University of Illinois at Urbana-Champaign
Department of Computer Science

Final Examination (Solution)

1. Concepts – 10 points (2 points each)

1. Carefully analyze the following mystery program. What is the last value to be printed?

Note: The Conflict exam had a different algorithm.

```
public class Mystery {
  public static void main(String[] arg) {
       mystery(1);
  }
  public static void mystery(int v)
       System.out.println(v);
       if(v == 4) return;
       if(v < 4) mystery(7-v);
       else mystery(9-v);
       System.out.println(-v);
  }
}
```

Your answer: **-1**

(This q is easier than you might think... The last value to be printed will be after the recursive call returns; so the parameter value will be the value of the very first activation)

2. I measure the running time of particular sorting algorithm for different sizes of arrays of randomly ordered integers. I can approximate the running time with the following formula (N is the number of integers to sort).
time (milliseconds) = $0.12 + 0.03\ N + 0.24\ N^2$

Suggest one sorting algorithm discussed in CS125 that is consistent with the above formula.

Your answer: **Selection (Insertion) sort**

(Any sorting algorithm that is $O(N^2)$ for random data. Thus Quicksort and Mergesort are incorrect)

3. Which one of the following must be true if the statement " `this.foo++;` " is part of a valid Java program?

> The above code must be inside an instance method.

4. Which data structure was used to hold Robots of Challenge7-RecursiveKnight? Circle the correct answer.

Red-black tree	Skip-list Hash		**(Linked-list)**
Priority queue	Map	FIFO queue	Tree

5. Circle the best response below to complete the following: When a software engineer has finished developing and testing the code on their local machine, they will __________ their work to the subversion server.

regionate debug checkout execute (**commit**) sandwich

2. Fast Flood Simulation – 14 points

You may not use loops in this question. Write a complete program to recursively simulate fast flooding of a large geographic area. The terrain is represented as an evenly-spaced square grid of N by N miles (N is odd), each grid square represents 1 square mile.

Flooding starts at the center square and will recursively attempt to flood the four nearest neighbors (NSEW). Water will flow into a neighbor if it is lower and dry. Write a main method and a recursive method to determine which grid squares quickly become wet and print the result.

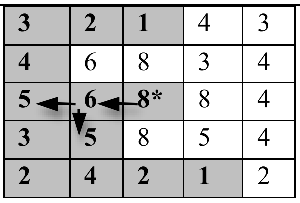
In a different class named `Given`, three public class methods are *already* implemented:

```
public class Given {
  public static int getN() {...}
  public static double getHeight(int x,int y) {...}
  public static void display(boolean[][] wet,int xx) {...}
}
```

getN	returns an odd integer, the number of rows (and columns) of the grid.
getHeight	returns the elevation (always positive for valid square) for the requested (x,y) square. Returns `0` if (x,y) is out-of-bounds, *x*<0, *y*<0, *x*>=*N*, or, *y*>=*N*.
display	prints out the flood map (W's and _'s). If you choose row-first, *wet[x][y]*, use xx=0; for column-first data, *wet[y][x]*, *xx=1*.

An example for N=5 is below. Flooding starts at the center square * and wets 13 squares.

`getHeight(x,y)`



3	**2**	**1**	4	3
4	6	8	3	4
5	**6**	**8***	8	4
3	**5**	8	5	4
2	**4**	**2**	**1**	2

2D Wetted Array

T	T	T	F	F
T	F	F	F	F
T	T	T	F	F
T	T	F	F	F
T	T	T	T	F

Program output

```
Wetted=13

W W W _ _
W _ _ _ _
W W W _ _
W W _ _ _
W W W W _
```

Write a *complete* Java class "`FloodSim`" with two methods: A main method and a recursive method. Your main method will create a two dimensional array of booleans to represent which squares become wet (for each square, true=wet, false=dry) so that the flood map can be calculated by the recursive method. After using your recursive method print out `Wetted=` followed by the total number of squares wetted and finally call the given *display* method to print out the map.

Your recursive method "`flood`" will modify the 2D array and return an integer – the total number of squares wetted. Consider base cases to handle out-of-bounds (x,y) coordinates, already wet squares and squares that are same height or higher than the calling square.

