CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2024**

Homework #2

Name(s): $FIRST_1$ $LAST_1$, $FIRST_2$ $LAST_2$, $FIRST_3$ $LAST_3$

Homework Policy

- If you leave a question completely blank, you will receive 20% of the grade for that question. This however does not apply to the extra credit questions.
- You may also consult all the materials used in this course (lecture notes, textbook, slides, etc.) while writing your solution, but no other resources are allowed.
- Unless specified otherwise, you may use any algorithm covered in class as a "black box" – for example you can simply write "sort the array in $\Theta(n \log n)$ time using merge sort".
- Remember to always **prove the correctness** of your algorithms and **analyze their running time** (or any other efficiency measure asked in the question). See "Practice Homework" for an example.
- The extra credit problems are generally more challenging than the standard problems you see in this course (including lectures, homeworks, and exams). As a general rule, only attempt to solve these problems if you enjoy them.
- **Groups:** You are allowed to form groups of size *two* or *three* students for solving each homework (you can also opt to do it alone if you prefer). The policy regarding groups is as follows:
  - You can pick different partners for different assignments (e.g., from HW1 to HW2) but for any single assignment (e.g., HW1), you have to use the same partners for all questions.
  - The members of each group only need to write down and submit a single assignment between them, and all of them will receive the same grade.
  - For submissions, only one member of the group submits the full solutions on Canvas and lists the name of their partners in the group. The other members of the group also need to submit a PDF on Canvas that contains only a single line, stating the name of their partner who has submitted the full solution on Canvas (Example: Say A, B, and C are in one group; A submits the whole assignment and writes down the names A, B, and C. B and C only submit a one-page PDF with a single line that says "See the solution of A").
  - You are allowed to discuss the questions with any of your classmates even if they are not in your group. **But each group must write their solutions independently.**

Problem 1. For this problem, we are assuming that arithmetic operations take $O(1)$ time. So addition, subtraction, division, and multiplication, are done in $O(1)$ time no matter how big the numbers are. (This is not exactly true in practice, but it's a reasonable approximation to reality.)

Now consider the following function SQRT(n)

- Input: a positive integer n
- Output: an integer k such that $k^2 = n$, or "No Solution" if none exists

Example: SQRT(9) should output 3, while SQRT(10) should output "no solution"

Write pseudocode for a recursive algorithm that computes SQRT(n) in time $O(\log(n))$. In addition to pseudocode, you should state the recursion formula for your algorithm. So all you need is pseudocode and the recursion formula, nothing else. You do NOT need to prove that this recursion formula solves to $T(n) = O(\log(n))$ **(15 points)**

Problem 2. Suppose we have an array $A[1:n]$ of n *distinct* numbers. For any element $A[i]$, we define the **rank** of $A[i]$, denoted by $rank(A[i])$, as the number of elements in A that are strictly smaller than $A[i]$ plus one; so $rank(A[i])$ is also the correct position of $A[i]$ in the sorted order of A.

Suppose we have an algorithm **magic-pivot** that given any array $B[1:m]$ (for any $m > 0$), returns an index i such that $rank(B[i]) = m/2$ and has worst-case runtime of $O(m)$[1].

Example: if $B = [1, 7, 6, 3, 13, 4, 5, 11]$, then **magic-pivot**$(B)$ will return index 7 as rank of $B[7] = 5$ is 4 which is $m/2$ in this case.

(a) Use **magic-pivot** as a black-box to obtain a deterministic quick-sort algorithm with worst-case running time of $O(n \log n)$. **(10 points)**

(b) Use **magic-pivot** as a black-box to design an algorithm that given the array A and any integer $1 \leq r \leq n$, finds the element in A that has rank r in $O(n)$ time[2]. **(15 points)**

[1]Such an algorithm indeed exists, but its description is rather complicated and not relevant to us in this problem.

[2]Note that an algorithm with runtime $O(n \log n)$ follows immediately from part (a) – sort the array and return the element at position r. The goal however is to obtain an algorithm with runtime $O(n)$.

Problem 3. Given an array A, we say that elements $A[i]$ and $A[j]$ are swapped if $j > i$ but $A[j] < A[i]$.

