| Points off | 1 | 2 | 3 | 4 | 5 | 6 | Total off | Net Score |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |

# CS 314 – Final Exam – Fall 2019

Your Name: _____

Your UTEID: _____

Instructions:

1. There are **6** questions on this test. 100 points available. Scores will be scaled to 300 points.
2. You have 3 hours to complete the test.
3. Place your final answers on this test. Not on the scratch paper. **Answer in pencil**.
4. You may not receive any outside assistance. No electronic devices or calculators may be used.
5. When answering coding questions, ensure you follow the restrictions of the question.
6. Do not write code to check the preconditions.
7. On coding questions, you may implement your own helper methods unless disallowed by the question.
8. On coding questions make your solutions as efficient as possible given the restrictions of the question.
9. Test proctors will not answer any questions regarding the content of the exam. If you think a question is ambiguous or has an error, state your assumptions and answer based on those assumptions.
10. When you complete the test show the proctor your UTID, give them the test and all the scratch paper, used or not, and leave the room quietly.

1. (1 point each, 22 points total) Short answer. Place your answer on the line next to or under the question. Assume all necessary imports have been made.

   a. If a question contains a syntax error or compile error, answer **compile error**.
   b. If a question would result in a runtime error or exception, answer **runtime error**.
   c. If a question results in an infinite loop, answer **infinite loop**.
   d. Assume $\log_2(1{,}000) = 10$ and $\log_2(1{,}000{,}000) = 20$.

A. Using the techniques developed in lecture, what is the T(N) of the following method? N = **n** _____

```
public static double a(int n, int div) { // pre: div != 0
     int t = 0;
     for (int i = 0; i < n; i++) {
          int val = i;
          for (int j = 1; j <= n; j *= 3) {
               t += (i * j) / div;
          }
          t += val / div;
     }
     return t;
}
```

B. What is the worst case order of method **b**? N = **list.size()**

_____

```
public static int b(ArrayList<Integer> list, int tgt) {
     int res = 0;
     for (int i = list.size() - 1; i >= 0; i--) {
          if (list.get(i) < tgt) {
               res += list.remove(i);
          } else {
               res += list.get(i);
          }
     }
     return res;
}
```

C. The following method takes 3 seconds to complete when **list.size()** is 20,000.
What is the expected time for the method to complete when **list.size()** is 40,000?
Assume in each case that approximately 35% of the elements in the list have a length less than 10.

_____

```
private static void c(LinkedList<String> list) {
     int i = 0;
     while (i < list.size()) {
          if (list.get(i) != null) {
               if (list.get(i).length() < 10) {
                    list.remove(i);
               } else {
                    i++;
               }
          } else {
               i++;
          }
     }
}
```

D. What is returned by the method call **d(12, 1)**? _____

```
public static int d(int x, int y) {
     if (x <= 0)
          return 2;
     else {
          int r = d(x - y, y * 2);
          r += y;
          return r;
     }
}
```

E. What is the minimum number of nodes in a complete binary tree with a height of 4?

_____

F. The following method takes 5 seconds to complete when **n** = 1,000,000. What is the expected time for the method to complete when **n** = 4,000,000? **TreeSet** is the **java.util.TreeSet** class.

_____

```
public static TreeSet<Integer> f(int n) {
     TreeSet<Integer> t = new TreeSet<>();
     for (int i = n; i > 0; i++) {
          t.add(i * 10);
     }
     return t;
}
```

G. What is returned by the method call **g("longhorns")**? _____

```
public static int g(String s) {
     if (s.length() <= 4)
          return 3;
     else
          return g(s.substring(2)) + g(s.substring(4)) + 1;
}
```

H. The following values are inserted in the order shown to an initially empty binary search tree use the simple insertion algorithm demonstrated in lecture.
What is a post order traversal of the resulting tree?

**9, 3, 0, 3, 6, 17, 10, 6** _____

I. True or false. Given a binary search tree that uses the simple insertion algorithm demonstrated in lecture, the worst order performance for inserting N elements only occurs when the elements are inserted in ascending or descending sorted order.

_____

J. Consider the following times it takes for a method to complete with the given amount of data.

| Amount of Data | Time to complete |
|---|---|
| 1,000 | 0.004 seconds |
| 2,000 | 0.036 seconds |
| 4,000 | 0.260 seconds |
| 8,000 | 1.9 seconds |
| 16,000 | 15 seconds |

What is the best estimate for the time it will take the method to complete with 32,000 elements of data?

