Solutions to Week 13 (MEGA FSG) Exercises

Ibrahim Chehab

April 11, 2024

1 Did Somebody say Palindrome?

Implement a *recursive* function that checks whether a given string is a palindrome.

RESTRICTIONS:

i. This function **MUST** be implemented using recursion.

ii. This function must **NOT** mutate the original word/sentence.

iii. You may use slicing, but you may **NOT** use the built-in reversal `[::-1]` or the `reversed()` function.

Here are some examples:

(a) `ewe`

(b) `anna`

(c) `borrow or rob`

(d) `taco cat`

(e) `was it a car or a cat i saw`

(f) `racecar`

Hint: Recall Ibrahim's recursion analogy

Solution:

This problem requires a common algorithm known as *two-pointer*, however we will need to modify it to fit the recursive nature of the problem. We will abuse *string splicing* to achieve this. The basic idea is to keep shrinking the string by comparing the first and last characters of the string. If they are equal, we remove them and continue the process. If they are not equal, we return `False`. If the string is empty or has only one character, we return `True`. If we reach a space, we move only the affected pointer over one and continue the process.

```
def is_palindrome(word: str) -> bool:
    if len(word) <= 1: # If we have an empty string or a string with one character
    , it is a palindrome. This is our base case.
        return True
    if word[0] == ' ': # If our "left" pointer is a space, we move it to the right
     and call the function again.
        return is_palindrome(word[1:])
    if word[-1] == ' ': # Similarly, if our "right" pointer is a space, we move it
     to the left and call the function again.
        return is_palindrome(word[:-1])
    return word[0] == word[-1] and is_palindrome(word[1:-1]) # If the characters
    at the left and right pointers are the same, we move both pointers towards the
    center and call the function again. If they are not the same, we return False.
```

Note that it is also possible to solve this problem iteratively, and the iterative solution will make the *two pointer* approach more explicit. However, the recursive solution is more elegant and concise.

Nonetheless, the iterative solution is provided below for reference:

```
def is_palindrome(word: str) -> bool:
    left, right = 0, len(word) - 1
    while left < right:
        if word[left] == ' ':
            left += 1
            continue
        if word[right] == ' ':
            right -= 1
            continue
        if word[left] != word[right]:
            return False
        left += 1
        right -= 1
    return True
```

It is reccomended to visually draw out the iterative solution if you're having trouble understanding it. Using a string and two arrows to represent the left and right pointers can be very helpful. If you're still having trouble, reach out to someone for help :)

2 InsertMii

Consider the following implementation of a Doubly Linked List:

```
class DLLNode:
  """A node in a linked list."""
  item: Any
  next: Optional[DLLNode]
  prev: Optional[DLLNode]

class DoublyLinkedList:
  """A doubly linked list."""
  _first: Optional[DLLNode]
  _last: Optional[DLLNode]

  # Implementation omitted
```

Implement the following method in the `DoublyLinkedList` class:

```
def insert_last(self, value: Any, after: Any) -> bool:
    """Insert a new Node with the value <value> after the LAST occurrence of the
    value <after> in this list.
    If <after> does not exist in the list, then do not insert anything and return
    False.
    The list must be correctly linked after this operation.
    >>> sl = CustomDLL([7, 2, 7, 3])
    >>> str(sl)
    '7 2 7 3'
    >>> sl.insert_last(5, 7)
    True
    >>> str(sl)
    '7 2 7 5 3'
    >>> sl.insert_last(9, 8)
    False
    >>> str(sl)
    '7 2 7 5 3'
```

```
    """
```

Solution:

There are *two* ways to approach this problem. The first way is to iterate *backwards* through the list and insert the new node after the first occurrence of the value `after`. The second way is to iterate *forwards* through the list and insert the new node after the last occurrence of the value `after`. The first way is more efficient, but the second way is harder and as such is reccomended. Nonetheless, we will provide both solutions below.

