CDS2003: Data Structures and Object-Oriented Programming

Lecture: Divide-and-Conquer and Recursion

Review

- Course description
- Ethical implications of data science and artificial intelligence
  - Seeking to enhance the value of data science for society
  - Avoiding harm
  - Applying and maintaining professional competence
  - Seeking to preserve or increase trustworthiness
  - Maintaining accountability and oversight
- Recommended readings
- Online resources
  - W3Schools (https://www.w3schools.com/), thanks to Kenneth

- LeetCode (Hands-on practice https://leetcode.com/playground)

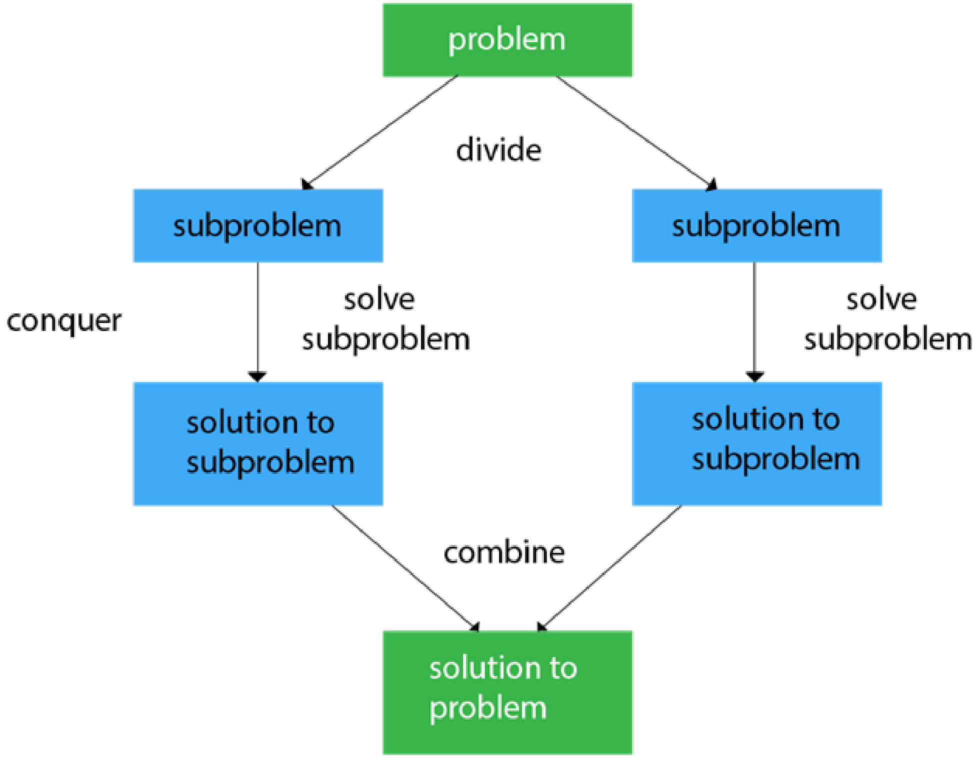
We shall discuss…

- What is divide-and-conquer
- What is recursion
- Recursion versus iteration
- Advantages or disadvantages of recursion
- Examples of recursion

Divide-and-Conquer

Divide-and-conquer

- A mechanism of solving a large problem
  - Solving a large problem by recursively breaking it down into smaller (more manageable) subproblems until they can be solved directly



[1] Divide and Conquer Introduction https://www.javatpoint.com/divide-and-conquer-introduction

Tree steps of divide-and-conquer

1. Divide
   - Breaking down the original problem into smaller independent subproblems
2. Conquer
   - Solving each of the smaller subproblems individually
   - Solving the independent subproblems concurrently in a multi-processor machine
3. Merge/Combine
   - Putting together the solutions of the subproblems to get the final solution to the original problem

