CS 5112 Fall 2023	HW1: *Stable Matching, Greedy, Divide and Conquer*
Name: Shou-Kai Cheng	**Due: September 26, 11:59pm ET**
NetID: sc2745	
Collaborators: Shoudong Zhu (sz273), Zhiduo Xie (zx324)	

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.

Answer:

The Gale-Shapley algorithm's theoretical complexity is $\mathcal{O}(n^2)$. I already fixed the bug from part (a), but achieving this complexity requires an optimized implementation.

The performance issue in my implementation is primarily due to the fact that I am using a nested loop structure.

1. Nested Loops: My algorithm uses two nested loops. The outer loop iterates over hospitals, and the inner loop iterates over students' preferences. This results in a worst-case time complexity of $\mathcal{O}(n^2)$ because, in the worst scenario, each hospital may need to iterate through all student preferences.

2. Pop Operation: I am using the pop(0) operation to remove elements from the front of the hospital's preference list. This operation has a time complexity of O(n) because it requires shifting all remaining elements to fill the gap.

3. Index Operation: In each iteration, I use the index operation to find the index of a hospital in the student's preference list. This operation also has a time complexity of O(n) because it may need to traverse the entire list.

My implementation, while correct in terms of matching stability, contains suboptimal operations like pop(0) and index, which increase the runtime.

To achieve the theoretical $\mathcal{O}(n^2)$ runtime, I would need to optimize the algorithm further. This optimization typically involves data structures like arrays or dictionaries to efficiently track and update preferences, eliminating the need for nested loops and costly operations.

In summary, my fixed implementation provides stable matchings, but it does not achieve $\mathcal{O}(n^2)$ runtime due to suboptimal operations and nested loops. Achieving the theoretical runtime would require further optimization of the algorithm. Please see part (c).

(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`.

Answer:

My optimization implementation strategy:

1. Doctor Preference Order:

- While reading the input, you not only store the preference list of each doctor but also create a doctor_pref_order list. This list maps each hospital to its rank in the doctor's preference list, enabling $\mathcal{O}(1)$ time complexity for preference lookups, which is crucial for maintaining the overall $\mathcal{O}(n^2)$ complexity.

2. Next to Propose:

- You use the next_to_propose array to keep track of the next doctor that each hospital needs to propose to. This eliminates the need to traverse the hospital's preference list from the beginning in each iteration.

3. Free Hospitals Queue:

- The algorithm maintains a list of free_hospitals, ensuring that only hospitals that haven't been matched yet, or those that are freed up in the current iteration, propose to doctors.

4. Pairs Dictionary:

- The pairs dictionary is used to keep track of the current matches between doctors and hospitals, allowing $\mathcal{O}(1)$ time complexity for both insertions and lookups.

Overall, these optimizations contribute to achieving the expected $\mathcal{O}(n^2)$ runtime while still ensuring that the implementation provides stable matchings.

Empirical Performance Analysis:

To verify the algorithm's correctness and performance, I test it with different scenarios, different size, and etc. and compare p1_b and p1_c. As we can see from the graph below (Figure 1), the algorithm of p1_c is much faster than p1_b, as the input size n increasing.

