Name: _____

USC Student ID Number: _____

USC NetID (e.g., *ttrojan*): _____

CS 455 Midterm 2
Fall 2018 [Bono]

Nov. 6, 2018

There are 6 problems on the exam, with 70 points total available. There are 10 pages to the exam (5 pages double-sided), including this one; make sure you have all of them. If you need additional space to write any answers or for scratch work, pages 8, 9, and 10 of the exam are left blank for that purpose. If you use these pages for answers you just need to direct us to look there. ***Do not detach any pages from this exam.***

Note: if you give multiple answers for a problem, we will only grade the first one. Avoid this issue by labeling and circling your final answers and crossing out any other answers you changed your mind about (though it's fine if you show your work).

Put your name, ID number, and USC username (a.k.a., NetID) at the top of the exam. Also, put your NetID at the top right of the front side of each page of the exam. Please read over the whole test before beginning. Good luck!

Java `Set<ElmtType>` Interface

The classes that implement this interface are: **`TreeSet`** and **`HashSet`**.

Selected methods:

`boolean contains(ElmtType elmt)`	Returns true iff `elmt` is in the set
`int size()`	Returns number of elements in the set
`boolean add(ElmtType elmt)`	Ensures that `elmt` is in the set. Returns true iff the set changed as a result of this call
`boolean remove(ElmtType elmt)`	Removes `elmt` from the set. Returns true iff the set changed as a result of this call
`boolean isEmpty()`	Returns true iff the set contains no elements.
`Iterator<ElmtType> iterator()`	Returns an iterator over the elements in the set.

Java `Iterator<ElmtType>` Interface

Selected methods:

`boolean hasNext()`

Returns `true` iff the iteration has more elements.

`ElmtType next()`

Returns the next element in the iteration. Each successive call returns a different element in the underlying collection.

Problem 1 [4 points]

Divide-and-conquer algorithms are ones where part of the solution to a larger version of a problem involves dividing the problem in half (roughly) and solving that smaller version as part of solving the larger problem. Which of the following algorithms listed below are divide-and-conquer algorithms (circle the letter to the left of all that apply):

a. linear search

b. mergesort

c. recursive isPalindrome (reminder: the method tells you whether its string argument is a word that reads the same forward and backward)

d. binary seach

e. insertion sort

f. the merge algorithm (reminder: an efficient algorithm we discussed that takes two ordered lists, and combines them to create one ordered list)

g. flood-fill (reminder: was used in minesweeper program for opening empty areas automatically)

Problem 2 [8 points]

Consider a `Map` class to store a collection of key-value pairs (no duplicate keys) with operations `insert(key)`, `remove(key)`, `lookup(key)`, and `printInOrder` (the last is to print all entries in order by key). For A-D assume a Map with *n* entries, and give worst case time, unless average case is better.

A. How long would `printInOrder` take in big-O terms if the Map used an *unordered array* representation?

O(nlgn)

B. How long would `remove` take if it used a *balanced search tree* representation?

O(lgn)

C. How long would `lookup` take if it used an *ordered array* representation (i.e., ordered by keys)?

O(lgn)

D. How long would `insert` take if it used an *unordered linked list* representation?

O(n)

USC NetID: _____

Problem 3 [6 points]

Consider the following Java program and suppose we are currently executing at the point labeled ****. **Show *two* different possible contents of the run-time stack at this point** (Note there are more than two correct answers). **Make it clear which direction the stack is going by labeling which end is the top of each stack you draw.** You should denote the element for one function activation with the name of that function – you do not need to show parameters or any other details.

```
public class MergeSorter {

   public static void main(String[] args) {
      . . .
      mergesort(a);
      . . .
   }
   public static void mergesort(int[] a)
      if (a.length > 1) {
         int[] first = copyArr(a, 0, a.length / 2);
         int[] second = copyArr(a, a.length / 2, a.length);
         mergesort(first);
         mergesort(second);
         merge(first, second, a);
      }
   }

   private static int[] copyArr(int[] orig, int startLoc, int end) {
      . . .
   }

   private static void merge(int[] first, int[] second, int[] a) {
      // ****
      . . .
   }
}
```

main()
mergeSort()
merge()

main()
mergeSore()
mergeSore()
mergeSore()
merge()

top

Problem 4 [24 points total]

Part A (18). One way to represent the set of locations of mines in a minefield for the mine sweeper game would be with a two-dimensional array of booleans. **Another way to represent this same data, taking less space for large sparse minefields, would be to store them in a Java `Set`. In this problem you are going to write *part* of the implementation of the `MineField` class using this `Set` representation.** Your answer will consist of (1) a definition of the instance variable `mines`; (2) the implementation of the `initMines` private method (called from the constructor below), (3) the implementation of the `hasMine` public method; (4) any other necessary code for this to work (put any such code after the rest of your answer).

