Datastructures and Algorithms: Exercise 2

David Sermoneta

April 18, 2024

Exercise 1:

```
    list = [0]*(log2(N) + 1)
    counter = 0
def convert2binary(N,counter,list):
    if N // 2 == 0:
      list[counter] = N % 2
      return list
    else:
      list[counter] = N % 2
      counter += 1
      N = N // 2
      return convert2binary(N, counter, list)
```

A bit of explaining: the length of our list needs to be $\log_2(N) + 1$ long. We get there by examining that if the binary representation of N is

$$N = 2^k + a_1 2^{k-1} + \ldots + a_{k-1} 2^1 + a_k, \text{ where } a_i = 0 \text{ or } 1,$$

then we can claim that all the terms after 2^k sum up to less than 2^k by using geometric sums to bound our expression:

$$N = 2^k + m \leqslant 2^k + \sum_{i=0}^{k-1} 2^i = 2^k + 2^k - 1$$
$$\implies m \leqslant 2^k - 1 \leqslant 2^k.$$

That is, $N \leqslant 2^{k+1}$, which finally gives us that

$$\log_2(N) \leqslant k + 1,$$

which is the length size that our array will need to store each instance of $a_i 2^{k-i}$.
Now what remains is the explanation of the actual algorithm. This is just straight forward recursion blended with some euclidean algorithm. The base case covers the case when the euclidean algorithm has terminated (that is, when `N // 2 = 0` and the `else:` part of the code adds N mod 2 (which obviously is either 1 or 0), and then divides N by 2 to run the function again, until we get to the base case in which case we are done. The counter is just there to make sure we are accessing the right part of memory.

Exercise 2:

We will use the standard rules $\log(a \cdot b) = \log(a) + \log(b)$, $\log(n^b) = b\log(n)$ and $\log_b(n) = \frac{\log_c(n)}{\log_c(b)}$.

Proof of (a.) We note that

$$\log_b(n!) \leqslant \log_b(n^n) = n\log_b(n) = O(n\log_b(n)) \text{ for all } n \geqslant 0,$$

and so we are done. □

Proof of (b.) It suffices to show that $a^{\log_b(n)} = n^{\log_b(a)}$. We have that

$$\mathsf{LS} = a^{\log_b(n)} = a^{\frac{\log_a(n)}{\log_a(b)}} = n^{\frac{1}{\log_a(b)}}$$

since $\frac{1}{\log_a(b)} = \frac{1}{\frac{\log_b(b)}{\log_b(a)}} = \log_b(a)$, we have that

$$\mathsf{LS} = n^{\log_b(a)} = \mathsf{RS}.$$

□

Proof of (c.) Suppose $4^n \in O(2^n)$, then for constants n_0, c, we'd have that

$$\begin{aligned} & 4^n \leqslant c2^n \text{ for all } n \geqslant n_0 \\ \iff & 2^n 2^n \leqslant c2^n \\ \iff & 2^n \leqslant c, \end{aligned}$$

which is clearly a contradiction, since 2^n is unbounded above. Therefore, we have disproven that $4^n \in O(2^n)$, and we are done. □

Exercise 3:

Solution to 3: The arrangement looks like

$$f_5, f_6, f_7, f_1, f_4, f_2, f_3$$

Further, the complexity class of each function is

$$\begin{aligned} f_1 \in & O(n^{\frac{3}{2}}), \\ f_2 \in & O(n^2), \\ f_3 \in & O(2^{\frac{n}{4}}), \\ f_4 \in & O(n^3), \\ f_5 \in & O(1), \\ f_6 \in & O(n^{\frac{1}{3}}, \\ f_7 \in & O(n\log(n)) \end{aligned}$$

□

Exercise 4:

Proof of (a.) Since $\mathsf{T}(n)$ is of the form $a\mathsf{T}\left(\frac{n}{b}\right) + \Theta(n^d)$, where $a = 27$, $b = 3$ and $d = 2$. Since $27 \geqslant 3^2$, we get via the Master theorem that

$$\mathsf{T}(n) = \Theta(n^{\log_3(27)} = \Theta(n^3),$$

and we are done. □

Proof of (b.) Same procedure, set $a = 32$, $b = 2$ and $d = 5$, and we get that

$$\mathsf{T}(n) = a\mathsf{T}\left(\frac{n}{b}\right) + \Theta(n^d).$$

Since $32 = 2^5$, Master theorem allows us to conclude that

$$\mathsf{T}(n) = \Theta(n^5 \log_2(n))$$

□

Proof of (c.) We prove by induction that $\mathsf{T}(n) = n!\Theta(1)$. Base case is true, since $\mathsf{T}(1) = \Theta(1) = 1!\Theta(1)$. Now suppose it were true for some integer $k \geqslant 2$, then

$$\mathsf{T}(k+1) = (k+1)\mathsf{T}(k) = k + 1 \cdot k!\Theta(1) \text{ by our induction hypothesis,}$$

and so $\mathsf{T}(k+1) = (k+1)!\Theta(1)$, and we are done, the induction step is proven and we have that $\mathsf{T}(n) = \Theta(n!)$. □

Exercise 5:

Proof of (a). For the program to terminate, we need that either Q or S become empty. If S is empty at the time in which the program terminates, then that means that each element in S can also be found in Q, since before removing an element of S in a given iteration of the `WHILE` loop, we need to check that the same element is in Q. And so, in the case that S is empty at the time the program ends, necessarily $S \subseteq Q$.

