Spring 2023

[EC2202] Data Structures

Final Exam: 1 pm, Tuesday, Jun. 13

INSTRUCTIONS

- **Do not open your exam sheet** until directed to do so, otherwise you would be considered cheating.
- You have **1 hour and 50 minutes** to complete the exam (7 problems; maximum of 100 points).
- The exam is closed book, closed notes, closed computer, and closed calculator.
- Mark your answers on the exam sheet itself and be sure to **write your answers in the space (boxes) provided**. We will not consider the answers given outside the boxes (use other space for brainstorming).

POLICIES & CLARIFICATIONS

- If you need to use the **restroom**, bring your exam sheet to the back of the room. Only one person is allowed at a time.
- You may use built-in Python functions that do not require import, such as min, max, pow, len, abs, and ord.
- For **all problems**, assume that you are implementing your program in the Colab environment.
- For **coding problems**, we will ignore minor grammar mistakes for evaluation (focus on the logic and flow). Moreover, your code must be indented correctly.
- Use **English** for your answers and name.

Student ID Number	Name

Q1. (10 points) Anagram

Implement the following function that checks whether two given strings are an anagram of each other or not. An ***anagram*** of a string is another string that contains the same characters, only the order of characters can be different. Two given strings `a` and `b` consist of lowercase characters.

```
def is_anagram(a, b):
  """
  >>> is_anagram('hello', 'random')
  False
  >>> is_anagram('allergy', 'allergic')
  False
  >>> is_anagram('a gentleman', 'elegant man')
  True
  >>> is_anagram('vacation time', 'i am not active')
  True
  >>> is_anagram('astronomer', 'moon starer')
  True
  """
```

```
  # an array of size 26, to store count of characters.
  mp = {}

  # storing count of each character in a.
  for i in a.replace(' ', ''):
    if i in mp.keys():
      mp[i] += 1
    else:
      mp[i] = 1

  # decrementing the count of characters encountered in b.
  for i in b.replace(' ', ''):
    if i not in mp.keys():
      return False
    else:
      mp[i] -= 1

  # check if the count at every index is equal to 0 or not.
  for i in mp.keys ():
    if mp[i] != 0:
      return False

  return True
```

Q2. (15 points) Trees

(a) (**5 points**) Implement the following function that returns the height of a tree. Recall that the height of a tree is the length of the longest path from the root to a leaf.

```
class Tree:
  def __init__(self, label, branches=[]):
    self.label = label
    for branch in branches:
      assert isinstance(branch, Tree)
    self.branches = list(branches)

  def is_leaf(self):
    return not self.branches


def height(t):
  """Returns the height of a tree.
  >>> t = Tree(3, [Tree(5, [Tree(1)]), Tree(2)])
  >>> height(t)
  2
  >>> t = Tree(3, [Tree(1), Tree(2, [Tree(5, [Tree(6)]), Tree(1)])])
  >>> height(t)
  3
  """
```

```
  # Non-list comprehension solution
  if is_leaf(t):
    return 0
  best_height = 0
  for b in branches(t):
    best_height = max(height(b), best_height)
  return best_height + 1

  # List comprehension solution
  if is_leaf(t):
    return 0
  return 1 + max([height(branch) for branch in branches(t)])

  # Alternate solutions
  return 1 + max([-1] + [height(branch) for branch in branches(t)])
  return max([1 + height(b) for b in branches(t)], default=0)
```

(b) (**5 points**) Implement the following function that takes in a tree and a value `x` and returns a list containing the nodes along the path required to get from the root of the tree to a node containing `x`. If `x` is not present in the tree, return `None`. Assume that the entries of the tree are unique.

```
def find_path(t, x):
  """
  >>> t = Tree(2, [Tree(7, [Tree(3), Tree(6, [Tree(5), Tree(11)])] ), Tree(15)])
  >>> find_path(t, 5)
  [2, 7, 6, 5]
  >>> find_path(t, 10)  # returns None
  """
```

```
  if label(t) == x:
    return [label(t)]
  for b in branches(t):
    path = find_path(b, x)
    if path:
      return [label(t)] + path
```

(c) (**5 points**) Implement the following function that takes in a tree and returns the maximum sum of the values along any path in the tree. Recall that a path is from the tree's root to any leaf.

```
def max_path_sum(t):
    """Returns the maximum path sum of the tree.
    >>> t = Tree(1, [Tree(5, [Tree(1), Tree(3)]), Tree(10)])
    >>> max_path_sum(t)
    11
    """
```

```
    # Non-list comprehension solution
    if is_leaf(t):
        return label(t)
    highest_sum = 0
    for b in branches(t):
        highest_sum = max(max_path_sum(b), highest_sum)
    return label(t) + highest_sum

    # List comprehension solution
    if is_leaf(t):
      return label(t)
    else:
      return label(t) + max([max_path_sum(b) for b in branches(t)])
```

Q3. (10 points) Objects Gone Wrong

Consider the following implementation of the Point class to represent a 2D point in the Cartesian space. For the following implementation, (a) (**3 points**) describe *two problems* of the given implementation, (b) (**5 points**) fix the problems by implementing additional methods, and (c) (**2 points**) describe how users might abuse your implementation in (b) and how to block the abuse.

