

CS/ECE 374 ✧ Spring 2017

☺ Homework 4 ☺

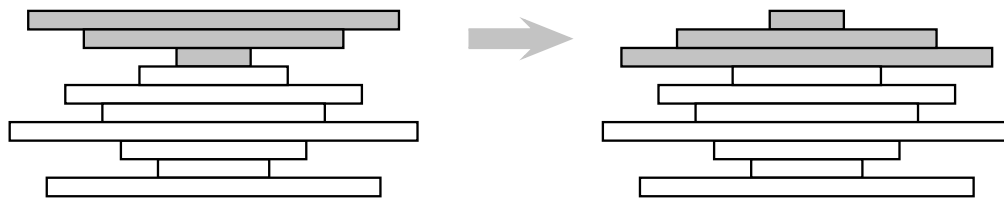
Due Wednesday, March 1, 2017 at 10am

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 homework.
- 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 $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ 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. 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 $\text{Flip}(k)$ 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 $\text{Flip}(k)$ is done on A , the array B is automatically updated to reflect the information on the current pancakes in A .

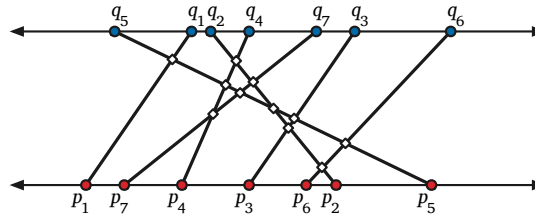
No proof of correctness necessary.

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.
3. (a) Describe an algorithm to determine in $O(n)$ time whether an arbitrary array $A[1..n]$ contains more than $n/4$ copies of any value.
- (b) Describe and analyze an algorithm to determine, given an arbitrary array $A[1..n]$ and an integer k , whether A contains more than k copies of any value. Express the running time of your algorithm as a function of both n and k .

Do not use hashing, or radix sort, or any other method that depends on the precise input values.

Solved Problem

4. Suppose we are given two sets of n points, one set $\{p_1, p_2, \dots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \dots, q_n\}$ 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 $i \leq \lfloor n/2 \rfloor$ and $j > \lfloor n/2 \rfloor$ as follows:
 - Color the elements in the Left half $Q[1.. \lfloor n/2 \rfloor]$ **blue**.
 - Color the elements in the Right half $Q[\lfloor n/2 \rfloor + 1..n]$ **red**.
 - Merge $Q[1.. \lfloor n/2 \rfloor]$ and $Q[\lfloor n/2 \rfloor + 1..n]$, 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 ← 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.