Algorithms for Applications	HW1: *Stable Matching, Greedy, Divide and Conquer*
CS 5112 Fall 2023	**Due: September 26, 11:59pm ET**

Homework Instructions: The homework contains mainly three types of tasks:

- **Coding.** You can find a template for all the coding problems inside the `problems.zip` in the module. Please use these templates, to ensure the files and functions have the same names that the autograder will check for.

  Inside the `problems` directory, you will find the `tests` directory, which contains some sample test cases for the different problems. These test cases are intended to help you test and debug your solution. However, note that Gradescope will run a more comprehensive test suite. Feel free to add additional test cases to further test your code.

  In order to run the tests for a particular problem n, navegate into `problems/tests` and run: `python3 test_problem_n.py`

- **Algorithm Description.** When a questions requires designing an algorithm, you should describe what your algorithm does in the writeup, in clear concise prose. You can cite algorithms covered in class to help your description. You should also argue for your algorithm's asymptotic run time or, in some cases when indicated in the problem, for the run time bounds of your implementation.

- **Empirical Performance Analysis.** Some questions may ask you to do an empirical performance analysis of one or more algorithms' runtime under different values of n. For these questions, you should generate at least ten test cases for the implementation, for various values of n (include both small and large instances). Then, measure the performance locally on your own system, by taking the median runtime of the implementation over ten or more iterations. Graph the resulting median run times. The x-axis should be instance size and the y-axis should be median run time. You can use any plotting library of your choice. If you have never plotted on Python before, this matplotlib turorial has some examples.

  If several subquestions in a problem ask you to provide performance analysis, please do them all in the same environment (same system and configuration), in order to have a more accurrate comparison between algorithms. You can include graphs for a same problem in the same plot, as long as the different graphs are properly labeled.

  For these types of questions, you will not need to submit the tests you generate or your code for benchmarking or plotting the algorithms. However, you should include the graphs you generate into the write-up. You should also write about what you observe from the analysis and potentially compare it to the complexity analysis. Finally, also include a description of the environment (CPU, operating system and version, amount of memory) you did the testing on.

Submission Instructions: Hand in your solutions electronically on Gradescope. There are two active assignments for this problem set on Gradescope: one with the autograder, where you submit your code, and another one where you submit the write-up.

Coding Submission: You only need to submit the code for the specific functions we ask you

to implement. While you will write additional code to generate the performance analysis graphs when required, you do not need to submit that code.

To submit to the autograder, please zip the `problems` directory we provided you with. Therefore, your zipped file should have the following structure:

```
problems
├── problem_1
│   ├── p1_a.py
│   ├── p1_b.py
│   └── ...
├── problem_2
│   └── p2.py
└── ...
```

In order to ensure your code runs correctly, do not change the names of the files or functions we have provide you. Additionally, do not import external python libraries in these files, you don't need any libraries to solve the coding assignments. Finally, if you write any helper functions for your solutions, please include them in the same files as the functions that call them.

We have some public tests in the autograder to verify that all the required files are included, so you will be able to verify on Gradescope that your work is properly formatted shortly after submitting. We strongly encourage checking well ahead of the due date that your solutions work on the autograder, and seeking out assistance from the TAs should it not. We cannot guarantee being responsive the night the assignment is due.

Write-up Submission: Your write-up should be a nicely formatted PDF prepared using the LaTeX template on Canvas, where you can type in your answer in the `main.tex` file. If you do not have previous experience using LaTeX, we recommend using Overleaf. It is an online LaTeX editor, where you can upload and edit the template we provide. For additional advice on typesetting your work, please refer to the *resources* directory on the course's website.

Academic Integrity: You may use online sources or tools (such as code generation tools), but any tools you use should be explicitly acknowledged and you must explain how you used them. You are responsible for the correctness of submitted solutions, and we expect you to understand them and be able to explain them when asked by teaching staff.

Collaboration Policy: Collaboration (in groups of up to three students) is encouraged while solving the problems, but:

1. list the netids of those in your group;
2. you may discuss ideas and approaches, but you should not write a detailed argument or code outline together;
3. notes of your discussions should be limited to drawings and a few keywords; you must **write up the solutions and write code on your own**.

Problem 1. Your friend is wrapping up medical school and applying for residency programs, but is concerned and confused about how the matching system works. As the local expert on algorithms, your colleague wants your help understanding how matchings work.

(a) (*2 points*) Your friend tried to implement the textbook Gale-Shapley (G-S) algorithm in Python. We provide this code in the homework materials in `problem_1/p1_a.py`. Using this implementation, your friend thinks they found a way to propose a ranking that unfairly advantages them in getting the school of their choice.

