PROBLEM 1: \textit{(Short ones (14 points))}

1. You are creating a boggle word search program and you want to find all valid words where the letters are adjacent. What data structure would be best suited to hold the dictionary (i.e. lexicon)?

   \textbf{Trie}

2. In order to use the class \texttt{Point} containing fields \texttt{x} and \texttt{y} in a \texttt{HashSet}, you are considering multiple hash functions. Of these hash functions, which one would give the best performance in a \texttt{HashSet}? Assume that your points are likely to be between (0, 0) and (1280, 1024) (the size of the average computer monitor).

   \begin{verbatim}
   public int hashCode() { return x * 1000 + y; }
   \end{verbatim}

3. \textbf{True or False} State whether the following statement is true or false. If false, you should give a specific counterexample.

   \begin{enumerate}
   \item A certain hash table contains \( N \) integer keys, all distinct, and each of its buckets contains at most \( K \) elements. Collisions are resolved using chaining. Assuming that the hashing function and the equality test require constant time, the time required to find all keys in the hash table that are between \( L \) and \( U \) is \( O(K \times (U - L)) \) in the worst case.

     \textbf{True}
   \item Instead of using a heap, we use an \textit{AVL tree} to represent a priority queue. The worst-case big-Oh of \texttt{add (insert)} and \texttt{poll (deleteMin)} do not change.

     \textbf{True}
   \item Instead of using a heap, we use a \textit{sorted ArrayList} to represent a priority queue. The worst-case big-Oh of \texttt{add} and \texttt{poll} do not change.

     \textbf{False, adding is \( O(n) \)}
   \item Given the preorder and postorder traversals of a binary tree (i.e. printing out all of the elements but not the null nodes), it is possible to reconstruct the original tree.

     \textbf{False, consider the a root with one child. Neither traversal will tell you whether it is a left or right child.}
   \item Given the preorder and inorder traversals of a binary tree, it is possible to reconstruct the original tree.

     \textbf{True}
   \end{enumerate}

PROBLEM 2: \textit{(Reverse (9 points))}

Each of the Java functions on the left take a string \( s \) as input, and returns its reverse. For each of the following, state the recurrence (if applicable) and give the big-Oh complexity bound.

Recall that concatenating two strings in Java takes time proportional to the sum of their lengths, and extracting a substring takes constant time.

\begin{verbatim}
A. public static String reverse1(String s) {
    int N = s.length();
    String reverse = ";
    for (int i = 0; i < N; i++)
    }
\end{verbatim}
Problem 2

B. public static String reverse2(String s) {
    int N = s.length();
    if (N <= 1) return s;
    String left = s.substring(0, N/2);
    String right = s.substring(N/2, N);
    return reverse2(right) + reverse2(left);
}
T(n) = 2T(n/2) + O(n)
T(1) = O(1)
T(n) = n log n

C. public static String reverse3(String s) {
    int N = s.length();
    char[] a = new char[N];
    for (int i = 0; i < N; i++)
        a[i] = s.charAt(N-i-1);
    return new String(a);
}
O(n)

Problem 3: (Bits (10 points))

You would like to implement the set operations for the integers 0-31 using a BitSet class. The ith bit is a 1 if and only if i is in the set. For example,

00000000000000000000000000000000 indicates that there are no elements in the set.
10000000000000000000000000001010 indicates that 1, 3, and 31 are in the set.

The second set can be created with the following client code:

BitSet s = new BitSet();
s.set(1, true);
s.set(3, true);
s.set(31, true);

Below, we have given the constructor and the set method for BitSet.

public class BitSet {
    private int myBits;

    private final static int BITS_PER_INT = 32;

    public class BitSet(int bits) {
        // set all bits to 0
        myBits = 0;
    }

    /**
     * Sets the bit at the specified index to the specified value.
     */

public void set(int bitIndex, boolean value) {
    if (bitIndex < 0 || bitIndex >= BITS_PER_INT) // out of bounds
        return;
    if (value) //
        myBits = myBits | (1 << bitIndex); // set bit to on
    else
        myBits = myBits & ~(1 << bitIndex); // set bit to off
}

