ORIE 5270: Big Data Technologies

Homework 2 – **Due**: Friday, 03/18/2022, 5pm

Instructions. The deadline to submit is **Friday, March 18td** at **5pm US EST**. Submit your answers to **Gradescope**. Please submit a single `.zip` archive containing a single file per problem, using the names suggested in each Problem.

Collaboration: for Problem 2 (and Problem 2 **only**), you may work in groups of 2. If you choose to do so, make sure to acknowledge your collaborator by adding a comment in the submitted source file. You and your collaborator should submit the same file for Problem 2 to Gradescope.

Problem 1 (Complexity classes)**.** In this problem, you are given a set of functions $f_i(n)$. Sort these functions **in ascending order** according to their complexity class, as indicated by $O(\cdot)$ notation; in particular, if f_i appears before f_j in your answer, it should satisfy $f_i(n) = O(f_j(n))$.

Note: there is *only one* correct ordering for the set of functions given below. For some of these comparisons, you might find Stirling's formula useful.

$$f_1(n) = 3n^2 - n + 10, \quad f_2(n) = \sqrt{n}\log(n), \quad f_3(n) = n^{3/4}, \quad f_4(n) = \log^{100}(n),$$
$$f_5(n) = 2^{n/3}, \quad f_6(n) = \log(n^{50}), \quad f_7(n) = 2^{n-\log_2 n}, \quad f_8(n) = \frac{\log(n!)}{n^{1/4}}.$$

What to submit: submit a single file called `complexity.txt` that contains a comma-separated list of indices i corresponding to the f_i's placed in ascending order. For example, if the first 3 functions in your answer are f_7, f_8 and f_2, your file should start like this: `7, 8, 2, ...`

Problem 2 (Video cache)**.** In this scenario, you are a developer in a video-streaming company. You want to create a caching mechanism for streaming that works as follows:

- There is a list of movies available, described by their title, and a list of geographical locations, each of which is described by a triple `(Location, X, Y)`.
- Your mechanism maintains a cache for each geographical location to accelerate streaming. Each cache stores a number of K movies, to be specified when the mechanism is first created. Initially, every cache is empty.

- When a user wants to see a movie, they submit a request to your mechanism that contains the title of the movie as well as their geographical coordinates: `(MovieTitle, X, Y)`. Your mechanism processes this requests as follows:

  1. First, it finds the nearest video cache by geographical location.
  2. Then, it checks if the requested movie title is available in the nearest cache. If it is available, it responds with `(True, Location)`, where `Location` is the geographical location of the nearest cache. If not, it responds with `(False, None)`.
  3. If the response was `False`, the mechanism updates the video cache by including the movie requested (so that a subsequent request for the same movie will return `True`). If the cache was full, the new movie *replaces the movie that was the least recently brought into the cache.*

Implement the caching mechanism as a Python class, using the template provided under `caching_mechanism.py`. In particular, you should implement the constructor, and 3 functions: `find_nearest_cache()`, `update_cache_state()`, and `lookup()`. Refer to the documentation in the template source file for the expected inputs and outputs.

You should use appropriate data structures for each task. In particular, the `lookup()` function should take time $O(1)$ on average. The `update_cache_state()` function should take time $O(1)$ in the worst-case. You will have to use multiple data structures to accomplish this.

Note 1: this problem might require some high-level planning (e.g., which data structures to use, how to implement a cache at a fixed location, etc.). Make sure you start early so as to not get overwhelmed.

Note 2: Use the Euclidean $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ to determine the nearest cache.

Problem 3 (K-largest elements in data streams)**.** In this scenario, you are a developer for a company that processes bids for auctions. In the simplest case, which we examine here, you conduct a single auction and receive a number of bids for the item auctioned.

You model the incoming beads as a data stream (of potentially unbounded length) of integer values, which are the bids themselves. Among all the incoming bids, you want to keep track of the K highest ones. Furthermore, you want to be able to dynamically maintain this catalogue of highest bids *without storing the entire stream.*

Implement a class emulating the auctioneer using the template provided in `auction.py`. In particular:

- The class should maintain an appropriate data structure that keeps track of the

highest K bids, where the number K will be provided to the constructor of your class, using $O(K)$ space.

- Your class should implement a method called `process_next_bid()`, which processes an incoming bid and updates the catalogue of the highest K bids, if necessary. The worst-case runtime of this method should be $O(\log K)$.
- Your class should also implement a method called `get_bids()`, which returns the highest K bids processed so far in increasing order.
- Your class is *not allowed* to store the entire bid history.

Problem 4 (Mergesort with 4 splits)**.** In this problem, you will implement a version of mergesort that splits the problem into 4 parts instead of 2. Using the template provided under `mergesort_4.py`:

1. Implement a function `merge_4(A, B, C, D)`, which accepts 4 sorted arrays and merges them into a single sorted array.
2. Implement `mergesort_4(A)`, which sorts the array A using divide-and-conquer similar to the ordinary `mergesort` function but with 4 instead of 2 subproblems.

Note: you are not allowed to use any Python libraries in your solution. You are, however, allowed to re-use the `merge` function shown in class.

Problem 5 (Personal website)**.** If you don't have a personal website, now might be the time to build one! Many repository hosting services, such as Github and Gitlab, allow you to build static websites (meaning: HTML + CSS + Javascript, but no backend components).

How it works: you create a repository named like `<username>.github.io` which contains the source for your website. Then, a *static site generator* specified by a configuration file that you include in your repository builds your website from the source files you included. The website is then made available under the same URL as the name of your project.

Because the process varies depending on the hosting service as well as the site generator of your choice, you should devote some time to browse and familiarize yourself with the service and the static website generator of your choice. Here are some links to help you get started.

1. Static website generators:

- Jekyll: https://jekyllrb.com/
- Hugo: https://gohugo.io/
- Hakyll: https://jaspervdj.be/hakyll/

2. Links to static website hosting services:
   - Github pages: https://pages.github.com/
   - Gitlab pages: https://docs.gitlab.com/ee/user/project/pages/

What to turn in: by the last lecture of the semester, you should build at least a minimal website (e.g. Home / About Me / Projects / Misc) with some meaningful content. For example, you should add your bio and academic interest and a showcase of projects you have done in the past or are currently undertaking. Then, you should send me an email with subject line `[ORIE 5270] - Personal Website` containing the following:

- If you make your website public, your email should contain a link to your website.
- If you would rather not make your website publicly available at the time, your email should contain a `.zip` file with your website's source code, as well as instructions on how to "build" it locally.