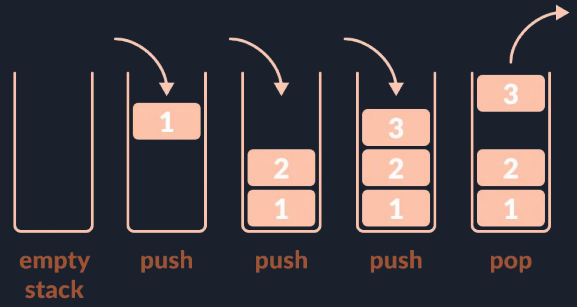
Stacks and Recursion

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.



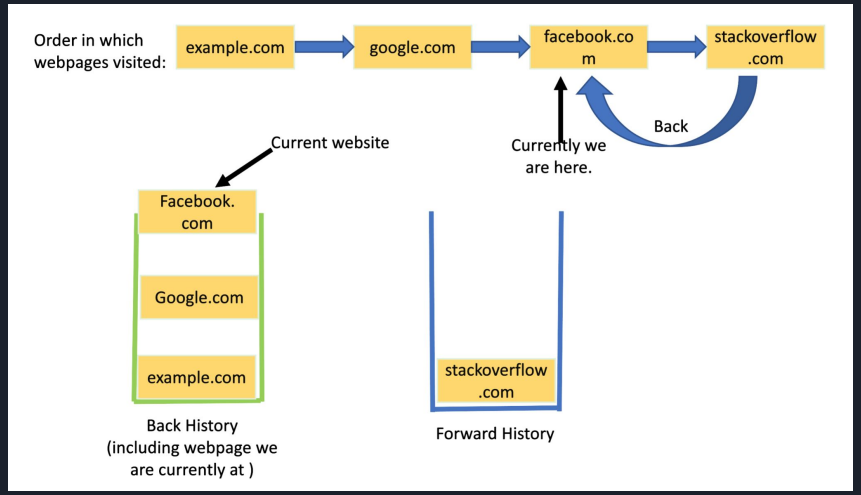
Stack Example

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.

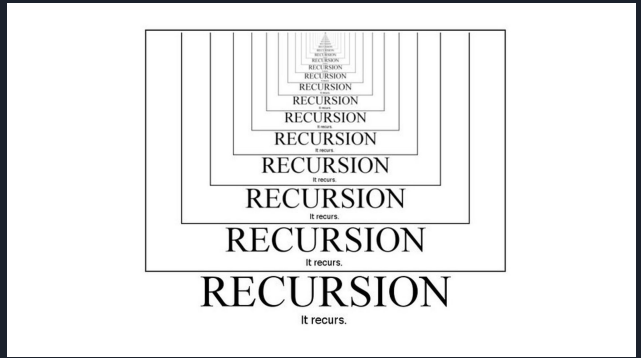
To navigate forwards, we would pop from our forward history and push it into our back history, making it our current page.



Presented by Alexander Mak

Recursion

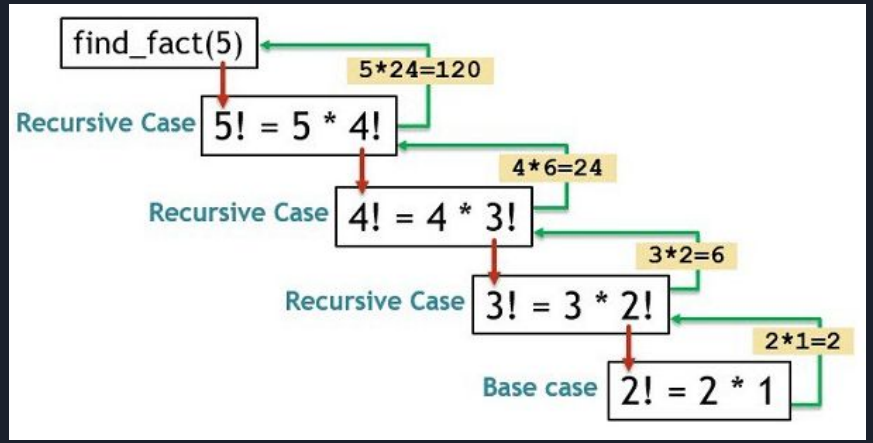
- Recursion is a concept where a function calls itself.
- Recursive solutions often involve breaking down a problem into smaller, similar subproblems.
- A powerful and elegant technique that can simplify code and solve certain types of problems more naturally.
- Recursion inherently creates a stack, 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.



