1. Consider the following program that sorts uses a Binary Heap.

```cpp
#include <iostream>
#include <string>
#include "binaryHeap.h"
using namespace std;

int main() {
    BinaryHeap<int> myHeap;
    int numElements = 6;
    int myArray[] = {50,10,30,60,100,15};

    cout << "Elements before sorting: ";
    for (int index = 0; index < numElements; index++) {
        cout << myArray[index] << " ";
    } // end for
    cout << endl << endl;
    for (int index = 0; index < numElements; index++) {
        myHeap.insert(myArray[index]);
    } // end for
    for (int index = 0; index < numElements; index++) {
        myArray[index] = myHeap.delMin();
    } // end for
    cout << "Elements after sorting: ";
    for (int index = 0; index < numElements; index++) {
        cout << myArray[index] << " ";
    } // end for
    cout << endl;
} // end main
```

a) Explain why the above program sorts.

b) What is the worst-case $\Theta()$ notation for the sorting program?
2. A **recursive function** is one that calls itself. The following `countDown` function is passed a starting value and proceeds to count down to zero and prints “Blast Off!!”.

```cpp
#include <iostream>
using namespace std;

// prototypes
void countDown(int count);

int main() {
    int startOfCountDown;
    cout << "Enter count down start: ";
    cin >> startOfCountDown;
    cout << endl << "Count Down: " << endl;
    countDown(startOfCountDown);
} // end main

void countDown(int count) {
    if (count == 0) {
        cout << "Blast Off!!!" << endl;
    } else {
        cout << count << endl;
        countDown(count-1);
    } // end if
} // end countDown
```

**Program Output:**

```
Enter count down start: 10
Count Down:
10
9
8
7
6
5
4
3
2
1
Blast Off!!!
```

The `countDown` function, like most recursive functions, solves a problem by splitting the problem into one or more simpler problems of the same type. For example, `countDown(10)` prints the first value (i.e., 10) and then solves the simpler problem of counting down from 9. To prevent “infinite recursion”, if-statement(s) are used to check for trivial **base case(s)** of the problem that can be solved without recursion. Here, when we reach a `countDown(0)` problem we can just print “Blast Off!!!”.

a) Trace the function call `countDown(5)` on paper by drawing the run-time stack and showing the output.

b) What do you think will happen if your call `countDown(-1)`?

c) Why is there a limit on the depth of recursion?
3. Some mathematical concepts are defined by recursive definitions. One example is the Fibonacci series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

After the second number, each number in the series is the sum of the two previous numbers. The Fibonacci series can be defined recursively as:

\[
\begin{align*}
\text{Fib}_0 &= 0 \\
\text{Fib}_1 &= 1 \\
\text{Fib}_N &= \text{Fib}_{N-1} + \text{Fib}_{N-2} \quad \text{for } N \geq 2.
\end{align*}
\]

a) Write the recursive function

b) Draw a recursion tree for \( \text{fib}(5) \).

c) On my office computer, the call to \( \text{fib}(40) \) takes 22 seconds, the call to \( \text{fib}(41) \) takes 35 seconds, and the call to \( \text{fib}(42) \) takes 56 seconds. How long would you expect \( \text{fib}(43) \) to take?

d) How long would you guess calculating \( \text{fib}(100) \) would take on my office computer?

e) Why do you suppose this recursive \( \text{fib} \) function is so slow?

f) How might we speed up the calculation of the Fibonacci series?
4. How would we write the BinaryHeap `siftUp` function recursively?

```
while currentPosition has not reached the root
    calculate the parentIndex
    if item at currentPosition < item at parentIndex then
        exchange the two item
        update the currentPosition
    else
        return since we are done sifting up
```

Algorithm for insert(T newItem)
- `heap[numItems] = newItem`
- `siftUp(numItems)`
- `numItems++`

Algorithm for the NON-RECURSIVE `siftUp(int currentPosition)`
- `while currentPosition has not reached the root`
- `calculate the parentIndex`
- `if item at currentPosition < item at parentIndex then`
  - `exchange the two item`
  - `update the currentPosition`
- `else`
  - `return since we are done sifting up`

a) What base case(s) (trival non-recursive cases) would we have?

b) What recursive case(s) would we have?

c) Write the recursive `siftUp` code.