COMP S265F Design and Analysis of Algorithms
Lab 5: Huffman Codes and the Master Theorem

In this lab, we implement the Huffman code algorithm for n characters using a min-heap (`heapq` module), which has a time complexity of $O(n \log n)$. Then, we introduce the Master Theorem for analyzing the time complexity of divide-and-conquer algorithms.

1. Huffman code algorithm

In Lab 4, we construct a Huffman tree using objects of `Node` class. When selecting the two nodes with the minimum frequencies, we search from a list of tuples `(freq, node)` containing the frequency `freq` of the node object `node`. We use another approach in this lab.

Totally ordered class. Python classes can support comparison by implementing a special method for each comparison operator. For example, to support the `>=` operator, you define a `__ge__(self, other)` method in the classes, where `self` is the current object and `other` is the object to compare with. It is tedious to create implementations of every possible comparison operator.

The `functools.total_ordering` decorator (a Python decorator is similar to a Java annotation, both start with `@`) can be used to simplify this process. We decorate a class with `@functools.total_ordering`, and define `__eq__()` and one other comparison method (`__lt__`, `__le__`, `__gt__`, or `__ge__`). The decorator then completes the other comparison methods automatically.

While this decorator makes it easy to create well behaved totally ordered types, it comes at the cost of slower execution and more complex stack traces for the derived comparison methods. If performance benchmarking indicates this is a bottleneck for a given application, implementing all six rich comparison methods instead (i.e., `__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, and `__ge__`) is likely to provide an easy speed boost.

We will make `Node` objects comparable using their frequencies `freq`.

```
1  from functools import total_ordering
2
3  @total_ordering
4  class Node:
5      def __init__(self, freq, ch, left, right):
6          global num_node
7          self.id = str(num_node)
8          num_node += 1
9          self.freq = freq
10         self.ch = ch # None for internal nodes
11         self.left = left
12         self.right = right
13
14     def __eq__(self, other):
15         return type(self) == type(other) and self.freq == other.freq
16
17     def __lt__(self, other):
18         return self.freq < other.freq
```

Note that in the `__eq__()`, we need to check whether we are comparing objects of the same type; otherwise, when you check the condition `node.left == None` in `traverse()`, the following error will occur:

```
AttributeError: 'NoneType'object has no attribute 'freq'
```

Your task. Complete the template `00huffman.py`, and test it with `python 00huffman.py < freq.txt`.

Code snippet of `00huffman.py`

```
# Task 1: Make Node totally ordered
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

class Huffman:
    def __init__(self, ch_freq):
        self.h = [] # a list of Node objects
        self.last = None # keep the last created Node
        self.tree = Graph()
        self.code = {}

        # Task 2: Create leaf nodes for self.h

        # Construct the Huffman tree
        while len(self.h) >= 2:
            # Task 3: Combine two min. freq. nodes and update self.h

        self.traverse(self.last, "")

    def traverse(self, node, c):
        if node.left == None:
            self.tree.node(node.id, label=node.ch+"/"+str(node.freq))
            self.code[node.ch] = c
        else:
            self.tree.node(node.id, label=str(node.freq))
            self.tree.edge(node.id, node.left.id, label="0")
            self.tree.edge(node.id, node.right.id, label="1")
            self.traverse(node.left, c+"0")
            self.traverse(node.right, c+"1")
```

2. Huffman code algorithm with `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.

Your task. Replace the list `self.h` by a min-heap of `Node` objects. This improve the time complexity of the Huffman code algorithm from $O(n^2)$ to $O(n \log n)$.

3. The Master Theorem

The Master Theorem is for analyzing the time complexity $T(n)$ of a divide-and-conquer algorithm on an input of size n, where $T(n)$ is in the form:

$$T(n) = a\, T\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$$

where $a \geq 1$ is the number of subproblems in the recursion, $b > 1$ is the factor by which the subproblem size is reduced in each recursive call, and $O(n^d)$ ($d \geq 0$) is the time complexity at the top level of the recurrence. Note that there is a more general result of the Master Theorem, but we will not be needing it.

$T(n)$ can be bounded asymptotically, as follows:

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

4. Exercises

Question 1. Find the time complexity of the following algorithms in Unit 3 using the Master Theorem.

(a) Finding the maximum of n numbers:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(1)$$

(b) MergeSort on n numbers:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

Question 2. In Unit 3, let $T(n)$ be the total number of comparisons the algorithm *max* made in the worst case to find the maximum of n numbers. Without simplifying n to a power of 2, we have

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1\ .$$

Show that $T(n) \leq cn - d$ for some constants c and d without using the Master Theorem.

Question 3 (Tower of Hanoi). The tower of Hanoi problem is defined as follows. There are n disks of different sizes arranged on a pole A in decreasing order of sizes. There are two other empty poles B and C. The purpose of the puzzle is to move all the disks, one at a time, from the pole A to another pole C in the following ways.

> Disks are moved from the top of one pole to the top of another. A disk can be moved to a pole only if it is smaller than all other disks on that pole. In other words, the ordering of disks by decreasing sizes must be preserved at all times. The goal is to move all the disks in as few moves as possible.

(a) Design an algorithm using divide and conquer to find a minimal sequence of moves that solves the tower of Hanoi problem for n disks.
Hint. To move n disks from pole A to pole C, you first move the top $n-1$ disks from pole A to pole B.

(b) How many moves are used in your algorithm?

(c) Argue that the solution given by your algorithm indeed uses the minimum number of moves.