Presented by Alexander Mak

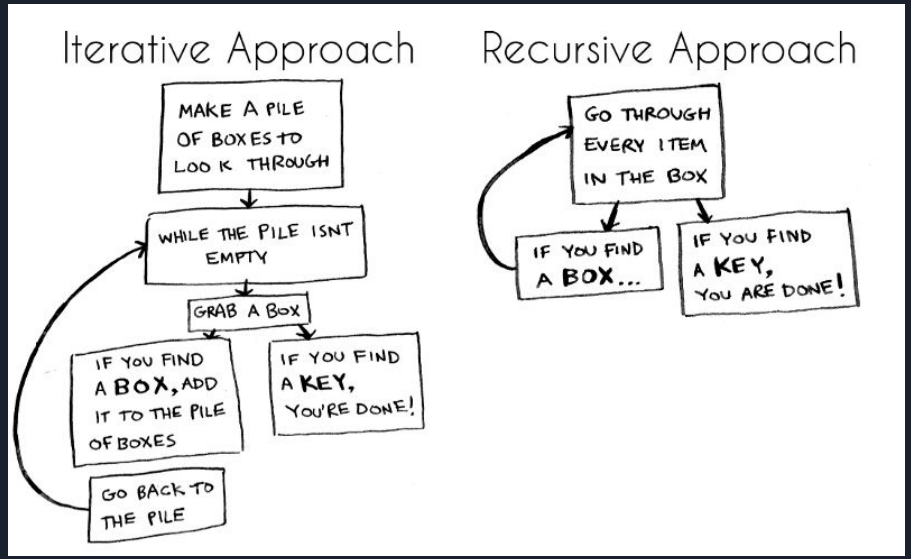
Relationship between stacks and recursion

- 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.



Presented by Alexander Mak

Relationship between iterative and recursive approaches



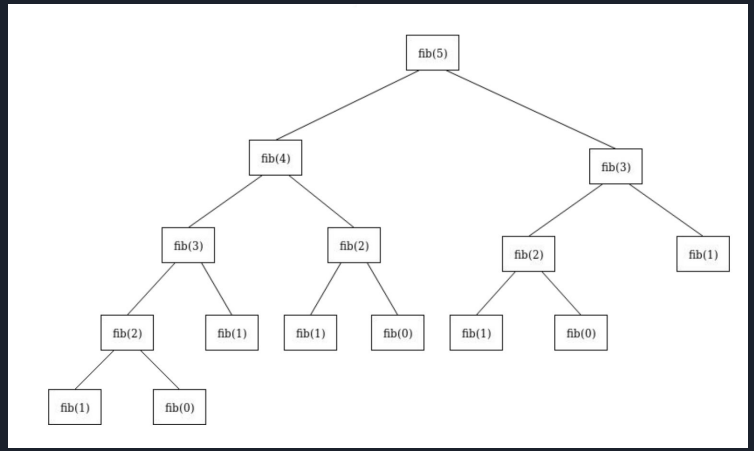

```python
pile = [2, 8, 4, 1, 7, 5, 3, 9, 0, 6]
key = 7

def iterative(boxes, key):
    i = 0

    while i < len(boxes):          # conditional
        box = pile[i]
        if box == key:
            return True
        i += 1                     # next step

    return False

def recursion(i):
    box = pile[i]
    if i == len(pile):             # base case
        return False
    if box == key:
        return True

    return recursion(i+1)          # next step
```

Presented by Alexander Mak

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:
  - (# 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



Presented by Alexander Mak

Questions?

Let’s practice!

https://github.com/Dijkstra-LLC/dsa_live_pro/tree/alex-m/W02D02/classwork