Foundations of Algorithms, Spring 2021: Homework 3

Due: Wednesday, March 10, 11:59pm

Problem 1 (10 points)

We are familiar with binary Huffman codes. Input symbols are converted into codewords comprising a sequence of 0's and 1's, or *bits*.

In a ternary Huffman code, codewords comprise a sequence of *trits*, each of which can have one of three values: 0, 1 or 2.

The Huffman coding algorithm to form ternary codewords is the same as for binary codewords except three symbols are combined into a new metasymbol at each step. (To make the combining work out, sometimes an additional 0-frequency placeholder symbol is used, but in the examples below that won't be necessary.)

1. Given the following input alphabet and corresponding frequencies (expressed as probabilities):

Symbol	**Frequency**
A	0.50
B	0.26
C	0.12
D	0.06
E	0.06

   (a) Build the corresponding binary Huffman code.

   (b) Compute the expected bits per symbol for this binary Huffman code. Remember that this is weighted by the probability of occurrence of each symbol.

   (c) Build the corresponding ternary Huffman code.

   (d) Compute the expected trits per symbol for this ternary Huffman code.

   (e) Convert the expected trits to expected bits per symbol by multiplying your previous answer by 1.585.

2. Repeat the same 5 steps from part 1) with the following input alphabet and corresponding frequencies (expressed as probabilities):

Symbol	**Frequency**
A	0.35
B	0.35
C	0.10
D	0.10
E	0.10

3. In information theory, the term *entropy* refers to the expected amount of information contained in one random symbol from a given probability distribution. The more "unknown" the value is, the higher the entropy.

For example, flipping a coin has an entropy of 1 bit of information. If the coin is weighted so that it almost always comes up heads, then flipping that coin has an entropy of less than 1 bit, because there is less uncertainty about the expected outcome. The entropy eventually drops all the way to zero if the outcome is known (heads always comes up).

Entropy also represents a bound on the efficiency possible when encoding symbols from a probability distribution.

Huffman coding yields an optimal prefix-code, but because it requires an integer number of bits (or trits) per codeword, it doesn't necessarily reach the entropy limit for efficiency.

For a given probability distribution, entropy in expected bits per symbol is computed as:

$$H(X) = -\sum_{x} p(x)log_2(p(x)).$$

In this equation, X is the random variable, $H(X)$ is the entropy of the random variable, x represents a symbol (outcome) from the distribution, and $p(x)$ represents the probability of occurrence of that symbol. The equation sums over all of the possible symbols in this distribution.

(a) Compute the entropy (in expected bits per symbol) for the probability distribution from part 1). All you need to do this is the probabilities from the table, plugged into the equation.

(b) Compute the entropy (in expected bits per symbol) for the probability distribution from part 2).

4. Draw conclusions. In which example did the binary Huffman code achieve an efficiency closer to entropy? In which example did the ternary Huffman code achieve an efficiency closer to entropy? What kind of probabilities does a binary Huffman code seem best suited to encode efficiently? What kind of probabilities does a ternary Huffman code seem best suited to encode efficiently?

Problem 2 (6 points)

Consider the following "minimize gaps" interval scheduling problem: Given is a global start time S and finish time F. Given also is a set of n intervals where the i-th interval starts at time s_i and ends at time f_i, $S \leq s_i \leq f_i \leq F$, for $i \in \{0, \ldots, n-1\}$. Find a set of non-overlapping intervals so that the overall time from S to F *not* covered by the selected intervals is as small as possible (we refer to this time as *gap time*).

Example: $S = 0$, $F = 10$, and the intervals are $(1,4), (2,7), (4,5), (6,10), (8,9)$. If we select intervals $(1,4), (4,5), (6,10)$, then the time between 0 and 10 that is not covered by the intervals is $(0,1)$ and $(5,6)$ – total time 2. This is the smallest possible overall gap time. Note that in the case when the end time of one interval equals the start time of a second interval, those two intervals are considered to be compatible (e.g. $(1,4)$ and $(4,5)$).

