# CS/ECE 374 A ✧ Fall 2025

# Homework 5

Due Tuesday, October 7, 2023 at 9pm Central Time

---

**Please read and understand the homework policies, especially our expectations about describing algorithms and the standard grading rubrics for algorithms questions.**

---

1. Prince Hutterdink and Princess Bumpercup are celebrating their recent marriage by inviting all the dukes and duchesses in the kingdom to sample the castle's celebrated wine cellars. Just before the drinking begins, the happy couple learns that exactly one of their $n$ bottles of wine has been laced with iocaine powder, one of the deadliest poisons known to man.

   Hutterdink and Bumpercup employ an army of Royal Tasters, who have built up an immunity to iocaine powder. To test a set $S$ of wine bottles, a Taster mixes one drop of wine from each bottle in $S$ and consumes the resulting mixture. If the mixture contains any iocaine, the Taster will become extremely ill, or as the Royal Miracle Workers optimistically put it, only *mostly* dead. (Anyone else who consumes iocaine quickly becomes *all* dead.)

   Each Taster must be paid 1000 guilders for each test they perform, so Hutterdink and Bumpercup want to use as few tests as possible. Fortunately, they have an unlimited supply of Tasters.

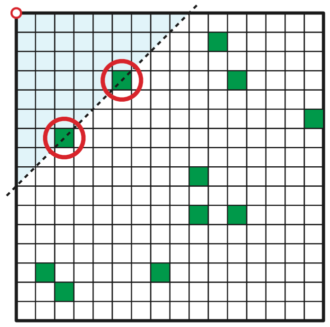
   (a) Describe an algorithm to find the poisoned bottle using at most $O(\log n)$ tests. (This is best possible in the worst case.)

   (b) Now suppose *two* of the $n$ bottles have been poisoned. Describe an algorithm to identify *both* poisoned bottles, using as few tests as possible in the worst case.

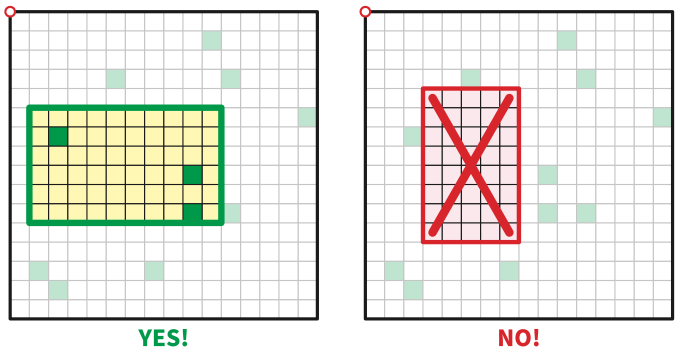
   (c) Finally, suppose the poisoners try to evade the Royal Tasters by reducing the amount of iocaine in each poisoned bottle. Now when a Taster tests a set $S$ of wine bottles, they become mostly dead if and only if *both* poisoned bottles are in $S$. Describe an algorithm to find *both* poisoned bottles using as few tests as possible in the worst case.

2. Before his untimely death, Captain Jack Sparrow buried several bottles of rum on Grid Island. You have a map of Rum Island in the form of an $n \times n$ grid of squares, with rows indexed from north to south and columns from west to east; each square on the map may or may not correspond to the location of a bottle of rum. The only safe port on Grid Island is in the northwest corner, next to square $(1, 1)$. You want to find a bottle of rum that is as close as possible to the port; specifically, you want to find a grid square $(i, j)$ that contains a bottle of rum, such that $i + j$ is as small as possible.

   For example, in the figure below, solid green squares represent bottles of rum; your ship is docked in the northwest corner. Your goal is to find one of the circled bottles of rum.



