CANDIDATE NUMBER:

[TO FILL]

Version: IB/1

EGT3
ENGINEERING TRIPOS PART IIB

Wednesday 8 February 2023 11.20 to 12.10

Module 4M26

ALGORITHMS AND DATA STRUCTURES

Answer one question.

The ***approximate*** *percentage of marks allocated to each part of a question is indicated in the right margin.*

Write your candidate number ***not*** *your name on the cover sheet and at the top of each answer sheet.*

The runtime of a successful test case should be no longer than 10 seconds when run on a standard DPO machine.

No access to internet or other resources is permitted.

Solutions should not use and import any python libraries. Only default Python data structures are to be used.

Make sure to verify that you are still getting correct answers after kernel restart (Kernel->Restart & Run All), before uploading your final version of the notebook to Moodle.

STATIONERY REQUIREMENTS
Single-sided script paper

SPECIAL REQUIREMENTS TO BE SUPPLIED FOR THIS EXAM
DPO computers
CUED approved calculator allowed
Engineering Data Book

10 minutes reading time is allowed for this paper at the start of the test.

You may not start to read the questions printed on the subsequent pages of this question paper until instructed to do so.

You may not remove any stationery from the Examination Room.

1.

(a) Given an integer array, A, write the function, **maximum_subarray**(A), to find the continuous subarray with the largest sum, and return its sum. Note that continuous subarray is any array derived from the original array by removing all elements except some continuous sequence of elements of the original array. Your solution should have its run time complexity strictly smaller than $O(n^2)$.

[30%]

Examples:

Input: `[-5,14,-5,-8,7,7,-20]`
Output: `15`
Explanation: Subarray `[14,-5,-8,7,7]` has the largest sum.

Input: `[1,2,3,4,5,6,7,-1]`
Output: `28`
Explanation: Subarray `1,2,3,4,5,6,7` has the largest sum.

Constraints:

- $1 \leq len(A) \leq 1000$
- $1 \leq |A[i]| \leq 100000$

Code:

In [ ]:
```
def maximum_subarray(A):

    #Write your code here
```

Tests:

Run example test case 1:

In [ ]:
```
input_value = [-5,14,-5,-8,7,7,-20]
print (maximum_subarray(input_value))
```

Run example test case 2:

In [ ]:
```
input_value = [1,2,3,4,5,6,7,-1]
print (maximum_subarray(input_value))
```

Automatic Evaluation:

Do not forget to run on all test cases when your final implementation is finished!

```
In [ ]:
#DO NOT EDIT THIS CODE!
from evaluation_script import evaluate_solution
evaluate_solution(question_id=1,question_part_id='a',function=maximum_subarray,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

(b) Sketch a detailed proof of correctness of your algorithm described in Part (a) and derive its runtime complexity.

[10%]

Write your answer here.

(c) Briefly answer multiple questions below.

(i) Derive big-Theta or big-O expressions used to characterise the following function:

$$f(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2f(\frac{n}{2}) + 3n\log_2 n + n & \text{otherwise.} \end{cases}$$

[10%]

Write your answer here.

(ii) What is a *loop invariant*? Give a concrete example of its use.

[10%]

Write your answer here.

(iii) Explain, in general terms, the main differences and similarities between the divide-and-conquer technique and dynamic programming.

[10%]

Write your answer here.

(d) Write the function, **distance**(A), which takes a list, A, as an input which contains two strings: a and b. This function outputs:

(1) the minimum number of edit operations required to make the string, b, match the string, a, as well as;

(2) an example of such a sequence of operations.

Edit operations, are executed in a sequence to build the string, a, from characters of string, b, starting from location index, $i = 0$. They are:

(i) $insert(<\mathrm{c}>)$ - inserting the character, $<\mathrm{c}>$, without affecting the location index, i, of the next symbol to be used from b;

(ii) $delete(<\mathrm{c}>)$ - deleting the character, $<\mathrm{c}>$ at the location, i, of b (ie. $b[i] = <\mathrm{c}>$) and incrementing the location index (ie. $i = i + 1$);

(iii) $replace(<\mathrm{c}>, <\mathrm{d}>)$ - replacing the character, $<\mathrm{c}>$ at location, i, of string, b (ie. $b[i] = <\mathrm{c}>$) with character, $<\mathrm{d}>$, and incrementing the location index, i.

There is also a non-edit operation, $use(<\mathrm{c}>)$, which uses the character, $<\mathrm{c}>$, of string, b (ie. $b[i] = <\mathrm{c}>$), and increments the location index. This operation is not counted towards the total number of edit operations.

[30%]

Examples:

Input: `["stand","stamps"]`
Output: `[3, ['use(s)', 'use(t)', 'use(a)', 'replace(m,n)', 'replace(p,d)', 'delete(s)']]`
Explanation:

- Location index is 0. We use character 's' and increment location index to 1.
- We use character 't' and increment location index to 2.
- We use character 'a' and increment location index to 3.
- Character, 'm', is at location 3 of string, b, so we replace it with character 'n' and increment location index to 4.
- Character, 'p', is at location 4 of string, b, so we replace it with character 'd' and increment location index to 5.
- We delete character, 's', at location 5.

The resulting string is 's'+'t'+'a'+'n'+'d' = 'stand'. Total number of edit operations: 3.

Input: `["asleep","steep"]`
Output: `[2, ['insert(a)', 'use(s)', 'replace(t,l)', 'use(e)', 'use(e)', 'use(p)']]`
Explanation:

- Location index is 0. We insert character 'a' and keep location index at 0.
- We use character 's' and increment location index to 1.
- Character 't' is at location 1 of string b so we replace it with character 'l' and increment location index to 2.
- We use character 'e' and increment location index to 3.
- We use character 'e' and increment location index to 4.
- We use character 'p' and increment location index to 5.

The resulting string is 'a'+'s'+'l'+'e'+'e'+'p' = 'asleep'. Total number of edit operations: 2.

Constraints:

- $1 \leq \mathrm{len}(a), \mathrm{len}(b) \leq 100.$

Code:

In [ ]:

```python
def distance(A):

    #Write your code here
```

Tests:

Run example test case 1:

In [ ]:

```
input_value = ["stand","stamps"]
print(distance(input_value))
```

Run example test case 2:

In [ ]:

```
input_value = ["asleep","steep"]
print(distance(input_value))
```

Automatic Evaluation:

Do not forget to run on all test cases when your final implementation is finished!

In [ ]:

```
#DO NOT EDIT THIS CODE!
from evaluation_script import evaluate_solution, valid_solution_string_edit
evaluate_solution(question_id=1,question_part_id='d',function=distance,
                  comparison_function=valid_solution_string_edit,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

END OF PAPER