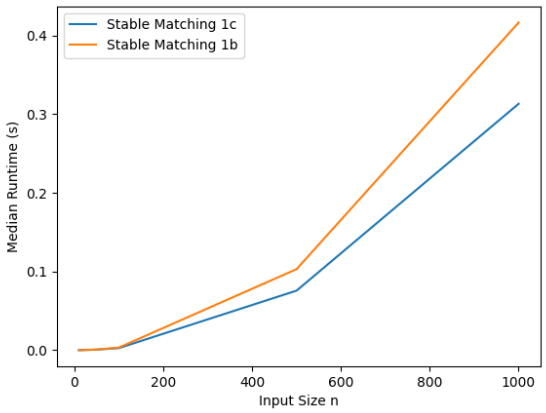

Figure 1: P1 Empirical Analysis

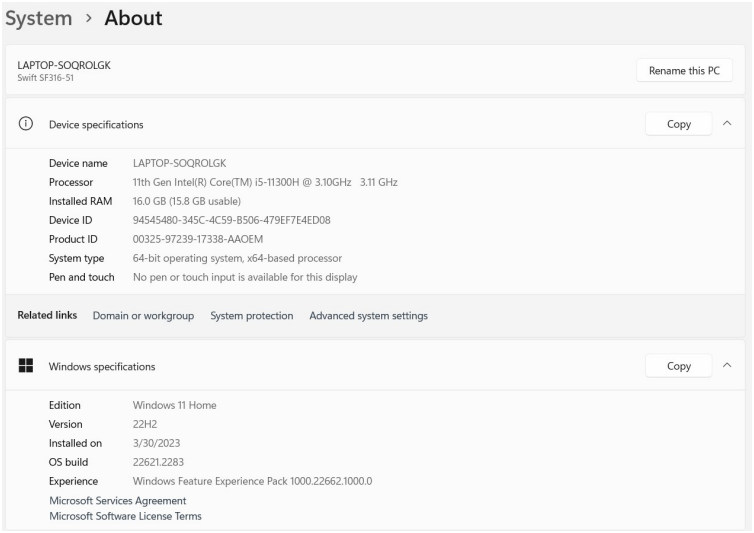

Figure 2: Environment

I also provide the information about environment (CPU, operating system and version, amount of memory) I did the testing on below (Figure 2).

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.

Answer: This algorithm is correct. It ensures that the selected interval has the latest finish time among those containing point c, which helps minimize the number of intervals needed to cover [0, M]. This algorithm runs in O(n log n) time complexity.

(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 $\mathcal{J}$.

Answer: This algorithm is incorrect. It selects the longest interval without considering its position in the timeline. This can lead to suboptimal solutions where intervals are selected that don't contribute effectively to covering [0, M]. You can provide an example where this algorithm fails to find the optimal solution.

Consider intervals [0, 3], [1, 4], [4, 5] and M=5. The algorithm will first select [0, 3] as it's the longest, but then there's no way to cover [3, 5] with a single interval, resulting in an incorrect solution.

(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 $\mathcal{J}$.

Answer: This algorithm is correct. It starts with all intervals in J and iteratively removes redundant intervals. The resulting collection J contains the minimal number of intervals required to cover [0, M]. This algorithm runs in $\mathcal{O}(n^2)$ time complexity in the worst case.

(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 $\mathcal{J}$.

Answer: This algorithm is incorrect. It deletes the redundant interval with the latest finish time. However, removing an interval with a late finish time doesn't necessarily lead to a correct minimal cover. You can provide an example where this algorithm fails to find the optimal solution.

Answer: I implement Algorithm (a) - Select Latest Finish Time. Here's a brief description of the implementation strategy:

1. Sort the input intervals based on their ending points (finish times) in descending order. This sorting step ensures that intervals with later finish times come first.

2. Initialize an empty list called covering to store the selected intervals that will cover [0, M].

3. Initialize a variable c to 0, representing the current point that needs to be covered.

4 Enter a loop that continues until the entire interval [0, M] is covered.

5. Within the loop, search for the interval with the latest finish time $(selected_interval)$ among the sorted intervals that also contain the current point c. This is done by iterating through the sorted intervals and selecting the first interval that satisfies the condition start <= c.

6. If no such interval is found (i.e., $selected_interval$ remains None), it means that the entire [0, M] cannot be covered, and the function returns an empty list.

7. If a $selected_interval$ is found, add it to the covering list to include it in the cover.

8 Update the current point c to be the finish time of the selected interval, ensuring that it moves forward in the timeline.

9. Continue the loop until the entire [0, M] is covered or until the function returns an empty list if coverage is not possible.

This implementation follows the greedy strategy of selecting intervals with the latest finish times that also contain the current point c. It provides a correct solution to the interval covering problem using Algorithm (a).

In conclusion, the code implements the "Select Latest Finish Time" strategy, aligning with the principles of the greedy algorithm described in option (a), and it runs in O(n log n) time complexity, as required.

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.

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`.

Answer:

My implementation strategy:

1. Input Reading:

- The algorithm starts by reading an input file. The first line of the file contains the integer n, representing the number of elements in the ranking. The second line contains the ranking of the integers from 1 to n.

2. Initialization:

- A counter variable count is initialized to 0. This variable will be used to count the number of large inversions.