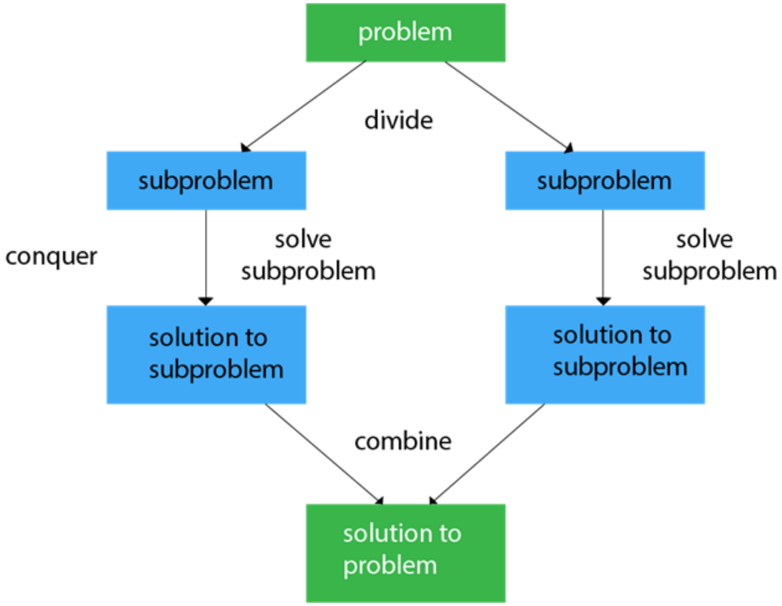
- Example: Find the maximum value in an unsorted array.
  - Divide: Given the array [4, 6, 2, 8, 3, 1], we can divide it into [4, 6, 2] and [8, 3, 1].
  - Conquer: For the first array, the maximum is 6. For the second, the maximum is 8.
  - Combine: Compare the two maximums obtained from the halves. In this case, the maximum between 6 and 8 is 8.

Pros and cons of divide-and-conquer

- Advantages
  - Solving difficult problems conceptually
  - Helping discover efficient algorithms
  - Parallelism in multi-processor machines
  - Efficient using cache for smaller problems instead of main memory

- Disadvantages
  - Additional resources for dividing and combining
  - Difficulty of debugging and implementation

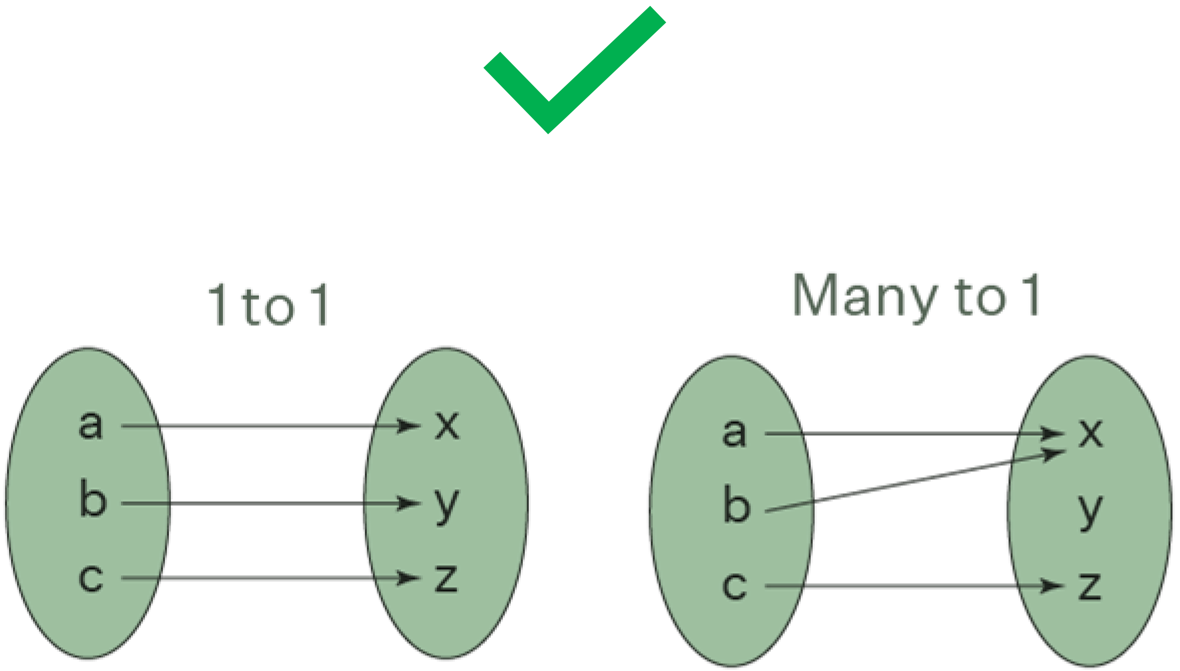
- Branch-and-bound?

