University of Maryland, College Park

Dept. of Computer Science

CMSC132 Final Exam

First Name (PRINT):_____

Last Name (PRINT): _____

University ID: _____

Instructions

- This exam is a closed-book, closed-notes, 120 minute exam.
- The exam is worth **200 pts**
- **WRITE NEATLY**. If we cannot understand your answer, we will not grade it (i.e., 0 credit).

Page 2	*(12)*	
Page 3	*(22)*	
Page 4	*(27)*	
Page 5	*(14)*	
Page 6	*(30)*	
Page 7	*(12)*	
Page 8	*(19)*	
Page 9	*(16)*	
Page 10	*(20)*	
Page 11	*(12)*	
Page 12	*(16)*	
Total	*(200*	

1. [3 points] State the complexity (using big-O notation) for the function below. Give the smallest bound possible and use the simplest notation possible.

$$13 \log_7 n + 25n^{10} + 2^n$$

Answer: O( )

2. [9 points] State the running time (using big-O notation) for a call to each of the methods below. Give the smallest bound possible and use the simplest notation possible.

a.
```
void f(int n) {
   for (int i = 0; i < n; i += 10) {
      for (int j = i; j > 0; j--) {
         for (int k = 0; k < 10000; k++) {
          …
         }
      }
   }
}
```

Answer: O( )

b.
```
void g(int n) {
   for (int k = n; k > 1; k /= 2) {
      …
   }
}
```

Answer: O( )

c.
```
void p(int n) {
    if (n == 0 || n == 1)
       return 15;
    return p(n-1) * p(n-2);
 }
```

[Hint: We did not talk about complexity of recursive methods *in general*, but we analyzed a specific example that was almost exactly the same as this one!]

Answer: O( )

3. [6 points] Consider the method below:

```
public void foo(List<String> list) {
   for (int i = 0; i < list.size(); i++) {
      System.out.println(list.get(i));
   }
}
```

a. What is the running time (in terms of big-O) if the method is called with an `ArrayList` as the argument? (Use the variable "n" to represent the size of the list.)

b. What is the running time (in terms of big-O) if the method is called with a `LinkedList` as the argument? (Use the variable "n" to represent the size of the list.)

4. [4 points] Regarding code coverage:

a. 100% statement coverage implies that every possible flow path has been tested.
TRUE/FALSE

b. 100% conditional or branch coverage implies that every possible flow path has been tested.
TRUE/FALSE

5. [12 points]

a. Name one advantage TCP has over UDP.

b. Give an example of an application where UDP would be preferable.

c. What do we call the numbers that identify a particular machine on a network?

d. What do we call the number that identifies a particular networking application running on a machine on a network?

6. [9 points]
    a. What is an advantage of HashSet over TreeSet?

    b. What is an advantage of TreeSet over HashSet?

    c. What is an advantage of LinkedHashMap over HashMap?

7. [6 points] Write a short code fragment that could result in *deadlock* if executed concurrently by more than one thread.

8. [12 points] Which algorithm strategies are employed for each example below? (You may circle more than one, but you will be penalized for incorrect circles.)

    a. Dijkstra's Algorithm

Greedy Backtracking Divide&Conquer Dynamic Programming Heuristics

    b. Merge Sort

Greedy Backtracking Divide&Conquer Dynamic Programming Heuristics

    c. DFS through a maze, trying to find the end point.

Greedy Backtracking Divide&Conquer Dynamic Programming Heuristics

    d. DFS through a maze, trying to find the end point. At each juncture where there is a choice, *first* exploring the direction that seems to be leading most directly toward the end point.

Greedy Backtracking Divide&Conquer Dynamic Programming Heuristics

9. [6 points] What is the definition of a min-heap?

10. [8 points] The following array represents a heap, stored sequentially instead of using a tree with nodes.

8	12	15	30	17	100	20

In the boxes below, draw the heap the way it would look **after the smallest element has been removed**. Be sure to follow the algorithm shown in class for removing the smallest element. (Hint: We strongly recommend drawing the heap as a tree first!)

11. [6 points] Think about "checked" versus "unchecked" exceptions.
    a. Briefly **characterize** the kinds of situations where it makes more sense to throw a *checked* exception.

    b. Briefly **characterize** the kinds of situations where it makes more sense to throw an *unchecked* exception.

12. [18 points] Complete the following table using big-O notation. Use the smallest bound possible and simplest notation possible.