_____

K. What is output by the following code? The **`Stack`** class is the one developed in lecture.

_____

```
Stack<Integer> st = new Stack<>();
int[] data = {-5, 3, 2, -4, 3, 1, -6, 4};
for (int x : data)
     if (x * 2 > 2)
          st.push(x * 2);
while (!st.isEmpty())
     System.out.print(st.pop() + " ");
```

L. What is output by the following code? _____

Recall the `toString` of Maps:
`{key1=value1, key2=value2, ... keyN=valueN}`

```
int[] vals = {5, 1, 0, 5, 0, 5, 1};
TreeMap<Integer, Integer> tm = new TreeMap<>();
for (int i = vals.length - 1; i >= 0; i--)
     tm.put(vals[i], i);
System.out.print(tm);
```

M. Assume a weighted, directed graph uses an adjacency matrix with no extra capacity as its internal storage container. The following code takes 1 second to complete when the number of distinct vertices in the edge list is 2,000. What is the expected time for the method to complete when the number of distinct vertices in the edge list is 6,000?

```
// Assume this edge class stores source, destination, and cost
public static Graph create(ArrayList<Edge> edges) {
     Graph g = new Graph();
     for (Edge e : edges)
          g.addEdge(e.soruce(), e.destination(), e.cost());
     return g;
}
```

N. The following values are added one at a time in the order shown to an initially empty min heap using the algorithm shown in lecture. Draw the resulting min heap

**`3, 12, 8, 3, 7, 8, 6`**

O. The following values are added one at a time in the order shown into an initially empty red black tree. Draw the resulting tree and label the color of all nodes.

**8, 6, 4, 2**

P. How is it possible that a hash table that uses chaining (or buckets) to resolve collisions could maintain O(1) performance for its add method when its load factor is set to 1.0?

---

Q. Assume the hash table below uses open addressing, linear probing to resolve collisions, a load factor of 0.75, determines position in the array container by finding the remainder of the hash code of an element when dividing by the length of the array container, and an initial capacity of 10. Assume the **Integer** class returns its int values as its hash code.
So for example, **new Integer(37).hashCode()** returns **37**.
What is output by the following code?

```
int[] nums = {37, 17, 8, 25, 5};
Hashtable<Integer> ht = new Hashtable<>();
for (int x : nums)
     ht.add(x);
System.out.print(ht);
```

R. Consider the following data. Of the sorting algorithms we studied (selection, insertion, radix, quick, merge), which one is most likely being used given the original ordering off the data and the ordering of the data shown at an intermediary step in the sorting process.

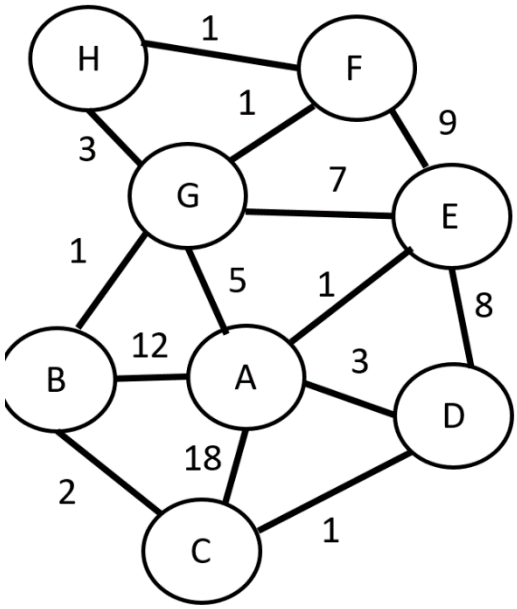
```
original data:       [99 40 38 92 53 86 80 33 24 85 37 51]

intermediary step:   [38 40 53 86 92 99 24 33 80 85 37 51]
```

S. What is output by the following code?

```
IntStream.of(12, 17, 8, 44, 5, 8, 46)
     .map(y -> y / 4)
     .filter(x -> x < 10)
     .forEach(z -> System.out.print(z + " "));
```

T. In the graph to the right what is the cost of the lowest cost path from vertex D to vertex H?




_____

U. Assume we perform Dykstra's algorithm for shortest path in the graph to the right starting at vertex E and finding the shortest path to all vertices in the graph. Obviously the vertex E will be the first one marked as visited. What are the next 4 vertices in the graph marked as visited and what order are they marked?

