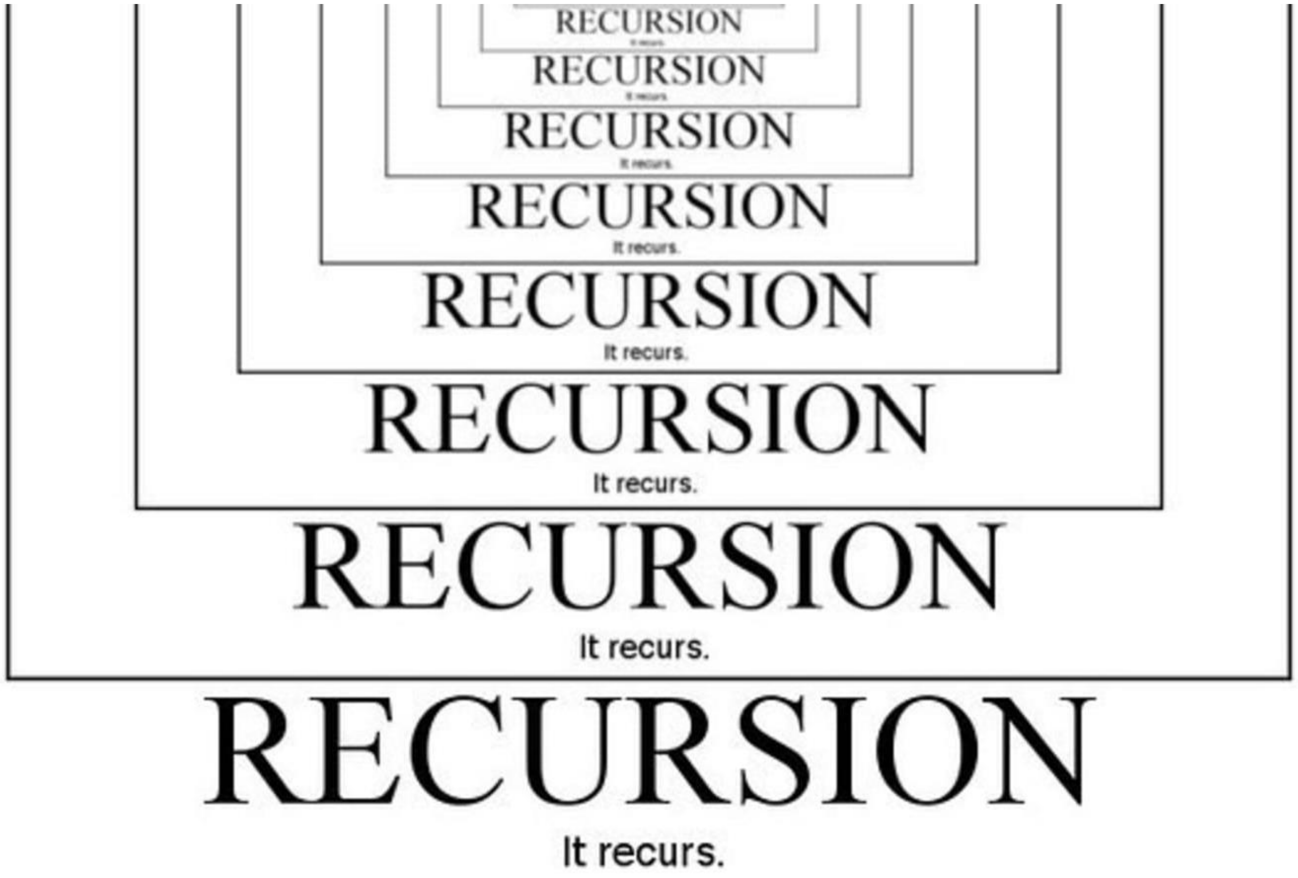
Mansoura University
Faculty of Computers and Information
Department of Information System
First Semester- 2020-2021

Course: Data Structure.

Grade: Second year

Eng. Kholood Salah

RECURSION


RECURSION
RECURSION
RECURSION
RECURSION
It recurs.
RECURSION
It recurs.
RECURSION
It recurs.
RECURSION
It recurs.
RECURSION
It recurs.

