NUMBER 1, SHORT ANSWER:

A. $3N\log_3 N + 6N + 4$ (+/- 1 on each term, must be log base 3)

B. $O(N^2)$ (Even though going from back, worst case is remove first half)

C. 12 seconds (method is $O(N^2)$)

D. 17

E. 16

F. 22 seconds (Infinite loop okay, runtime error okay)

G. 19

H. 0 6 3 10 17 9

I. false (try this 60, 10, 50, 20, 40, 30)

J. 120 seconds (timing data indicates $O(N^3)$)

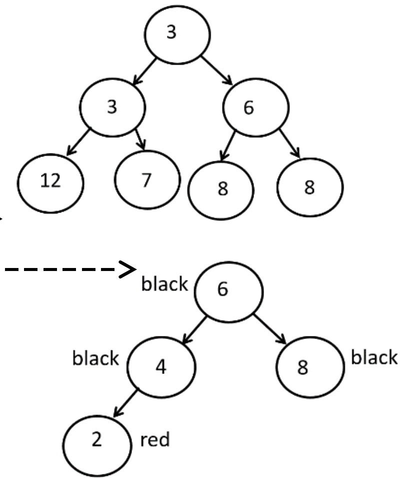
K. 8 6 4 6

L. {0=2, 1=1, 5=0}

M. 27 seconds (operation as described would be $O(N^3)$)

N. -->

O. -->



P. The chains (or buckets) will have a small number of elements. (Or words to that effect.)

Q. 25 5 37 17 8

R. merge (first half sorted, third quarter sorted

S. 3 4 2 1 2

T. 6 (D -> C -> B -> G -> F -> H)

U. E A D C G

V. The capacity of the internal array. (not the size of the hash table or number of elements. The iterator must look through all elements of the array.)

2. Suggested Solution:

```
public boolean remove(E val) {
    // determine the correct bucket
    int index = val.hashCode() % con.length;
    if (index < 0)
        index *= -1;
    boolean changed = false;
    if (con[index] != null) {
        // bucket exists
        Node<E> temp = con[index].next; // first node with data
        while (temp != null && !val.equals(temp.data))
            temp = temp.next;
        if (temp != null) {
            // found val
            changed = true;
            size--;
            removeNode(temp); // remove from bucket
            removeNode(temp.listNode); // remove from iteration list
            // all the data nodes gone from this bucket?
            if (con[index].next == null)
                con[index] = null; // don't need header node
        }
    }
    return changed;
}

private void removeNode(Node<E> n) {
    // always a prev
    n.prev.next = n.next;
    // is there a next?
    if (n.next != null)
        n.next.prev = n.prev;
}
```

19 points, Criteria:

- use hashCode and % to calculate bucket index, 3 points
- ensure index not negative, 1 point
- check that bucket not empty, 1 point
- correctly search bucket for value
  - stop if null, 2 points
  - move node reference correctly, 2 points
  - check equals correctly, 2 points (okay if assume no nulls)
- if value found:
  - decrement size, 2 points
  - remove node from bucket correctly, 2 points (-2 once if NPE on last node)
  - remove node from iteration structure correctly, 2 points
  - null out bucket if no more elements there, 1 point
- return correct result, 1 point

Other penalties:

- creating other data structures, -5
- not calling hashCode method including with (), -1
- Destroys bucket, -6
- using iterator for hash table, -6
- infinite loop, -3 to -6 depending on severity of error
- NPE, -2 to -4 depending on severity of error
- doesn't stop when answer found, -4
- scan whole array or list, -7

3. Suggested Solution:

```
public void add(E val) {
    if (first == null) {
        first = new Node<>(val, null);
    } else if (val.compareTo(first.data) >= 0) {
        // special case, val goes into new first node
        Node<E> newNode = new Node<>(val, first);
        first = newNode;
    } else {
        // general case, first not altered
        Node<E> current = first.next;
        Node<E> trailer = first;
        while (current != null && val.compareTo(current.data) < 0) {
            trailer = current;
            current = current.next;
        }
        // trailer refers to node before insert location
        Node<E> newNode = new Node<>(val, current);
        trailer.next = newNode;
    }
}
```

15 points, Criteria:

- handle case when list empty, 2 points
- handle case when element goes in new first node, but list not empty, 2 points
- correct check on loop for current null if trailer or temp.next != null if look ahead, 2 points
- correctly check value being added is less than value in current node, 3 points (okay if equal elements put after instead of in front)
- correctly move current, 3 points
- correctly create new Node with data and proper next, 1 point
- correctly link old nodes to new node, 2 points

Other deductions:

