1 Short Answer: 5 Points Each

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	$\Omega(n \lg(n))$	$\Theta(n \lg(n))$	$O(n^2)$
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. Using the definition of $O(f(n))$, prove that $T(n) = 3n^2 + 5n + 1$ 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 $n \geq 1$, $T(n) = 3n^2 + 5n + 1 \leq 3n^2 + 5n^2 + n^2 = 9n^2$. So let $c = 9$ and $n_0 = 1$.

4. 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. 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. Write the following Linux console commands to do the following.

   (a) The command to get a directory listing of all files (including hidden files) in long format. You may not use any aliases.

   Solution: `ls -al`

   (b) The command to create a new and empty text file named `main.cpp`.

   Solution: `touch main.cpp`

   (c) The command to move back a single directory (that is from the current directory to the parent directory) and the command to move back to the user's home directory no matter where they are.

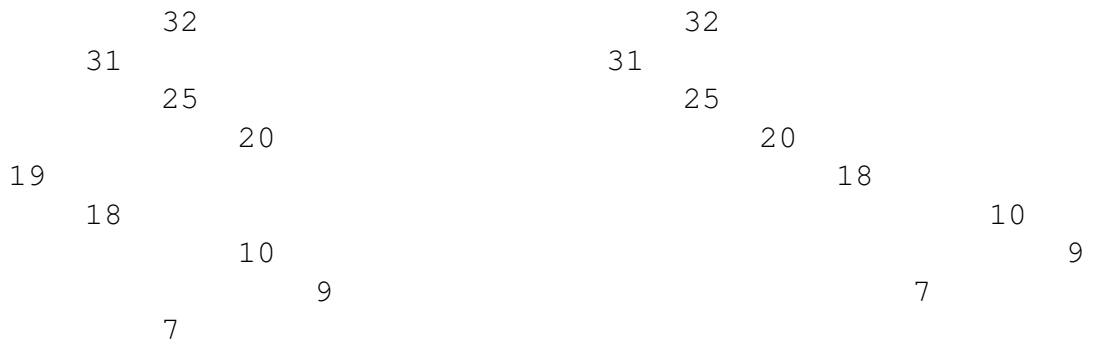
   Solution: `cd ..` and `cd ~` (or just `cd`).

7. Draw the binary search tree after the following values have been inserted in this order.

   `19, 18, 31, 25, 32, 20, 7, 10, 9`

   Then draw the tree after the root node has been removed.

   Solution:



8. Write a complete make file for compiling the following code project. The compilation must do the minimal amount of recompiling dependent on the changes in the files. Also, the recompilation should track changes in all dependent files. The project to be compiled has three files in it, main.cpp, NumberList.h and NumberList.cpp. The main.cpp and NumberList.cpp both include NumberList.h. There is no templating in these class structures so the specification and implementation of the NumberList class is divided between the two files. Use an arrow "⟶" to denote a tab character.
   Solution:

```
PROG = prog
CC = g++
CPPFLAGS = -g -Wall -std=c++11
OBJS = main.o NumberList.o

$(PROG) : $(OBJS)
⟶$(CC) -o $(PROG) $(OBJS)

main.o : main.cpp NumberList.h
⟶$(CC) $(CPPFLAGS) -c main.cpp

NumberList.o : NumberList.h NumberList.cpp
⟶$(CC) $(CPPFLAGS) -c NumberList.cpp

clean:
⟶rm -f core $(PROG) $(OBJS)

rebuild:
⟶make clean
⟶make
```

2 Coding Exercises: 20 Points Each

1. This exercise is to write a linked list class with some basic functionality. The class is to be templated so that it can store any data type that supports assignment, streaming out, and equality testing (i.e. ==). Specifically, the class structure is to implement the following. As usual, there is not to be any inline code in the specification for any of the functions.

   - The list node should be an internal private struct (or class) named `ListNode`.
   - A default constructor only.
   - A destructor, obviously. Make sure that there are no memory leaks.
   - `appendNode` that takes a single parameter, the element to be added to the list, and appends the element on to the end of the list.
   - `insertFront` that takes a single parameter, the element to be added to the list, and puts the element on to the front of the list.
   - `deleteNode` that takes a single parameter, the element to be deleted from the list, and removes the first occurrence of the element from the list. If the element is not in the list then the list is unaltered.
   - `displayList` that will display the list to the console screen horizontally.

   The following three pages are for you answer to this exercise. There is a sample program below and its output. Read this very closly, your class structure is to produce exactly the same output to this sample code. The next three pages are for your answer.

```
#include "LinkedList.h"
#include <iostream>

using namespace std;

int main() {
    LinkedList<int> list;

    list.appendNode(5);
    list.appendNode(2);
    list.appendNode(1);
    list.appendNode(3);
    list.appendNode(7);

    list.displayList();

    list.insertFront(10);
    list.insertFront(15);
    list.insertFront(25);

    list.displayList();

    list.deleteNode(2);
    list.displayList();
    list.deleteNode(15);
    list.displayList();
    list.deleteNode(12345);
    list.displayList();
    list.deleteNode(7);
    list.displayList();
    list.deleteNode(25);
    list.displayList();

    return 0;
}
```

