CANDIDATE NUMBER:

[TO FILL]

Version: IB/1

EGT3
ENGINEERING TRIPOS PART IIB

Thursday 16 March 2023 4 to 6.30

Module 4M26

ALGORITHMS AND DATA STRUCTURES

*Answer not more than **three** questions.*

All questions carry the same number of marks.

*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.*

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.

STATIONERY REQUIREMENTS
Single-sided script paper

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

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) Let $S = \{a_1, a_2, \cdots, a_n\}$ be a set of n activities that wish to use a resource (e.g. a lecture hall), which can serve only one activity at a time. The set of activities satisfies the following properties:

(i) each activity, a_i, has a start time, s_i, and a finish time, f_i;
(ii) if selected, activity, a_i, takes place during the half-open time interval $[s_i, f_i)$;
(iii) activities, a_i and a_j, are compatible if the intervals, $[s_i, f_i)$ and $[s_j, f_j)$, do not overlap.

Write the function, **activity_selection**(A), which takes in a list of lists, A, such that $A[i][0] = s_i$, $A[i][1] = f_i$, as an input and outputs the size of the maximum subset of mutually compatible activities.

Your solution should have $\Theta(n \log n)$ run time.

[30%]

Examples:

Input: `[[1,3],[1,8],[2,6],[7,10]]`
Output: 2

Input: `[[0,11],[2,6],[4,7],[5,10],[7,11],[10,13],[12,14]]`
Output: 3

Constraints:

- $1 \leq n \leq 1000$.
- $0 \leq s_i < f_i \leq 10^6$.

Code:

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

    # Insert code here
```

Tests:

Run example test case 1:

In [ ]:
```
input_value = [[1,3],[1,8],[2,6],[7,10]]
print (activity_selection(input_value))
```

Run example test case 2:

In [ ]:
```
input_value = [[0,11],[2,6],[4,7],[5,10],[7,11],[10,13],[12,14]]
print (activity_selection(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=activity_selection,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

(b) Explain why your algorithm described in Part (a) is correct. What is its run time complexity?

[15%]

Write your answer here.

(c) Give brief answers to the following questions.

(i) A set of items are held in a linked list. The list is (already) sorted, and it contains n items. What are the best, worst and average costs of looking up an item that is already present in the list, and what are the costs of verifying that an item is not present? In case of average cost analysis, state your assumptions made about the nature of inputs to your look up procedure.

[10%]

Write your answer here.

(ii) Show that the solution to the recurrence $f(n) = \begin{cases} 1 & n \leq 2, \\ 2f(\lfloor \frac{n}{2} \rfloor + 1) + n & n > 2 \end{cases}$ is asymptotically bounded by $O(n \log n)$.

[10%]

Write your answer here.

(iii) Explain the differences (if any) in the run time performance and/or memory requirements between a recursive solution with memoization and a dynamic programming solution. In your answer use an example of a concrete algorithm.

[10%]

Write your answer here.

(d) Write the function **lcs**(L), which takes a list, L, as an input. This list contains three strings. The function should output the longest common sub-sequence string which is present in all three given strings. If no common subsequence exist, an empty string should be returned.

Note, a subsequence string is a string that can be derived from another string by deleting some or no elements without changing the order of the remaining elements.

[25%]

Examples:

Input: `["le2ap","3le2apto","l1eaptoleap"]`
Output: `"leap"`

Input: `["xyzk1e2","xc17ef","xx1ex"]`
Output: `"x1e"`

Constraints:

- $1 \leq \text{len}(L[i]) \leq 100$ for $i \in \{0, 1, 2\}$.

Code:

In [ ]:
```
def lcs(L):

    # Insert code here
```

Tests:

Run example test case 1:

In [ ]:
```
input_value = ["le2ap","3le2apto","l1eaptoleap"]
print (lcs(input_value))
```

Run example test case 2:

