RECURSION

Geeks For Geeks Explanation: https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/

What is Recursion?
Process in which a function calls itself directly or indirectly, the corresponding function is called a recursive function.
Example Problems:

- Towers of Hanoi (TOH)
- Inorder/Preorder/Postorder Tree Traversals
- DFS of Graph

A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems.
A certain case should be provided in order to terminate a recursion process!

Need of Recursion
Recursion is one of the best solutions for a task that can be defined with a similar subtask (i.e. the Factorial of a number)

Recursion Properties

- Performing the same operations multiple times with different inputs
- In every step, we try smaller inputs to make a smaller problem
- A **base condition** is needed to stop the recursion

Mathematical Interpretation
Recursive adding – Instead of f(n) = 1 + 2 + 3 + … + n
f(n) = 1 n= 1
f(n) = n + f(n-1) n>1

- f() is being called inside of itself so it is recursive

Recursive Functions in Memory
More memory is used because a recursive function adds to the stack with each recursive call and keeps the values there until the call is finished.

- Uses LIFO structure like the stack data structure

Base Condition

The solution to the base case should be provided and the solution to the bigger problem should be expressed in terms of smaller problems

Example:

```
int fact(int n) {
  if (n <=1) //base case
    return 1;
  else
     return n*fact(n-1);
}
```

The base case is n <= 1, and the solution is 1. The larger value of a number can be solved by converting to a smaller one until the base case is reached.

Stack Overflow

If the base case is undefined or never reached, stack overflow may occur.

- wrong base case can cause this

Direct vs Indirect Recursion

Direct Recursive: Function that calls itself

Indirect Recursive: Function that calls a different function that calls the original function (Function A calls Function B and vv)

Tailed vs Non-Tailed Recursion

Tail Recursive: Recursive call is the last to be executed by the function

Memory Allocation to Different Function Calls

A function called from main(), the memory is allocated to it on the stack.

Recursion vs Iteration

Recursion	Iteration
Terminates when the base case becomes true	Terminates when the condition becomes false
Used with functions	Used with loops
Every recursive call needs extra space in the stack memory	Every iteration does not require any extra space
Smaller code size	Larger code size

Summary

- Two case types: recursive case & base case
- Base case is used to determine when the recursive function terminates (when base case is true)
- Each recursive call makes a copy of that method in the stack memory
- Infinite recursion may lead to running out of stack memory
- Examples
    - Merge Sort
    - Quick Sort
    - Towers of Hanoi
    - Fibonacci Series
    - Factorial Problem

Khan Academy Explanation: https://www.khanacademy.org/computing/computer-science/algorithms/recursive-algorithms/a/recursion

Recursion is like Russian Nesting Dolls

- The large doll has many smaller dolls within it, until it reaches one so small that it can not contain another

Factorial Function

The factorial of n (n!) is the product of the integers 1 through n. (5! = 5 * 4 * 3 * 2 * 1 = 120)

- useful for trying to count how many different orders there are for things or how many different ways there are to combine things
- useful to count how many ways you can choose things from a collection of things

Factorial function is defined for all positive integers, along with 0. 0! = 1
For n! if n is positive it is 1 * 2 … (n-1) * n and if n=0 it is 1

Properties of Recursive Algorithms

To solve a problem, solve a subproblem that is a smaller instance of the same problem, and then use the solution to that smaller instance to solve the original problem

- subproblems must arrive at a base case

Two Rules

- Each recursive call should be on a smaller instance of the same problem (subproblem)
- The recursive calls must eventually reach a base case, solved without further recursion

Using Recursion to Determine if a Word is a Palindrome

Base Case: Any string with exactly zero letters or one letter is a palindrome
Recursive Case: If a string contains two or more letters, if the first and last letters differ, then the string is not a palindrome. Otherwise, strip off the first and last letters

Pseudocode:

- If the string is made of no letters or just one letter, then it is a palindrome
- If the string has two or more letters, compare the first and last letters of the string
- If the first and last letters differ, then return false
- If the first and last letters are the same, strip them from the string and determine whether the string that remains is a palindrome. Take the answer for the smaller string and use it as the answer for the original string

Computing powers of a number

Psuedocode:

- When n equals 0, x^n = 1 (base case)
- Otherwise, if n is positive and even, recursively compute y = x^n/2 and then x^n = y*y
- If n is positive and odd recursively compute x^n-1 so that the exponent is either 0 or is positive and even. x^n = x^n-1 * x
- If n is negative recursively compute x^-n so that the exponent becomes positive. x^n = 1/x^-n

Improving efficiency of recursive functions

Recursive functions can become inefficient in terms of time and space.

Memoization

A form of caching, remembers the result of a function call with particular inputs in a lookup table, and returns that result when the function is called again with the same inputs

Pseudocode for factorial function:

- If n is 0, return 1
- Otherwise if n is in the memo, return the memo's value for n
- Otherwise,
    - Calculate (n-1)! * n
    - Store result in memo
    - Return result

Bottom-Up

The computer solves the sub-problems first and uses the partial results to arrive at the result

Pseudocode for Fibonacci number generation:

- If n is 0 or 1, return n
- Otherwise,
    - Create variable twoBehind to remember result of (n-2) and initialize to 0
    - Create variable oneBehind to remember result of (n-1) and initialize to 1
    - Create variable result to store final result and initialize to 0
    - Repeat (n-1) times:
        - update result to the sum of twoBehind & oneBehind
        - update twoBehind to store the current value of oneBehind
        - update oneBehind to store the current value of result
        - Return result

Dynamic Programming

A problem solving strategy used in mathematics and computer science

- can be used when a problem has optimal substructure and overlapping subproblems
- Optimal substructure – the optimal solution to the problem can be created from optimal solutions of its subproblems
- Overlapping subproblems – When a subproblem is solved multiple times

