Hash Tables IV

AGENDA

- More Hash Table Applications
- *I will host an AMA tomorrow 6-7PM PST*

Hash Table Applications

CANDY SWAP

- Help Alice and Bob Swap Candies
- Leetcode Link

Understand

Input : A = [1,1] B = [2,2]
Output: [1,2]

Input: A = [1] B = [2,3]
Output: [1,3]

Input: A = [1] B = [1]
Output: [1,1]

runtime: O(len(A) + len(B))
space: O(len(B))
Bob's candy will grow linearly with swap

Plan:
- for every candy x that Alice gives out, she expects a candy y from Bob.
- for every candy that Bob gives out, he expects a candy x from Alice
- since we're doing a fair swap, the candies Alice and Bob has to be equal

sum(alice) - x + y = sum(bob) - y + x
y = ((sum(bob) - sum(alice))/2) + x
make a set out of Bob's candies
and see if has candy y.
try every candy x that Alice has
and see if Bob has candy y that satisfy above equation.

```
class Solution:
    def fairCandySwap(self, A: List[int], B: List[int]):
        bobCandies = set(B)
        bobSum, aliceSum = sum(B), sum(A)
        for x in A:
            y = ((bobSum - aliceSum)/2) + x
            return [x, y]

        return []
```

Candy Swap Demo

MEMOIZATION

fib(n) = fib(n-1) + fib(n-2)

nth
fib(1) = 0
fib(2) = 1
fib(3) = fib(2) + fib(1) = 1

nth → val.
0: 0
1: 0
2:
3: 2 ← gets stored

- Store a previously computed value in a dictionary so you don't have to compute it again
- This can easily be used in computing the *nth* Fibonacci number
- Leetcode Link

```
class Solution:
    def fib(self, N: int) -> int:
        computed Values = {1:1, 0:0}
        return self.memoized Fib (N, computed Values)
    def memoized fib (self, N, computed Values):
        if N in computed Values:
            return computed Values [N]
        computed Values [N] =
            self.memoized Fib (N-1, computed Values)
            + self.memoized Fib (N-2, computed Values)
        return computed Values [N]
```

Using dictionary because we need the key value pair, instead of a set.

repeated calculation
store instead of computing again

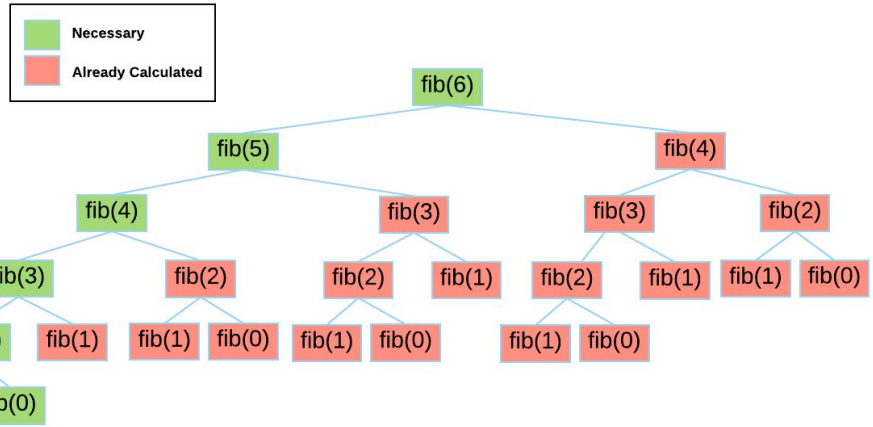


Fig: Fibonacci Number Recursive Implemention

Example:
Fib(3)
Computed Values = {1:1, 0:0}
~~Computed Fib(0)~~ 2:1
memoized Fib(1) 3:2
memoized Fib(2)

Fibonacci Demo

MEMOIZATION: CLIMBING STAIRS

- You can either climb 1 or 2 steps. How many distinct ways can you climb to the top?
- [Leetcode Link]

similar to code above

Plan:
num_ways(n) = num_ways(n-1) + num_ways(n-2)

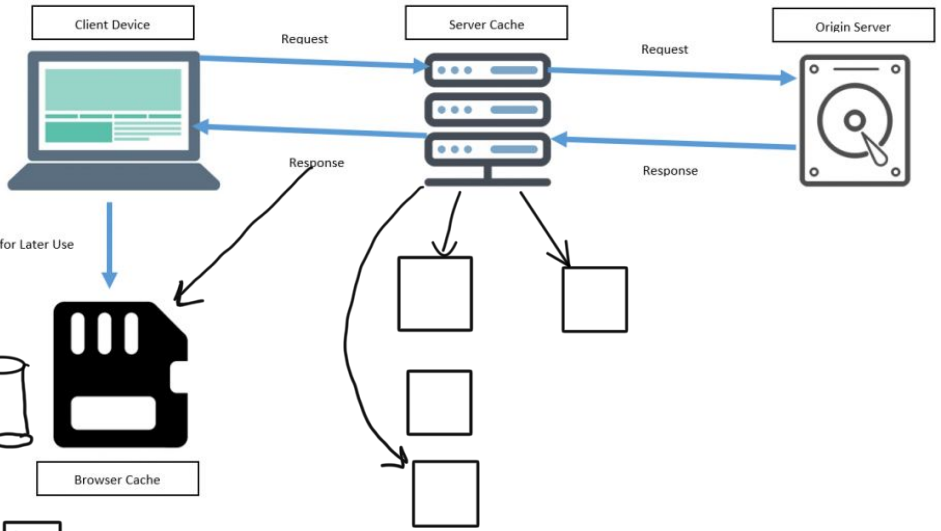
```
class Solution:
    def climbStairs(self, n: int) -> int:
        computed_values = {2:2, 1:1, 0:0}
        return self.climbStairsHelper(n, computed_values)
    def climbStairsHelper(self, n, computed_values):
        if n in computed_values:
            return computed_values[n]
        computed_values[n] =
            self.climbStairsHelper(n-1, computed_values)
            + self.climbStairsHelper(n-2, computed_values)
        return computed_values[n]
```

Climbing Stairs Demo

CACHING ⟷ Hash Tables

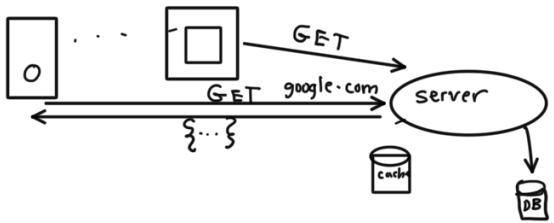
- Similar to memoization, store a value after it's been computed so that you have faster access to it in the future
- **Widely used** in the industry for a bunch of different use-cases: networking, performance, etc.

Ram → caching
browser Caching



map key from associated value

Key value
url payload that has been cached



Caching Demo

```
simulate expensive computation
import random
import time

class Server:
    cache = {}
    def processGETRequest(self,url):
        if url in cache:
            print(url)
            return self.cache[url]
        print(url + "caching")
        self.cache[url] = self.doExpensiveComputation(url)
        print("caching" + url + "with value")

    def doExpensiveComputation(self,url):
        time.sleep(5)
        return random.randint(0,101)
    def processPUTRequest(self,url)
        * update DB with new values
        * evict url from cache
```