Problem Set Four

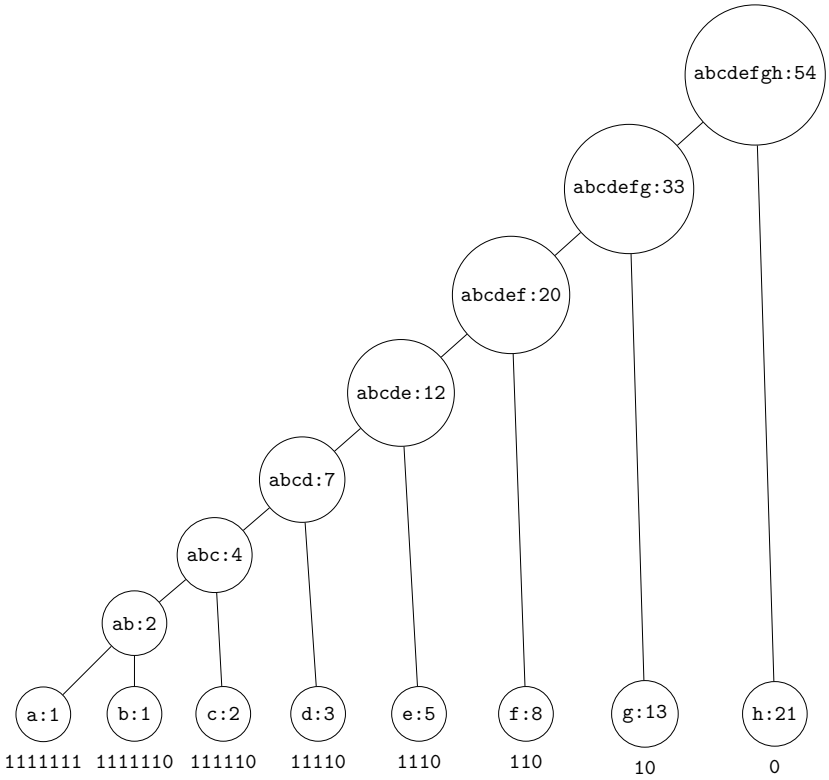
Zoe Farmer
Jeremy Granger
Ryan Roden

February 26, 2024

1. Recall that Huffman codes are constructed in a greedy fashion.
   (a) What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

   [a:1, b:1, c:2, d:3, e:5, f:8, g:13, h:21]

   i.



   (b) How many optimal Huffman codes are there for this set of frequencies? Justify your answer.

i. There are three different cases to produce a unique Huffman code. One, if the symbols `a` & `b` are switched in the tree. Two, if the parent symbol `ab` and symbol `c` are switched in the tree. Or three, if the symmetry is flipped, i.e. the tree goes up and left instead of up and right, switching the positions of all 0's with 1's and vice versa.

Each case creates 2 different outcomes and none are dependent on any others, so the Huffman code outcomes are represented by:

$$
\begin{aligned}
1, 2, 3 \\
1, 2, \neg 3 \\
1, \neg 2, 3 \\
1, \neg 2, \neg 3 \\
\neg 1, 2, 3 \\
\neg 1, 2, \neg 3 \\
\neg 1, \neg 2, 3 \\
\neg 1, \neg 2, \neg 3
\end{aligned}
$$

Therefore this set of symbols and frequencies has 8 different but equally optimal Huffman Codes

(c) Generalize your answer to find an optimal code when the frequencies are the first n Fibonacci numbers.

i. We can generalize our answer as is shown in Table 1

Symbol	*Code*
a	$1^{n-2} \cdot 0$
b	$1^{n-2} \cdot 1$ or 1^{n-1}
c	$1^{n-3} \cdot 0$
$\vdots$	$\vdots$
nth	$1^{n-n} \cdot 0$ or 0

Table 1: Generalize Fibonacci Huffman Codes

2. Professor Hagrid is struggling with the problem of making change for n cents using the smallest number of coins. Let the coin values be $v_1 > v_2 > \cdots > v_r$ for r coins types, and let each coin's value v_i be a positive integer. The output will be a set of counts $\{d_i\}$, one for each coin type, such that $\sum_{i=1}^{n} d_i = n$ and where k is minimized. (Note: v_1 is the most valuable coin.)

