You have 80 minutes for this exam. The exam is closed book, except that you are allowed to use one page of notes (8.5-by-11, one side, in your own handwriting). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. You may use the back of each page for scratch space, or to continue long answers.

Name: P01 9:00 Andy Guna
NetID: P02 10:00 Andy Guna
Room: P02A 10:00 Elena Sizikova
Precept: P03 11:00 Maia Ginsburg
P03A 11:00 Nora Coler
P04 12:30 Maia Ginsburg
P04A 12:30 Miles Carlsten
P05 1:30 Tom Wu

Write and sign: “I pledge my honor that I have not violated the Honor Code during this examination.”

<table>
<thead>
<tr>
<th>Problem</th>
<th>Score</th>
<th>Problem</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Sub 1</td>
<td></td>
</tr>
<tr>
<td>Sub 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total:
0. Constructor. (1 point)

In the space provided on the front of the exam, write your name and Princeton netID; write the name of the room in which you are taking the exam; mark your precept number; and write and sign the honor code.

1. The Usual COS226 Sorting Question. (16 points)

The column on the left is an array of strings to be sorted or shuffled. The column on the right is in sorted order. The other columns are the contents of the array at some intermediate step during one of the algorithms below. Write the number of each algorithm under the corresponding column. Use each number exactly once.