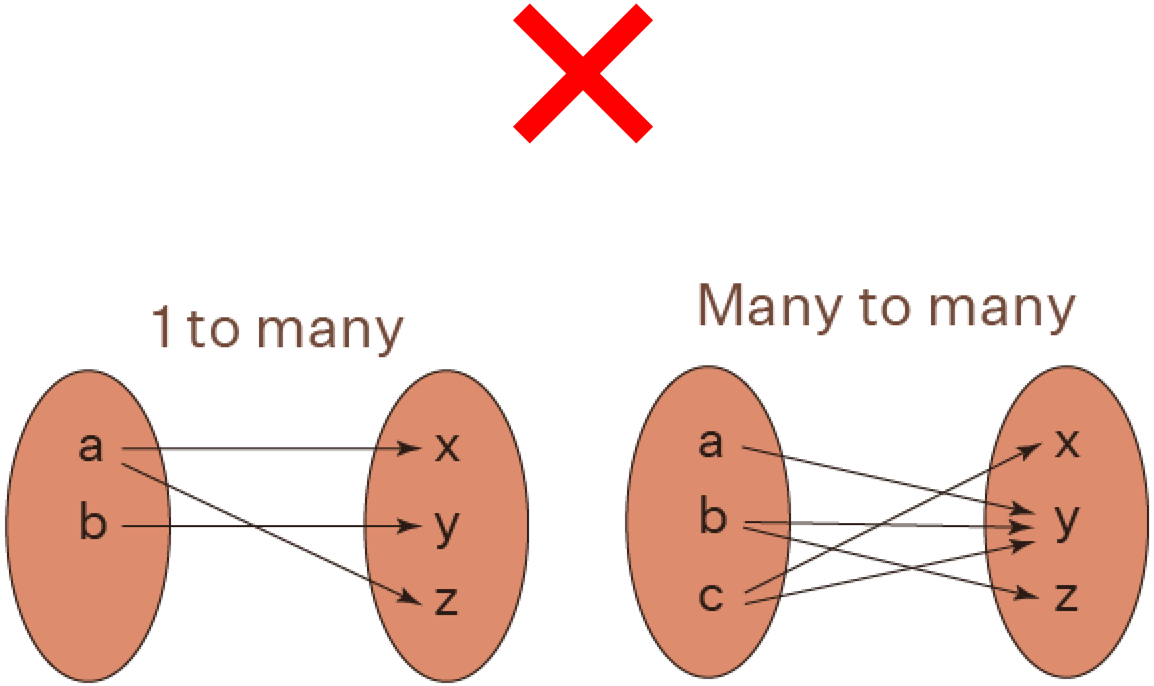


[1] Divide and Conquer Introduction https://www.javatpoint.com/divide-and-conquer-introduction

Function

- A function is a relation between a set of inputs and a set of possible outputs where each input is related to exactly one output.





[1] Divide and Conquer Introduction https://www.javatpoint.com/divide-and-conquer-introduction

Optimization problem

- An optimization problem is a mathematical problem that seeks to find the best solution from a set of feasible solutions, according to a specific criterion.

- Components of an optimization problem:
  - Decision variables: The variables that can be controlled or adjusted to achieve the desired outcome, e.g., $\boldsymbol{x} = [x_1\ x_2]'$. A solution is a specific set of values assigned to the decision variables, e.g., $\boldsymbol{x} = [3\ 2]$.
  - Constraints: Limitations or restrictions on the decision variables to define the feasible region, e.g., $3x_1 + 2x_2 \leq 2$ & $x_2 \geq -5$. A feasible solution is a solution that satisfies the constraints, e.g., $\boldsymbol{x} = [0.5\ 0.1]$. All feasible solutions constitute the feasible region.
  - An objective function: The function that needs to be maximized or minimized. For example, maximize $f(\boldsymbol{x}) = 2x_1 + x_2$. Among all feasible solutions, the optimal solution is the one that maximizes or minimizes the objective function.