Example: If $A = [8, 5, 9, 7]$, then there are a total of 3 swapped pairs, namely 8 and 5; 8 and 7; and 9 and 7.

Example:: if $A = 2, 3, 4, 5, 1, 6, 7, 8$ then there are a total of 4 swapped pairs: 1 and 2; 1 and 5; 1 and 3; 1 and 4.

The goal for this problem is to write a recursive algorithm that given an array A determines the number of swapped pairs in the array in $O(n \log(n))$ time. The algorithm to do this is extremely similar to merge sort; in fact, the algorithm does all of merge sort (verbatim), plus a few extra things to keep track of the number of swaps. In particular, one side effect of the algorithm is to sort the array A.

what you need to do: In order to save you the trouble of retyping the pseudocode for merge sort,I have written the pseudocode for most of the algorithm. Your job is simply to fill in two lines. These are the key lines that keep track of the number of swaps.

Note that the algorithm below has an outer function MergeSortAndCountSwaps which I wrote entirely; nothing to fill in. But MergeSortAndCountSwaps then calls MergeAndCountSwapsBetween as a subroutine; in that subroutine you will have to fill in two lines.

The Algorithm MergeSortAndCountSwaps(A)
INPUT: array A with unique elements (no duplicates)
OUTPUT: pair (SortedA, S), where SortedA is the array A sorted in increasing order and S is the total number of swapped pairs in A.

- A1 $\leftarrow$ first half of A. So $A1 = A[0], ..., A[\frac{n}{2} - 1]$
- A2 $\leftarrow$ second half of A. So $A2 = A[\frac{n}{2}], ..., A[n-1]$
- (SortedA1,S_1) $\leftarrow$ MergeSortAndCountSwaps(A1)
- (SortedA2,S_2) $\leftarrow$ MergeSortAndCountSwaps(A2)
- (SortedA, S_3) $\leftarrow$ MergeAndCountSwapsBetween(SortedA1,SortedA2)
- Return (SortedA, $S_1 + S_2 + S_3$).

We now need to define MergeAndCountSwapsBetween(A, B). This is the part where I will have you fill in a few lines. Note that MergeAndCountSwapsBetween(A, B) is very similar to Merge(A,B) from class, and in particular it is NOT a recursive function.

MergeAndCountSwapsBetween(A,B):

1. Initialize array C of size $2n$
2. $i \leftarrow 0$
3. $j \leftarrow 0$
4. $S' \leftarrow 0$ Comment: S' will count swaps; the answer lines below which you have to add will be responsible for changing S'.
5. While (i < n OR j < n)
   (a) if $j = n$ or A[i] < B[j] :
   - **YOUR ANSWER 1 GOES HERE**
   - set next element of C to $A[i]$

- (formally, $C[i+j] \leftarrow A[i]$)
- i $\leftarrow$ i+1

(b) if i = n or A[i] > B[j]:

- **YOUR ANSWER 2 GOES HERE**
- set next element of C to $B[j]$
- j $\leftarrow$ j+1

6. output pair (C, S').

Questions you need to answer:

- What goes in Answer 1? *your answer should be a single line.*
- What goes in Answer 2? *your answer should be a single line.*

NOTE: please don't recopy all of the pseudocode above. **(20 points)**

Problem 4. Write an algorithm in pseudocode:

- Input: An array A with n distinct (non-equal) elements
- Output-1: numbers x and y in A that minimize $|x - y|$, where $|x - y|$ denotes absolute-value(x-y). (If there are multiple closest pairs, you only have to return one of them.)
- Output-2: a pair of numbers w and z in A that minimize $|w + z|$.

The run-time should be significantly better than $O(n^2)$. **(25 points)**

Problem 5. Given an integer array, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum. Give a divide and conquer algorithm that runs much faster than $O(n^2)$. **(20 points)**

Problem 6. a) An **unimodal array** first increases to a peak and then decreases. Design a divide-and-conquer algorithm to find the maximum element of a unimodal array in $O(\log n)$ time. **(10 points)**

b) You are given an array $A[1..n]$ with the following properties:

- $A[1] > A[2]$ (the first element is greater than the second).
- $A[n-1] \leq A[n]$ (the second-to-last element is less than or equal to the last).

