X) 10 points: Radix Sort

Perform a radix sort of this list of numbers, using a radix of 10, into ascending order:

620  696  298  395  568  971  29  41  531  21

Show the bin/bucket sort conducted in each pass of the radix sort (i.e., as a table).

Write and circle the order of the numbers after each pass of the radix sort.

```
0  1  2  3  4  5  6  7  8  9
620  971  395  696  298  29  568
21  41  531
```

```
620  971  41  531  21  395  696  298  568  29
```

```
620  531  41  568  971  395  696
21  29
```

```
620  21  29  531  41  568  971  395  696  298
```

```
21  298  395  531  620  971
29  568  696
```

```
21  29  41  298  395  531  568  620  696  971
```

Unique ID: «Unique_ID»
X) 10 points : Graph Representation

Consider the following directed, weighted graph:

a) Draw an adjacency matrix representation of the above graph.

\[
\begin{matrix}
& A & B & C & D & E & F & G & H \\
A & 5 & 2 & 1 & 3 & 3 & & 4 \\
B & & & & & & & & \\
C & & & & & & & & \\
D & 1 & & & & & & \\
E & & 2 & & & & & \\
F & & & & & & & \\
G & & & & & 4 & & 1 \\
H & & & & 3 & & & \\
\end{matrix}
\]

b) Provide an appropriately tight \( O \) (Big-Oh) bound on the time for:

- For a given vertex pair \((v_1, v_2)\), testing whether there is an edge from \(v_1\) to \(v_2\): \( O(1) \)
- Computing the in-degree of a given vertex: \( O(v) \)
- Enumerating the vertices adjacent to a given vertex: \( O(v) \)
c) Draw an adjacency list representation of the above graph.

```
A → [B 5, C 2, G 4]
B → [D 1]
C → [G 1]
D → [A 1, H 4]
E → [D 2, F 2]
F
G → [D 4, H 1]
H → [F 3]
```

d) Provide an appropriately tight O (Big-Oh) bound on the time for:

For a given vertex pair \((v_1, v_2)\), testing whether there is an edge from \(v_1\) to \(v_2\):

\[ O(d) \] or \[ \mathcal{O}(u) \]

Computing the in-degree of a given vertex:

\[ O(E) \]

Enumerating the vertices adjacent to a given vertex:

\[ O(d) \] or \[ \mathcal{O}(u) \]

\[ d = \text{average out-degree} \]
X) 10 Points : Topological Sort

Consider the following directed graph:

You will perform two topological sorts on this graph.

In each sort, maintain a “bag” of “pending” vertices. When the processing of a vertex creates more than one new “pending” vertex, add the new “pending” vertices to the “bag” in alphabetical order (e.g., push (x), push (y), push (z)).

For each topological sort, write and circle the order the graph’s vertices are processed. Show your work to allow partial credit (e.g., show adding and removing from the “bag”).

a) Perform a topological sort using a queue to maintain the set of “pending” vertices:

\[ \begin{array}{cccccccc}
A & B & C & D & E & F & G & H & I & J \\
1 & 1 & 1 & 0 & 3 & 3 & 3 & 1 & 1 & 1 \\
6 & 6 & 6 & 0 & 2 & 2 & 2 & 0 & 0 & 0 \\
\end{array} \]

Q: DAHBECIFGJ

b) Perform a topological sort using a stack to maintain the set of “pending” vertices:

\[ \begin{array}{cccccccc}
A & B & C & D & E & F & G & H & I & J \\
1 & 1 & 1 & 0 & 3 & 3 & 3 & 1 & 1 & 1 \\
6 & 6 & 6 & 0 & 2 & 2 & 2 & 0 & 0 & 0 \\
\end{array} \]

