CS-GY 6033: Design and Analysis of Algorithms I

Fall 2024

Instructor: Nairen Cao **NYU Tandon School of Engineering**

Problems	1	2	3	4	5 (ungraded)	6 (Bonus)	Total
Max. Score	20	25	30	25	0	15	100

Table 1: Score Summary

Homework #3

Assigned: 10/11/2024 **Due: 10/24/2024**

General Instructions Write all solutions clearly, concisely, and legibly. It is okay to provide handwritten and scanned/photographed solutions as long as they are clearly legible. If typing, using LaTeX is recommended.

You should generally describe algorithms in words and high-level pseudocode. Specifically, emphasize the goal of each step in your algorithm rather than the coding details. After giving a high-level description, provide more detail on the implementation of each step where appropriate.

The purpose of having these problem sets is to enhance your mathematics and programming skills in the scope of designing algorithms. These problems may be quite challenging, so discussions with classmates are encouraged. However, you should spend at least half an hour thinking about the problems individually before discussing them. Although discussions (and consulting ChatGPT and WolframAlpha) are allowed, you need to write the solution in your own words and by yourself. Please acknowledge any person or reference you discuss with or consult from.

Special Instructions

- I don't want you getting bogged down in the details of corner cases with floors, ceilings, etc. If convenient, you may make reasonable assumptions about n to simplify your algorithm description. Natural assumptions for this sort of divide-and-conquer problem are $n = 2^k$, $n = 2^k - 1$, or $n = 2^k + 1$ depending on what you do. You should explicitly specify the assumption you make, and you should make sure that it's the right one, i.e., that it still holds when you recurse.
- Problem 6 is about the lower bound of the algorithm. If you can solve it, I will give you a bonus of 15 points.
- **Theme for Homework:** Divide and Conquer.

Problem 3-1. Solving Recurrences

For each of the following recurrences, use the master theorem or the recursion tree to give the tight asymptotic upper bound.

(a) $T(n) = T\left(\frac{3n}{4}\right) + n^{3/4}$

(b) $T(n) = 2T\left(\frac{n}{8}\right) + n^{1/3}$

(c) $T(n) = 4T\left(\frac{n}{3}\right) + n \lg n$

(d) $T(n) = 2T(\sqrt{n}) + \lg^2 n$

(e) $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{n}{4}\right) + n$

Problem 3-2. Double median

You are given as input two *sorted* length-n arrays $A[1\,..\,n]$ and $B[1\,..\,n]$, where all elements are distinct. The goal of this problem is to find the median of the union of the arrays.

Example. If $A = \langle 5, 8, 9, 10, 20\rangle$ and $B = \langle 7, 30, 31, 32, 35\rangle$, then the two middle-most values are 10 and 20, so the median is 15.

Since the total number of elements is even, the answer will always be the average of the lower and upper medians.

(a) Describe an efficient divide-and-conquer algorithm for finding the median of two sorted arrays.

The naive algorithm would be to merge the two sorted arrays in $O(n)$ time, producing a size-$2n$ array of all the elements. You could then output the average of the n-th and $(n+1)$th elements. To be considered efficient, your algorithm has be strictly better than $\Theta(n)$.

(b) Analyze the running time of your algorithm.

Analyze means show and explain how you're coming up with the bound. For a divide and conquer problem, that means writing and solving a recurrence. Just stating a bound with no justification is not an analysis.

Problem 3-3. Local Minimum

Suppose we are given an array $A[1 \ldots n]$ of distinct numbers (i.e., no two of the numbers are the same). We say that cell i is a local minimum if its value is strictly smaller than the values of its neighbors, i.e., if $A[i] < A[i-1]$ and $A[i] < A[i+1]$. For example, there are four local minima (circled) in the following array:

2	(1)	15	18	(9)	12	10	8	(7)	13	17	21	19	10	(6)

It is easy to find all local minima in $\Theta(n)$ time by scanning through the array and comparing each cell against its neighbors. If we only need to output a single local minimum, as opposed to all of them, it's possible to do better. This problem asks you to produce sublinear algorithms that do not read the entire input.

(a) Describe an $O(\log n)$-time divide-and-conquer algorithm for returning a local minimum. Justify that your algorithm achieves the desired running time.

Now let's generalize the problem to a 2-dimensional array of numbers, provided as input $A[1 \ldots n, 1 \ldots C]$, so we have an $n \times C$ table. You may again assume that all numbers are distinct. To design your algorithm, we're primarily going to be considering the size-n dimension, so you can think of C as a constant. Nevertheless, part (d) will ask you to state your bound in terms of C.

The definition of being a local minimum in a 2-dimensional table is the same — a cell is a local minimum if its value is smaller than those of the neighboring cells — but now a cell (i, j) has up to 4 neighbors: $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, and $(i, j+1)$.

Here is an example of such a table for $n = 7$ and $C = 4$. All local minima are circled.

