Fibonacci web API

Jeban Kanagarajan

Published with GitBook

Table of Contents

Fibonacci Web API

Problem Definition: Implement a web service.

1. to support a REST GET call.

a. The web service accepts a number, n, as input and returns the first n Fibonacci numbers, starting from 0. I.e. given n = 5, appropriate output would represent the sequence "0 1 1 2 3".

b. Given a negative number, it will respond with an appropriate error.

1. Include whatever instructions are necessary to build and deploy/run the project. "deploy/run" means the web service is accepting requests and responding to them as appropriate.
2. add enough unit tests. add some functional tests. list all other tests you would think of.
3. While this project is admittedly trivial, approach it as representing a more complex problem that you'll have to put into production and maintain for 5 years.

Basic Approach:

Features

- Implemented using flask-restful module in python.
- The Web service accepts get call at the defined endpoints and throws forbidden error otherwise
- The Rest API exposes a novel implementation of Fibonacci which serves a fibonacci series for a given * integer value n as a response to a GET call.
- If a negative value is passed error message will appear in response.
- If a non-numeric value is passed error message will appear in response.
- As problem size grows option of using caching
- All configurable parameters are placed in a central yaml config file

Design and Implementation

API Design:

API Version: v1

GET : /v1/fibonacci/get/{number}

type: application/json

Response:

{

"results":['0','1','1','2','3','5'],

"status_code": 200

}

Error Response:

{

"Exception": {"message"},

"status_code" : [errorcode]

}

Response Status Codes:

NEGATIVE_CODE: 400

SUCCESS_CODE: 200

FORBIDDEN_CODE: 405

CACHE_LIMIT_EXCEEDED_CODE: 414

Web Server:

- The Web Server used here is Flask, built on Python.
- flask-restful module provides the necessary RESTful capabilities.
- Caching is done using "pickle" module in python, which creates serialized cache of our list and stores it for later retrieval

- A Decorator called timeit has been used to time the execution of fibonacci method
- Flask web server scaled well when checked with apache benchmarking tool for 100000 requests through ten concurrent connections
- No authentication is involved in accessing the API.
- Output is dumped to the browser or stdout if accessed via shell.

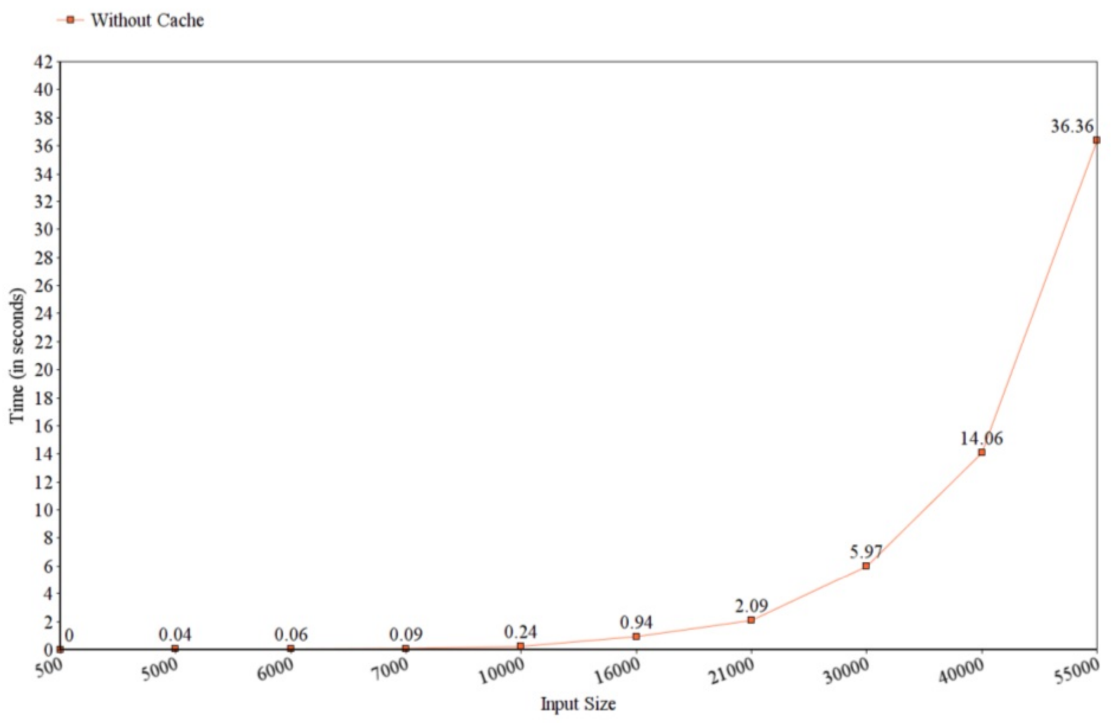
Fibonacci Implementation

- The Fibonacci Sequence is the series of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... The next number is found by adding up the two numbers before it
- In this application a straightforward loop based implementation with O(n) complexity has been implemented.
- Also Memoization is used so as to cache few expensive calculations , although they do not contribute to speeding up the computation by a large margin as the API by itself is stateless.
- Since recursion limit restricts the usage of recursion to compute large sequences this method works better for this use case.
- The method takes in the number of elements as argument and returns a list of fibonacci strings.
- Why string? Why not int? As the problem size grows int will not have the range to cover higher order numbers so it is better to store them as strings to get the correct results.

