Programming Assignment 2 (Closest Pair, Sorting Strings, Priority Queue, and Huffman Coding)

Department of Computer Science, University of Wisconsin – Whitewater
Theory of Algorithms (CS 433)

Instructions For Submissions

- **Each group to have at most 2 members**. Although you can work individually, I encourage you to get a partner.
- One submission per group. **Mention the name of all members.**
- **Submit code and a brief report**. Submission is via Canvas as a single zip file.
- No need to include the algorithm description in the report.

---

1 Overview

We are going to:

- **Find a Closest Pair of Points**
- **Sort an Array of Strings using Radix-Sort**
- **Implement Huffman coding**.

To this end, **your task is to implement the following methods**:

- In **StringSorter**: `radixSort` and `countSortOnLowerCaseCharacters`
- In **HuffmanEncoder**: `encode`, `buildTree`, and `createTable`
- In **HuffmanDecoder**: `decode`

Additionally, **you will write a report** on the code in **ClosestPairOfPoints**.

The project also contains additional files which you do not need to modify (but need to use). You will use `TestCorrectness` to test your code. For each part, you will get an output that you can match with the output I have given to verify whether your code is correct, or not.

Output is provided separately in the `ExpectedOutput` file. Should you want, you can use `www.diffchecker.com` to tally the output.

2 Before You Start

2.1 C++ Helpful Tips

For C++ programmers, remember to use DYNAMIC ALLOCATION for declaring any and all arrays/objects. DO NOT forget to clear memory using *delete* (for objects) and *delete*[ ] for arrays when using dynamic allocation.

Remember to return an array from a function, you must use dynamic allocation. So, if you want to return an array x having length 10, it must be declared as **int *x = new int[10];**

2.2 Priority Queue

You do not need to write code for a priority queue, but you need to use it for Huffman Encoding. Your task will be to create a priority queue and use its main operations – `setPriority`, `getMinimumItem`, and `deleteMinimum`. You'll need to use a BinaryTreeNode priority queue , i.e., the items are binary tree nodes (which represent the nodes of the Huffman tree).

C++

- To create a priority queue for BinaryTreeNode type items, the syntax is:
  PriorityQueue<BinaryTreeNode*> pq;
- To set the priority of an item i to priority p, the syntax is pq.setPriority(i, p);
- To get the item with the minimum priority, the syntax is pq.getMinimumItem();
- To delete the item with the minimum priority, the syntax is pq.deleteMinimum();

Java

- To create a priority queue for BinaryTreeNode type items, the syntax is:
  PriorityQueue<BinaryTreeNode> pq = new PriorityQueue<BinaryTreeNode>();
- To set the priority of an item i to priority p, the syntax is pq.setPriority(i, p);
- To get the item with the minimum priority, the syntax is pq.getMinimumItem();
- To delete the item with the minimum priority, the syntax is pq.deleteMinimum();

C#

- To create a priority queue for BinaryTreeNode type items, the syntax is:
  PriorityQueue<BinaryTreeNode> pq = new PriorityQueue<BinaryTreeNode>();
- To set the priority of an item i to priority p, the syntax is pq.setPriority(i, p);
- To get the item with the minimum priority, the syntax is pq.getMinimumItem();
- To delete the item with the minimum priority, the syntax is pq.deleteMinimum();

2.3 Hashtable

You will be required to use hashtables for Huffman Encoding and Huffman Decoding. A hash table entry comprises of a *key* and *value* pair, whereby you can search the hash table for a particular key, and if it exists, retrieve the corresponding values back.

We use two hash tables – `charToEncodingMapping` for Huffman encoding and `encodingToCharMapping` for Huffman decoding. The first is used to get the encoding for each character, and then to encode the string, whereas the latter is used to decode the Huffman-encoded string back; hence, for the former, keys are of type `char` and values are of type `string` and vice-versa for the latter.

