End semester Exam — January 4, 2024

This exam has 4 questions for a total of 200 marks, of which you can score at most 125 marks. You may answer any subset of questions or parts of questions. All answers will be evaluated. Go through all the questions once before you start writing your answers. Use a pen to write. Answers written with a pencil will *not* be evaluated. Do *not* overwrite, instead, strike off cleanly and write in a fresh spot.

Do not use hash functions or hash tables/maps in your solutions. All arrays in the questions are zero-indexed. If you wish to use one-indexing in your solutions, please state this explicitly in each such solution. You may split up the pseudocode for an algorithm into two or more functions, if you wish; you don't have to write the entire pseudocode for an algorithm as one function.

You may use the following functions, and assume in your analysis that they work correctly with the stated worst-case running times:

- LENGTH(A): returns the length n of array A, and runs in $\Theta(n)$ time,
- MERGESORT(A): returns the version of integer array A which is sorted in non-decreasing order, and runs in $\Theta(n \log n)$ time when A has n elements.

You may freely invoke functions that you have written as part of a different answer in the same answer sheet. You do *not* have to use loop invariants while proving the correctness of algorithms; but you must correctly explain why each loop (if there are some) does what you expect it to do.

Clearly describe the meaning of any Python (or other) syntax that you use, so that I can correctly evaluate your answer. E.g., if you use the notation A[i : j] to denote a sub-array, clearly explain what you mean by this. And similarly for any other notation.

Unstated assumptions and lack of clarity in solutions can and will be used against you during evaluation. You may freely refer to statements from the lectures in your arguments. You don't need to reprove these unless the question explicitly asks you to, but you must be precise. Please ask the invigilators if you have questions about the questions.

Warning: CMI's academic policy regarding cheating applies to this exam.

1. Recall the problem of efficiently multiplying a chain of matrices that we saw in class:

Least-Cost Matrix-chain Multiplication

Input: An array $D[0 \ldots n]$ of $n+1$ positive integers.

Output: The least cost (total number of arithmetic operations required) for computing the matrix product $A_1 A_2 \cdots A_n$ where each matrix A_i ; $1 \leq i \leq n$ has dimensions $D[i-1] \times D[i]$. Assume that multiplying a $p \times q$ matrix with a $q \times r$ matrix requires pqr arithmetic operations involving two numbers.

(a) Write the *complete* pseudocode for a *recursive* algorithm LCMMREC(D) that solves LEAST COST MATRIX CHAIN MULTIPLICATION. [10]

You will get the credit for this part only if your algorithm is (i) recursive, and (ii) correct.

(b) Prove that your algorithm of part (a) correctly solves the problem. [10]

(c) Write a recurrence for the total *number of times* $T(n)$ that the function LCMMREC gets called, starting with an initial call LCMMREC(D) where D is an array with $n+1$ positive integers. Include the first call LCMMREC(D) in this count. Ensure that you include the base case(s). [10]

(d) Solve your recurrence of part (c) to obtain a tight asymptotic bound, of the form $T(n) = \Theta(f(n))$, for $T(n)$. [10]

(e) Write the *complete* pseudocode for a *non-recursive* algorithm LCMM(D) that solves LEAST COST MATRIX-CHAIN MULTIPLICATION in time *polynomial* in n. [10]

You will get the credit for this part only if your algorithm is (i) *non-recursive*, (ii) correct, and (iii) runs in time polynomial in n.

(f) Prove that your algorithm of part (e) correctly solves the problem. [10]

(g) Prove that your algorithm of part (e) runs in time polynomial in n. [10]

2. Consider the following problems that we discussed in class:

2D Closest Pair

Input: Input: An array $A = [(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})]$ of n points in the 2D plane where all the coordinates are integers.

Output: Two indices $i \neq j$ that minimize the Euclidean distance between (x_i, y_i) and (x_j, y_j).

Repeated Elements

Input: An array $A[0 \ldots (n-1)]$ of n positive integers.

Output: True if at least one number appears at least twice in A, and False otherwise.

Recall that a *comparison-based algorithm* that takes an array A as input (i) can make the call LENGTH(A) to obtain the number of elements in A, and (ii) can otherwise query the input only using pairwise comparisons between elements of A.

