CS/ECE 374 A ✧ Fall 2023

Homework 5

Due Tuesday, October 3, 2023 at 9pm Central Time

---

1. In the lab on Wednesday, you'll see an algorithm that finds a local minimum in a one-dimensional array in $O(\log n)$ time. This question asks you to consider two higher-dimensional versions of this problem.

   (a) Suppose we are given a two-dimensional array $A[1..n, 1..n]$ of distinct integers. An array element $A[i, j]$ is called a ***local minimum*** if it is smaller than its four immediate neighbors:

   $$A[i,j] < \min\left\{A[i-1,j],\ A[i+1,j],\ A[i,j-1],\ A[i,j+1]\right\}$$

   To avoid edge cases, we assume all cells in row 1, row n, column 1, and column n have value $+\infty$.

   Describe and analyze an algorithm to find a local minimum in A as quickly as possible. (Remember that faster algorithms are worth more points, but only if they are correct.)

   [Hint: Suppose $A[i,j]$ is the smallest element in row i. If $A[i,j]$ is smaller than both of its vertical neighbors $A[i-1,j]$ and $A[i+1,j]$, we are clearly done. But what if $A[i,j] > A[i+1,j]$?]

   *[Hint: This problem is more subtle than it appears at first glance; many published solutions for this problem on the internet are incorrect. The main issue is that a local minimum in a rectangular subarray is not necessarily a local minimum in the original array. Design a recursive algorithm for the following more general problem: Given a two-dimensional array that contains a local minimum **whose value is less than the value of every border cell**, find such a local minimum.]*

   (b) Now suppose we are given a three-dimensional array $A[1..n, 1..n, 1..n]$ of distinct integers. An array element $A[i, j, k]$ is called a ***local minimum*** if it is smaller than its six immediate neighbors:

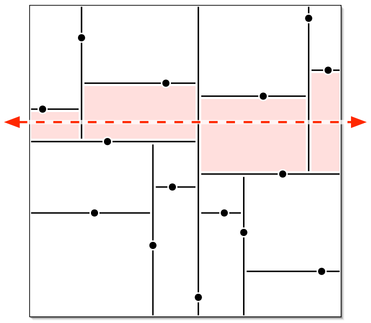
   $$A[i,j] < \min\left\{\begin{matrix}A[i-1,j,k],\ A[i+1,j,k],\\ A[i,j-1,k],\ A[i,j+1,k],\\ A[i,j,k-1],\ A[i,j,k+1]\end{matrix}\right\}$$

   To avoid edge cases, we assume all cells on the boundary of the array have value $+\infty$.

   Describe and analyze an algorithm to find a local minimum in A as quickly as possible.

   (Remember that faster algorithms are worth more points, but only if they are correct.)

2. Suppose we have n points scattered inside a two-dimensional box. A *kd-tree* recursively subdivides the points as follows. First we split the box into two smaller boxes with a *vertical* line, then we split each of those boxes with *horizontal* lines, and so on, always alternating between horizontal and vertical splits. Each time we split a box, the splitting line partitions the rest of the interior points *as evenly as possible* by passing through a median point in the interior of the box (*not* on its boundary). If a box doesn't contain any points, we don't split it any more; these final empty boxes are called *cells*.



A kd-tree for 15 points. The dashed line crosses the four shaded cells.

(a) How many cells does the kd-tree have, as a function of n? Prove that your answer is correct.

(b) In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n? Prove that your answer is correct. Assume that $n = 2^k - 1$ for some integer k. *[Hint: There is more than one function f such that $f(15) = 4$.]*

(c) Suppose we have n points stored in a kd-tree. Describe and analyze an algorithm that counts the number of points above a given horizontal line (such as the dashed line in the figure) as quickly as possible. *[Hint: Use part (b).]*

I should have specified that the following information is stored in each internal node v in the kd-tree:

- $v.x$ and $v.y$: The coordinates of the point defining the cut at v
- $v.dir \in \{vertical, horizontal\}$: The direction of the cut at v.
- $v.left$ and $v.right$: The children of v if $v.dir = vertical$
- $v.up$ and $v.down$: The children of v if $v.dir = horizontal$
- $v.size$: the number of points=cuts in the subtree rooted at v.

Instead I allowed arbitrary information to be computed in preprocessing; that freedom allows a much simpler and more efficient query algorithm!

(d) Describe and analyze an efficient algorithm that counts, given a kd-tree storing n points, the number of points that lie inside a given rectangle R with horizontal and vertical sides. *[Hint: Use part (c).]*

Assume that all x-coordinates and y-coordinates are distinct; that is, no two points lie on the same horizontal line or the same vertical line, no point lies on the query line in part (c), and no point lies on the boundary of the query rectangle in part (d).

*3. ***Practice only. Do not submit solutions.***

The following variant of the infamous StoogeSort algorithm[1] was discovered by the British actor Patrick Troughton during rehearsals for the 20th anniversary *Doctor Who* special "The Five Doctors".[2]

```
WhoSort($A[1..n]$):
  if $n < 13$
      sort $A$ by brute force
  else
      $k = \lceil n/5 \rceil$
      WhoSort($A[1..3k]$)          ⟨⟨Hartnell⟩⟩
      WhoSort($A[2k+1..n]$)        ⟨⟨Troughton⟩⟩
      WhoSort($A[1..3k]$)          ⟨⟨Pertwee⟩⟩
      WhoSort($A[k+1..4k]$)        ⟨⟨Davison⟩⟩
```

(a) Prove by induction that WhoSort correctly sorts its input. *[Hint: Where can the smallest k elements be?]*

(b) Would WhoSort still sort correctly if we replaced "if $n < 13$" with "if $n < 4$"? Justify your answer.