---

W3Schools Explanation: https://www.w3schools.com/java/java_recursion.asp

Java Recursion

The technique of making a function call itself

- provides a way to break complicated problems down into simple problems easier to solve

Example

```
public class Main {
  public static void main(String[] args) {
    int result = sum(10);
    System.out.println(result);
  }
  public static int sum(int k) {
    if (k > 0) {
      return k + sum(k-1);
    } else {
      return 0;
    }
  }
}
```

These are the steps:

- 10 + sum(9)
- 10 + (9 + (sum(8)))
- 10 + (9 + 8 + (sum(7))))
- ….
- 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
- 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0

Halting Condition

Infinite recursion is when the function never stops calling itself. Each recursive function should have a halting condition that causes the function to stop calling itself.

Purdue Explanation:
https://web.ics.purdue.edu/~cs180/Spring2004Web/lecture_notes/recursion-cs180.pdf

What is Recursion?

- Technique for solving problems
- Independent of programming language, can be used in almost any well known language

What is a tree?

- Recursion is easy to understand with the help of a tree

Developing a Recursive Solution

- Step 1: Identify a base case
  - solution is derived without recursion
  - simple solution in most cases
  - winding phase terminates when the base case is encountered
- Step 2: Identify the recursive, or general, case
  - solution is derived by invoking the same method recursively until a base case is encountered
  - continues the winding phase

Loops in Recursion

- Solutions with recursion are not necessarily free of loops
- A solution may use recursion and iteration so it may use loops and recursive calls

Difficulty in writing a recursive program

- Difficulty is identifying the base and recursive cases
- It is easy to identify the base case in most problems
  - suggestion is to focus on identifying the base and general cases before starting to write a recursive program

For 12/16 – https://www.khanacademy.org/computing/computer-science/algorithms/towers-of-hanoi/a/towers-of-hanoi

---

Recursion Videos

Neetcode Video 1: https://neetcode.io/courses/dsa-for-beginners/8
One-Branch
n! = n * (n-1) * (n-2) * … * 1
5! = 5 * 4 * 3 * 2 * 1 = 120
5! = 5 * 4!
n! = n * (n-1)!

- took a big problem and made it a smaller problem

- we can use recursion because we have a subproblem

5!
 4!
  3!
   2!
    1! (also base case)
     0! = 1 (base case)

```
int factorial (int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n-1);
}
```

- call function within function (recursive step)
- if the base case wasn't included, the recursive step would repeat forever

Time Complexity: We have n steps to compute, O(n)

- same as a while loop or for loop

Space Complexity: O(n) memory

- worse than a while or for loop

NeetCode Video 2: https://neetcode.io/courses/dsa-for-beginners/9

Two-Branch

Fibonacci Number: F(n) = F(n-1) + F(n-2)

Base Cases: F(0) = 0, F(1) = 1

F(2) = F(1) + F(0) = 1 + 0 = 1

F(3) = F(2) + F(1) = 1 + 1 = 2

```
int fib(int n) {
    if (n <= 1) {
      return n;
    }
    return fib(n-1) + fib(n-2);
}
```

F(5) has a height of 5

Udemy Master the Coding Interview Video 1: https://www.udemy.com/course/master-the-coding-interview-data-structures-algorithms/learn/lecture/12393986#content

https://www.udemy.com/course/master-the-coding-interview-data-structures-algorithms/learn/lecture/12393998#content

Introduction to Algorithms

Algorithms are steps in a process that we take to perform a desired action with computers

- allow us to use data structures to perform actions on data

Data Structures + Algorithms = Programs

Algorithms

- Sorting
- Dynamic Programming
- BFS + DFS (Searching)
- Recursion

Certain algorithms allow us to simplify our Big O complexities for time and space complexity

Recursion Introduction

Is not technically an algorithm

ls -R in command line R= recursive

- shows files inside of directories too

Recursion is when you define something in terms of itself

- function that refers to itself inside of the function

```
function inception() {
    inception();
}
```

Recursion is good for tasks that have repeated subtasks to do

- used in searching and sorting algorithms
    - traversing a tree

Udemy Video 2: https://www.udemy.com/course/master-the-coding-interview-data-structures-algorithms/learn/lecture/12394008#content

---

Hackerrank Video: https://www.youtube.com/watch?v=KEEKn7Me-ms

Algorithms: Recursion

Recursion is a construct and not really a data structure or algorithm

Example: How many .txt files are on my computer?

Base Case: a folder with no subfolders

Recursive Case: go through root directory and see how many txt files there are and see if there are any sub directories and what txt files are in those directories

Recursion: Taking a problem and breaking it down into subproblems

- Must have a base case (stopping point)
    - folder with no subfolders
- Memoization optimizes recursion

FreeCodeCamp Video: https://www.youtube.com/watch?v=vPEJSJMg4jY

Recursion is used in many algorithms

- can be used in problems such as the example given: searching for a key in nested boxes

TRY TO RUN THROUGH A RECURSIVE FUNCTION WITH PEN AND PAPER

Example: Find a key in nested boxes

Iterative Psuedocode solution:

```
look_for_key(box)
 pile = box.newPile
 while pile is not empty
   box = pile.grab_box
   for item in box:
    if item is box
     pile.append(item)
    else item is a key
     print “found key”
```

Recursive Psuedocode solution:

```
look_for_key(box)
 for item in box
  if item = box
   look_for_key(item)
  else
   print “found key”
```

Base Case + Recursive Case

- avoid going into an infinite loop

Recursive Case = when a function calls itself

Base Case = when a function no longer calls itself