1 Huffman codes, continued

Recall that Huffman's algorithm is a greedy strategy for encoding a string of symbols. The result is a coding tree T in which every symbol from the input alphabet Σ is represented by a leaf node in T and the encoding cost of a particular symbol is simply its depth in the tree.

Or, more precisely, Huffman uses a simple convention that labels, in a consistent fashion, the left- and right-child branches in T with 0s or 1s. The codeword for a particular input symbol is then the sequence of 0s or 1s representing the sequence of left- or right-child steps as we traverse from the root to that leaf.

The cost of a tree T is then

$$\text{cost}(T) = \sum_{i=1}^{n} f_i \, d_i \ ,$$

where f_i is the frequency in the original message of the ith symbol, and d_i is its depth in T.

1.1 Huffman codes are optimal

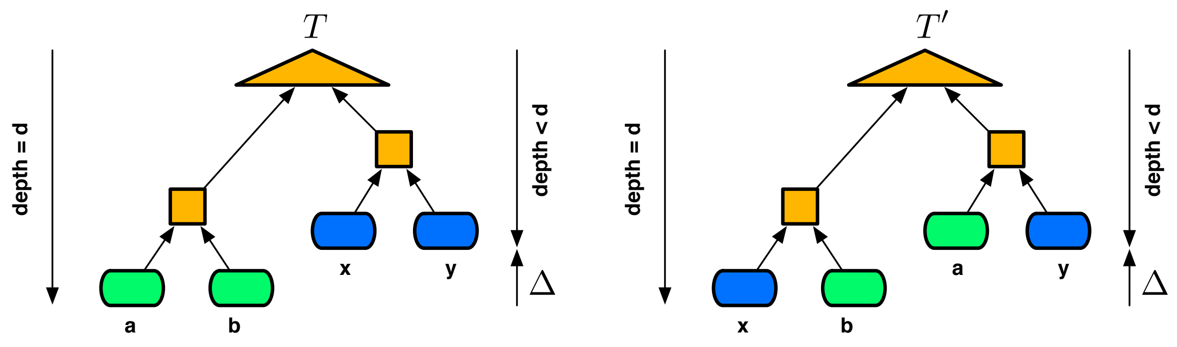
We now prove that Huffman encoding is optimal.

Lemma 1: Let x and y be two least frequent symbols (with ties broken arbitrarily). There is an optimal code tree in which x and y are siblings and their parent is the largest depth of any leaf.

Proof: Let T be an optimal code tree with depth d. Because T is an optimal code tree, it must also be a "full" binary tree, in which every tree node has either 0 or 2 children. It must be a full binary tree because if there were a tree node with only 1 child, then we could that node, connecting its one child directly to its one parent, and reduce the cost of all codewords in its subtree by 1, implying that T was not optimal.

Thus, because T is a full binary tree, it must have at least two leaves at depth d that are siblings.

Assume that these leaves are not x and y, but rather some other pair a and b, with x and y being located at some other depth $d' = d - \Delta$, for $\Delta > 0$. The following figure (left panel) illustrates this situation.



Now let T' be the code tree if we swapped x and a (right panel in the figure). The depth of x increases by some amount at most Δ, and the depth of a decreases by the same amount. Thus,

$$\text{cost}(T') = \text{cost}(T) - (f_a - f_x)\Delta \ .$$

However, by assumption, x is one of two least frequent symbols while a is not. This implies $f_a \geq f_x$, and thus swapping x and a cannot increase the total cost of the code.

Moreover, since T was an optimal code tree in the first place, swapping x and a cannot decrease the total cost of the code.

Thus, T' must also be an optimal code tree, which implies $f_a = f_x$.

A similar argument applies for swapping y and b, and thus carrying out both swaps yields an optimal tree T'' in which x and y as siblings and at maximum depth. $\square$

We can now prove that the greedy encoding rule of Huffman is optimal.

Theorem 1: Huffman codes are optimal prefix-free binary codes.

Proof: If the message has only one or two different symbols, the theorem is trivially true.

Otherwise, let $f_1, \ldots, f_n$ be the frequencies in the original message. Without loss of generality, let f_1 and f_2 be the smallest frequencies.

When the Huffman algorithm makes a recursive call, it creates a new frequency at the end of the list, e.g., $f_{n+1} = f_1 + f_2$. By Lemma 1, we know that the optimal tree T has symbols 1 and 2 as siblings.

