



Master algorithms together on [Binary Search](#)! Create a room, invite your friends, and race to finish the problems.

Daily Coding Problem

[Blog](#)

Daily Coding Problem #261

Problem

This problem was asked by Amazon.

Huffman coding is a method of encoding characters based on their frequency. Each letter is assigned a variable-length binary string, such as `0101` or `111110`, where shorter lengths correspond to more common letters. To accomplish this, a binary tree is built such that the path from the root to any leaf uniquely maps to a character. When traversing the path, descending to a left child corresponds to a `0` in the prefix, while descending right corresponds to `1`.

Here is an example tree (note that only the leaf nodes have letters):



With this encoding, `cats` would be represented as `0000110111`.

Given a dictionary of character frequencies, build a Huffman tree, and use it to determine a mapping between characters and their encoded binary strings.

Solution

First note that regardless of how we build the tree, we would like each leaf node to represent a character.

```
class Node:
    def __init__(self, char, left=None, right=None):
        self.char = char
        self.left = left
        self.right = right
```

When building the tree, we should try to ensure that less frequent characters end up further away from the root. We can accomplish this as follows:

- Start by initializing one node for each letter.
- Create a new node whose children are the two least common letters, and whose value is the sum of their frequencies.
- Continuing in this way, take each node, in order of increasing letter frequency, and combine it with another node.
- When there is a path from the root to each character, stop.

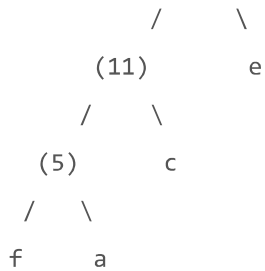
For example, suppose our letter frequencies were {'a': 3, 'c': 6, 'e': 8, 'f': 2}.

The stages to create our tree would be as follows:

```
(5)
 /  \
f    a
```

```
      (11)
     /    \
    (5)    c
   /  \
  f    a
```

```
(19)
```



In order to efficiently keep track of node values, we can use a priority queue. We will repeatedly pop the two least common letters, create a combined node, and push that node back onto the queue.

```

import heapq

def build_tree(frequencies):
    nodes = []
    for char, frequency in frequencies.items():
        heapq.heappush(nodes, (frequency, Node(char)))

    while len(nodes) > 1:
        f1, n1 = heapq.heappop(nodes)
        f2, n2 = heapq.heappop(nodes)
        node = Node('*', left=n1, right=n2)
        heapq.heappush(nodes, (f1 + f2, node))

    root = nodes[0][1]

    return root

```

Each pop and push operation takes $O(\log N)$ time, so building this tree will be $O(N * \log N)$, where N is the number of characters.

Finally, we must use the tree to create our encoding. This can be done recursively: starting with the root, we traverse each path of the tree, while keeping track of a running string. Each time we descend left, we add 0 to this string, and each time we descend right, we add 1. Whenever we reach a leaf node, we assign the current value of the string to the character at that node.

```

def encode(root, string="", mapping={}):
    if not root:

```

```
        return

    if not root.left and not root.right:
        mapping[root.char] = string

    encode(root.left, string + "0", mapping)
    encode(root.right, string + "1", mapping)

    return mapping
```

As a result, the encoding for the tree above will be {'f': '000', 'a': '001', 'c': '01', 'e': '1'}.

It will take, on average, $O(\log N)$ time to traverse the path to any character, so encoding a string of length M using this tree will take $O(M \log N)$.

© Daily Coding Problem 2019

[Privacy Policy](#)

[Terms of Service](#)

[Press](#)