There is a logical bug in the implementation. Provide a minimal test case demonstrating the bug, i.e., an input with the smallest possible n that, when run with the buggy implementation, outputs a non-stable matching. Write your test case input in `problem_1/p1a_test.txt`.

(b) (*8 points*) Now we turn to characterizing the performance of this implementation. Fix the bug from part (a) and conduct an of the implementation (see homework instructions in the first page). Include the fixed code in `problem_1/p1_b.py`.

Provide a brief explanation of the performance you observe, and explain why the implementation does not achieve $\mathcal{O}(n^2)$ run time advertised for the G-S algorithm.

(c) (*10 points*) Correct and improve the run time of the G-S implementation and turn it in for auto-grading. It must be correct (always outputting a stable matching) and should run in the expected $\mathcal{O}(n^2)$ time. Provide a description of the optimizations you implemented. Additionally, provide an empirical performance analysis of your implementation in the same environment (same system and configuration) that you performed the earlier analysis. Include your new implementation in `problem_1/p1_c.py`.

Code input and output format. The input to the algorithm is a file to read from. The first line contains an integer n, which is the number of medical students and hospitals. This is followed by $2n$ rows. Each row contains all the numbers from 1 to n in some order, separated by a single space character. The first n rows represent the preferences of the medical students. In a given row r, if an integer x is in the i-th position, it means that Hospital x has rank i in Student r's preferences. The next n rows represent the preferences of the hospitals, and their encoding is analogous to the one for the students. We include an example below.

```
2
0 1
1 0
0 1
1 0
```

In the example, the first row tells us that there are 2 hospitals and 2 students. The next row indicates that student 0 prefers Hospital 0 over Hospital 1. The row after shows that student 1's first preference is Hospital 1, and their second preference is Hospital 0. The following 2 rows indicate that Hospital 0 ranked Student 0 first, while Hospital 1 ranked Student 1 first.

The expected output is a Python dictionary mapping a student to the hospital it is assigned to. The keys and values are both expected to be integers. In the example above, the expected output would be {0:0, 1:1}.

Problem 2. In the *interval covering problem*, one is given a time interval $[0, M]$ and a collection of closed subintervals $\mathcal{I} = \{[a_i, b_i] \mid i = 1, 2, \ldots, n\}$ whose union is $[0, M]$. (Interpret intervals as jobs and, interpret a_i and b_i as the *start time* and *finish time* of job $[a_i, b_i]$.) The goal is to find a subcollection $\mathcal{J} \subseteq \mathcal{I}$, containing as few intervals as possible, such that the union of the intervals in $\mathcal{J}$ is still equal to $[0, M]$.

You can assume that the numbers $M, a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$ are all non-negative integers, and that $a_i < b_i$ for all i. You should assume that the list of intervals given in the problem input could be arbitrarily ordered; in other words, don't make an assumption that the input to the algorithm presents the intervals sorted according to any particular criterion.

(*20 points*) Below we have listed four greedy algorithms for this problem. At least one of them is correct, and at least one is incorrect. For *every* incorrect algorithm, provide an example of an input instance such that the algorithm outputs an incorrect answer in your write-up. (Either a set of intervals that fails to cover $[0, M]$, or a set that covers $[0, M]$ but has a greater number of intervals than necessary.) For *every* correct algorithm, write, "The algorithm is correct." For *at least one* of the correct algorithms, implement the algorithm to run in $O(n \log n)$; that is $\Theta(n \log n)$ or faster. In the write-up, indicate which algorithm you implemented and briefly describe your implementation strategy. Include your implementation in `problem_2/p2.py`.

(a) **Select Latest Finish Time:** Initialize $\mathcal{J} = \emptyset$. Until the intervals in $\mathcal{J}$ cover all of $[0, M]$, repeat the following loop: find the maximum $c \in [0, M]$ such that $[0, c]$ is covered by the intervals in $\mathcal{J}$, or if there is no such c then set $c = 0$; of all the intervals containing c select one whose finish time is as late as possible and insert it into J.

(b) **Select Longest Interval:** Initialize $\mathcal{J} = \emptyset$. Until the intervals in $\mathcal{J}$ cover all of $[0, M]$, repeat the following loop: among all the intervals in $\mathcal{I}$ that are not already contained in the union of the intervals in $\mathcal{J}$, select one that is as long as possible (i.e., that maximizes $b_i - a_i$) and add it to $\mathcal{J}$.

(c) **Delete Earliest Redundant Interval:** Initialize $\mathcal{J} = \mathcal{I}$. While there exists a redundant interval in $\mathcal{J}$ — one that is contained in the union of the other intervals in $\mathcal{J}$ — find the redundant interval in $\mathcal{J}$ with the earliest finish time and remove it from $\mathcal{J}$.

