1. **10 points** Apply the Partition algorithm to the following array. Write down the array after each step, drawing lines between the partitions and the pivot.

   \[2, 4, 1, 5, 3\]

   **Solution:**

   \[
   \begin{align*}
   & \begin{array}{c|c}
   2 & , 4, 1, 5 | 3 \\
   \end{array} & (2 \text{ points}) \\
   & \begin{array}{c|c}
   2 & 4, 1, 5 | 3 \\
   \end{array} & (2 \text{ points}) \\
   & \begin{array}{c|c}
   2 & 4, 1, 5 | 3 \\
   \end{array} & (2 \text{ points}) \\
   & \begin{array}{c|c}
   2, 1 & 4, 5 | 3 \\
   \end{array} & (2 \text{ points}) \\
   & \begin{array}{c|c}
   2, 1 & 4, 5 | 3 \\
   \end{array} & (2 \text{ points}) \\
   & \begin{array}{c|c}
   2 & 3, 5, 4 \\
   \end{array} & (2 \text{ points}) \\
   & \begin{array}{c|c}
   2 & 3, 5, 4 \\
   \end{array} & (2 \text{ points})
   \end{align*}
   \]

2. **10 points** Apply Radix Sorting to the following array of integers, showing the intermediate array after each iteration of the outer loop of the algorithm.

   \[19, 22, 29, 53, 81\]

   **Solution:** (1 point per integer in the correct position)

   \[
   \begin{align*}
   & 81 \quad 19 \\
   & 22 \quad 22 \\
   & 53 \quad 29 \\
   & 19 \quad 53 \\
   & 29 \quad 81
   \end{align*}
   \]
3. **10 points** Fill in the blanks to complete the following implementation of an adjacency-list representation of an undirected graph.

```python
class Edge:
    def __init__(self, src, tgt):
        self.source = src
        self.target = tgt

class UndirectedAdjList:
    def __init__(self, num_vertices):
        self.array = [[],
                      for i in (a)]
    def add_edge(self, u, v):
        self.array[v].append(u)
    def out_edges(self, u):
        return [Edge(u,v)
                for v in self.array[u]]
    def find_edge(self, u, v):
        for w in self.array[u]:
            if w == v:
                return Edge(u,v)
        return None
```

**Solution**: (2 points each)

(a) range(0,num_vertices)
(b) self.array[u].append(v)
(c) Edge(u,v)
(d) Edge(u,v)
(e) None

4. **10 points** Fill in the blanks to complete the following implementation of breadth-first search for an adjacency-list representation of a graph.

```python
def bfs(g, root, parent):
    for u in range(0, g.num_vertices()):
        ___(a)___ = None
    queue = deque([root])
    while len(queue) != 0:
        u = ___(b)___
        for e in ___(c)___:
            if not parent[e.target]:
                ___(d)___ = ___(e)___
```

---

**Solution**: (2 points each)

(a) range(0, g.num_vertices())
(b) len(queue) != 0
(c) parent[e.target]
(d) parent[e.target] = ___(e)___
Solution: (2 points each)
(a) parent[u]
(b) queue.popleft()
(c) g.out_edges(u)
(d) e.source
(e) queue.append(e.target)
5. **8 points** Apply Dijkstra’s shortest paths algorithm to the following graph, starting at vertex f, indicating the resulting shortest paths tree by making the tree edges into solid lines. Next to each vertex, write the distance of the shortest path from f to that vertex.

![Graph](image)

**Solution:** The solid lines indicate the shortest paths tree. (4 points) Each vertex is labeled with the shortest path from f to that vertex. (4 points)

![Tree](image)

6. **8 points** Identify a minimal spanning tree (MST) in the following graph using Kruskal’s algorithm. Off to the side, write down the edges in the MST in the order in which Kruskal’s algorithm adds them to the tree and make those edges into solid lines. Also write down the total weight of the minimum spanning tree that you have identified.
Solution: Edges in the MST in Kruskal order: (4 points)

(f, g), (e, i), (g, k), (j, k), (a, e), (b, f), (a, b), (c, g)

The graph with darkened MST edges: (2 points)