[1] Divide and Conquer Introduction https://www.javatpoint.com/divide-and-conquer-introduction

Branch-and-bound

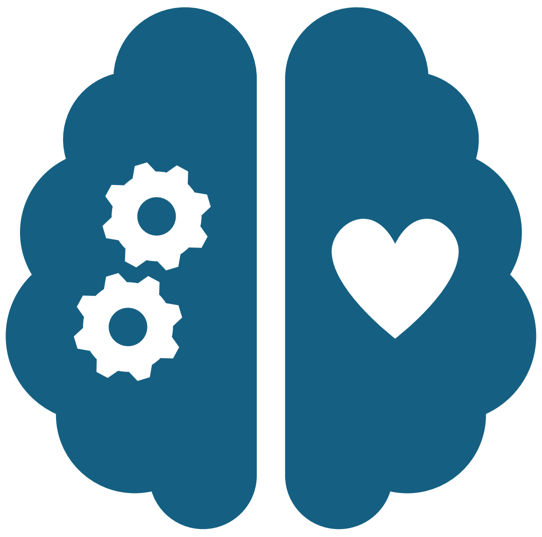
- Branch-and-bound uses a “divide and conquer” approach to solve optimization problems.

- The key concepts
  - Branching: Divide the problem into smaller subproblems with each subproblem representing a part of the solution space.
  - Bounding: For each subproblem, estimate the bounds of the best possible solution that can be obtained.
  - Pruning: If the bound of a subproblem indicates that it cannot yield a better solution than the best one found so far, discard that subproblem.

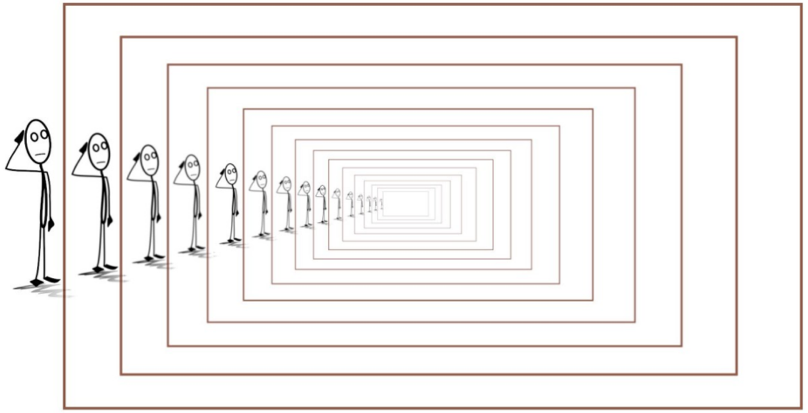
- Keep you cool! We will revisit the concept later.

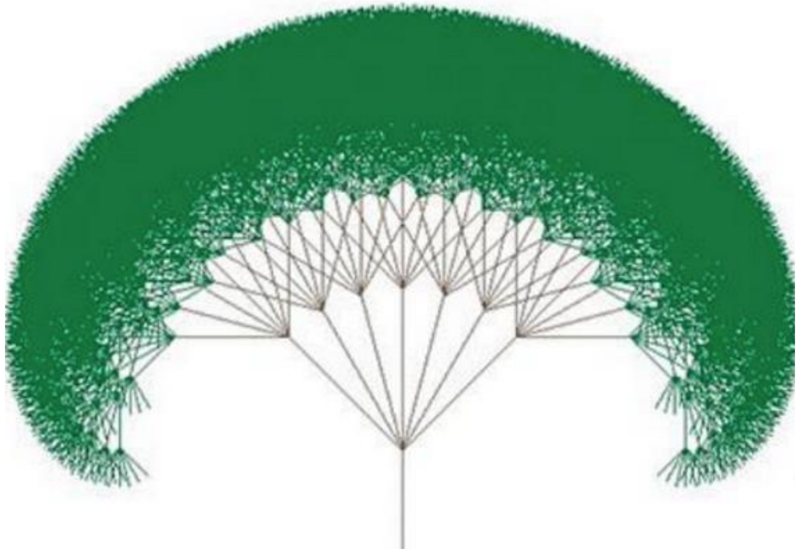
[1] Divide and Conquer Introduction https://www.javatpoint.com/divide-and-conquer-introduction

