Lab 3

Luca Novello | DSA456V1A | March 3, 2025

Part A – Recursive Functions

- Write the following python functions recursively.
- A non-recursive solution that works will not be given credit (even if it passes testing)

Function 1:

This function is passed a number and returns (number)(number-1)(number-2)...(3)(2)(1). By definition, 0! = 1. Only a recursive solution will be accepted!
def factorial(number)

Answer:

```
def factorial(number):
    if number == 0 or number == 1:
        return 1
    return number * factorial(number - 1)
```

Function 2:

*Write the **RECURSIVE** function linear_search. linear_search() is passed a list of values and a key. If a matching key is found in the list, the function returns an index of where the key was found. If the key is not found, function returns -1.*

NOTE**: you are not allowed to use any of the built in list functions for this problem. The only function you are allowed to use is **len()

def linear_search(list, key)

***HINT**: you may need to write the actual recursive function with a different set of arguments to accomplish this task.*

Answer:

```
def linear_search(lst, key, index=0):
    if index >= len(lst):
        return -1
    if lst[index] == key:
        return index
    return linear_search(lst, key, index + 1)
```

Function 3:

Write the **RECURSIVE** function binary_search. **binary_search()** is passed a sorted list of values and a key. If a matching key is found in the list, the function returns an index of where the key was found. If the key is not found, function returns -1.

NOTE: you are not allowed to use any of the built in list functions for this problem. The only function you are allowed to use is **len()**

def binary_search(list, key)

HINT: you may need to write the actual recursive function with a different set of arguments to accomplish this task.

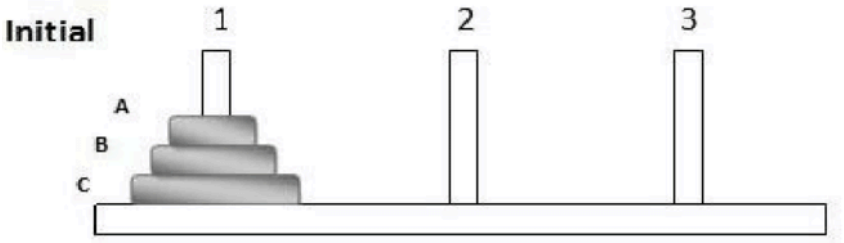
Answer:

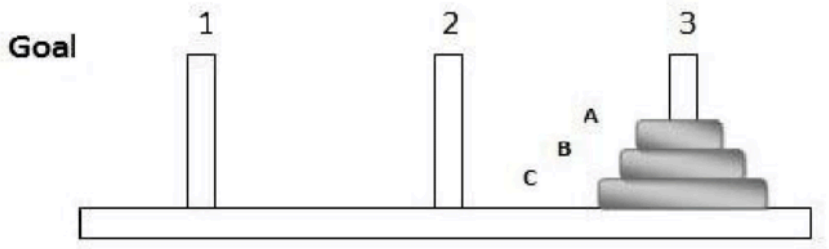
```
def binary_search(lst, key, left=0, right=None):
    if right is None:
        right = len(lst) - 1

    if left > right:
        return -1

    mid = (left + right) // 2

    if lst[mid] == key:
        return mid
    elif lst[mid] > key:
        return binary_search(lst, key, left, mid - 1)
    else:
        return binary_search(lst, key, mid + 1, right)
```

Function 4:

The purpose is to move all the disks from tower 1 to tower 3 using tower 2. The rule is that no disk can be put on a smaller disk. In other words, the order of the disks should be maintained all through the transitions. Each move consists of moving ONE disk from a tower to the other. You can only move a disk if there is no other disk on top of it; otherwise the disk on top should be moved first. To summarize, here are the rules:

1. Only one disk can be moved at a time.
2. A disk can only be moved if it is the uppermost disk in the pole.
3. A larger disk can't be placed on a smaller disk.





Write a recursive program that gets the number of disks on tower 1 and lists all the moves to take them to tower 3 using tower 2. The moves should be like:

- 1 to 2 (which means the top disk on tower 1 should be moved to tower 2)
- 1 to 3
- 2 to 3

Answer:

```
def tower_of_hanoi(n, source=1, auxiliary=2, destination=3):
    if n == 1:
        print(f"{source} to {destination}")
        return
    tower_of_hanoi(n - 1, source, destination, auxiliary)
    print(f"{source} to {destination}")
    tower_of_hanoi(n - 1, auxiliary, source, destination)
```

Part B – Analysis

- Perform an analysis of the following recursive functions.