Solution 1: Iterating backwards

The general idea of this solution is to iterate backwards through the linked list (remember, it's doubly so we have `prev` pointers) like a traditional linkedlist. Once we find the node we're looking for, we insert normally (albeit with more pointers to update) and return `True`. If we reach the beginning (end) of the list without finding the node, we return `False`.

```
def insert_last(self, value: Any, after: Any) -> bool:
    if self._last is None: # If the list is empty, we return False.
        return False
    current = self._last
    while current is not None:
        if current.item == after: # If we find the node we're looking for, we
    insert the new node after it.
            new_node = DLLNode(value)
            new_node.prev = current
            new_node.next = current.next
            if current.next is not None:
                current.next.prev = new_node
            current.next = new_node
            if current == self._last: # If the node we're looking for is the last
    node, we update the last pointer.
                self._last = new_node
            return True
        current = current.prev
    return False # If we reach the beginning of the list without finding the node,
     we return False.
```

Solution 2: Iterating forwards

This approach is similar to the previous approach, however requires iterating forward and keeping track of the last occurrence of the value `after`. Once we reach the end of the list, we insert the new node after the last occurrence of the value `after`. If we don't find the value, we return `False`.

```
def insert_last(self, value: Any, after: Any) -> bool:
    if self._first is None: # If the list is empty, we return False.
        return False
    current = self._first
    last_occurrence = None
    while current is not None:
        if current.item == after: # If we find the node we're looking for, we
    update the last occurrence pointer.
            last_occurrence = current
        current = current.next
    if last_occurrence is None: # If we don't find the node, we return False.
        return False
    new_node = DLLNode(value) # Otherwise, we insert the new node after the last
    occurrence of the node.
    new_node.prev = last_occurrence
    new_node.next = last_occurrence.next
    if last_occurrence.next is not None:
        last_occurrence.next.prev = new_node
    last_occurrence.next = new_node
```

```
    if last_occurrence == self._last: # If the last occurrence is the last node,
    we update the last pointer.
        self._last = new_node
    return True
```

3 The Even-Worse-Stack

Nugget has entered their *evil era* and designed an evil ADT known as the `EvenWorseStack`. They've subjected Therapist to the `EvenWorseStack` and now Therapist is in a state of despair. Help Therapist by analyzing the time complexity of the `pop` method of the `EvenWorseStack` class.

```
class EvenWorseStack:
    """
    A Stack implementation designed to be slow and inefficient.
    """
    _stack: Queue

    def __init__(self) -> None:
        self._stack = Queue()

    def push(self, value: int) -> None:
        self._stack.enqueue(value)

    def pop(self) -> int:
        temp = Queue()
        while self._stack.size() > 1:
            temp.enqueue(self._stack.dequeue())
        value = self._stack.dequeue()
        self._stack = temp
        return value

class Queue:

  _queue: list[int]

  def __init__(self) -> None:
      self._queue = []

  def enqueue(self, value: int) -> None:
      self._queue.insert(0, value)

  def dequeue(self) -> int:
      index_to_remove = self.size() - 1
      value = self._queue[index_to_remove]
      self._queue = self._queue[:index_to_remove]
      return value

  def size(self) -> int:
      return len([i for i in self._queue])
```

What is the time complexity of the `pop` method?

Solution:

To find the time complexity of the `pop` method, we must first analyze the time complexities of the relevant `Queue` methods.

1. `enqueue`: The `enqueue` inserts at the front of a list. We know that this is a $\mathcal{O}(n)$ operation, where n is the size of the list.

2. `dequeue`: The `dequeue` method removes the last element of the list. However, it does this via slicing, which is also a $\mathcal{O}(n)$ operation.

3. `size`: The `size` method uses a list comprehension to recreate the list then invokes the `len` function. The list comprehension is a $\mathcal{O}(n)$ operation, and the `len` function is a $\mathcal{O}(1)$ operation. Therefore, the `size` method is a $\mathcal{O}(n)$ operation.

Now, we can find the time complexity of `EvenWorseStack.pop`.

Finding the time complexity of the `pop` method is a bit tricky due to the way the `while` loop is setup. Recall `Queue.size` is a $\mathcal{O}(n)$ operation.

To make our lives easier, we will refactor the `EvenWorseStack.pop` method to make it more readable:

```
def pop(self) -> int:
    temp = Queue()
    while True:
        if self._stack.size() == 1:
            break
        x = self._stack.dequeue()
        temp.enqueue(x)
    value = self._stack.dequeue()
    self._stack = temp
    return value
```

The new method is logically equivalent to the old one, but it is easier to analyze. The `while` loop will run until the size of the `self._stack` is 1. In the worst case, the `while` loop will run $n-1$ times, where n is the size of the `self._stack`. Hence, this is an $\mathcal{O}(n)$ operation.