(a) Give a greedy algorithm, that takes $O(n)$ time, to make change consisting of quarters (worth 25 cents), dimes (10 cents), nickels (5 cents) and pennies (1 cent). Prove that your algorithm yields an optimal solution.

i. The strategy for this problem is defined as the following. At each step, choose the coin of largest denomination possible without exceeding the total.

```python
# Input: array with coin values (vals)
#        change value (n)
# Output: Array of coin counts
def change(vals=[c1, c2, ..., cn], n):
    c = [0 for item in vals]   # Number of coins we have
    for i in range(0, len(c)): # Check each denomination
        if n - vals[i] > 0: # If we can subtract
            c[i] += 1  # Add to count
            return [x + y for x,y in zip(c, # Recurse
                    change(n - vals[i], vals))] # combine
        elif n - vals[i] == 0: # Base case when we get to 0
            c[i] += 1 # Add to denomination count
            return c # return count up the tree
```

This algorithm runs in $O(n)$ time. The `for`-loop runs in constant time, which is the number of coins we have, `len(c)`. The function is then called as many times as it takes until n goes to zero, which by nature is dependent on n. Therefore the total runtime can be found with the recurrence relation

$$T(n) = T(n - vals[i]) + O(1) \Rightarrow O(n)$$

The reason this algorithm provides the optimal solution is demonstrated as follows. For any coin denomination, there is at least one way to equivalently equal that single coin with several smaller coins, but *no* ways to equate a smaller coin with a larger one. To demonstrate,

$$25¢ = \underbrace{10¢ + 10¢ + 5¢}_{3 \text{ Coins}}$$
$$10¢ = \underbrace{5¢ + 5¢}_{2 \text{ Coins}}$$
$$5¢ = \underbrace{1¢ + 1¢ + 1¢ + 1¢ + 1¢}_{5 \text{ Coins}}$$

Given the above statement, we can state several facts about our algorithm. One, any solution will have less than 3 dimes. One quarter is better than 3 dimes, therefore we will always have 2 or less. Two, any solution will have less than 2 nickels. One dime is better than 2 nickels, therefore the most nickels we can have is 1. Three, any solution will have less than 5 pennies. One nickel is better than 5 pennies, therefore the most pennies we can have is 4. The facts hold true about any optimal solution.

(b) Suppose that the available coins are in the denominations that are powers of c, i.e., denominations of $c^0, c^1, \ldots, c^l$ for some integers $c > 1$ and $l \geq 1$. Prove that the greedy algorithm always yields an optimal solution in this case.

i. To start, we know that every value of c has c^0 (the "penny") so every positive integer n has a solution with the greedy algorithm.

Similarly to above every denomination can be created with some combination of lower denominations. An illustrative example is when $c = 2$ and we'll let

$l = 3$.

$$c^3 = 8¢ = \underbrace{4¢ + 4¢}_{\text{2 Coins}}$$
$$c^2 = 4¢ = \underbrace{2¢ + 2¢}_{\text{2 Coins}}$$
$$c^1 = 2¢ = \underbrace{1¢ + 1¢}_{\text{2 Coins}}$$

Using similar logic we can state several facts about our algorithm. Any solution will have at most one 4¢ coin, and at most one 2¢ coin, because any two 4¢ coins could be replaced with one 8¢ coin, and two 2¢ coins could be replaced with one 4¢ coin. the optimal solution in this case.

Using a more extensible approach, we refer to our algorithm above. If we take the largest i^{th} denomination, such that $n - c^i \geq 0$ we now know that every coin c^i where $i > 0$ (our base case) can be comprised of no fewer than c coins of value $c^{(i-1)}$. Therefore one c^i coin will always yield a smaller coin count than c of $c^(i-1)$ for any value of n.

(c) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution, and explain why. The set should include a penny so that there is a solution for every value of n.

i. One such situation is when we simply exclude the nickel. For this set of coin denominations, we can look at when $n = 30$, and our algorithm will yield

$$30¢ = \quad 25¢ + 1¢ + 1¢ + 1¢ + 1¢ + 1¢ \Rightarrow \text{ Algorithm Solution}$$
$$30¢ = \quad 10¢ + 10¢ + 10¢ \Rightarrow \text{ Optimal Solution}$$

In this case our algorithm cannot provide the optimal solution, and is stuck with an unoptimal solution. This occurs because the algorithm is stuck choosing what it believes to be the "best" option at each step of the process. In this case it starts with a quarter, and then the *only* remaining denomination that can fit (since we've removed the nickel) is the penny, of which we need 5. Now if we had removed the quarter as well our algorithm would see that the dime would fit at the first, second, and third steps, yielding three dimes optimally instead of our unoptimal solution with a quarter.