S: DAHJIBECFG
X) 10 points : Single-Source Shortest Paths

Consider the following directed, weighted graph:

```
\begin{center}
\begin{tikzpicture}
\node[shape=circle,draw=black](A) at (1,2) {A};
\node[shape=circle,draw=black](B) at (3,2) {B};
\node[shape=circle,draw=black](C) at (1,0) {C};
\node[shape=circle,draw=black](D) at (2,-1) {D};
\node[shape=circle,draw=black](E) at (3,-1) {E};
\node[shape=circle,draw=black](F) at (4,0) {F};
\node[shape=circle,draw=black](G) at (1,-2) {G};
\node[shape=circle,draw=black](H) at (3,-2) {H};

\draw[->,thick] (A) -- (B) node [midway, above] {5};
\draw[->,thick] (A) -- (C) node [midway, above] {2};
\draw[->,thick] (A) -- (G) node [midway, above] {1};
\draw[->,thick] (B) -- (D) node [midway, above] {1};
\draw[->,thick] (B) -- (E) node [midway, above] {3};
\draw[->,thick] (C) -- (D) node [midway, above] {4};
\draw[->,thick] (D) -- (E) node [midway, above] {2};
\draw[->,thick] (D) -- (G) node [midway, above] {4};
\draw[->,thick] (E) -- (F) node [midway, above] {2};
\draw[->,thick] (E) -- (H) node [midway, above] {3};
\draw[->,thick] (G) -- (H) node [midway, above] {1};
\end{tikzpicture}
\end{center}
```

a) Step through Dijkstra's algorithm to calculate the single-source shortest paths from vertex \( A \) to every other vertex. Show your steps in the table below. Cross out old values and write in new ones, from left to right in each cell, as the algorithm proceeds. Also list the vertices in the order which Dijkstra's algorithm marks them known:

Order vertices marked as known: \( A \ C \ G \ H \ B \ D \ F \ E \)

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Known</th>
<th>Distance</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>( \times )</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>( B )</td>
<td>( \times )</td>
<td>5</td>
<td>( A )</td>
</tr>
<tr>
<td>( C )</td>
<td>( \times )</td>
<td>2</td>
<td>( A )</td>
</tr>
<tr>
<td>( D )</td>
<td>( \times )</td>
<td>7</td>
<td>( G \ B )</td>
</tr>
<tr>
<td>( E )</td>
<td>( \times )</td>
<td>8</td>
<td>( B )</td>
</tr>
<tr>
<td>( F )</td>
<td>( \times )</td>
<td>7</td>
<td>( H )</td>
</tr>
<tr>
<td>( G )</td>
<td>( \times )</td>
<td>4</td>
<td>( A \ C )</td>
</tr>
<tr>
<td>( H )</td>
<td>( \times )</td>
<td>4</td>
<td>( G )</td>
</tr>
</tbody>
</table>

b) What is the lowest-cost path from \( A \) to \( F \) in the graph, as computed above?

\[ A \rightarrow C \rightarrow G \rightarrow H \rightarrow F \]

c) To guarantee correctness of Dijkstra's algorithm, all edge costs must be non-negative. Imagine the edge from \( A \) to \( B \) had cost -3. Why would this make it impossible for any algorithm to provide a correct answer for single-source shortest paths?

\( \text{Negative cycle } A \rightarrow B \rightarrow D \rightarrow A \) would allow any path to be made shorter by going around the cycle one more time.
**X) 10 points : Minimum Spanning Tree**

Consider the following undirected, weighted graph:

- **a)** Step through Prim's algorithm to calculate a minimum spanning tree, starting from vertex *A*. Show your steps in the table below. Cross out old values and write in new ones, from left to right in each cell, as the algorithm proceeds. Also list the vertices in the order which Prim's algorithm marks them known:

<table>
<thead>
<tr>
<th>Order vertices marked as known:</th>
<th>A</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>G</th>
<th>H</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Known</th>
<th>Distance</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>×</td>
<td>D</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>×</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>×</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>×</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>×</td>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>×</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>×</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>×</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b)</th>
<th>What are the edges in the minimum spanning tree, as computed above?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B, D)</td>
<td>(C, A)</td>
</tr>
</tbody>
</table>
X) 10 Points : Work and Span

Consider the following directed acyclic graph representing the dependencies in a parallel computation implemented using a fork/join technique.