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __repr__(self):
    return "Point(%s, %s)" % (self.x, self.y)
```

(a) The naive implementation does not support
  (1) the comparison of two points (checking equality) and
  (2) the hash functionality (checking the inclusion of an object in a set)

(b)

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __repr__(self):
    return "Point(%s, %s)" % (self.x, self.y)

  def __eq__(self, rhs):
    return self.x == rhs.x and self.y == rhs.y

  def __hash__(self):
    return hash((self.x, self.y))
```

(c) Users of (b) could *mutate the objects inside a set* (or a dictionary as key). Then, this incurs hashcode changes; objects located in a wrong slot afterwards. To block this type of abuse, we can make the object *immutable*.

Q4. (15 points) Minimum Number of Rooms

Implement the following function that returns the *minimum number of conference rooms* required to hold the given list of meetings. The list of meetings contains time intervals where intervals[i] = [$start_i$, end_i] represents the start and end times of each meeting.

```
import heapq

def min_num_rooms(intervals):
  '''
  >>> min_num_rooms([[0, 30], [5, 10], [15, 20]])
  2
  >>> min_num_rooms([[7, 10], [2, 4]])
  1
  >>> min_num_rooms([[5, 8], [3, 7], [10, 12], [6, 9]])
  3
  >>> min_num_rooms([[11, 15], [2, 9], [0, 12], [8, 13]])
  3
  '''
```

```
  # If there is no meeting to schedule then no room needs to be allocated.
  if not intervals:
    return 0

  # The heap initialization
  free_rooms = []

  # Sort the meetings in increasing order of their start time.
  intervals.sort(key=lambda x: x[0])

  # Add the first meeting. We have to give a new room to the first meeting.
  heapq.heappush(free_rooms, intervals[0][1])

  # For all the remaining meeting rooms
  for interv in intervals[1:]:
    # If there is a room freed up, assign that room to this meeting.
    if free_rooms[0] <= interv[0]:
      heapq.heappop(free_rooms)

    # If a new room is to be assigned, then we add it to the heap,
    # If an old room is allocated, then we add it to the heap with updated end time.
    heapq.heappush(free_rooms, interv[1])

  # The size of the heap tells us the minimum rooms required for all the meetings.
  return len(free_rooms)
```

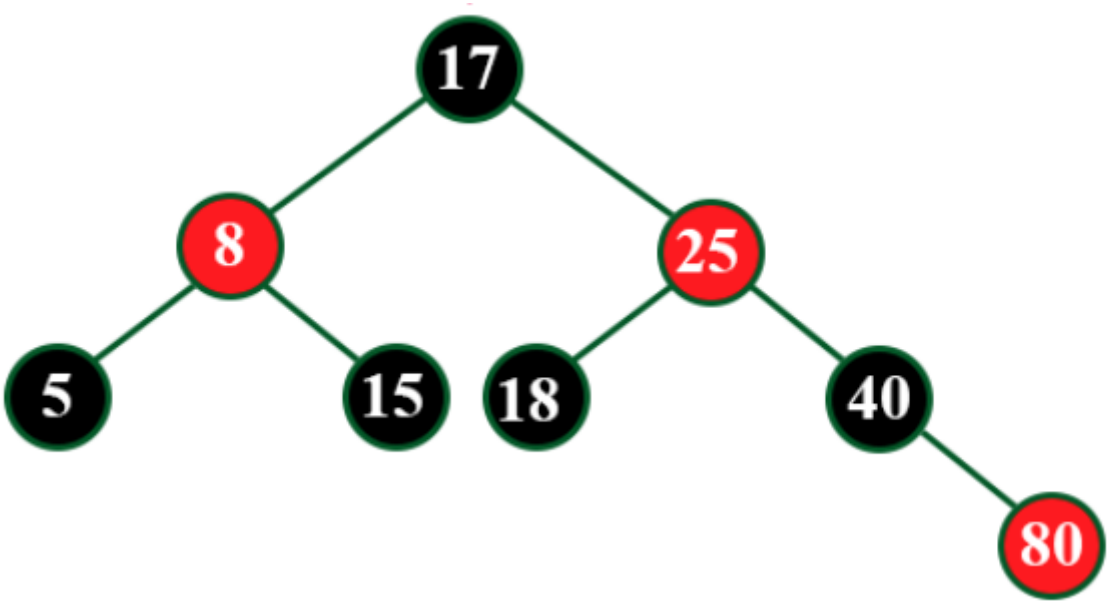
Q5 (15 points) Various Trees

(a) (**5 points**) State the *smallest* number of nodes N(h) that an AVL tree with height h can have in a recurrence form, justify your statement, and derive the upper bound of h in respect of n (the number of nodes in the tree).

