LAB 3 ANALYSIS REPORT

by Chih Yuan Chan ID: 28A8CA

Date: 04/18/2023

1. PROBLEM DEFINITION

1.1 General approach

The objective of this Lab assignment is to build a Huffman Encoding Tree and use it to encode clear text and decode encoded binary strings.

The following abstract data structures are used to construct the Huffman Encoding Tree:

1.1.1 Primary data structures:

- **Min Heap**: A min heap is used to implement the priority queue. This will enable the highest priority element to be extracted from the Huffman Frequency Table and allow the Huffman Encoding Tree to be constructed efficiently, since the 2 highest priority nodes (with 2 lowest frequency values) are combined at each step of constructing the tree. The min heap is implemented using an array and the space complexity is **O(n)**, where n is the number of elements in the heap.

  Justification:
  - The use of a minimum heap structure will permit random retrieval of the highest priority element in O(log n) time. Given that multiple insertions (pushes) and deletions (pops) are made on the Priority Queue, the min heap will perform superior to using a direct array which will require implementing an sorting algorithm (best time of O(n log n) which will need to be executed multiple times for each time a push or pop occurs (to reorder the tree).
  - Other alternatives to using a minimum heap implementation include:
    - Unsorted List
    - Sorted List
    - Hash table

    However, the minimum heap implementation is considered the best balance between speed performance, complexity of implementation and memory overhead, for the following reasons:

    - Efficient time complexity for insert, extract-min, and decrease-key operations (O(log n)). By comparison, the Sorted List has poor time complexity for insert operation (O(n)) and requires additional time to maintain the sorted order of the list
    - Allows for frequent modifications (insertions / deletions) to the priority queue without excessive performance degradation. By comparison, the Unsorted List has poor time complexity performance for both insert and extract-min operations (O(n)) and is not suitable for large data sets or frequent modifications to the priority queue.
    - Simple implementation compared to most of the more complex alternatives
    - Small memory overhead compared to other alternative data structures, such as the hash table, which requires additional memory to store the key-value pairs and handle collisions.

  - Can be efficiently implemented using an array or binary tree data structure (refer below discussion of binary tree)

- **Implicit Binary Tree**: An implicit binary tree structure (using a hybrid dynamic array (Python List) and Binary Tree Structure approach) will be adopted for a minimum heap implementation of the Priority Queue as well as for the Huffman Encoding Tree.

  The space complexity of the Heap and Huffman Tree are both **O(n)** where n is the number of nodes in the tree.

  Justification:
  - A Huffman coding tree or Huffman tree is a full binary tree in which each leaf of the tree corresponds to a letter in the given alphabet. Therefore the Huffman Encoding Tree can be easily defined using the Binary Tree Structure. This is similar to a Heap tree.
  - Allows recursion to be employed, which closely mirrors the operations in traversing the tree, allowing for intuitive operations. As a result, the recursive code in this case is very simple and easy to implement.
  - The fast random access / retrieval in O(1) time from indexing of the array structure can be exploited to access the parent and child nodes.
  - For the Heap Tree implementation of the Priority Queue, "percolate down", push and pop operations during heapification can take advantage of the Binary Tree structure to conform to a heap format in O(log n) time.

1.1.2 Secondary data structures

- **Miscellaneous string storage:** Python Lists (dynamic array) and Python Dictionaries (associative array) were used to store Huffman encodings, string expressions during the parsing of the input clear text and binary encoding files, as well as the output text strings for the Huffman encoded and decoded strings.

  (NOTE: The dictionary structure was only used AFTER the Heap Tree and Huffman Tree had been fully traversed at least once.)

  Justification:
  - There is no reason to repeatedly traverse the Huffman Tree to retrieve the codes for required character symbols if these have already been obtained previously from the initial full tree traversal. Use of a Dictionary to store the results following full tree traversal eliminates the redundant operations.
  - The Python Dictionary and List data structures are modified array structures that permit random access and retrieval through the use of indexing in the fastest O(1) time.