**E** _____

V. The following method will run in O(N) time. What does N represent?

_____

```
public static int print(HashSet<Integer> set) {
      int total = 0;
      Iterator<Integer> it = set.iterator();
      while (it.hasNext()) {
            total += it.next();
      }
      return total;
}
```

**2. Hash tables and linked lists. (19 points)** The Java standard library includes a class name **`LinkedHashSet`**. The purpose of this class is to allow the typical O(1) operations of hash tables, but when iterating through the elements of the set, return them in the order they were added as opposed to the order they may appear in the internal array. Implement the **`remove(E val)`** instance method for the **`LinkedHashSet`** below.

Consider the representation of the **`LinkedHashSet`** below.

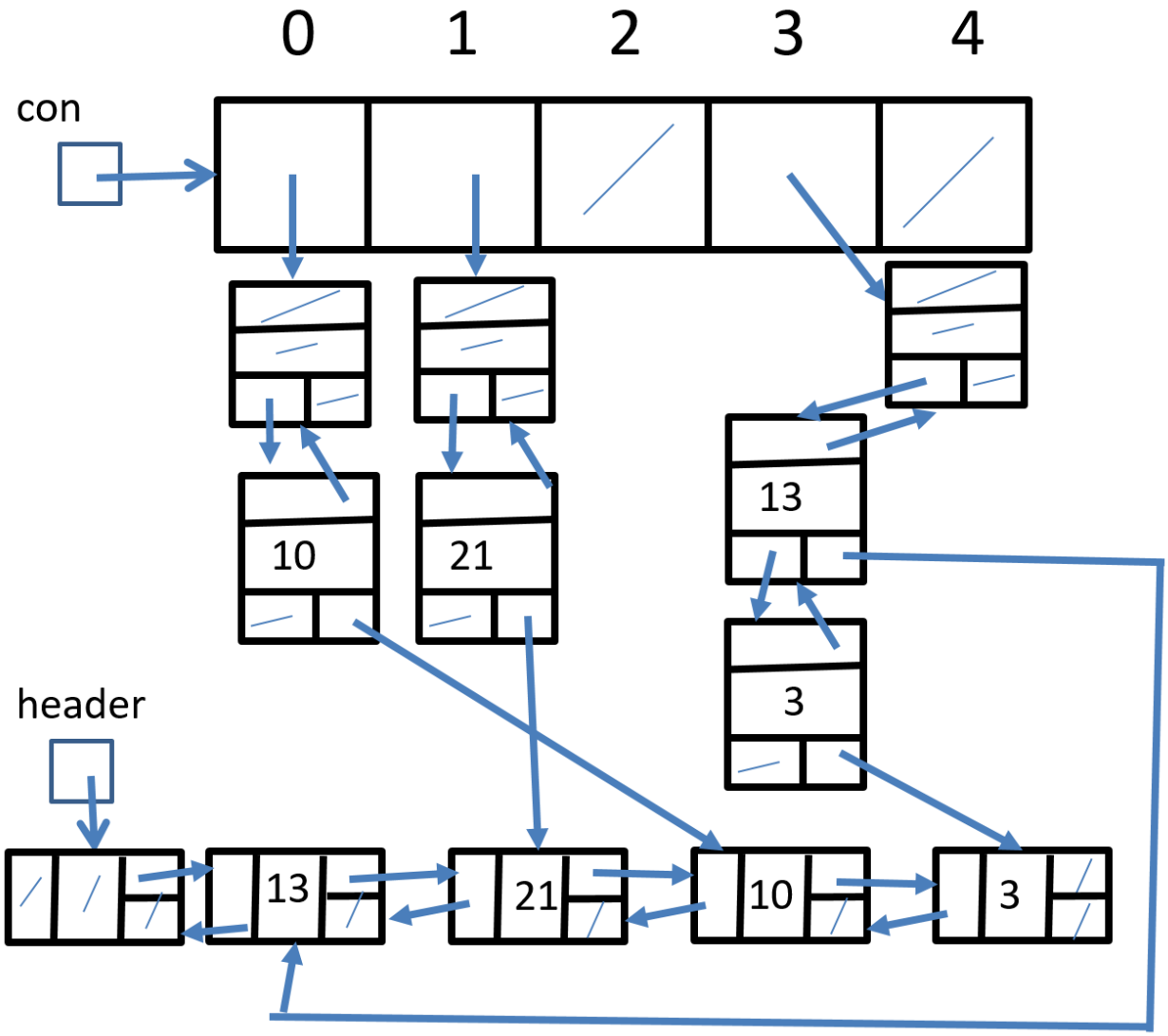
The nodes are doubly linked and there is a header node. Slashes indicate object variables that store **`null`**.

