Faculty of Engineering & Information Technology

Algorithms Analysis and Design

230213150

Thaer Thaher
Thaer.Thaher@aaup.edu

Fall 2023/2024

Introduction to Recursion

Problem-Solving approaches

- In algorithms and programming, there are different ways to approach problem-solving.
- Two common approaches are:
  - Bottom-up "Iterative"
  - Top-down "recursive"

Problem-Solving approaches

- **bottom-up "iterative" approach:**
  - Starting with the simplest subproblem, solving it, and building up to the main problem.
  - Example: Add numbers from *1* to *n* using an iterative approach, starting from 1 and adding the next number in each step.

$$1 + 2 + 3 + 4 + 6 + \ldots\ldots\ldots\ldots + n$$

$$\longrightarrow$$

Write a pseudocode to calculate the sum of numbers from 1 to n using iterative approach

Problem-Solving approaches

- **bottom-up "iterative" approach:**

Write a pseudocode to calculate the sum of numbers from 1 to n using iterative approach

```
function iterativeSum(n):
    // Initialize a variable to store the sum
    sum = 0

    // Iterate from 1 to n and accumulate the sum
    for i from 1 to n:
        sum = sum + i

    // Return the final sum
    return sum
```

Problem-Solving approaches

- **Top-down "recursive" approach:**

- Starting with the main problem and breaking it down into smaller subproblems until reaching the base case.
- Example: Add numbers from 1 to n using a recursive approach, where we first break the problem into **adding n to the sum of numbers from 1 to n-1**.

$$\boxed{1 + 2 + 3 + 4 + 6 + \ldots\ldots n-1} + \boxed{n}$$

$$\text{sum}(1 \text{ to } n-1) \qquad + n$$

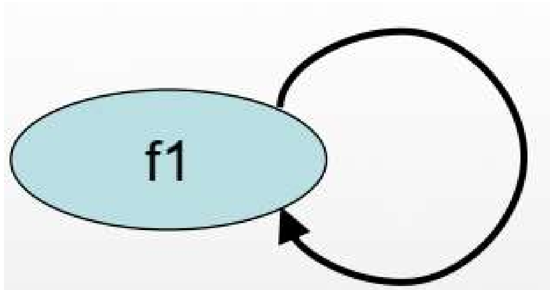
So, What is Recursion???

What is recursion?

- Definition: Recursion is a technique where a function calls itself to solve a problem.
- It **breaks** a problem into smaller, similar subproblems.

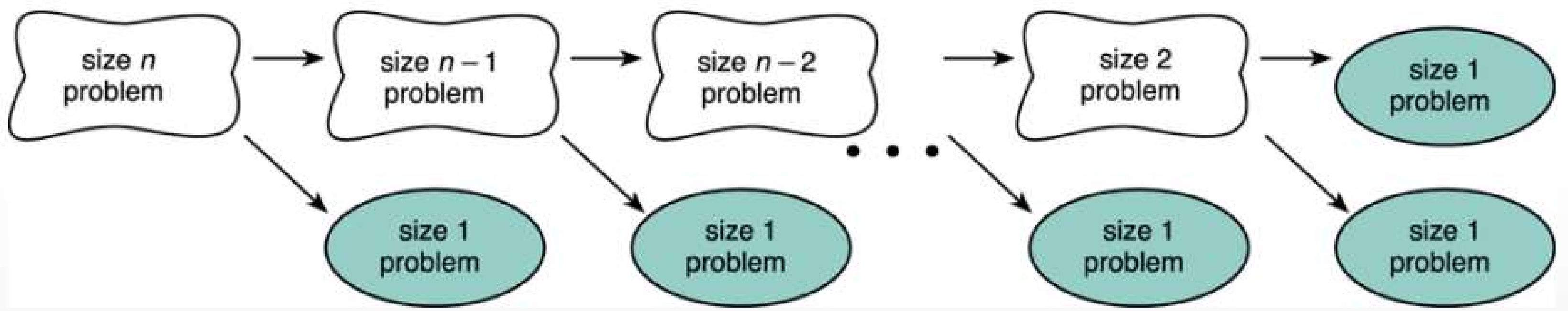
```
void f1()
{
    … .. …
    f1() ;
    … .. …
}

int main()
{
    … .. …
    f1();
    … .. …
}
```

recursive call



Splitting a problem into smaller problems

- Assume that the problem of size 1 can be solved easily (i.e., the simple case).
- We can recursively split the problem into a problem of size 1 and another problem of size n-1