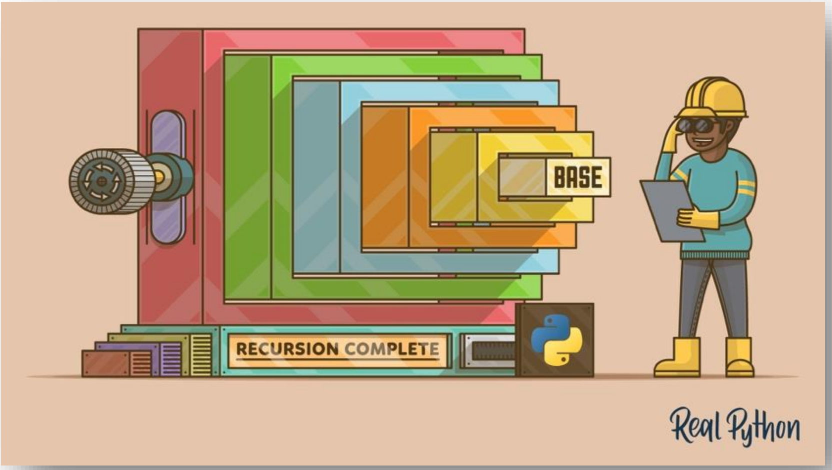
Recursion



Illustrative figures of recursion







[1] https://www.linkedin.com/pulse/recursion-explained-understand-you-must-first-ignacio-chitnisky/
[2] https://medium.com/enjoy-algorithm/recursion-explained-how-recursion-works-in-programming-b22113006fe3
[3] https://realpython.com/python-thinking-recursively/

What is recursion

- A recursive definition is one in which the defined term appears in the definition itself.
  - Your ancestors = (your parents) + (your parents' ancestors)
- Recursion is the process of defining something (a problem or a solution to a problem) in terms of (a simpler version of) itself.
- A function is recursive if it calls itself, directly or indirectly.
- Why is recursion needed?
  - One of the best solution for a task that can be defined with its similar substask.
  - Reducing the length of our code and making it easier to read and write

Examples of recursion

- Task: Find your home.
  - Checking whether you are at home
  - Stopping moving when you are at home
  - Finding a route to home and taking one step toward home when you are not at home

Examples of recursion

- Task: Count down a nonnegative number to zero

$$n, n-1, n-2, \ldots, 0.$$

  - Counting down 5 to zero
  - Saying 5 and reducing the problem to counting down 4 to 0
  - Saying 4 and reducing the problem to counting down 3 to 0
  - Saying 3 and reducing the problem to counting down 2 to 0
  - Saying 2 and reducing the problem to counting down 1 to 0
  - Saying 1 and reducing the problem to counting down 1 to 0 (Just Saying 0!!!)

Examples of recursion

- Task: Count down a nonnegative number to zero

$$n, n-1, n-2, \dots, 0$$

- Base case: $n = 0$
- Recursive case: $n > 0$

- Countdown(5)
- Print(5) and Countdown(4)
- Print(4) and Countdown(3)
- Print(3) and Countdown(2)
- Print(2) and Countdown(1)
- Print(1) and Countdown(0)

```
# Count down to zero
def countdown(n):
    print(n)
    if n == 0: # Terminate condition
        return
    else:      # Recursive call
       countdown(n-1)

countdown(5)
```

Output:

```
5
4
3
2
1
```

Two parts in a recursive function

- Termination condition:
  - A recursive function always contains one or more terminating condition.
  - A condition in which the recursive function is processing a simple case (called base case) and will not call itself.
  - Each recursive call makes a new copy of that function in the stack memory.
  - Without termination condition, the recursive function may run forever and will finally run out of the stack memory.
- Body:
  - The main logic of the recursive function contained in the body of the function. It also contains the recursion expansion statement that in turn calls the method itself.