Data elements are shown in the nodes for clarity although there would actually be references to the objects instead.

For the nodes in the **`con`** array the bottom left reference is the **`next`** reference and the bottom right reference is the **`listNode`** reference.

For nodes in the list used for iteration the top right reference is the **`next`** reference and the bottom right reference is the **`listNode`** reference.

The elements were added in the following order: **`13  21  10  3`**



Here is the **LinkedHashSet** for this question:

```
public class LinkedHashSet<E> {

     private Node<E>[] con; // stores the buckets

     private int size; // number of data elements in this hash set

     // refers to the header node in the linked structure of nodes used
     // for iterating through the set's data
     private final Node<E> header = new Node<>();

     // other instance variables and methods not shown

     private static class Node<E> {
          private E data;
          private Node<E> prev;
          private Node<E> next;

          /* If this node is in con, then this refers to the Node in
             the list that contains the corresponding piece of data.
             If this node is in the list, then this is set to null. */
          private Node<E> listNode;
     }
}
```

The buckets (linked structure of **Node**s) in the **con** array have a header node if the bucket is not empty. If there are no elements at a given index in **con**, then that element in **con** is set to **null**.

The linked structure of **Node**s that stores the elements in the order to iterate through has a header node.

The header nodes do not store actual data and always have their data reference set to **null**.

Neither the buckets in the **con** array or the linked structure used for iteration are circular. The last node in any linked structure has its **next** reference set to **null**.

Complete the **remove(E val)** instance method for the **LinkedHashSet**.

You may use the **hashCode** and **equals** methods on **Object**s and the nested **Node** class.

**Do not use any other Java classes or methods. Do not use any other methods from the LinkedHashSet class unless you implement them yourself as a part of your answer.**

**Complete the method on the next page.**

```
/* pre: val != null
post: val removed from this linked hash set if present. Returns true if this
linked hash set was altered as a result of this call, false otherwise. */
public boolean remove(E val) {
```

**3**. **Linked Lists (15 points)** - Complete the **`add`** instance method for the **`SortedLinkedList`** class.
The method adds a given value to the list.

- **You may not use any other methods in the `SortedLinkedList` class unless you implement them yourself as a part of your solution.**
- The **`SortedLinkedList`** stores the elements in the list in a linked structure of nodes and stores the values in **descending** order.
- The **`SortedLinkedList`** class uses singly linked nodes.
- **`first`** refers to the first node in the linked structure of nodes.
- If the list is empty, **`first`** stores **`null`**.
- The last node in the chain of nodes has its **`next`** reference set to **`null`**.

```
public class SortedLinkedList <E extends Comparable<? super E>> {
      // refers to first node in linked structure of nodes
      private Node<E> first;

      // no other instance variables

      // The nested Node class.
      private static class Node<E extends Comparable<? super E>> {
            private E data;
            private Node<E> next;

            private Node(E data, Node<E> next) {
                  this.data = data;
                  this.next = next;
            }
      }
}
```

```
[].add(12) list becomes [12]

[12].add(12) list becomes [12, 12]

[12].add(-5) list becomes [12, -5]

[12, -5].add(35) list becomes [35, 12, -5]

[35, 12, -5].add(17) list becomes [35, 17, 12, -5]

[35, 17, 12, -5].add(35) list becomes [35, 35, 17, 12, -5]

[35, 35, 17, 12, -5].add(-10) list becomes [35, 35, 17, 12, -5, -10]
```

**Do no use any other Java classes or methods besides the nested `Node` class and the `compareTo` method.**

**Do assume or use any other methods from the `SortedLinkedList` unless you implement them as a part of your answer.**

**Your method must be O(1) *space* .**

**Complete the method on the next page.**

```
/* pre: val != null
   post: Per the problem description.*/
public void add(E val) {
```