Identifying Key Components of a Recursive Approach

1) Base case

- **A condition that determines when the recursion stops**
- prevent infinite recursion.
- Provide a direct solution when the problem is small and directly solvable.

2) Recursive Formula or Recursive Case

- **The part of the function that calls itself.**
- The problem divided into smaller, similar subproblems
- make one or more recursive calls to solve these subproblems.
- Results of subproblems are combined to solve the main problem

Recursive problem

```
void message()
{
    cout << " This is a recursive function . \n";
    message ();
}
```

Infinite loop

- The function is like an infinite loop because there is no code to stop it from repeating.
- Like a loop, a recursive function must have stop condition to control the number of times it repeats.

Recursive function

- A simplified pseudocode for a recursive approach

```
function recursiveFunction(input):
    // Base case: Return a specific value
    if base_case_condition:
        return base_case_result
    else:
        // Recursive case: Make a recursive call with modified input
        modified_input = modify(input)
        return recursiveFunction(modified_input)
```

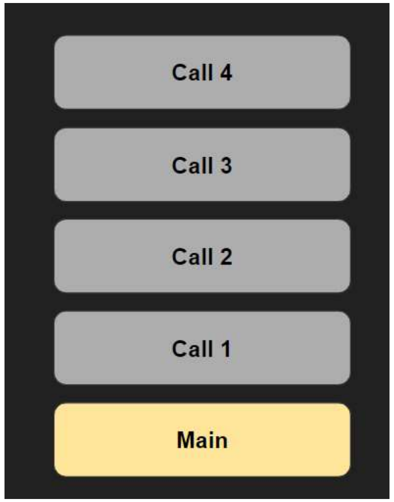
Base case

Recursive call

The Role of the Stack in Recursive Functions

The stack is essential for **pushing** and **popping** function calls, maintaining the order of recursive function calls and their respective state. This allows the program to return to the correct context when each call completes.

- Recursive functions rely on a **stack** to manage function calls.
- Each function call is **pushed** onto the stack, forming a call stack.
- The call stack keeps track of the state of each function call.
- As each function call returns, it is **popped** from the stack.
- The stack operates on the Last-In, First-Out (LIFO) principle.



Cont. calculate the sum of numbers from 1 to n

- **Top-down "recursive" approach:**

Write a pseudocode to calculate the sum of numbers from 1 to n using **recursive approach**

```
function recursiveSum(n):
    // Base case: When n is 1, return 1
    if n == 1:
        return 1
    // Recursive case
    else:
        return n + recursiveSum(n - 1)
```

Exercise: Summation Challenge: Iterative vs. Recursive

- Write two functions to find the sum of integers from *1 to n* using both an iterative and a recursive approach in a programming language of your choice (Choose any programming language you are comfortable with). Compare the execution time and behavior when n is increased.

Instructions:

- Implement an iterative function to calculate the sum of integers from 1 to n.
- Implement a recursive function to calculate the sum of integers from 1 to n.
- Test both functions with increasing values of n (e.g., 10, 100, 1000, 10000, 100000, 1000000 ….).
- Measure and compare the execution time for each approach using timing libraries or built-in functions.
- Note the execution time differences and behavior.
- Try to use the recursive method with a very large value of n (e.g., n = 100000) and **observe the stack overflow issue.**

Exercise: Summation Challenge: Iterative vs. Recursive

Discuss the differences between the iterative and recursive methods in terms of execution time and any issues encountered with the recursive method for large n values.