1.1.3 Recursion Vs Iteration for traversing Huffman Tree

Refer to Section 1.7 for justifying a recursive approach.

The Huffman encoding tree is a binary tree, so its height is at most $\log_2(n)$, where n is the number of symbols in the input alphabet. Therefore, the maximum depth of the recursion stack is also at most $\log_2(n)$.

The amount of data pushed onto the stack in each recursive call is O(1) since it only includes node pointers.

Therefore, the space complexity of recursion on the Huffman encoding tree is proportional to the maximum depth of the recursion stack, which is O(log n).

Overall, the auxiliary space complexity of recursion on a Huffman encoding tree is O(log n), where n is the number of symbols in the input alphabet.

1.1.4 Sorting of Huffman Frequency Node characters

Huffman Frequency Node compound characters are formed as 2 highest priority nodes are merged during the formation of the Huffman Tree.
To implement the sorting of the characters so that the precedence can efficiently evaluated, I have chosen to use a simple Bubble Sort algorithm, for the following reasons:

- Data set is small so $O(n^2)$ worst case performance on large data sets will generally not occur. If the data set is large then a faster algorithm should be used.
- Minimal overhead O(1) space complexity since the sort is in-place.

1.1 Assessment of Algorithmic Time Complexity

1.1.1 Huffman Encoding – Mapping characters to their encoding

The time complexity of Huffman code mapping is effectively **O(n log n)**, where n is the number of unique characters. This is because for each unique character, the Huffman Encoding Tree needs to be traversed in O(log n) time in the worst case.

1.1.2 Min-heapify function

This function ensures that the array of Huffman Frequency nodes conforms to the minimum heap property by swapping out-of-order nodes.

This function first finds the node with the smallest value amongst the given node and its children. It then swaps the given node with the found minimum frequency value node (say 'min node'), and then calls the min-heapify function (recursively) over 'min node'. This ensures that the new value assigned to 'min node' conforms to the heap property in its subtree.

In the worst case, node swapping traverses through the depth of the tree. This produces a worst time complexity of **O(h)**, where h is the height of the tree. Alternatively, in terms of the number of nodes, **O(log n)**, where n is the number of elements in the heap.

1.1.3 Build-Heap function

This is the function building the heap from the Huffman Frequency Table. The function will need to rearrange each element to satisfy the heap property, and will call "min-heapify" O(log n) repeatedly. Since the min-heapify function is called whenever a new node is inserted to the heap, the time complexity of this function comes out to be **O(n log n)** where n is the number of elements in heap.

1.1.4 Heap Pop function

This function pops out the highest priority value (root) of the heap which has the lowest frequency value in the heap.

The root node is swapped with the last node followed by deleting the new last node (now containing minimum value). Next, min-heapify is called for the root node to maintain the heap property after the changes due to swapping.

The time complexity of this operation is **O(log n)**, where **n** is the number of elements, or **O(h)**, where **h** is the height of the tree (h is equal to log n for a complete binary tree).

1.1.5 Heap Push function

This function pushes a new element into the heap and rearranges the heap to maintain the minimum heap property.

To maintain the heap property, the node is percolated up by swapping with its parent node, as required.

The time complexity of this operation is also **O(log n)** since only the height of the subtree needs to be traversed.

1.2 Project Enhancements

The following project enhancements were incorporated:

- New module to build a Huffman Frequency Table from scratch using a user-supplied source text.
- Allowing for all characters, including lower and upper case, whitespaces and punctuation to be recognized and incorporated into encoding / decoding.
- Comprehensive checking of all edge cases when handling encoding and decoding file text input, including warning messages to detect illegal characters (when run in 'non-enhanced' mode) and identification of encoding sub-strings that could not be identified after traversing the Huffman Tree.
- Data Compression reporting advising the percentage compression achieved for each text expression based on standard ASCII encoding.
- Use of argument sub-parsers to permit commands from the terminal to run several combinations of module runs efficiently.