Let T' be the Huffman tree for $f_3, \ldots, f_{n+1}$, which serves as our inductive hypothesis that T' is an optimal code tree for this smaller set of frequencies. We now prove that the tree T, on the larger set of frequencies, is also an optimal tree, in which we replace the tree node representing the frequency f_{n+1} with an internal node that has 1 and 2 as children.

To show that T is also optimal, we write down its cost and show that it is equal to the cost of T', an optimal but smaller tree, plus the cost of replacing one leaf in T' with a subtree representing leaves for symbols 1 and 2.

Let d_i be the depth of node i:

$$\begin{aligned}
\text{cost}(T) &= \sum_{i=1}^{n} f_i d_i \\
&= \left(\sum_{i=3}^{n+1} f_i d_i\right) + f_1 d_1 + f_2 d_2 - f_{n+1} d_{n+1} \\
&= \text{cost}(T') + f_1 d_1 + f_2 d_2 - f_{n+1} d_{n+1} \\
&= \text{cost}(T') + (f_1 + f_2) d_T - f_{n+1}(d_T - 1) \\
&= \text{cost}(T') + (f_1 + f_2) d_T - (f_1 + f_2)(d_T - 1) \\
&= \text{cost}(T') + (f_1 + f_2)(d_T - d_T + 1) \\
&= \text{cost}(T') + f_1 + f_2
\end{aligned}$$

Thus, the cost of T is no more than the cost of T' itself, which is minimal by assumption, plus the cost of adding one binary symbol to each of the codewords represented by the $(n+1)$th node in T'. □

2 On your own

1. Read Chapter 16.2-3 (Elements of the greedy strategy and Huffman codes)

3 Two more greedy algorithms

3.1 Insertion sort is an optimal but inefficient greedy algorithm

A greedy algorithm can be optimal, but not efficient. To illustrate this, we will consider the behavior of *insertion sort*. Recall that insertion sort takes $\Theta(n^2)$ time to sort n numbers and that we know a number of sorting algorithms that are more efficient, taking only $O(n \log n)$ time.

Insertion sort is a simple loop. It starts with the first element of the input array A. For each subsequent element j, it then inserts $A[j]$ into the sorted list $A[1..j-1]$.

```
INSERTION-SORT(A)
  for j=2 to n
     // assert: A[1..j-1] is sorted
     insert A[j] into the sorted sequence A[1..j-1]
     // assert: A[1..j] is sorted
}}
```

To see that insertion sort is correct, we observe that the following loop invariant, i.e., a property of the algorithm (either its behavior or its internal state) that is true each time we begin or end the loop.

Note that at the start of the `for` loop, the subarray $A[1..j-1]$ contains the original elements of $A[1..j-1]$ but now in sorted order. When $j = 2$ ("initialization") this fact is true because a list with one element is, by definition, sorted. When we insert the jth item into the subarray $A[1..j-1]$, we do so in a way that $A[1..j]$ is now sorted; thus, if the invariant is true at the beginning of the loop, it will also be true at the end of the loop ("maintenance"). Finally, when the loop terminates, $j = n$ and we have inserted the last element correctly into the sorted subarray ("termination"); thus, the entire array is sorted.

Now that we know insertion sort is correct, we can show that it is, in fact, an *optimal greedy algorithm*, meaning that (i) at any intermediate step, the algorithm always makes the choice that increases the quality of the intermediate state (greedy property) and (ii) it returns the correct answer upon termination (optimum behavior).

To begin, we first define a score function that allows us to build sorted sequences one element at a time. Clearly, if A is already sorted, score(A) must yield a maximal value, but it must also give partial credit if A is partially sorted and that credit should be larger the more sorted A is.

A sufficient score function is to count the number of sequential comparisons that violate the sorting requirement, i.e.,

$$\text{score}(A) = \sum_{i=1}^{n-1} \left(A[i] \leq A[i+1]\right) \quad ,$$

where we assume that the comparison operator is a binary function that returns 1 if the comparison yields true and 0 if it yields false. By definition, this property is true for all i in a fully sorted array, i.e., $A[1] \leq A[2] \leq \cdots \leq A[n]$, and so a fully sorted sequence will receive the maximal score of $n-1$. A sequence in reverse order will receive the lowest score of 0, and every other sequence can be assigned something between these two extremes.

With the score function selected, let's analyze the behavior of insertion sort under this function. An intermediate solution here is the partially sorted array. Given such an array, our "local" move is to take one element $A[j]$ and insert it into the subarray $A[1..j-1]$. Not all choices of where within $A[1..j-1]$ we put $A[j]$ will lead to a sorted list upon termination, and most of the possible choices of where to insert $A[j]$ are suboptimal.