<table>
<thead>
<tr>
<th>mars</th>
<th>care</th>
<th>dart</th>
<th>barn</th>
<th>barn</th>
<th>barn</th>
<th>yard</th>
<th>bark</th>
<th>lard</th>
<th>bark</th>
</tr>
</thead>
<tbody>
<tr>
<td>part</td>
<td>gary</td>
<td>lard</td>
<td>fare</td>
<td>care</td>
<td>care</td>
<td>warm</td>
<td>barn</td>
<td>care</td>
<td>barn</td>
</tr>
<tr>
<td>care</td>
<td>mars</td>
<td>care</td>
<td>rare</td>
<td>fare</td>
<td>dart</td>
<td>vary</td>
<td>card</td>
<td>gary</td>
<td>card</td>
</tr>
<tr>
<td>gary</td>
<td>part</td>
<td>gary</td>
<td>jars</td>
<td>gary</td>
<td>fare</td>
<td>part</td>
<td>care</td>
<td>barn</td>
<td>care</td>
</tr>
<tr>
<td>barn</td>
<td>barn</td>
<td>barn</td>
<td>harp</td>
<td>harp</td>
<td>gary</td>
<td>tart</td>
<td>dart</td>
<td>card</td>
<td>dart</td>
</tr>
<tr>
<td>park</td>
<td>fare</td>
<td>card</td>
<td>mars</td>
<td>jars</td>
<td>harp</td>
<td>park</td>
<td>earn</td>
<td>farm</td>
<td>earn</td>
</tr>
<tr>
<td>rare</td>
<td>park</td>
<td>farm</td>
<td>gary</td>
<td>mars</td>
<td>jars</td>
<td>rare</td>
<td>fare</td>
<td>fare</td>
<td>fare</td>
</tr>
<tr>
<td>fare</td>
<td>rare</td>
<td>fare</td>
<td>warm</td>
<td>park</td>
<td>mars</td>
<td>gary</td>
<td>farm</td>
<td>farm</td>
<td>farm</td>
</tr>
<tr>
<td>warm</td>
<td>harp</td>
<td>harm</td>
<td>care</td>
<td>part</td>
<td>park</td>
<td>mars</td>
<td>gary</td>
<td>earn</td>
<td>gary</td>
</tr>
<tr>
<td>tarp</td>
<td>jars</td>
<td>earn</td>
<td>tarp</td>
<td>rare</td>
<td>part</td>
<td>tarp</td>
<td>harm</td>
<td>jars</td>
<td>harm</td>
</tr>
<tr>
<td>jars</td>
<td>tarp</td>
<td>jars</td>
<td>part</td>
<td>tarp</td>
<td>rare</td>
<td>oars</td>
<td>harp</td>
<td>harp</td>
<td>harp</td>
</tr>
<tr>
<td>harp</td>
<td>warm</td>
<td>harm</td>
<td>park</td>
<td>warm</td>
<td>tarp</td>
<td>nary</td>
<td>jars</td>
<td>bark</td>
<td>jars</td>
</tr>
<tr>
<td>vary</td>
<td>bark</td>
<td>bark</td>
<td>vary</td>
<td>bark</td>
<td>vary</td>
<td>care</td>
<td>vary</td>
<td>dart</td>
<td>lard</td>
</tr>
<tr>
<td>dart</td>
<td>dart</td>
<td>mars</td>
<td>dart</td>
<td>dart</td>
<td>warm</td>
<td>dart</td>
<td>part</td>
<td>mars</td>
<td>mars</td>
</tr>
<tr>
<td>bark</td>
<td>vary</td>
<td>vary</td>
<td>bark</td>
<td>earn</td>
<td>bark</td>
<td>bark</td>
<td>mars</td>
<td>yard</td>
<td>nary</td>
</tr>
<tr>
<td>yard</td>
<td>yard</td>
<td>yard</td>
<td>yard</td>
<td>harm</td>
<td>yard</td>
<td>fare</td>
<td>yard</td>
<td>vary</td>
<td>oars</td>
</tr>
<tr>
<td>earn</td>
<td>earn</td>
<td>tarp</td>
<td>earn</td>
<td>vary</td>
<td>earn</td>
<td>earn</td>
<td>park</td>
<td>tarp</td>
<td>park</td>
</tr>
<tr>
<td>harm</td>
<td>farm</td>
<td>warm</td>
<td>harm</td>
<td>yard</td>
<td>harm</td>
<td>harm</td>
<td>tarp</td>
<td>warm</td>
<td>part</td>
</tr>
<tr>
<td>farm</td>
<td>harm</td>
<td>rare</td>
<td>farm</td>
<td>card</td>
<td>farm</td>
<td>farm</td>
<td>rare</td>
<td>tart</td>
<td>rare</td>
</tr>
<tr>
<td>tart</td>
<td>tart</td>
<td>tart</td>
<td>tart</td>
<td>farm</td>
<td>tart</td>
<td>barn</td>
<td>tart</td>
<td>rare</td>
<td>tarp</td>
</tr>
<tr>
<td>card</td>
<td>card</td>
<td>park</td>
<td>card</td>
<td>card</td>
<td>card</td>
<td>warm</td>
<td>park</td>
<td>tart</td>
<td>tart</td>
</tr>
<tr>
<td>lard</td>
<td>lard</td>
<td>part</td>
<td>lard</td>
<td>nary</td>
<td>lard</td>
<td>lard</td>
<td>lard</td>
<td>lard</td>
<td>oars</td>
</tr>
<tr>
<td>oars</td>
<td>nary</td>
<td>oars</td>
<td>oars</td>
<td>oars</td>
<td>oars</td>
<td>jars</td>
<td>oars</td>
<td>nary</td>
<td>warm</td>
</tr>
<tr>
<td>nary</td>
<td>oars</td>
<td>nary</td>
<td>tary</td>
<td>nary</td>
<td>harp</td>
<td>nary</td>
<td>part</td>
<td>yard</td>
<td>----</td>
</tr>
</tbody>
</table>

0 5 7 1 4 3 6 2 8 9

(0) Original input
(4) Mergesort
(7) Quicksort
(top-down) (no shuffle)
(1) Knuth shuffle
(5) Mergesort
(8) 3-way Quicksort
(bottom-up) (no shuffle)
(2) Selection sort
(6) Heapsort
(3) Insertion sort
(9) Sorted
2. Playing Cards. (16 points)

We would like to sort playing cards from a deck. Associated with each card is a denomination (1 to 13) and a suit (CLUBS < DIAMONDS < HEARTS < SPADES).