- Task: Find your home.
  - Checking whether you are at home
  - Stopping moving when you are at home
  - Finding a route to home and taking one step toward home when your are not at home

```
# Count down to zero
def countdown(n):
   print(n)
   if n == 0: # Terminate condition
      return
   else:     # Recursive call
     countdown(n-1)
```

About the base case in recursion

- We already know the answer to the base case or it is easy to find the answer to the base case.
- The function stops calling itself when the base case is reached.
- Each successive recursive call to the function should bring it closer to the base case.

```python
# Count down to zero
def countdown(n):
    print(n)
    if n == 0: # Terminate condition
        return
    else:      # Recursive call
       countdown(n-1)
```

Examples of recursion

- Problem: Calculate the factorial of a nonnegative integer n, where

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 \text{ and } 0! = 1$$

- Top-down approach for design:
- factorial_Recur(4)
- 4 × factorial_Recur(3)
- factorial_Recur(3)
- 3 × factorial_Recur(2)
- factorial_Recur(2)
- 2 × factorial_Recur(1)
- factorial_Recur(1)
- 1 × factorial_Recur(0)
- 1 (base case: $n = 0$)

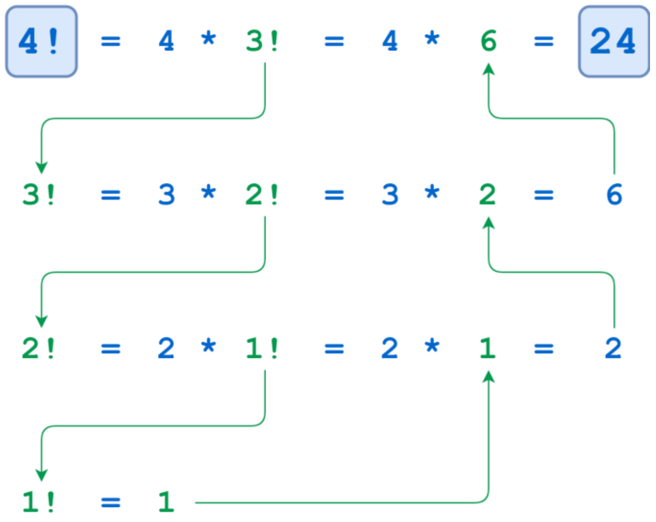
```python
# factorial with Recursion
def factorial_Recur(n):
    if n == 0:
        return 1
    return n * factorial_Recur(n-1)
```

Examples of recursion

- Problem: Calculate the factorial of an integer n, where

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 \text{ and } 0! = 1$$

- factorial_Recur(4)
- 4 × factorial_Recur(3) $= 4 \times 6 = 24$
- factorial_Recur(3)
- 3 × factorial_Recur(2) $= 3 \times 2 = 6$
- factorial_Recur(2)
- 2 × factorial_Recur(1) $= 2 \times 1 = 2$
- factorial_Recur(1)
- 1 × factorial_Recur(0) $= 1 \times 1 = 1$
- 1 (base case: $n = 0$)



[1] Recursion in Python: An Introduction https://realpython.com/python-recursion/

Exercise (5 mins)

- Please use recursion to define a function to calculate the sum of first n natural numbers

$$1 + 2 + \cdots + n$$

- Starting from `def sum_n_Recur(n):`

- Testing at `sum_n_Recur(4)`

- https://leetcode.com/playground

Examples of recursion

- Example: Calculate the sum of first n natural numbers

$$1 + 2 + \cdots + n$$

- sum_n_Recur(4)
- 4 + sum_n_Recur(3)
-         sum_n_Recur(3)
-         3 + sum_n_Recur(2)
-                 sum_n_Recur(2)
-                 2 + sum_n_Recur(1)
-                         1 (base case: $n = 1$)

```python
# sum with Recursion
def sum_n_Recur(n):
    if n <= 1:
        return n
    return n + sum_n_Recur(n-1)
```

Examples of recursion

- Problem: In a Fibonacci sequence (starting from 0 and 1), each number is the sum of the two preceding ones $F_n = F_{n-1} + F_{n-2}$.

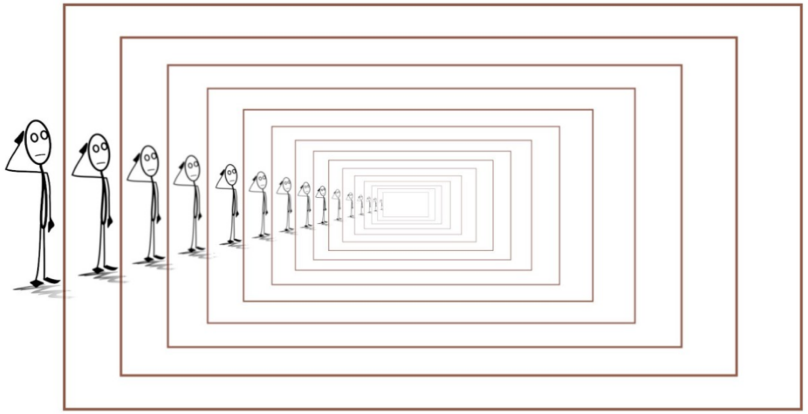
n	0	1	2	3	4	5	6	7	8	9	...
F_n	0	1	1	2	3	5	8	13	21	34	...

