CS/ECE 374 ✧ Fall 2018

Homework 4

Due Wednesday, October 10, 2018 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. 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.

2. Suppose you are given k sorted arrays $A_1, A_2, \ldots, A_k$ 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.

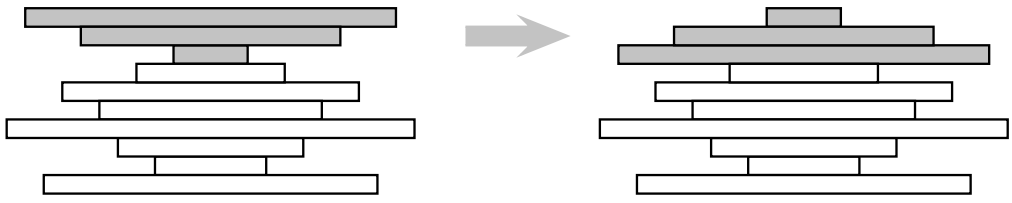
   - Use a divide and conquer strategy to merge the sorted arrays in $O(nk \log k)$ time. To prove the correctness of the algorithm you can assume a routine to merge two sorted arrays.
   - In MergeSort we split the array of size N into two arrays each of size $N/2$, recursively sort them and merge the 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.

3. 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 i.

   - Describe an $O(n)$-time 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. Note that sorting A will easily solve the problem but will take $\Omega(n \log n)$ time.
   - 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 $\alpha_1 < \alpha_2 < \ldots < \alpha_k$ 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 $1 \leq i \leq k$. 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. *Hint:* Use the previous part with $\alpha_{k/2}$ first and then use divide and conquer.

   You should prove the correctness of the second part of the problem. It helps to write a recursive algorithm so that you can use induction to prove correctness.

4. **Not to submit:** Suppose we have a stack of n pancakes of different sizes. We want to sort the pancakes so that the smaller pancakes are on top of the larger pancakes. The only operation we can perform is a *flip* - insert a spatula under the top k pancakes, for some k between 1 and n, and flip them all over.



   (a) Describe an algorithm to sort an arbitrary stack of n pancakes and give a bound on the number of flips that the algorithm makes. Assume that the pancake information is given to you in the

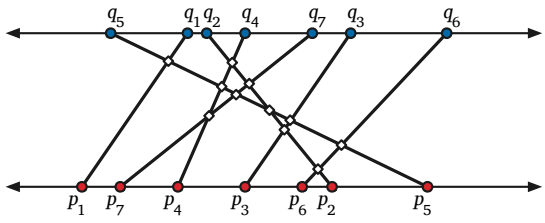
form of an n element array A. $A[i]$ is a number between 1 and n and $A[i] = j$ means that the j'th smallest pancake is in position i from the bottom; in other words $A[1]$ is the size of the bottom most pancake (relative to the others) and $A[n]$ is the size of the top pancake. Assume you have the operation Flip(k) which will flip the top k pancakes. Note that you are only interested in minimizing the number of flips.

(b) Suppose one side of each pancake is burned. Describe an algorithm that sorts the pancakes with the additional condition that the burned side of each pancake is on the bottom. Again, give a bound on the number of flips. In addition to A, assume that you have an array B that gives information on which side of the pancakes are burned; $B[i] = 0$ means that the bottom side of the pancake at the i'th position is burned and $B[i] = 1$ means the top side is burned. For simplicity, assume that whenever Flip(k) is done on A, the array B is automatically updated to reflect the information on the current pancakes in A.

5. **Not to submit:** Describe an algorithm to determine in $O(n)$ time whether an arbitrary array $A[1..n]$ contains more than $n/6$ copies of any value.

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.