

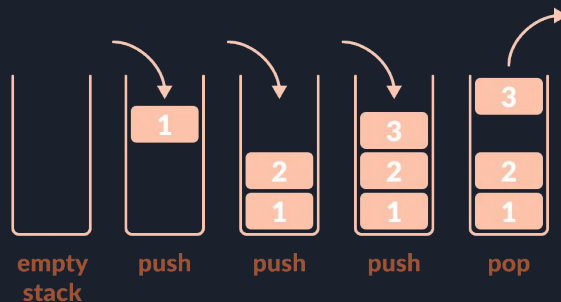
A decorative graphic on the left side of the slide consisting of two overlapping parallelograms. The front one is blue and the back one is light green. They are positioned diagonally, with the blue one partially covering the green one.

# Stacks and Recursion

Presented by Alexander Mak

# Stacks

- Last In, First Out (LIFO)
- Two main operations:
  - push (to add an item)
  - pop (to remove the last-added item).
- Used in various applications, including function call management and undo mechanisms.
- Provide a systematic way of managing data, helping in organizing and keeping track of information.
- Crucial in programming for managing function calls, tracking execution history, and storing temporary data.



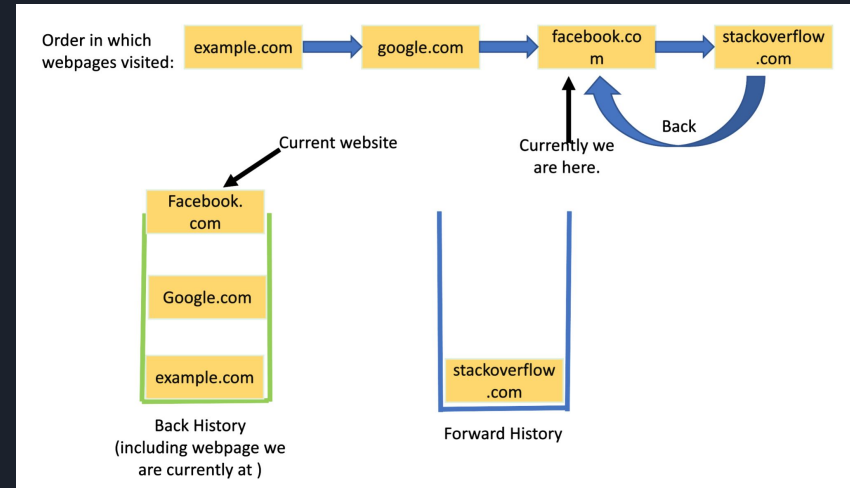
# Stack Example

Suppose you want to create a browser history management system where you can move both backwards and forwards in your browser history.

We can use two stacks to keep track of web pages in the order they were accessed.

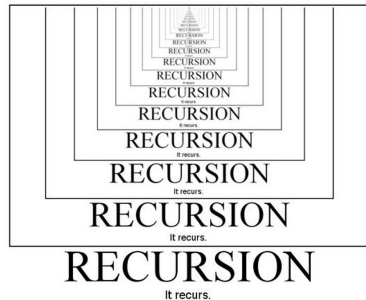
To navigate backwards, we would pop the current web page from our back history and push it into our forward history.

To navigate forwards, we would pop from our forward history and push it into our back history, making it our current page.



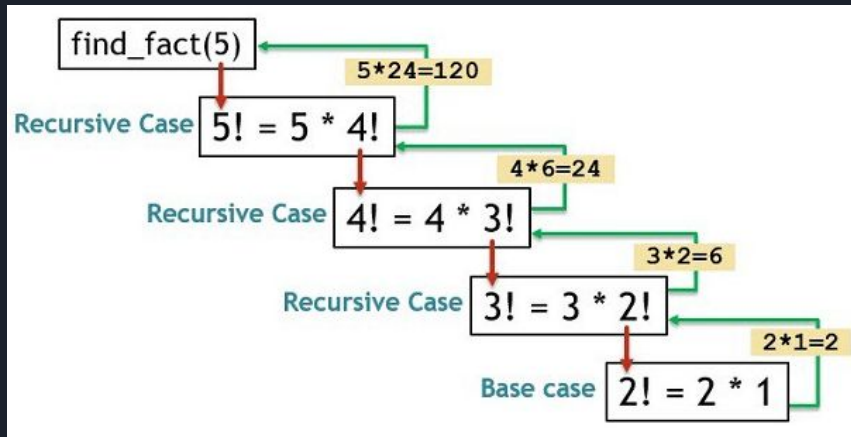
# Recursion

- Recursion is a concept where a function calls itself.
- Recursive solutions often involve **breaking down a problem into smaller, similar subproblems**.
- A powerful and elegant technique that can simplify code and solve certain types of problems more naturally.
- Recursion inherently creates a stack, specifically the call stack, which works very similarly to a stack created by an array.
- All recursion solutions can also be written iteratively, but this can sometimes overly complicate some problems.



