1 Short Answer

1. State the precise mathematical definitions of Big-O, Big-Ω, and Big-Θ. Also give the common meaning of each, specifically, what bound does it indicate?

   Solution:

   - A function $g(n)$ is $O(f(n))$ if there exist a constants $c > 0$ and n_0 such that, for every $n > n_0$, $|g(n)| \leq cf(n)$. This is an Upper Bound on the complexity.
   - A function $g(n)$ is $\Omega(f(n))$ if there exist a constants $c > 0$ and n_0 such that, for every $n > n_0$, $|g(n)| \geq cf(n)$. This is a Lower Bound on the complexity.
   - A function $g(n)$ is $\Theta(f(n))$ if there exist a constants $c_1 > 0$, $c_2 > 0$, and n_0 such that, for every $n > n_0$, $c_1 f(n) \leq |g(n)| \leq c_2 f(n)$. This is a Tight Bound on the complexity.

2. Fill out the time complexity table below.

   Solution:

Algorithm	Best	Average	Worst
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Tree Sort with BST	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
Linear Search on Array	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search on Sorted Array	$\Omega(1)$	$\Theta(\log(n))$	$O(\log(n))$

3. Write a recursive function that will compute the double factorial. The double factorial is defined as

$$n!! = n \cdot (n-2) \cdot (n-4) \cdots 1$$

   and $0!! = 1$. For example, $3!! = 3$, $4!! = 8$, $5!! = 15$, $6!! = 48$, $7!! = 105$, ...

   Solution:

```
long dfact(long n) {
    if (n < 2)
        return 1;
    return n * dfact(n - 2);
}
```

4. Write a templated recursive binary search function for an array, assume the array is already sorted.

   Solution:

```
template<class T>
int binarySearch(T A[], int left, int right, T target) {
    if (right >= left) {
        int mid = (right + left) / 2;

        if (A[mid] == target)
            return mid;
        else if (A[mid] > target)
            return binarySearch(A, left, mid - 1, target);
        else
            return binarySearch(A, mid + 1, right, target);
    }
    return -1;
}
```

5. Write four functions to be added to the (singularly linked) LinkedList class, the specifications to these are below. Your implementation should be written as functions that are outside the specification.

```
void displayListRec();
void displayListRecRev();
void displayListRec(ListNode<T> *t);
void displayListRecRev(ListNode<T> *t);
```

- The functions `displayListRec()` and `displayListRec(ListNode<T> *t)` work together to print out the list to the console in order.
- The functions `displayListRecRev()` and `displayListRecRev(ListNode<T> *t)` work together to print out the list to the console in reverse order.
- `displayListRec()` is non-recursive, public, and does not print anything directly to the console. It simply does the appropriate call to `displayListRec(ListNode<T> *t)`.
- `displayListRec(ListNode<T> *t)` is recursive, private, and prints the data to the console.
- `displayListRecRev()` is non-recursive, public, and does not print anything directly to the console. It simply does the appropriate call to `displayListRecRev(ListNode<T> *t)`.
- `displayListRecRev(ListNode<T> *t)` is recursive, private, and prints.

With these added to the LinkedList class the following program will produce the following output. The data of the ListNode is stored in field named `value`.

```
int main() {
    LinkedList<int> list;
    list.appendNode(7);
    list.appendNode(2);
    list.appendNode(4);
    list.appendNode(1);
    list.appendNode(9);
    list.appendNode(8);
    list.displayListRec();
    cout << endl;
    list.displayListRecRev();
    cout << endl;
    return 0;
}
```

Output:

```
7 2 4 1 9 8
8 9 1 4 2 7
```

Solution:

```
template <class T> void LinkedList<T>::displayListRec() {
    displayListRec(head);
}

template <class T> void LinkedList<T>::displayListRecRev() {
    displayListRecRev(head);
}

template <class T> void LinkedList<T>::displayListRec(ListNode<T> *t) {
    if (t) {
        cout << t->value << " ";
        displayListRec(t->next);
    }
}

template <class T> void LinkedList<T>::displayListRecRev(ListNode<T> *t) {
    if (t) {
        displayListRecRev(t->next);
        cout << t->value << " ";
    }
}
```

2 Coding Exercise

This exercise is to code portions of a general templated binary search tree. The specification for the class is below.

```
template <class T> class BinaryTree {
  private:
    class TreeNode {
      public:
        T value;
        TreeNode *left;
        TreeNode *right;

        TreeNode(T nodeValue) {
            value = nodeValue;
            left = nullptr;
            right = nullptr;
        }
    };

    TreeNode *root;

    void insert(TreeNode *&, TreeNode *&);
    void destroySubTree(TreeNode *);
    void deleteNode(T, TreeNode *&);
    void makeDeletion(TreeNode *&);
    void displayInOrder(TreeNode *) const;
    int numberOfNodesRec(TreeNode *);

  public:
    BinaryTree();
    ~BinaryTree();
    BinaryTree(const BinaryTree &obj);
    const BinaryTree operator=(const BinaryTree &right);

    void displayInOrder() const;
    void insertNode(T);
    bool searchNode(T);
    void remove(T);
    int numberOfNodes();
};
```

