

CS/ECE 374 ✧ Fall 2016

☯ Homework 4 ☯

Due Tuesday, October 4, 2016 at 8pm

1. 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?

2. 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.

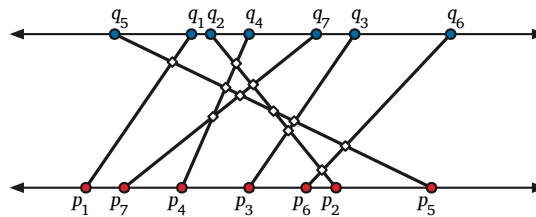
- (a) Prove by induction that CRUEL correctly sorts any input array. [Hint: Consider an array that contains $n/4$ 1s, $n/4$ 2s, $n/4$ 3s, and $n/4$ 4s. Why is this special case enough? What does UNUSUAL actually do?]
- (b) Prove that CRUEL would *not* correctly sort if we removed the for-loop from UNUSUAL.
- (c) Prove that CRUEL would *not* correctly sort if we swapped the last two lines of UNUSUAL.
- (d) What is the running time of UNUSUAL? Justify your answer.
- (e) What is the running time of CRUEL? Justify your answer.

3. 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.
- (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.
 - (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 $p = O(1)$.

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

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.