Solution Set

CIS 121—Data Structures and Algorithms—Fall 2020

Heaps & Huffman—Monday, October 5/Tuesday, October, 6

Readings

- Lecture Notes Chapter 15: Huffman Coding
- Lecture Notes Chapter 16: Graph Representations & BFS

Problems

Problem 1. What are pros and cons of Huffman Coding?

Solution

Pros: Huffman codings represents only the characters that are contained in the text, without wasting space in the encoding length for characters that are not present. Additionally, characters that occur most take less bits to encode each time.
Cons: It does not help with longer patterns in the text. Storing the symbol table takes up additional space. Therefore it isn't appropriate for very small files where the table size would be significant. Encoding a text requires a preprocessing step to generate the table, does not work with a data stream.

Problem 2. Construct an optimal Huffman coding for the following alphabet and frequency table:

Character:	A	B	C	D	E
Frequency:	0.4	0.3	0.15	0.1	0.05

What is the average encoded character length for the above encoding?

Solution

The following tree would be produced:

```
 /\
A /\
 B /\
  C /\
   E  D
```

$$l_{avg} = \sum_{c \in R} d(c) * freq(c) = .4 * 1 + .3 * 2 + .15 * 3 + .1 * 4 + .05 * 4 = 2.05$$

Problem 3. You have an alphabet with $n > 2$ letters and frequencies. You perform Huffman encoding on this alphabet, and notice that the character with the largest frequency is encoded by a 0. In this alphabet, symbol i occurs with probability $p_i; p_1 \geq p_2 \geq p_3 \geq ... \geq p_n$.
Given this alphabet and encoding, is it true or false that there exists an assignment of probabilities to p_1 through p_n such that $p_1 < \frac{1}{3}$?

Solution

This claim is false. We can use a simple proof by contradiction. Assume that there exists an assignment such that $p_1 < \frac{1}{3}$. Look at the last step of the Huffman algorithm, that is the step when our two final nodes are merged into one node. Let these two final nodes be called x and y. Because character 1 has an encoding length of 1, it must have been included in this step. WLOG let x be our single character 1. This is the final step of Huffman, so we know that $p_x + p_y = 1$. Given our assumption that $p_1 < \frac{1}{3}$, this tells us that $p_y = 1 - p_x \to p_y > \frac{2}{3}$.

We know that because $n > 2$, y must be a group containing at least 2 characters. So, examine the time when y was created. Let the nodes which combined to y be called a and b. We know that $p_a + p_b > \frac{2}{3}$, which implies that $max\{p_a, p_b\} > \frac{1}{3}$. Here we have reached contradiction. We know Huffman uses the smallest two nodes at all steps, but at the step y was created node x was still available and unpaired. $p_x < \frac{1}{3}$, so it would have been chosen instead of $max\{a, b\}$. This contradiction then proves the original claim.

Problem 4.
Design an $O(n \log k)$ time algorithm to find the k-th highest test score, where n is the total number of scores in the stream. Since PEFS provides minimal monetary resources, the 121-CIS staff have limited access to storage space and and can only afford you $O(k)$ space for your feature, where $k \ll n$.

Solution

Algorithm: Take the first k tests, and construct a min-heap of size k, where tests are inserted into the min-heap and ordered by their score. For each test in the input, if the score is greater than or equal to the score at the root of the heap, remove the root, insert the test as the new root, and perform MIN-HEAPIFY. Else, if the score of the new test is less than or equal to the score of the root, do nothing. After processing all input, return the test score at the root of the heap.

Proof of correctness: We want to show that the algorithm, as described above, returns the score of the input that is the k-th highest. Consider any test that enters the heap. By construction, any test that enters the heap must have a score that is greater than or equal to some other score. Assume for the sake of contradiction that a test a_{bad} with a score greater than order k (i.e., lower than the k-th highest score) remains on the heap at the termination of the algorithm. But because we always maintain a heap that has a maximum size of k, this implies that some test a_{good} with a score less than or equal to k (higher score than the k-th highest test) is excluded from the heap. But by construction, it is impossible for a_{good} to have been excluded, since it would have compared higher than a_{bad}, which, in turn, would also be higher than the root element by transitivity! This is a contradiction, so our algorithm must be correct.

