1. (20 Points) Using only the List ADT operations defined in the project description for pa1, write a
client function with the heading

```
List Merge(List A, List B, List C)
```

Your function will return a newly allocated List containing the elements of Lists $A$, $B$ and $C$, merged
together, one from $A$, then one from $B$, then one from $C$. If one of the Lists becomes exhausted before
the others, then continue merging from the other Lists until all elements from $A$, $B$ and $C$ have been
used. For instance, if $A = [1, 2, 3], B = [4, 5]$ and $C = [6, 7, 8, 9]$, then Merge($A, B, C$) will return the
List $[1, 4, 6, 2, 5, 7, 3, 8, 9]$. Merge() will make no changes to the states of its three arguments, and has
no preconditions.

One of Several Possible Solutions:

```
List Merge(List A, List B, List C){
    List L = newList();

    moveFront(A);
    moveFront(B);
    moveFront(C);
    while( index(A)>=0 || index(B)>=0 || index(C)>=0 ){
        if( index(A)>=0 ) {
            append(L, get(A));
            moveNext(A);
        }
        if( index(B)>=0 ) {
            append(L, get(B));
            moveNext(B);
        }
        if( index(C)>=0 ) {
            append(L, get(C));
            moveNext(C);
        }
    }
    return L;
}
```
2. (20 Points) Using only the List ADT operations defined in the project description for pa1, write a *client* function with the heading

```plaintext
List Find(List L, int x)
```

Your function will return a new List consisting of all index positions within $L$ at which the element $x$ is located. For instance, if $L = [1, 5, 1, 6, 4, 1, 3, 1, 2]$ and $x = 1$, then `Find()` will return the List $[0, 2, 5, 7]$. If $x$ is not contained in the argument List $L$, then `Find()` will return an empty List. `Find()` will make no changes to the state of its argument $L$, and has no preconditions.

**One of Several Possible Solutions:**

```plaintext
List Find(List L, int x){
    List S = newList();
    moveFront(L);
    while( index(L)>=0 ){
        if( x==get(L) ){
            append(S, index(L));
        }
        moveNext(L);
    }
    return S;
}
```
3. (20 Points) Given a connected (undirected) graph $G$, and a vertex $x$ in $G$, the eccentricity of $x$ is the maximum possible distance from $x$, to any other vertex $y$ in $G$, i.e.

$$\text{eccentricity}(x) = \max \{ \delta(x, y) | y \in V(G) \}$$

Using only the Graph ADT operations defined in the project description for pa2, write a client function with the heading

$$\text{int Eccentricity(Graph G, int x)}$$

Your function will compute and return the eccentricity of vertex $x$ within Graph $G$.

One of Several Possible Solutions:

```c
int Eccentricity(Graph G, int x) {
    int max, y;
    BFS(G, x);
    max = getDist(G, 1);
    for(y=2; y<=getOrder(G); y++){
        if( getDist(G, y)>max )
            max = getDist(G, y);
    }
    return max;
}
```
4. (20 Points) Run the BFS algorithm on the graph pictured below, with vertex $s = 5$ as the source. Fill in the table giving the adjacency list representation, colors, distances from the source, and parents in the BFS tree. List the vertices in the order that they enter the queue. Draw the resulting BFS tree.

Queue: 5 2 9 10 1 3 6 7 4 11 8 12

BFS Tree:

<table>
<thead>
<tr>
<th>vertex</th>
<th>adj</th>
<th>color</th>
<th>distance</th>
<th>parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>black</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1 3 5 6</td>
<td>black</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2 7</td>
<td>black</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>7 8</td>
<td>black</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>2 9 10</td>
<td>black</td>
<td>0</td>
<td>nil</td>
</tr>
<tr>
<td>6</td>
<td>2 7</td>
<td>black</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3 4 6 10 11</td>
<td>black</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>4 11</td>
<td>black</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>black</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>5 7</td>
<td>black</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>7 8 12</td>
<td>black</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>black</td>
<td>4</td>
<td>11</td>
</tr>
</tbody>
</table>
5. (20 Points) Run the DFS algorithm on the digraph pictured below. Process vertices in the main loop of DFS() by increasing vertex label. Process vertices in the for loop of Visit() by increasing vertex labels. As vertices finish, push them onto a stack. Fill in the table below giving the adjacency list representation, discover times, finish times and parents in the DFS forest. Draw the resulting DFS forest, and show the state of the stack when DFS is complete. Classify all edges as of type tree, back forward or cross.

```
<table>
<thead>
<tr>
<th>vertex</th>
<th>adj</th>
<th>discover</th>
<th>finish</th>
<th>parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 5</td>
<td>1</td>
<td>12</td>
<td>nil</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>3 6</td>
<td>3</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>5 8</td>
<td>13</td>
<td>18</td>
<td>nil</td>
</tr>
<tr>
<td>8</td>
<td>5 9</td>
<td>14</td>
<td>17</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>15</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>
```

DFS Forest:

```
1 -> 2 -> 5 -> 3
|
 v   v   v
2   5   3
|
 v   v
5   6
|
 v   v
3   6
|
 v   v
4
```

Stack:

```
7
8
9
1
2
5
6
4
3
```

Edge Classification:

Tree: (1, 2) (2, 5) (5, 3) (5, 6) (6, 4) (7, 8), (8, 9)

Back:

Forward: (1, 5)

Cross: (4, 3) (7, 5) (8, 5) (9, 6)