Recall our loop invariant from above. For any input sequence A, when the loop initializes, it has score$(A) \geq 0$, where the lower bound is achieved by a strict reverse ordering. Because the sorted subarray's size grows by 1 each time we pass through the loop, so too does the number of correct sequential comparisons. That is, our loop invariant is equivalent to score$(A) \geq j-1$ where j is the loop index. And, when the loop completes, $j = n$ and score$(A) \geq n-1$. Thus, insertion sort is a kind of optimally greedy algorithm.

3.2 Linear storage media

Here's another good example of a simple greedy algorithm. Suppose we have a set of n files and that we want to store these files on a tape, or some other kind of linear storage media.[1] Once we've written these files to the tape, the cost of accessing any particular file is proportional to the length of the files stored ahead of it on the tape. In this way, tape access is very slow and costly relative to either magnetic disks or RAM. Let $L[i]$ be the length of the ith file. If the files are stored on the tape in order of their indices, the cost of accessing the jth file is

$$\text{cost}(j) = \sum_{i=1}^{j} L[i] \quad .$$

[1]Although tape memory is not often used by individuals, it remains one of the most efficient storage media for both very large files and for archival purposes. A linked list can be used to simulate a linear storage medium if the amount of data that can be stored in each node is limited; in fact, this kind of abstraction is precisely how files are stored on magnetic media.

That is, we first have to scan past (which takes the same time as reading) the first $j-1$ files, and then we read the jth file.

If files are requested uniformly at random, then the expected cost for reading one is

$$E[\text{cost}] = \sum_{j=1}^{n} \Pr(j) \cdot \text{cost}(j) = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{j} L[i] \enspace .$$

What if we change the ordering of the files on the tape? If not all files are the same size, this will change the cost of accessing some files versus others. For instance, if the first file is very large, then the cost of accessing every other file will be larger by an amount equal to its length. We can formalize and analyze the impact of a given ordering by letting $\pi(i)$ give the index of the file stored at location i on the tape. The expected cost of accessing a file is now simply

$$E[\text{cost}(\pi)] = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{j} L[\pi(i)] \enspace .$$

What ordering π should we choose to minimize the expected cost? Intuitively, we should order the files in order of their size, smallest to largest. Let's prove this.

Lemma 2: $E[cost(\pi)]$ is minimized when $L[\pi(i)] \leq L[\pi(i+1)]$ for all i.

Proof: Suppose $L[\pi(i)] > L[\pi(i+1)]$ for some i. If we swapped the files at these locations, then the cost of accessing the first file $\pi(i)$ increases by $L[\pi(i+1)]$ and the cost of accessing $\pi(i+1)$ decreases by $L[\pi(i)]$. Thus, the total change to the expected cost is $(L[\pi(i+1)]-L[\pi(i)])/n$, which is negative because, by assumption, $L[\pi(i)] \leq L[\pi(i+1)]$. Thus, we can always improve the expected cost by swapping some out-of-order pair, and the globally minimum cost is achieved when the files are sorted. $\square$

Thus, any greedy algorithm that repeatedly swaps out-of-order pairs on the tape will lead us to the globally optimal ordering. (At home exercise: can you bound the expected cost in this case?)

Suppose now that files are not accessed with equal probability, but instead the ith file will be accessed $f(i)$ times over the lifetime of the tape. Now, the total cost of these accesses is

$$\text{total-cost}(\pi) = \sum_{j=1}^{n}\sum_{i=1}^{j} f(\pi(j)) \cdot L[\pi(i)] \enspace .$$

What ordering π should we choose now? Just as when the access frequencies were the same but the lengths were different we would sort the files by their lengths, if the lengths are all the same but the access frequencies different, we should sort the files in decreasing order of their access frequencies. (Can you prove this by modifying Lemma 2?) But, what if the sizes and frequencies are both non-uniform? The answer is to sort by the length-frequency ratio L/f.

Lemma 3: total-cost(π) *is minimized when* $\frac{L[\pi(i)]}{f(\pi(i))} \leq \frac{L[\pi(i+1)]}{f(\pi(i+1))}$ *for all* i.

Proof: Suppose $\frac{L[\pi(i)]}{f(\pi(i))} > \frac{L[\pi(i+1)]}{f(\pi(i+1))}$. The proof follows the same structure as Lemma 2, but where we observe that the proposed swap changes the total cost by $L[\pi(i+1)] \cdot f(\pi(i)) - L[\pi(i)] \cdot f(\pi(i+1))$, which is negative. $\square$

Thus, the same class of greedy algorithms is optimal for non-uniform access frequencies.