Caching Mechanism

- Before implementing caching for the application tests were run for growing problem size , and execution time was recorded for the same. Though It might not be a right heuristic of measuring efficiency (due to hardware constraints) , it gives a fair idea of how time grows linearly.

- Here is the graph of Input Size Vs time without caching



- From the above graph it is clear that as the problem size increases execution time increases gradually upto a certain value and increases exponentially beyond certain values. This gave the clue on how and when to cache

Caching Mechanism 1:

- The first mechanism tried for caching was to cache the output to disk using pickle and access the cache for subsequent requests lesser than cache output size.

Pseudocode: Cache Creation:

```
        createcachefile(fibonacci(noOfElements))
```

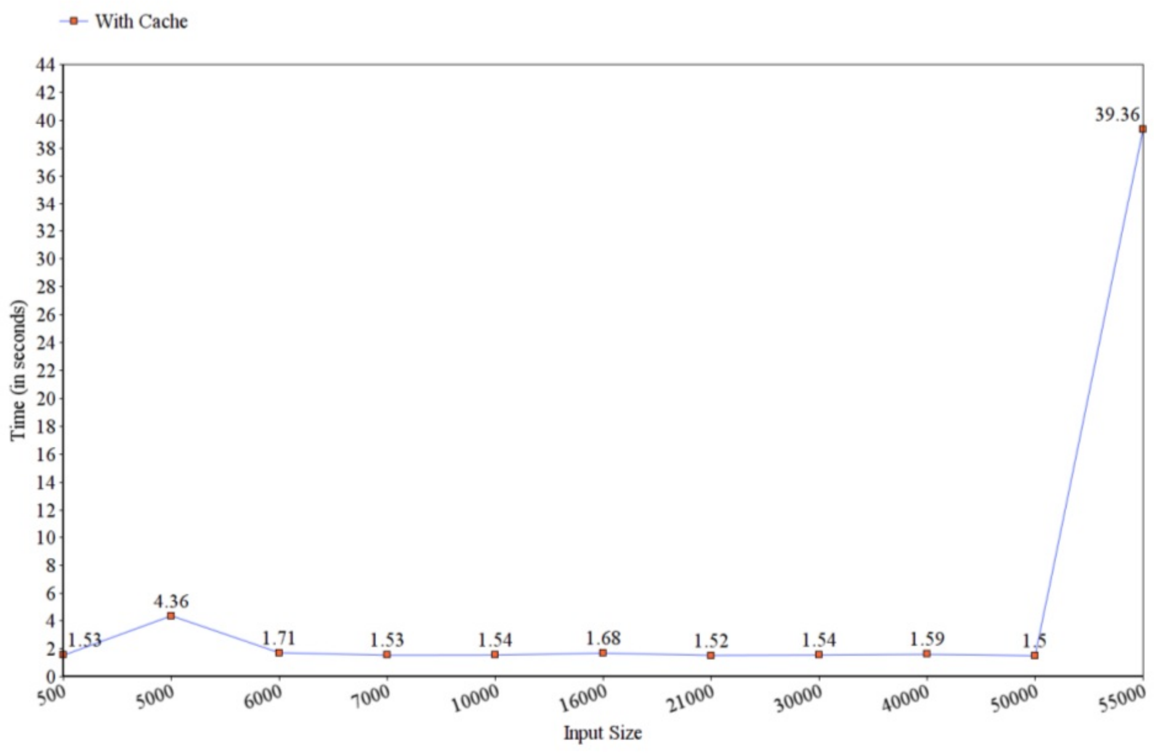
Cache Retrieval:

```
   if noOfElements in available_list_of_caches:
        return readfromcache(noOfElements)

   elif noOfElements < max(available_list_of_caches):
        return readfromcache(maxCache[0:noOfElements]
```

Here is the graph for this mechanism: Please note that:

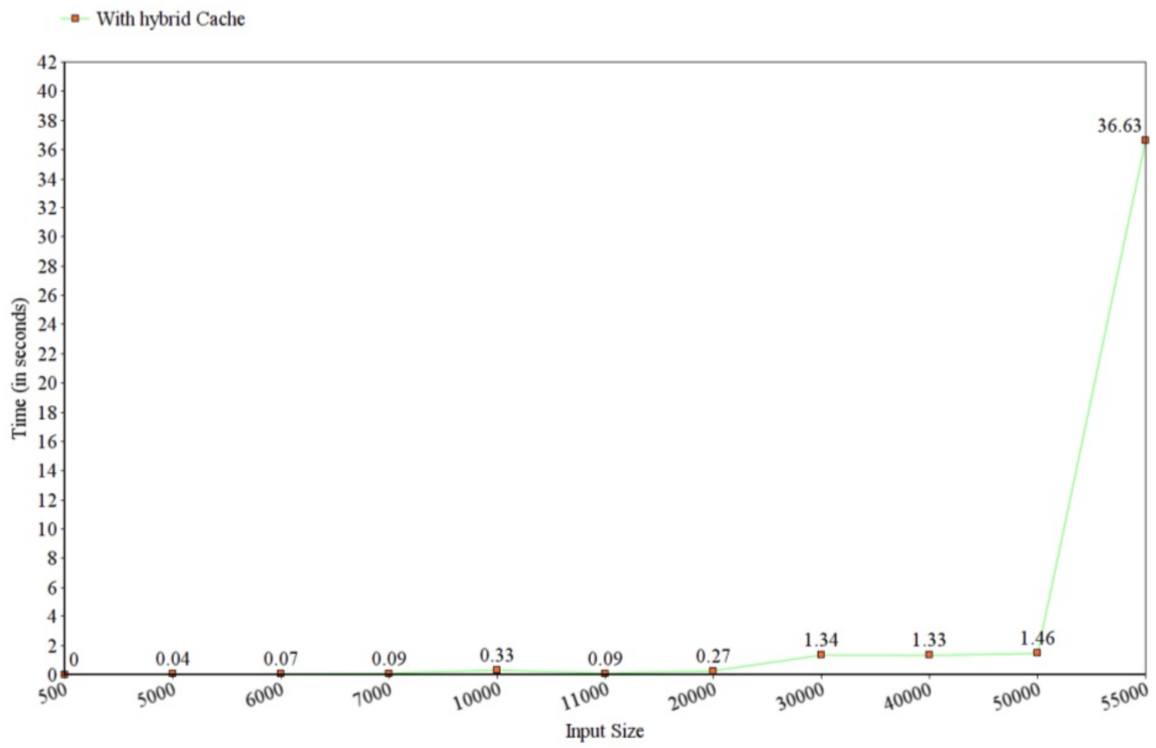
- A Cache for 50000 was made prior to testing which means that every value accessed the cache



As you can see when compared to graph 1 we get better execution times but only for values above 16000, which means that caching doesn't help much if the problem size is small or when the difference between the cached output size and the no of elements is significantly larger.

Caching Mechanism 2:

- So Why can't we have the best of both worlds?
- In this mechanism a lower threshold and a upper threshold have been specified.
- Cache is generated only for inputs above lower threshold, for the rest normal fibonacci generation is done
- And certain outputs are pre-cached at different intervals from 10000
- Upper threshold is set at 60000 (For my machine) which will throw an error when values above it are given to the API.
- A Cache is chosen by finding the cache output which is closer to the input number of elements so as to reduce access complexities which hampered execution time in the previous mechanism. Here is the Graph:



As you can see since it does not use cache for values below threshold the execution time is faster, similarly by having caches at different intervals , input value closest to cache is chosen everytime there by reducing execution time considerably.

Points to Consider

- Cache binary sizes increase significantly for larger problems. Eg. fibonacci(50000) cache is 250MB in size.
- Can we improve this ? How to handle larger cache sizes and growing access complexity of a list ?
- May be an alternative data storage so as to get linear access time ?
- Or can we rotate cache on a time to time basis so as to not fill the server storage.
- Can a key value store like redis help solve this ?
- How to scale without running out of memory?

Ideas for production

Though this application is production ready, Few thoughts

- For this application to scale on a production level load balancing has to be done so as to evenly distribute computational load.
- On security level, offer token based authentication to requests where in each request should carry a authentication token header to prevent unauthorized access. Enable SSL always.
- Log analysis to identify any critical errors and inform the system admin via mail.
- Have a distributed caching mechanism.
- Option to download the output instead of dumping it as a json response which becomes

unmanagebale for large inputs on browser or stdout.

- Parallelise the computation for large input sizes by using threads level or core level parallelism (Python doesn't offer parallelism though, due to interpreter level lock)

Code Setup

- Code is completely modularised and generic.
- Configurations are set in config.yaml
- helper functions are implemented in helper.py
- actual fibonacci implementation and caching read and write is done in fibonacci.py
- main app, app.py multiplexes the input value between reading output from cache and by computation
- test.py has unittests covering the fibonacci implementation
- func_test.py has the functional tests
- Any new feature can be added easily through fibonacci.py and new routes can be added at app.py.

Some tests that needs to be done

- Out of memory test: When does my application run out of memory ?
- Crash test: When does my application crash ?
- pylint for code quality (done for this application)
- DDOS vulnerability