3. Iterating Over Pairs:

- The algorithm then iterates over every pair of points (i, j) such that i < j. For each pair, it calculates the gap j - i. If the gap is greater than the given threshold delta and the rank of the element at index i is greater than the rank of the element at index j, the counter count is incremented.

4. Result:

- After iterating over all pairs, the algorithm returns the counter count as the result, representing the number of large inversions in the ranking.

The time complexity of the algorithm is $\mathcal{O}(n^2)$ due to the nested loops iterating over every pair of elements in the ranking.

(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.

Answer:

My implementation strategy:

1. Input Reading:

- The algorithm begins by reading an input file where the first line contains the integer n representing the number of elements in the ranking, and the second line contains the ranking of integers from 1 to n.

2. Initialization:

- A counter variable count is initialized to 0 to count the number of large inversions. The array of rankings is enumerated to keep track of the original indices, which represent the ranki values.

3. Merge Sort:

- The algorithm uses a modified version of the merge sort to sort the array while counting the number of large inversions. This is done using two helper functions: merge_sort and merge_and_count.

- merge_sort: A standard implementation of merge sort that recursively divides the array into two halves, sorts them, and merges them.

- merge_and_count: This function is responsible for merging two sorted halves and counting the number of large inversions during the process.

4. Counting Large Inversions:

- During the merge step, for each element in the right half, the algorithm traverses the left half to count how many elements satisfy the large inversion condition, i.e., the gap between the original indices is greater than delta, and the rank value of the element in the right half is less than the corresponding element in the left half.

- This count is accumulated in the count variable.

5. Result:

- After the merge sort is completed, the algorithm returns the count as the number of large inversions.

Time Complexity:

This algorithm has a time complexity of $\Theta(n \log n)$ due to the merge sort, which is more efficient than the $\Theta(n^2)$ algorithm provided earlier. The implementation keeps track of large inversions efficiently while sorting the rankings.

Empirical Performance Analysis:

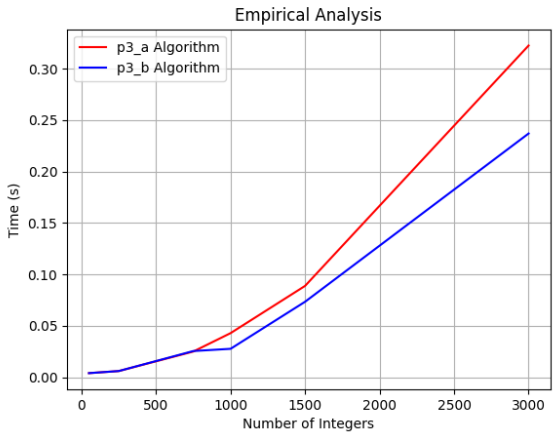

Figure 3: P3 Empirical Analysis

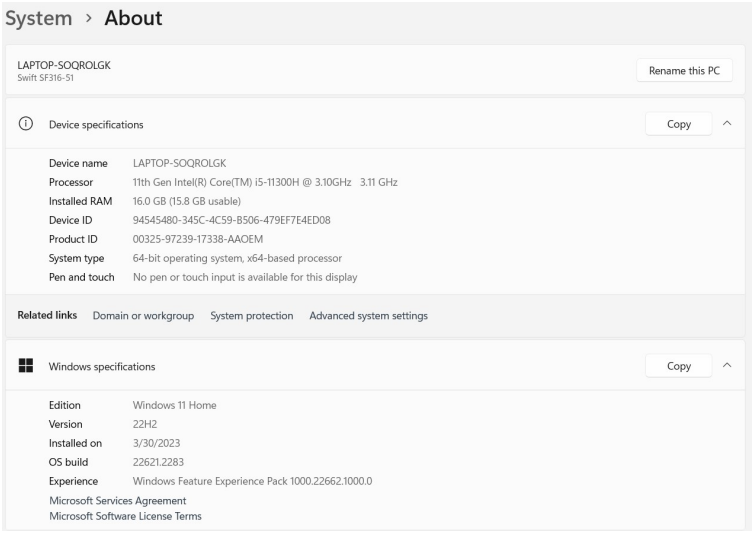

Figure 4: Environment

To perform an empirical performance analysis, I generate test cases for different values of n, measure the performance of both implementations, and determine the point at which the p3_b ($\Theta(n \log n)$) implementation becomes more efficient than the p3_a ($\Theta(n^2)$) implementation in practice. As we can see from the graph above (Figure 3), the algorithm of p3_b is much faster than p1_a, as the input size n increasing.

