Holt Crews, jhc6we, 11/26/17, postlab10.pdf

Encoding:

For the encoding section, I began by creating an STL map for each character where the key is a unique character and the value associated with the key is the number of times a character occurs in the file. I also created a vector that has a single occurrence of each character in the file so that it would be easy to iterate through the keys in the map later on. The time it takes to create this map is linear with respect to the number of characters in the file (s). The time it takes to create the vector is linear (n) with respect to the number of unique characters which is has an upper bound so this is treated as constant time (1). I chose a hashtable to store the frequency because it would be a good way to store a key:value pair and it would allow me to access the character frequencies in constant time when creating the tree. I chose the vector just because it would be easy to iterate through.

I then iterated through each unique character and created nodes that contained the character and its frequency and put them on the min heap which was then sorted based on frequency. This will take log(n) time with respect to the number of unique characters which reaches a maximum so it can be treated as constant time. Once all of the characters were sorted in the heap, my code steps through the algorithm of converting a heap into a huffman tree which is a linear time with respect to the number of elements in the heap which has an upper bound of the number of possible unique characters and is therefore treated as constant time.

Finally, to encode and compress the message, each unique character has to be traversed through the tree and then is placed in a hash table with the character as the key and the prefix code as the value. Then every letter in the message is iterated through and is replaced by the prefix code in the hash table which is found in constant time with a lookup. In total, this process will take nlog(n) time for traversing the tree, where n is the number of unique characters, and replacing characters with their prefix codes will take a linear time s where s is the total number of characters in the file, a total of (s+nlog(n)). N, however, has an upper bound so it has a negligible effect on the time complexity.

The total time complexity of my algorithm is therefore (s+s+n+nlog(n)+log(n)) with n being a constant in the worst case as the number of printable characters available. nlog(n) is the greatest member of this time complexity but it is restricted by the total number of printable characters available, therefore the greatest hindrance to the speed of this algorithm is the number of characters in the file as this increases linearly and can go unto infinity.

Because of my use of hash tables, the space complexity is not entirely known, depending on the implementation of the hash table. Assuming every hash table has a load factor of 1 the space complexity of my algorithm will be ((n+x)(y)+n(4)+n(z)/8+n), where x is the number of non-leaf nodes in the huffman tree, y is the number of bytes in my huffman node class, and z is the average number of bits per prefix code. (n+x)(y) represents the huffman tree, n(4) is the hash table of frequencies, n(z)/8 is the hash table of prefixes, and n is the array of unique characters.

Decoding:

The first step to decoding a file that has been encoded via huffman encoding is to take the set of characters and their prefix codes and build a tree. This step will take nlog(n) time where n is the number of unique characters. After the tree has successfully been built, it has to be traversed as the encoded bits are read all the way through. This process will take slog(n)

time as the tree has to be traversed s times at a speed of log(n). My implementation of a decoder only used a huffman tree as a data structure, therefore the total time complexity of the decoding section of the lab was (nlog(n) + slog(n)). In the worst case, every printable character will appear in the file, putting an upper bound on the value of n. This limits the amount of time it takes to create the tree which is represented as nlog(n). This can therefore be treated as constant time and the resulting time complexity is now just (slog(n)+1).

As for the space complexity, it is really only limited to the size of the huffman tree and the number of characters in the input file. This space complexity can be represented as ((n+x)y + s) where n is the number of unique characters in the file, x is the number of non-leaf nodes in the huffman tree, y is the size of a member of the HNode class, and s is the size of the character string received as input. Before being put into the huffman tree, the input string is stored as a single string and is then iterated over. The space complexity is thus bounded by the total number of characters in the file which has no limit and increases linearly.