IITM-CS2800-2025 : Design and Analysis of Algorithms

Practice Problem Sheet #7 Released On : Mar 27, 2 PM

Part A - to be done in-class.

1. In this question, we will show that an algorithm that you are aware of (from PDS course) is actually a greedy algorithm and we will prove the correctness of the same.

   The problem is that of prefix-free coding. We want to send messages by encoding messages character-by-character by bit strings. One of the oldest such schemes is that of morse code. Different alphabet symbols may be encoded by different length bit strings.

   But there is one problem, if the encoded string for one character is a prefix of the encoded string of another character, then it creates ambiguity while decoding the message at the receiver's end. For example, the morse code suffers from this problem[1]. So, we would like to design prefix-free codes. We want to minimize the total length of the encoded string that is being sent. In other words, we want to minimize average the number of bits sent per alphabet symbol.

   What we have is information about the frequencies of occurrences (as a fraction of the length) of each letter in our message, and we need to choose, based on that,which character to encode using what length bit string. For example, if the letter `e` occurs with highest frequency, it makes sense to give a very short encoded (1 bit, say 0) to represent e, and the character with next second highest frequency can be given the string 10 as the encoding (why are we not using 1?).

   Let $A = \{a_1, a_2, \ldots, a_n\}$ be the set of alphabet symbols. Let f_i be the frequency of the symbol a_i. Given the frequencies of the alphabet symbols that occurs in a given message to be sent, the goal is to come up with an encoding scheme, H, that minimizes the cost (average number of bits per alphabet symbol):

   $$c(H) = \sum_{i=1}^{n} f_i \; \ell(a_i)$$

   where $\ell(a_i)$ is the length of the encoding string that is assigned by H represent the character a_i.

   (a) Design a prefix-free code of minimum length for the following scenario:

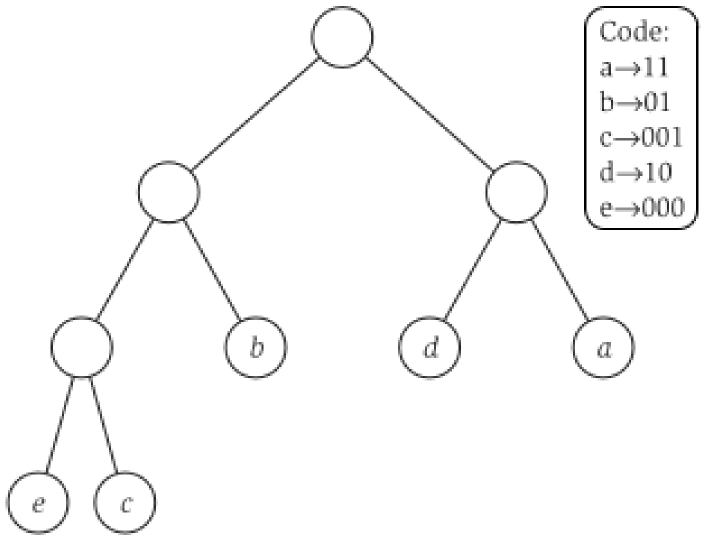
   $$A = \{a, b, c, d, e\}$$

   $$f_a = 0.32, f_b = 0.25, f_c = 20, f_d = 0.18, f_e = 0.05$$

   Note: There can be different prefix-free codes. So design a prefix-free code different from what is given as an example in the next part.

[1]because the code for E (•) is a prefix of the codes for I (••), S (•••), and H (••••)

(b) Any prefix-free binary code can be visualized as a binary tree[2] with the encoded characters stored at the leaves. The code word for any symbol is given by the path from the root to the corresponding leaf; 0 for left, 1 for right.



But this is not the optimal one. Can you find the optimal one for the above example?

(c) Conclude that in this representation, let T be the representation of the prefix-free code H, it suffices to optimize:

$$c(T) = \sum_{i=1}^{n} f_i \; depth(a_i)$$

where $depth(a_i)$ is the length of the root-to-leaf path for the symbol a_i.