I also provide the information about environment (CPU, operating system and version, amount of memory) I did the testing on above (Figure 4).

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`.

Answer:

My implementation strategy:

1. Sorting Step: The given values list is sorted initially, which takes O(n log n) time.

2. Binary Search: For each element in values, two binary searches are performed to find the left end and the right end of the potential difference, and then the count of such differences is calculated.

The algorithm efficiently counts the frequency of mode while taking advantage of the sorted input list. The time complexity of this algorithm is $\Theta(n \log n)$ due to the sorting step.

(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`.

Answer:

My implementation strategy:

1. Hashmap Creation: The algorithm creates a hashmap from the input values list where the key is the value from the list and the value is its frequency in the list.

2. Counting: For each element in values, it calculates the complement by adding d_mode and checks if the complement exists in the hashmap to update the result.

3. Special Case Handling: If d_mode is 0, the algorithm subtracts 1 from the result for every found complement.

This algorithm efficiently counts the frequency of mode by using a hashmap to store the frequency of each value in the input list and then checking for pairs that form mode. The time complexity of this algorithm is $\Theta(n)$ because it iterates through the input list twice but performs constant-time operations for each element.

(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.*

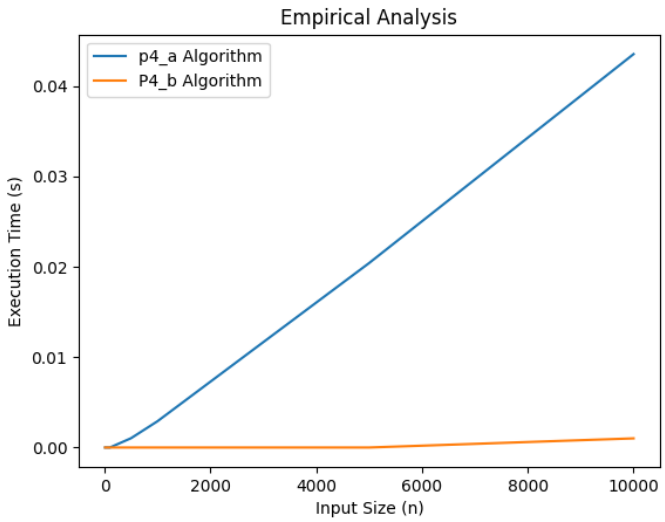

Figure 5: P4 Empirical Analysis

Answer:

Trade-offs:

1. most_frequent_difference_a (Time Complexity: (n log n)) Trade-Offs: This algorithm is time-efficient when dealing with unsorted input but does not use any extra space except for the input and some constant space variables.

2. most_frequent_difference_b (Time Complexity: (n)) Trade-Offs: While this approach runs in linear time, it has additional space complexity due to the use of a hashmap to store frequencies, making it less space-efficient than the first approach.

Empirical Performance Analysis:

To perform an empirical performance analysis, I generate test cases with varying input sizes (n), measure the performance of the algorithm for different values of n, and determine how the algorithm's runtime scales with the input size. The time complexity of p4_a is $\Theta(n \log n)$ and the time complexity of p4_b is $\Theta(n)$. As we can see from the graph above (Figure 5), the algorithm of p4_b is much faster than p4_a, as the input size n increasing.

I also provide the information about environment (CPU, operating system and version, amount of memory) I did the testing on below (Figure 6).

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.

