course_content

Data Structures and Algorithms

Recursion

Week 6

What is Recursion?

- When a solution to a problem depends on smaller instances of the same problem and a function calls itself to solve the problem

Example:

- The nth term of the Fibonacci sequence = the sum of the last two terms in the sequence.
- The factorial of a non-negative integer n is the product of n and the factorial of (n - 1)

Rules of a Recursive Algorithm

- A base case of the algorithm
- An action that changes the state of the algorithm towards the base case
- A recursive call to the same function

You should be thinking about **induction** here. It is the programming equivalent.

Iterative Algorithms vs Recursive Algorithms

- A program is called recursive when an entity calls itself. A program is call iterative when there is a loop (or repetition).
- **Iterative** - for loops and while loops
- **Recursive** - calls back to the function itself

Iterative to Recursive - I

➔ Given a non-negative integer n as input, write an iterative function to find the factorial of that integer.

Iterative

```
def factorial(n):
    product = 1
    for i in range(1,n+1):
        product *= i
    return product
```

Recursive

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Iterative to Recursive - II

➔ Given an array of integers, write an iterative function to find the sum of the values in the array. Afterwards, write the recursive version of the function.

Iterative

```
def findSum(arr):
    mySum = 0
    for i in range(len(arr)):
        mySum += arr[i]
    return mySum
```

Recursive

```
def findSum(arr):
    if (len(arr) == 0):
        return 0
    else:
        return arr[-1]+findSum(arr.pop())
```

Iterative to Recursive - III

➔ Given a positive integer n as an input, write an iterative and recursive function that prints all the powers of 2 from 1 through n (inclusive). For example, if n is 4, it would print 1, 2, and 4

Iterative

```
def powersOfTwo(n):
    i = 1
    while i <= n:
        print(i)
        i = i * 2
```

Recursive

```
def powersOfTwo(n):
    if n == 1:
        print(1)
        return 1
    else:
        prevPow = powersOfTwo(n//2)
        nextPow =  prevPow * 2
        print(nextPow)
        return nextPow
```

Asymptotic Analysis
of
Recursive Algorithms

Space Complexity vs Auxiliary Space

- **Auxiliary Space** is the extra space or temporary space used by an algorithm.
- **Space Complexity** of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

Time:
5 min

Factorial - Asymptotic Analysis

Iterative	Recursive

<pre>def factorial(n):
    product = 1
    for i in range(1,n+1):
        product *= i
    return product</pre>
	<pre>def factorial(n):
	    if n == 0:
	        return 1
	    else:
	        return n * factorial(n-1)</pre>

Time: O(N) Space Complexity: O(1) Auxiliary Space: O(1)	Time: O(N) Space Complexity: O(N) Auxiliary Space: O(N)

Time:
5 min

Find Sum - Asymptotic Analysis

Iterative	Recursive

<pre>def findSum(arr):
    mySum = 0
    for i in range(len(arr)):
        mySum += arr[i]
    return mySum</pre>
	<pre>def findSum(arr):
	    if (len(arr) == 0):
	        return 0
	    else:
	        return arr[-1]+findSum(arr.pop())</pre>

Time: O(N)
Space Complexity: O(N)
Auxiliary Space: O(1)

Time: O(N)
Space Complexity: O(N^2) (O(N) with good implementation)
Auxiliary Space: O(N^2) (O(N) with good implementation)

Time:
5 min

Powers of Two - Asymptotic Analysis

Iterative	Recursive

Iterative

```
def powersOfTwo(n):
    i = 1
    while i <= n:
        print(i)
        i = i * 2
```

Time: O(logN)
Space Complexity: O(1)
Auxiliary Space: O(1)

Recursive

```
def powersOfTwo(n):
    if n == 1:
        print(1)
        return 1
    else:
        prevPow = powersOfTwo(n//2)
        nextPow =  prevPow * 2
        print(nextPow)
        return nextPow
```

Time: O(logN)
Space Complexity: O(logN)
Auxiliary Space: O(logN)

Time:
5 min

Closing thoughts

- All recursive algorithms can be implemented iteratively.
- Sometimes implementing an algorithm iteratively may be too complex. You may want to turn to recursion.
- Recursive algorithms are often **more elegant** than iterative algorithms.
- Tradeoffs with Recursion: You may be using **more auxiliary space** when compared to the iterative solution of the problem.

Implementations for the Week - Qn1

➔ An iterative method to search for a given data item in a linked list is shown below. Write the recursive version of this method.

Iterative

```
def search(self, item):
    current = self.head
    while current != None:
        if current.data == item:
            return True
            current = current.next
    return False
```

Implementations for the Week - Qn5

➔ The iterative version of binary search is shown below. **Implement binary search recursively**. Be sure to analyse the time and space complexity of the recursive version.

Iterative Binary Search

```
def binarySearch(arr, searchValue):
    low = 0
    high = len(arr) - 1

    while (low <= high):
        mid = (low + high) // 2
        if (arr[mid] < searchValue):
            low = mid + 1
        elif (arr[mid] > searchValue):
            high = mid - 1
        else:
            return True

    return False
```

Time:
1 min

Questions