(d) Argue that the binary tree corresponding to the optimal prefix-free code is always full.

(e) In 1951, as a PhD student at MIT, David Huffman developed the following greedy algorithm to produce such an optimal code:

Huffman: Merge the two least frequent letters and recurse

Given the simple structure of Huffman's algorithm, it's rather surprising that it produces an optimal prefix-free binary code. We will prove that this greedy choice is correct by an *exchange argument.* Prove the following:

Let a and b be the two least frequent alphabet symbols (breaking ties between equally frequent characters arbitrarily). There is an optimal code tree in which a and b are siblings in the tree. In fact, some thing stronger can be proved There is an optimal code tree in which a and b are siblings in the tree and have the largest depth of any leaf.

[2]Although they are superficially similar, binary code trees are not binary search trees; we don't care at all about the order of symbols in the leaves.

Start the argument like this: Consider an optimal prefix-free code represented by a tree T. Let d be its depth. Using part (d), argue (by induction on d) that it has at least two leaves at depth d that are siblings. Suppose those two leaves are not a and b but some other characters x and y. Can you think of an exchange argument? Formally derive the contradiction.

(f) Let us actually prove the correctness of the algorithm. The theorem is *Every Huffman code is an optimal prefix-free binary code.*

Start an inductive argument (on n) like this: If the message has only one or two distinct characters, the theorem is trivial, so assume otherwise. Let $f[1 \ldots n]$ be the original input frequencies, and assume without loss of generality that $f[1]$ and $f[2]$ are the two smallest frequencies. By previous part, a_1 and a_2 are (deepest) siblings in some optimal code for $f[1 \ldots n]$.

To set up the recursion/induction correctly. Define $f[n+1] = f[1] + f[2]$. Let T' be the Huffman tree for the frequencies $f[3, 4, \ldots, n+1]$. Buy induction hypothesis, T' is an optimal code tree for the smaller set of alphabets (only $n-1$ now). We simply replace the leaf labeled a_{n+1} with an internal node with two children labeled as a_1 and a_2. How do we show that this is correct? Express the cost of T' in terms of the cost of T. More specifically, prove that:

$$c(T) = c(T') + f[1] + f[2]$$

Thus minimising $c(T)$ is exactly same as minimising $c(T')$ and by induction hypothesis T' minimises the cost already. Hence T that we obtained above must also be an optimal prefix-free code tree for the frequencies $f[1, \ldots n]$.

(g) We need to maintain the frequencies in a data structure that supports finding the smallest element quickly, and insertions of new elements quickly. What data structure would you use to implement this? Describe how to get the running time of the above algorithm to $O(n \log n)$ by a suitable choice of a data structure.

2. We want to practice amortization in this exercise. Consider the problem of implementing a k-bit binary counter that counts upward from 0. We use an array $A[0, \ldots k-1]$ to store the bits. A binary number x that is stored in the counter has the LSB as the $A[0]$ and MSB as $A[k-1]$. Hence $x = \sum i = 0^{k-1} 2^i A[i]$. Initially $A[i] = 0$ for all i such that $x = 0$. To increment the counter, we need to add 1 (modulo 2^k) to it. We do it as follows:

```
INCREMENT(A)
{
    i=0
    while ((i < k) and (A[i] == 1))
        A[i] = 0
        i = i+1
    if (i < k)
        A[i] = 1
}
```

(a) What is the cost of `INCREMENT(A)` operation in the worst case? Show that if an algorithm uses n `INCREMENT(A)` operations, then the time taken must be $O(nk)$.

(b) This analysis is not tight. Not all bits get flipped in all incrememnt operations. To analyse better. Count the number of times a single bit get flipped. Prove that the total number of flips is at most $2n$ thus improving the analysis. Note that implies that `INCREMENT` incurs only constant time in amortized cost.