Test 2 Solutions: CompSci 100

PROBLEM 1:  (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)?

   Trie

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

   public int hashCode () { return x * 1000 + y; }

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

   I. 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.

      True

   II. Instead of using a heap, we use an AVL tree to represent a priority queue. The worst-case big-Oh of add (insert) and poll (deleteMin) do not change.

      True

   III. Instead of using a heap, we use a sorted ArrayList to represent a priority queue. The worst-case big-Oh of add and poll do not change.

      False, adding is \( O(n) \)

   IV. 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.

      False, consider the a root with one child. Neither traversal will tell you whether it is a left or right child.

   V. Given the preorder and inorder traversals of a binary tree, it is possible to reconstruct the original tree.

      True

PROBLEM 2:  (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.

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

\textbf{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);
}\n\begin{align*}
T(n) &= 2T(n/2) + O(n) \\
T(1) &= O(1) \\
T(n) &= n \log n
\end{align*}

\textbf{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);
}\nO(n)

\textbf{PROBLEM 3 : \textit{(Sorting (7 points))}}

The column on the left is the original input of strings to be sorted. The columns to the right are the contents at some intermediate step during one of the 8 sorting algorithms listed below. Match up each algorithm by writing its number under the corresponding column. Use each number exactly once.

\begin{tabular}{llllllllll}
Jane & Adam & Anna & Will & Jada & Abby & Adam & Abby \\
Adam & Alex & Adam & Seth & Emma & Adam & Dave & Adam \\
Mary & Cole & Abby & Ryan & Ella & Alex & Erik & Alex \\
Jeff & Dave & Ella & Sean & Maya & Anna & Erin & Anna \\
Erik & Erik & Emma & Mark & Anna & Cole & Evan & Cole \\
Dave & Erin & Dave & Noah & Sara & Dave & Jack & Dave \\
Evan & Evan & Alex & Owen & Eric & Ella & Jada & Ella \\
Sean & Jack & Cole & Sara & Jane & Emma & Jane & Emma \\
Erik & Jada & Eric & Hart & Dave & Eric & Jeff & Eric \\
Jada & Jane & Jada & Joey & Luke & Erik & Mary & Erik \\
Jack & Jeff & Jack & Jack & Kyle & Erin & Noah & Erin \\
Noah & Kyle & Noah & Maya & Cole & Evan & Sean & Evan \\
Kyle & Mary & Kyle & Jeff & Kyle & Jack & Jack \\
Owen & Noah & Owen & Mary & Noah & Owen & Jada \\
Seth & Owen & Seth & Jeff & Seth & Seth & Seth & Jake \\
Cole & Sean & Sean & Eric & Leah & Jada & Cole & Jane \\
Alex & Seth & Evan & Alex & Josh & Mary & Alex & Jeff \\
Hart & Abby & Hart & Erin & Erik & Hart & Hart & Joey \\
Mark & Anna & Mark & Jada & Jack & Mark & Mark & John \\
Joey & Ella & Joey & Erik & Mark & Joey & Joey & Josh \\
Emma & Emma & Erik & Emma & Will & Sean & Emma & Kyle \\
Ella & Eric & Jeff & Ella & Adam & Noah & Ella & Leah \\
\end{tabular}
A. Why can such a tree not be created using the Huffman encoding algorithm discussed in class?

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.

```
    E
   / \  
  A   B
 / \   \ 
C   D   C
```

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

```
/**
 * 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); // tighten children
    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;
    return root; // otherwise just return yourself
}
```

C. State the recurrence and the big-Oh for your solution.
\[ T(n) = 2T(n/2) + O(1) \]
\[ T(0) = O(1) \]
\[ T(n) \in O(n \log n) \]

**PROBLEM 5: (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 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.
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 \( n \)).

*textbf{Options from slowish to really good}:

- **Slowish**
  Sort the numbers and list the \( m \) largest
  \begin{align*}
  \text{sort: } & O(n \log n) \\
  \text{list: } & m \cdot O(1) \in O(m) \\
  \text{Total: } & O(n \log n)
  \end{align*}

- **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 \( m \).
  \begin{align*}
  \text{O}(m) & \text{ to build heap} \\
  (n - m) \text{ insertions and delete mins: } & (n - m)O(\log m) \\
  \text{O}(m) & \text{ to read out final elements of heap} \\
  \text{Total: } & O(m + (n - m) \log m)
  \end{align*}

- **Good**
  Build a max-heap from from the numbers and call \textit{deleteMax} \( m \) times.
  \begin{align*}
  \text{Build heap: } & O(n) \\
  \text{Calls to delete max: } & m \cdot O(\log n) \\
  \text{O}(n + m \log n)
  \end{align*}

- **Special**
  Sort using radix sort and then grab \( m \) smallest
  \begin{align*}
  \text{sort 32-bits: } & 32 \cdot O(n) \\
  \text{list: } & m \cdot O(1) \in O(m) \text{ Total: } O(d \cdot n) \text{ where } d \text{ is the number of digits}
  \end{align*}

- **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:

  \begin{verbatim}
  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)
  \end{verbatim}

  After running \texttt{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 \textit{good} heap example above is \( O(n) \) for small \( m \) and \( O(n \log n) \) as \( m \to n \).