CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, APRIL 22, 2022 AT 11:30 PM VIA GRADESCOPE

CADV(WYZ5RGE)

Collaboration Policy: You are encouraged to collaborate with up to 3 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite by naming the book etc. or listing a website's URL. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

Collaborators: jhs8cue, spj6s

Sources: Cormen, et al, Introduction to Algorithms. *(add others here)*

PROBLEM 1 *Backpacking*

You are going on a backpacking trip through Shenandoah National park with your friend. You two have just completed the packing list, and you need to bring n items in total, with the weights of the items given by $W = (w_1, w_2, \ldots, w_n)$. You need to divide the items between the two of you such that the difference in weights is as small as possible. The total number of items that each of you must carry should differ by at most 1. Use dynamic programming to devise such an algorithm, and prove its correctness and running time. You may assume that M is the maximum weight of all the items (i.e., $\forall i,\ w_i \leq M$). The running time of your algorithm should be a polynomial function of n and M. The output should be the list of items that each will carry and the difference in weight.

Solution: This is similar to the gerrymandering algorithm. The algorithm will determine whether a certain distribution of the first i items in the list is valid by taking in an integer value representing the first i items, the number of items assigned to the first backpack j, and the weight difference between backpacks k. Just like the gerrymandering algorithm, we will store the boolean values for each combination of inputs in a 3D table. We can then backtrack by searching the table for a "true" value for where j is equal to the total number of items n that yields the smallest difference in weight between the two backpacks, and the difference in number of items assigned $n - k$ is k or $k + 1$ (the difference in items assigned is at most 1). Similarly to the gerrymandering algorithm, the running time depends on its nested loops, two of which depend on input size n and one depending on input value m times n, the running time is $\Theta(n3m)$. Because m is a value and not a size, the running time is pseudo-polynomial.

PROBLEM 2 *Course Scheduling*

The university registrar needs your help in assigning classrooms to courses for the fall semester. You are given a list of n courses, and for each course $1 \leq i \leq n$, you have its start time s_i and end time e_i. Give an $O(n \log n)$ algorithm that finds an assignment of courses to classrooms which minimizes the *total number* of classrooms required. Each classroom can be used for at most one course at any given time. Prove both the correctness and running time of your algorithm.

Solution: This algorithm is essentially the interval scheduling algorithm, but for several schedules. We can imagine each classroom having a "schedule" of classes to host, thus the optimal solution to this problem is to repeatedly find the collection of optimal schedules until all classes have been assigned to a classroom. This can be done by running a modified version of the greedy algorithm for interval scheduling in class. First start by creating a min heap of size n, as the worst case is that every class conflicts and we need n classrooms for n classes. This min heap represents which classrooms are free, and the root will be the smallest numbered classroom that is available. Iterate through the time interval, minute by minute. At any given time, if a class is supposed to start, peek at the min heap, assign the class to the classroom corresponding to that number, then remove the smallest value from the min heap. Classroom numbers can be assigned to classes by adding them to a list/set data structure that is labeled with the classroom number. If a class ends at a given time, add its classroom number back to the min heap, signifying that that classroom is free again. When we finish iterating through the time interval, we will be left with several schedules in the form of sets, corresponding to the classroom numbered with the same label, all in sorted order.

We can use an exchange argument to prove correctness for this algorithm. Consider a set of classes where all are not conflicting except for the two earliest classes, a and a'. The optimal solution determined by this algorithm is to have all of the classes assigned to classroom 1, which includes a. Then class a' is assigned to classroom 2. Assume toward a contradiction that the solution assigning class a to classroom 1 is not optimal. Since a doesn't belong to classroom 1, then it is assigned to classroom 2, and since a' conflicts with this new assignment but does not conflict with any other events, it can take the new vacancy left in classroom 1's schedule. This solution is neither better nor worse than our original solution. Because our goal is to simply minimize classroom use and not to evenly distribute classes among classrooms, our greedy algorithm will always find some solution such that the fewest classrooms are used.

