CSCI 4041 Algorithms and Data Structures
Greedy Algorithms: Huffman's Algorithm
Part 2

Last revision April 6, 2020

The agenda goes like this.

How fast is the procedure HUFFMAN?
Encoding text with a Huffman tree.
Decoding text with a Huffman tree.
Why does HUFFMAN work?

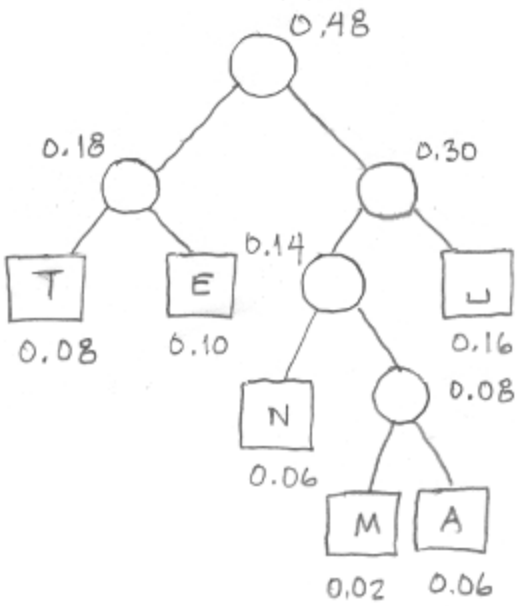
How fast is HUFFMAN? Here's the procedure HUFFMAN from the previous lecture. We'll refer to it occasionally in this lecture.

```
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)
```

It's easy to see how fast HUFFMAN runs (see Cormen page 433). We'll assume that the priority queue Q is represented as a min-heap. In line 01, we assume that it takes $O(1)$ time to determine n, the number of chars in C. In line 02, we can construct Q in $O(n)$ time using the procedure BUILD-MIN-HEAP, which you may remember from a pre-pandemic lecture. In lines 03–08, we have a loop that runs exactly $n - 1$ times. Inside the loop, we'll assume that we can make a new node at line 04 in $O(1)$ time, and that we can also update *freq* slots at line 07 in $O(1)$ time. That leaves only the procedures EXTRACT-MIN, which is called twice per loop iteration, and INSERT, which is called once per iteration. EXTRACT-MIN is also called once at the end, to retrieve the last tree from Q. If Q is a min-heap, then both run in $O(\log_2 n)$ time. Adding all these up according to the rules for O, we get $O(n \log_2 n)$ time.

Note that we don't actually have to implement Q as a min-heap: we could use some other data structure. If Q was implemented as a simple unordered array, then EXTRACT-MIN and INSERT would take $O(n)$ time, so HUFFMAN would run in $O(n^2)$ time. We might implement Q this way if we don't have to run HUFFMAN often, so we don't care how fast it is.

Encoding text with a Huffman tree. The last lecture ended with the procedure HUFFMAN making the following Huffman tree. We claimed that we could use the tree to encode the string `MEET␣ME␣AT␣TEN` efficiently. Let's see how to do that.



The first thing we need is a *bitstring.* As its name suggests, a bitstring is an ordered finite sequence of zero or more *bits:* `0`'s and `1`'s. For example, `0`, `1`, `001`, `110`, `10101`, etc. are bitstrings. There is also an *empty* bitstring, which we'll write as ϵ, the Greek letter *epsilon*. The empty bitstring is the bitstring of length zero, so there are no bits in it.

Now recall that we can have a *path* through a binary tree. For the purposes of this lecture, a path is a ordered finite sequence of zero or more nodes that starts at the root of a binary tree, and ends at an external node. For example, in the Huffman tree shown above, we can have a path consisting of four nodes: it starts at the root, moves right, then left, then left again, until it ends up at the external node containing `N`. There is a unique path to each external node in the tree.

Every path through a binary tree can be represented as a bitstring. Within the bitstring, `0` means *move left,* and `1` means *move right.* So, the path to the node containing `N` can be represented as the bitstring `100`. There is therefore a unique bitstring that identifies each external node in the tree.

Huffman's algorithm finds an optimal (minimum length) code for each char in the set *C*, given their frequencies. The chars end up as external nodes in the Huffman tree. The codes for these chars are just the bitstrings that specify paths to those external nodes. For example, the Huffman tree shown above gives us the following codes, which are (unsurprisingly) called *Huffman codes.*

CHAR	CODE
␣	11
A	1011
E	01
M	1010
N	100
T	00

Once we have the Huffman codes, we can use them to encode a string of chars as a bitstring. We find the Huffman code for each char, in order of their appearances, and assemble a bitstring from those codes. The bitstring has the minimum number of bits, given the frequencies of the chars. For example, our example char string `MEET␣ME␣AT␣TEN` can be encoded as the following bitstring.

```
1010  01  01  00  11  1010  01  11  1011  00  11  00  01  100
 M    E   E   T   ␣    M    E   ␣    A    T   ␣   T   E   N
```

This encoding requires only 35 bits. In the previous lecture, the ASCII and Unicode representations, which used eight bits for each character, needed 112 bits. But 35/112 = 0.3125, so we're now encoding `MEET␣ME␣AT␣TEN` in less than a third the number of bits we needed before.

