Lab 10 Post Lab Hasan Khan

Structure of Program, Time complexity

The following major steps are taken in huffmanenc:

1. The first while loop obtained from fileio.cpp. This code runs through the inputted file character by character, and has been modified to increment the frequency of each character into a frequency vector as they appear in the inputted file (where the character's asci value corresponds to the index of the frequency vector). Time complexity: O(n), where n is the length of the input file.
2. The next for loop creates a vector comprised of Huffman nodes (called "huffnodes"), and stores the frequency and character value in each node. Utilizes the frequency vector from the last step. Time complexity: O(n)
3. Heap heap1 created. "huffnodes" is passed to heap1. Time complexity: O(1)
4. Next, while loop in huffmanenc. Creates a Huffman tree on the heap using the process described in class. Two deleteMin() method calls are called, each with average complexity of O(log n). Then, and insert() method is called, which on average has time complexity of O(1) (since inserts happen at the end of the heap, and there is a high probability that the value being inserted will be a leaf, since half the nodes of the tree are leaves). Overall time complexity: O(1) + O(log n) + O(log n) → O(1) + O(2*log n).
5. Next, "encode" method called, sets prefix values for the huffman tree. Since the method has a recursive implementation, the general time complexity will be O(log n).
6. Next, for loop to print out letters and their expected prefix vals. Simple for loop that runs through the huffnode vector. Time complexity: O(n).
7. Next, nested for loop to print out encoded message. Outer loop runs 128 times through the vector, inner for loop runs based on huffnode size (n). Time complexity: O(128*n)
8. Last, for loop that runs through huffnode vector to calculate tree cost and compressed bit size. Time complexity O(n).

Approximate overall time complexity for huffmanenc: O(n) + O(n) + O(1) + O(1) + O(2*log n) + O(n) + O(128n)+ O(n) → O(131n) + O(2*log n)

For huffmandec:

1. While loop that stores prefix vals of each character into vector. Time complexity: O(n)
2. Next, for loop that goes through the prefixVec vector 128 times. At each iteration, calls populate Tree, a recursive method that creates a prefix code tree. Thus, time complexity is: O(128 * log n).
3. Next, while loop that concatenates all the bits from the inputted file into a single string. O(n) time complexity.

Approximate overall time complexity for huffmandec: O(n) + O(n) + O(128 * log n) → O(2n) + O(128*log n)

Space

Space and memory allocation depends on the amount and sizes of variables used in each program[1]. Calculating the sum of bytes gives us the memory allocations for each program.

Memory taken up by huffmanenc:

Variable	*# of Bytes*
Integer frequency vector (128)	4* 128 = 512 bytes
Int currentChar	4
Char g, j, y, d	4
hNode * h , hNode * dummy	8
huffnodes vector	depends on number of nodes
String prefix	depends on number of characters (+ 1)
Int dSymbols, oBits, symbolNo	12
Double tCost, cRatio, cBits	24
Loop integer iterators (of which there were 3)	12

Memory taken up by each node:

Variable	*# of Bytes*
Int frequency	4
Char charVal	1
hNode * left, * right	8
string prefix	depends on number of characters (+1)

Memory taken up by a heap

Variable	*# of Bytes*
heapVec vector	depends on number of nodes in this vector
int heap_size	4

Memory taken up by huffmandec not included to keep this report from being too long. Similar to huffmanenc.

[1] All byte values assumed under a 32 bit system