A. Complete the get method. In the above example s.get(3) should return true while s.get(7) should return false.

    public boolean get(int bitIndex) {
        if (bitIndex < 0 || bitIndex > BITS_PER_INT -1)
            return false;
        return ((myBits >> bitIndex) & 1) == 1;
    }

B. You would like to implement the union operation for the BitSet class. The union operation (∪) says that if
    \( A = (1, 2, 3, 4, 5) \) and \( B = (1, 3, 5, 7, 9) \), then \( A \cup B = (1, 2, 3, 4, 5, 7, 9) \). Write the method union below.

    public void union(BitSet set) {
        myBits = myBits | set.myBits;
    }

PROBLEM 4:  (Huffman Trees (19 points))

The Huffman compression algorithm uses a tree to encode the codewords, where each node has either two or zero
children. Someone has given you a tree that contains some nodes with only one child.

A. Why can such a tree not be created using the Huffman encoding algorithm discussed in class?
   Trees start as leaves (0 children) and internal nodes are created by bringing two nodes together (2 children). There are no other types of nodes in a Huffman tree.
B. Write a function called *tighten* that given such an encoding tree will remove those nodes with one child. The diagram below shows a “loose” tree on the left and its tightened equivalent on the right. The three shaded nodes are the ones that were removed.

![Diagram showing loose and tightened trees]

The definition of a Huffman *TreeNode* is attached to the end of the test.

```java
/**
 * remove nodes with one child from Huffman tree
 * @param root is the root of a Huffman tree (may be null)
 * @return tree where nodes with one child are removed
 */
public static TreeNode tighten(TreeNode root)
{
    if (root == null)
        return null;
    root.left = tighten(root.left);
    root.right = tighten(root.right);
    if (root.left != null && root.right == null) // one left child
        return root.left;
    if (root.left == null && root.right != null) // one right child
        return root.right;
}
```

C. State the recurrence and the big-Oh for your solution.

\[
T(n) = 2T(n/2) + O(1)T(0) = O(1)T(n) \in O(n \log n)
\]

D. You would like to check to make sure the character counts in the tree sum up correctly. That is the the *weight* field of a node should equal the sum of the weights of its children. Write *validWeights* that checks to see if each node is the sum of its children.

```java
/**
 * @return true iff each internal node’s weights is the sum of its children’s weights
 */
public static boolean validWeights(TreeNode root)
{
    if (root == null || (root.left == null && root.right == null))
        return true;
    return root.left.weight + root.right.weight == root.weight &&
    validWeights(root.left) && validWeights(root.right);
}
```

**PROBLEM 5 : (Puzzle Hunt)**

You are given a matrix of positive integers to represent a game board, where the (0, 0) entry is the upper left corner. The number in each location is the number of squares you can advance in any of the four primary compass directions, provided that move does not take you off the board. You are interested in the total number of distinct ways one could travel from the upper left corner to the lower right corner, given the constraint that no single path should ever visit the same location twice.
Your task is to write a method called `numPaths`, which takes a 2-d array of integers and computes the total number of ways to travel to the lower right corner of the board. Note that you never want to count the same path twice, but two paths are considered to be distinct even if they share a common sub-path. And because you want to prevent cycles, you should change the value at any given location to a zero as a way of marking that you’ve been there. Just be sure to restore the original value as you exit the recursive call. You may want to write a helper function to handle the recursion and a utility function to decide it you are on the board or not.

