1. (30 pts) Implement Huffman's algorithm from scratch using a priority queue data structure. You may not use any library that implements a priority queue (or equivalent) data structure—the point here is to implement it yourself, using only basic language features. Within the priority queue, ties should be broken uniformly at random.

   Your implementation should take as input a string `x` of ASCII characters, perform the Huffman encoding, and then output the following: (i) the "codebook," which is a table containing each of the symbols in `x` and its corresponding binary Huffman encoding and (ii) the encoded string `y`.

   Submit your code implementing Huffman. Be sure to include code comments.

   Hint: Break your implementation into three parts, one that takes `x` and extracts its alphabet Σ and corresponding frequencies $\{f_i\}$; a second part that takes as input the set $\{f_i\}$ and constructs a Huffman code, and a third part that writes out the codebook and encoding `y`.

2. (20 pts total) The data file on the class website for PS5 contains the text of a famous poem by Robert Frost. The text comes in the form of a string σ containing $\ell = 761$ symbols drawn from an alphabet Σ with $|\Sigma| = 31$ symbols (24 alphabetical characters, 6 punctuation marks and 1 space character).

   (a) (5 pts) A plaintext ASCII character normally takes 8 bits of space. How many bits does σ require when encoded in ASCII?

   (b) (5 pts) The theoretical lower limit for encoding is given by the *entropy* of the frequency of the symbols in the string:

   $$H = -\sum_{i=1}^{|\Sigma|} \left(\frac{f_i}{\ell}\right) \log_2 \left(\frac{f_i}{\ell}\right) \quad ,$$

   where f_i is the frequency of the ith symbol of the alphabet Σ. Because we take $\log_2$, H has units of "bits." What is the theoretical lower limit for the number of bits required to encode σ?

   (c) (5 pts) Encode σ using your Huffman encoder and report the number of bits in the encoded string. Comment on this number relative to your theoretical calculation from part (b).

   (d) (5 pts) How many additional bits would be required to write down the codebook? Could this number be made any smaller?

3. (25 pts) Using your Huffman implementation from question 1, conduct the following numerical experiment. (This will be easier if you followed the hint in question 1.)

   Show via a plot on log-log axes that the asymptotic running time of your Huffman encoder is $O(n \log n)$, where $n = |\Sigma|$ is the size of the input alphabet Σ. The deliverable here is a single figure (like the one below) showing how the number of atomic operations T grows as a function of n. Include two trend lines of the form $c \times n \log n$ that bound your results from above and below. Label your axes and trend line. Include a clear and concise description (1-2 paragraphs) of exactly how you ran your experiment.

   No credit will be given if you don't label your axes and trend lines, or if your plot is not on log-log axes.

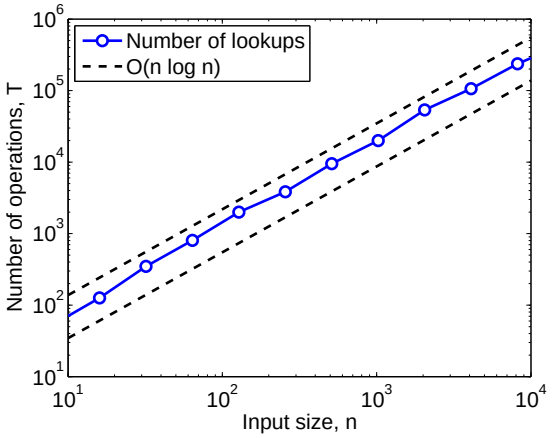

Figure 1: An example of what your Huffman results should look like.

Hint 1: You will need to implement some measurement code within your Huffman encoder that counts the number of atomic operations it performs. You do not need to count all the atomic operations, only the ones whose numbers depend on the alphabet size n. First, identify which operation in your Huffman implementation has this property and then add a counter that increments each time that operation occurs. Modify your implementation so that when Huffman is done, it also returns this count.

Hint 2: In order to plot the function $T(n)$, you will need to pass your encoder a set of frequencies $f_1, f_2, \ldots, f_n$, where n is the number of symbols in the input alphabet. Increasing the number of frequencies in the input is the same as increasing the alphabet

size. The particular values of the frequencies do not matter for this task, so choose them any way you like, e.g., via a pseudorandom number generator.

Hint 3: To get a good range of behavior across n, choose a dozen or so values of n, spaced logarithmically between 10^1 and 10^5, e.g., $n = \{2^3, 2^4, 2^5, \ldots, 2^{13}\}$.

