- This exam has 3 questions for a total of 100 marks.
- Warning: CMI's academic policy regarding cheating applies to this exam.

All arrays in this question paper have their indices starting at 0. If you wish to use 1-based indexing, you must clearly state this in each such answer. The arrays in these questions 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, $\mathtt{append}()$. Also: note that Python's $\mathtt{list.append}()$ 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. Please ask the invigilators if you have questions about the questions.

✓ 1. The input to this problem is an array A of length $n \geq 1$. Each element of array A is a pair of the form $(\mathtt{StudentID}, \mathtt{marks})$ where $\mathtt{StudentID}$ is an alphanumeric string, and $\mathtt{marks}$ is a non-negative integer. The pair $(\mathtt{StudentID}, \mathtt{marks})$ being present in A means that the student whose ID is $\mathtt{StudentID}$, got $\mathtt{marks}$ marks in some exam. There may be more than one pair with the same $\mathtt{StudentID}$; each such pair denotes the marks that this particular student got in a different exam. All pairs with the same $\mathtt{StudentID}$ occur *consecutively* in the array. The goal is to find the *maximum* marks that each student scored, among all the exams whose marks for this student are present in A. [30]

Write the *complete pseudocode* for an algorithm $\textsc{MaxMarks}(n, A)$ that **prints out** the highest marks corresponding to each $\mathtt{StudentID}$ that is present in array A. For each distinct $\mathtt{StudentID}$ present in array A, the output should have *exactly one line* that lists this $\mathtt{StudentID}$ and the corresponding maximum marks. The algorithm must run in $\mathcal{O}(n)$ time and take at most *constant* extra space, in the worst case. Assume that two $\mathtt{StudentID}$s can be compared for equality in constant time, using the $==$ operator. Assume also that you can use

- $A[i][0]$ to access the $\mathtt{StudentID}$ in location i of array A in constant time,
- $A[i][1]$ to access the $\mathtt{marks}$ in location i of array A in constant time, and
- A Python-like function $\mathtt{print}()$ that prints out its arguments, and then a newline, in constant time. You may use Python-like—or any other sensible—string interpolation to include values of variables in the printed-out string.

You will get the credit for this question *only if* your solution *correctly* solves *all* valid instances of the stated problem *within the required time and space bounds.*

You do *not* have to explain why your pseudocode is correct. You do *not* have to provide an analysis of its running time and space.

✓ 2. The input to this problem consists of an array A of $n \geq 1$ integers, and an integer x. Array A is *sorted* in non-decreasing order. The goal is to find the *number of times* that integer x appears in A. Note that this number could be zero. [30]

Write the *complete pseudocode* for an algorithm $\textsc{Counter}(A, x)$ that returns the number of times that x appears in A. The algorithm must run in $\mathcal{O}(\log_2 n)$ time in the worst case. Assume that two numbers can be compared using the operators $\{<, >, ==\}$ in constant time. Assume also that you can use $A[i]$ to access the number in location i of array A, in constant time.

You will get the credit for this question *only if* your solution *correctly* solves *all* valid instances of the stated problem *within the required time bound.*

You do *not* have to explain why your pseudocode is correct. You do *not* have to provide an analysis of its running time.

3. 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: **[40]**

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, and without rearranging the pieces) into one or more non-trivial palindromes, and False otherwise.

Some examples with $n = 10$:

- True instances: `abacabaddd`, `fjbubjfhuh`, `mttmzizzcz`, `fjfyspsyqq`, `abcdeedcba`
- False instances: `daabbbcccd`, `azatznnzth`, `ummnurlxmv`, `xqeppajynx`, `tyjglnvmaa`

Write the *complete* pseudocode for a *non-recursive* algorithm IsNTPSequence(n, S) that solves Palindrome Sequence using *dynamic programming*. The algorithm should have a worst-case running time of $\mathcal{O}(n^c)$ for some small constant c.

You will get the credit for this question *only if* your pseudocode is *complete*, and implements a *non-recursive DP algorithm* which *correctly* solves *all* valid instances of the problem *within the required time bound.*

Make sure that you correctly initialize your DP table, that you check for sentinel values wherever required, and that you always compute and store the value in any cell of the table *before* you access the value in that cell for further computation.

You do *not* have to explain why your pseudocode is correct. You do *not* have to provide an analysis of its running time.