Making a min heap of size n will have time complexity n. When a class begins, we perform a heap removal and an array/list insertion. When a class ends, we perform a heap insertion. Thus the worst case at any given time in the time interval is when a class ends at the same time another class ends, meaning the worst case time complexity at any given time is $\Theta(2logn + 1)$. Since we are iterating through the time interval, we are doing this n times, thus the overall time complexity of the iteration is $\Theta(n(2logn + 1) = \Theta(nlogn)$. Including the time to sort classes, as well as the time to create the min heap, our overall time complexity is $\Theta(nlogn + n + nlogn) = \Theta(nlogn)$

PROBLEM 3 *Ubering in Florin*

After the adventures with Westley and Buttercup in *The Princess Bride*, Inigo decides to turn down the "Dread Pirate Roberts" title and to instead moonlight as the sole Uber driver in Florin. He usually works after large kingdom-wide festivities at the castle and takes everyone home after the final dance. Unfortunately, since his horse can only carry one person at a time, he must take each guest home and then return to the castle to pick up the next guest.

There are n guests at the party, guests $1, 2, ..., n$. Since it's a small kingdom, Inigo knows the destinations of each party guest, $d_1, d_2, ..., d_n$ respectively, and he knows the distance to each guest's destination. He knows that it will take t_i time to take guest i home and return for the next guest. Some guests, however, are very generous and will leave bigger tips than others; let T_i be the tip Inigo will receive from guest i when they are safely at home. Assume that guests are willing to wait after the party for Inigo, and that he can take guests home in any order he wants. Based on the order he chooses to fulfill the Uber requests, let D_i be the time he returns from dropping off guest i. Devise a greedy algorithm that helps Inigo pick an Uber schedule that minimizes the quantity:

$$\sum_{i=1}^{n} T_i \cdot D_i.$$

In other words, he wants to take the large tippers the fastest, but also want to take into consideration the travel time for each guest. Prove the correctness of your algorithm. (Hint: think about a property that is true about an optimal solution.)

Solution: Inigo has a preference for passengers first by tip amount, then by distance to their destination. This is shown in the cost function described in the assignment description:

$$\sum_{i=1}^{n} T_i * D_i$$

The greedy choice would then be to choose whichever passenger for whom the round trip to their house earns the most money per minute spent driving. Since Inigo knows how long it takes to get to each destination, we can calculate the dollar per minute value by dividing each passenger's tip by the the time it takes to reach their destination. This then prioritizes passengers with larger tips while also taking into account how far away their home is. Each dollar per minute value can then be stored in a matrix of size n, with a given value at index i corresponding to passenger i. This array can then be sorted in ascending order so that Inigo can take the passenger at the front of the array, iterating after each ride.

Consider a proof by cases by running this algorithm on a set of two passengers, where passenger 1 lives c_1 minutes away and tips c_2 dollars, and passenger 2 lives c_2 minutes away and tips c_1 dollars, and $c_1 > c_2$. We will consider the two cases where one passenger is chosen over the other. In the case that passenger 1 is taken first, the cost is $c_1c_2 + (c_1 + c_2)c_1 = c_1^2 + c_2c_1$. In the case that passenger 2 is taken first, the cost is $c_2c_1 + (c_2 + c_1)c_2 = c_2^2 + c_1c_2$. Since $c_1 > c_2$, the optimal choice is case 2, where the passenger who lives farther away but pays a larger tip is taken first. This mathematical relationship holds for any combination of values, as long as passenger 1 lives closer and tips less, and passenger 2 lives farther away but tips more. This choice minimizes $\sum_{i=1}^{n} T_i * D_i as \mathrm{T}_i$ doesn't change for each passenger but D_i does depending on the order they are taken in, thus taking passengers with the highest tips first results in less elapsed time, minimizing the function. Because our algorithm chooses passengers who pay relatively more, our algorithm is correct.

The running time of our algorithm is dependent on the creation of an n sized array and then sorting said array. Everything else is simple arithmetic and array accesses, so our algorithm running time is $\Theta(n + n \log n) = \Theta(n \log n)$

PROBLEM 4 *Gradescope Submission*

Submit a version of this `.tex` file to Gradescope with your solutions added, along with the compiled PDF. You should only submit your `.pdf` and `.tex` files.