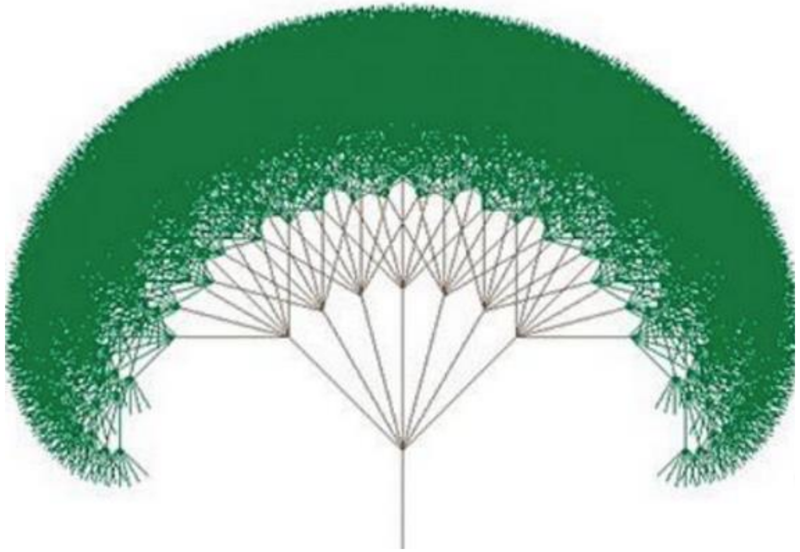
- Base case: $n < 2$

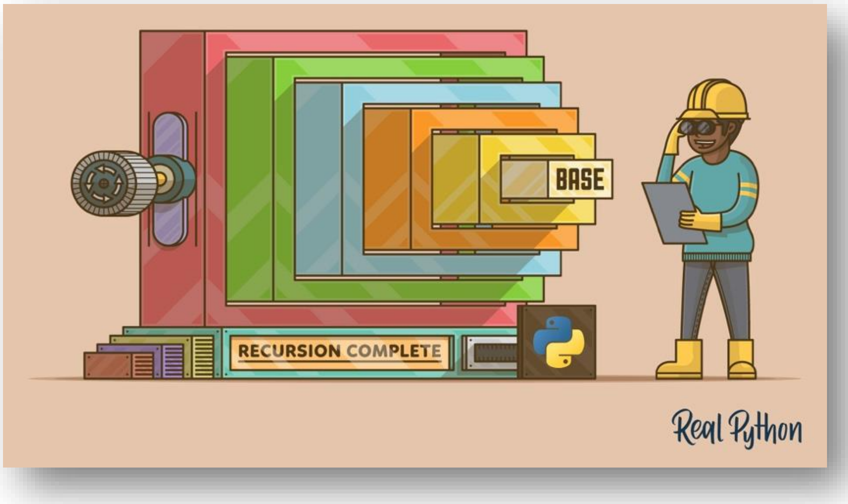
- Recursive case: $n > 2$
  - Breaking the problem
  - Calling the function recursively

```python
# Fibonacci number with recursion
def get_fn_Recur(n):
    if n < 2:
        fn = n
    else:
        fn = get_fn_Recur(n-1) + get_fn_Recur(n-2)
    return fn
```

Illustrative figures of recursion







[1] https://www.linkedin.com/pulse/recursion-explained-understand-you-must-first-ignacio-chitnisky/
[2] https://medium.com/enjoy-algorithm/recursion-explained-how-recursion-works-in-programming-b22113006fe3
[3] https://realpython.com/python-thinking-recursively/

Four steps for implementing recursion in a function

- Step 1 -- Defining a base case:
  - Identifying the simplest case for which the solution is known or trivial, relating to the stopping condition for the recursion, as it prevents the function from infinitely calling itself.
- Step 2 -- Defining a recursive case:
  - Defining the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself, and call the function recursively to solve each subproblem.
- Step 3 -- Ensuring the recursion terminates:
  - Making sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.
- Step 4 -- Combining the solutions:
  - Combining the solutions of the subproblems to solve the original problem.

