Name: _____

This is a **50-minute** exam. Notes, calculators, or electronic devices are not allowed. Please put away all phones and smart watches. Follow code style writing guidelines for the course (excluding requirements for code documentation). The back of each page may be used if you run out of room. Code will be partially graded on efficiency.

1. (30 pts) Given a **HuffmanTreeNode** and two binary encodings as strings, finish the method below so it returns whether or not (true or false) the **plain text** of the two encoded strings are the same **length**. For example, if "101" was the encoding for "dog" and "001" was the encoding for "a", the method would return **false**. You may assume that the tree has at least one node. The description of the **HuffmanTreeNode** class can be found on the last page. Your method should **not** use recursion.

```
public bool CheckEncodeLength(HuffmanTreeNode tree, string a, string b){
     StringBuilder sb1 = new StringBuilder();
     StringBuilder sb2 = new StringBuilder();

     HuffmanTreeNode temp = tree;
     foreach(char c in a){
         if (c == '0')
         {
             temp = temp.Left;
         }
         else
         {
             temp = temp.Right;
         }
         if (temp.Right == null && temp.Left == null && temp.Frequency > 0)
         {
             sb1.Append(temp.Data);
             temp = tree;
         }
     }
     foreach (char c in b){
         if (c == '0')
         {
             temp = temp.Left;
         }
         else
         {
             temp = temp.Right;
         }
         if (temp.Right == null && temp.Left == null && temp.Frequency > 0)
         {
             sb2.Append(temp.Data);
             temp = tree;
         }
     }
     if (sb1.Length != 0) return sb1.Length == sb2.Length;
            return false;
}
```

2. (40 pts) Write the method below to find the most frequent occurring word in a set of documents. The documents are stored as a dictionary. The **key** of the dictionary represents the document identifier (an **int**). The **value** is also a dictionary, which contains the words of the document as keys and the occurrences of each word (word count) as values. The method should return a **KeyValuePair<string, int>** that has the most commonly occurring word and its total number of occurrences across all the given documents. You may assume that there is at least one word in each document.

   Hint: Use a dictionary to accumulate the total number of occurrences of each word.

```
public KeyValuePair<string, int> FindMostFrequentWord(Dictionary<int,
Dictionary<string, int>> documents){

     Dictionary<string, int> count = new Dictionary<string, int>();
     foreach(Dictionary<string, int> doc in documents.Values)
     {
         foreach(string word in doc.Keys)
         {
             if (count.ContainsKey(word))
             {
                 count[word]+=doc[word];
             }
             else
             {
                 count[word] = 1;
             }
         }
     }

     KeyValuePair<string, int> max = new KeyValuePair<string, int>("",0);
     foreach (string word in count.Keys)
     {
         if (count[word] > max.Value)
         {
             max = new KeyValuePair<string, int>(word, count[word]);
         }
     }

            return max;
 }
```

3. (30 pts) Below is a partial definition of a class implementing a trie. The **_isWord** field indicates whether this trie is the last character of a word. The array field stores the children such that the child labeled 'a' is in location 0, the child labeled 'b' is in location 1, etc. If there is no child with a given label, **null** is stored at that location.

   Using **recursion**, finish the CountWords method to return the total number of words in the trie that have at least **n** characters. You should assume that other members of the class may modify **_children** and **_ isWord** fields. You are not allowed to add any code outside of the CountWords method.

```
public partial class Trie
{
     private bool _isWord = false;
     private Trie[] _children = new Trie[26];

     public int CountWords(int n)
     {

         if (n == 0)
         {
             if (_IsWord)
             {
                 return 1;
             }
             else
             {
                 return 0;
             }
         }
         int total = 0;
         foreach(Trie t in _children)
         {
             if(t != null)
             {
                 total += t.CountWords(n - 1);
             }
         }
         return total;
     }
}
```

This page is only a reference page and may be torn off the test.

- **HuffmanTreeNode**:
  - **HuffmanTreeNode** Left: Gets the left subtree of the current node.
  - **HuffmanTreeNode** Right: Gets the right subtree of the current node.
  - **char** Data: Gets the text that is stored at this node
  - **int** Frequency: Gets the number of times the character of this node has occurred.