In [ ]:
```
input_value = ["xyzk1e2","xc17ef","xx1ex"]
print (lcs(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='d',function=lcs,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

2.

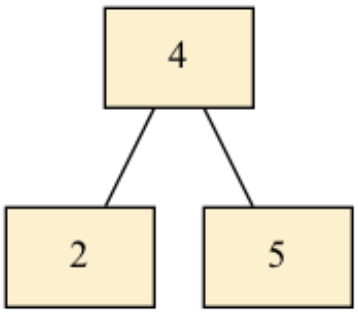
A Binary Search Tree (BST) is a general-purpose data structure for storing keys. In the question below, you may assume that all keys stored in a Binary Search Tree will be unique integers.

(a) Implement a function to construct a Binary Search Tree from a given list of keys, $keys$, by completing the **construct_tree**($keys$) function below. Your function should make use of the `Node` class provided. It should return a list of the keys in the resulting Binary Search Tree in the order visited by a postorder traversal.

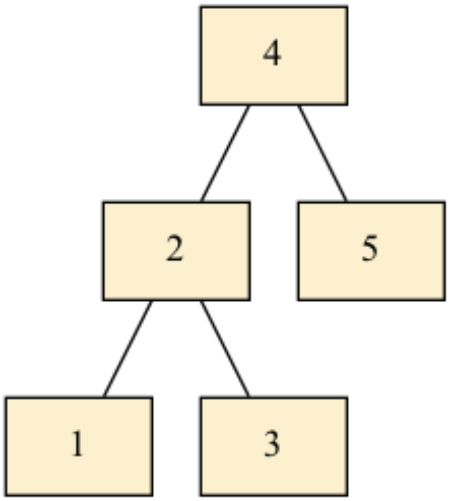
[30%]

Examples:

Input: `[4, 2, 5]`
Output: `[2, 5, 4]`
Explanation: the tree structure is visualised below



Input: `[4, 2, 5, 1, 3]`
Output: `[1, 3, 2, 5, 4]`
Explanation: the tree structure is visualised below



Constraints:

- Every key is a unique integer.
- $0 \leq \text{len}(keys) \leq 10^3$.

Code:

In [ ]:
```
class Node:

    def __init__(self, key, left=None, right=None):
        self.key = key
        self.left = left
        self.right = right

def construct_tree(keys):

    # Insert code here
```

Tests:

Run example test case 1:

In [ ]:
```
keys = [4, 2, 5]
print(construct_tree(keys))
```

Run example test case 2:

In [ ]:
```
keys = [4, 2, 5, 1, 3]
print(construct_tree(keys))
```

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=2,question_part_id='a',function=construct_tree,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

(b) Describe the asymptotic complexity of the operation implemented in Part (a) under average case and worst case assumptions. Provide an example of an input sequence of keys that induces worst case behaviour.

[15%]

Write your answer here.

(c) Give brief answers to the following questions.

(i) Describe the *Binary Search Tree property*.

[5%]

Write your answer here.

(ii) Describe the operation of *preorder* and *inorder* traversals and show how each can be implemented with recursion (either with a written description, pseudocode or Python code). What is the asymptotic runtime complexity for both operations?

[10%]

Write your answer here.

(iii) Explain how the inorder successor of a node is found in a Binary Search Tree. What is the average case and worst case asymptotic complexity of this operation?

[10%]

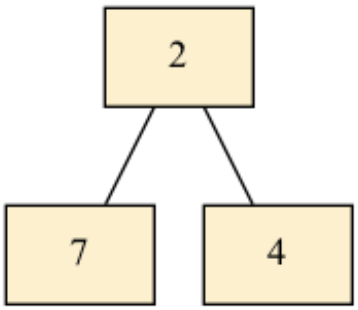
Write your answer here.

(d) Given as input a dictionary, $traversals$, containing two lists of keys of the form `{"preorder": <preorder_keys>, "inorder": <inorder_keys>"}` where `<preorder_keys>` is a list containing the keys produced by a preorder traversal of a Binary Tree and `<inorder_keys>` is the inorder traversal of the same tree, implement a function to construct the Binary Tree. Note that this tree will *not* necessarily be a Binary Search Tree. Your implementation should complete the **construct_tree_from_traversals**$(traversals)$ function below and should return the keys in postorder traversal order for the constructed Binary Tree.

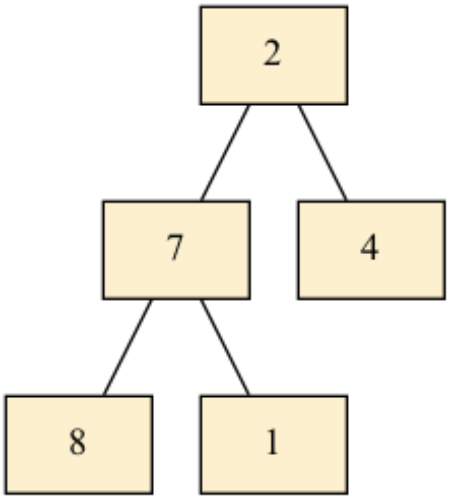
[30%]

Examples:

Input: `{"preorder": [2, 7, 4], "inorder": [7, 2, 4]}`
Output: `[7, 4, 2]`
Explanation: the tree structure is visualised below



Input: `{"preorder": [2, 7, 8, 1, 4], "inorder": [8, 7, 1, 2, 4]}`
Output: `[8, 1, 7, 4, 2]`
Explanation: the tree structure is visualised below



In [ ]:
```
class Node:

    def __init__(self, key, left=None, right=None):
        self.key = key
        self.left = left
        self.right = right


def construct_tree_from_traversals(traversals):

    # Insert code here
```

Tests:

Run example test case 1:

In [ ]:
```
traversals = {"preorder": [2, 7, 4], "inorder": [7, 2, 4]}
print(construct_tree_from_traversals(traversals))
```

Run example test case 2:

In [ ]:
```
traversals = {"preorder": [2, 7, 8, 1, 4], "inorder": [8, 7, 1, 2, 4]}
print(construct_tree_from_traversals(traversals))
```

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=2,question_part_id='d',function=construct_tree_from_tra
                  test_case_list=[1,2,3,4,5],verbose=True)
```

3.