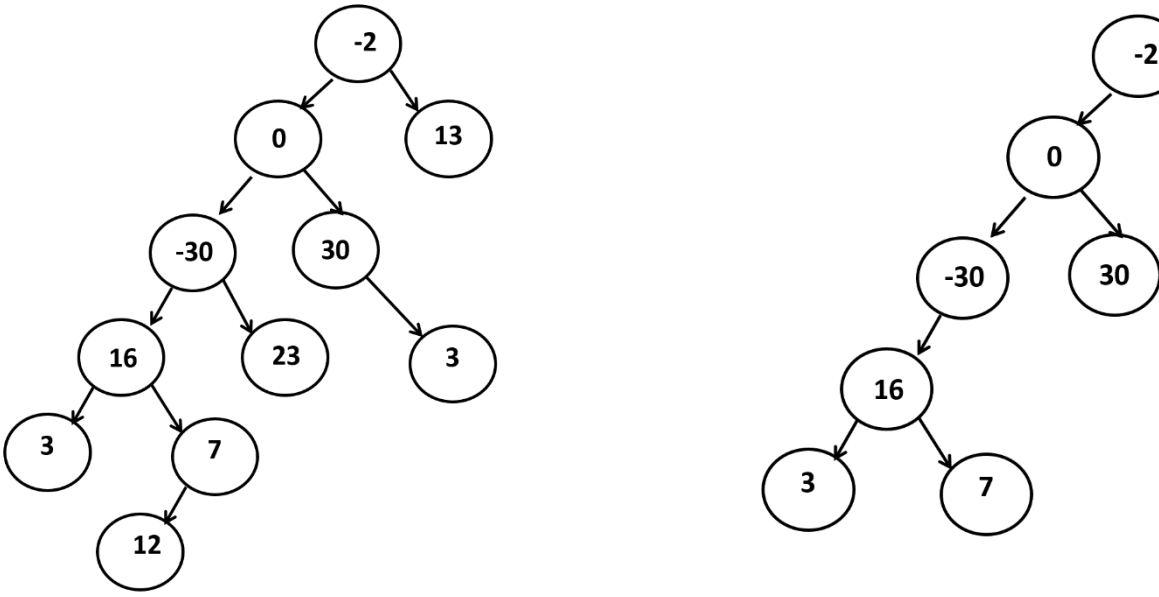
**4. Binary Trees - (15 points.)** Implement an instance method for a binary tree class that removes any leaves from the tree that have an odd depth.

```
public class BinaryTree {

     private BNode root; // root == null if size == 0
     private int size; // number of nodes in this tree

     // nested BNode class
     private static class BNode {
          private int data;
          // left and right child are null if no child on that side
          private BNode left;
          private BNode right;
     }
}
```

Consider the tree `t` on the left. This is not a binary search tree. If we remove all leaf nodes that at are at at odd depth the resulting tree is shown on to the right.




**BEFORE** **AFTER**

**The method updates the `size` field for the `BinaryTree` class and returns the number of nodes removed by the method.**

**Do not create any new nodes or other data structures.**

**You may use the `BNode` class, but do not use any other methods or classes. This includes arrays.**

**You may NOT add any class or instance variables to the `BinaryTree` or `BNode` classes.**

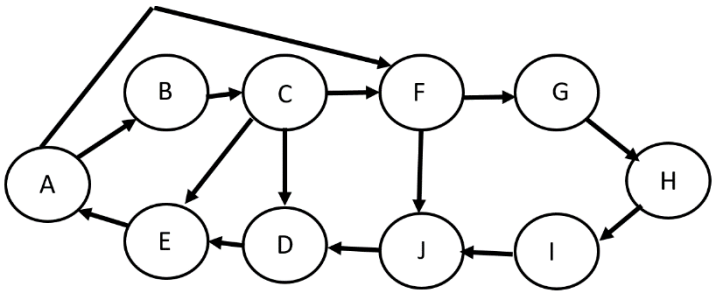
**Part of the question grade (2 points) will be based on the clarity and conciseness of your solution.**

```
// pre: none, post per the problem description.
public int removeOddLeaves() {
```

**5. Graphs (16 Points)** Complete an instance method for the **`Graph`** class that uses **recursive backtracking** to determine the length (number of edges) in the **shortest cycle** a given vertex is a part of. The graphs for this question are directed, but unweighted. All costs are simply set to 1.