Each vertex is annotated with the cost of performing its work.

```
A 10
B 10
C 10
D 5
E 30
F 30
G 5
H 10
I 10
J 10
```

a) What is the work of this computation (i.e., a number)?

130

b) More generally, what is the work of a computation represented in this manner (i.e., described in terms of the graph and the cost of each vertex)?

\[\text{sum of the work in all vertices}\]

1.5

1.5

c) What is the span of this computation (i.e., a number)?

80

d) More generally, what is the span of a computation represented in this manner (i.e., described in terms of the graph and the cost of each vertex)?

\[\text{sum of the work in the most expensive path in the graph}\]

1.5
e) Assume two threads are executing a computation. We can illustrate the parallel work of multiple threads by drawing timelines of the work they execute. For example:

```
T1: 10 15 20 100
   V W Y
T2: X Z
   10 20 95
```

This pair of timelines illustrates a hypothetical computation in which:
- T1: V for 10 units, W for 5 units, idle for 5 units, Y for 80 units
- T2: idle for 10 units, X for 10 units, Z for 75 units, idle for 5 units

Draw a timeline using two threads to execute the above graph as quickly as possible.

Be sure your timeline illustrates the start and stop time of each task on each thread.

f) Would additional threads be able to perform the computation more quickly? Why?

No. This timeline completes in 80, which is the span.
7) 10 points: MoveToFrontList Concurrency

Consider this pseudocode for a MoveToFrontList, which is correct in a sequential context. It
does not map keys to data items, but instead just tests whether a key is in the list.

```java
01: class Node {
02:     Key key;
03:     Node next;
04: }
05: Node(...) { // Constructor that stores these 2 fields }
06: }
07: 
08: class MoveToFrontList {
09:     Node front = null;
10: 
11:     void insert(Key key) {
12:         front = new Node(key, front);
13:     }
14: 
15:     Boolean contains(Key find) {
16:         if(front == null) { return false; }
17:         if(front.key == find) { return true; }
18:         Node prev = front;
19:         Node current = front.next;
20:         while(current != null) {
21:             if(current.key == find) {
22:                 prev.next = current.next;
23:                 current.next = front;
24:                 front = current;
25:                 return true;
26:             }
27:         }
28:         current = current.next;
29:     }
30:     return false;
31: }
32: }
```

We have numbered the lines of code so that you can reference them in your answers. Please do
this, as it will be faster and will keep your answer more concise.

In describing an interleaving, you might write:

Thread 1 runs contains(...), stopping between lines 16 and 17.

In describing a modification of the code, you might write:

Insert additional code after line 9:

```
09a:     Node middle = null;
09b:     Node back = null;
```

Replace line 28:

```
30:         return true;
```

Unique ID: 89
Now consider using our MoveToFrontList in a multi-threaded context:

a) Describe an interleaving of insert("a") and contains("b") that results in the insert("a") being “missed” (i.e., contains("a") will return false).

\[ \text{T1 runs contains('b'), finds 'b' in list, pauses between lines 24 and 25} \]
\[ \text{T2 runs entire insert('a')} \]
\[ \text{T1 resumes and completes} \]

Line 25 in T1 overwrites front with the node containing the 'old' front as its 'next' value, the 'new' front is lost.

b) Describe an interleaving of insert("a") and insert("b") that results in the insert("a") being “missed” (i.e., contains("a") will return false).

\[ \text{T1 runs insert('b'), pauses on line 12 after Node constructor completes but before assignment to Front} \]
\[ \text{T2 runs entire insert('a')} \]
\[ \text{T1 resumes and completes} \]

Line 12 assignment in T1 overwrites front with node constructed using 'old' front, the 'new' front is lost.
c) Describe an interleaving of \texttt{contains("a")} and \texttt{contains("a")} that results in them returning different values (i.e., one returns \texttt{true} and one returns \texttt{false}).