A card \( c_1 \) is considered less than a card \( c_2 \) if either of the following is true:

- the suit of \( c_1 \) is less than the suit of \( c_2 \), or
- \( c_1 \) and \( c_2 \) are of the same suit, but the denomination of \( c_1 \) is less than the denomination of \( c_2 \).

(a) Let us first consider sorting cards of the same suit, based purely on their denominations. Specifically, consider using 2-way quicksort to sort the 3, 4, 5, 6, 7, 8 and 9 of hearts. After a random shuffle, we have the following sequence of denominations: 5, 6, 8, 3, 9, 4, 7. Show the result of the first call to \( \text{partition() \ by giving contents of the array after each exchange.} \) Please write only the two elements that were exchanged.

\[
\begin{array}{ccccccc}
5 & 6 & 8 & 3 & 9 & 4 & 7 \\
\hline
 & 4 & & & & 6 & \\
\hline
 & & 3 & 8 & & \\
\hline
3 & & 5 & & & & \\
\end{array}
\]

Now write the entire contents of the partitioned array, and draw a box around each of the left and right subarrays on which recursive calls will be executed.

\[
\begin{array}{ccccccc}
3 & 4 & 5 & & 8 & 9 & 6 & 7 \\
\end{array}
\]
(b) The Card class is implemented in Java as follows. Complete the `compareTo()` function, implementing the ordering described on the previous page, and assuming that the argument is not null.

```java
public class Card implements Comparable<Card> {
    // Comparators by suit and by denomination
    public static final Comparator<Card> SUIT_ORDER = new SuitOrder();
    public static final Comparator<Card> DENOM_ORDER = new DenomOrder();

    // Suit of the card (CLUBS = 1, DIAMONDS = 2, HEARTS = 3, SPADES = 4)
    private final int suit;

    // Denomination of the card
    private final int denom;

    public Card(int suit, int denom) {
        if (suit < 1 || suit > 4)
            throw new IllegalArgumentException("Invalid suit");
        if (denom < 1 || denom > 13)
            throw new IllegalArgumentException("Invalid denomination");
        this.suit = suit;
        this.denom = denom;
    }

    // COMPLETE THE FOLLOWING FUNCTION
    public int compareTo(Card that) {
        if (suit < that.suit) return -1;
        if (suit > that.suit) return 1;
        if (denom < that.denom) return -1;
        if (denom > that.denom) return 1;
        return 0;
    }

    // Compare cards according to the suit only
    private static class SuitOrder implements Comparator<Card> {
        // Implementation not shown
    }

    // Compare cards according to the denomination only
    private static class DenomOrder implements Comparator<Card> {
        // Implementation not shown
    }
}
```
(c) Suppose that the variable cards is an array of cards. We could sort it, using your compareTo function, with a call to MergeX.sort(cards). Which of the following code fragments would produce an equivalent final result? Circle all equivalent code fragments.

**Option 1: No.**

```java
MergeX.sort(cards, Card.SUIT_ORDER);
MergeX.sort(cards, Card.DENOM_ORDER);
```

**Option 2: Yes.**

```java
MergeX.sort(cards, Card.DENOM_ORDER);
MergeX.sort(cards, Card.SUIT_ORDER);
```

**Option 3: Yes.**

```java
MergeX.sort(cards);
MergeX.sort(cards, Card.SUIT_ORDER);
```

**Option 4: Yes.**

```java
MergeX.sort(cards, Card.DENOM_ORDER);
MergeX.sort(cards);
```

**Option 5: No.**

```java
Quick.sort(cards, Card.SUIT_ORDER);
Quick.sort(cards, Card.DENOM_ORDER);
```

**Option 6: No.**

```java
Quick.sort(cards, Card.DENOM_ORDER);
Quick.sort(cards, Card.SUIT_ORDER);
```

**Option 7: No.**

```java
MergeX.sort(cards);
Quick.sort(cards, Card.SUIT_ORDER);
```

**Option 8: Yes, assuming cards are distinct.** (No points were deducted on the exam).

```java
Quick.sort(cards, Card.DENOM_ORDER);
MergeX.sort(cards);
```
3. **Traversing Trees.** (10 points)

(a) Circle the correct *binary tree* (not necessarily a BST) that would produce both of the following traversals:

- In-order: AQVNBRMSP
- Pre-order: BQAVNRSMP

(b) Circle the correct *Binary Search Tree* that would produce the following traversal:

- Post-order: ABCDEFG

(c) If you know that a tree is a BST, which of the following is or is not always sufficient to reconstruct it? For each one, write *yes* if it is enough to reconstruct the tree, or *no* if it is not.