All grid squares are initially dry. Water only flows downhill: A wet square will recursively cause a neighboring cell also to be wetted if the neighbor cell has a lower elevation than its wet neighbor. For this simple simulation recursively visit the four immediate neighbors (x +1,y) (x-1,y) (x,y+1) and (x, y-1), and the water's height remains the land elevation of the square. Cells that are already wet are not re-wetted and are not doubly-counted.
Hint: Do you need to change the wet array before or after the recursive call?

2. Fast Flood Simulation Continued

Write your program here

```
public class FloodSim {
        public static void main(String[] args) {
                int n = Given.getN();
                boolean [][] map = new boolean[n][n];
                // make initial water height is higher than starting square.
                double hStart = Given.getHeight(n/2,n/2) + 1;
                int count = flood(map, n/2,n/2, h);
                System.out.println("Wetted=" + count);
                Given.display(map,0); // 0 = [x][y] convention
        }
        public static int flood(boolean[][] map, int x, int y, double hIn) {
                        double h = Given.getHeight(x,y);
// if we're off the map (h=), not lower than the 'calling' water,
// or already wet just return
                        if (h ==0 || h >= hIn || map[x][y]) return 0;
                        map[x][y] = true;
        // recursively explore NSEW
                        return 1 + flood(map,x+1,y,h)
                                + flood(map,x-1,y,h)
                                + flood(map,x,y+1,h)
                                + flood(map,x,y-1,h);
                }
        }
```

3. Recursion on Linked Lists – 14 points (3+3+4+4)

Complete the Link class by writing the methods described below. **Do not use loops, or create any more methods (other than those specified), class or instance variables.**

```
public class Link {
   private Link next; // null if this is the last link
   private int value;

   public Link(Link n, int v) {next = n; value = v;}
```

1. Write a recursive instance method `count` that takes no parameters and returns an *int*. Return the number of links that hold the value zero, with the exception that the last link in the list is not included in the count. For example for the linked list, {(first) 5→0→3→1→0→0 (last)} it would return the value 2.

Note: The Conflict exam had a different algorithm.

```
int count() {
        if(next==null) return 0; // Don't check value for last link
        if(value ==0 ) return 1 + next.count();
        return next.count();
}
```

2. Write a recursive instance method `square` that takes no parameters and returns a boolean. The effect of calling this method is to square the value of each link. Return true if the last link's new value is greater than 99. Otherwise return *false*. For example, the linked list, {5→0→11} would become {25→0→121} and return the value *true*.

Note: The Conflict exam had a different algorithm.

```
boolean square() {
        value *= value;
        if(next == null) return value > 99;
        return next.square();
}
```

3. Recursion on Linked Lists Continued

Do not use loops, or create any more methods (other than those specified), class or instance variables.

3. Write a recursive instance method named `findBad` that takes no parameters and returns a reference to a Link. The result of calling `findBad` is a reference to the first link that has a value greater than the next link's value. If no such link exists return null. For example, for {5→6→7→3→2} it would return a reference to the third link ("7").

Note: The Conflict exam had a different algorithm.

```
public Link findBad() {
        if(next ==null) return null; // Last link, there are no more next values to check
        if(value > next.value) return this;
        return next.findBad();
}
```

4. Write a recursive method `modify` to replace each link's value with the next link's original value. The last link's value should be set to zero. For example if the linked list is {(first)4→3→2(last)}. The final values will be {3→2→0}. You are free to choose the parameters and return value required (if any) to implement your recursive function.

```
public void modify() {
        if(next ==null)  {
                value =0;
                return;
        }
        value = next.value(); // Grab the next value before it's modified...
        next.modify();
}
```

4. Algorithm Analysis – 15 points (3 points each)

For each method determine the order of growth of the *worst case* running time as N increases. Write your answer to the right of the method using big O notation. You only need to write the order of growth; you do NOT need to explain how you got your answer.

Note: The conflict exam had different algorithms

```
public static double f1(double[][] array) {
   int N = array[0].length;
   double b = Math.random();
   for(int i = 0; i < 3; i++)
      for(int j = N - 1; j > 0; j--)
          b *= N * N * array[i][j];
   return b ;
}
```

O(N)

```
public static int f2(int[] arr) {
  // N is arr.length
  int count = arr.length -1;
  while(count >0) {
    if(arr[count] < 0)
       count -= 7;
    count = count - arr.length / 50;
  }
  return count;
}
```

O(1)
(For large N the loop iterates ~50 times)

```
public static boolean f3(double[][] array) {
 //N is array.length
    for(int i=0;i<array.length;i++)
      for(int j=0;j<i;j++)
       if(array[i][j] == 0) return true;
   return false;
}
```