   To aid in your quest, Davy Jones has given you a magical Device that will correctly report whether any *rectangular* area on Grid Island contains at least one bottle of rum. That is, given four integers $t, b, l, r$ between 1 and $n$, if there is a rum bottle in any square $(i, j)$ such that $t \leq i \leq b$ and $l \leq j \leq r$, the Device sounds a gentle mellifluous chime; if there is no rum in that rectangular region, the Device emits a fart noise and a noxious cloud of green "smoke". By definition, each use of the Device takes $O(1)$ time.



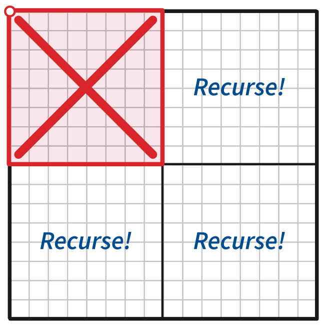
   You *could* use the Device to search the grid one square at a time, but that would require $\Theta(n^2)$ time in the worst case. Surely we can do better!

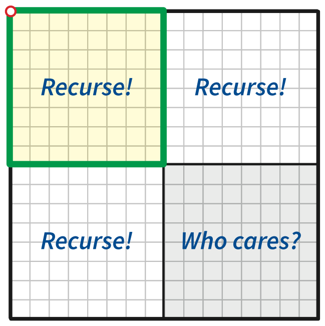
   (•) ***Practice only. Do not submit solutions.***

   Describe how to find an *arbitrary* bottle of rum (not necessarily closest to the northwest corner) in $O(\log n)$ time.

   (a) Consider the following recursive algorithm to find a bottle of rum closest to the northwest corner of the island. If $n = 1$, the only square must contain the only bottle of rum. Otherwise, split the $n \times n$ into four $n/2 \times n/2$ quadrants, use the Device to test the northwest quadrant, and then proceed as follows:

- If the northwest quadrant does not contain any rum, then it certainly does not contain the closest rum to your ship. Recursively search the northeast, southwest, and southeast quadrants.
- On the other hand, if the northwest quadrant does contain rum, then the closest rum to your ship cannot be in the southeast quadrant. Recursively search the northwest, northeast, and southwest quadrants.





In both cases, each of the three recursive calls either returns the location of the closest rum to your ship *in that quadrant* or correctly reports that there is no rum in that quadrant. Moreover, at least one of the recursive calls finds some rum. The closest rum to your ship is the closest of the one, two, or three bottles found by the recursive calls.

What is the overall running time of this recursive algorithm?

(b) Now suppose we modify the algorithm in part (a) to partition into nine regions instead of four. At each level of recursion, the new algorithm splits Grid Island into nine $(n/3) \times (n/3)$ subgrids, tests all nine subgrids using the Device, and then recursively searches a subset of those nine subgrids.

How many recursive calls do we need to make in the worst case? Prove your answer is correct. What is the running time of the resulting recursive algorithm?

*(c) Describe an even faster algorithm to find a bottle of rum that is closest to the northwest corner of Grid Island.

(As described in the homework policies and standard grading rubrics, full credit requires meeting the target running time that we have in mind. Slower correct algorithms that are still faster than parts (a) and (b) will receive significant partial credit.)

*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 = ⌈n/5⌉
      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 = ⌈n/5⌉
      NuWhoSort(A[1..3k])                 ⟨⟨Grant⟩⟩
      NuWhoSort(A[2k+1..n])               ⟨⟨Whittaker⟩⟩
      Merge(A[1..2k], A[2k+1..4k])        ⟨⟨Tennant⟩⟩
```

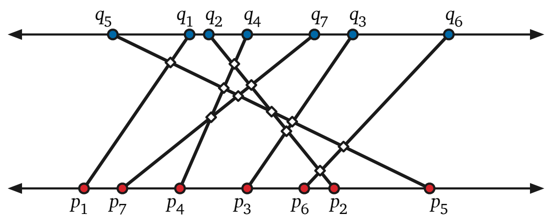
What is the running time of NuWhoSort?

[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.