Problem Set Five

Zoe Farmer
Jeremy Granger
Ryan Roden

February 26, 2024

1. 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.

2. 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 $l = 761$ symbols drawn from an alphabet Σ with $|\Sigma| = 31$ symbols (24 alphabetical characters, 6 punctuation marks and 1 space character).

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

      i. This is simply $8 \cdot l = 6088$.

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

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

   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?

      i. This can be restated with variables filled, and subsequently solved with a python one-liner.

   $$H = -\sum_{i=1}^{31} \left(\frac{f_i}{761}\right) \log_2 \left(\frac{f_i}{761}\right) = 4.12543135174$$

```
Hsum = -1 * sum([((alpha[f] / length) *
                  numpy.log2(alpha[f] / length))
                  for f in alpha])
```

(c) 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).

i. Using our encoder this is 2323 bits.

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

i. Assuming 8 bits per character and 4 bits per integer value, we'd need 384 bits with the following code.

$$\{'r':2,'a':3, \quad ',':30,';':172, \quad ':':173,'y':28, \quad 'n':12,'w':30,$$
$$'v':21,'u':29, \quad 't':4,'s':3, \quad '-':253,'q':255, \quad 'p':62,'o':6,$$
$$'.':87,'m':42, \quad 'l':11,'k':20, \quad 'j':254,'i':4, \quad 'h':14,'g':20,$$
$$'f':62,'e':0, \quad 'd':13,'c':63, \quad 'b':11,'!':252, \quad ' \ ':2\}$$

3. Using your Huffman implementation from question 1, conduct the following numerical experiment.

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.

(a) Once we have a workable Huffman Encoder, we can now run trials to determine its runtime complexity. We add an `operations` class variable to our encoder, and then account for every elementary operation that takes place. Using Python3's extended unicode support we can generate large frequency dictionaries quickly that still have meaning. The below code when given the desired length of our frequency dictionary will create a randomized dictionary of unicode characters and their "frequencies".

```
def generate_alpha(length):
    ''' Generates random frequency dict '''
    alpha = {}
    for i in range(length):
        alpha[chr(32 + i)] = random.randint(0, 100)
    return alpha
```

In order to successfully complete the trials we then wrap this in another `for`-loop which checks runtime complexity at logarithmic intervals.

```
1   n  = []  # Lengths
2   op = []  # Operation count
3   for length in [2**x for x in range(1, 14)]:
4       huffman = Huffman()
5       # Our alphabet generator from before
6       alpha = generate_alpha(length)
7       # Create the encoder with our alphabet
8       huffman.create_code(alpha)
9       # X-value: number of elements in alphabet
10      n.append(length)
11      # Y-value: number of operations required
12      op.append(huffman.operations)
```

We then use `matplotlib.pyplot` to create a graph of our results. Please see figure 1. Note, it is highly recommended to examine the source code written. Any questions you have about the code itself will most likely be answered by either the code or the comments.