Notes/Hints:

- **To make this problem easier, you do not have to define `equals`, `hashCode`, or implement `Comparable`, or `Comparator` (although you might need one or more of these in a working version of this code.)**
- besides the ones explicitly mentioned, do not implement any of the other public methods or instance variables of the MineField class from assignment 3.
- See the front page of the exam for more about the Java `Set` interface.
- Hint: the number of rows in a 2D array, `a`, is `a.length` ; the number of columns is `a[0].length`

```
public class MineField {
   // define your mines variable here:  (1)
```

Sets < String > mines;

```
   . . .  // other constants and instance variables not shown

   /**
      Create a minefield with same dimensions as the given array,
      and populate it with the mines in the array such that if
      mineData[row][col] is true, then hasMine(row,col) will be true
      and vice versa.  . . . (other details left out)
      @param mineData  the data for the mines;
                       must have at least one row and one col.
    */
   public MineField(boolean[][] mineData) {
      initMines(mineData);
      . . . [code to initialize the other instance variables not shown]
   }
```

(problem continued on the next page)

Problem 4 (cont.)

(continuation of MineField class definition from previous page)

```
   /**
      Initializes the mines variable from mineData such that if
      mineData[row][col] is true, then hasMine(row,col) will be true
      and vice versa.
      @param mineData the mine data; must have at least one row and one col.
   */
   private void initMines(boolean[][] mineData)  { // (2)
            mines = new HashMap<String>();
            for (int i=0 ; i < mineData.length ; i++ ) {
                  for (int j=0; j <mineData[0].length ;j++ ){
                        if( mineData[i][j] == true ) {
                              String key = Integer.toString(i) + " " + Integer.toString(j);
                              mines.add(key);
                        }
                  }
            )
   /**
      Returns whether there is a mine in this square
      @param row  row of the location to check
      @param col  column of the location to check
      @return whether there is a mine in this square
      PRE: inRange(row, col)
   */
   public boolean hasMine(int row, int col) { // (3)
            String key = Integer.toString(row) + " " + Integer.toString(col);
            return   mines.contains(key);
   }

   // (4) any additional code
}
```

Problem 4 (cont.)

For parts B and C, assume the mine field is square, with side-length *n*, and number of mines *m*.

Part B (2). What's the worst case big-O time to do `hasMine` using the 2D array representation?

O(1)

Part C (2) . What's the worst case big-O time to do `hasMine` using the Java `Set` representation from part A?

O(1)

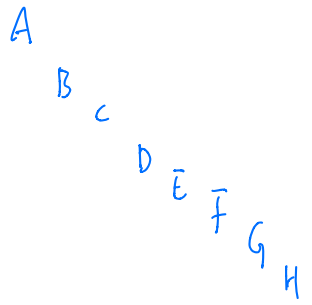
Part D (2). Suppose you implemented the rest of the changes to the `MineField` class (i.e., reimplemented the other methods not shown here) with the new representation from Part A in your own program. Briefly describe how your `VisibleField` class implementation would have to change in that program. (limit your response to one sentence).

No change.

Problem 5 [8 pts]

Consider the following sorted array of strings. **Draw the binary search tree of the same values that would involve the equivalent sequence of comparisons for any target key as binary search in this array would.** E.g., if doing binary search on the target key, "M", involved comparing with keys "A", "B", "C", and then "D" in the array, then doing binary search on the same target using your BST would involve comparing with exactly the same keys in the same order.

0	1	2	3	4	5	6	7
"A"	"B"	"C"	"D"	"E"	"F"	"G"	"H"



Problem 6 [20 points]

Write a *recursive* method `isUpDown` which, when given an array of integers, `a`, returns `true` if `a` has the structure described below, and `false` otherwise:

> **It starts with a 1, then is followed by values that each increase by 1, with the largest value in the middle, and then it has values that decrease by 1, ending up with a 1 at the last position (so, the contents read the same forwards and backwards). When there are an even number of values, the two middle values will be the same.**

A solution that doesn't use recursion will receive little to no credit. For full credit, your function must do roughly `(a.length/2)` recursive calls total. Note: you may create a helper method that does the actual recursion. Examples:

parameter `a`	return value of `isUpDown(a)`	parameter `a`	return value of `isUpDown(a)`
`[1]`	`true`	`[2, 3, 4, 3, 2]`	`false`
`[1, 1]`	`true`	`[1, 2, 2, 2, 1]`	`false`
`[1, 2, 1]`	`true`	`[1, 2]`	`false`
`[1, 2, 2, 1]`	`true`	`[1, 2, 3]`	`false`
`[1, 2, 3, 2, 1]`	`true`	`[1, 3, 5, 3, 1]`	`false`

```
// PRE: a.length >= 1
boolean isUpDown(int[] a) {
    int end = a.length-1;
    return R(a, 0, end);
}
private boolean R(int[]a, int start, int end){
    if (start >= end-1) {
        if (a[start] == a[end]) {
            return true;
        }
        return false;
    }
    if (a[start] == a[start+1]-1 && a[end] == a[end-1]-1) { return false; }
    Return R(a, start+1, end-1);
}
```

s end
0 0 0 0

S = end-1
S = end
S > end
S > = end-1

0 0
start end

Extra space for answers or scratch work. (DO NOT detach this page.)

If you put any of your answers here, please write a note on the question page directing us to look here. Also label any such answers here with the question number and part, and circle the answer.

Extra space for answers or scratch work (cont.) (DO NOT detach this page.)

If you put any of your answers here, please write a note on the question page directing us to look here. Also label any such answers here with the question number and part, and circle the answer.

Extra space for answers or scratch work (cont.) (DO NOT detach this page.)

If you put any of your answers here, please write a note on the question page directing us to look here. Also label any such answers here with the question number and part, and circle the answer.