C++. To create a hash table object *ht* whose keys are of type *int* and values are of type *string*, the syntax is unordered_map<int, string> ht. You are not required to create a hash table as they have already been created, but it is good to know about them. What you will need are the following:

- To insert a value *v* with key *k*, the syntax is ht[k] = v;
- To check whether or not *ht* contains a key *k*, the syntax is: if(ht.find(k) != ht.end()). The statement inside the if evaluates to *true* if *ht* contains *k*.
- To retrieve the value *v* corresponding to a key *k*, the syntax is: string v = ht[k];

For your code, *ht* is either `charToEncodingMapping` or `encodingToCharMapping`, depending on whether you are working on encoding or decoding. Note that the type of *key* and *value* have to be modified accordingly. Refer to the `setPriority` and `insert` methods in PriorityQueue for usage of the above three; here, we are using a hash table to map the items in the priority queue to their respective heap index. Refer to `TestCorrectness` file to learn more stuff.

Java. To create a hash table object *ht* whose keys are of type *int* and values are of type *string*, the syntax is Hashtable<Integer, String> ht = new Hashtable<Integer, String>(). You are not required to create a hash table as they have already been created, but it is good to know about them. What you will need are the following:

- To insert a value *v* with key *k*, the syntax is ht.put(k, v);
- To check whether or not *ht* contains a key *k*, the syntax is: if(ht.containsKey(k)). The statement inside the if evaluates to *true* if *ht* contains *k*.
- To retrieve the value *v* corresponding to a key *k*, the syntax is: string v = ht.get(k);

For your coding, *ht* is either `charToEncodingMapping` or `encodingToCharMapping`, depending on whether you are working on encoding or decoding. Note that the type of *key* and *value* have to be modified accordingly. Refer to the `setPriority` and `insert` methods in PriorityQueue for usage of the above three; here, we are using a hash table to map the items in the priority queue to their respective heap index. Refer to `TestCorrectness` file to learn more stuff.

C#. To create a hash table object *ht*, the syntax is Hashtable ht = new Hashtable().[1] You are not required to create a hash table as they have already been created, but it is good to know about them. What you will need are the following:

- To insert a value *v* with a key *k*, the syntax is ht.Add(k, v);

[1] You can use generics if you use a Dictionary instead of Hashtable, which apparently is slower.

- To check whether or not *ht* contains a key *k*, the syntax is: if(ht.ContainsKey(k)). The statement inside the if evaluates to *true* if *ht* contains *k*.
- To retrieve the value *v* corresponding to a key *k*, the syntax is: String v = (String) ht[k]; Note the need to typecast over here due to the absence of generics.

For your coding, *ht* is either `charToEncodingMapping` or `encodingToCharMapping`, depending on whether you are working on encoding or decoding. Note that the type of *key* and *value* have to be modified accordingly. Refer to the `setPriority` and `insert` methods in PriorityQueue for usage of the above three; here, we are using a hash table to map the items in the priority queue to their respective heap index. Refer to `TestCorrectness` file to learn more stuff.

3 Closest Pair (30 points)

You don't have write code for this part. Check the methods `findClosestPair` and `findClosestPairHelper` in the **ClosestPairOfPoints** file. There are 8 parts in the methods – Part 0 through Part 7. Your task is to explain what each part is doing based on the lecture on closest pair of points, which can be found here: `https://drive.google.com/file/d/1be9Kco607XVAgx5ykdkqfX5WuX7TX0hK/view?usp=sharing`.

Explain briefly what each part achieves and report your explanations in a text file along with rest of the assignment. Please explain the purpose of each part in light of the algorithm and do not make statements like "this part calls this function with these parameters", or "in this part, we run a for-loop from $j = 1$ to $j = 7$", etc.

4 Sorting Strings of Lower Case English Letters (20 points)

We are going to sort an array of strings using radix sort; see the following for explanations:

- `https://drive.google.com/file/d/1Wwmtl6m-cCGEOi1WkqWffEytkd2jt6sc/view?usp=sharing`
- `https://drive.google.com/file/d/1sWoMGyPZFDmPcVfXEwiTeZPxtEeAeMST/view?usp=sharing`

The idea is to essentially treat each character in the string as a number between 1 - 26; thus, *a* is mapped to 1, *b* to 2, and so on all the way upto *z* being mapped to 26. This is achieved by subtracting 96 from the ASCII value of each character. Now, since a string may be shorter than another, we use 0 to treat a blank/null character. Hence, each character is represented in base-27 digit. Thus, if we compare the two strings *abcyx* and *abcz*, then we treat them as (1)(2)(3)(25)(24) and (1)(2)(3)(26)(0) respectively; the last 0 is padded essentially to make the strings have the same number of digits in base-27 notation.