Hint 4: If you find that your function $T(n)$ is not particularly smooth, you can make it look nicer by averaging the observed count $T(n)$ across several independent runs of the algorithm on different inputs of size n. The more independent runs, the smoother the line should become.

4. (25 pts total) You and Gollum are having a competition to see who can compute the nth Fibonacci number more quickly. Gollum asserts the classic recursive algorithm:

```
Fib(n) {
   if n < 2
      return n
   else
      return Fib(n-1) + Fib(n-2)
   end
}
```

which he claims takes $R(n) = R(n-1) + R(n-2) + c = O(\phi^n)$ time and space.

You counter with a dynamic programming algorithm, which you claim is asymptotically faster. Recall that dynamic programming is a kind of recursive strategy in which instead of simply dividing a problem of size n into two smaller problems whose solutions can be merged, we instead construct a solution of size n by merging smaller solutions, starting with the base cases. The difference is that in this "bottom up" approach, we can reuse solutions to smaller problems without having to recompute them.

   (a) (10 pts total) You suggest that by "memoizing" (a.k.a. memorizing) the intermediate Fibonacci numbers, by storing them in an array `F[n]`, larger Fibonacci numbers can be computed more quickly. You assert the following algorithm.[1]

[1] Gollum briefly wails about your `F[n]=undefined` trick ("an unallocated array!"), but you point out that `MemFib(n)` can simply be wrapped within a second function that first allocates an array of size n, initializes each entry to `undefined`, and then calls `MemFib(n)` as given.

```
MemFib(n) {
   if n < 2
      return n
   else
      if (F[n] == undefined)
         F[n] = MemFib(n-1) + MemFib(n-2)
      end
      return F[n]
   end
}
```

i. (5 pts) Describe the behavior of `MemFib(n)` in terms of a traversal of a computation tree. Describe how the array `F` is filled.

ii. (5 pts) Compute the asymptotic running time. Prove this result by induction.

(b) (5 pts) Golum then claims that he can beat your algorithm in both time and space by eliminating the recursion completely and building up directly to the final solution by filling the F array in order. Gollum's new algorithm[2] is

```
DynFib(n) {
   F[0] = 0
   F[1] = 1
   for i = 2 to n
      F[i] = F[i-1] + F[i-2]
   end
   return F[n]
}
```

How much time and space does `DynFib(n)` take? Justify your claim and compare your answers to those of your solution in part (a).

(c) (5 pts) With a gleam in your eye, you tell Gollum that you can do it even more efficiently because you do not, in fact, need to store all the intermediate results. Over Gollum's pathetic cries, you say

[2]Note that Gollum is now using your undefined array trick; assume he also uses your solution of wrapping this function within another that correctly allocates the array.

```
FasterFib(n) {
   a = 0
   b = 1
   for i = 2 to n
      c = a + b
      a = b
      b = c
   end
   return b
}
```

Derive the time and space requirements for `FasterFib(n)`. Justify your claims.

(d) (5 pts) Give a table that lists each of the four algorithms, its asymptotic time and space requirements, and the implied or explicit data structures each requires. Briefly compare and contrast the algorithms in these terms.

5. (40 pts extra credit total) Gollum hands you a set X of $n > 0$ intervals on the real line and demands that you find a subset of these intervals $Y \subseteq X$, called a *tiling cover*, where the intervals in Y cover the intervals in X, that is, every real value contained within some interval in X is contained in some interval Y. The *size* of a tiling cover is just the number of intervals. To satisfy Gollum, you must return the minimum cover Y_{min}: the tiling cover with the smallest possible size.

   For the following, assume that Gollum gives you an input consisting of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X.

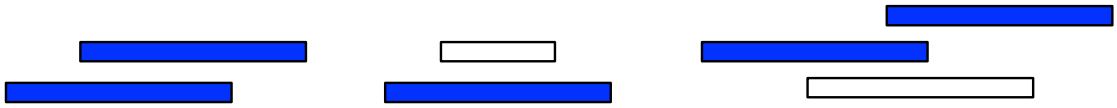

Figure 2: A set of intervals X and a tiling cover Y shown in blue.

   (a) (20 pts extra credit) Describe in words and give pseudo-code for a *greedy* algorithm to compute the smallest Y in $O(n \log n)$ time.
   Hint: Starting with the left-most tile.

   (b) (20 pts extra credit) Prove that this algorithm (i) covers each component and (ii) produces the minimal cover for each component. Explain why these conditions are necessary and sufficient for the correctness of the algorithm.