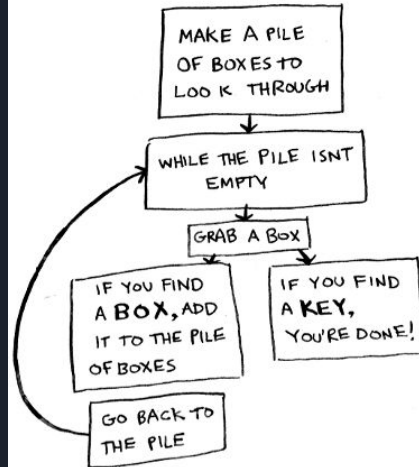
# Relationship between stacks and recursion

- Stacks play a vital role in managing the recursive function calls in a program.
- Each recursive call gets its own stack frame, allowing the program to keep track of multiple function calls and their respective local variables.

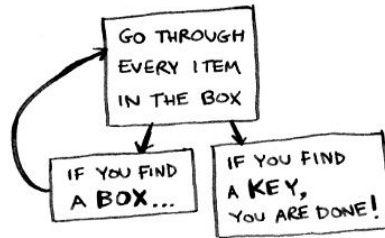


# Relationship between iterative and recursive approaches

## Iterative Approach



## Recursive Approach

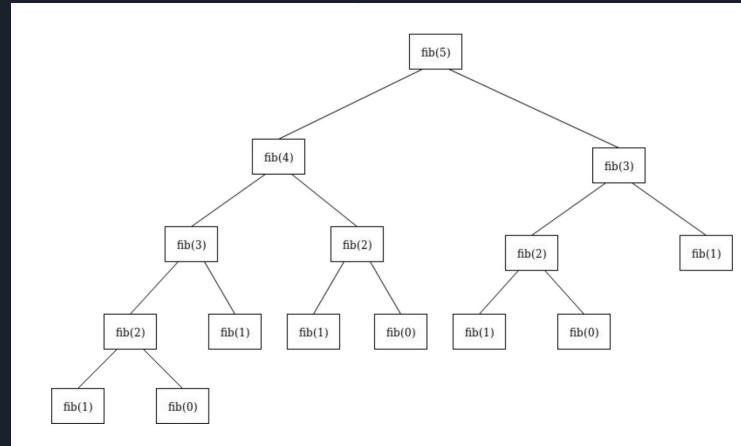


```
1 pile = [2, 8, 4, 1, 7, 5, 3, 9, 0, 6]
2 key = 7
3
4 def iterative(boxes, key):
5     i = 0
6
7     while i < len(boxes):
8         box = pile[i]
9         if box == key:
10            return True
11        i += 1
12    return False
13
14
15 def recursion(i):
16     box = pile[i]
17     if i == len(pile):
18         return False
19     if box == key:
20         return True
21
22     return recursion(i+1)
```

The code snippet shows two functions: `iterative` and `recursion`. The `iterative` function uses a `while` loop to iterate through the `pile` list. The `while` loop condition `i < len(boxes)` is highlighted with a red oval and labeled 'conditional'. The increment `i += 1` is highlighted with a purple oval and labeled 'next step'. The `recursion` function uses a recursive call. The base case `if i == len(pile): return False` is highlighted with a red oval and labeled 'base case'. The recursive call `return recursion(i+1)` is highlighted with a purple oval and labeled 'next step'.

# Runtime Complexity of Recursion

- Single-path recursive function usually run in  $O(n)$  where  $n$  is the depth of the stack
- Multi-path recursive functions can be analyzed with the following formula:
  - $(\text{\# of paths})^{(\text{depth of stack})}$
- The tree on the right represents a recursive Fibonacci function.
- Each node has two branching paths
- The depth of the stack is 5, which is our variable input.
- Thus, we can call this  $O(2^n)$  where  $n$  is our input number



# Questions?





# Let's practice!

[https://github.com/Dijkstra-LLC/dsa\\_live\\_pro/tree/alex-m/W02D02/classwork](https://github.com/Dijkstra-LLC/dsa_live_pro/tree/alex-m/W02D02/classwork)