(6)	14	11	2
22	(1)	5	53
(15)	27	91	35
36	(18)	31	39
(13)	21	(9)	25
17	24	12	33
19	(10)	16	41

(b) Suppose that you were also provided with an array $M[1 \ldots n]$ where $M[i]$ is the (global) minimum element in the i-th row of A, i.e., $M[i] = \min_{1 \le j \le C}\{A[i, j]\}$. For our example table above, you'd be given the following:

A:

6	14	11	2
22	1	5	53
15	27	91	35
36	18	31	39
13	21	9	25
17	24	12	33
19	10	16	41

M:

2
1
15
18
9
12
10

Given both A and M, describe how to efficiently find a local minimum in A. What is the running time of your algorithm? How many cells of M does your algorithm read?

Hint: this part should be pretty easy given what you've already done. You should be leveraging your previous algorithm.

(c) Now let's drop the assumption that you've been given M. Describe an $O(C \log n)$-time algorithm for finding a local minimum in A.

Note: you cannot afford to precompute M as that would take $\Theta(nC)$ time. Nevertheless, you do want to build off the previous part. Hint: how much of M do you really need, and when do you need it? Your answer here should be short.

(d) (Optional. Not graded) The solution to part (d) only "divided" in one dimension. For this part, suppose we have an $n \times n$ table, i.e., $C = n$. What would happen if you tried dividing in two dimensions? Would the recurrence give a better answer if it worked? Where does the correctness argument break down? Can you figure out how to produce a better divide-and-conquer algorithm that divides in both dimensions? (You're only looking to beat the algorithm from part (d) by a logarithmic factor.)

Problem 3-4. Popular Testing

Widget Co. produces many types of electronic widgets. You've decided to get into the widget business as well, so you'd like to reverse engineer their designs. Since your resources are limited, you'd like to focus your energy on the most popular widget types. The good news is that you located a shipment of n widgets that "fell off a truck." The bad news is that the widgets all look the same, so identifying the popular widget types is going to be a challenge. You don't know up front whether all the widgets are different, or whether there are a few prevailing popular types.

We say that a widget type is **popular** if at least a third of the widgets in the shipment are of that type. You'd like to identify one representative widget from each of the popular types. (There can be anywhere

from 0 to 3 popular types in the shipment.) Of course, this would be impossible without some way of comparing widgets. You have at your disposal an equivalence tester, which takes two widgets as input and determines whether they are of the same type.

Your widgets are provided as input $w_1, w_2, \ldots, w_n$ (or as an array $w[1 \ldots n]$ if you prefer that notation). The equivalence tester is the function `EQ`, where `EQ`(w_i, w_j) returns `TRUE` if w_i and w_j have the same type.

(a) Describe a simple $\Theta(n)$-time iterative algorithm for testing whether a particular candidate widget w_i is popular.

(b) Describe a simple $O(n^2)$-time iterative algorithm for identifying exactly one representative widget from each of the popular types. Note that your algorithm should return at most 3 widgets.

(c) Describe an efficient divide-and-conquer algorithm that returns a representative widget from each of the (up to 3) popular widget types. Analyze the running time of your algorithm.
Hint: Can a widget be popular overall without being popular in at least one subproblem?

Problem 3-5. (Not Graded) Inversions Revisited

Let $A[1 \ldots n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > 2A[j]$, then the pair (i, j) is called an ***significant inversion*** of A.

Give an efficient algorithm that determines the number of significant inversions for a given array A. Analyze the running time of your algorithm.

Problem 3-6. (Bonus) Lower bound on merging sorted lists

The problem of merging two sorted lists arises frequently. We have seen a procedure for it as the subroutine **Merge** used by merge sort. In this problem, we will prove a lower bound of $2n - 1$ on the worst-case number of comparisons required to merge two sorted lists, each containing n items.

First we will show a lower bound of $2n - o(n)$ comparisons by using a decision tree.

(a) Given $2n$ numbers, compute the number of possible ways to divide them into two sorted lists, each with n numbers.

(b) Using your answer to part (a), show that any decision-tree algorithm that correctly merges two sorted lists must perform at least $2n - o(n)$ comparisons.
Note this is a little oh, not a big oh. So you should be showing that the number of comparisons is very close to $2n$, but for a lower-order term that is subtracted off. You can use the formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

Now we will show a slightly tighter $2n - 1$ bound using an entirely different analysis, i.e., not based on a decision tree. This second analysis is actually easier (or at least it involves far less algebra), but it's also very specific to this problem. In contrast, the above decision-tree approach generalizes a bit more readily to other problems.

(c) Argue that if two elements are consecutive in the sorted order and from different lists, then they must be compared.

(d) Use your answer to the previous part to show a lower bound of $2n - 1$ comparisons for merging two sorted lists. *Hint: provide a specific input for which you can leverage the previous part many times.*