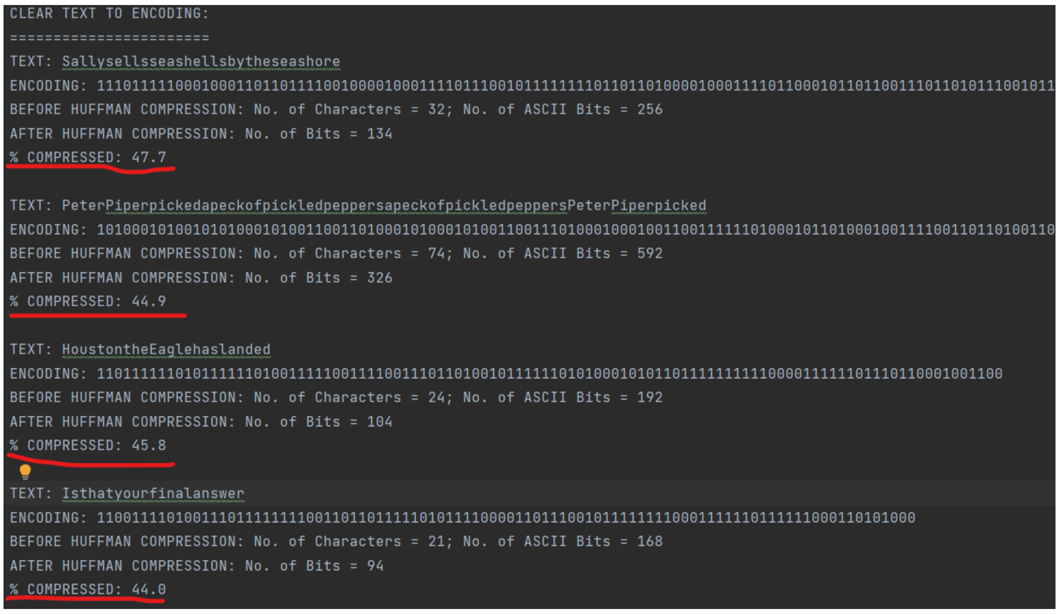
1.3 Data Compression of Huffman Encoding Vs Conventional Encoding

The Huffman Encoding is generally more efficient than conventional encoding such as ASCII because it accounts for a variable length of encoding rather than fixed length encoding.

The length of the encoding is dependent on the frequency of occurrence of the character, which is not accounted for in conventional encoding.

Here are some of the results:

Standard file input – standard Frequency Table; only recognizes alphabetical characters



```
CLEAR TEXT TO ENCODING:
=======================
TEXT: Sallysellsseashellsbytheseashore
ENCODING: 11101111100010001101101111001000010001111011100101111111101101101000010001111011000101101100111011010111001011
BEFORE HUFFMAN COMPRESSION: No. of Characters = 32; No. of ASCII Bits = 256
AFTER HUFFMAN COMPRESSION: No. of Bits = 134
% COMPRESSED: 47.7

TEXT: PeterPiperpickedapeckofpickledpeppersapeckofpickledpeppersPeterPiperpicked
ENCODING: 1010001010010101000101001100110100010100010100110011101000100010011001111110100010110100010011110011011010011
BEFORE HUFFMAN COMPRESSION: No. of Characters = 74; No. of ASCII Bits = 592
AFTER HUFFMAN COMPRESSION: No. of Bits = 326
% COMPRESSED: 44.9

TEXT: HoustontheEaglehaslanded
ENCODING: 110111111010111111010011111001111001110110100101111110101000101011011111111111000011111101110110001001100
BEFORE HUFFMAN COMPRESSION: No. of Characters = 24; No. of ASCII Bits = 192
AFTER HUFFMAN COMPRESSION: No. of Bits = 104
% COMPRESSED: 45.8

TEXT: Isthatyourfinalanswer
ENCODING: 1100111101001110111111110011011011111010111100001101110010111111110001111110111111000110101000
BEFORE HUFFMAN COMPRESSION: No. of Characters = 21; No. of ASCII Bits = 168
AFTER HUFFMAN COMPRESSION: No. of Bits = 94
% COMPRESSED: 44.0
```

