FIT1008 Introduction to Computer Science (FIT2085 for Engineers)

Tutorial 9

Semester 1, 2019

Objectives of this tutorial

- To understand recursion.
- To understand quicksort and merge sort.

Exercise 1 *

Consider a `Node` class which defines a node for a linked data structure, and which is defined as follows:

```python
class Node:
        def __init__(self, item = None, link = None):
                self.item = item
                self.next = link
```

Suppose you have a `List` class that implements a Linked List using the `Node` class above, and has the following method.

```python
def mystery(self):
        return mystery_aux(self.head)

def mystery_aux(self, current):
        if current == None:
                return 0
        else:
                current.item +=  mystery_aux(current.next)
                return current.item
```

(a) What does the `mystery` method do? Explain in terms of its effect on the value of `a_list`, that consists of the following items in order 1,2,3,4,5.

(b) What is the best and worst complexity in Big O notation of our `mystery()` method in terms of the length of the list (N)?

(c) How would you define the method iteratively?

Solution

(a) A call to `a_list.mystery()` where `a_list` contains the elements 1,2,3,4,5 with 1 being at the head, would traverse the entire list doing nothing in the way to the last node, while in the way back will modify the content of each node by adding to it the content of the successor node (i.e., the content of the node that used to be 4, will now be 4+5=9, the content of the node that had a 3 will now be 3+9=12, and so on). For this example, the list would be returned as 15, 14, 12, 9, 5, with 15 being at the head.

(b) The best and worst case time complexity is O(N) where N is the number of elements in the list, since the recursion traverses every element (regardless of the actual numbers in the list) and, for each, if performs exactly the same constant time operations (+=).

(c) The method could be defined iteratively by using two stacks as follows:

```python
def mystery(self):
        current = self.head
        my_stack = Stack()
        while current is not None:
                my_stack1.push(current.item)
                current = current.next

        my_stack2 = Stack()
        int temp = 0
        while not my_stack1.is_empty():
                temp += my_stack1.pop()
                my_stack2.push(temp)

        current = self.head
        while current is not None:
                current.item = my_stack2.pop()
                current = current.next
```

The first loop puts the elements in reverse order into `my_stack1` (i.e., the elements are pushed into `my_stack1` as 5,4,3,2,1 with 5 being at the top). The second loop empties `my_stack1` creating a new stack `my_stack2` whose elements are the sum of those already popped from `my_stack1` (i.e., the elements are pushed into `my_stack2` as 15,14,12,9,5 with 15 being at the top). And the third loop empties `my_stack2` transferring the content to the nodes in the list.

Clearly, the recursive method is much simpler and clearer.

Exercise 2 *

(a) Write a *recursive* method for computing the sum of the digits of a number. For example, for number 979853562951413, the sum of its digits is $9+7+9+8+5+3+5+6+2+9+5+1+4+1+3 = 77$. To do this you can use integer division by 10 (//10) which returns an integer with the same digits except the last one, and reminder by 10 (% 10), which returns the last digit. For example, if you have X = 3456, then X//10 gives you 345, while X%10 gives you 6.

(b) Determine its complexity, in Big-O notation.

Solution

1. **Simple recursive implementation**

```
def sumDigit(n):
        if n==0:
                return 0
        else:
                return (n%10)+sumDigit(n//10)
```

   Tail recursive implementation

```
def sumDigit(n,s=0):
        if n==0:
                return s
        else:
                return sumDigit(n//10,s+(n%10))
```

2. The complexity of both of these implementations is based on the base 10 logarithm of the input number (as this tells you have many digits there are and each digit is considered exactly once). Hence complexity for both is $O(\log n)$. This is because this recursive function effectively implements the same algorithm as the one we saw in Week 5, hence the same argument holds.

Exercise 3 *

In Quicksort, the choice of pivot is crucial. Discuss the reasons for this and give some examples of good/bad choices

Solution

The pivot is the deciding factor in how "balanced" are the two partitions. The best pivot is one that guarantees that both partitions have the same length plus/minus one (i.e., absolute difference value of one). Such pivot will result in a Quick sort method that has a complexity of $O(N \log N)$, where N is the size of the list.

The worst pivot is one that forces all elements into a single partition, leaving the other one empty. Such pivot will result in a Quick sort method that has a complexity of $O(N^2)$.

Note that the position of the pivot in the array (i.e., whether you choose the first, last, middle, etc) makes no difference, the difference is in how it partitions the elements in the array.

An approach that achieves reasonable behaviour while being simple and fast to compute, is to read three numbers and take the median one. Also, if the input is relatively sorted, the middle element is quite a good choice. For reasonably large lists, a (truly) random index is also a good choice.

Exercise 4 *

Are Mergesort, or Quicksort stable? Discuss and provide examples.

Solution

Merge sort is stable as coded in the lectures because it always takes elements (if any) from the right hand side of the array before it considers elements from the left hand side.

The common Quick sort is not a stable algorithm due to the way the partitioning algorithm is defined (think about the swap with the pivot element and how it can modify the relative order between two elements with the same key).

However, note that any sort algorithm can be made stable by adjusting the compare function. Before sorting, annotate each element with an integer denoting its index. Then, adjust the compare function so that whenever it says that two elements are equal, it uses the extra index data to determine the correct order. This adds a small linear cost (annotating the array) and a small constant cost to every comparison that would return "equal".

Exercise 5

Definition: The *digital root* of a decimal integer is obtained by adding up its digits, and then doing the same to *that* number, and so on, until you get a single digit, which is the digital root of the number you started with.

For example, to find the digital root of 979853562951413, we calculate: sum of digits $= 9+7+9+8+5+3+5+6+2+9+5+1+4+1+3 = 77$, then sum of digits $= 7+7 = 14$, then sum of digits $= 1+4 = 5$. Now we have just one digit, 5, so that's the digital root of the number we started with.

(c) Write a *recursive* method to compute the digital root of a positive integer.

(d) Determine its complexity, in Big-O notation.

Solution

1. **Tail recursive implementation using sumDigit**

```
def digitalRoot(n):
        if n\\10==0:
                return n
        else:
                return digitalRoot(sumDigit(n))
```

2. Even though the implementation is recursive, the algorithm is the same as the one we saw in Week 5. Hence the analysis and the running time are the same as well, and the complexity is $O(\log n)$.