A **local minimum** is an element $A[x]$ such that:

$$A[x-1] \geq A[x] \leq A[x+1].$$

For example, in the array $A = [9, 7, 7, 2, 1, 3, 7, 5, 4, 7, 3, 3, 4, 8, 6, 9, 6]$, there are six local minima.

Design an $O(\log n)$-time algorithm to find a local minimum. **(10 points)**

c) An array $A[0..n-1]$ of distinct elements is **bitonic** if there exist unique indices i and j such that:

- $A[i-1] < A[i] > A[i+1]$ (a peak),
- $A[j-1] > A[j] < A[j+1]$ (a valley).

For example:

- $[4, 6, 9, 8, 7, 5, 1, 2, 3]$ is bitonic.
- $[3, 6, 9, 8, 7, 5, 1, 2, 4]$ is not bitonic.

Design an $O(\log n)$-time algorithm to find the smallest element in a bitonic array. **(10 points)**

Problem 7. Given an array A of length n, and two indices i and j with $i \leq j$, define $\text{Prod}(A, i, j)$ as the product of elements from $A[i]$ to $A[j]$, i.e., $A[i] \times A[i+1] \times \cdots \times A[j]$. If $i = j$, then $\text{Prod}(i, j) = A[i]$.

For example, if $A = [3, 0.1, 5, 20, 4]$, then $\text{Prod}(A, 1, 3) = 0.1 \times 5 \times 20 = 10$.

Now, consider the following problem MaxProd(A):

- **Input**: An array A of length n where all elements are positive, but may include fractional values (e.g., $A[i] = 0.1$ is valid).
- **Output**: The maximum possible value of $\text{Prod}(A, i, j)$ for some subarray $A[i..j]$.

For example, if $A = [3, 0.2, 4, 0.5, 1, 4, 0.3, 2]$, the maximum product is 8, which is achieved by the subarray $[4, 0.5, 1, 4]$.

a) Write pseudocode for a recursive algorithm MaxProd(A) that solves the problem in $O(n \log n)$ time. You only need to provide the pseudocode and the recurrence relation for the algorithm. **(10 points)**

Hint: The recurrence formula is similar to one we've used before, so you don't need to prove that it solves to $T(n) = O(n \log n)$.

Note: While there are faster, non-recursive algorithms for this problem, you must use a recursive algorithm for this homework.

b) Now, write pseudocode for a recursive algorithm that solves the problem in $O(n)$ time. As in Part 1, provide the pseudocode and the recurrence relation for the algorithm. **(15 points)**

Hint: Similar to the Max Profit problem covered in class, solve a helper problem MaxProdX(i, j), which gives additional information beyond just the maximum product.

If you're confident with the $O(n)$ algorithm, you can directly write a single solution for both Part 1 and Part 2. However, if you're unsure, first implement the $O(n \log n)$ solution for Part 2, then write the separate $O(n)$ algorithm.

Problem 8. Suppose we have an array $A[1:n]$ which consists of numbers $\{1,\ldots,n\}$ written in some arbitrary order (this means that A is a *permutation* of the set $\{1,\ldots,n\}$). Our goal in this problem is to design a very fast randomized algorithm that can find an index i in this array such that $A[i] \mod 5 = 0$, i.e., $A[i]$ is divisible by five. For simplicity, in the following, we assume that n itself is a multiple of 5 and is at least 5 (so a correct answer always exist).

For instance, if $n = 10$ and the array is $A = [8, 7, 2, 5, 4, 6, 3, 1, 10, 9]$, we can output either of indices 4 or 9.

(a) Suppose we sample an index i from $\{1,\ldots,n\}$ uniformly at random. What is the probability that i is a correct answer, i.e., $A[i] \mod 5 = 0$? **(5 points)**

(b) Suppose we sample m indices from $\{1,\ldots,n\}$ uniformly at random and with repetition. What is the probability that none of these indices is a correct answer? **(5 points)**

Now, consider the following simple algorithm for this problem:

Find-Index-1$(A[1:n])$**:**

- Let $i = 1$. While $A[i] \mod 5 \neq 0$, sample $i \in \{1,\ldots,n\}$ uniformly at random. Output i.