(a) Write the *complete* pseudocode for an algorithm HASREPEATS(A) that uses $O(n \log n)$ pairwise comparisons, in the worst case, to solve REPEATED ELEMENTS. [10]

You will get the credit for this part only if your algorithm is (i) correct, and (ii) uses $O(n \log n)$ pairwise comparisons in the worst case.

(b) Use *either* an adversary argument *or* a decision-tree argument to prove that *any* comparison-based sorting algorithm will take $\Omega(n \log n)$ comparisons, in the worst case, to sort an array with n integers. [20]

(c) Prove that *any* comparison-based algorithm that solves REPEATED ELEMENTS, must make $\Omega(n \log n)$ comparisons in the worst case. [20]

Hint: Let X be any algorithm that solves REPEATED ELEMENTS. Suppose we modify X to get an algorithm Y which keeps track of all the answers to the pairwise comparison questions that X makes. Can you use an adversary argument to establish that if X correctly solves REPEATED ELEMENTS then Y can be further modified to get an algorithm Z which can *sort* the input array A *without* further comparison queries?

Note that just answering the question in the hint will *not* get you the credit for this part. You must use your answer to the hint, to construct a *complete* argument that proves the stated lower bound. The hint is provided just to point you towards a solution. (Of course, you could also ignore the hint and provide a completely different solution!)

(d) Prove that any comparison-based algorithm that solves 2D CLOSEST PAIR must make $\Omega(n \log n)$ comparisons in the worst case. [10]

3. Assume that all capacities and flows in this question are integral.

(a) Write the pseudocode for a polynomial-time algorithm MAXFLOWPATH($G = (V, E), c, s, t$) that solves the MAXIMUM-FLOW PATH problem that we discussed in class, and which is defined below. Your algorithm must return the *sequence of vertices* that forms an $s \rightsquigarrow t$ path with the required property. [20]

You must *clearly describe* the various operations of any non-trivial data structure(s) that your pseudocode uses, but you do *not* have to provide the code that implements these operations.

You will get the credit for this part if (i) your algorithm is correct, (ii) it runs in polynomial time, and (iii) you have given clear descriptions of all non-trivial data structure operations that you use in the pseudocode.

The pseudocode for Dijkstra's algorithm is given after the problem definition, for your reference.

Input: A flow network $(G = (V, E), c, s, t)$.
Output: Find an $s \rightarrow t$ path with the maximum flow.

```
DIJKSTRA(G = (V, E), w, s)
    for v ∈ V
        v.dist = ∞
        v.parent = NIL
    s.dist = 0, S = ∅
    Q = an empty min-priority queue.
    for v ∈ V
        Enqueue v in Q, keyed by v.dist
```

```
... CONTINUED
    while Q is not empty
        v = EXTRACT-MIN(Q)
        S = S ∪ {v}
        for each edge v → x
            newdist = v.dist + w(v → x)
            if newdist < x.dist
                x.dist = newdist
                x.parent = v
                DECREASE-KEY(Q, x, x.dist)
```

(b) Suppose you are given a flow network $(G = (V, E), c, s, t)$, together with a function $f : E \rightarrow \mathbb{N}$. Describe, in words or in pseudocode, a polynomial-time algorithm that can figure out if f is a maximum-valued (s, t)-flow in this network. You must clearly describe all non-trivial constructions and steps that are part of the algorithm. [10]

4. Assume that creating an array takes time linear in the size of the array, and accessing/writing an array element takes constant time. Ignore the other costs (such as the cost for comparing two numbers) in the following analysis. For each of parts (b) to (d), assume that we start with an empty data structure, and call Insert() n times.

(a) Write the pseudocode for implementing a dynamic array that triples in size when it runs out of space. Specify how this data structure is initialized, and how Retrieve(i) and Insert(x) work. [10]

(b) What is the worst-case cost of a call to Insert()? What is an upper bound on the worst-case cost of n calls to Insert()? Justify your answers. You will get the credit for this part only if you provide correct justification. [10]

(c) Use aggregate analysis to show that a call to Insert() has constant amortized cost. [10]

(d) Use the *accounting* method to show that a call to Insert() has constant amortized cost. [10]