(a) Write the function, **bfs_distances**(L), which takes a list, L, as an input. This list contains three elements. The first element is a number, n, of vertices in a graph. The second element is an adjacency list, A, which represents an undirected graph. The third element is a single value, s, which represents the starting vertex. The function should perform the Breadth-First Search traversal of the graph, starting from vertex, s.

The output of the **bfs_distances** function should be a list, O, of n elements. Element $O[i]$ is a list itself which contains vertices (in any order) that are at the distance i away from the starting vertex, s.

Note that the distance from the starting vertex, s, to itself is asumed to be 0.

Also note that in this adjacency list representation we assume that graph vertices are numbered from 0 to $n-1$ and the neighbours of the vertex, u, are stored as a list, $A[u]$. All edges, (u, v), in the graph have two corresponding entries in the adjacency list (ie. u being a neighbour of v and v being a neighbour of u).

[35%]

Examples:

Input: `[7, [[1, 2], [0, 2, 3, 4], [0, 1, 5, 6], [1], [1, 5], [2, 4], [2]], 0]`
Output: `[[0], [1, 2], [3, 4, 5, 6], [], [], [], []]`

Input: `[10,[[3,4,8,9],[2,9],[1,4,5,6],[0,7],[0,2],[2,7],[2],[3,5],[0],[0,1]],9]`
Output: `[[9], [0, 1], [3, 4, 8, 2], [7, 5, 6], [], [], [], [], [], []]`

Constraints:

- $1 \leq n \leq 100$.

Code:

In [ ]:
```
def bfs_distances(L):

    # Insert code here
```

Tests:

Run example test case 1:

In [ ]:
```
input_value = [7, [[1, 2], [0, 2, 3, 4], [0, 1, 5, 6], [1], [1, 5], [2, 4], [2]], 0]
print (bfs_distances(input_value))
```

Run example test case 2:

In [ ]:
```
input_value = [10,[[3,4,8,9],[2,9],[1,4,5,6],[0,7],[0,2],[2,7],[2],[3,5],[0],[0,1]],9
print (bfs_distances(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, bfs_equivalent
evaluate_solution(question_id=3,question_part_id='a',function=bfs_distances,compariso
                  test_case_list=[1,2,3,4,5],verbose=True)
```

(b) Give brief answers to the following questions.

(i) Explain why your algorithm described in Part (a) is correct. Derive its run time complexity in detail.

[10%]

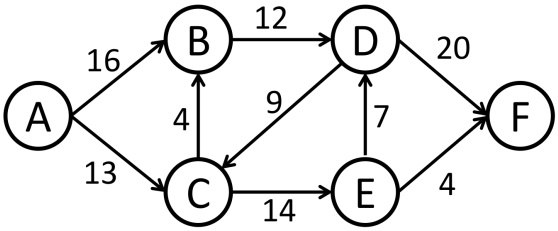
Write your answer here.

(ii) Why is it said that Floyd-Warshall algorithm for all-pairs shortest path computation is a dynamic programming algorithm? What makes Dijkstra's algorithm a greedy algorithm? Justify your answers.

[10%]

Write your answer here.

(c) Answer questions about the graph illustrated below.



(i) Calculate the maximum flow from vertex A to vertex F. Provide the detailed steps of your computation.

[5%]

Write your answer here.

(ii) Perform a topological sort of this graph. Any valid ordering of vertices can be provided.

[5%]

Write your answer here.

(iii) Assume this graph is transformed in to an undirected graph by replacing all directed edges with equivalent undirected edges with corresponding weights. Write out the edges of the minimum spanning tree (MST) discovered by Kruskal's algorithm in the same order as they would be seleted to form the MST.

[5%]

Write your answer here.

(d) Write the function, **strongly_connected**(L), which takes in a list, L, of 2 elements as an input. The first element, $|V|$, is the total number of vertices, V, in a directed graph. The second element, A, is a list where each element $A[u]$ in this list, is itself a list of vertex indexes. Element, $A[u][i]$, represents a directed edge, $(u, A[u][i])$, in the graph. Note that vertices in the graph are represented as integer numbers between 0 and $|V| - 1$.