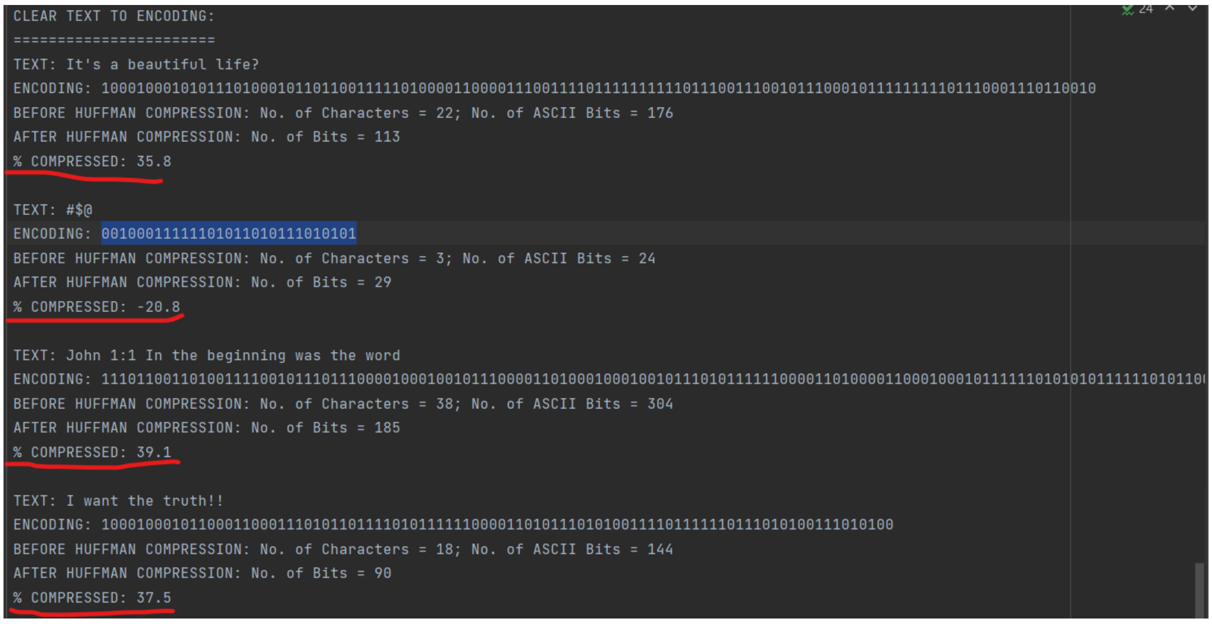
Fig 1. Output from Standard Mode on supplied Frequency Table

An average of above 40% data compression (vs ASCII 8 bit character encoding) was achieved which is more than half the conventional encoding approach.

This is not surprising since the above sentences feature relatively common occurring alphabets on average, which have higher frequencies and therefore higher compression (fewer bits to represent each character).

1.3.1 Enhanced mode – Custom Frequency Table, recognizes all characters

As an experiment, we now run the text files containing all types of different characters, which are recognized by the enhanced mode.



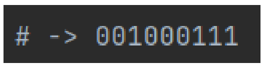
```
CLEAR TEXT TO ENCODING:
=======================
TEXT: It's a beautiful life?
ENCODING: 10001000101011101000101101100111110100001100001110011110111111111101110011100101110001011111111101110001110110010
BEFORE HUFFMAN COMPRESSION: No. of Characters = 22; No. of ASCII Bits = 176
AFTER HUFFMAN COMPRESSION: No. of Bits = 113
% COMPRESSED: 35.8

TEXT: #$@
ENCODING: 00100011111101011010111010101
BEFORE HUFFMAN COMPRESSION: No. of Characters = 3; No. of ASCII Bits = 24
AFTER HUFFMAN COMPRESSION: No. of Bits = 29
% COMPRESSED: -20.8

TEXT: John 1:1 In the beginning was the word
ENCODING: 11101100110100111100101110111000010001001011100001101000100010010111010111111000011010000110001000101111110101010111111010110
BEFORE HUFFMAN COMPRESSION: No. of Characters = 38; No. of ASCII Bits = 304
AFTER HUFFMAN COMPRESSION: No. of Bits = 185
% COMPRESSED: 39.1

TEXT: I want the truth!!
ENCODING: 100010001011000110001110101101111010111111000011010111010100111101111110111010100111010100
BEFORE HUFFMAN COMPRESSION: No. of Characters = 18; No. of ASCII Bits = 144
AFTER HUFFMAN COMPRESSION: No. of Bits = 90
% COMPRESSED: 37.5
```