If Q is empty when the program terminates, then that means that the queue is emptied after a finite amount of iterations of the `WHILE` loop. If originally, $|Q| = n$, then that implies that out of all the iterations of the `WHILE` loop, n times of those iterations the `IF` check will be satisfied, that is, the n elements of Q will also be found in S. Further, since S remains unchanged, after every iteration, this gives us that the elements of Q are found in S, as the n top-most elements, in some random order. That is, S's final n elements consistute a permutation of Q. So if $Q = [q_1, q_2, q_3, \ldots, q_n]$, then

$$S = [s_1, s_2, \ldots, s_m, q_1', q_2', \ldots, q_n'], \text{ where } q_i' \in Q.$$

To prove that these conditions are sufficient by themselves, we consider the two cases by themselves: Suppose that $S \subseteq Q$, at time $t = 0$. Since we are never adding any new elements to S or Q, at any given time $t = t_i$, $S \subseteq Q$ and $|S| = s_i \leqslant r_i = |Q|$. Then we can claim that after at most r_i iterations of the `WHILE` loop, a match between S and Q must be found, since after r_i iterations Q at time t_i and Q at time $t_i + r_i$ will be the same, meaning no elements of Q have been found in S contradicting our assumption. Therefore, in the case that $S \subseteq Q$, S will be emptied of one element after at most $|Q|$ iterations, leaving us with S being empty after at most

$$\sum_{k=0}^{|S|-1} |Q| - k \text{ iterations.} \tag{1}$$

And that at a given iteration of the `WHILE` loop at time $t = t_0$,

For the other case, suppose $Q \subset S$, as well as them being at the top of S as mentioned above. Then each iteration of the `WHILE` loop will leave the top of S unchanged, and either change the position of the front of Q, or find a match, and thus decrease the queue by one element. Given the assumption, and with the same reasoning as before, Q will eventually be emptied, since if we don't find a match between the front of Q and top of S in $|Q|$ iterations, then we've gone through all elements of Q without finding a match, contradicting our assumptioin. □

Solution of (b.i). If $S =$`[Theo]` then we'll only need one iteration of the `WHILE` loop, since S will be emptied after the first. □

Solution of (b.ii). If $S =$`[Theo, Max, Paula, Annemarie, Otto]`, it will require the most amount of iterations, in fact, it will require 15 iterations, since before removing each element from Q, we need to run the while loop `length.`(Q) times (the length of Q before the removal of the element). □

Solution of c. Suppose $S \subseteq Q$, then as we have shown, the program terminates when S ends up empty, which will happen when the `IF` statement runs $|S|$ times. To make this the worst case scenario, we want the `WHILE` loop to iterate as many times as possible before each eventual removal. As we have shown earlier, the maximal amount of times we can iterate the loop before eventually removing an element is $|Q| - k$ where k is the amount of elements we have removed from S so far. So before removing anything, it's just $|Q|$, then $|Q| - 1$ and so on. Giving us the sum

$$\sum_{k=0}^{|S|-1} |Q| - k = \sum_{k=1}^{|S|} |Q| - k = |S|\,|Q| - \frac{|S|}{2}(|S| + 1).$$

Since $|S|\,, |Q| \in O(n)$, and using product and sum rules for big-O, we get that this is all

$$\frac{1}{2}(O(n^2) - O(n)) = O(n^2).$$

In the case that $Q \subseteq S$, and the additional condition on S, we get the same sum, but over $|Q|$ instead, that is, still $O(n^2)$. $\square$

Exercise 6:

First we note that if we have k people, 2^k different subsets of people can be created, and we can reason that each subset can represent a room being visited by the people that are in the subset, giving us that we need at most $\lfloor \log_2(n) \rfloor + 1$ people to be able to identify each room uniquely, in the following way. For a given subset of $G = \{p_0, p_1, \ldots, p_k\}$, the room that subset visits can be expressed in terms of the binary reresentation of the room number, ranging from 1 to n. That is, if R_i is the room with index i, where $1 \leqslant i \leqslant n$. Then we know that i can be uniquely represented in binary using an ordered list of length $k + 1$ so that

$$[i]_2 = (a_0, a_1, \ldots, a_{k+1}), \text{ where } a_j \in \{0, 1\}.$$

From here we can say that person P_h is assigned to room R_i, if $a_h = 1$ in the binary representation, and so each subset of people is assigned to a unique room, and we are done.