- Constructor, destructor, copy constructor, and overloaded assignment do their usual jobs.
- `destroySubTree` removes the subtree starting at the input node.
- `displayInOrder` and its recursive counterpart prints the tree contents to the console using an in-order traversal of the tree.
- `insertNode` and its recursive counterpart inserts the item in the correct place in the binary search tree.
- `searchNode` returns true if the item is in the tree and false if not.
- `remove` invokes the `deleteNode` and `makeDeletion` functions to remove the item from the tree. `deleteNode` recursively finds the node to delete and `makeDeletion` does the actual deletion of the node.
- `numberOfNodes` and its recursive counterpart counts the total number of nodes in the tree.
- There are, of course, to be no memory leaks.
- If you find the need to add in another function, feel free to do so but you must, of course, write the implementation of the functions you add.
- No inline code for these implementations.

Solution:

```
template <class T> BinaryTree<T>::BinaryTree() { root = nullptr; }
template <class T> BinaryTree<T>::~BinaryTree() { destroySubTree(root); }
template <class T> BinaryTree<T>::BinaryTree(const BinaryTree &obj) { buildTreeRec(root, obj.root
    ); }

template <class T> const BinaryTree<T> BinaryTree<T>::operator=(const BinaryTree &right) {
    if (this != &right) {
        destroySubTree(root);
        buildTreeRec(root, right.root);
    }
    return *this;
}

template <class T> void BinaryTree<T>::buildTreeRec(TreeNode *&nodePtrD, TreeNode *nodePtrS) {
    if (!nodePtrS)
        return;

    nodePtrD = new TreeNode(nodePtrS->value);
    buildTreeRec(nodePtrD->left, nodePtrS->left);
    buildTreeRec(nodePtrD->right, nodePtrS->right);
}

template <class T> void BinaryTree<T>::destroySubTree(TreeNode *nodePtr) {
    if (nodePtr) {
        if (nodePtr->left)
            destroySubTree(nodePtr->left);
        if (nodePtr->right)
            destroySubTree(nodePtr->right);
        delete nodePtr;
    }
}

template <class T> void BinaryTree<T>::displayInOrder() const {
    displayInOrder(root);
}

template <class T> void BinaryTree<T>::displayInOrder(TreeNode *nodePtr) const {
    if (nodePtr) {
        displayInOrder(nodePtr->left);
        cout << nodePtr->value << endl;
        displayInOrder(nodePtr->right);
    }
}

template <class T> void BinaryTree<T>::insertNode(T item) {
    TreeNode *newNode = new TreeNode(item);
    insert(root, newNode);
}

template <class T> void BinaryTree<T>::insert(TreeNode *&nodePtr, TreeNode *&newNode) {
    if (nodePtr == nullptr)
        nodePtr = newNode;
    else if (newNode->value < nodePtr->value)
        insert(nodePtr->left, newNode);
    else
        insert(nodePtr->right, newNode);
}

template <class T> bool BinaryTree<T>::searchNode(T item) {
    TreeNode *nodePtr = root;

    while (nodePtr) {
        if (nodePtr->value == item)
            return true;
        else if (item < nodePtr->value)
            nodePtr = nodePtr->left;
        else
            nodePtr = nodePtr->right;
    }
    return false;
```

```
}

template <class T> void BinaryTree<T>::remove(T item) {
    deleteNode(item, root);
}

template <class T> void BinaryTree<T>::deleteNode(T item, TreeNode *&nodePtr) {
    if (!nodePtr)
        return;

    if (item < nodePtr->value)
        deleteNode(item, nodePtr->left);
    else if (item > nodePtr->value)
        deleteNode(item, nodePtr->right);
    else
        makeDeletion(nodePtr);
}

template <class T> void BinaryTree<T>::makeDeletion(TreeNode *&nodePtr) {
    TreeNode *tempNodePtr = nullptr;

    if (nodePtr == nullptr)
        cout << "Cannot delete empty node.\n";
    else if (nodePtr->right == nullptr) {
        tempNodePtr = nodePtr;
        nodePtr = nodePtr->left;
        delete tempNodePtr;
    } else if (nodePtr->left == nullptr) {
        tempNodePtr = nodePtr;
        nodePtr = nodePtr->right;
        delete tempNodePtr;
    } else {
        tempNodePtr = nodePtr->right;
        while (tempNodePtr->left)
            tempNodePtr = tempNodePtr->left;

        tempNodePtr->left = nodePtr->left;
        tempNodePtr = nodePtr;
        nodePtr = nodePtr->right;
        delete tempNodePtr;
    }
}

template <class T> int BinaryTree<T>::numberOfNodes() { return numberOfNodesRec(root); }

template <class T> int BinaryTree<T>::numberOfNodesRec(TreeNode *nodePtr) {
    if (!nodePtr)
        return 0;

    return numberOfNodesRec(nodePtr->left) + numberOfNodesRec(nodePtr->right) + 1;
}
```