Stack Overflow

Search

Avoiding Stack Overflow Errors in Recursion

Challenge: When using recursion to solve problems with a large problem size (resulting in a substantial number of function calls), how can you avoid stack overflow errors?

Search: Explore and research strategies to prevent stack overflow errors in recursive solutions.

Choosing the right approach

- Some problems can be solved using both approaches but that one may be more suitable than the other.
- Discuss trade-offs between the two approaches:
  - Recursive methods may be more intuitive but can have higher memory overhead.
  - Iterative methods are often more efficient but may require more code.

Mathematical Thinking

- Consider the problem of finding the sum of integer numbers from 1 to n. You have previously explored iterative and recursive approaches in class. Now, let's think mathematically and explore if there's an even more efficient solution based on a mathematical perspective..
  - Recall the formula for the sum of the first n natural numbers. $S_n = \frac{n \cdot (n+1)}{2}.$
  - Use this formula to calculate the sum more efficiently, avoiding the need for iteration or recursion?
  - What about the time efficiency?

Example

- Let $f(x) = f(x-1) + 3$, $f(0) = 4$ find $f(7)$

$f(7) = f(7-1)+3 \rightarrow f(7)=f(6)+3$

$f(6) = f(6-1)+3 \rightarrow f(6)=f(5)+3$

$f(5) = f(5-1)+3 \rightarrow f(5)=f(4)+3$

$f(4) = f(4-1)+3 \rightarrow f(4)=f(3)+3$

$f(3) = f(3-1)+3 \rightarrow f(3)=f(2)+3$

$f(2) = f(2-1)+3 \rightarrow f(2)=f(1)+3$

$f(1) = f(1-1)+3 \rightarrow f(1)=f(0)+3$

$f(7)=22+2=25$

$f(6)=19+3=22$

$f(5)=16+3=19$

$f(4)=13+3=16$

$f(3)=10+3=13$

$f(2)=7+3=10$

$f(1)=4+3=7$

$f(0)=4$

Base case

Example

- Let f(x)= f(x -1) + 3 , $\boxed{\text{f(0)= 4}}$

Recursive call

Base case

```
function f(x):
    if x is 0:
        return 4  // Base case: f(0) is defined as 4
    else:
        return f(x - 1) + 3  // Recursive case: f(x) = f(x - 1) + 3
```

Recursive form

Recursive function terminates when a base case is met.

Tracing recursive function

- **Let $f(x) = f(x-1) + 3$, $\boxed{f(0) = 4}$ find $f(7)$**

25
f (7)
22
f (6) + 3
19
f(5) + 3
16
f(4) + 3
13
f(3) + 3
10
f(2) + 3
7
f(1) + 3
4
~~f(0)~~ + 3

	stack
return 4	**f(0)**
return 3+f(0)=7	f(1)
return 3+f(1) = 10	f(2)
return 3+f(2) = 13	f(3)
return 3+f(3) = 16	f(4)
return 3+f(4) = 19	f(5)
return 3+f(5) = 22	f(6)
Return 3+f(6) = 25	f(7)

base case is met
Terminate recursion

Factorial function using recursive

- n! = n * n-1 * ………. 3 * 2 * 1 if n > 0
  = 1 if n = 0

- We can write n! as follows: n! = n * (n-1)!

So we can use recursion to define the factorial of a number:

fact (n) = n * fact (n-1) if n > 0

1 if n = 0 [base case]

```
function factorial(n):
    if n is 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Base case `0! = 1`

Recursive call `n! = n * (n - 1)!`

Exercise

- Trace the factorial function fact (4)   [show your work]

Interactive tools and simulations

some websites that provide interactive tools and simulations to help students understand recursion and the use of the call stack:

1. **Pythontutor.com** : allows you to visualize the execution of Python code, including recursion, step by step. It provides a visual representation of the call stack.

- Website: http://pythontutor.com

2. **Visualgo.net:** offers visualizations for various data structures and algorithms, including recursion. It allows you to see how the call stack works for different programming languages.

- Website: https://visualgo.net/en/recursion