Within this while loop, we have a call to `self._stack.size()`, which is also an $\mathcal{O}(n)$ operation. This brings us to a total of $(n-1)\cdot(n)$ steps.

We then have to assign the value of `self._stack.dequeue()` to `x`, which is an $\mathcal{O}(n)$ operation. We then enqueue `x` into `temp`, which is also an $\mathcal{O}(n)$ operation. This brings us to a total of $(n-1)\cdot(n+n)$ steps.

Finally, we dequeue the last element from `self._stack`, which is an $\mathcal{O}(n)$ operation. This brings us to a total of $(n-1)\cdot(n+n)+n = (n-1)\cdot(2n)+n$ steps.

Expanding the above, we get $2n^2-2n+n = 2n^2-n$. This is a $\mathcal{O}(n^2)$ operation.

Therefore, the time complexity of the `pop` method is $\mathcal{O}(n^2)$.

4 Efficiencii

Select all the statements that are **TRUE**:

1. The n_0 you choose does not change the final result of the efficiency class
2. If a function is upper-bounded by $\mathcal{O}(n^2)$, it might still be $\Theta(n)$
3. If a function is upper-bounded by $\mathcal{O}(n)$, it might still be $\Theta(n^2)$
4. The iterative part of `QuickSort` is faster than the iterative part of `MergeSort`
5. The recursive part of `QuickSort` is faster than the recursive part of `MergeSort`
6. If a function is $\mathcal{O}(g(n))$, then it is also $\Theta(g(n))$
7. If a function is $\Theta(g(n))$, then it is also $\mathcal{O}(g(n))$

Solution: We will go through each statement one by one:

1. **False.** Consider the following function:

```
def mystery(n: int):
    L = []
    for i in range(n):
        for j in range(min(50, n)):
            L.insert(0, j)
```

Choosing $n_0 \leq 50$ will make this function *appear* to be $\mathcal{O}(n^3)$, however choosing $n_0 > 50$ will make this function $\mathcal{O}(n^2)$. Therefore, choosing n_0 *can* change the final result of the efficiency class. (But not always)

2. **True**. Recall the definition of $\mathcal{O}$ notation:

$$f(n) \in \mathcal{O}(g(n)) \iff \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n) \text{ for all } n \geq n_0 \tag{1}$$

Generalizing this for $g(n) = n^2$, we have:

$$f(n) \in \mathcal{O}(n^2) \iff \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cn^2 \forall n \geq n_0 \tag{2}$$

Consider the following function:

```
def mystery(n: int):
    for i in range(n):
        print(i)
```

This function is clearly $\mathcal{O}(n)$ by inspection, however it is vacuously true that if $f(n) \leq cn$ for all $n \geq n_0$ for some $c > 0$ and $n_0 > 0$, then $f(n) \leq cn^2$ (By definition of $\mathcal{O}$) Therefore, if a function has a time complexity of $\mathcal{O}(n^2)$, it might still be $\Theta(n)$.

3. **False.** Recall the definition of Θ notation:

$$f(n) \in \Theta(g(n)) \iff \exists c_1, c_2 > 0, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0 \tag{3}$$

If $f(n) \in \mathcal{O}(n)$, then n^2 grows too fast to lower-bound $f(n)$, no matter which constant we choose.

Hint! If this is confusing you, try messing around in Desmos!

4. **True.** Refer to your course notes

5. **False.** Refer to your course notes

6. **False.** Consider the following function:

```
def mystery(n: int):
    for i in range(n):
        print(i)
```

This function is clearly $\mathcal{O}(n)$ by inspection, and we know from a previous example that it is also $\mathcal{O}(n^2)$. However, we can clearly see the tightest upper-bound is $\mathcal{O}(n)$. n^2 grows too fast to lower-bound $f(n)$, no matter which constant we choose. Therefore this is false and not true for all functions $f(n)$.

7. **True.** Recall the definition of Θ notation:

$$f(n) \in \Theta(g(n)) \iff \exists c_1, c_2 > 0, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0 \tag{4}$$

Notice, the right-hand side of the inequality is the definition of $\mathcal{O}$ notation:

$$f(n) \in \mathcal{O}(g(n)) \iff \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n) \text{ for all } n \geq n_0 \tag{5}$$

Therefore, if a function is $\Theta(g(n))$, then it is also $\mathcal{O}(g(n))$.

5 The Final Challenge

Refer to Week 10 solutions for the final challenge, as it is the same question.