Consider the following greedy strategies for this problem:

1. Select the earliest finishing interval and discard overlapping intervals. Keep doing this until all intervals have been eliminated (either selected or discarded).

2. Select the earliest starting interval and discard overlapping intervals. Keep doing this until all intervals have been eliminated (either selected or discarded).

3. Select the pair of non-overlapping intervals that have the smallest gap between them: find a pair of intervals $i \neq j$ such that $s_j - f_i \geq 0$ is the smallest possible. Select both intervals and discard overlapping intervals. Recursively do the same selection process with intervals that finish at or before s_i: the recursive call will have $S_{\text{new}} = S$ and $F_{\text{new}} = s_i$. Similarly, recursively do the same selection process with intervals that start at or after f_j: now $S_{\text{new}} = f_j$ and $F_{\text{new}} = F$. If there is no such pair of intervals, select a single interval that minimizes the gap between S and F (do not make any further recursive calls).

None of these strategies works all the time. Find a counterexample for each strategy. In other words, for each strategy,

- find a set of intervals and S and F so that the strategy does not produce an optimal solution,
- highlight intervals selected by the strategy and state the corresponding gap time, and
- highlight intervals in an optimal solution (one that minimizes the overall gap time) and state the corresponding gap time.

Problem 3 (20 points: 14 for implementation / 6 for writeup)

Given are n different points in 2-dimensional space. Design an $O(n^2 \log n)$ algorithm that determines the number of right triangles that can formed by choosing sets of three points. Half credit can be received for an $O(n^3)$ algorithm.

Problem 4 (20 points: 14 for implementation / 6 for writeup)

You just got your allowance and the money is burning a hole right through your pocket. At the candy store, you decide to spend all you've got, and you want to try as many different types of candy as possible. Each type of candy in the store has an associated cost per piece. Given that the candy store has n different types of candy, design an $O(n)$ algorithm to determine the maximum number of different types of candy that you can buy with your allowance.

NOTE: linear sorting will not satisfy $O(n)$ for this problem. There is no guarantee that the necessary assumptions on the range and integrality of the input values will be satisfied for counting sort or radix sort. Nor is the data necessarily uniformly distributed, so bucket sort is not applicable.

Problem 5 (10 points: 6 for implementation / 4 for writeup)

It's time to tile the backsplash in your kitchen. It has dimension $2 \times n$, and you have an unlimited supply of 1×1 square tiles, and L-shaped tiles. (An L-shape is a 2×2 with a 1×1 corner removed.) Design an $O(n)$ algorithm to determine how many overall patterns there are with which you can tile your backsplash using these pieces. Each possible orientation of an L-shape is allowed, and represents a unique tiling. No pieces of tile may extend outside of the $2 \times n$ boundary.

So, for example, a 2×2 backsplash can be tiled with 5 different patterns: 1 pattern uses only 1×1 tiles; the other 4 patterns use an L-shaped tile in one of 4 possible orientations, along with a single 1×1 tile.

Note 1: Select Algorithm

If for any problem you utilize the select algorithm, you may use the random version with expected $O(n)$ performance and treat it as nonetheless guaranteeing $O(n)$ performance. (You're welcome to implement the median-of-medians approach if you'd like.)

Note 2: Heart of the Algorithm

For any problem in which you use dynamic programming (in this or any future homework!), to explain how your algorithm works and provide the correctness argument, describe the "heart of the algorithm" (you do not need to include any other explanation). Recall that the heart of the algorithm consists of three parts:

- Precise verbal description of the meaning of your dynamic programming array; see the slides for examples.
- A mathematical formula that describes how to compute the value of each cell of the array (using previously computed array values).
- Return value of the algorithm.

This will satisfy the "correctness" argument for your algorithm. You will still need to provide some additional analysis for the running time.