CSCI 4041: Algorithms and Data Structures
Greedy Algorithms: Huffman's Algorithm
Part 1

Last revision April 6, 2020

Here's the agenda for this lecture.

Greedy algorithms.
History of Huffman's Algorithm.
What an encoding is.
Huffman's Algorithm.
An example of Huffman's algorithm.

What's greedy? Some algorithms work by making *decisions.* For example, the algorithm that searches a BST makes a series of decisions to move left or right down the tree, until it finds the node it's looking for, or until it runs out of nodes. These are easy decisions to make, but some algorithms make harder ones.

A *greedy algorithm* makes decisions in the following way: If there are many ways to make a decision, it always chooses the one that maximizes something good, or minimizes something bad. (*Good* or *bad* is determined by the algorithm's designer.) A greedy algorithm follows no grand strategy, but simply makes decisions based on whatever limited information is available to it when the decisions are required. (See Cormen pages 414–415.)

Some problems can't be solved by greedy algorithms. Others can be, but not optimally, so they get a solution, but not necessarily the best one. However, greedy algorithms are often easy to design, and when they work, they can be very efficient. In this course, we'll look at something called *Huffman's Algorithm* for text compression, which can be implemented as a greedy algorithm. It finds an optimal binary encoding for a string of characters. We'll also see at least one other greedy algorithm later in the course, when we look at algorithms for graphs.

Who's Huffman? He's David A. Huffman (1925–1999), an American computer scientist and electrical engineer. Huffman was a faculty member first at MIT, and later at the University of California, Santa Cruz, where he helped found its Computer Science department.

Huffman's algorithm can perform *text compression.* Given a string, Huffman's algorithm finds an optimally efficient way of encoding it as a sequence of *bits*: `0`'s and `1`'s. It does that by encoding frequently occurring characters by short sequences of bits, and infrequently occurring ones by long sequences.

Huffman invented his text compression algorithm as a doctoral student at MIT in 1951. He wrote it as a term paper for a course he was taking. A well-known (but possibly not true) story is that Huffman's professor didn't believe the algorithm worked, and gave him a low grade on his paper. Huffman contested his grade, and eventually convinced the professor that his algorithm was correct. In 1952, he published a revised version of his paper, called ''A Method for the Construction of Minimum-Redundancy Codes.'' Google can find it, if you're curious.

The problem. Here's the problem solved by Huffman's algorithm. Characters (I'll call them *chars* for short) are often represented as sequences of bits, all the same length. For example, this table shows how a few chars are represented in the well-known ASCII and Unicode character sets. (The symbol ␣ means a blank or space.) Each char is represented in eight bits, so it fits in a single byte.

CHAR	CODE
␣	00100000
A	01000001
E	01000101
M	01001101
N	01001110
T	01010100

Suppose we want to represent the string `MEET␣ME␣AT␣TEN`. (Perhaps this is an invitation to dinner or a movie.) To do that, we look up each char in the table, and find its eight-bit code. Then we concatenate the codes together, so we get a sequence of bits that starts as follows.

```
01001101 01000101 01010100 01010100 00100000 01001101 01000101 ...
    M        E        E        T        ␣        M        E
```

The string `MEET␣ME␣AT␣TEN` has 14 chars, so we can represent it in of 8 × 14 = 112 bits. Can we find a better representation for this string, one that lets us encode it in fewer bits? This isn't important for short strings like `MEET␣ME␣AT␣TEN`, but it may be important for long ones. For example, Tolstoy's *War and Peace,* one of the longest published novels ever written, has over 3.3 million chars. We could save a lot of computer memory if we could represent *War and Peace* more efficiently. We can do find a more efficient encoding by using short sequences of bits for chars that appear many times in the string, and long sequences for chars that appear few times.

In 1978, Bell Laboratories (the research division of a former American telephone company) examined some 213,553 chars of English text. (Anyone with a cheap laptop computer could examine far more text than this today, and in far less time—but computers were much less powerful forty years ago.) They found that letters appeared with the following *frequencies,* with larger frequencies indicating more frequently appearing letters.

CHAR	FREQ
␣	0.16
A	0.06
E	0.10
M	0.02
N	0.06
T	0.08

Just as we'd expect, the chars ␣ and `E` have large frequencies because they appear more often in English text. The char `M` has a smaller frequency because it appears less often.

Huffman's algorithm takes a table of chars and their frequencies, like this one. It constructs a *Huffman tree:* a binary tree that computes the optimal binary code for the chars, based on their frequencies. Using the Huffman tree, we can encode a string of chars in a minimum number of bits. We can also turn that string of bits back into the original string of chars again.

