COMP S265F Design and Analysis of Algorithms
Lab 4: Huffman Codes

In this lab, we implement the Huffman Code algorithm for n characters. Our first version runs in $O(n^2)$ time. We then introduce the `heapq` module in Python, which implements a min-heap (also known as a priority queue), and use it to improve our Huffman Code algorithm to $O(n \log n)$ time.

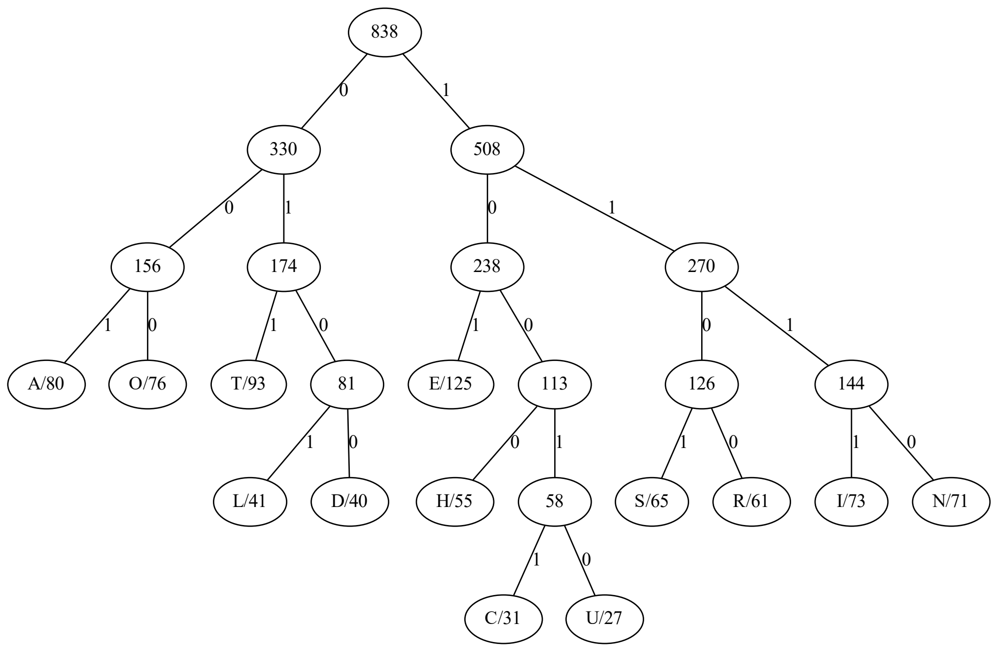
1. Construct a Huffman tree

The input of our program is a list of characters and their frequencies, and the output is a Huffman tree printed by the `graphviz` module.

Sample Input

```
E 125
T 93
A 80
O 76
I 73
N 71
S 65
R 61
H 55
L 41
D 40
C 31
U 27
```

Sample Output



Code template. You may start with the following template. After completion, you can be run it using `python 00huffman.py < freq.txt`:

00huffman.py

```python
from sys import stdin
from graphviz import Graph

num_node = 0

class Node:
    def __init__(self, freq, ch, left, right):
        global num_node
        self.id = str(num_node)
        num_node += 1
        self.freq = freq
        self.ch = ch # None for internal nodes
        self.left = left
        self.right = right
```

```python

class Huffman:
    def __init__(self, ch_freq):
        self.h = [] # a list for (freq, node)
        self.last = None # keep the last created Note
        self.tree = Graph()

        # Your task:
        # Create leaf nodes for the Huffman tree
        # and store them in the list self.h

        # Construct the Huffman tree
        while len(self.h) >= 2:
            # Your task:
            # Select the node with minimum frequency and remove it
            # Select another node with minimum frequency and remove it
            # Add the combined new node to self.h

    def showTree(self):
        self.tree.view()

def main():
    ch_freq = []
    for line in stdin:
        ch, freq = line.split()
        ch_freq.append( (ch,int(freq)) )

    huffman = Huffman(ch_freq)
    huffman.showTree()

if __name__ == "__main__":
    main()
```

In the `main` function of the above template, we read each character `ch` and its frequency `freq`, and store the tuple `(ch, freq)` in a list `ch_freq`. Then, we instantiate a `Huffman` object stored in the variable `huffman`.

The constructor of `Huffman` will read the character frequency list `ch_freq`, and immediately run the *Huffman Code Algorithm* to create the Huffman tree. The Huffman tree will be formed by a number of `Node` objects. Each `Node` object has the following instance variables:

- `id`: a unique id in string
- `freq`: frequency for the tree node
- `ch`: character for leaf node; `None` for an internal node
- `left`: left child for an internal node; `None` for a leaf node
- `right`: right child for an internal node; `None` for a leaf node

All these nodes will be stored as a tuple `(freq, node)` in the instance variable `h` of the `Huffman` object, where `freq` is the frequency of the node object `node`. We will also keep the latest created node `last` such that after constructing the Huffman tree, `last` will be the root node.