Recursion versus iteration

- Recursion and iteration are key techniques in algorithm design.
- A recursive function is one that calls itself to repeat some code block.
  - A divide-and-conquer approach: breaking the problem into sub-problems
- An iterative function is one that loops to repeat some code block.
  - Sequential execution
- Recursion problems can generally be solved by iteration (using loops).

Recursion versus iteration

- Problem: Count down a nonnegative number to zero

$$n, n-1, n-2, \dots, 0$$

```python
# Count down to zero with Recursion
def countdown(n):
    print(n)
    if n == 0: # Terminate condition
        return
    else:     # Recursive call
       countdown(n-1)
```

```python
# Count down to zero with Iteration
def countdown(n):
    while n >= 0:
        print(n)
        n -= 1
```

Recursion versus iteration

- Problem: Calculating the sum of first n natural numbers

$$1 + 2 + \cdots + n$$

```python
# sum with Recursion
def sum_n_Recur(n):
    if n <= 1:
        return n
    return n + sum_n_Recur(n-1)
```

```python
# sum with Iteration
def sum_n_Iter(n):
    result = 0
    for i in range(n+1):
        result += i
    return result
```

Recursion versus iteration

- Problem: Calculating the factorial of an integer n, where

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 \text{ and } 0! = 1$$

```
# factorial with Recursion
def factorial_Recur(n):
    if n == 0:
        return 1
    return n * factorial_Recur(n-1)
```

```
# factorial with Iteration
def factorial_Iter(n):
    result = 1
    for i in range(1,n+1):
        result *= i
    return result
```

Recursion versus iteration

- Problem: In a Fibonacci sequence (starting from 0 and 1), each number is the sum of the two preceding ones $F_n = F_{n-1} + F_{n-2}$.

```
# Fibonacci number with recursion
def get_fn_Recur(n):
    if n < 2:
        fn = n
    else:
        fn = get_fn_Recur(n-1) + get_fn_Recur(n-2)
    return fn
```

```
# Fibonacci number with iteration
def get_fn_Iter(n):
    if n < 2:
        fn = n
    else:
        first = 0
        second = 1
    for _ in range(n-1):
        sum = first + second
        first = second
        second = sum
    fn = second
return fn
```

Recursion versus iteration

#	Recursion	Iteration
1	Terminates when the base case becomes true.	Terminates when the condition becomes false.
2	Used with functions	Used with loops
3	Every recursive call needs extra space.	Every iteration does not require any extra space.
4	Smaller code size	Larger code size
5	Divide-and-conquer	Sequential execution

When we use recursion

- Pros:
  - Breaking a complex task into simpler sub-problems
  - Making the code look clean and elegant
  - Generating sequence more easily than nested iteration

- Cons:
  - Resulting the logic that is hard to follow through sometimes
  - Taking up a lot of memory and time
  - Complicating the debug process

- Notice
  - The speed of a recursive program is slower because of stack overheads.
  - If the same task can be done using an iterative solution (using loops), it is often better to use an iterative solution in place of recursion to avoid stack overhead.

Individual assignment 02

Notice

- Mandatory
  - P01: Please use recursion to define and test a function to calculate the sum of a list of numbers. `def list_sum_Recur(num_list):`
  - P02: Please use recursion to define and test a function to find the greatest common division of two positive integers. `def gcd_Recur(a,b):`
  - P03: Please use recursion to define and test a function to calculate the harmonic series upto n terms. `def harmonic_sum_Recur(n):`
  - P04: Please use recursion to define and test a function to calculate the value of x to the power of n. `def power_Recur(x,n):`

- Optional
  - P05: Please use recursion to define and test a function to accept a decimal integer and display its binary equivalent. `def Dec2Binary_Recur(num):`
  - P06: Please use recursion to define and test a function to take in a string and returns a reversed copy of the string. `def reverse_Recur(a,b):`
  - P07: Please use recursion to define and test a function to check whether a number is Prime or not. `Def isPrime_Recur(a,b):`