The weight of this minimal spanning tree is 60. (2 points)
7. **10 points** Fill in the blanks to complete the following implementation of a hashtable that uses the division method for hashing and uses chaining to handle collisions.

```python
class Hashtable(dict):
    def __init__(self, table_size=101):
        self.table = [[] for i in range(0, table_size)]
    def hash(self, key):
        prehash = hash(key)
        return prehash % len(self.table)
    def __getitem__(self, key):
        h = self.hash(key)
        for (k,v) in self.table[h]:
            if key == k:
                return v
        raise KeyError(key)
    def __setitem__(self, key, value):
        found = False
        h = self.hash(key)
        for i in range(0, len(self.table[h])):
            if key == self.table[h][i][0]:
                self.table[h][i] = (key,value)
                found = True;
                break
        if not found:
            self.table[h].append((key,value))
```

**Solution**: (2 points each)

(a) []
(b) prehash % len(self.table)
(c) len(self.table[h])
(d) self.table[h][i]
(e) self.table[h].append((key,value))

8. **7 points** Write down the equation for hashing based on the multiplication method. Suppose you have a hashtable with a table of size 8 and your keys are restricted to be 4-bit integers. Using the multiplication method, which hashtable bucket does the key 10 hash to? Use $A = 13$ as the random number used in the multiplication method.

**Solution**: The equation for the multiplication method is

$$h(k) = ((A \times k) \mod 2^w) >> (w - \lg m)$$  (3 points)
We have $A = 13, w = 4, k = 10, m = 8$, so we compute $h(k)$ as follows

$$h(10) = (13 \times 10 \mod 2^4) \gg (4 - 3)$$
$$= (130 \mod 16) \gg 1$$
$$= 2 \gg 1 = 10_b \gg 1$$
$$= l_b = 1$$

(4 points)
9. **10 points** Working modulo \( q = 13 \), how many spurious hits (false positives) does the Rabin-Karp matcher encounter in the text \( T = 314159 \) when looking for the text 20?

**Solution:** There is one spurious hits (4 points). They occur when considering the substring 59.

\[
\begin{align*}
20 \mod 13 &= 7 \\
31 \mod 13 &= 5 \\
14 \mod 13 &= 1 \\
41 \mod 13 &= 2 \\
15 \mod 13 &= 2 \\
59 \mod 13 &= 7
\end{align*}
\]

(Solutions that use rolling hashing and show results for append/skip are OK but not necessary.)

10. **7 points** Identify the strongly connected components in the following directed graph.

**Solution:** The strongly connected components are

\[
\{A, F, J, I\}, \{C, H, G, K\}, \{B\}, \{D\}, \{E\}, \{L\}
\]

(3 points for each 4-element component. 1 point for the rest.)
11. **10 points** Perform depth-first search on the following directed graph, with a preference for visiting lower-numbered vertices before higher-numbered vertices. Write down the discover and finish times (start counting from 1). Based on the finish times, write down the corresponding topological ordering of the vertices.

![Graph Diagram]

**Solution:** The discover and finish times are given in the superscripts: *(5 points)*

The topological ordering is *(5 points)*

8, 2, 5, 7, 1, 0, 3, 4, 6
12. **Extra Credit** Consider the following disjoint sets data structure for the integers $0 \ldots n - 1$. The `rep` array maps each integer to its representative. The `setof` array maps each representative to the list of the integers in its set. What is the worst-case time complexity of a sequence of $n$ union operations? (Hint: how many times can an element be moved from a smaller set into a larger set given that the size of the larger set becomes at least twice the size of the smaller set during each union.)

```python
class SimpleDisjointSets:
    def __init__(self, n):
        self.rep = [i for i in range(0, n)]
        self.setof = [[i] for i in range(0, n)]
    def find(self, i):
        return self.rep[i]
    def union(self, i, j):
        rep_i = self.rep[i]; rep_j = self.rep[j]
        if len(self.setof[rep_i]) < len(self.setof[rep_j]):
            small, large = rep_i, rep_j
        else:
            small, large = rep_j, rep_i
        for i in self.setof[small]:
            self.setof[large].append(i)
        self.rep[i] = large
        self.setof[small] = []
```

**Solution:** The time complexity $O(n \lg n)$. We obtain this by counting how many times we move an integer from one set to another. Each integer can only be moved a maximum of $\lg n$ times because the size of its set doubles on each move. There are $O(n)$ integers, so we multiply to get the overall time complexity of $O(n \lg n)$. 