THE UNIVERSITY OF MELBOURNE

SCHOOL OF COMPUTING AND INFORMATION SYSTEMS

COMP90038 Algorithms and Complexity

Semester 1 Assessment 2020

Writing Time 3 hours.

Marks Available: 70 marks.

Instructions to Students:

The exam consists of two parts:

- **Part A**: 3 questions worth 30 marks
    - You must prepare answers to these questions in a PDF document. Please use a text editor. Handwritten answers are **NOT** acceptable.
    - Submit your solutions using the Modules→Final Exam→Part A submission link within the allocated 3-hour time window.
    - Your algorithms must be written in pseudocode – using a format consistent with the algorithms introduced in the lectures – or plain English where appropriate. Python and/or Java code is **NOT** acceptable.
    - Plagiarism detection software will be used to check all submitted solutions. If you include pseudocode from any internet site, your solution may be penalised.
    - Full marks will be given if your algorithm is correct, with appropriate time complexity and is unambiguous – the examiner/marker should not have to 'second guess' what you mean.
    - Solutions that somehow arrive at what appears to be a correct solution with obvious mistakes and/or ambiguous lines of pseudocode will be penalised.
- **Part B**: 26 quiz questions worth 40 marks
    - The quiz questions can be accessed via the Modules→Final Exam→Part B
    - You must complete the quiz questions in the allocated 3-hour time window

Part A

Question 1 (10 marks).

Design a **recursive algorithm** to evaluate arithmetic expressions that consist of integer numbers, opening and closing parentheses and the operators +, -, * and /. Here / is integer division.

A well-defined arithmetic expression is defined as:

- a digit $d \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is a well-formed arithmetic expression.
- If X and Y are well-formed arithmetic expressions, then the four expressions (X + Y), (X - Y), (X * Y) and (X / Y) are also well-formed arithmetic expressions.

Examples of well-formed arithmetic expressions are “((8+7)*3)”, “(4-(9-2))” and “0”. However, the expressions “3+) 2 (())”, “(9 + ())”, “-4”, “5-8”, “108” or “(8)” are not well-formed arithmetic expressions.

Your algorithm should check whether an input string is a well-formed arithmetic expression. If it is not well-formed then your algorithm must return a constant NOTWELLFORMED, otherwise it must evaluate the expression and return the integer result. Your algorithm does not have to deal with division by 0.

You may assume that the input string is an array or list of n characters. You should make sure that your pseudocode matches any assumptions that you make.

Your algorithm must be written in unambiguous pseudocode using a style consistent with the pseudocode introduced on the lecture slides. Python and/or Java code are **NOT** acceptable for this question.

Part A

Question 2 (10 marks).

Consider the problem of processing a sequence of real numbers $X = (x_1, \ldots, x_n)$ of length n.

You are required to design an **augmented data structure** that provides (implements) two $O(log\ n)$ time operations:

> INJECT(y, i): inserts item y between x_i and x_{i+1} in the data structure
>
> SUM(i, j): computes the sum $\sum_{t=i}^{j} x_t$ items in the data structure

Your tasks:

a. Briefly describe the augmented data structure – based only on a data structure discussed in the COMP90038 lectures – that meets the criteria above.

   You may include a labelled diagram to help illustrate your answer.

b. Explain what happens when the INJECT() operation is called.

   You do not have to write an algorithm in pseudocode to answer part (b) of this question. We are expecting that you write a couple of sentences or a short list of bullet points describing the important steps of the INJECT() function.

c. Explain what happens when the SUM() operation is called.

   You do not have to write an algorithm in pseudocode to answer part (c) of this question. We are expecting that you write a couple of sentences or a short list of bullet points describing the important steps of the SUM() function.

Part A

Question 3 (10 marks).

You are part of a task force put together to trace the evolution of SARS-CoV-2, the virus causing the COVID-19 pandemic. The main goal of your team is to understand the differences between thousands of genomes (represented by large character strings) of strains of the virus collected worldwide to better guide the development of a new drug.

As a COMP90038 student, you realise that a string processing algorithm can be used to help analyse viral evolution. That is, an algorithm can be used to compare and quantify differences between genomes that are represented as a long string of four characters: `A`,`C`,`G` & `U`.

Over the course of genome evolution, the long character strings change via multiple *mutations*, which are modifications to the the characters in a given string accumulated over time. Each of these modifications may include one of the following operations (or mutations):

- Substitution of a character with a different character (eg., `A` is replaced by a `U`).
- Insertion or deletion of one character (eg., given the string `GAUCG` the random deletion of the character `U` results in a new string `GACG`). Thus, an insertion or deletion results in variable length genomes.

In order to compare genomes, your challenge is to develop an algorithm to **superimpose** any two genomes in a way that maximises a similarity score based on simulating mutations (or the types of modifications described above). The algorithm uses a 2D-superimposition matrix `S` to record the similarity scores. That is, an entry in the superimposition matrix `S(i,j)`, denotes the optimal superimposition between the `i` first characters of one genome and the `j` first characters of the other.

When maximising `S(i,j)`, you must consider the following possibilities and corresponding scores:

- *Matches* or *Mismatches*
  - If the corresponding characters from each of the strings match, a score of **+1** is attributed. This score means that the genome characters did not change during evolution.
  - If the corresponding characters from each of the strings do not match, a score of **-1** is attributed. This score means that the at least one of the genome characters was substituted (mutated) during evolution.
- *Insertions* or *deletions*
  - Your algorithm can also simulate an insertion or deletion on one of the genomes, with a score of **-2** attributed.

Two examples are provided to illustrate the superimposing process and the corresponding similarity score.

Example 1: providing the strings `CAAGACG` and `AAGAACG` as input, your algorithm should identify the following superposition(by tracing back the `S` matrix) to maximise the score.

```
C A A G A - C G
- A A G A A C G
```

Here, there were 2 insertion/deletions represented by the `-` character (scoring -4) and 6 matches (scoring 6) resulting in an optimal total score of 2.

Example 2: providing the strings `AACTGA` and `ACTGT` as input, your algorithm should identify the following superposition (by tracing back the `S` matrix) to maximise the score.

```
A A C T G A
A - C T G T
```

Here, there was 1 insertion/deletion (scoring -2), 1 mismatch (scoring -1) and 4 matches (scoring 4), resulting in an optimal total score of 1.

Design your algorithm using **dynamic programming**. Assuming that the input to your algorithm are two valid character strings (genomes), your algorithm should fill the entries in the matrix `S(i,j)` and return the optimal score based on the superimposing process describe above.

Your algorithm must be written in unambiguous pseudocode using a style consistent with the pseudocode introduced on the lecture slides. Python and/or Java code are **NOT** acceptable for this question.

Reminder: Have you submitted your answers to the Quiz questions?