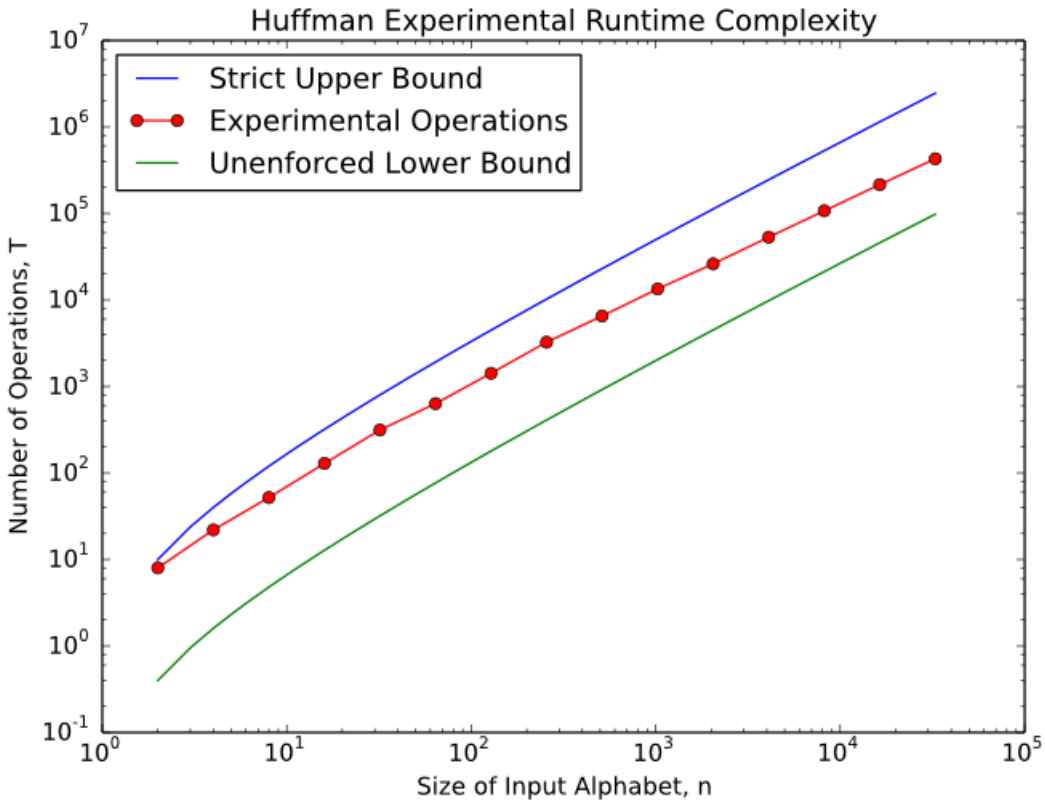

Figure 1: Runtime Complexity of Huffman Encoding

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

```
1  def Fib(n):
2      if n < 2:
3          return n
4      else:
5          return Fib(n-1) + Fib(n-2)
```

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) 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  def MemFib(n):
2      if n < 2:
3          return n
4      else:
5          if F[n] is None:
6              F[n] = MemFib(n-1) + MemFib(n-2)
7          return F[n]
```

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

A. MemFib will always see MemFib$(n-1)$ as the element that needs to be determined until $n-1 < 2$, at which point it will merely return n. This means that when MemFib$(n-2)$ is executed, the algorithm will just pull the value from the filled array, and not traverse that branch. This essentially means that *only* the left branches of the recursion tree are executed until we reach the base case. This also means that F is filled from index 2 to index n. To see the computation tree, please reference Figure 2, where only the left branches lead to recursion and all right branches are array indexing operations.

ii. Compute the asymptotic running time. Prove this result by induction.

A. The asymptotic running time of this algorithm is $O(n)$ due to its perfectly unbalanced recursion tree.

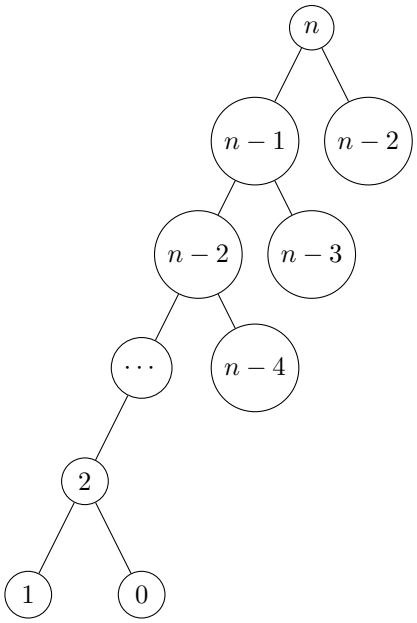

Figure 2: Computation Tree

$$\begin{aligned}
\text{Recurrence Relation of MemFib} \rightarrow \quad & M(n) = M(n) + M(1) \\
\text{Index Cost} \rightarrow \quad & M(1) = O(1) \\
\text{Assume} \rightarrow \quad & M(k) = k \\
\text{Therefore} \rightarrow \quad & M(k) \Rightarrow M(k+1) \\
\text{Substituting} \rightarrow \quad & M(k+1) \Rightarrow M(k+1) \\
\text{Therefore} \rightarrow \quad & M(n) = O(n) \blacksquare
\end{aligned}$$

(b) Gollum 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 is