$O(N^2)$

```
public boolean f4(int[] arr, int lo, int hi) {
  // N = initial value of (hi-lo+1) of 1st activation of f4
  if(lo == hi) return false;
  if(arr[hi] * arr[lo]>10)
     return true;
   return f4(arr,lo+1 , hi);
}
```

O(N)

```
public int f5(int[] array, int x) {
   // N = array.length
   int m = array.length;
   while(array[m] > x && m>0)
      m=m/2;
   return m;
}
```

O(lg N)

5. Complexity Analysis of Mergesort – 14 points

Use Big O notation to complete the following Order of Growth table for sorting N items using the simple Mergesort studied in CS125. (Pseudo-code is shown to the right).

For N items	Best Case	Worst Case
1. Running Time	O(N lg N)	O(N lg N)
2. Space requirements	O(N)	O(N)

```
mergesort(arr, lo, hi){
   if(lo>=hi) return
   mid = ( lo + hi ) / 2
   mergesort(arr,lo,mid)
   mergesort(arr,mid+1,hi)

   merge(arr,lo,mid,mid+1,hi)
}
```

Mergesort Pseudo-code

Note: The merge function allocates an array to merge two sorted sub-arrays back together.

3. Why does Mergesort need more memory than Quicksort?

Both algorithms need O(lg N) memory to hold activation record. However, Mergesort also uses an additional O(N) array to merge the two sub-arrays together. Quicksort does not need the additional storage- it swaps values in original array.

The following questions refer to the above pseudo-code for the Mergesort algorithm.

4. For the following array what are the array values just after executing the very first call to the `merge` function?

Initial values	13	8	26	53	14	9	2	38
After first *merge* call	**8**	**13**	26	53	14	9	2	38

(Only first two items are swapped)

5. How many times will `merge` (not *mergesort; see code above*) be executed (activated) to completely sort 8 items? Hint: Sketching an activation diagram may be useful.

# Activations of merge(arr,lo,mid,mid+1,hi): **7**

6. For the following array what are the array values just before executing the *very last* call to `merge` ?

Initial values	13	8	26	53	14	9	2	38
Before final *merge* call	8	13	26	53	2	9	14	38

7. It takes 100 milliseconds for Mergesort to process an array of 10,000 integers that are initially in random order. If the array was initially in a sorted order, how many integers could now be sorted in approximately 100 milliseconds?

a) 100 or fewer
b) 1,000
c) 10,000
d) 100,000
e) 1,000,000 or greater

Your choice: **C**

Note: Unlike Quicksort, Mergesort does not care if the data is sorted or not; the structure of the recursion is always the same (half, then merge). So the number of items that can be sorted will be the same.

In practice on modern hardware, Mergesort tends to run slightly quicker if the data is already sorted, probably because the CPU is correctly predicting the future code path during the merge function

6. Quicksort – 12 points

1. "For quicksort, the median value would be an excellent pivot value but it is impractical to find." Briefly explain the two facts (good pivot but impractical) in the above statement.

Good pivot: Likely to partition array into two equal halves.
Impractical: Takes too long to find (e.g. would require array to be sorted).

Notes: Actually you can find the median without completely sorting using Hoare's method. Roughly: partition the data and then compare the size of each partition to figure out which one needs to be further partitioned to find the median - this would still be too long as we are seeking an O(1) method.

2. Circle the one best answer: The last value of the subarray to be partitioned is a reasonable pivot value to choose when the array contents are initially______.

(**randomly ordered**), pre-sorted, sorted in reverse

3. An adversary wishes to disable your surgical robot. They know your application uses a particular Quicksort and are able to send it a large amount of carefully ordered data to be sorted. What is the worst case running time of Quicksort in this case? Use Big O notation.

