Question 1.

Given the function \( f(n) = n \log n \), find functions \( g_i(n) \) that differ from \( f(n) \) by more than constants such that:

a. \( f(n) \) is \( O(g_0(n)) \)
b. \( f(n) \) is \( O(g_1(n)) \), \( \Omega(g_1(n)) \), and \( \Theta(g_1(n)) \)
c. \( f(n) \) is \( \omega(g_2(n)) \)
d. \( f(n) \) is \( o(g_3(n)) \) (i.e. “little o”)
e. \( f(n) \) is \( O(g_4(n)) \) and \( o(g_4(n)) \)

a. \( n^2 \)
b. \( n \log n + n \)
c. \( n \)
d. \( n^2 \)
e. \( n^2 \)
Question 2.

Consider a black box that generates integer numbers, one at a time, with the property that a new number is within 10 of the previous number. In other words,

$$|A_{i+1} - A_i| \leq 10$$, for any $$A_{i+1}$$ and $$A_i$$ generated consecutively.

Here is an example of a sequence of numbers generated by the black box:

101, 102, 107, 107, 99, 89, 89, 99, 102, 102, 102, …

Design a data structure that stores the numbers, does not waste memory space, does not store duplicates, stores the number of times a number was generated, and allows inserting a newly generated number in constant time. Your answer should include a description of the data structure, a pseudocode description of the algorithm for inserting a newly generated number, and a justification of the fact that insertion takes constant time.

class DLL {
    int val;
    int appsN;
    DLL *next, *prev;
};

DLL* Insert(int newNumber, DLL *prevInsert) {

    if (prevInsert->val == newNumber) {
        prevInsert->appsN++;
        return prevInsert;
    }

    if (newNumber < prevInsert->val) {
        while (prevInsert->prev) {
            if (prevInsert->prev->val == newNumber) {
                prevInsert->prev->appsN++;
                return prevInsert->prev;
            }
            if (prevInsert->prev->val < newNumber) {
                return InsertAfter(prevInsert->prev, newNumber);
            }
            prevInsert = prevInsert->prev;
        }
        prevInsert = prevInsert->prev;
        return InsertFirst(prevInsert, newNumber);
    }

    if (prevInsert->val < newNumber) { similar to previous case}
DLL* **InsertFirst**(DLL *oldHead, int newNumber) {

    DLL *newNode = new DLL();
    newNode->val = newNumber;
    newNode->appsN = 1;

    newNode->prev = NULL;
    newNode->next = oldHead;

    oldHead->prev = newNode;

    return newNode;
}

DLL* **InsertAfter**(DLL *afterThis, int newNumber) {

    DLL *newNode = new DLL();
    newNode->val = newNumber;
    newNode->appsN = 1;

    newNode->prev = afterThis;
    newNode->next = afterThis->next;

    if (afterThis->next)
        afterThis->next->prev = newNode;
    afterThis->next = newNode;

    return newNode;
}

**Justification**: the while loop is executed at most 10 times since each time it is executed the current number decreases/increases by at least 1 because we store unique integers, and since it has to decrease/increase by at most 10.
Question 3.

A binary tree is defined as a tree with a single node, or, a tree whose root has an ordered pair of children which are binary trees.

a. Show that, for any binary tree $T$, $e \leq 2^h$, where $e$ is the number of leaves in $T$, and $h$ is the height of $T$.

b. Give a pseudocode description of an algorithm that takes a binary tree $T$ as input and trims it down to a binary tree in which all leaves have the same depth $d$, where $d$ is the minimum depth of any leaf in $T$.

Point values for problems and pieces listed in []. Points were awarded based on correctness or at least attempting something.

a. [5]
   Induction hypothesis: $e \leq 2^h$
   [1] proof is by induction on height of tree

   base case
   [1] a binary tree of height 0 is just a single node and has $e = 1 \leq 2^d$ leaves

   induction step
   [1] by definition can create a tree by taking a node a making as its children 2 binary trees: X of height x and Y of height y. Thus our new tree $Z$ consisting of a node with X and Y as children will have height $1 + \max \{x, y\}$
   [2] by the inductive hypothesis, $e = e$ in $X + e$ in $Y \leq 2^x + 2^y \leq 2^{\max \{x, y\}} = 2^{1 + \max \{x, y\}}$

b. [5]
   [1]
   trimtomindepth(T)
   d=findd(T)
   trim(T,d,0)

   [2]
   findd(T)
   if T->left==NULL
       return 0
   ld=findd(T->left)
   rd=findd(T->right)
   if ld < rd
       return ld+1
   else
       return rd+1
trim(T, d, h)
if h == d
    delete(T->left)
    T->left = NULL
    delete(T->right)
    T->right = NULL
else
    trim(T->left, d, h + 1)
    trim(T->right, d, h + 1)

[3]

delete(T)
    if (T->left)
        delete(T->left)
    if (T->right)
        delete(T->right)