```
1  def DynFib(n):
2      F[0] = 0
3      F[1] = 1
4      for i in range(2, n):
5          F[i] = F[i-1] + F[i-2]
6      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).

i. Using Gollum's new algorithm we see that it is very similar to the previous one. In fact, Gollum's algorithm is no more time efficient than the original, and they both run in $O(n)$ time. They also have the same space requirements, however the difference between the two is the fact that MemFib makes

recursive calls, which add to the memory requirements in the stack, while DynFib avoids this through application of a `for`-loop. Thus we can establish that the space complexity of MemFib is $O(2n)$ and DynFib is $O(n)$, which when measured asymptotically we see that both are $O(n)$.

(c) 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

```
1  def FasterFib(n):
2      a = 0
3      b = 1
4      for i in range(2, n):
5          c = a + b
6          a = b
7          b = c
8      return b
```

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

i. Again, FasterFib runs no faster than the other two, it just uses far less space. It still has to iterate through n values, but it only keeps the values that it needs for the next calculation. This means that it's runtime complexity is $O(n)$, but the space complexity drops to $O(1)$.

(d) 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.

i. Please reference Table 1

Algorithm	Time Complexity	Space Complexity	Data Structures
Fib	$O(\Phi^n)$	$O(\Phi^n)$	Binary Recursion Tree
MemFib	$O(n)$	$O(n)$	Binary Recursion Tree/Array
DynFib	$O(n)$	$O(n)$	Array
FasterFib	$O(n)$	$O(1)$	Variables

Table 1: Algorithm's Complexity

5. Gollum hands you a set X of $n > 0$ intervals on the real line and demands that you find a subset of these intervals $Y \subset X$, 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 $X_L[1, \ldots, n]$ and $X_R[1 \ldots n]$, representing the left and right endpoints of the intervals in X.

   (a) Describe in words and give pseudo-code for a greedy algorithm to compute the smallest Y in $O(n \log n)$ time.

i. This algorithm first formats the intervals into a sorted array of tuples that have the same start point. Once this has been completed we iterate through the array in sorted order, taking the longest interval gain in each and adding it to our cover. This establishes a cover that takes only the largest gain and uses it to extend the gain further, as it then ignores any new interval with starting value less than the ending value of our previous.
This is $O(n \log(n))$ as it iterates through the array only once, yielding $O(n)$, and at each step it performs $O(\log(n))$ operations, yielding $O(n \log(n))$.

```
1  def main():
2      xl = [0, 1, 2, 3, 4, 1, 5]
3      xr = [1, 4, 3, 8, 5, 3, 13]
4
5      zipped = list(zip(xl, xr))
6      raw_sets = sorted(zipped, key=lambda pair: pair[0])
7
8      sets = []
9      for item in raw_sets:
10         start = item[0]
11         group = [item]
12         for n in raw_sets:
13             if n[0] == start and n != item:
14                 group.append(n)
15                 raw_sets.pop(raw_sets.index(n))
16         sets.append(group)
17
18     print(sets)
19
20     cover = []
21     previous = sets[0][0]
22     for item in sets:
23         if item[0][0] < previous[1]:
24             pass
25         longest = None
26         for pair in item:
27             length = pair[1] - pair[0]
28             if previous:
29                 length = pair[1] - previous[1]
30             if longest is None:
31                 longest = pair
32             elif length > (longest[1] - longest[0]):
33                 longest = pair
34         cover.append((longest[0], longest[1]))
35         previous = (longest[0], longest[1])
36     print(cover)
```

(b) 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.

i. This algorithm must cover each component in the interval as it traverses the entire array once, and since it takes only the greedy choice, it also produces the minimal cover.