Sorting Algorithm	Average Case Complexity	Worst Case Complexity
QuickSort		
BucketSort *(Assume the range is at least as large as the data set, and that the data is random, but over a uniform distribution.)*		
HeapSort		

13. [6 points] Describe the Tree Sort algorithm.

14. [6 points] Write a class called `StringLengthComparator`, which implements the `Comparator` interface and defines an ordering for String objects according to the length of the strings.

15. [3 points] Under what circumstances can a "data race" occur?

16. [3 points] What is wrong with the following class?

```
public class Apple {
   private int size;
   public Apple(int size) {
      this.size = size;
   }
   public boolean equals(Object x) {
      // assume equals has been implemented correctly so that apples
      // with the same size are considered equal
   }
}
```

17. [19 points] Consider a class called `Squares`. Below is an illustration of how the class may be used:

```
Squares s = new Squares();
Iterator<Integer> it = s.iterator();
while(it.hasNext()) {
   System.out.println(it.next());
}
```

The output for this code fragment is:
1
4
9
16
25
… // it goes on "forever"

The `Squares` class implements `Iterable`, so your job will be to **implement the `iterator` method in the space below.** Your iterator method must return an Iterator that will iterate over all of the squares: 1, 4, 9, 16, 25, 36, 49….. [You may imagine that the values go on "forever", even though eventually an overflow will occur.]

You must use an anonymous inner class. **The iterator's remove method should throw an UnsupportedOperationException.**

```
public class Squares implements Iterable<Integer> {
```

18. [16 points] Assume we are implementing a **sorted** linked list using the Node class below:

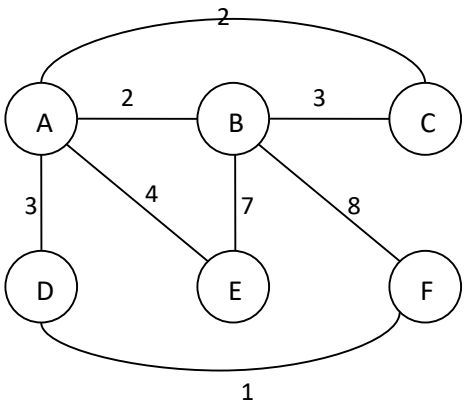
```
private class Node {
      private int data;
      private Node next;
}
```

Implement the method below, which accepts the head of a **sorted** linked list as a parameter. The method returns the head of an updated list containing no duplicates. (Return the head of a list containing the same elements as the original list, but without duplicates). **You must implement this using recursion.** You may not use any loops of any kind, and you must not allocate any new Node instances. **Remember: The list is assumed to be sorted!** You may NOT write a helper method for this one, and you must not create any unnecessary data structures.

```
public static Node removeDupsFromSortedList(Node head) {
```

19. [20 points] Write a "blocking queue", by wrapping an `ArrayList` inside a class called "Queue". Be sure to use correct generic notation so that your collection will work with any Java objects. The API should consist of just two methods: `enqueue` (which adds an element to the queue) and `dequeue` (which removes an element from the queue.) Your implementation must be thread-safe. When a thread attempts to perform the dequeue operation on an empty queue, it should "block" until an element becomes available.

20. [12 points] Use the following graph to answer the question below.



a. [10 points] Apply Dijkstra's algorithm using **B** as the starting (source) node. Indicate the cost/distance and predecessor for each node in the graph after processing 2 nodes. Don't leave any of the boxes blank or you will lose points!

Starting table (filled in for you):

Node	A	B	C	D	E	F
Cost	∞	0	∞	∞	∞	∞
Predecessor	-	-	-	-	-	-

FILL IN THE TABLES BELOW FOR PROCESSING TWO SUCCESSIVE NODES. (You will need to copy many of the entries from the preceding table.)

After processing first node:

Node	A	B	C	D	E	F
Cost						
Predecessor						

After processing second node:

Node	A	B	C	D	E	F
Cost						
Predecessor						

21. [16 points] Consider the following implementation of a Binary Search Tree:

```
public class BinarySearchTree<K extends Comparable<K>, V> {
   private class Node {
      private K key;
      private V data;
      private Node left, right;
      public Node(K key, V data) {
         this.key = key;
         this.data = data;
      }
   }
   private Node root;
}
```

Write an instance method called **getMin** that finds the smallest key and returns the data value associated with that key. For an empty tree the method will return null. **Your implementation must be efficient!** You may write an auxiliary (helper) method if you find it useful.