(d) **Delete Latest Redundant Interval:** Initialize $\mathcal{J} = \mathcal{I}$. While there exists a redundant interval in $\mathcal{J}$ — one that is contained in the union of the other intervals in $\mathcal{J}$ — find the redundant interval in $\mathcal{J}$ with the latest finish time and remove it from $\mathcal{J}$.

A note about tie-breaking: When implementing any of the algorithms above one must choose a tie-breaking rule. In other words, when more than one interval meets the criterion defined in the algorithm specification (such as the latest finishing redundant interval) one must specify which of the eligible intervals is chosen. In the context of this problem, if you are asserting that an algorithm is correct, it should mean that you believe the algorithm gives the correct answer on every input instance *no matter how the tie-breaking rule is implemented;* that is what your proof of correctness should show. When you assert an algorithm is incorrect, for full credit you should supply an input instance that leads to an incorrect output *no matter how the tie-breaking rule is implemented.* However, significant partial credit will be awarded for providing an input instance

that leads to an incorrect output *for some choice of tie-breaking rule*, though not necessarily for every tie-breaking rule.

Code input and output format. The input to the algorithm is an integer M and a list of lists with two integers, representing the intervals $[a_i, b_i]$. The output should be a list of lists of pairs of integers, which are a subset of the input list.

Problem 3. We consider how to count the number of *large* inversions between two rankings. Consider a set N with size n. A ranking is a 1-1 function rank : $N \to [1..n]$. In other words, rank maps each item in N to a unique integer, and therefore represents an ordering of the N items where lower integer values indicate higher rankings.

As described in the Kleinberg-Tardos textbook, Section 5.3, we can compare two rankings, rank_i and rank_j, by counting the number of inversions. We label each $x \in N$ with $\text{rank}_i(x)$, hence making $\text{rank}_i = [1, \ldots, n]$. Then, we define rank_j in terms of rank_i and count the number of inversions in rank_j.

Suppose now we want to consider large inversions, where we define a threshold $0 < \delta < n$ and let $\text{NLI}_{\delta,i,j}$ equal the number of pairs $x, y \in [1..n]$ such that $x < y$ and $\text{rank}_j(x) > \text{rank}_j(y) + \delta$. In other words, we only consider inversions large if their ranking differs by at least δ.

(a) (*6 points*) Design and implement a $\Theta(n^2)$ algorithm that iterates over every pair of points and determines $\text{NLI}_{\delta,i,j}$ given two rankings rank_i and rank_j and a threshold δ. Describe the algorithm in the write-up, and include your implementation in `problem_3/p3_a.py`.

(b) (*14 points*) Design and implement a $\Theta(n \log n)$ algorithm to calculate $\text{NLI}_{\delta,i,j}$. Include your implementation in `problem_3/p3_b.py` and explain your algorithm in the write-up. Using your local experimental system perform an empirical performance analysis of both implementations (see homework instructions in the first page) and determine for what n does the asymptotically faster implementation dominates.

Code input and output format. The input is a two line file to read from, with the first line being n the second line being an ordering of the integers 1 to n, representing rank_j. An example is shown below.

```
3
1 3 2
```

In this example, given $N = \{x_1, x_2, x_3\}$, we could have that:

$\text{rank}_i(x_1) = 1; \text{rank}_i(x_2) = 2; \text{rank}_i(x_3) = 3$ and $\text{rank}_j(x_1) = 1; \text{rank}_j(x_2) = 3; \text{rank}_j(x_3) = 2$.

The output should be an integer counting the number of large inversions.

Problem 4. Given a sequence of integers $a_1, \ldots, a_n$ we want to find the number of times the most frequent pairwise difference appears. Let $\delta_{i,j} = a_i - a_j$ for $i \neq j$ in the list. Suppose you know one of the modes (i.e. the most frequent value) of the list of all the $\delta_{i,j}$'s is δ_{mode}. We want to count for how many values $\delta_{i,j} = \delta_{mode}$.

(a) (*8 points*) Design and implement a $\Theta(n \log n)$ algorithm that given a sequence of integers $a_1, \ldots, a_n$ and δ_{mode} as described above, outputs the frequency of δ_{mode}. Include your implementation in `problem_4/p4_a.py`.

(b) (*7 points*) Design and implement a $\Theta(n)$ algorithm that given a sequence of integers $a_1, \ldots, a_n$ and δ_{mode} as described above, outputs the frequency of δ_{mode}. Include your implementation in `problem_4/p4_b.py`.