The proof of correctness of this algorithm is straightforward and we skip it in this question.

(c) What is the worst-case **expected** running time of **Find-Index-1**$(A[1:n])$? Remember to prove your answer formally. **(7 points)**

The problem with **Find-Index-1** is that in the worst-case (and not in expectation), it may actually never terminate! For this reason, let us consider a simple modification to this algorithm as follows.

Find-Index-2$(A[1:n])$**:**

- For $j = 1$ to n:
  - Sample $i \in \{1,\ldots,n\}$ uniformly at random and if $A[i] \mod 5 = 0$, output i and terminate; otherwise, continue.
- If the for-loop never terminated, go over the array A one element at a time to find an index i with $A[i] \mod 5 = 0$ and output it as the answer.

Again, we skip the proof of correctness of this algorithm.

(d) What is the **worst-case running time** of **Find-Index-2**$(A[1:n])$? What about its worst-case **expected** running time? Remember to prove your answer formally. **(8 points)**

Problem 9. Given an array A of length n, an element x is *frequent* if it appears at least $n/4$ times. There can be between 0 and 4 frequent elements.

The array A consists of incomparable objects, meaning that you can only check equality (i.e., $A[i] == A[j]$), and there is no notion of ordering or sorting.

Part 1: Write pseudocode for the algorithm `IsFrequent(A, x)` that checks if x is frequent in A. The algorithm should run in $O(n)$ time. You may not use dictionaries or hash functions. **(10 points)**

Hint: This is a short, non-recursive solution.

Part 2 : Write pseudocode for a recursive algorithm that finds all frequent elements in A, or returns "no solution" if none exist. The algorithm should run in $O(n \log n)$ time. Provide the recurrence formula for your algorithm. **(20 points)**

Hint: Reuse `IsFrequent` from Part 1.

Note: You may not use dictionaries, hash functions, or sorting algorithms for this problem. Recursive solutions are required for Part 2.

Problem 10. You are given two strings A and B of length n, each consisting of possibly repeated characters from an alphabet of size M. Two strings A and B are said to be **anagrams** of one another if it is possible to reorder the characters in A to obtain string B (without removing or adding any characters). For instance, the strings "ELEVEN PLUS TWO" and "TWELVE PLUS ONE" are anagrams of one another (think of empty-space also as a character in the alphabet).

Your goal is to design an algorithm that decides if a given pair of strings are anagrams of one another. You may assume that all characters in the alphabet can represented by distinct integers in the set $\{1, 2, \ldots, M\}$, and that you can convert any character to its integer representation in $O(1)$ time (note that M can be much larger than n).

(a) Design an algorithm for this problem with worst-case runtime of $O(n + M)$. **(15 points)**

(b) Design a randomized algorithm for this problem with worst-case expected runtime of $O(n)$. **(10 points)**

Problem 11. Consider the following more complex variant of the Tower of Hanoi puzzle The puzzle has a row of k pegs, numbered from 1 to k. In a single turn, you are allowed to move the smallest disk on peg i to either peg $i-1$ or peg $i+1$, for any index i; as usual, you are not allowed to place a bigger disk on a smaller disk. Your mission is to move a stack of n disks from peg 1 to peg k.

Describe a recursive algorithm for the case $k = n + 1$ that requires at most $O\left(n^2\right)$ moves. **(20 points)**

Challenge Yourself. You are given an array A of non-negative integers. Every element except one has a duplicate. Design an algorithm that finds the element with no duplicate in $O(n)$ time (You are promised that there is exactly one element with no duplicate). **(25 points)**

Example. A couple of examples for this problem:

1. $A = [4, 3, 1, 1, 4]$

   Output: 3

   Explanation: 3 is the only number that does not have a duplicate

2. $A = [3, 1, 4, 5, 1, 4, 3]$

   Output: 5

   Explanation: 5 is the only number that does not have a duplicate