people as an n element *unsorted* array A, where $A[i]$ gives the income of person Describe an algorithm that given A checks whether the top 1% of earners together make more than ten times the bottom 70% together. Assume for simplicity that n is a multiple of 100 and that all numbers in A are distinct.
   - Describe an algorithm to determine whether an arbitrary array $A[1..n]$ contains more than $n/6$ copies of any value.
   - The *square distance* between a pair of integers x, y is defined as the quantity $(x - y)^2$. The input is an an array A of n integers and an integer k such that $1 \le k \le n$. Describe an algorithm to find k elements in A with the smallest square distance to the median (i.e. the element of rank $\lfloor n/2 \rfloor$ in A). For instance, if $A = [9, 5, -3, 1, -2]$ and $k = 2$, then the median element is 1, and the 2 elements in A with the smallest square distance to the median are $\{1, -2\}$. If $k = 3$, then you can output either $\{1, -2, -3\}$ or $\{1, -2, 5\}$.
   - **Not to submit:** Related to the first part. More generally we may want to compute the total earnings of the top α% of earners for various values of α. Suppose we are given A and k numbers $\alpha_1 < \alpha_2 < \dots < \alpha_k$ each of which is a number between 0 and 100 and we wish to compute the total earnings of the top α_i% of earners for each $1 \le i \le k$. Assume for simplicity that $\alpha_i n$ is an integer for each i. Describe an algorithm for this problem that runs in $O(n \log k)$ time. Note that sorting will allow you to solve the problem in $O(n \log n)$ time but when $k \ll n$, $O(n \log k)$ is faster. Note that an $O(nk)$ time algorithm is relative easy.

   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.

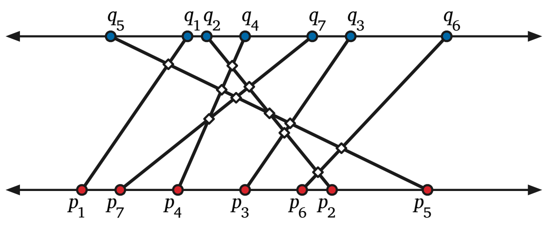
4. **Not to submit:** A **two-dimensional** Turing machine (2D TM for short) uses an infinite two-dimensional grid of cells as the tape. For simplicity assume that the tape cells corresponds to integers (i, j) with $i, j \geq 0$; in other words the tape corresponds to the positive quadrant of the two 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($A[1..n]$):
    $count \leftarrow 0$
    $total \leftarrow 0$
    for $i \leftarrow 1$ to $n$
        if $A[i]$ is red
            $count \leftarrow count + 1$
        else
            $total \leftarrow total + count$
    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($A[1..n], m$):
    $i \leftarrow 1$; $j \leftarrow m+1$; $count \leftarrow 0$; $total \leftarrow 0$
    for $k \leftarrow 1$ to $n$
        if $j > n$
            $B[k] \leftarrow A[i]$; $i \leftarrow i+1$; $total \leftarrow total + count$
        else if $i > m$
            $B[k] \leftarrow A[j]$; $j \leftarrow j+1$; $count \leftarrow count + 1$
        else if $A[i] < A[j]$
            $B[k] \leftarrow A[i]$; $i \leftarrow i+1$; $total \leftarrow total + count$
        else
            $B[k] \leftarrow A[j]$; $j \leftarrow j+1$; $count \leftarrow count + 1$
    for $k \leftarrow 1$ to $n$
        $A[k] \leftarrow B[k]$
    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($A[1..n], m$):
    $i \leftarrow 1$; $j \leftarrow m+1$; $total \leftarrow 0$
    for $k \leftarrow 1$ to $n$
        if $j > n$
            $B[k] \leftarrow A[i]$; $i \leftarrow i+1$; $total \leftarrow total + \mathbf{j - m - 1}$
        else if $i > m$
            $B[k] \leftarrow A[j]$; $j \leftarrow j+1$
        else if $A[i] < A[j]$
            $B[k] \leftarrow A[i]$; $i \leftarrow i+1$; $total \leftarrow total + \mathbf{j - m - 1}$
        else
            $B[k] \leftarrow A[j]$; $j \leftarrow j+1$
    for $k \leftarrow 1$ to $n$
        $A[k] \leftarrow B[k]$
    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.