(c) (*5 points*) Provide a comparison of the empirical performance analysis of your algorithms from parts (a) and (b). There is a clear performance advantage of using the algorithm from part (b) over the one from part (a). Is there any trade-off? ***Hint:*** *Consider the space complexity of the algorithms, which is the space an algorithm takes to execute.*

Note that the mode can be negative. If $\delta_{i,j}$ is a mode of the list, then $-\delta_{i,j} = \delta_{j,i}$ will be a mode too. The input to the function will be one of these modes, which can be positive, negative, or 0.

Code input and output format. The input will be a Python list integers and an integer representing δ_{mode}. The output should be an integer counting the number of pairs $i \neq j$ for which $\delta_{i,j} = \delta_{mode}$.

Problem 5. Eman baked some cookies for her friends, and asked Avital to help by distributing them among their friends. Their group of friends is very close-knit, so they all live in a building together, called "The Building".

The architect of "The Building" designed the property such that every floor is a very long corridor. To one side of the corridor are all the apartments in that floor, with the distance between every pair of consecutive doors being always the same. The architect is not very fond of elevators, so to the other side of the corridor, there is a pair of staircases in front of every apartment door, leading to all the floors in the building. Apartments are labeled first by floor and then by index in the floor, hence $apt_{(12,21)}$ is the 21st apartment in the 12 floor.

Avital is not a big fan of walking, so as she delivers the cookies, she may give some extra cookies to her friends, and ask them to distribute them to some other friends. All the friends may subsequently divide their cookies and ask other friends to help with the deliveries too.

Avital wants to minimize everyone's total number of steps. Assume that the distance between an apartment $apt_{(f,i)}$ and $apt_{(f,i+1)}$ is the same as that between $apt_{(f,i)}$ and $apt_{(f+1,i)}$.

Assume the friends are distributed across n apartments, where each apartment, including Eman's, has $p > n$ friends currently hanging out. Finally, assume that Avital starts at apartment $apt_{(1,1)}$, where Eman lives, and she can recruit some of the friends currently there.

(*20 points*) Design and implement an algorithm that given a list of apartments where the friends live, returns pairs (a, b), where friends from apartment a take the cookies to the friends from apartment b, that minimize the total number of steps. Your algorithm should run in $O(n^2 \log n)$. Describe your algorithm in the write-up and include your implementation in `problem_5/p5.py`.

Code input and output format. The input will be a Python list containing the apartments of Eman's friends, where $apt_{(f,i)}$ will be represented as the tuple (f, i). The output should be a list of apartment tuples. For example, if the input was $[(1, 2), (2, 1)]$, the output would be $[((1, 1), (1, 2)), ((1, 1), (2, 1))]$.

Hint: The distance between apartments is the L_1 distance or Manhattan distance:

$$\text{dist}(\text{apt}_{x_1,y_1}, \text{apt}_{x_2,y_2}) = |x_1 - x_2| + |y_1 - y_2|.$$

Algorithms for Applications	HW1: Challenge Problem
CS 5112 Fall 2023	**Due: September 26, 11:59pm ET**

Instructions: Challenge problems are, as the term indicates, *challenging*. They do not count for the homework score (90% of your course grade); instead, they are considered separately as extra credit over your course grade (additional 15% in total, 3.75% per assignment).

Questions about challenge problems will have lowest priority in office hours, and we do not provide assistance beyond a few hints to help you know whether you are on the right track.

Submission Instructions: You can choose not to hand a submission, but we encourage everyone to attempt the challenge problem. If you solve it, please hand in your answers (both, the coding and the write-up) through Gradescope, along with the rest of your assignment.

Academic Integrity and Collaboration Policy: The same guidelines apply to challenge problems as for the regular homework problems.

A closer look at Integer Multiplication

Given two n-bit integers (represented as a list of binary values, i.e. base 2), $x, y \in \{0,1\}^n$, implement three approaches to computing their product $x \cdot y$ and perform an empirical performance analysis of the algorithms.

(*2 points*) The first approach will be a reference implementation. Compute the product by converting from their binary representation to an int, then multiplying. Please provide a brief description of your implementation here.

(*5 points*) For the second approach, we would like you to implement the Karatsuba algorithm, described in section 5.5 of the Algorithm Design textbook. We also recommend the Wikipedia article for further implementation hints. Please provide a brief description of your implementation here.

(*10 points*) The final algorithm to implement is the Fast Fourier Transform (FFT), described in section 5.6 of the Algorithm Design textbook. Please provide a brief description of your implementation here.

(*3 points*) Finally, provide an empirical performance analysis of the algorithms. Ensure that the analysis samples enough problem sizes (i.e. integer sizes n) such that the runtimes can be clearly differentiated.

Code input and output format. The input to the algorithms are two lists x and y of the same length, containing 0s and 1s (binary representation). The output should be a list of 0s and 1s as well.