A. Write `numPaths` below.

```java
/**
   * Calculates total number of distinct ways one could travel from the
   * upper left corner of grid to the lower right corner, given the
   * constraint that no single path should ever visit the same location twice.
   *
   * @param board square matrix board[i][j] is the number of squares
   * one can advance vertically or horizontally from (i,j)
   *
   * @return the number of possible paths from (0,0) to the lower
   * right corner of board (board.length-1, board[0].length - 1)
   */
public static int numPaths(int[][] board)
{
    return numPaths(board, 0, 0);
}

// HELPER FUNCTIONS
/**
   * @return true if (row,col) is within the bounds of the board
   * (i.e. 0 <= row < board.length and 0 <= col < board[0].length)
   * false otherwise
   */
public static boolean onBoard(int[][] board, int row, int col)
{
    return row >= 0 && row < board.length &&
    col >= 0 && col < board[0].length;
}

/**
   * @return the number of possible paths from (row,col) to the lower
   * right corner of board (board.length-1, board[0].length - 1)
   */
public static int numPaths(int[][] board, int row, int col)
{
    if (row == board.length - 1 && col == board[0].length - 1) // found goal
        return 1;
    if (!onBoard(board, row, col) || board[row][col] == 0) // off board or
        return 0; // already seen
    int skip = board[row][col];
    board[row][col] = 0;
    int total = numPaths(board, row + skip, col) +
                numPaths(board, row - skip, col) +
                numPaths(board, row, col + skip) +
                numPaths(board, row, col - skip);
    board[row][col] = skip;
    return total;
}
```
B. Give a recurrence for your solution. You do not need to solve the recurrence.
\[
T(n) = 4T(n-1) + O(1)
\]
\[
T(0) = O(1)
\]

PROBLEM 6: (Tradeoffs (8 points))
You are given an array of \( n \) ints (where \( n \) is very large) and are asked to find the largest \( m \) of them (where \( m \) is much less than \( n \)).

A. Design an efficient algorithm to find the largest \( m \) elements. You do not need to write your solution in Java. Precise English or pseudocode will suffice.

You can assume the existence of all data structures we discussed in class. You do not have to explain how any of the standard methods (e.g. constructing a heap) work. Be specific, however, about which data structures you are using and how these data structures are interconnected.

Your algorithm should work well for all values of \( m \) and \( n \), from very small to very large.

Options from flawed to really good:

- slowish Sort the numbers and list the \( m \) largest
  
  sort: \( O(n \log n) \)
  
  list: \( m * O(1) \in O(m) \)
  
  Total: \( O(n \log n) \)

- OK Keep a min-heap of size \( m \). Add first \( m \) entries. Then add an element and remove minimum element (i.e. element not in top \( m \)). All of the elements in the heap after inserting the last element are the top \( m \).
  
  \( O(m) \) to build heap
  
  \( (n - m) \) insertions and delete mins: \( (n - m)O(\log m) \)
  
  \( O(m) \) to read out final elements of heap
  
  Total: \( O(m + (n - m) \log m) \)

- Good Build a max-heap from the numbers and call deleteMax \( m \) times.
  
  Build heap: \( O(n) \)
  
  Calls to delete max: \( m * O(\log n) \)
  
  \( O(n + m \log n) \)

- Special Sort using radix sort and then grab \( m \) smallest
  
  sort 32-bits: \( 32 * O(n) \)
  
  list : \( m * O(1) \in O(m) \) Total: \( O(d * n) \) where \( d \) is the number of digits

- Extra Special Find \( m \)th largest element using a modified version of partition (below) from Quicksort. Copy over the elements after the \( m \)th largest element.

  Adapted from Wikipedia’s entry on the Selection algorithm:
  
  function findTopM(a, left, right, m)
  
  if right > left
  
  pivotIndex = partition(a, left, right)
  
  if pivotNewIndex > a.length - m
  
  findTopM(a, left, pivotIndex-1, m)
  
  if pivotNewIndex < a.length - m
  
  findTopM(a, pivotNewIndex+1, right, m)

  After running findTopM(a, 0, a.length-1, m)!, the last \( m \) entries of \( a \) will be the top \( m \). The expected time for this algorithm is \( O(n) \). That’s about as good as you can possibly do.

B. What is the running time of your algorithm? What is it for small \( m \)? What is it as \( m \to n \) (i.e. as \( m \) approaches \( n \))?

Plug in values as appropriate. For example, the good heap example above is \( O(n) \) for small \( m \) and \( O(n \log n) \) as \( m \to n \).