HW 5: Extra problems

Instructors: Chan, Hassanieh, Har-Peled

CS/ECE 374: Algorithms & Models of Computation, Spring 2019 Version: **1.1**

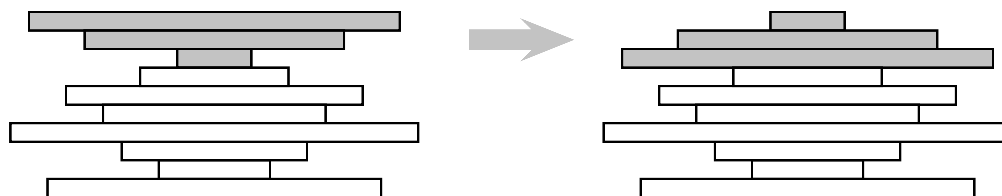
1 Problem 9 in Jeff's note on counting inversions. This is also a solved problem in Kleinberg-Tardos book. This is the simpler version of the solved problem at the end of this home work.

2 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.

3 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.

4 Solve some recurrences in Jeff's notes.

5 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.



5.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.

5.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.

No proof of correctness necessary.

6 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.

6.A. Use a divide and conquer strategy to merge the sorted arrays in $O(n \log k)$ time. To prove the correctnes of the algorithm you can assume a routine to merge two sorted arrays.

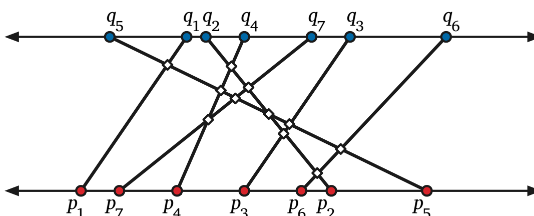
6.B. In MergeSort we split the array of size N into two arrays each of size $N/2$, recursively sort them and merge the two sorted arrays. Suppose we instead split the array of size N into k arrays of size N/k each and use the merging algorithm in the preceding step to combine them into a sorted array. Describe the algorithm formally and analyze its running time via a recurrence.

7 **7.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.

7.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.

8 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]
```

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.

9 Consider the following restricted variant of the Tower of Hanoi puzzle. The pegs are numbered 0, 1, and 2, and your task is to move a stack of n disks from peg 1 to peg 2. However, you are forbidden to move any disk *directly* between peg 1 and peg 2; *every* move must involve peg 0.

Describe an algorithm to solve this version of the puzzle in as few moves as possible. *Exactly* how many moves does your algorithm make?

10 Consider the following cruel and unusual sorting algorithm.

```
CRUEL(A[1..n]):
  if n > 1
        CRUEL(A[1..n/2])
        CRUEL(A[n/2+1..n])
        UNUSUAL(A[1..n])
```

```
UNUSUAL(A[1..n]):
if n = 2
  if A[1] > A[2]              the only comparison!
     swap A[1] ↔ A[2]
else
  for i ← 1 to n/4            swap 2nd and 3rd quarters
     swap A[i+n/4] ↔ A[i+n/2]
  UNUSUAL(A[1..n/2])          // recurse on left half
  UNUSUAL(A[n/2+1..n])        // recurse on right half
  UNUSUAL(A[n/4+1..3n/4])     // recurse on middle half
```

Notice that the comparisons performed by the algorithm do not depend at all on the values in the input array; such a sorting algorithm is called ***oblivious***. Assume for this problem that the input size n is always a power of 2.

10.A. Prove by induction that CRUEL correctly sorts any input array. (**Hint:** Consider an array that contains $n/4$ 1s, $n/4$ 2s, $n/4$ 3s, and $n/4$ 4s. Why is this special case enough? What does UNUSUAL actually do?)

10.B. Prove that CRUEL would *not* correctly sort if we removed the for-loop from UNUSUAL.

10.C. Prove that CRUEL would *not* correctly sort if we swapped the last two lines of UNUSUAL.

10.D. What is the running time of UNUSUAL? Justify your answer.

10.E. What is the running time of CRUEL? Justify your answer.

11 You are a visitor at a political convention (or perhaps a faculty meeting) with n delegates. Each delegate is a member of exactly one political party. It is impossible to tell which political party any delegate belongs to. In particular, you will be summarily ejected from the convention if you ask. However, you *can* determine whether any pair of delegates belong to the *same* party or not simply by introducing them to each other. Members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.

11.A. Suppose more than half of the delegates belong to the same political party. Describe and analyze an efficient algorithm that identifies every member of this majority party.

11.B. Now suppose precisely p political parties are present and one party has a plurality: more delegates belong to that party than to any other party. Please present a procedure to pick out the people from the plurality party as parsimoniously as possible.[1] Do *not* assume that $p = O(1)$.

[1]Describe and analyze an efficient algorithm that identifies every member of the plurality party.