CS/ECE 374 Sec A ✧ Spring 2025

Homework 5

Due Wednesday, Mar 5th, 2025 at 9am

- You can work in a group of up to **three** students. Read the instructions on the course website for additional details.
- **Submit your solutions electronically on the course Gradescope site as PDF files.** Submit a separate PDF file for each numbered problem. If you plan to typeset your solutions, please use the LaTeX solution template on the course web site. If you must submit scanned handwritten solutions, please use a black pen on blank white paper and a high-quality scanner app (or an actual scanner, not just a phone camera).
- Late submissions will be accepted (for 75% credit) until midnight the day of the deadline.

☞ Some important course policies ☜

- **You may use any source at your disposal**—paper, electronic, or human—but you ***must*** cite ***every*** source that you use, and you ***must*** write everything yourself in your own words. See the academic integrity policies on the course web site for more details including the policy on using AI tools.
- **Avoid the Two Deadly Sins!** Any homework or exam solution that breaks any of the following rules will be given an ***automatic zero***, unless the solution is otherwise perfect. Yes, we really mean it. We're not trying to be scary or petty (Honest!), but we do want to break a few common bad habits that seriously impede mastery of the course material.
  - Always give complete solutions, not just examples.
  - Always declare all your variables, in English. In particular, always describe the specific problem your algorithm is supposed to solve.

See the course web site for more information.

If you have any questions about these policies,
please don't hesitate to ask in class, in office hours, or on Ed.

---

1. A Turing Machine (TM) M decides a language L if on any input string w the machine M halts in an accept state if $w \in L$ and in a reject state if $w \notin L$. In other words M is an algorithm for deciding membership in L. Note that we do not have any upper bound on the running time of M. We say that L is decidable if there is a TM M that decides L. We also called such a language L recursive. The purpose of this problem is to show that decidable languages are closed under basic operations.

   - (2 pts) Show that if L_1, L_2 are decidable then $L_1 \cap L_2$ and $L_1 \cup L_2$ are decidable.
   - (2 pts) Show that if L_1 and L_2 are decidable then $L_1 L_2$ is decidable (concatenation).
   - (2 pts) Show that if L is decidable then L^* is decidable.
   - (4 pts) Now suppose L is recursively enumerable. That is, there is a TM M with the following properties (i) if $w \in L$ then M on input w halts and accepts w (ii) if $w \notin L$, M either halts and rejects w or does not terminate. Show that L^* is recursively enumerable by reading about the technique called *dovetailing*. A useful resource is the following set of notes https://courses.grainger.illinois.edu/cs373/sp2009/lectures/lect_24.pdf.
   - **Not to submit:** Show that if L_1 and L_2 are decidable then $(L_1 \cup L_2)^*$ is decidable.

   For each of the problems you should describe a simple TM that for the given language in a high-level fashion assuming that L_1 has a TM M_1 and L_2 has a TM M_2. One way to think of the problem is to give a program for the given language using sub-routines for L_1 and L_2. No proof is necessary but clarity of your algorithm is important; give a brief description if necessary. Note that in decidability we do not pay attention to the quality of the running time so brute-force algorithms are fine.

2. This problem is on sorting and selection.

   - (4 pts) 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. Use a divide and conquer strategy to merge the sorted arrays in $O(N \log k)$ time where $N = nk$ is the total number of elements in the k arrays.
   - You would like to do some data analysis. Suppose you are given the income of people as an n element unsorted array A, where $A[i]$ gives the income of i.
     - (2 pts) Describe an $O(n)$-time algorithm that given A checks whether the top 2% of earners together make more than ten times the total of the bottom 80%. Assume for simplicity that n is a multiple of 100 and that all numbers in A are distinct. Note that sorting A will easily solve the problem but will take $\Omega(n \log n)$ time.
     - (4 pts) More generally we may want to compute the some additional statistics. Suppose we are given A and k numbers $\alpha_1 < \alpha_2 < \ldots < \alpha_k$ each of which is a number between 0 and 100 and we wish to compute the total earnings of the top $\alpha_i\%$ of earners for $1 \leq i \leq k$. Assume for simplicity that $\alpha_i n$ is an integer for each i. Describe an algorithm for this problem that runs in $O(n \log k)$ time. Note that sorting will allow you to solve the problem in $O(n \log n)$ time but when $k \ll n$, $O(n \log k)$ is faster. An $O(nk)$ time algorithm is relative easy.

3. **Not to submit:** Consider n intervals $I_1, I_2, \ldots, I_n$. Each interval I_i is specified by its two end points a_i and b_i with $a_i \leq b_i$. Two intervals I_i and I_j overlap if there is a number x such that $x \in [a_i, b_i]$ and $x \in [a_j, b_j]$. The overlap length between I_i and I_j is the geometrically natural one — the length of the longest interval shared between I_i and I_j. We can express this overlap length formally as the quantity:

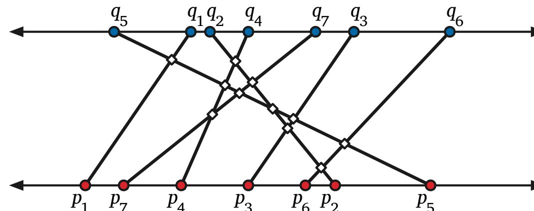
$$\max\{0, \min(b_i, b_j) - \max(a_i, a_j)\}$$

You may want to draw a picture to see the meaning of the formula. Given the n intervals we wish to find the two intervals I_i and I_j that have the maximum overlap length. You can assume that the intervals are specified in two arrays A and B of length n where $A[i] = a_i$ and $B[i] = b_i$. Describe an efficient algorithm for this problem. An $O(n^2)$ algorithm is straight forward. You should aim to beat this easy bound. You may want to first think of the conceptually easier setting where the a_i and b_i values are distinct.

4. **Not to submit:** Given 4 *sorted* arrarys A_1, A_2, A_3, A_4 with a total of n elements, and an index k between 1 and n, describe an $O(\log n)$ time algorithm to find the k'th ranked element in the union of the four arrays.

Solved Problem

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 $P[1 \mathinner{..} n]$ and $Q[1 \mathinner{..} n]$ 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 $P[1 \mathinner{..} n]$ and permuting the array $Q[1 \mathinner{..} n]$ 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) $Q[1 \mathinner{..} \lfloor n/2 \rfloor]$.
- Recursively count inversions in (and sort) $Q[\lfloor n/2 \rfloor + 1 \mathinner{..} n]$.
- 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 $Q[1 \mathinner{..} n/2]$ bLue.
  - Color the elements in the Right half $Q[n/2 + 1 \mathinner{..} n]$ Red.
  - Merge $Q[1 \mathinner{..} n/2]$ and $Q[n/2 + 1 \mathinner{..} n]$, 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]
  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.