Finding min/max tuple in a list. Below are some example code illustrating how to find a minimum/-maximum from a list of tuples:

01tuples.py

```python
a = [(1,20), (1,10), (2,10), (2,20)]
print(min(a), max(a))
print(min(a, key = lambda t: t[0]), max(a, key = lambda t: t[0]))
print( a.index(min(a)), a.index(max(a)) )
```

By default, the built-in `min` and `max` functions compare *every* item in the tuples one by one, i.e., 1st, and then 2nd. Therefore, `min(a)` will give `(1,10)`; while `max(a)` will give `(2,20)`.

You may modify this behavior by specifying the `key` function. Here, we define a *lambda function*, which is a function without function name. The function `lambda t:t[0]` takes a single argument `t` (the tuple) and returns the first item `t[0]` in tuple `t`. Therefore, `min(a, key = lambda t:t[0])` will give `(1,20)` (the first minimum tuple); while `max(a, key = lambda t:t[0])` will give `(2,10)` (the first maximum tuple).

You can use the list method `index(item)` to get the index of `item` (its first occurrence) in the list. In list `a`, you can use `a[ind]` get the item at index `ind`, and use `del a[ind]` to delete it.

Huffman tree by `graphviz`. The instance variable `tree` of the `Huffman` class is a `graphviz`'s `Graph` object, which has the following methods:

- `node(`x`, label=`L`)`: Create a node with id x and displayed it as L.
- `edge(`x`, `y`, label=`L`)`: Create an edge connecting nodes with id x and y and add an edge label L.
- `view()`: Display the graph as a PDF file.

We can view the constructed Huffman tree by calling the `showTree()` method of the `Huffman` object.

2. Reformat the Huffman tree by `graphviz`

By default, `graphviz` displays the tree nodes of the same depth in their creation order. To reformat the Huffman tree such that the left child nodes are shown on the left, we can first construct the tree structure with the root `self.last`, and then use the following recursive function `traverse(node)` to create `self.tree` during the tree traversal from the root `self.last`:

```python
def traverse(self, node):
    if node.left == Node:
        # Your task:
        # Create the leaf node
    else:
        # Your task:
        # Create the internal node with edges to its left and right children
        self.traverse(node.left)
        self.traverse(node.right)
```

3. Build the Huffman code

We can create a dictionary `code` as an instance variable of the `Huffman` class, and extend `traverse(node, c)` to include binary code `c` for a tree node `node`:

1. In the `Huffman` constructor, add the definition `self.code = {}`, and update `self.traverse(self.last, "")` to begin with an empty binary code `""`.
2. Update the instance method `traverse(self, node, c)` such that
   - when `node` is a leaf node, `self.code[node.ch] = c`.
   - when `node` is an internal node, it appends `0` or `1` to the binary code `c` in the recursive call.
3. Create an instance method `getCode(self)` to return the dictionary `code`.
4. In `main()`, print the dictionary `huffman.getCode()`.

4. Improve the time complexity using `heapq`

The `heapq` module implements the min-heap data structure, which is a binary tree for which every parent node has a value less than or equal to any of its children. Thus, the root of a min-heap is the smallest item.

You can start with an empty list or a list of items; if the list `h` is not empty, then you need to call `heapq.heapify(h)` to transform `h` to a min-heap, and this transformation takes $O(n)$ time, where n is the number of items in list `h`.

The min-heap supports the following operations in $O(\log n)$ time:

- `heapq.heappush(h, item)`: Push the item `item` to the min-heap `h`.
- `heapq.heappop(h)`: Pop and return the smallest item from the min-heap `h`. To access the smallest item without popping it, use h[0].

More details about `heapq` can be found in https://docs.python.org/3/library/heapq.html.

In our program, `Huffman`'s instance variable `h` is a tuple containing `Node` that cannot be compared. We can define the following new class `Tuple` such that its less-than function `__lt__(self, other)` is defined:

```
class Tuple:
    def __init__(self, val):
        self.val = val

    def __lt__(self, other):
        return self.val[0] < other.val[0]
```

Then using a min-heap of `Tuple` objects for `self.h` in the `Huffman` class will improve the time complexity of the Huffman code algorithm from $O(n^2)$ to $O(n \log n)$.

5. Exercises

Question 1. Given a set of characters A, B, C, D, E, F and their corresponding frequencies. Construct the Huffman code for these characters by drawing the code tree.

Character	A	B	C	D	E	F
Frequency	30	8	7	6	5	1

Question 2. Keith drives a motorcycle from City A to City B along a highway. His bike's gas tank, when full, holds enough gas to travel n miles, and the map on his smartphone gives the distances between gas stations on the highway. Keith wishes to make as few gas stops as possible along the way.

Suppose there are k gas stations from A to B, denoted by $s_1, s_2, \ldots, s_k$ in order. For convenience, let s_0 and s_{k+1} be Cities A and B, respectively. Let d_i denote the distance between s_i and s_{i+1} for $i = 0, 1, \ldots, k$. Assume that $d_i \leq n$ for any $0 \leq i \leq k$; otherwise, it is impossible for Keith to complete the journey.

(a) Give an efficient method by which Keith can determine at which gas stations he should stop.

(b) Prove that your strategy in (a) yields an optimal solution.