Your function should output the number of strongly connected components in the graph. Run time complexity of your solution should be $O(|V| + |E|)$, where $|E|$ is the number of edges in the graph.

[20%]

Examples:

Input: `[3,[[1],[0,2],[]]]`
Output: 2

Input: `[4,[[1],[2],[3],[0]]]`
Output: 1

Constraints:

- $1 \leq |V| \leq 100$.
- $1 \leq |E| \leq 1000$.

Code:

In [ ]:
```
def strongly_connected(L):

    # Insert code here
```

Tests:

Run example test case 1:

In [ ]:
```
input_value = [3,[[1],[0,2],[]]]
print (strongly_connected(input_value))
```

Run example test case 2:

In [ ]:
```
input_value = [4,[[1],[2],[3],[0]]]
print (strongly_connected(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=3,question_part_id='d',function=strongly_connected,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

(e) Explain why your algorithm proposed in Part (d) is correct?

[10%]

Write your answer here.

4.

(a) You are provided as input an array A of n keys that has been generated by a stochastic process of the form:

$$A(i) = i + \mu$$

where μ is random noise drawn from the discrete uniform distribution, $\mu \sim \text{unif}\{-k, k\}$, for some k, where $k \ll n$. Implement an in-place variant of the quicksort algorithm as function, **quicksort**(A), to sort the array into increasing order. Your solution should have a worst-case run time of $O(n \log n)$.

[30%]

Examples:

Input: `[0, 1, 3, 2, 4]`
Output: `[0, 1, 2, 3, 4]`
Explanation: the input array has been sorted into increasing order.

Input: `[0, 1, 1]`
Output: `[0, 1, 1]`
Explanation: the input array was already in increasing order, so it remains unchanged.

Constraints:

- Every key is an integer.

Code:

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

    # Insert code here
```

Tests:

Run example test case 1:

In [ ]:
```
input_value = [0, 1, 3, 2, 4]
print(quicksort(input_value))
```

Run example test case 2:

In [ ]:
```
input_value = [0, 1, 1]
print(quicksort(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=4,question_part_id='a',function=quicksort,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

(b) Explain why code achieves the desired worst-case run time behaviour.

[10%]

Write your answer here.

(c) Suppose the stochastic process described in Part (a) is now corrupted by an additional source of noise. Regardless of the length of the sequence, n, all but 3 of the entries are set to zero. Does your implementation in Part (a) still guarantee $O(n \log n)$ worst-case runtime? Explain your answer.

[10%]

Write your answer here.

(d) Give brief answers to the following questions.

(i) What is the expected run time of the quicksort algorithm if the rank of the pivot selected during each recursion does not depend on the input array size? Explain your answer.

Note that the rank refers to 1-indexed position of an element in a list if the elements were to be sorted

into increasing order (for example, in the list `[0, 4, 3]`, the rank of `3` is 2, because `3` is the second largest element in the list).

[10%]

Write your answer here.

(ii) What is the expected run time of the quicksort algorithm if the rank of the pivot selected during each recursion is $K \cdot n$ for some K such that $0 < K < 1$. Explain your answer.

[10%]

Write your answer here.

(iii) Describe briefly how quicksort can be modified to guarantee $O(n \log n)$ worst-case runtime, and explain why such a modification may not be desirable in practice.

[10%]

Write your answer here.

(e) Given an input array B comprising n triplets of integers (i, j, k) (not necessary distinct), implement an in-place sorting algorithm that sorts the keys into increasing lexicographic order with worst-case $O(n \log n)$. Your code should not make use of Python's builtin sort functions.

[20%]

Examples:

Input: `[[2, 1, 0], [1, 0, 2]]`
Output: `[[1, 0, 2], [2, 1, 0]]`
Explanation: the triplets of B have been sorted into increasing lexicographic order.

Input: `[[1, 0, 2], [1, 0, 2]]`
Output: `[[1, 0, 2], [1, 0, 2]]`
Explanation: the triplets of B were already in increasing lexicographic order, so the array remains unchanged.

Constraints:

- $0 \leq n \leq 20$.

Code:

In [ ]:
```
def triplet_sort(B):

    # Insert code here
```

Examples:

Run example test case 1:

In [ ]:
```
input_value = [[2, 1, 0], [1, 0, 2]]
print(triplet_sort(input_value))
```

Run example test case 2:

In [ ]:
```
input_value = [[1, 0, 2], [1, 0, 2]]
print(triplet_sort(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=4,question_part_id='e',function=triplet_sort,
                  test_case_list=[1,2,3,4,5],verbose=True)
```

END OF PAPER