- call any Node constructor besides given, -2
- worse than O(N), -4
- destroy list, remove nodes, -5
- use any methods besides compareTo, -2 to -6 depending on severity
- OBOE, -2
- NPE -4
- use prev, -3
- add multiple times, -4
- recursion, O(N) space, -3
- not using new keyword correctly, -1
- adding at first.next when not empty and new node is the first node, -3

4. Suggested solution:

```
public int removeOddLeaves() {
    int oldSize = size;
    root = help(root, 0);
    return oldSize - size;
}

private BNode help(BNode n, int depth) {
    if (n != null) {
        // System.out.println(depth + " " + n.data);
        if (depth % 2 == 1 && ((n.left == null && n.right == null))) {
            n = null;
            size--;
        } else {
            n.left = help(n.left, depth + 1);
            n.right = help(n.right, depth + 1);
        }
    }
    return n;
}
```

15 points, Criteria:

- add helper, 1 point
- helper has 2 or fewer parameters (not sure how you could get less than 2 given restrictions), 2 points
- base case of node not null (or use look ahead correctly), 2 points
- base case of leaf node at an odd depth, node removed correctly
  (return null or using look ahead null child) 2 points
- decrement size when removing nodes, 2 points
- recursive case called if not at base case (even if at odd depth, careful with logic), 2 points
- recursive correctly increment depth and checks left and right side, 3 points
- correctly return number of nodes removed, 1 point

Other:

- Null Pointer Exception, -4
- over counting nodes removed
- early return, not going to all nodes -5
- use any methods, -2 to -6 depending on severity
- adding class or instance vars to BinaryTree or BNode, -5
- using array, -4 (even length 1)
- no return, -4
- empty tree, NPE -4

5. Suggested Solution:

```
private double helper(String start, Vertex curr,
        double currentPathLen) {
    if (start.equals(curr.name) && currentPathLen != 0) {
        return currentPathLen;
    } else if (curr.scratch == 1) {
        return INFINITY; // going in circles, but not back to start
    }
    // let's deal with the current Vertex
    curr.scratch = 1;
    double shortestFromHere = INFINITY;
    currentPathLen++;
    // choices are where to go next
    for (Edge e : curr.adjacent) {
        double temp = helper(start, e.dest, currentPathLen);
        if (temp < shortestFromHere) {
            shortestFromHere = temp;
        }
    }
    // in case this Vertex is part of a shorter cycle
    curr.scratch = 0;
    return shortestFromHere;
}
```

16 points, Criteria:

- cycle found base case, at start vertex and path length > 0, 2 points
- failure base case, if scratch variable non-zero, we are in cycle,
  but not at desired end vertex, return INFINITY, 2 points
- recursive case
  - mark scratch variable, 1 point
  - local variable for shortest cycle through this node, initialized to INFINITY, 2 points
  - loop through current Vertex edges, 1 point
  - recursive call correct and store result, 3 points
  - compare to best so far and update if better, 2 points
  - set scratch back to zero in case shorter cycles exists through current, 2 points
  - return statement correct, 1 point

Other:

- early return, -5
- infinite loops, -4
- altering anything besides scratch variable of vertices.
- okay to return early if best is 2 (or even 1)
- add another helper, -5
- == on Strings -2

6. Comments:

```java
public static int[] maxImprove(int[] houses) {
    int[] best = new int[houses.length];
    best[0] = houses[0];
    if (houses.length > 1) {
        best[1] = (houses[1] > houses[0]) ? houses[1] : houses[0];
        for (int i = 2; i < houses.length; i++) {
            // options: If I don't imrpove this house, then
            // the best I can do is the previous value.
            // If I do imrpove this house, total improvement is this
            // house improvement value plus best answer from house
            // two spots previous
            int dontImproveThisHosue = best[i - 1];
            int improveThisHouse = houses[i] + best[i - 2];
            best[i] = (dontImproveThisHosue > improveThisHouse)
                    ? dontImproveThisHosue : improveThisHouse;
        }
    }
    return best;
}
```

13 points, Criteria:

- create array of ints equal in length to parameter, 1 point
- initialize local array[0] to houses[0], 2 points
- check more than 1 house, 1 point
- set second optimal result correctly, 2 points
- loop through rest of houses, 1 point
- calcualte previous optimal result [i - 1] and this house value + result[i - 2] (optimal result for house 2 spots previous.) 2 points
- correctly compare the two possible results and set current optimal value to max, 3 points
- return correct array, 1 point

Other:

- AIOBE, -4
- using new as identifier , -1
- not including size of new array, new Object[], -2
- alter size -2
- call resize -5
- add duplicates -4
- assume >1 element, -1
- > O(N) -5
-