1 Greedy Algorithm

We continue our discussion of greedy algorithms.

1.1 Problem: Huffman Codes

Suppose, for simplicity, that our alphabet was a, b, c, d. If we have a string of letters that we wanted to encode, say, abcdacbdadccb, we would need to store it in the computer as a bunch of `0`'s and `1`'s. So, we can let $a =$ `00`, $b =$ `01`, $c =$ `10`, and $d =$ `11`. Then, we can encode the string above like so

a	b	c	d	a	c	b	d	a	d	c	c	b
00	01	10	11	00	10	01	11	00	11	10	10	01

With this encoding, it's easy to decode as long as we have the mapping. This has an issue, though: suppose we have the string aaaaaaaaaaabaacaa. Then, it's obvious that we're encoding `a` many more times than the other three letters. So, is there a way to map `a` to one bit instead of two bits so we can code the `a`'s with fewer bits?

Suppose $a =$ `0`, $b =$ `1`, $c =$ `00`, and $d =$ `01`. Then, we have soem ambiguity. If we had the string `00`, then this would map to either `aa` or `c`. So, how do we make this work?

Now suppose $a =$ `0`, $c =$ `10`, $b =$ `110`, and $d =$ `111`. Then, if we have the string `000010001011000011110100`, how do we decode this? Well, if we start reading from the left, then we note that:

- We first read the `0`. This can only map to `a` as the other characters start with `1`.
- Doing this three more times, we end up with `aaa`. Now, we consider the character `1`. There are three mappings where `1` starts first: `b`, `c`, and `d`. So, we read in the next character, `0`. Here, we note that this can only be `c` as `c` maps to a string `10` but the other two mappings start with `11`.
- We can repeat this process until we're done reading the string.

At the end, the decoded string is:

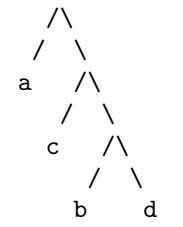
0	0	0	0	10	0	0	10	110	0	0	0	111	0	10	0
a	a	a	a	c	a	a	c	b	a	a	a	d	a	c	a

This is an example of **prefix-free encoding**: that is, no letters x, y is the encoding of x a prefix of the encoding of y.

Our problem statement is as follows: Given an alphabet where each letter x has a frequency[1] $f(x)$, we want a prefix-free encoding of the alphabet such that the encoding length $\sum_x f(x) \cdot |\text{encoding}(x)|$ is minimized.

1.1.1 Rewording the Problem

Our first observation is that there is a binary tree representation of the prefix-free encoding, where taking the left branch is the `0` bit and the right branch is the `1` bit. Then, you can place letters in locations corresponding to their encoding. For example, the tree representation of the above prefix-free encoding is:



[1] The number of times it appears in the string.

So, for a prefix-free encoding, the letters are at the leaves of the binary tree. So, we can rephrase the problem

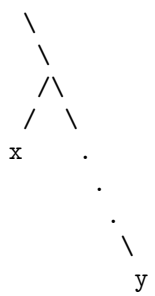
$$\sum_x f(x) \cdot \text{depth}(x)$$

where the depth is the same as the length of the encoding of some x.

1.1.2 Easy Case

Suppose we fix the tree structure. That is, our tree must look like

We know that we have four letters where one letter has depth 1, one letter has depth 2, and two letters have depth 3. Now, the best way to assign letters is to assign the *high frequency* letters to the smaller depths. To see this, we note that



To compare x and y, we want to compare

$$d_1 f(x) + d_2 f(y)$$

with

$$d_2 f(x) + d_1 f(y)$$

where d_i is the depth. Then, we can factor this out like so

$$(d_1 - d_2)(f(x) - f(y))$$

If x has lower frequency and it was stored at the smaller depth, then switching them would decrease the total expense. So, if we have a fixed tree, there is a greedy algorithm:

- Sort letters by frequency.
- Sort location by depth.
- Assign letters with higher frequencies to lower depth.

1.1.3 Observation

Two of the deepest elements are siblings. Note that the deepest elements correspond to the letters with little frequency. Thus, the key insight here is

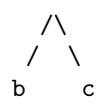
The two least frequent letters in the alphabet might as well be siblings.

1.1.4 An Example

Suppose we had:

- 30 copies of `a`
- 15 copies of `b`
- 25 copies of `c`
- 50 copies of `d`
- 65 copies of `e`

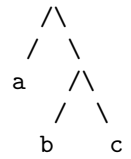
We can assume that `b` and `c` are siblings since they have the lowest frequencies. So:



Next, note that we can call the parent of `b` and `c` *`b` or `c`*, and thus treat it as a "different letter." We can also give this "different letter" a value of `40`, as `b` and `c` appear 40 times. Thus, we now have the alphabet:

- 30 copies of `a`
- 40 copies of `b` or `c`
- 50 copies of `d`
- 65 copies of `e`

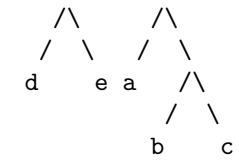
Notice that we can repeat the process again. In particular, notice that `b` or `c` *and* `a` are siblings since they have the lowest frequencies. So:



Again, we can call the parent of `a` and `b` or `c` *`a` or `b` or `c`*, and treat it as a "different letter" with the value being 70. Thus, we now have the alphabet:

- 70 copies of `a` or `b` or `c`
- 50 copies of `d`
- 65 copies of `e`

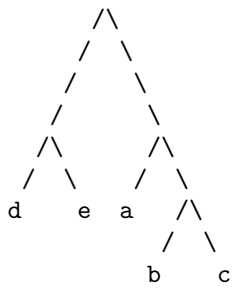
We now repeat this process. Here, we see that `d` and `e` are siblings since they have the lowest frequencies. So:



We call the parent of `d` and `e` *`d` or `e`* and give it a value of 115. Thus, we now have the alphabet:

- 70 copies of `a` or `b` or `c`
- 115 copies of `d` or `e`

But since there are only two "letters," we can just pair them up:



1.1.5 Algorithm

The algorithm is as follows, where

- S is the alphabet.
- f is the frequency function.

```
HuffmanCode(S):
    while |S| > 1:
        Let x, y be least frequent elements of S
        Create z with f(z) = f(x) + f(y)
        Make x, y children of z
        Remove x, y from S
        Add z

    Return element in S
```

If S has size n, then the runtime is $\mathcal{O}(n^2)$. This is because we need to iterate over the `while` loop, and for each iteration we need to do a linear scan. But, we can make use of a *priority queue* to optimize our algorithm.

```
OptimizedHuffmanCode(S):
    Insert S into priority queue Q
    while |Q| > 1:
        Let x = deleteMin(Q)
        Let y = deleteMin(Q)
        Create z with f(z) = f(x) + f(y)
        Make x, y children of z
        Q.Insert(z)

    Return deleteMin(Q)
```

This runs in $\mathcal{O}(n \log(n))$.

1.1.6 Takeaways

There are some takeaways from this.

- To find a greedy decision procedure, a good thing to do first is to find a safe first step. The safe first step in this algorithm is that the two elements might as well be siblings. Once you made that decision, you need to reduce the problem back to a copy of the original problem.

- Often, in order to turn something into a greedy algorithm, you might need to rearrange the way the problem is phrased; i.e. rephrase the problem. The original phrasing of this problem – finding the prefix-free encodings of letters of this alphabet which minimize the total encoding length – is awkward to phrase as a greedy algorithm. The idea that we can pair two elements as siblings would be incredibly awkward to phrase under the original problem statement.