CS2201: Introduction to Computation
Make Up Test (26/04/2025 10:15AM)

Duration: 1 hour 45 mins **Total marks: 25**

Instructions

- All programs should be compatible with Python 3.
- There will be a plagiarism check. The suspected copies will attract a deduction of marks for the problem irrespective of who the original author of the code is. Copying from the internet will also be considered as plagiarism. However, you may reuse the codes shared by the instructor.
- Late submissions are not allowed.
- You are advised to compile the code before submitting it to WeLearn. If a code does not compile at our end, marks will be deducted for the problem.
- Each program should follow a strict naming convention: **QNo.py** (e.g. Q1.py, Q2a.py etc.). Programs not adhering to the convention will not be considered for evaluation.
- All codes (.py files) should be submitted in a single zipped folder named **YourWeLearnID.zip** (e.g. 23MS123.zip) to WeLearn.
- You should put appropriate comments in the code without which marks will be deducted.

1. **(Marks: 10)** In computer science, *graph* forms an important data structure. An example graph is shown in Figure 1. A graph consists of *nodes* (shown as filled black circles) which are connected by *edges* (shown as straight lines). The nodes are labeled as 1, 2, 3, 4, 5. Node 1 is connected to nodes 2, 4 and 5 by edges; node 3 is connected to 4 and so on. This can be thought of as five cities (nodes) connected by roads (edges). In our example, suppose that city 1 (node 1) is connected to cities (nodes) 2, 4 and 5 by roads (edges). Graphs are used in many interesting applications in computer science, e.g. in representing a network of cities and finding the shortest path (in terms of the number of edges) between cities (e.g. used in Google maps).

   Note that a graph can be represented as pairs of adjacent (connected by an edge) nodes. The graph in Figure 1 can be represented as (1, 2), (1, 4), (1, 5), (3, 4), (2, 4). Write a python program to input (use input()) a graph by asking from the user pairs of nodes and store the graph as a `list of lists` named *graph* which for our example will be be [[1, 2], [1, 4], [1, 5], [3, 4], [2, 4]].

   A graph can also be represented as an *adjacency list*. The adjacency list representation of the graph in Figure 1 is shown in Figure 2. The adjacency list here shows each node (in blue) with the other nodes to which it is connected in the graph shown by arrows. For example, here the node 1 is connected to the nodes 2, 4 and 5.

   Continue on the above program to create a `dictionary of lists` named *adjlst* from *graph* such that each *key* will be a node (as shown in blue in Figure 2) and the *value* will be a list of the nodes connected to it. For our example, *adjlst* will be {1: [2, 4, 5], 2: [1, 4], 3: [4], 4: [1, 3, 2], 5: [1]}.

2. **(Marks: 7)** In *Genetic Algorithms*, `Crossover` is a vital operation performed on chromosome sequences in bit arrays. An example of Crossover operation at position 4 on two bit arrays (indexed from 0) of the same length 10001101 and 11001100 is shown in Table 1. That is, the bit substrings from the position p till the end are interchanged.

   Write a program to input two bit strings (of 'str' type) named *s1* and *s2* (use input()). If they are not of the same length terminate with an appropriate message, else ask for the value of p (use input()). Finally,

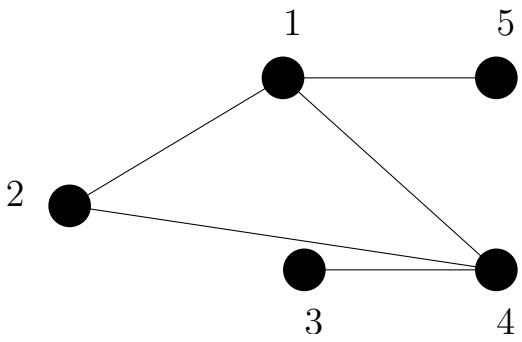

Figure 1: Graph

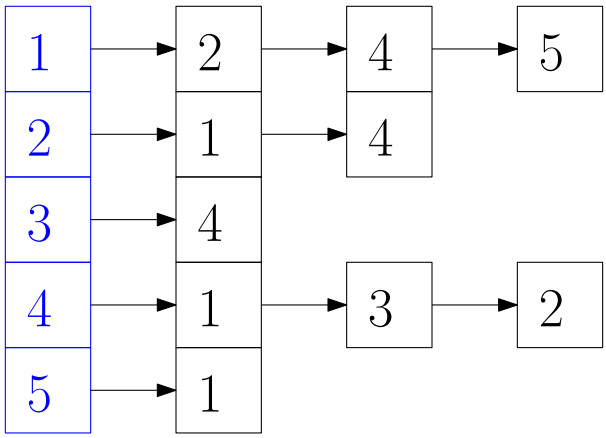

Figure 2: Adjacency List

perform crossover at p on *s1* and *s2* and print the changed versions of *s1* and *s2* after crossover. Note that, the original strings *s1* and *s2* should show the updated values after the crossover operation.

Before Crossover								
Positions	0	1	2	3	4	5	6	7
String 1	1	0	0	0	1	1	0	1
String 2	1	1	0	0	1	1	0	0
After Crossover at position 4								
Positions	0	1	2	3	4	5	6	7
String 1	1	0	0	0	1	1	0	0
String 2	1	1	0	0	1	1	0	1

Table 1: Crossover

3. **(Marks: 8)** Generate a *capcha* string of length six such that each element is either a numeric digit (0-9), a lower case character (ASCII range: 97–122) or an upper case character (ASCII range: 65–90), randomly chosen with approximately equal probability. Also, within a type (numeric digit, upper case character, or lowercase character), the element is chosen randomly from a uniform distribution. E.g. if the generated capcha is *1cYp5D*, the choice of an element (e.g. a numeric digit as the first element) is random among the three types, and, the digit '1' is generated from the digits 0–9 using a uniform distribution. A similar process is followed for the other two types (hint: use functions like `random.random()`, `random.uniform()` etc.).