(c) Would WhoSort still sort correctly if we replaced "$k = \lceil n/5 \rceil$" with "$k = \lfloor n/5 \rfloor$"? Justify your answer.

(d) What is the running time of WhoSort? (Set up a running-time recurrence and then solve it, ignoring the floors and ceilings.)

(e) Forty years later, 15th Doctor Ncuti Gatwa discovered the following optimization to WhoSort, which uses the standard Merge subroutine from mergesort, which merges two sorted arrays into one sorted array.

```
NuWhoSort($A[1..n]$):
  if $n < 13$
      sort $A$ by brute force
  else
      $k = \lceil n/5 \rceil$
      NuWhoSort($A[1..3k]$)                 ⟨⟨Grant⟩⟩
      NuWhoSort($A[2k+1..n]$)               ⟨⟨Whittaker⟩⟩
      Merge($A[1..2k]$, $A[2k+1..4k]$)      ⟨⟨Tennant⟩⟩
```

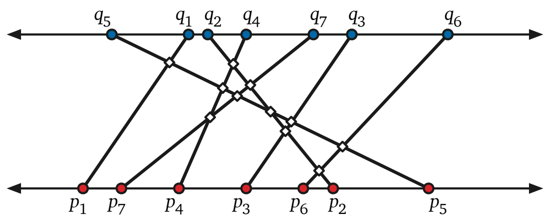
What is the running time of NuWhoSort?

[1] https://en.wikipedia.org/wiki/Stooge_sort

[2] Tom Baker, the fourth Doctor, declined to return for the reunion; hence, only four Doctors appeared in "The Five Doctors". (Well, okay, technically the BBC used excerpts of the unfinished episode "Shada" to include Baker, but he wasn't really *there*—to the extent that any fictional character in a television show about a time traveling wizard arguing with several other versions of himself about immortality can be said to be "really" "there".)

Solved problems

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:

- 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.
- Recursively count inversions in (and sort) the blue subarray $Q[1 .. \lfloor n/2 \rfloor]$.
- Recursively count inversions in (and sort) the red subarray $Q[\lfloor n/2 \rfloor + 1 .. n]$.
- Count red/blue inversions as follows:
  - Merge the sorted subarrays $Q[1 .. n/2]$ and $Q[n/2 + 1 .. n]$, maintaining the element colors.
  - For each blue element $Q[i]$ of the now-sorted array $Q[1 .. n]$, 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($A[1..n]$):
    $count \gets 0$
    $total \gets 0$
    for $i \gets 1$ to $n$
        if $A[i]$ is red
            $count \gets count + 1$
        else
            $total \gets total + count$
    return $total$
```

Merge and CountRedBlue each run in $O(n)$ time. Thus, the running time of our inversion-counting algorithm obeys the mergesort recurrence $T(n) = 2T(n/2) + O(n)$. (We can safely ignore the floors and ceilings in the recursive arguments.) We conclude that the overall running time of our algorithm is $O(n \log n)$, as required.

Rubric: This is enough for full credit.

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($A[1..n], m$):
    $i \gets 1$;  $j \gets m+1$;  $count \gets 0$;  $total \gets 0$
    for $k \gets 1$ to $n$
        if $j > n$
            $B[k] \gets A[i]$;  $i \gets i+1$;  $total \gets total + count$
        else if $i > m$
            $B[k] \gets A[j]$;  $j \gets j+1$;  $count \gets count + 1$
        else if $A[i] < A[j]$
            $B[k] \gets A[i]$;  $i \gets i+1$;  $total \gets total + count$
        else
            $B[k] \gets A[j]$;  $j \gets j+1$;  $count \gets count + 1$
    for $k \gets 1$ to $n$
        $A[k] \gets B[k]$
    return $total$
```

We can further optimize MergeAndCount by observing that *count* is always equal to $j - m - 1$, so we don't need an additional variable. (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 $j > n$
          $B[k] \leftarrow A[i]$;  $i \leftarrow i+1$;  $total \leftarrow total + j - m - 1$
      else if $i > m$
          $B[k] \leftarrow A[j]$;  $j \leftarrow j+1$
      else if $A[i] < A[j]$
          $B[k] \leftarrow A[i]$;  $i \leftarrow i+1$;  $total \leftarrow total + j - m - 1$
      else
          $B[k] \leftarrow A[j]$;  $j \leftarrow j+1$
  for $k \leftarrow 1$ to $n$
      $A[k] \leftarrow B[k]$
  return $total$
```

MergeAndCount2 still runs in $O(n)$ time, so the overall running time is still $O(n \log n)$, as required. ■

Rubric: 10 points = 2 for base case + 2 for divide (split and recurse) + 4 for conquer (merge and count) + 2 for time analysis. 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.

Max 3 points for a correct $O(n^2)$-time algorithm.

Notice that each boxed algorithm is preceded by a clear English description of the task that algorithm performs—not how the algorithm works, but the relationship between its input and its output. **Each English description is worth 25% of the credit for that algorithm** (rounding to the nearest half-point). For example, the CountRedBlue algorithm is worth 4 points ("conquer"); the English description alone ("For each blue element $Q[i]$ of the now-sorted array $Q[1\,..\,n]$, count the number of smaller red elements $Q[j]$.") is worth 1 point.