Function 1:

Analyze the following function with respect to number:

```
def function1(value, number):
    if (number == 0):
        return 1
    elif (number == 1):
        return value
    else:
        return value * function1(value, number-1)
```

Answer:

Step 1: Establish Variables and Functions

- Let ***number*** represent the input exponent.
- Let ***T(n)*** represent the number of operations required to compute the ***value^number***.

Step 2: Count Your Operations

- Base case (number == 0): 1 operation (return 1).
- Base case (number == 1): 1 operation (return value).
- Recursive call:
  - ***value * function1(value, number - 1)***: 1 multiplication per call.
  - Calls itself ***number*** times (each time reducing ***number*** by 1).

Total:

- If ***number = n***, the function makes ***n*** recursive calls, each performing 1 multiplication.
- Thus, total operations: n multiplications + 1 return statement
- Total: n + 1 operations

Step 3: Establish Mathematical Expression

$$T(n) = n + 1$$

Step 4: Simplify the Equation

- The highest-order term is n, and constants are ignored.

Step 5: State Your Final Result

Therefore, T(n) is O(n).

Function 2:

Analyze function2 with respect to the length of the mystring. Hint, you will need to set up two mathematical functions for operator counting. one for function2 and the other for recursive_function2

```
def recursive_function2(mystring,a, b):
    if(a >= b ):
        return True
    else:
        if(mystring[a] != mystring[b]):
            return False
        else:
            return recursive_function2(mystring,a+1,b-1)

def function2(mystring):
    return recursive_function2(mystring, 0,len(mystring)-1)
```

Answer:

Step 1: Establish Variables and Functions

- Let ***n*** be the length of ***mystring***.
- Let ***T(n)*** be the number of operations to check if ***mystring*** is a palindrome.

Step 2: Count Your Operations

- function2(mystring): Calls ***recursive_function2(mystring, 0, len(mystring) - 1)*** - 1 operation.
- recursive_function2(mystring, a, b):
  - Base case (a >= b): 1 operation (returns True).
  - Recursive case (a < b):
    - Comparison: 1 operation.
    - Recursive call: 1 operation.
  - Each recursive step reduces the problem by 2 (***a++***, ***b--***).
- Total operations: ***n/2*** comparisons + initial call → $T(n) = \frac{n}{2} + 1$

Step 3: Establish Mathematical Expression

- recursive_function2: $T_{recursive}(n) = \frac{n}{2}$
- function2: $T(n) = \frac{n}{2} + 1$

Step 4: Simplify the Equation

- Discard constants: $T(n) = O(n)$

Step 5: Final Result

- Time complexity: ***O(n)***.

Function 3:

Analyze the following function with respect to number:

```
def function3(value, number):
    if (number == 0):
        return 1
    elif (number == 1):
        return value
    else:
        half = number // 2
        result = function3(value, half)
        if (number % 2 == 0):
            return result * result
        else:
            return value * result * result
```

Answer:

Step 1: Establish Variables and Functions

- Let ***n*** represent the input ***number***.
- Let ***T(n)*** represent the number of operations required to compute $value^n$ using this optimized function.

Step 2: Count Your Operations

- Base case (***number == 0***): 1 operation (***return 1***).
- Base case (***number == 1***): 1 operation (***return value***).
- Recursive case:
  - Compute **half = number // 2** → 1 operation.
  - Recursively call ***function3(value, half)*** → 1 recursive call.
  - 1 multiplication for ***result * result*** or ***value * result * result*** depending on even/odd.

Step 3: Establish Mathematical Expression

- The recursion depth is ***O(log n)*** due to the halving of ***number*** (***number // 2***).
- Each recursive step performs a constant amount of work (***O(1)***).

Thus, the total operations:

$T(n) = O(logn)$

Step 4: Simplify the Equation

- The highest-order term is **log n**.

Step 5: State Your Final Result

Therefore, the time complexity is ***O(log n)***.