Write your answer here: $O(N^2)$

Consider the following algorithm (*swap* swaps the values at the left and right indices).

```
int mystery(int[] data, int t) {
  int left = 0, right = data.length-1
  while (left < right) {
    if (data[left] >= t) {
      swap(data, left, right);
      right --;
    }
    else left ++;
  }
  return left;
}
```

4. Show the contents of the following array after `mystery(data, 19)` completes.

Initial data	13	8	14	18	1	20	2	38
After *mystery* returns	13	8	14	18	1	2	38	20

5. Circle the one best description of the above *mystery* function.

quicksort, find-minimum , binary-search, merge-sort, selection-sort, (**partition**)

7. Arthur Dent (Objects and Conditionals) – 12 Points

You are writing a Java program to help Arthur Dent choose how to respond to a question. Arthur has 3 options: Agree, Disagree, or Shrug. Use the following rules to decide which option Arthur will use:

1. Arthur will *`shrug`* if the audience is larger than one person or the question text is empty.
2. If rule 1 does not apply, Arthur will *`agree`* if the question is a joke and is 140 characters or less.
3. If neither rule 1 nor rule 2 applies, Arthur will *`disagree`*.

Complete the following *Arthur* class by writing the following:

- A public constructor that takes a string, int and boolean and sets all instance variables defined below with parameter values of the same types as the instance variables. However, if the *question's* parameter value is `null` set the question variable to an empty string.
- A public instance method, "`choose`" with no parameters that returns one of 3 Strings, "`Agree`", "`Disagree`", or "`Shrug`", based on the above rules. Your code must have exactly 3 return statements, one for each condition.

`Note: The conflict exam had different specification.`

```
public class Arthur {

    private String question;  // the question text
    private int audience; // size of audience
    private boolean joke; // true if the question is a joke
    public Arthur(String q, int a, boolean j) {
        this.question = q ==null ? "" : q;
        this.audience = a;
        this.joke = j;
    }
    public String choose() {
        if(question.length() ==0 II audience>1) return "Shrug";
        if(joke && question.length() <= 140) return "Agree"
        return "Disagree";
    }
}
```

8. Santa's Elves Todo List – 9 Points

Read the given code. Complete the class "Presents" started below, according to the following specification. You may use loops in this question.

- A public constructor that takes an integer parameter – the maximum capacity of the list and will also initialize data to a new String array. This is the only time you will create a new array.
- A public instance method *add* that takes a String – the name of the present to add – and returns a boolean. Assume the name is non-null. If there is space left in the array, store if after the last present previously added, increment size and return true, otherwise ignore the present and return false. **Do not create a new array**.
- A public instance method *equals* that takes a reference to a Java `Object` and returns a boolean. Return *true* if and only if the other object is also a *Presents* object and contains an identical list of presents in the same order: You are required to use `instanceof` to verify object type and `String.equals` to compare present names.

```
public class Presents {
  private String[] data; // valid strings stored in 0... size-1
  private int size; // initially =0, =data.length once array is full
  public int getSize() { return size;}
  public String get(int i) {return data[i];}
```

```
public Presents(int capacity) {
	data = new String[capacity];
}
public boolean add(String s) {
	if(size == data.length) return false;
	data[size++] = s;
	return true;
}
public boolean equals(Object other) {
	if(other instanceof Presents) {
		Presents p = (Presents) other;
		if(p.size != this.size) return false;
		for(int i=0;i < size; i++) {
			if( ! get(i).equals( p.get(i) )) return false;
		}
		return true;
	}
	return false;
}
```

9. Social Network Bonus Challenge – 5 points

You may use loops in this question. See the Movie and Actor class below. Each actor object represents a famous actor and the movies that they appeared in. Each movie object represents a movie - specifically its title and the listed actors. *The boolean flag, 'printed' is initially false for all actors.* The same actor may appear in many movies i.e. the same actor object may be referenced in several *actors* arrays of different movies. You can assume a constructor and instance variables have already been written. Write the following recursive instance method '*visit*' in the Actor class. Do not create any other instance or class variables or any other methods.

The *visit* method takes no parameters and returns an integer. The effect of calling this recursive method is such that all actors' names in this network are printed once (one name per line) and their *printed* flag is set to *true*. Return the total number of unique actors visited. Assume all references in the arrays are valid and non-*null*.

```
public class Movie {
	public String title;
	public Actor[] actors;
}
```

```
public class Actor {
      private String name;
      private Movies[] movies;
      private boolean printed;
      // constructor not shown.
      int visit() {
            if(printed) return 0;
            printed = true;
            System.out.println(name);
            int count =1;
            for(int i =0;i < movies.length; i++)
                  for(int j=0;j<movies[i].actors.length;j++)
                        count += movies[i].actors[j].visit();
            return count;
      }
}
```

The social network of actors can be large. The simplest way to explore this network is to use recursion (though non-recursive algorithms are possible by keeping a note of which actors have been seen but not yet been processed).