- This exam has 4 questions for a total of 170 marks, of which you can score at most 100 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.
- Warning: CMI's academic policy regarding cheating applies to this exam.

You do *not* have to use loop invariants for proving the correctness of algorithms; but you must correctly explain why each loop (if there are some) does what you expect it to do. Of course, you *may* use loop invariants if you wish.

The arrays in this question paper are objects whose sizes are fixed, in the sense that the size cannot be changed after the array is created. In particular, these are *not* Python's lists, whose sizes can be changed using, say, `append()`. Also: note that Python's `list.append()` does *not* run in *worst-case* constant time; be mindful of this when writing your pseudocode.

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.

20

1. Recall the SecondBest problem from Quiz 1:

10

SecondBest

- Input: An integer $n \geq 1$ and an array A of n integers. Array A is indexed from 0; its elements are thus $A[0], A[1], \ldots, A[(n-1)]$.
- Output: The *second largest* number x which is present in A, or the special value None if there is no such number in A.

Each part below shows (a Python version of) the pseudocode offered as a solution to this problem by someone among you. And in each case the pseudocode is *wrong*; it produces an incorrect output for certain valid inputs.

For each part, come up with a valid input of the form (n, A) where n is *at most* 5, which the given pseudocode fails to solve correctly. Clearly explain *how and why* the pseudocode fails to correctly solve this input.

You will get the credit for each part only if (i) the input that you specified is valid, (ii) the given pseudocode produces a wrong output for this input, (iii) you have described the actual output that the pseudocode produces when given this input, and, (iv) you have explained the reason why the pseudocode produces this wrong output.

Please make sure that you write the part number correctly, since there is no other way for me to match your answer to the part number in this question.

(a) [10]

```
1   def secondBest(n, A):
2       if n == 1:
3           return None
4       else:
5           x = A[0]
6           y = None
7       for i in range(1,n):
8           if A[i] > x:
9               y = x
10              x = A[i]
11      return y
```

(b) [10]

```
1   def secondBest(n, A):
2       if n == 1:
3           return None
4       largest = A[0]
5       secLargest = None
6       for i in range(1,n):
7           if A[i] < largest:
8               secLargest = A[i]
9       for i in range(1,n):
10          if A[i] > largest:
11              secLargest = largest
12              largest = A[i]
13      return secLargest
```

40 2. Consider the following problem:

Max Pair Product

- Input: An integer $n \geq 2$ and an array A of n non-negative integers. Array A is indexed from 0; its elements are thus $A[0], A[1], \ldots, A[(n-1)]$.
- Output: The *maximum value* of the product of two elements of A that are *at distinct indices* in A. Note that the two elements of A whose product is the required output, need not be distinct as numbers; but they must appear at different *indices* in A.

(a) Write the *complete* pseudocode for an algorithm MaxPairProduct(n, A) that solves the above problem in $\mathcal{O}(n)$ time and uses at most a constant amount of extra space, in the worst case. You will get the credit for this part only if your algorithm is correct and complete, and runs within the required time and space bounds. [15]

(b) Explain why your algorithm of part (a) is correct. [15]

(c) Prove that your algorithm from part (a) runs in $O(n)$ time and constant extra space in the worst case. For this you may assume that each array operation, and each comparison of a pair of numbers, take constant time. Clearly state any other assumptions that you make. [10]

3. Assume for the sake of this question that 5000 candidates attempted Part A of CMI's 2023 MSc DS entrance exam. Consider the following claim and its proof: [10]

10

Claim

All the 5000 candidates who attempted part A of this exam, scored the exact same marks for part A.

Proof

By induction on the size of subsets of candidates who attempted Part A.

- Base case: Take any subset consisting of one candidate. Clearly, all candidates in this subset scored the same marks for part A (Because there is only one candidate in the subset.).
- Inductive assumption: suppose the claim holds for all subsets with at most k candidates.
- Inductive step: Let S be a subset with $k+1$ candidates. Remove an arbitrary candidate x from S to get the subset S' with k candidates. By the inductive assumption, all candidates in the set S' scored the same marks—say m'—for part A. Now remove another arbitrary candidate y ; $y \neq x$ from the *original* subset S to get a subset S'' with k candidates. By the inductive assumption, all candidates in the set S'' scored the same marks—say m''—for part A.

  Since the set S'' contains candidate x, we get that x scored the same marks—namely, m''—as every other candidate in the set $(S'' \setminus \{x\})$. Similarly, the set S' contains candidate y, and so we get that y scored the same marks—namely, m'—as every other candidate in the set $(S' \setminus \{y\})$.

  But notice that $(S'' \setminus \{x\}) = (S' \setminus \{y\})$. So we get that m' is in fact equal to m'', and that all the candidates in set S scored the same marks for part A.

Hence we get, by induction, that all the candidates scored the same marks.

It is obvious (I hope ...) that the claim cannot possibly be correct. Clearly explain what is wrong with the above proof.

100

4. Recall that a palindrome is a string that reads the same in either direction. A *non-trivial palindrome* is a palindrome with length (number of characters) at least two. Consider the

following problem:

Palindrome Sequence

- Input: An integer $n \geq 2$ and a string S of length n. String S is an array indexed from 0; its elements are thus $S[0], S[1], \ldots, S[(n-1)]$.
- Output: True if S can be obtained by concatenating one or more non-trivial palindromes, and False otherwise. Equivalently: True if S can be partitioned (that is: cut up, without dropping any element) into one or more non-trivial palindromes, and False otherwise.

Some examples with $n = 10$:

- True instances: `nnllknkoyo`, `fjbubjfhuh`, `mttmzizzcz`, `fjfyspsyqq`, `abcdeedcba`
- False instances: `cmmcmhdaba`, `azatznnzth`, `ummnurlxmv`, `xqeppajynx`, `tyjglnvmaa`

10 (a) Write the *complete* pseudocode for a function IsNTP(w) that returns True if string w is a non-trivial palindrome and False otherwise, and runs in *linear time* in the length of w in the worst case. You will get the credit for this part only if your algorithm is correct and runs within the required worst-case time bound. [10]

20 (b) Write the *complete* pseudocode for a *recursive* function IsNTPSequence(n, S) that solves Palindrome Sequence. You may use the function IsNTP() that is described in part (a) as a black box, even if you have not solved part (a). You will get the credit for this part only if your algorithm is (i) correct, and (ii) recursive. In particular, make sure that you have correctly handled all the base cases. [20]

10 (c) Explain why your algorithm of part (b) correctly solves Palindrome Sequence. You may assume that the function IsNTP() works correctly. [10]

10 (d) Write a recurrence for the *number of recursive calls* that your algorithm from part (b) makes, in terms of n. Explain why your recurrence correctly captures this number. [10]

(e) Solve your recurrence of part (d) to get an upper bound on the number of recursive calls that your algorithm makes, in terms of n. [10]

(f) Write the *complete* pseudocode for a *memoized version* of your algorithm from part (b). As in part (b), you may use IsNTP() as a black box even if you haven't solved part (a). You will get the credit for this part only if your pseudocode is a correctly memoized version of your *correct* algorithm from part (b). Make sure that you have correctly handled all the sentinel values/base cases. [20]

(g) Show that your memoized algorithm of part (f) runs in $\mathcal{O}(n^c)$ worst-case time for input strings of length n, for some fixed constant c. What is the value of c that you get? [20]