N(h) = N(h - 1) + N(h - 2) + 1

- One of the subtrees (say the left subtree) must be an AVL tree of **height h - 1**
- Thus, it has at least **N(h - 1) nodes**
- The other subtree has height **either h - 1 or h - 2**
- Therefore, it has at least **N(h - 2) nodes**
- In conclusion, **N(h) = N(h - 1) + N(h - 2) + 1**
- Since clearly N(h - 1) ≥ N(h - 2), **N(h) ≥ 2N(h - 2)**
- Therefore, if the number of nodes in the tree is n: **$n \geq N(h) \geq 2^{h//2}$**
- which implies **(log n) ≥ h // 2**
- Thus, **h ≤ 2 log n**

(b) (**10 points**) Create a red-black tree by inserting the following sequence of numbers: 8, 18, 5, 15, 17, 25, 40 and 80. Assume that the red-black tree you are creating is *initially empty*.



Q6. (15 points) Quick Sort

Implement the quick sort algorithm in an in-place manner. Correct in-place implementation that does not require additional memory space will receive ***15 points***; other working implementations ***5 points***.

```
# 1. Correct in-place implementation
def partition(a, lo, hi):  # partition range a[lo:hi+1] and return index of pivot
  p = (lo + hi)//2
  pivot = a[p]
  a[p] = a[hi]  # Swap pivot with last item
  a[hi] = pivot

  i = lo - 1
  j = hi
  while i < j:
    i += 1
    while a[i] < pivot:
      i += 1
    j -= 1
    while a[j] > pivot and j > lo:
      j -= 1
    if i < j:
      t = a[i]; a[i] = a[j]; a[j] = t  # swap a[i] and a[j]
  a[hi] = a[i]
  a[i] = pivot # Put pivot where it belongs
  return i     # index of pivot

# sort range a[lo:hi+1]
def quick_sort(a, lo, hi):
  if (lo < hi):
    pivot = partition(a, lo, hi)
    quick_sort(a, lo, pivot - 1)
    quick_sort(a, pivot  + 1, hi)

# 2. Implementation requiring additional memory space
def quick_sort(a):
  if len(a) <= 1:
    return a
  pivot = a[len(a) // 2]
  small = []
  equal = []
  large = []
  for x in a:
    if x < pivot:
      small.append(x)
    elif x == pivot:
      equal.append(x)
    else:
      large.append(x)
  return quick_sort(small) + equal + quick_sort(large)
```

Q7. (20 points) Prim's Algorithm

You are given an array of points representing integer coordinates of some points on a 2D-plane, where points[i] = [x_i, y_i]. The cost of connecting two points [x_i, y_i] and [x_j, y_j] is the Manhattan distance between them: $|x_i - x_j| + |y_i - y_j|$. Implement the `min_cost` function that returns the minimum cost to make all points connected. All points are connected if there is exactly one simple path between any two points. For implementation, ***use Prim's algorithm***; Kruskal's algorithm will receive no points.

```
def min_cost(points):
  '''
  >>> min_cost([[0, 0], [2, 2], [3, 10], [5,
2], [7, 0]])
  20
  >>> min_cost([[3, 12], [-2, 5], [-4, 1]])
  18
  '''
```

```
class Edge:
  def __init__(self, point1, point2, cost):
    self.point1 = point1
    self.point2 = point2
    self.cost = cost

  def __lt__(self, other):
    return self.cost < other.cost
```

```
  if not points or len(points) == 0:
    return 0
  size, visited, pq = len(points), [False] * size, []
  result, count = 0, size - 1

  x1, y1 = points[0]
  for j in range(1, size):
    # Calculate the distance between two coordinates.
    x2, y2 = points[j]
    cost = abs(x1 - x2) + abs(y1 - y2)
    edge = Edge(0, j, cost)
    pq.append(edge)

  # Convert pq to a heap.
  import heapq; heapq.heapify(pq)

  visited[0] = True
  while pq and count > 0:
    edge = heapq.heappop(pq)
    point1, point2 = edge.point1, edge.point2
    cost = edge.cost
    if not visited[point2]:
      result += cost
      visited[point2] = True
      for j in range(size):
        if not visited[j]:
          distance = abs(points[point2][0] - points[j][0]) + \
                     abs(points[point2][1] - points[j][1])
          heapq.heappush(pq, Edge(point2, j, distance))
      count -= 1
  return result
```