5_slow_code

June 24, 2025

1 Why is my code slow?

1.1 Outline

- Caching, Memoization, and Vectorization
- Parallel Computing
- Faster Implementations versus Faster Algorithms

2 Caching, Memoization, and Vectorization

2.1 Caching

- *Caching* refers to storing things for later use
  - Your browser probably does by temporarily downloading page details on your local disk
  - Faster, reduces server load
  - Other examples include 3D rendering and saving common database queries
- However, caching usually takes space in exchange for faster run times
- The *space-time* trade off is a case where an algorithm trades increased space usage for faster runtimes

2.2 Memoization

- *Memoization* refers to storing results of function calls to use for later
  - Specific method of caching
- This is useful for methods with a lot of repeated computations
- For instance, in our recursive Fibonacci number function.
- `fib(12)` is called by `fib(13)`, `fib(14)` etc.
  - And `fib(3)` is called many many times
- $F(5) = F(4) + F(3) = F(3) + F(2) + F(2) + F(1)$ Which calculates repeated subproblems

2.3 How Memoization Works

- Since we store the results, each function call is only made once, making the time complexity $O(n)$, much better than $O(2^n)$ [1]
- Memoization can also avoid the maximum recursion depth error because the call stack is smaller

2.4 Memoization Python

[1] From Bhargava chapter 8

[1]:
```python
def fib_original(n):
    if n <= 1:
        return n
    else:
        return fib_original(n-1) + fib_original(n-2)
```

[5]:
```python
fib_original(40) # 30 seconds calculate O(2^n)
```

[5]: 102334155

[ ]:
```python
```

[ ]:
```python
```

[9]:
```python
```

[9]: 102334155

[6]:
```python
cache = {0: 0, 1: 1}

def fib(n):
  if n in cache:
    return cache[n]
  else:
    cache[n] = fib(n - 1) + fib(n - 2)
    return cache[n]
```

[8]:
```python
fib(40) # Does not take 30 seconds
```

[8]: 102334155

[12]:
```python
fib(3000) # Does not take 30 seconds O(n)
```

[12]:
```
4106158863079712603335683787192671052201251086373692524088854309269055842741134037313304916608500445608300368357069422745885693621454765026743730454468521604866062924973605034697734537331968874058472552900820490869075126220590545421958897580311092226708492747938595391333183712447955431476110732762400667379340851917318l
```

```
0993201706776838934766764778739502174470268627820918553842225858306408301661862900358266857238210235802504351951472997919676524004784236376453347268364152648346245840573214241419937917242918602639810097866942392015404620153818671425739835074851396421139982713640679581178458198658692285968043243656709796000
```

[ ]:
```

```

[ ]:
```

```

[ ]:
```

```

[ ]:
```

```

For the base cases, we replace calling `fib(0)` and `fib(1)` by getting the values from the dictionary

2.5 Memoization Python

- We can use the `functools` library, which is included in the standard library (no pip install needed!)
  - `functools` does memoization for you!
- We can use the `@cache` decorator, but the cached dictionary can grow to massive sizes
- Instead, `@lru_cache(maxsize = n)` uses the LRU (least recently used) `n` computations
- Alternatively, we can use `joblib` to store the memoized results in a file

2.6 Memoization Python

[20]:
```python
from functools import lru_cache

@lru_cache(maxsize=3)
def fib_rec(n):
  if n == 0 or n == 1:
    return n
  else:
    return fib_rec(n-1) + fib_rec(n-2)
```

[21]:
```python
fib_rec(300)
```

[21]:
```
222232244629420445529739893461909967206666939096499764990979600
```

[ ]:
```

```

[33]:
```python
@lru_cache(maxsize=5)
def x(n):
    return n
```

[34]:
```python
x(40)
```

[34]: 40

[38]:
```python
from cache_em_all import Cachable
```

[ ]:
```python

```

[ ]:
```python
np.array([1, 2, 3, 4, 5]).max()
```

[ ]:
```python

```

2.7 Vectorized Operations

- *Vectorization* is a technique of implementing array operations without for loops
- We use functions defined by various modules that are highly optimized for the specific problem
- NumPy provides a lot of functions that vectorized and are faster than for loops
  - Array add/subtract/multiply/divide by scalar
  - Sum of array
  - Max/min of array
- Keep this in mind for some ML processes that are iterative, such as gradient descent

2.8 Why Vectorized Operations Work

- Python is an interpreted language. There is no compiler and the languages are dynamic
- C language, for instance, makes optimization at the compiler level (before execution) to speed up your code
- Thus, NumPy implements arrays in C, which speeds things up
- The other reason vectorization works in because of parallelization

3 Parallel Computing

3.1 Parallelization

Compare the following codes. What are their run times?

[3]:
```python
@lru_cache(maxsize=10) # O(n)
def fib(n):
  if n <= 1:
    return n
  else:
    return fib(n - 1) + fib(n - 2)
```

3.2 Parallelization

[4]:
```
import numpy

def add_one(n, x): #O(n)
  y = np.zeros(n)
  for i in range(n):
    y[i] = x[i] + 1

  return y
```

[ ]:
```
5, [1, 2, 3, 4, 5]
[2, 3, 4, 5, 6]
```

[ ]:
```
# joblib
```

[ ]:
```

```

3.3 Parallelization

- Both are $O(n)$, but the second code chunk can be done in *parallel* because the n computations are independent.
- Fibonacci depends on the previous two values
- The requirements for code to the parallelized and vectorized are similar, but not the same
- The Numba library can help will parallelizing your code
- Note parallel means the process takes place on one machine, but *distributed* means the computation is shared across many machines

[ ]:
```

```

From leetcode

4 Faster Implementations versus Faster Algorithms

4.1 Faster Implementations versus Faster Algorithms

- There are two ways we speed up our code
  - Use a faster algorithm, such as dynamic programming instead of brute force. Algorithms are concerned with the approach to the problem
  - Use a faster implementation, such as vectorization instead of loops
- It is useful to think about these separately when developing a programming, then combining them to create a super-fast approach!

5 Recommended Problems and References

5.1 Recommended Problems and Readings

- Cormen: Chapter 34 on NP-Completeness (highly optional)
- Bhargava: Chapter 8 exercises
  - 8.1 - 8.8
- Vectorize the second code chunk in the Parallelization section
- Find the longest palindrome from a string Hint: use a greedy alogrithm
- Computing Pascal's triangle Hint: use dynamic programming

5.2 References

- Bhargava, A. Y. (2016). *Grokking algorithms: An illustrated guide for programmers and other curious people.* Manning. Chapter 1.
- Cormen, T. H. (Ed.). (2009). *Introduction to algorithms* (3rd ed). MIT Press. Chapter 1 and 3.