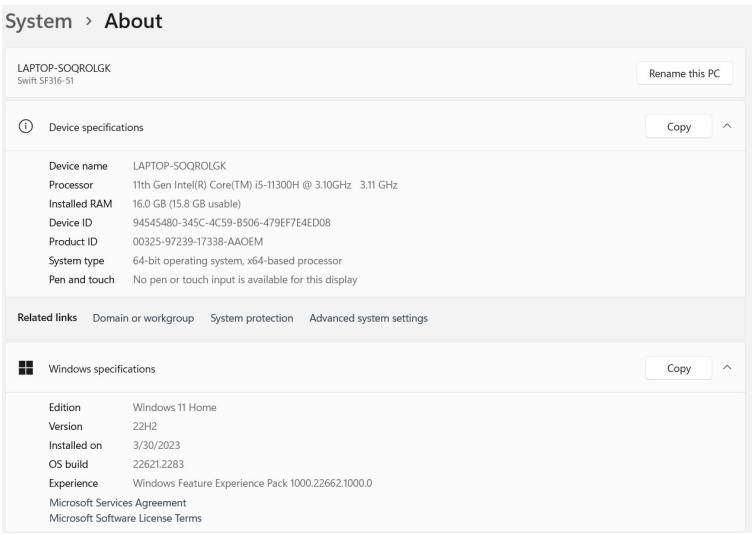

Figure 6: Environment

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`.

Answer:

My implementation strategy:

1. Initialization:

- A min-heap heap is initialized to keep track of the distances between apartments.

- A list visited is used to store the visited apartment pairs.

- A set unassigned contains all the apartments that are yet to be visited.

2. Calculate Initial Distances:

- The algorithm calculates the Manhattan distances between Eman's apartment (located at (1, 1)) and all other apartments, and pushes them into the min-heap along with the apartment pairs.

3. Apartment Pair Selection and Visitation:

- The algorithm pops the apartment pair with the smallest distance from the heap.

- If the next room is unassigned, it is added to the visited list and removed from the unassigned set.

- The distances between this newly visited apartment and all other unassigned apartments are calculated and pushed into the min-heap.

4. Termination:

- The algorithm continues the process until there are no unassigned apartments left.

5. Result:

- The algorithm returns the visited list containing pairs of apartments representing the distribution path.

Empirical Performance Analysis:

1. Time Complexity:

- The main loop of the algorithm runs until all apartments are assigned, which will be n iterations.

- In each iteration, the algorithm calculates the Manhattan distance for each unassigned apartment and pushes it into the heap, which takes O(n log n) time (n for iterating over unassigned apartments and log n for each heappush operation).

- Therefore, the overall time complexity of the algorithm is $\mathcal{O}(n^2 logn)$.

2. Space Complexity:

- The space complexity of the algorithm is primarily influenced by the heap, visited, and unassigned data structures.

- The heap can store up to n distances at any time, visited will eventually store n apartment pairs, and unassigned will also store n apartments.

- Hence, the space complexity of the algorithm is O(n).

CS 5112 Fall 2023	HW1: Challenge Problem
Name: Shou-Kai Cheng	**Due: September 26, 11:59pm ET**
NetID: sc2745	
Collaborators: Shoudong Zhu (sz273), Zhiduo Xie (zx324)	

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.

Answer: My implementation strategy:

1. The binary_to_decimal function takes a list of binary values (0s and 1s) and converts them into a decimal integer. It achieves this by joining the binary values as a string and using the int function with base 2 to convert the binary string to an integer.

2. The decimal_to_binary function takes a decimal integer and converts it into a binary representation. It uses the bin function to get the binary string representation and converts each character in the string to an integer.

3. The reference_multiply function receives two n-bit binary numbers, x and y. It first converts these binary numbers to decimal integers using the binary_to_decimal function. Then, it performs integer multiplication on these decimal values and stores the result in the ans variable. Finally, it converts the decimal result back to a binary representation using the decimal_to_binary function and returns the binary list.

(*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. Please provide a brief description of your implementation here.

Answer:

My implementation strategy:

The karatsuba function is a recursive implementation of the Karatsuba multiplication algorithm. It follows the steps outlined in the algorithm:

1. Base Case: If both x and y are single-digit numbers, it performs a simple multiplication and returns the result as a binary list.

2. Calculate the size of the numbers, n, and the midpoint, m, to split the binary representations into two parts.

3. Split x and y into four parts: a, b, c, and d.

4. Recursively apply the Karatsuba algorithm to calculate ac, bd, and ad$_{p}lus_{b}c$.

5. Calculate the final result using the derived values, following the Karatsuba formula.

6. Convert the result back to a binary list.

(*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.

Answer: Please answer 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.

Answer: Please answer here.