Huffman's Algorithm. Cormen's version of Huffman's algorithm appears on pages 431–432. My version of Cormen's version of Huffman's algorithm appears below.

```
00 HUFFMAN($C$)
01   $n = |C|$
02   $Q = C$
03   for $i = 1$ to $n - 1$
04     $z$ = MAKE-NODE()
05     $z.left$ = EXTRACT-MIN($Q$)
06     $z.right$ = EXTRACT-MIN($Q$)
07     $z.freq = z.left.freq + z.right.freq$
08     INSERT($Q$, $z$)
09   return EXTRACT-MIN($Q$)
```

What's happening here? In line 00, the procedure HUFFMAN takes a single parameter C, a set of chars. Each char in C has an attribute *freq,* its frequency. For example, `A`.*freq* = 0.06 and `E`.*freq* = 0.10. These are the char frequencies we showed in the table above, so that C acts like the table.

In line 01, we set n to the number of chars in C. The usual convention is to use the absolute value notation $|x|$ to mean the length of x, or the size of x, whatever x is.

In line 02, we make a min-priority queue Q and put the elements of C into it. I guess the only way we can tell it's a min-priority queue is that it's called Q, although Cormen tells us what it is in the text. Recall that a min-priority queue is one in which we remove the minimum element: in this case, the char with the minimum frequency. If we implement Q as a min-heap with n elements, as discussed in live lectures before the pandemic, then we can remove the char with the minimum frequency in $O(\log_2 n)$ time.

In line 03, we enter a loop that runs $n - 1$ times. Its variable i plays no part in the algorithm except as a counter. Why $n - 1$? It's because we'll repeatedly remove two elements from Q, and add one element back again, until only one is left. That needs a loop that runs one fewer time than the number of elements in Q.

As the loop runs, we'll build binary trees, one node at a time. In line 04, inside the loop, we make a new node z by calling a procedure called NEW-NODE. Each node has three slots. The slot *freq* is a frequency—we'll see how that's computed later. The slot *left* is the node's left subtree. The slot *right* is the node's right subtree.

By the way, Q will hold two different things. It will hold individual chars, which we'll use as external nodes in the binary trees we'll build. It will also hold internal nodes of the binary trees, like the one we made in line 04. As a result, Q is really a set of binary trees, or a *forest* of binary trees. (A set of trees really is called a forest, just like you'd expect.)

In line 05, we'll call a procedure called EXTRACT-MIN that removes the tree with the minimum *freq* slot from Q. The trees in Q are either single chars (external nodes) or else nodes themselves (internal nodes), and both of these have *freq* slots. It's those slots that EXTRACT-MIN looks at when it decides what to remove from Q. We'll set *z.left* to the tree we removed from Q.

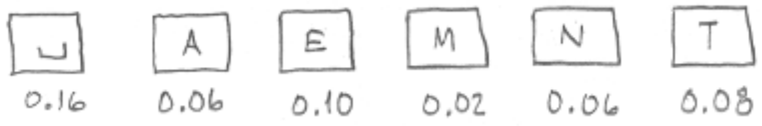
In line 06, we'll do the same thing as we did in line 05, removing another tree with the minimum *freq* slot and setting *z.right* to that tree. The effect of lines 05 and 06 is to take two trees from Q, the ones with minimum *freq* slots, and combine them into a new tree whose root is z.

In line 07, we set *z.freq* to the sum of the *freq* slots of the two trees we just removed. As a result, every tree in Q has a *freq* slot at its root that tells us the sum of all its *freq* slots. And in line 08, we'll put z back into Q again by calling INSERT.

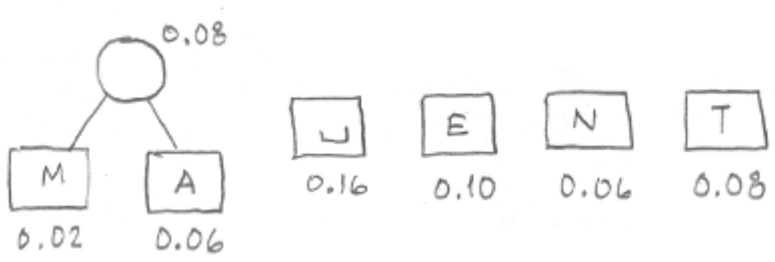
We execute lines 04 through 08 a total of $n - 1$ times. When the loop stops, there is only one binary tree left in Q. In line 09, we call EXTRACT-MIN to get that tree, and return it. I suppose we wouldn't really need to use EXTRACT-MIN here, but Cormen assumes this is the only way to do it.

Is this really a greedy algorithm? It is, because every time it decides which tree to remove from Q, it always removes the one with the smallest *freq* slot. If we think small *freq* slots are good, and large *freq* slots are bad, then we see that HUFFMAN really is making its choices in a greedy way.

