ADS2324 Coursework

Question 1: Algorithmic Thinking and Recursion

(a) Implement a function using recursion named `alpha_recursive` that takes a lowercase alphabetic string `s` and returns the number of elements in the string which are vowels. For example, `alpha_recursive('abeeafqi')` returns 5, as a, e, and i are vowels, with a and e appearing twice in the input string.

(b) Implement an iterative version of your solution to the above and name the function `alpha_iterative`.

(c) Implement a function named `jacobstahl` that uses recursion to compute the n-th Jacobstahl number J_n, which is defined as follows:

$$J_n = \begin{cases} 0 & \text{for } n = 0 \\ 1 & \text{for } n = 1 \\ J_{n-1} + 2J_{n-2} & \text{for } n \geq 2 \end{cases}$$

Your function should take one argument (`n`) and return the n-th Jacobstahl number.

(d) Adapt your implementation in some way to minimize the number of recursive calls your function makes, short of rewriting it entirely to remove the recursion. Name this function `jacobstahl_optimized`. You may use any of the data structures you have encountered in Part I of this module for this purpose. You must not use any `import` statements.

Question 2: Hash Tables and Collision Handling

(a) Draw the hash table that results from applying the hash function

$$h(k) = (4k + 13) \mod 11$$

to the keys 6, 27, 13, 36, 25, 5, 17, 41, and 23 in that order. Assume that collisions are handled using Separate Chaining.

(b) Using the same hash function and keys as above, draw the hash table assuming we now deal with collisions using Linear Probing.

(c) Using the same hash function and keys as above, draw the hash table assuming we now deal with collisions using Quadratic Probing.

(d) Finally, draw the hash table using the same hash function and keys as above, but now collisions are handled using Double Hashing. The secondary hash function is $h'(k) = 5 - (k \mod 5)$.

Question 3: Sequential Abstract Data Types

(a) Starting from the scaffolding provided in `q3_supplementary.ipynb`, complete the definition for a new class called `ArrayDeque` that implements a Deque (double-ended queue), where your implementation is backed by an array. The methods that your Deque must support are detailed in the notebook. You can use the Python `list()` object in your class for this purpose.

(b) Using your new implementation, write a function `lvp(s)` that accepts a non-empty string `s`, consisting of a sequence of open and close parentheses. Your function should return the length of the longest substring consisting of valid matched parentheses. Valid matched parentheses are defined in the lecture material.

For example:

- Given the input `s1 = '()())'`, the function returns 4.
- Given the input `s2 = '(((()))(()'`, the function returns 8.

- Given the input `s3 = ')((((('`, the function returns 0.

Note that valid matched parentheses may themselves be nested (as in `s2`).

Question 4: Recursive Functions for Target Sum and Balanced Code

(a) The function `target_sum(S, t)` has as input a collection of positive integers `S` and a further integer `t`, and requires that numbers from `S` are selected whose sum is `t`. For example, if `S = [1, 2, 3]` and `t = 5`, then `target_sum([1, 2, 3], 5)` returns `[2, 3]`. Or, for a different pair of inputs, if `S = [1, 4, 5, 8, 12, 16, 17, 20]` and `t = 23`, then the solution is `[1, 5, 17]`.

Note that the order of the numbers in the solution is not important. If there is more than one possible solution, only one needs to be found.

(b) A binary string is a string consisting exclusively of digits drawn from the set $\{0, 1\}$. Let us call the balanced binary code of size k the collection of all binary strings of length $2k$ such that for each string, the number of zeros in the first k bits is the same as the number of zeros in the second k bits.

For example:

- The balanced code of size 1 is: `00, 11`
- The balanced code of size 2 is: `0000, 0101, 0110, 1001, 1010, 1111`
- The balanced code of size 3 is: `000000, 001001, 001010, 001100, 010001, 010010, 010100, 011011, 011101, 011110, 100001, 100010, 100100, 101011, 101101, 101110, 110011, 110101, 110110, 111111`

Write a function `balanced_code(k)` which takes an integer k as its input and returns the balanced binary code of size k, represented as a Python list of strings.

Question 5: Asymptotic Notations

Prove or disprove each of the following statements. You will get 1 mark for correctly identifying whether the statement is True or False, and 1 mark for a correct argument.

(a) $4x^2 - 2x$ is $O(x^2)$.
(b) $2x\sqrt{x}$ is $o(2^x)$.
(c) $x + x \log_2 x$ is $\Theta(x)$.
(d) $100x^4$ is $\Omega(16 \log_2 x)$.
(e) $x + x^2 + 100x^3 + 10x^2 \log_2 x$ is $\omega(x^3)$.

Question 6: Master Theorem Applications

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, state why the Master Theorem cannot be applied. You should justify your answers.

(a) $T(n) = 3T(n/9) + \sqrt{n}$
(b) $T(n) = 4T(n/4) + 5n \log_2 n$
(c) $T(n) = 2T(n) + n$
(d) $T(n) = 4T(n/2) + n^3$
(e) $T(n) = 4T(n/8) + 7n^{0.6}$

Question 7: AverageQuickSort

Consider the problem of sorting a list of numbers into decreasing order. You can assume that no two of the numbers in the list are identical. This question requires you to implement (in Python) an algorithm called `AverageQuickSort` that is similar to QuickSort, but sorts into decreasing order and partitions a list based on the average of all the elements of the list. The algorithm `AverageQuickSort`, when called for the part from index ℓ to index r (inclusive) of a list `L`, should work as follows:

- If $r \leq \ell$, do nothing.
- Otherwise, proceed as follows:
  1. Let $L(\ell, r)$ denote the part of `L` from index ℓ to index r (inclusive).
  2. Calculate the average v of the values in $L(\ell, r)$ by adding up the values and dividing the sum by $r - \ell + 1$.
  3. Rearrange (in $O(r-\ell+1)$ time) the values in $L(\ell, r)$ so that the values $\geq v$ are in the beginning of $L(\ell, r)$ and the values $< v$ are at the end of $L(\ell, r)$. Let p be the largest index containing a value $\geq v$ after this rearrangement.
  4. Make recursive calls for $L(\ell, p)$ and for $L(p + 1, r)$.

(a) Write a Python program that contains the following functions:

```
def find_average(L, l, r):
    """ Calculate the average of the values in L[l:r+1] and return it """

def rearrange(L, l, r, v):
    """ Rearrange the part of L from index l to index r
    (inclusive) so that values >=v come first, then the
    values <v. It is guaranteed that L contains at least
    one value <= v and at least one value > v. Return
    the largest index of any element with value >= v.
    """

def average_quick_sort(L, l, r):
    """ Sort the list L in-place (i.e. by modifying L directly)
    using AverageQuickSort. No return value.
    """
```

(b) Now assume that we use a call `average_quick_sort(L, 0, len(L) - 1)` to sort a list `L` with n numbers into decreasing order.

(i) What is the worst-case time complexity of `AverageQuickSort`? Find the best $O(f(n))$ that you can, and justify your answer.

(ii) Describe how one can construct inputs for `AverageQuickSort` on which the algorithm's time complexity is equal to the worst case (up to constant factors).