Question 1. Suppose that in some class we define two instance variables:

```java
private int[] t = new int[8];
private int count;
```

and that the only code executed to set the values associated with these variables is:

```java
t[0] = 8;
t[1] = 3;
t[2] = 6;
t[3] = 2;
t[4] = 9;
t[5] = 1;
t[6] = 7;
t[7] = 3;
```

```java
count = 5;
```

so that the contents of the array could be described by the table below:

| 8 | 3 | 6 | 2 | 9 | 1 | 7 | 3 |

Assume we have defined the method:

```java
public int indexOfSmallest(int size) {
    int posOfMin = 0;
    for (int i = 1; i < size; i++) {
        if (t[i] < t[posOfMin]) {
            posOfMin = i;
        }
    }
    return posOfMin;
}
```

a) What value would be returned by the invocation `indexOfSmallest(count)`?

b) What value would be returned by the invocation `indexOfSmallest(count-2)`?

c) Consider the following method:

```java
public void mystery(int size) {
    int remaining = size;
    while (remaining > 1) {
        int pos = indexOfSmallest(remaining);
        remaining = remaining - 1;
        int smallest = t[pos];
        t[pos] = t[remaining];
        t[remaining] = smallest;
    }
}
```

d) Describe the general behavior of the method named `mystery`.

Question 2.

Table 1 shows the results of executing the main loop of Dijkstra’s shortest path algorithm several times to build a forwarding table for the router named A. Unfortunately, the table is incomplete and much of the information about the neighbors of the routers disappeared during a freak transcription accident involving cod liver oil and a small rodent.
Please follow the steps of Dijkstra’s Algorithm to complete the entries in the *Best Route Length* and *First Step* columns of the table. The rows in which the *connections to neighboring routers* rows are completely empty are the places where the information disappeared. You should leave these table entries blank. You will discover you do not need this information to complete the *Best Route Length* and *First Step* columns of the table if you simply follow algorithm 1 as described below. Assume that all edges have positive integer lengths.

### Table 1: Shortest Path Table

<table>
<thead>
<tr>
<th>Cities</th>
<th>Best Route Length</th>
<th>First Step</th>
<th>Status</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>-</td>
<td>Known</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>C</td>
<td>Adjacent</td>
<td>K: 1</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>C</td>
<td>Known</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>11</td>
<td>L</td>
<td>Adjacent</td>
<td>B: 1</td>
</tr>
<tr>
<td>E</td>
<td>7</td>
<td>E</td>
<td>Adjacent</td>
<td>C: 3</td>
</tr>
<tr>
<td>F</td>
<td>-</td>
<td></td>
<td>Distant</td>
<td>L: 3</td>
</tr>
<tr>
<td>G</td>
<td>6</td>
<td>L</td>
<td>Known</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>10</td>
<td>C</td>
<td>Adjacent</td>
<td>I: 1</td>
</tr>
<tr>
<td>I</td>
<td>10</td>
<td>C</td>
<td>Adjacent</td>
<td>K: 3</td>
</tr>
<tr>
<td>J</td>
<td>5</td>
<td>C</td>
<td>Known</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>14</td>
<td>C</td>
<td>Adjacent</td>
<td>F: 1</td>
</tr>
<tr>
<td>L</td>
<td>4</td>
<td>L</td>
<td>Known</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>7</td>
<td>C</td>
<td>Known</td>
<td></td>
</tr>
</tbody>
</table>

---

**Algorithm 1** DISJKSTRA’S SHORTESTPATH

1. Mark starting point as *KNOWN* with length 0
2. Identify each neighbor of start as *ADJACENT*
3. Set first step of each neighbor of start to itself
4. Set route length of each neighbor to first step distance
5. While you don’t know the shortest path to all the cities:
   6. Select adjacent city with shortest ‘best route’ value
   7. Identify that adjacent city with shortest best route as *KNOWN*
   8. Mark neighbors of new *KNOWN* city that were *DISTANT* as *ADJACENT*
   9. Update path lengths and record first steps to *ADJACENT* neighbors of new *KNOWN* city.

---

**Question 3.**

Earlier in the semester, we introduced Huffman’s algorithm which finds an optimal binary prefix-free code for a set of \( n \) symbols \( s_1, \ldots, s_n \) which occur \( o_1, \ldots, o_n \) times respectively in some document. The procedure creates a set of trees – one for each symbol – and initializes the weight \( w_i \) of each tree to \( o_i \). It then selects two trees with minimum weight, and replaces them with a new tree composed of the subtrees. This new tree has weight equal to the sum of weights of its subtrees. This procedure is repeated until only a single tree remains. The resulting tree can be interpreted as a description of a set of codewords for the symbols used.
The cost of a Huffman tree is equivalent to the size of the document when it’s encoded using the code defined by the tree. If symbol $s_i$ appears $o_i$ times and $s_i$ is a leaf at depth $d_i$ in the tree then each occurrence of $s_i$ in the document is encoded with $d_i$ bits. As a result, $s_i$ contributes cost $o_i \times d_i$ to the total cost of the encoded document. In general the total cost of the encoded document (and hence the tree) is

$$o_1 \times d_1 + o_2 \times d_2 + \cdots + o_n \times d_n$$

(1)

For example, consider the following document (of length 11) composed of 5 symbols:

ABRACADABRA

The table below lists each symbol and its number of occurrences.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occurrences</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Running Huffman’s algorithm on the table might produce the following Huffman tree:

![Huffman Tree Diagram]

Encoding the document according to the code defined by the Huffman tree yields

10100110001001100

which has a length of 23. In other words, the cost of the tree is 23. We can verify this is indeed the cost of the code by noting that in the tree, $A$ has depth 1 and the remaining symbols have depth 3. Hence, using just the tree, we can also arrive at the cost by using Equation 1.

$$1 \times 3 + 1 \times 3 + 2 \times 3 + 2 \times 3 + 5 \times 1 = 3 + 3 + 6 + 6 + 5 = 23$$

Surprisingly, one does not need to actually create a tree to compute the cost of a Huffman code. To see how, imagine that we are creating the Huffman tree shown above. Initially, our running total is $T = 0$. Every time we merge two trees, the codewords for the symbols in those trees increase by 1 bit. Our procedure works by adding in this additional codeword cost at the time of a merge.

- We begin by merging $C$ and $D$. We now know that each occurrence of $C$ and $D$ will be replaced by codewords with length at least 1, so each occurrence of $C$ and each occurrence of $D$ will contribute cost at least 1 to the overall cost. Since $C$ and $D$ each occur only once, $T = T + 2 = 0 + 2 = 2$.

- Next we merge $B$ and $R$. Since $B$ and $R$ will be replaced by codewords with length at least 1, each occurrence of $B$ and each occurrence of $R$ will contribute cost at least 1 to the overall cost. Since $B$ and $R$ each occur twice they will contribute an additional cost of $2 + 2 = 4$ to $T$. Hence, $T = 6$.

- Next we merge the $CD$ tree with the $BR$ tree. The $CD$ tree has weight 2. This means $C$ and $D$ occur 2 times collectively in the document. The $BR$ tree has weight 4. This means $B$ and $R$ occur 4 times collectively in the document. Since we are merging these two nodes, we know the codewords for $C$ and $D$ will be 1 bit longer in the document. The same is true for $B$ and $R$. Since $C$ and $D$ occur twice collectively, this merge adds a cost of 2 to the total. Similarly, since $B$ and $R$ occur 4 times collectively, the merge adds a cost of 4 to the total. Hence the total is now $T = T + 2 + 4 = 6 + 2 + 4 = 12$. 


• Finally, we merge the CDBR node with the A node. Each occurrence of A will be encoded with a codeword of length 1 and the symbols in the CDBR node each require an additional bit in their keywords. Since A occurs 5 times and C,D,B, and R occur 6 times collectively, our total is $T = T + 5 + 6 = 12 + 5 + 6 = 23$.

Note that while we discussed computing the running total in terms of tree merges, we don’t actually need to create or merge any trees; we just need to find the two smallest weights and replace them with their sum in the current list of weights. This means each merge operation is really just a sum operation. Each sum operation has an associated update to the running total so it makes sense to talk about the value of the running total after each sum operation.

The series of steps that would be performed when applying this tree-less version of the algorithm to the occurrence counts for the ABRACADABRA example are shown below. In each step, the value in the list of occurrence counts that has been produced by merging two of the smallest occurrence counts in the previous list is shown in bold face. It is this value that is added to “Running total” at each step.

\[
\begin{align*}
\text{occurrences} &= \begin{bmatrix} 5 & 2 & 1 & 1 & 2 \end{bmatrix} \quad \text{Running total} = 0 \\
\text{occurrences} &= \begin{bmatrix} 5 & 2 & 2 & 2 \end{bmatrix} \quad \text{Running total} = 2 \\
\text{occurrences} &= \begin{bmatrix} 5 & 4 & 2 \end{bmatrix} \quad \text{Running total} = 6 \\
\text{occurrences} &= \begin{bmatrix} 5 & 6 \end{bmatrix} \quad \text{Running total} = 12 \\
\text{occurrences} &= \begin{bmatrix} 11 \end{bmatrix} \quad \text{Running total} = 23
\end{align*}
\]

Consider a document using 8 symbols whose occurrence counts are shown in the following table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occurrences</td>
<td>2</td>
<td>5</td>
<td>20</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>24</td>
<td>2</td>
</tr>
</tbody>
</table>

Fill in the cells in the tables provided below to show how the algorithm described above would unfold when applied to a message with these occurrence counts. To help you know whether you did this correctly, we have filled in the final values.

\[
\begin{align*}
\text{occurrences} &= \begin{bmatrix} 2 & 5 & 20 & 7 & 3 & 2 & 24 & 2 \end{bmatrix} \quad \text{Running total} = 0 \\
\text{occurrences} &= \begin{bmatrix} \end{bmatrix} \quad \text{Running total} = \\
\text{occurrences} &= \begin{bmatrix} \end{bmatrix} \quad \text{Running total} = \\
\text{occurrences} &= \begin{bmatrix} \end{bmatrix} \quad \text{Running total} = \\
\text{occurrences} &= \begin{bmatrix} \end{bmatrix} \quad \text{Running total} = \\
\text{occurrences} &= \begin{bmatrix} \end{bmatrix} \quad \text{Running total} = \\
\text{occurrences} &= \begin{bmatrix} \end{bmatrix} \quad \text{Running total} = \\
\text{occurrences} &= \begin{bmatrix} 65 \end{bmatrix} \quad \text{Running total} = 157
\end{align*}
\]