\begin{itemize}
  \item \texttt{T1} runs \texttt{contains(\texttt{"a")}}, finds \texttt{\texttt{"a")}, or pauses between lines 231 and 235
  \item \texttt{T2} runs \texttt{contains(\texttt{"a")}}, does not find \texttt{\texttt{"a")}
  \item \texttt{T1} resumes and completes
\end{itemize}

\begin{itemize}
  \item \texttt{T1} takes \texttt{\texttt{"a")} out of the list in order to move it to the front, \texttt{T2} sees this bad state not containing \texttt{\texttt{"a")}
\end{itemize}

d) Using any of the mutual exclusion mechanisms discussed in lecture (including those unique to Java), describe how to fix this class so that it is correct in concurrent usage. Your only concern is correctness (i.e., performance is not a concern).

\begin{itemize}
  \item synchronize both \texttt{insert} and \texttt{contains}
\end{itemize}

\begin{itemize}
  \item \texttt{11 \texttt{008}: synchronized void insert (Key key)}
  \item \texttt{15 \texttt{608}: synchronized Boolean contains (Key key)}
\end{itemize}
8) 10 points: Heap Concurrency

You and a partner are implementing an array-based binary heap. Your partner wants to use
fine-grained locking to simultaneously allow multiple concurrent operations in the heap.
They propose the following strategy for implementing the locking:

1) Guard each location in the array with a lock (i.e., guard each node in the heap with a lock).
   Always obtain the lock before reading from or writing to the array location (i.e., the node).

2) Implement percolateUp and percolateDown such that they lock nodes in the course of
   the percolation. Before comparing the keys of two nodes, for example, they will lock both
   nodes. To ensure nothing else is reading or manipulating the portion of the heap affected by
   percolation, they will hold locks they obtain until the percolation completes.

Your partner claims this is a good strategy and that you can work out the details in the course of
the implementation. But you already see two major problems.

a) As described, this approach includes a data race. A critical variable for implementing the
   heap’s insert and deleteMin operations is unguarded. What is that critical variable?
   Give an example of a bad behavior might result from it being unprotected.

   The size of the heap is unguarded

   Insert or DeleteMin may fail due

b) Why will it be extremely difficult to guard the variable from (a) while also preserving your
   partner’s desire to allow multiple concurrent operations in the heap?

   Guarding size will be coarse-grained

c) In addition to potential race conditions, the proposed approach has another serious
   concurrency problem. What is the name for that problem? How could the problem occur
   with this strategy for implementing the locking?

   Deadlock
a) Provide pseudocode for the class InternTask. We provide its member declarations and its constructor. You just need to implement the compute() method. Do not use a sequential cutoff: the base case should process a single Customer. Your implementation should perform the computation in $O(n)$ work and $O(\log n)$ span.

```java
class InternResult {
    Customer[] customers;
    Boolean[] migrate;
    Letter[] letters;
    int numMigrate;

    InternResult(...) { // Constructor that stores these 4 fields }
}

class InternTask extends RecursiveTask<InternResult> {
    Customer[] customers;
    Boolean[] migrate;
    Letter[] letters;
    int low;
    int high;

    InternTask(...) { // Constructor that stores these 5 fields }

    InternResult compute() {
        // Base case
        if (low == high) {
            migrate[low] = InternProject.isProfitableToMigrate(customers[low]);
            letters[low] = InternProject.generateLetter(customers[low]);
        }
        return new InternResult(
            customers, migrate, letters,
            migrate[low] ? 1 : 0
        );
    }
}
```
Recursive Case

Intern Task left Task = new Intern Task(
    customers, migrate, letters, low, (low + high) / 2
);

Intern Task right Task = new Intern Task(
    customers, migrate, letters, (low + high) / 2, high
);

left Task. Fork();

Intern Result right Result = right Task. Compute();

Intern Result left Result = left Task. Join();

return new Intern Result(
    customers, migrate, letters,
    left Result. numMigrate + right Result. numMigrate
);

}