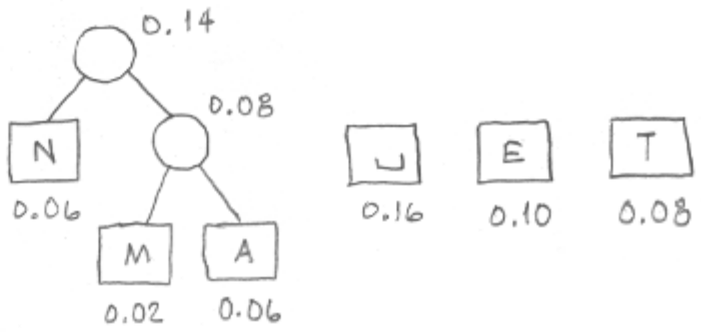
Example. Here's an example that will make all this clearer. We'll construct a Huffman tree for the characters in the string `MEET␣ME␣AT␣TEN`. Line 02 sets up Q so it contains these external nodes, shown as squares. Inside each square is a char, and the number next to it is its *freq* slot.



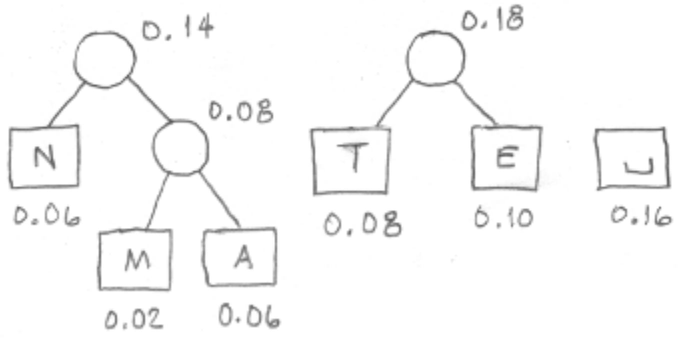
We'll now execute the loop five times, because there are six external nodes. The first time through the loop, we remove the external nodes `M` and `A` from Q, because they have the smallest *freq* slots. We then put them into a new binary tree, whose *freq* slot is the sum of the *freq* slots for `M` and `A`. We could have chosen `N` instead of `A`, because it has the same *freq* slot, but we just happened to get `A`. (We say that EXTRACT-MIN *resolves ties arbitrarily.*)



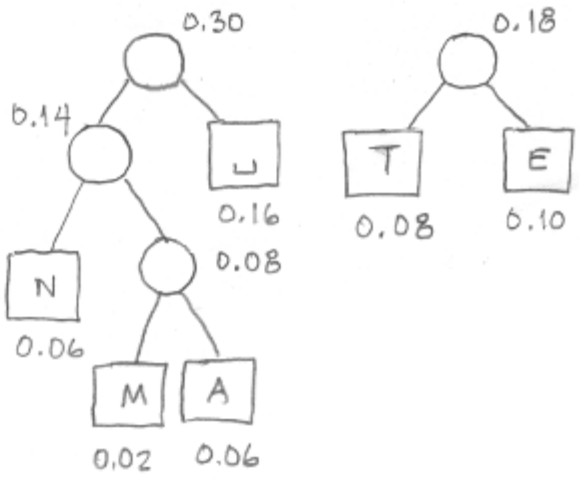
The second time through the loop, we get the external node `N` and the tree, again because they have the smallest *freq* slots. We put them into a new tree, whose *freq* slot is the sum of `N`'s *freq* slot and the sum of the tree's *freq* slot at its root. After that, *Q* contains the following:



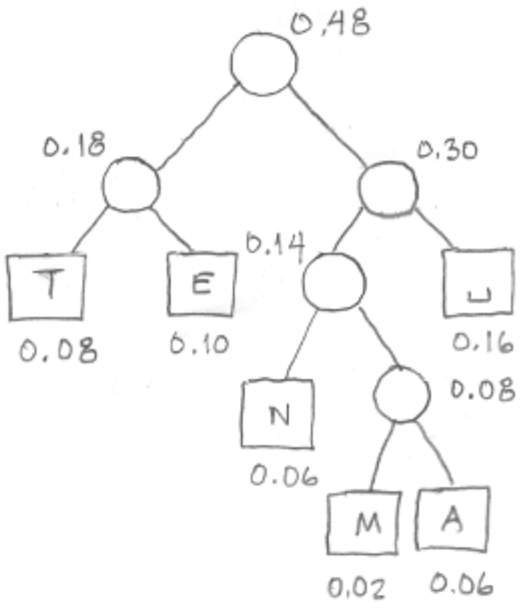
The third time through the loop, we get the external nodes `T` and `E`, with the smallest *freq* slots. We put them into a new tree and add the tree back to *Q*, so it looks like this.



The fourth time through the loop, we get the tree containing `N`, `M`, and `A`, and the external node ␣, both of which have the smallest *freq* slots. We put them into a new tree and add it to *Q*.



The fifth time through the loop, all that is left in *Q* are two trees, so there's no choice about what to do. We remove the trees from *Q*, put them into a new tree, and add that tree back again.



At this point, the loop stops. There's only one tree left in *Q*, so we remove it and return it. This is the Huffman tree that the procedure HUFFMAN has constructed.

In the following lecture, we'll see how to use the Huffman tree to encode the string `MEET␣ME␣AT␣TEN` as a sequence of bits having minimum length. We'll also see how to use the tree to do the reverse, turning the sequence of bits back into the string. Stay tuned for our next exciting episode.