My Algorithm

* Insertion Sort
  + Why
    - Was once the default algorithm for Chrome when using the Array.prototype.sort() method. It appears now it has moved onto a Timsort algorithm which is a combination of merge and insertion sorts.
    - Was not the best use-case for large inputs
  + Runtime depends on input size and input order.
    - Runtime is the number of steps/ operations executed
    - It is the sum of the running times for each statement executed.
      * Statement that takes c steps to execute and executes n times with contribute cn to the total running time.
  + When
    - Preferred for smaller inputs. Fasted when input is already sorted or almost sorted
    - Efficient in terms of space
      * Sorts the entire array with only 1 extra variable

Big O Notation

Is a way to describe the time it takes a function to run depending on the size of the input. It can also be used to show memory consumption. But it wont give us an actual number, like how much time or how much memory, but rather it shows the growth.

Getting into my algorithm, I chose an insertion sort which is not a great performer. I chose this because I was curious to which algorithm was behind the Array.sort method and in an earlier article I read, it mentioned it was the insertion sort. So I got started on with that but when I went to verify that claim, I couldn’t find much backing that up. Right now I believe the algorithm behind array.sort to be a timsort , at least in a chrome browser, which is a combination of a merge sort and insertion sort.

So as I said, my algorithm performs poorly. It has a runtime of O(n2), (ill get into how that’s determined in a minute) but this is the average and worst case scenario. In it’s best case, it has a runtime of O(n), but generally we just look at the worst case scenarios.

RESTful Architecture

* **Representational State Transfer** 
  + A set of rules for creating web services. Allows for uniformity when requesting to access and manipulate a resources and helps with scalability
  + Constraints
    - **Uniform Interface**
* **Defines the interface bwtn client and server** and requires every service to use HTTP the same way. There’s a few guidelines that further define this, the first being
  + ***Identification of a resource***. When doing so, it should be **unique** and follow a **hierarchical approach**, for example facebook.com/users/user(id)/pictures/picture(id).
  + Another aspect of a uniform interface is ***Resource manipulation***, which is when the client requests a resource from an URI (unique resource identifier), the **server responds with the representation of the resource**, which can be JSON, xml, html, images etc. The client then has the ability to manipulate this representation of the resource.
  + The third part is using ***self-descriptive messages****.* Because the server is stateless, we need to **send all the necessary information with our request to gain a resource.** When can add this info to our method types, like GET, POST, DELETE etc.. So with the same URL and different methods, we can achieve different goals.
    - **EVENT SCHEDULER- put, patch, delete all same enpoint, methods vary**
  + The last guideline for having a uniform interface is ***hypermedia as the engine of application state*** which means the response of a **resource should include “links”** which define what we are able to do. So if I make a comment on a facebook photo, I should get links saying I am able to edit (PUT/PATCH) or delete(DELETE).
    - **Stateless**
* The server should contain no client state and that session states should be held with the client on the front end. When a client makes a request, they should provide all the necessary information that the server needs. Allows for **greater availability** if the server doesn’t have to maintain, update or communicate session states. **Better transportability**.
  + **EVENT SCHEDULER- server is on Heroku, frontend is on GH pages**
    - **Cacheable**
      * Each response should include whether or not the response is **cacheable or not and for how long** responses can be held on the client side.
      * Well managed caching can **reduce some client-server interactions**, improving availability and performance.
    - **Client-Server**
      * Assumes a disconnect from the server and client which adheres to the separation of concerns principle.
        + Client requests resources and not concerned with data-storage
        + Server holds resources and not concerned with UI or user state
      * Can **evolve independently**
    - **Layered System**
      * Application architecture is made up of layers
        + Ex: pages, components, config/authorizations, DAL, API routing
        + **Each component cannot see beyond the immediate layer that they are interacting**
* REST APIs
  + REST is based on the resource or noun, not the action/verb
    - /api/users
  + HTTP verbs used to identify the action
    - GET, POST, PUT, PACTH, DELETE
  + Web apps should be organized into resources like users and then use HTTP verbs to modify those resources.
    - For the developer, should be clear what needs to be done just by looking at the endpoint and HTTP method used.
      * Examples
        + api/users + GET = get all users
        + api/users + POST = add a user
        + api/users/1 + PUT = update user with id = 1
        + api/users/1 + DELETE = delete user with id = 1
        + api/users/1 + GET = get user with id = 1
  + Always use plurals in URL to keep an API URI consistent throughout application
  + Send proper HTTP code to indicate a success or error

My tech

* **cors**:
  + Security measure implemented on the browser-side. Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources on a web page to be requested from another domain. Need it for server to browser interactions, not server to server.
* Node
  + Servers use node to interpret JS
* Express
  + Web app framework for Node.js
  + Acts as the server in the client-server relationship in restful architectures