Sample Core Algorithm Overview

Stated Problem:

The purpose of this project is to create an algorithm using a common programming language (C, C++, Python or Java) to develop an effective and efficient Huffman code. At a minimum, the Huffman code should contain all twenty-six characters of the English alphabet. The project team will select a sample text article with a minimum length of one thousand characters. This article will be analyzed by the program to determine the character frequency distribution. The output of the code is the Huffman code of each character. For extra credit, use the Huffman code to encode and decode another text file (Gettysburg Address).

Algorithm Overview:

The Huffman code is created in the following manner:

1. Create a leaf for each character used in the sample file.
2. Dequeue the two leafs with the smallest values. Assign these two children to a new parent node. The value of the parent node is the sum of the two children.
3. Enqueue the parent node into the queue.
4. Repeat this process until one node remains on the root.

Operation time for the Huffman code depends on several factors. Is the frequency data sorted prior to running the program? What mechanism did the programmer use to implement the Huffman code? (priority queue, array, etc.) If the data has been sorted, then the operation time for running the program is O(n). If the data has not been sorted, then the operational time of the Huffman increases to O($nlogn$). For this project, the Huffman algorithm was implemented using a priority queue. An analysis of the pseudo code used as a basis for this program will help explain the structure of the program and the operational time of the Huffman algorithm.

Heap Sort

A[1 … n], n = length[A] = heap-size[A]

After Build-Heap; A[1] is the max. Exchange A[1] « A[n], and decrement

heap-size. A[1 … (n-1)} is almost a heap, it requires at most one

Heapify (A, 1).

The cost of Heapsort is O($n \log n$) as Build-heap is called once and Heapify is called at most n times.

```
Heapsort (A)
Build-Heap (A)
for i ¬ length[A] downto 2
    do exchange A[1] « A[i]
    heap-size[A] ¬ heap-size[A] - 1
    Heapify ( A, 1 )
```

MAX-HEAPIFY runs in O(h) for node of height h which is O($\log n$).

A Priority Queue maintains a set of S elements, each element associated with a key. Priority Queues are used for queuing systems and event simulation.

Methods:

Heap-Insert (S, x) :: S ¬ S È { x }

Maximum (S) :: returns max

Heap-Extract-Max (S) :: returns and removes max

Running Times:

Heap-Extract-Max : O (log n)

Heap-Insert : O (logn)

```
Heap-Extract-Max (A)
if heap-size[A] < 1
    then error "heap underlfow"
max ¬ A[1]
A[1] ¬ A[heap-size[A]}
heap-size[A] ¬ heap-size[A] - 1
Heapify ( A, 1 )
return max
Heap-Insert ( A, key )
heap-size[A] ¬ heap-size[A] + 1
i ¬ heap-size[A]
while i > 1 and A[Parent (i)] < key
    do A[i] ¬ A[Parent (i)]
        i ¬ Parent (i)
A[i] ¬ key
```

The pseudo code above shows all of the O(*) terms necessary to determine the time complexity for the HEAP data structure. Then, the priority queue is built upon the heap structure. So, ENQUEUE is simply HEAP-INSERT and DEQUEUE is HEAP-MAX-EXTRACT.

Coding of the extra credit Gettysburg Address data text file is then just N *{characters(I)--> Huffman code(I)} where the Huffman codes are pre-extracted from the frequency analyses (priority queue) and stored in an array of strings. On reading the input test text file, the strings are simply printed to screen or file as output which is, of course, very fast. The final coding should be in O(n). The decoding reads one Huffman code and then sorts through the loop of Huffman codes from most to least frequently occurring, and when a match is found, it outputs the corresponding ASCII character code. Thus, in the worst case, the decoding loop could take O(N characters * N Huffman codes) = O(N^2).