Lesson #4

September 3, 2019

Warm up #0

- <u>Description</u>
    - You are designing a new javascript interpreter to replace V8
- <u>Constraints</u>
    - It can only add two numbers at a time
    - It is a *stack virtual machine*, which means it does work on a stack (eg pop, push)
- <u>Task</u>
    - Implement function `op_add` that can sum numbers in a list
        - Given `arr = [1, 4, 5, 8]` and a stack object `Stack`, `op_add(arr, Stack)` returns 18
    - Don’t forget to insert the appropriate guard clauses

Warm up #1

- Description
  - You work at Cloudflare, a web performance company. Bandwidth is expensive so you want to minimize number of duplicate requests.
- Task
  - Implement `checkCache(url)` , where url is a string
  - Returns True if the user has visited the website before, otherwise False
  - Don’t forget about guard clauses!
- Constraints
  - These are web requests so we want fast lookups

Bonus question

- In a vanilla hashmap, we have constant lookup times (ie O(1)) but O(n) when there is a *hash collision*
- How do avoid O(n)?
  - In engineering, everything is a tradeoff. In this case, we are willing to live with “false positives”

Warm up #3

- Description
  - You work at Travis CI, a continuous integration service
- Task
  - Implement `findBug(listCommits)` , which returns the guilty commit hash `(eg 90fb64)`
  - Input is a list of commits `listCommits`
    - Each element is an unique commit hash.
  - You can invoke a helper function `checkValid(commit_hash)` , which returns True if the commit passed tests and False otherwise
- Constraints
  - `listCommits` is ordered by time, from oldest to newest. Assuming a history of commits [A, B, C, D, E], if commit C contains a bug, then D and E will also fail (eg `checkValid(C)`, `checkValid(D), and checkValid(E)` will return False)
  - We want fast searches (that is, not linear time)

In the previous episode...

- Data structures and algorithms are lego blocks
    - Combined linked lists, stacks, etc for recursion
    - Combined hash functions with arrays to form hashmaps
- Discussed sorting upfront to simplify search
    - Sorting is O(nlogn) but returns a highly structured data structure, which makes our lives easier down the line
- Discussed why “log” appears in time complexity
    - We usually split things into two
- Dipped our toes into bubble sort and selection sort

Agenda for tonight

- Review recursion until your head hurts
  - To understand algorithms, you need to understand iteration; for iteration, you need recursion; for recursion, you need stacks and call frames; for stacks and call frames, you need pointers; and so on
    - This is a rabbit hole - you will understand why Stack Overflow is called Stack Overflow
  - We are not leaving this classroom until everyone gets recursion
- Then, for a nightcap, ease into merge sort, etc.

What is a Fibonacci sequence?

- It looks like this: `[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]`
- It is stateful, meaning it depends on its past states
  - F(3) = F(2) + F(1)

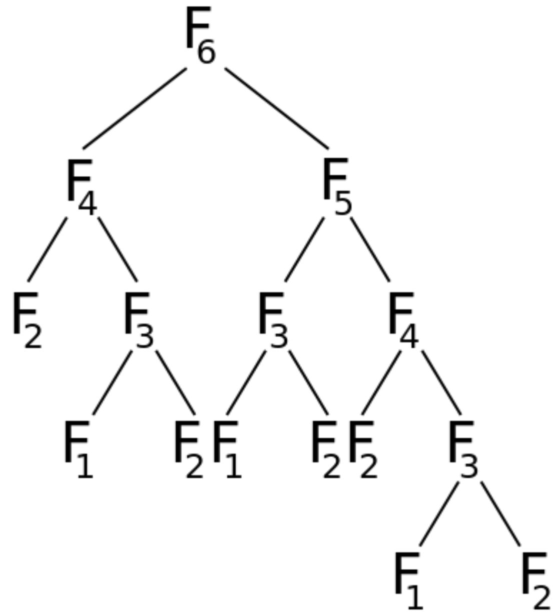
Let's implement with recursion

- Write some function `Func(n)`
    - N is the number of elements in a Fibonacci sequence
        - `Func(4)` returns an array `(1, 1, 2, 3)`
- Remember to define base case, then recursive case
    - Base case: any expression or condition whose result does not depend on anything else
        - For example, the first element in a Fibonacci sequence is 1, and does not depend on anything else

Where is the data structure in recursion?

- We don’t explicitly use a stack in recursion

Notice the duplicate computation!



What happens if we do `Func(999999999)`?

- Interpreter allocates 999,999,999 frames
- Each call frame consumes 1 mb of memory
    - 1mb is over-resourcing, especially a number in javascript takes up 8 bytes (JavaScript does not define different types of numbers, like integers, short, long, floating-point, and every number in javascript is 64 bit)
- We would either run out of memory or, more likely, trigger a stack overflow

Memoization to the rescue

- If we have a fixed CPU budget, then we need to compute only absolutely necessary
    - Don’t do the same thing twice (aka DRY don’t repeat yourself)

Mini assignment

- Modify Fibonacci solution
    - Introduce a conditional ("does this thing exist?")
    - Hint: How do we know if we've seen something before? (This was the first thing we discussed tonight)

Mini assignment

- Description
  - You work at Mozilla on the browser security team. Your job is to ensure the Firefox sandbox works as intended.
- Task
  - Implement a function that estimates the maximum number of call stacks allowed by SpiderMonkey

Why stack overflow?

- The interpreter (js engine) produces a call stack for each function call
- Each call stack consumes 1MB of memory
- But the interpreter doesn't have infinite memory to handle infinite recursion
    - Node.js: 11034 call frames
    - Firefox: 50994 call frames
    - Chrome: 10402 call frames

Let’s try another example

What's a factorial?

- A factorial is the repeated multiplication of a series of numbers
  - 5! = 5 x 4 x 3 x 2 x 1
  - 2! = 2
  - 1! = 1
- Factorials are not just mathematical curiosities; they show up in computer science all the time
  - If we want to hack someone's 5 digit pin, we could brute force with 5^9 combinations.
  - Factorials get really, really big - remember our discussion about time complexity!

Possible Approaches: Recursion

- Implement a recursive solution
- Tips:
  - Draw it out
  - Always define the base case first

A stack trace: What the interpreter sees

- F(5) = 5 * F(4)
    - F(4) = 4 * F(3)
        - F(3) = 3 * F(2)
            - F(2) = 2 * F(1)
                - F(1) = 1
            - F(2) = 2 * 1 = 2
        - F(3) = 3 * 2 = 6
    - F(4) = 4 * 6 = 24
- F(5) = 5 * 24 = 120

Could we be more efficient?

- Why can’t we memoize?

Solve it iteratively

- Notice that, for factorials (eg f(5) = 5 x 4 x 3 x 2 x 1), we track two states (the accumulated value and the next item in the sequence) and nothing else
- That means we can solve it dynamically, and explicitly track state

Mini assignment: Dynamic Programming

- Implementation hints
  - Explicitly define state
  - Think about ways to iterate through the list of numbers