- Pre-order traversal: *Yes.*
- In-order traversal: *No.*
- Post-order traversal: *Yes.*
- Level-order traversal: *Yes.*
4. BSTs, LLRB and otherwise. (12 points)

(a) Label each node in the following binary tree with numbers from the set \{2, 26, 10, 27, 20, 15, 42\} so that it is a legal Binary Search Tree. (Hint: use the back of the page as scratch space, and only write down the answer once you have it.)

(b) Now label each edge in the figure with \(r\) or \(b\), denoting RED and BLACK, so that the tree is a legal Left-Leaning Red-Black Tree.

(c) Considering your labeling in (b), is it possible to assign different red/black labels and still satisfy the LLRB-tree conditions? (Answer yes if a different labeling is possible, or no if your labeling in (b) is unique.) No.

(d) If the answer to (c) is yes, draw and label the second tree. If the answer is no, how do you know that the red-black labeled tree must be unique?

Sample answer:
Thinking about the equivalence to a 2–3 tree, 27–42 must be a 3-node since 42 can’t have only 1 child. Therefore 42–27 is red. Therefore every path from root to leaf has to have 1 black edge. Since red edges can’t be consecutive, 10–20 must be black and 26–10 and 20–15 red. Finally, 10–2 must be black because 26–10 was red.
5. **Heaps.** (10 points)

Starting from the following max-heap (using the array representation presented in lecture), give the resulting array after each operation:

<table>
<thead>
<tr>
<th>X</th>
<th>10</th>
<th>7</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>

(a) After `insert(9)`

| X | 10 | 9 | 4 | 5 | 7 | 2 | 3 | 0 | 1 | 6 |

(b) After `delMax()`, starting from the original heap (i.e., assuming that (a) has **not** been performed)

| X | 7 | 6 | 4 | 5 | 1 | 2 | 3 | 0 |

(c) For implementing a max-priority queue, which of the following are advantages of a resizing-array implementation of a heap over a **linked list maintained in descending sorted order**? Circle all that apply.

- expected time for `insert` is lower **Yes.**
- expected time for `delMax` is lower **No.**
- expected storage cost is lower **Yes.**
- `insert` has lower worst-case order of growth **No.**
- `delMax` has lower worst-case order of growth **No.**
- `max` has lower worst-case order of growth **No.**
6. FortyTwoPQ. (15 points)

You have been hired by Deep Thought Enterprises to implement a priority-queue-like data structure supporting the following operations:

- `insert()` an item in $O(\log N)$ time.
- `fortytwo()` — return the 42nd smallest item in constant time.
- `delFortyTwo()` — delete the 42nd smallest item in $O(\log N)$ time.

Explain how you would implement the required functionality, using one or more data structures that we have seen in class. Write pseudocode for each of the three operations listed above. You may assume that $N > 42$, and omit all checks for smaller $N$.

For full credit, your implementation should support finding the $k^{th}$ smallest item with an order-of-growth running time independent of $k$. That is, it should be possible to change 42 to some other constant (at compile time) without changing the order-of-growth running time.