We'll now write a procedure called MAKE-CODES that constructs the Huffman codes for each char in a Huffman tree. This procedure isn't explicitly in Cormen, although he talks about it in the text.

```
01  MAKE-CODES(code, r)
02      MAKING-CODES(code, ϵ, r)
```

At line 01, we see that MAKE-CODES takes two parameters. The first parameter is an array called *code*, whose indexes are chars, and whose elements are bitstrings. (We can use chars as indexes because they're represented as small nonnegative integers, just like ordinary array indexes.) If *c* is a char, then MAKE-CODES will set *code*[*c*] to the bitstring that is *c*'s Huffman code. The second parameter *r* points to the root of the Huffman tree.

MAKE-CODES does nothing for itself. At line 02, it calls a helper MAKING-CODES to do all its work. MAKING-CODES's parameter *code* is just the array from MAKE-CODES. Its parameter *b* is a bitstring, which is initially ϵ, and its parameter *p* points to a node in the Huffman tree *r*.

```
03  MAKING-CODES(code, b, p)
04      if IS-EXTERNAL(p)
05          code[p.char] = b
06      else
07          MAKING-CODES(code, b · 0, p.left)
08          MAKING-CODES(code, b · 1, p.right)
```

The operator '·' at lines 07 and 08 is the *concatenation* operator for bitstrings. (I just made it up for this lecture.) If b_1 and b_2 are bitstrings, then $b_1 \cdot b_2$ is the bitstring that begins with all the bits in b_1, followed by all the bits in b_2. Here are some examples of how it works.

```
    ϵ · ϵ = ϵ
    0 · ϵ = 0
    ϵ · 1 = 1
   01 · 1 = 011
101 · 100 = 101100
```

MAKING-CODES traverses the Huffman tree *p*, visiting each of *p*'s nodes exactly once. You may recall how to traverse a binary tree using a recursive backtracking procedure similar to this one, from whatever data structures course you took before CSCI 4041.

If *p* is an internal node (at line 06), then MAKING-CODES visits all nodes in *p*'s left subtree (at line 07), and visits all nodes in *p*'s right subtree (at line 08). As it visits nodes, it uses *b* to remember their paths. When it moves left, it adds a `0` to the end of *b* (at line 07), and when it moves right, it adds a `1` there (at line 08).

If *p* is an external node (the procedure IS-EXTERNAL tests for this at line 04), then *b* is the bitstring that represents a path to that node. As a result, MAKING-CODES gets the char from node *p*, which we'll call *p.char*, and records that char's bitstring in the array *code* at line 05.

When we call MAKE-CODES on a *code* array and a Huffman tree *r*, it constructs the Huffman codes for the chars in *r*, leaving them in *code*. Its helper MAKING-CODES visits each node exactly once, and we'll assume its operations run in $O(1)$ time. That means if there are n nodes in the Huffman tree *r*, then MAKE-CODES must run in $\Theta(n)$ time.

Decoding text with a Huffman tree. Of course it's not enough to just *encode* text using Huffman codes. We also need to go the other way, *decoding* a bitstring of Huffman codes back into the original text again. But this presents some difficulties.

When we encoded text using ASCII or Unicode, each char took exactly eight bits. That means the first eight bits tell you the first char, the second eight bits tell you the second char, etc. But when we encode text using Huffman codes, each char may be encoded with a different number of bits. How can you tell where the bits for one char end, and the bits for the next char start? For example, if someone gave you the bitstring `10100101001110100111101100110001100`, then how can you tell it's the Huffman encoding of `MEET␣ME␣AT␣TEN`? The answer is: use the Huffman tree again.

Suppose someone gives you a bitstring. They tell you it's the Huffman encoding of a char string, and ask you to decode it. Start at the root of the Huffman tree and look at the first bit. If the first bit is `0`, then move one node to the left in the tree. If the first bit is `1`, then move one node to the right instead. Now look at the second bit, and move left or right again. Do the same with the third bit, the fourth bit, etc. Keep going like this, following a path down the tree, until you reach an external node. That node tells you the first char. To get the rest of the chars, go back to the root of the Huffman tree again, and repeat the process with the remaining bits, until you run out of bits.

For example, if you get a bitstring whose first four bits are `1010`, then you'd move down the Huffman tree, starting at the root, going right, then left, then right, then left. In the tree shown above, you'd end up at an external node with the char `M` in it—so that's the first char. If the next two bits are `01`, then you'd go back to the root of the tree and move left, then right. You'd end up at an external node with the char `E` in it—so that's the second char, etc. I won't show a procedure for decoding a bitstring here, because I might ask you to write one for a homework assignment.

Why does HUFFMAN work? On pages 433–435, Cormen shows a proof that Huffman's algorithm produces an optimal encoding, in the way we've described. Remember that when someone says something is optimal, you should always find out what they mean by *optimal.* In this case it means that the algorithm produces codes for a minimum length encoding of a string, given a set of frequencies.

You don't have to know the proof for this course. Instead, we'll finish the discussion with a very informal verbal argument that Huffman's algorithm works.

We've seen that Huffman's algorithm uses a greedy strategy to build trees from which binary encodings can be derived. It builds these trees ''bottom up.'' That is: it begins by building trees from nodes whose chars have small frequencies. Later, it constructs trees by making them be subtrees of larger trees, until finally only one tree is left.

This has the effect of forcing chars with small frequencies toward the bottom of the tree, and forcing chars with large frequencies toward the top. As a result, the binary code for each char increases in length, depending on how deep the char is in the tree. Chars toward the top of the tree have short codes, while chars toward the bottom have longer codes. Although this isn't a proof, it suggests that Huffman's algorithm does what it's supposed to.