15CSE374
INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS

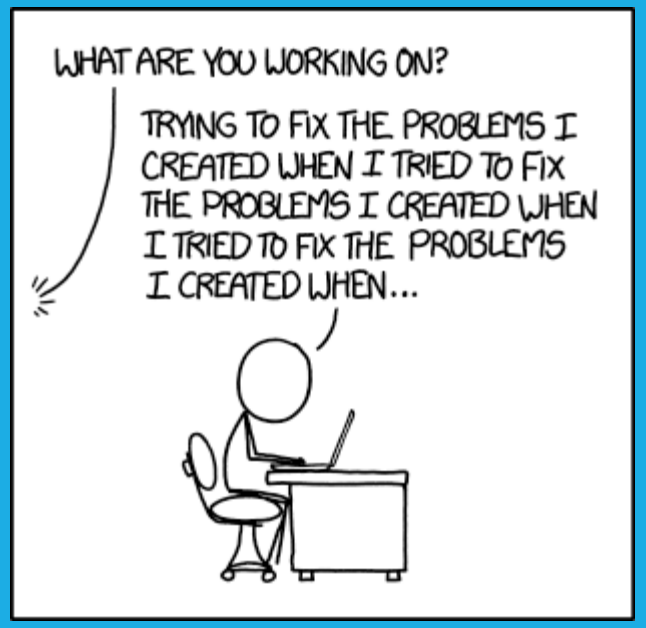
Sarath tv

Last Lecture

- Queue as ADT.
- Implementation of Queue.

Introduction

- When we think about repeating a task, we usually think about the for and while loops.
- Another form, by calling a function within itself, to solve a smaller instance of the same problem.
- Recursion is defined as defining anything in terms of itself.



- In programming languages-A module or function calling itself.
- Can go infinite like a loop.
  - *Base criteria* – there must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
  - *Progressive approach* – the recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria

```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = 3
print("The factorial of", num, "is", factorial(num))
```

Types

- Linear Recursion
- Binary Recursion

- **Linear Recursion**
  - Linear recursion begins by testing for a set of base cases there should be at least one.
- In Linear recursion :
  - Perform a single recursive call. This recursive step may involve a test that decides which of several possible recursive calls to make, but it should ultimately choose to make just one of these calls each time we perform this step.
  - Define each possible recursion call, so that it makes progress towards a base case.
- **Binary Recursion**
  - Binary recursion occurs whenever there are two recursive calls for each non base case.

Linear Recursion

- That only makes a single call to itself each time the function runs

$$n! = n \times (n-1)!$$

$$n! = n \times (n-1) \times (n-2)!$$

$$n! = n \times (n-1) \times (n-2) \times (n-3)!$$

.

.

$$n! = n \times (n-1) \times (n-2) \times (n-3) \cdots \times 3!$$

$$n! = n \times (n-1) \times (n-2) \times (n-3) \cdots \times 3 \times 2!$$

$$n! = n \times (n-1) \times (n-2) \times (n-3) \cdots \times 3 \times 2 \times 1!$$

```
def factorial_recursive(n):
    # Base case: 1! = 1
    if n == 1:
        return 1

    # Recursive case: n! = n * (n-1)!
    else:
        return n * factorial_recursive(n-1)
```

- A *binary-recursive* routine (potentially) calls itself twice.
- The Fibonacci numbers are the sequence:
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, … .
- Each number is the sum of the two previous numbers.

```
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

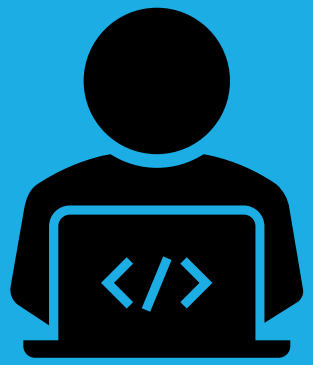
Advantages of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

- It consumes more storage space the recursive calls.
- The computer may run out of memory if the recursive calls are not checked.
- It is not more efficient in terms of speed and execution time.
- Recursive solution is always logical and it is very difficult to trace.(debug and understand).
- In recursive we must have an if statement somewhere to force the function to return without the recursive call being executed, otherwise the function will never return.
- Recursion uses more processor time.

•Recursion may be treated as a software tool *to be applied carefully and selectively*.



THANK YOU!!!!!

Additional Links

- https://www.youtube.com/watch?v=aI3XLMLim8E&feature=youtu.be
- https://www.youtube.com/watch?v=AGhMaWZbivk&feature=youtu.be
- https://www.youtube.com/watch?v=1jm_YCGVwOc&feature=youtu.be
- https://github.com/sarathtv/18ES601_ESP_2020_FALL