Recall, a cycle is a series of unique edges that form a path that start and end at the same vertex.

If the given vertex is not part of any cycles, the method returns the **`Graph`** class's constant for **`INFINITY`**.




Consider the graph to the right. If the given vertex is A, there are several cycles that start and stop at the vertex including:

```
A, B, C, E, A
A, B, C, D, E, A
A, F, J, D, E, A
A, B, C, F, J, D, E, A
```

The shortest of these cycles contains 4 edges, **`A -> B -> C -> E -> A, returns 4.0`**

Recall the **`Graph`**, **`Vertex`**, and **`Edge`** class. Recall all **`Edge cost`**s are set to 1 for this graph.

```
public class Graph {
    // The vertices in the graph.
    private Map<String, Vertex> vertices;
    private static final double INFINITY = Double.MAX_VALUE;

      // for all vertices, set scratch to 0 and prev to null.
    private void clearAll()

    private static class Vertex {
        private String name;
        private List<Edge> adjacent;
        private int scratch;
        private Vertex prev; }

    private static class Edge {
        private Vertex dest;
        private double cost;
    }
}
```

You may use the **`get`** and **`size`** methods from the **`List`** interface.
You may use for-each loops.
You may use the **`equals`** method for **`String`**s.
You may use the nested **`Vertex`** and **`Edge`** classes.

Do not alter the **`Graph`** or its elements in any way other than changing the **`scratch`** instance variable of **`Vertex`** objects.

**Do not create ANY additional data structures.**

**You must implement a recursive backtracking solution in the helper method.**
**For this question only do not add any other helper methods**.

```
/* pre: start != null, start is present in this Graph.
   post: per the problem description. */
public double shortestCycle(String start, String dest) {
     clearAll();
     return helper(start, vertices.get(start), 0);
}

// This question only, do not add any other helper methods.
private double helper(String start, Vertex curr, int currentPathLen) {
```

**6. Dynamic Programming (13 Points)** Write a method that determines which houses on a block to improve in order to maximize the total amount of improvement made to the block of houses.
**Due to zoning restriction we cannot improve houses that are adjacent to each other.**

The input to the method is an array of **`int`**s with a **`length`** >= **1**. The array represents the houses on the block and each element represents the amount of value we generate by improving the house at that position on the street. Each improvement takes the same amount of work but can produce different amounts of value.

For example, if we have the array: **`[5, 10, 6]`** we have a simple block with three houses. Improving the first house results in an added value of 5, improving the second house results in an added value of 10, and improving the third house results in an added value of 6.

Recall, due to zoning restrictions we cannot improve houses next to each other. So if we improve the second house we cannot improve the first or third houses. The optimal answer in this case is to improve the first and third houses for a total value of 11.

Here is another example: **`[5, 4, 5, 7]`** In this case the optimal answer is to improve the first and last houses for a total value of 12 (5 + 7).

Your method shall use the dynamic programming technique as presented in lecture.

Create and return a single array of **`int`**s. At each element i, the array stores the optimal answer if the block only consisted of the houses from index 0 to index i inclusive. The last element stores the actual optimal value for the given block of houses.

Here are examples of calls to the method **`maxImprove(int[] houses)`** and the expected result. The houses that would actually be improved are underlined for clarity.

**`maxImprove([5]) -> [5]`**

**`maxImprove([5, 10]) -> [5, 10]`**

**`maxImprove([10, 5]) -> [10, 10]`**

**`maxImprove([5, 10, 6]) -> [5, 10, 11]`**

**`maxImprove([5, 15, 6]) -> [5, 15, 15]`**

**`maxImprove([5, 4, 5, 7]) -> [5, 5, 10, 12]`**

**`maxImprove([5, 3, 4, 11, 2]) -> [5, 5, 9, 16, 16]`**

**`maxImprove([6, 8, 1, 3, 8, 2, 4]) -> [6, 8, 8, 11, 16, 16, 20]`**

**Do not use any other Java or classes besides the single array of `int`s you create and return.**

**Complete the method on the next page.**

```
/* pre: houses != null, houses.length >= 1
   post: per the problem description */
public static int[] maxImprove(int[] houses) {
```