Solution #1:
Maintain a MaxPQ called `firstFortyTwo` with 42 items on it, and a MinPQ called `theRest` with all the rest of the items.

```java
insert(x):
    firstFortyTwo.insert(x);
    if (firstFortyTwo.size() > 42)
        theRest.insert(firstFortyTwo.delMax());

fortytwo():
    if (firstFortyTwo.size() < 42) // Check may be omitted in answer
        throw new NoSuchElementException("N < 42");
    return firstFortyTwo.max();

delFortyTwo():
    if (firstFortyTwo.size() < 42) // Check may be omitted in answer
        throw new NoSuchElementException("N < 42");
    x = firstFortyTwo.delMax();
    if (!theRest.isEmpty())
        firstFortyTwo.insert(theRest.delMin());
    return x;
```

Solution #2:
Maintain a Red-Black BST, as well as the 42nd-smallest element.

```java
insert() adds to the RBST, then calls select(42) to find the 42nd-smallest element, saving it in an instance variable.

fortytwo() returns the cached 42nd-smallest element.

delFortyTwo() deletes the 42nd-smallest element from the RBST, then calls select(42) to save the new 42nd-smallest element.
```

Incorrect solution:
Maintain a Red-Black BST, but call select(42) in fortytwo() — takes $\log(N)$ time.
7. Divide and Conquer. (12 points)

Consider the following three algorithms:

- **Algorithm 1** solves problems of size $N$ by recursively dividing them into 2 sub-problems of size $N/2$ and combining the results in time $c$ (where $c$ is some constant).
- **Algorithm 2** solves problems of size $N$ by solving one sub-problem of size $N/2$ and performing some processing taking some constant time $c$.
- **Algorithm 3** solves problems of size $N$ by solving two sub-problems of size $N/2$ and performing a linear amount (i.e., $cN$ where $c$ is some constant) of extra work.

(a) For each algorithm, write down a recurrence relation showing how $T(N)$, the running time on an instance of size $N$, depends on the running time of a smaller instance.

Algorithm 1: $T(N) = 2T(N/2) + c$

Algorithm 2: $T(N) = T(N/2) + c$

Algorithm 3: $T(N) = 2T(N/2) + cN$

(b) For each recurrence relation, pick the solution for $T(N)$ from the following list. Just write the letter corresponding to the correct running time.

| Algorithm 1: C | A: $T(N) \sim c$ |
| Algorithm 2: B | B: $T(N) \sim c \log N$ |
| Algorithm 3: D | C: $T(N) \sim cN$ |
|               | D: $T(N) \sim cN\log N$ |
|               | E: $T(N) \sim cN^2$ |

(c) For each of the following algorithms, pick which of the above classes of algorithms (1, 2, or 3) applies to that algorithm:

Mergesort: 3

Binary search in a sorted array: 2

Quicksort (if partitioning always divides the array in half): 3
8. You didn’t think we forgot about the assignments, did you? (8 points)

(a) Suppose we wanted to simulate percolation in a cube with $N$ sites on a side, with each site connected to its neighbors up, down, left, right, forward, and back. If we used WeightedQuickUnionUF, what would be the order of growth of the expected running time, as a function of $N$?

a. $N^2$

b. $N^2 \log N$

c. $N^3$

d. $N^3 \log N$

e. $N^4$

f. $N^4 \log N$

g. None of the above.

d

(b) If you run your BinarySearchDeluxe on a sorted array with $N$ items but only 3 distinct keys, what is the order of growth of the expected running time for a call to firstIndexOf()?

a. constant

b. $\log N$

c. $\log_3 N$

d. $N$

e. $N \log N$

f. None of the above.

(b)

(c) True or False: The amount of memory necessary to solve 8puzzle is equal to some constant times the size of the game board. **False**

(d) True or False: 8puzzle will still work without implementing the critical optimization, but it may take much more memory and running time to find the answer. **True**

(e) True or False: it is always legal to call equals on two objects that do not have the same type. **True**

(f) True or False: a KdTreeST always has a lower order-of-growth running time than the brute-force PointST for the contains() operation, for all possible query points. **False**

(g) True or False: a KdTreeST always has a lower order-of-growth running time than the brute-force PointST for the range() operation, for all possible query rectangles. **False**

(h) True or False: a KdTreeST always has a lower order-of-growth running time than the brute-force PointST for the nearest() operation, for all possible query points. **False**