Recursion

What is Recursion?

- Recursion allows a function to call itself. Fixed steps of code get executed again and again for new values.
- We also must set criteria for deciding when the recursive call ends.

When to Use Recursion?

- Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially.

Recursion

The Three Laws of Recursion:

- A recursive algorithm must have a **base case.**
- A recursive algorithm must **change its state** and move toward the base case.
- A recursive algorithm must call itself, **recursively**.

A **base case** is the condition that allows the algorithm to stop recursing.

A **change of state** means that some data that the algorithm is using is modified. Usually, the data that represents our problem gets smaller in some way.

We have a problem to solve with a function, but that function solves the problem by calling itself

FACTORIAL FUNCTION USING RECURSION

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

```
print(factorial(6))
```

```
720
```

RECURSIVE BINARY SEARCH

```
def binary_search (arr, item):
    if len(arr)==0:
        return False
    else:
        midpoint=len(arr)//2
        if arr[midpoint]==item:
            return True
        else:
            if item< arr[midpoint]:
                return binary_search(arr[:midpoint],item)
            else:
                return binary_search(arr[midpoint+1:],item)
```

```
arr=[1,2,6,8,9]
print(binary_search(arr,4))
print(binary_search(arr,8))
print(binary_search(arr,1))
```

```
False
True
True
```

BACKTRACKING

What is Backtracking?

- Backtracking is a form of recursion. But it involves choosing only option out of any possibilities.
- In backtracking, if the current solution is not suitable, then we backtrack and try to find other solutions. So, recursion is used.

When to use Backtracking?

- Backtracking is used to solve problems that have multiple solutions.

WHAT IS THE DIFFERENCE BETWEEN BACKTRACKING AND RECURSION?

.. o.	Recursion	Backtracking
	Recursion does not always need backtracking	Backtracking always uses recursion to solve problems
	A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems.	Backtracking at every step eliminates those choices that cannot give us the solution and proceeds to those choices that have the potential of taking us to the solution.
	Recursion is a part of backtracking itself and it is simpler to write.	Backtracking is comparatively complex to implement.
	Applications of recursion are Tree and Graph Traversal, Towers of Hanoi, Divide and Conquer Algorithms, Merge Sort, Quick Sort, and Binary Search.	Application of Backtracking is N Queen problem, Rat in a Maze problem, Knight's Tour Problem, Sudoku solver, and Graph coloring problems.

```python
def permute (repeat,list1):
    xy=[]
    if repeat==1:
        return list1
    else:
        for y in permute(1,list1):
            for x in permute(repeat-1,list1):
                xy.append(x+y)
        return xy
```

```python
print(permute(1,['a','b','c','d']))
print(permute(2,['a','b','c','d']))
```

```
['a', 'b', 'c', 'd']
['aa', 'ba', 'ca', 'da', 'ab', 'bb', 'cb', 'db', 'ac', 'bc', 'cc', 'dc', 'ad', 'bd', 'cd', 'dd']
```

PERMUTE

EXERCISE

Q-1: How many recursive calls are made when computing the sum of the list [2,4,6,8,10]?

A. 6

There are only five numbers on the list, the number of recursive calls will not be greater than the size of the list.

B. 5

The initial call to listsum is not a recursive call.

C. 4

the first recursive call passes the list [4,6,8,10], the second [6,8,10] and so on until [10].

D. 3

This would not be enough calls to cover all the numbers on the list

EXERCISE

Q-2: Suppose you are going to write a recursive function to calculate the factorial of a number. fact(n) returns n * n-1 * n-2 * … Where the factorial of zero is defined to be 1. What would be the most appropriate base case?

A. n == 0

✗ Although this would work there are better and slightly more efficient choices. since fact(1) and fact(0) are the same.

B. n == 1

✗ A good choice, but what happens if you call fact(0)?

C. n >= 0

✗ This basecase would be true for all numbers greater than zero so fact of any positive number would be 1.

D. n <= 1

✓ Good, this is the most efficient, and even keeps your program from crashing if you try to compute the factorial of a negative number.