We use the same idea as the radix sort algorithm on integers, but with the following changes:

- Make the following changes to the **radixSort** method:
  - *max* is the length of the longest string.
  - The loop for going over the digits now conceptually changes to going over the characters in each string. So, the outer while-loop changes to a loop that runs from $e = max - 1$ to $e = 0$; obviously, there is no need to multiply e by 10 anymore.
  - Within the inner for-loop, if e is greater than or equal to the length of $strings[i]$, then we let $digits[i] = 0$ because the string does not contain any character at index e (as it is shorter), else we let $digits[i] =$ the character at index e of $strings[i] - 96$.

- **countSortOnLowerCaseCharacters** method is essentially the **countSortOnDigtis** method. Make the following changes:
  - the *digitCount* array should have length 27. Make the necessary changes for the corresponding loops.
  - the *temp* array has to be a string array.

5 Huffman Coding (55 points)

Let σ (sigma) be the number of distinct characters. Let $alphabet[\,]$ be an array of size σ which stores the characters. Let $frequencies[\,]$ be an array of size σ which stores the frequency of each character, i.e., $frequencies[i]$ stores the frequency of $alphabet[i]$, $0 \le i \le \sigma - 1$.

You will need to use PriorityQueue and Hashing here; so, if you have not read how to use them, check out Section 2.2 and Section 2.3.

In Java or C#, to create a BinaryTreeNode with character c and value v, the syntax is: BinaryTreeNode node = new BinaryTreeNode(c, v);

In C++, to create a BinaryTreeNode with character c and value v, the syntax is: BinaryTreeNode *node = new BinaryTreeNode(c, v);

Implement the `encode`, `buildTree`, and `createTable` methods using the pseudo-codes.

```
BuildTree
• Create a BinaryTreeNode priority queue $PQ$
• For $i = 0$ to $\sigma - 1$, do the following:
    – create a binary tree node $x$ for the character $alphabet[i]$ and value $frequencies[i]$
    – set the priority of $x$ in PQ with priority $frequencies[i]$
• While the size of $PQ$ is greater than 1, do the following:
    – get the minimum binary tree node $min$ from PQ and remove it
    – get the second minimum binary tree node $secondMin$ from PQ and remove it
    – create a binary tree node $y$ with the character \0 and value the sum of the values
      of $min$ and $secondMin$
    – let $min$ be the left child of $y$ and $secondMin$ be the right child of $y$
    – set the priority of $y$ in PQ with priority $y$'s value
• return the minimum item from PQ
```

```
CreateTable
• If node's left child and right child are both null, then insert node's character as key
  and the string encoding as value into the hash table charToEncodingMapping
• Else do the following:
    – If node's left child $\neq$ null, CREATETABLE(node's left child, encoding + "0")
    – If node's right child $\neq$ null, CREATETABLE(node's right child, encoding + "1")
```

Encode

```
• root = BUILDTREE()
• CREATETABLE(root, “”);
• For $i = 0$ to $\sigma - 1$, do the following:
    – let $c = alphabet[i]$ and $str = getEncoding(c)$
    – encodingLength = encodingLength + frequencies[i] $*$ length of $str$
    – tableSize = tableSize + length of $str$ + 8
```

Implement the `decode` method using the following pseudo-code.

Decode

```
• Let $encode$ = “” and $decodedMsg$ = “”
• Let $n$ be the length of the encoded message
• For $i = 0$ to $n - 1$, do the following:
    – encode = encode + the character at index $i$ of $encodedMsg$
    – If the hash table encodingToCharMapping contains the key $encode$, then
        ∗ let $c$ be the value for the key $encode$ in encodingToCharMapping
        ∗ decodedMsg = decodedMsg + c
        ∗ encode = “”;
• return decodedMsg
```