Fig 2. Output from Enhanced Mode on Custom Frequency Table

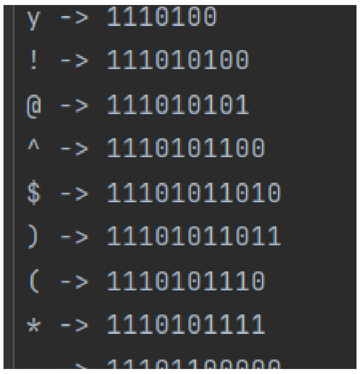
This time the achieved compression was above 35% for 3 out of 4 text sentences. This can be attributable to how each sentence now contains punctuation and upper-cased characters, which occur less frequently and reduce the compression efficiencies.

Notably, the second sentence which is "#$@' has a % compression of -20.8% which means that it is 20% bigger than a conventional encoding.

This is not surprising because all of the characters have Huffman encoding longer than 8 bits as they are the least frequently occurring characters (refer below figure). Notably, the '$' sign has 11 bits to represent it.



```
# -> 001000111
```



```
y -> 1110100
! -> 111010100
@ -> 111010101
^ -> 1110101100
$ -> 11101011010
) -> 11101011011
( -> 1110101110
* -> 1110101111
```

Key take-aways

Where encoded elements are well represented in the Huffman Frequency Table, Huffman Encoding can achieve very significant compression (more than twice) of conventional encoding approaches.

However, if the encoded characters are very infrequently occurring, then it is possible for Huffman Encoding to be worse than conventional encoding in very rare circumstances.

This also means that the source text code used to form the Huffman Frequency Table should be representative of the encoded texts. If the source is not representative (say using a Math Text book as a source to encode a poem from Shakespeare), then the compression results would be expected to decrease.

1.4 Discussion of Tie break scheme

The results of the Huffman encoding and decoding are critically dependent on the Tie Break scheme.

A change in the tie break scheme will result in different encodings / decodings, because the paths traversed along the Huffman tree to reach the encodings / decodings would be different.

As such, it is especially important for both the encrypter and decrypter to be using the same precedence rules.

1.5 Lessons learnt

I gained the following new knowledge from this lab assignment:

- Building a Huffman Frequency table from a raw text source
- Building a Huffman Tree using a hybrid array / binary tree approach (implicit Binary Tree)
- Traversing of a binary tree using Pre-order traversal
- Understanding the data compression efficiencies from using Huffman Encoding arising from explicitly accounting for the frequency of occurrences of characters
- Recursively traversing a binary tree
- Using subparsers to manage manage multiple groups of alternative arguments to be run from the command line interface

1.6 Future Considerations

As stated previously, recursion was adopted as the means to traverse the Heap tree and Huffman encoding tree instead of an iterative approach.

The recursive approach incurs the penalty of maintaining the additional stack frame calls and can lead to stack overflows if the depth of the Trees becomes too great.

As a future consideration, I would like to investigate the use of an alternative iterative approach to traverse the trees and to perform run-time assessments to compare the recursive vs iterative approaches.

I would also like to investigate the use of a hash map instead of a minimum heap to implement the priority queue due to the potential faster O(1) retrieval times. However, it is noted that this best time scenario would only occur if there were no collisions.