To be fair, our algorithm is still pretty good. This case does not fail every scenario, and still works for many, including the case where $n = 36$. This yields one quarter, one dime, and one penny; which is in other words the optimal solution.

3. Let A and B be arrays of integers. Each array contains n elements, and each array is in sorted order (ascending). A and B do not share any elements in common. Give a $O(\lg(n))$-time algorithm which finds the median of $A \cup B$ and prove that it is correct. This algorithm will thus find the median of the $2n$ elements that would result from putting A and B together into one array.

(a) Let's start out by supposing that the (lower) median is in X. Let's call this median value m, and let's suppose that it is in $X[k]$. Then k elements of X are

less than or equal to m and $n-k$ elements of X are greater than or equal to m. We know that in the two arrays combined there must be n elements less than or equal to m and n elements greater than or equal to m. So there must be $n-k$ elements of Y that are less than or equal to m and $n-(n-k)=k$ elements of Y that are greater than or equal to n.

Thus, we can check that $X[k]$ is the lower median by checking $Y[n-k] \leq X[k] \leq Y[n-k+1]$. A boundary case occurs for $k=n$. Then $n-k=0$ and there is no array entry $Y[0]$, so we only need to check $X[n] \leq Y[1]$. Now if the median is in X, but is not in $X[k]$, then the above condition will not hold. If the median is in $X[k\prime]$ where $k\prime < k$, then $X[k]$ is above the median and $Y[n-k+1] < X[k]$. Conversely, if the median is in $X[k\prime\prime]$ where $k\prime\prime > k$ then $X[k]$ is below the median and $X[k] < Y[n-k]$.

Thus we can use a binary search tree to determine whether there is an $X[k]$ such that either $k < n$ and $Y[n-k] \leq X[k] \leq Y[n-k+1]$ or $k=n$ and $X[k] \leq Y[n-k+1]$. If we find such an $X[k]$, then it is the median. Otherwise we know that the median is in Y, and we use a binary search to find $Y[k]$ such that either $k < n$ and $X[n-k] \leq Y[k] \leq X[n-k+1]$ or $k=n$ and $Y[k] \leq X[n-k+1]$. Such $Y[k]$ is the median.

```
1  n = len(X) # |X| = |Y|
2  def findk(X, Y, k):
3      if k == n:  # If we've reached the end of the set
4          if X[n] <= Y[1]: # If the end of X is less than the first of Y
5              return k  # We've found our median
6      else:
7          if ((Y[n - k] <= X[k]) and  # If we've correctly identified k
8              (X[k] <= Y[n - k + 1])):
9              return k
10         else:
11             X1 = X[1:k - 1]  # Endpoints inclusive
12             X2 = X[k + 1: n] # Endpoints inclusive
13             if X1 != [] and X2 != []: # If X_{1|2} ≠ ∅, search both
14                 return findk(X1, Y, n / 4) or findk(X2, Y, (3 * n) / 4)
15             elif X1 != []: # Otherwise search the non-empty sets
16                 return findk(X1, Y, n / 4)
17             elif X2 != []:
18                 return findk(X2, Y, (3 * n) / 4)
19             else: # X1 = X2 = ∅
20                 return None   # Otherwise if both are empty we failed.
21 flag = False         # Did we guess correctly?
22 k = n / 2            # Initial Guess
23 k = findk(X, Y, k)  # Attempt to find k, the index of the lower median
24 if k is None:
25     flag = True      # Picked the wrong starting set
26     k = findk(Y, X, n/2)  # Use the other one
27 if flag:  # If we resorted to Y
28     if k == n: # if lower median is at upper bound
29         median = (Y[k] + X[1]) / 2 # take k and X[1]
30     else: # Otherwise k and k + 1
31         median = (Y[k] + Y[k + 1]) / 2
32 else:  # If X worked
33     if k == n: # As above, if at end take k and start of other
34         median = (X[k] + Y[1]) / 2
35     else: # Otherwise take k and k+1
36         median = (X[k] + X[k + 1]) / 2
```