Output

```
5 2 1 3 7
25 15 10 5 2 1 3 7
25 15 10 5 1 3 7
25 10 5 1 3 7
25 10 5 1 3 7
25 10 5 1 3
10 5 1 3
```

Solution:

```
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

#include <iostream>

using namespace std;

template <class T> class LinkedList {
  private:
    struct ListNode {
        T value;
        ListNode *next;
    };

    ListNode *head;

  public:
    LinkedList();
    ~LinkedList();

    void appendNode(T);
    void insertFront(T);
    void deleteNode(T);
    void displayList() const;
};

template <class T> LinkedList<T>::LinkedList
    () { head = nullptr; }

template <class T> LinkedList<T>::~
    LinkedList() {
    ListNode *nodePtr;
    ListNode *nextNode;

    nodePtr = head;

    while (nodePtr != nullptr) {
        nextNode = nodePtr->next;
        delete nodePtr;
        nodePtr = nextNode;
    }
}

template <class T> void LinkedList<T>::
    appendNode(T newValue) {
    ListNode *newNode;
    ListNode *nodePtr;

    newNode = new ListNode;
    newNode->value = newValue;
    newNode->next = nullptr;

    if (!head)
        head = newNode;
    else {
        nodePtr = head;

        while (nodePtr->next)
            nodePtr = nodePtr->next;

        nodePtr->next = newNode;
    }
}
```

```
template <class T> void LinkedList<T>::
    insertFront(T newValue) {
    ListNode *newNode;
    newNode = new ListNode;
    newNode->value = newValue;
    newNode->next = nullptr;

    if (!head) {
        head = newNode;
    } else {
        newNode->next = head;
        head = newNode;
    }
}

template <class T> void LinkedList<T>::
    deleteNode(T searchValue) {
    ListNode *nodePtr;
    ListNode *previousNode;

    if (!head)
        return;

    if (head->value == searchValue) {
        nodePtr = head->next;
        delete head;
        head = nodePtr;
    } else {
        nodePtr = head;

        while (nodePtr != nullptr && nodePtr
            ->value != searchValue) {
            previousNode = nodePtr;
            nodePtr = nodePtr->next;
        }

        if (nodePtr) {
            previousNode->next = nodePtr->
                next;
            delete nodePtr;
        }
    }
}

template <class T> void LinkedList<T>::
    displayList() const {
    ListNode *nodePtr;
    nodePtr = head;

    while (nodePtr) {
        cout << nodePtr->value << " ";
        nodePtr = nodePtr->next;
    }
    cout << endl;
}

#endif
```

2. Write the implementation of either the templated Quick Sort algorithm we did in class or the templated Merge Sort algorithm we did in class. Do only one of these.

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);
}
```

```
template <class T>
void merge(T A[], T Temp[], int startA, int
    startB, int end) {
  int aptr = startA;
  int bptr = startB;
  int i = startA;

  while (aptr < startB && bptr <= end)
    if (A[aptr] < A[bptr])
      Temp[i++] = A[aptr++];
    else
      Temp[i++] = A[bptr++];

  while (aptr < startB)
    Temp[i++] = A[aptr++];

  while (bptr <= end)
    Temp[i++] = A[bptr++];

  for (i = startA; i <= end; i++)
    A[i] = Temp[i];
}

template <class T> void mergeSort(T A[], T
    Temp[], int start, int end) {
  if (start < end) {
    int mid = (start + end) / 2;
    mergeSort(A, Temp, start, mid);
    mergeSort(A, Temp, mid + 1, end);
    merge(A, Temp, start, mid + 1, end);
  }
}

template <class T> void mergeSort(T A[], int
     size) {
  T *Temp = new T[size];
  mergeSort(A, Temp, 0, size - 1);
  delete[] Temp;
}
```

3. 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: 5 Points

1. Prove that $T(n) = 2^n$ is $O(n!)$ and $T(n) = n!$ is not $O(2^n)$.

   Solution:

   $T(n) = 2^n$ is $O(n!)$

   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, $2^n \leq cn!$.

   $$2^n = 2 \cdot 2 \cdot 2 \cdot 2 \cdots 2 = 2 \cdot 1 \cdot 2 \cdot 2 \cdot 2 \cdots 2 \leq 2 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdots n = 2n!$$

   So let $c = 2$ and $n_0 = 1$.

   $T(n) = n!$ is not $O(2^n)$

   We need to show that there are no constants $c > 0$ and n_0 such that, for every $n > n_0$, $|g(n)| \leq cf(n)$. Stated another way, given any constant $c > 0$ there exists an n_0 such that for every $n > n_0$, $n! > c2^n$.

   Take any constant $c > 0$ and let d be the smallest integer greater than or equal to c and at least 8, that is $d = \max(\lceil c \rceil, 8)$, and take $n_0 = d + 1$. Then

   $$\begin{aligned} n! &= 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8 \cdots n \\ &= d \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8 \cdots (d-1) \cdot (d+1) \cdots n \\ &= d \cdot 1 \cdot 2 \cdot 3 \cdot 2^2 \cdot 5 \cdot 6 \cdot 7 \cdot 2^3 \cdots (d-1) \cdot (d+1) \cdots n \quad \text{(replace all } a > 2 \text{ with 2)} \\ &> d \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdots 2 = d2^{n+1} \geq c2^n \end{aligned}$$