Running time analysis: Constructing a min-heap from the first k elements (unsorted) takes time $O(k)$. We maintain the heap at size k by removing the root in constant time and inserting a new score at the root and percolating downwards. Since each test score can be inserted in the heap as the root at most once, each test score is percolated downwards to its final position in time $O(\log k)$. Since the input has n scores, our overall running time is $O(k + n \log k)$. But since $n \gg k$, we have a final complexity of $O(n \log k)$.

Additional Problems

Problem 5. Your task is to connect n ropes with a minimum cost. You are given n ropes of different lengths, and you need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths. You need to connect the ropes with minimum cost.
For example if we are given 4 ropes of lengths 4, 3, 2 and 6. We can connect the ropes in following ways.

1. First connect ropes of lengths 4 and 3. Now we have three ropes of lengths 2, 7, 6.
2. Now connect ropes of lengths 2 and 7. Now we have two ropes of lengths 6 and 9.
3. Finally connect the two ropes and all ropes have connected.

Total cost for connecting all ropes is $7 + 9 + 15 = 29$. Give an algorithm which always finds the minimum cost of attaching the rope.

Solution

If we observe the above problem closely, we can notice that the lengths of the ropes which are picked first are included more than once in total cost. Therefore, the idea is to connect smallest two ropes first and recur for remaining ropes. This is a very similar principle as in Huffman encoding. We put smallest ropes down the tree so that they can be repeated multiple times rather than the longer ropes.

Following is complete algorithm for finding the minimum cost for connecting n ropes. The total runtime is the same as Huffman, so $O(nlg(n))$. Let there be n ropes of lengths stored in an array $len[0...n-1]$

1. Create a min heap and insert all lengths into the min heap.
2. Do following while number of elements in min heap is not one.
    (a) Extract the minimum and second minimum from min heap
    (b) Add the above two extracted values and insert the added value to the min-heap.
    (c) Maintain a variable for total cost and keep incrementing it by the sum of extracted values.
3. Return the value of this total cost

Problem 6. Consider an indefinitely long stream of unsorted integers. We are interested in knowing the median (in sorted order) at any given time. How would we do this in an efficient manner?

Solution

Algorithm: We can maintain a min-heap and a max-heap simultaneously. The max-heap contains the smaller half of numbers and the min-heap contains the larger half of numbers. Maintain the following two invariants:

1. The difference in size of the max-heap and the size of the min-heap is at most 1.
2. The root of the max-heap is always less than or equal to the root of the min-heap.

For the first two elements of the stream, put the smaller element into the max-heap, and the larger element into the min-heap. Whenever a new element of the stream is encountered, insert the element into the max-heap. If the element is smaller than the root of the max-heap, insert it into the max-heap, otherwise insert it into the min-heap. If invariant (1) is violated, remove the root from the larger heap and insert that newly-removed element into the smaller heap. To retrieve the median at any given time, if the number of total elements is odd, take the root of the max-heap; otherwise, take the average of the roots of both heaps.

Proof of correctness: The correctness of the computation of the median from the invariants is immediate. We want to show that our algorithm maintains these invariants. We leave justification of these facts to the reader.
Running time analysis: Let n be the number of elements seen in the stream. In this algorithm, we perform at most two insertions into heaps of size at most $\lceil n/2 \rceil$, which is a running time that is $O(\log n)$. We can access the roots of the heaps for the median computation in constant time. Hence, for every element of the stream, we maintain our data structures in $O(\log n)$ time. Since every element is stored internally, we use $O(n)$ space.