Extra Credit Homework

Due on May 24th, 2021

NOTE: This assignment is OPTIONAL
In this assignment, you will implement a simple Bloom Filter and the Knapsack dynamic programing algorithm.

Key instructions:

- 1. Please indicate your name at the top of each assignment(in comments for each piece of code you submit) and cite any references you used (including articles, books, code, websites, and personal communications). If you're not sure whether to cite a source, err on the side of caution and cite it. Remember not to plagiarize: all solutions must be written by yourself.
- 2. Please use **Python 3** for this assignment, the grading of this assignment will be done in a Python 3.7.5 environment.
- 4. Please do NOT use any additional imports, only write your code where you see `TODO: YOUR CODE HERE`, and change your return value accordingly.
- 5. Note on Late Policy: You will **NOT** be able to use a slip day on this assignment. You **MUST** submit your assignment by May 24, so that final grades can be submitted on time.
- 6. Please modify and submit the following files:
  - `bloom_filter.py`
  - `knapsack.py`

Bloom filter

The first part of the homework is to implement a simple Bloom filter.

TODO In the provided class (`bloom_filter.py`), implement

- `add_elem`
- `check_membership`

Important: Your Bloom filter implementation **must** use the three hash functions that we have provided (`cs5112_hash1`, `cs5112_hash2`, and `cs5112_hash3`).

Do not use Python's built-in hash function! You must use these three.

You are asked to implement a Bloom filter with k=3 hash functions and a m=10 bit array.

The comments in `bloom_filter.py` say more about what we expect for inputs and outputs.

Testing Bloom filter

For your Bloom filter, in addition to testing the output of your functions, we will also be inspecting how you're storing data in the underlying array. Therefore, be sure to implement the algorithms to spec.

Run `python3 test_bloom_filter.py` for a very simple test. Note that this test is not exhaustive, so you are encouraged to write tests of your own as well.

Knapsack

Problem statement

The second part of the assignment is to implement the algorithm for solving the Knapsack problem using dynamic programming (as was covered in lecture). As a reminder, the Knapsack problem takes as input a list of items, each represented as a (value, weight) tuple with the respective value and weight of the item, as well as an integer W representing the overall weight that the knapsack can carry. The goal is to find which items we can put in the knapsack that maximizes the sum of the item values while still insuring that the sum of the item weights is less than or equal to W.

For example, lets say we had the following inputs:

- items = [(3, 2), (4, 3), (5, 4), (6, 5)]
- W = 5

In this situation, the correct output would be [(3, 2), (4, 3)], with a total value of 7.

TODO

In `knapsack.py`:

fill_cell . You should fill in `fill_cell(table, i, j, items, W)`, which will be called once per cell. `table` is the table as filled in so far; use its `get` method to find these values. Your function should not mutate the table directly, but rather return a `KnapsackCell`. `fill_cell` should run in $O(1)$ time.

cell_ordering . You should fill in `cell_ordering(items, W)`, which will be called once. `cell_ordering` should return a list of two-tuples; these are the coordinates of cells in the order you would like `fill_cell` to be called. No cell should be repeated, but you do not need to use all cells.

knapsack_from_table . You should fill in `knapsack_from_table(items, W, table)`. `knapsack_from_table` should run in $O(len(items))$ time. It should return a two-tuple (included_items, total_value) representing the items that are chosen to put in the knapsack and the total sum of those items' values. NOTE: the order of the items in your included_items output does NOT matter, all that matters is that the correct items are in the list.

Once you have finished implementing these 3 functions, you can do a quick sanity check by running the `main` given at the bottom of `knapsack.py` which puts all the pieces together with our dynamic programming framework. See if the result is coherent with the example we gave you in this document.

```
python3 knapsack.py
```

Test harness To help you concentrate more on the actual algorithms part, we have also provided you a test harness `test_knapsack.py` that runs tests on your `knapsack.py` implementation using `knapsack_tests.txt` as the input testfile. It is hence your responsibility to make sure your code works before submission. However, the test harness and the test cases we gave you are **non-exhaustive**, meaning *passing these tests alone does not necessarily guarantee a perfect score*.

```
python3 test_knapsack.py
```

Dynamic Programming Framework

We have provided the skeleton code for you, in `dynamic_programming.py` and `knapsack.py`. It is designed to let you just think about the algorithm and worry less about the software engineering stuff, while also ensuring that the algorithm you design is in fact a dynamic program. You will write some code to fill in a dynamic programming table, and then to compute the final answer from this table.

The table for knapsack is $(N+1) \times (W+1)$, where N is the length of the input list of items, and W is the input W representing the total weight the knapsack can carry. Each cell may contain a single boolean and a single integer.

Testing and logistics

Outside resources

You are welcome to reference lecture notes and office hours for examples/guidance on how to approach the homework problems. However, we take plagiarism seriously and expect the final output of your work to be your own. **You should NOT use or seek any other resources that describe how to solve these problems, even in pseudocode.**

Testing

In addition to the provided tests, we encourage you to write additional ones.