# 1 Short Answer

1. (*10 Points*) State the precise mathematical definitions of Big-$O$, Big-$\Omega$, and Big-$\Theta$. 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. (*10 Points*) 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 | $\Omega(n\lg(n))$ | $\Theta(n\lg(n))$ | $O(n^2)$ |
| Merge Sort | $\Omega(n\lg(n))$ | $\Theta(n\lg(n))$ | $O(n\lg(n))$ |
| Linear Search on Array | $\Omega(1)$ | $\Theta(n)$ | $O(n)$ |
| Binary Search on Sorted Array | $\Omega(1)$ | $\Theta(\lg(n))$ | $O(\lg(n))$ |

3. (*5 Points*) Using the definition of $O(f(n))$, prove that $T(n) = 2n^2 + 3n + 1$ is $O(n^2)$.

   **Solution:** A function $g(n)$ is $O(f(n))$ if there exist a constants $c > 0$ and $n_0$ such that, for every $n > n_0$, $|g(n)| \leq cf(n)$.

   For $n \geq 1$, $T(n) = 2n^2 + 3n + 1 \leq 2n^2 + 3n^2 + n^2 = 6n^2$. So let $c = 6$ and $n_0 = 1$.

4. (*5 Points*) 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);
}
```

5. (*5 Points*) 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;
}
```

6. (*5 Points*) Draw the binary search tree after the following values have been inserted in this order.

`17, 3, 21, 5, 2, 19, 18, 32, 31, 25, 20, 7, 10, 9`

**Solution:**

```
            32
                31
                    25
        21
                20
            19
                18
17
                    10
                        9
                7
            5
        3
            2
```

# 2 Coding Exercises

1. (*15 Points*) 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. No inline code.

```
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. (*20 Points*) Write the implementation of the templated Quick Sort algorithm we did in class.

**Solution:**

```
template<class T>
void quickSort(T A[], int left, int right) {
    int i = left;
    int j = right;
    int mid = (left + right) / 2;

    T pivot = A[mid];

    while (i <= j) {
        while (A[i] < pivot)
            i++;

        while (A[j] > pivot)
            j--;

        if (i <= j) {
            T tmp = A[i];
            A[i] = A[j];
            A[j] = tmp;
            i++;
            j--;
        }
    }

    if (left < j)
        quickSort(A, left, j);

    if (i < right)
        quickSort(A, i, right);
}

template<class T>
void quickSort(T A[], int size) {
    quickSort(A, 0, size - 1);
}
```

3. (*25 Points*) This exercise is to code portions of the integer binary search tree we discussed in class. The specification for the class is below.

```
class IntBinaryTree {
private:
    struct TreeNode {
        int value;
        TreeNode *left;
        TreeNode *right;
    };

    TreeNode *root;

    void insert(TreeNode*&, TreeNode*&);
    void destroySubTree(TreeNode*);
    void deleteNode(int, TreeNode*&);
    void makeDeletion(TreeNode*&);
    void displayInOrder(TreeNode*) const;
    void displayPreOrder(TreeNode*) const;
    void displayPostOrder(TreeNode*) const;
    void IndentBlock(int);
    void PrintTree(TreeNode*, int, int);

public:
    IntBinaryTree() { root = nullptr; }
    ~IntBinaryTree() { destroySubTree(root); }

    void insertNode(int);
    bool searchNode(int);
    void remove(int);

    void displayInOrder() const { displayInOrder(root); }
    void displayPreOrder() const { displayPreOrder(root); }
    void displayPostOrder() const { displayPostOrder(root); }
    void PrintTree(int Indent = 4, int Level = 0);
};
```

Code only the following functions. Your implementation should be written as functions that are outside the specification. No inline code.

- Write the `insertNode` and the `insert` functions that will collectively insert a node into the tree in the correct position.
- Write the `searchNode` function that will return true if the value being searched for is in the tree and false otherwise.
- Write the `remove`, `deleteNode`, and the `makeDeletion` functions that will collectively delete a node from the tree.

**Solution:**

```
void IntBinaryTree::insertNode(int num) {
    TreeNode *newNode = nullptr;
    newNode = new TreeNode;
    newNode->value = num;
    newNode->left = newNode->right = nullptr;
    insert(root, newNode);
}

void IntBinaryTree::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);
}

bool IntBinaryTree::searchNode(int num) {
    TreeNode *nodePtr = root;
    while (nodePtr) {
        if (nodePtr->value == num)
            return true;
        else if (num < nodePtr->value)
            nodePtr = nodePtr->left;
        else
            nodePtr = nodePtr->right;
    }
    return false;
}

void IntBinaryTree::remove(int num) {
    deleteNode(num, root);
}

void IntBinaryTree::deleteNode(int num, TreeNode *&nodePtr) {
    if (!nodePtr)
        return;

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

void IntBinaryTree::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;
    }
}
```

# 3 Extra Credit

1. (*5 Points*) Prove that $T(n) = 2^{\sqrt{\lg n}}$ is $O(n^a)$ for any constant $a > 0$.

   **Solution:** For a constant $a > 0$ we need to show that there are constants $c > 0$ and $n_0$ such that, for every $n > n_0$, $|g(n)| \leq cf(n)$. That is, $2^{\sqrt{\lg n}} \leq cn^a$.

$$\begin{aligned}
2^{\sqrt{\lg n}} &\leq cn^a \\
\lg(2^{\sqrt{\lg n}}) &\leq \lg(cn^a) \\
\sqrt{\lg n}\lg(2) &\leq a\lg(n) + \lg(c) && \text{Let } c = 1. \\
\sqrt{\lg n}\lg(2) &\leq a\lg(n) \\
\frac{\lg(2)}{a} &\leq \sqrt{\lg(n)} \\
\left(\frac{\lg(2)}{a}\right)^2 &\leq \lg(n) \\
n &\geq 2^{\left(\frac{\lg(2)}{a}\right)^2}
\end{aligned}$$

   So let $c = 1$ and $n_0 = 2^{(\lg(2)/a)^2}$.