CS4102 - ALGORITHMS - SPRING 2022

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

CADV(MST3K)

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: list collaborators's computing IDs

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:*Lets use a algorithm similar to that of gerrymandering but also record backtracking information. First, lets define our "S(j, k, x)" function. Like Gerrymandering, lets let j be the number of items we've assigned, k be the number of items (out of j) that are assigned to the first backpack, and x be the mass of backpack 1. Unlike the Gerrymandering problem, we have all the information we need in the weight of each object, the actual details of the object are irrelevant. In Gerrymandering, we had both the population of the precinct and the number of R voters, here we have only the weights. So we don't need to keep track of the contents of the second bag; we know it implicitly by knowing the mass of bag 1. The algorithm will be defined as such:*

```
Initialize S(0, 0, 0) = True
initialize BackTrackMatrix // stores the coordinates that we got our true from

if ( size(W) is odd )
    oddCheck = size(W)/2 - 1
else
    oddCheck = size(W)

for j = 1,...,size(W):
    for k = 1,...,min(j, oddCheck):
        for x = 0,...,sum(W):
            S(j, k, x) = S(j - 1, k - 1, x - $w_j$) V S(j - 1, k, x)
            if ( S(j - 1, k - 1, x - $w_j$) )
                BackTrackMatrix(j, k, x) = [j - 1, k - 1, x - $w_j$]
            if ( S(j - 1, k, x) )
                BackTrackMatrix(j, k, x) = [j - 1, k - 1, x]

Search for True entry at S(size(W), oddCheck, any x)
```

```
bestWeight = INFIN
bestS

for x = sum(W),...,0
    if ( S( size(W), oddCheck, x) )
        if ( sum(W) - x < bestWeightDiff )
            bestWeight = sum(W) - x
            bestS = [size(W), oddCheck, x]

while(bestS != [0, 0, 0])
    print(bestS)
    bestS = BackTrackMatrix[bestS]
```

Now for a runtime analysis. The first for loop runs for size(W) (so n), the second for loop runs for a factor of size(W) (so n), and the last for loop runs for sum(W) (so m). Because of that, our runtime is $\Theta(n^2 * m)$. Unlike Gerrymandering, our third/fourth loops are not dependent on n, we do not multiply nm, instead we can just use the sum(W). This is because in Gerrymandering we had information on the size of each precinct, because we don't have that, we can get away with a $\Theta(n^2 * m)$ runtime. The other parts of the algorithm have negligible additions to the runtime. Namely the check and while loop at the end have additions of $\Theta(m)$ and $\Theta(n)$, respectively.

Now for a proof of correctness. I will proceed via proof by induction. The base case is S(0, 0, 0) this has a True, indicating a valid state by definition. The next state that we check is the allocation of the first weighted object. This allocation is only valid if there was a way to get to the prior world state without allocating w_j because in the current step we are allocating it to a prior world state. And the current world state is only a valid world state if the prior world states were valid as well. This ensures that only if there is a valid prior world state, will be able to get to the current world state and the lack of a valid prior world will ensure that the current world state is impossible to attain. Additionally, this method gives us the optimal division of weights into each backpack because prior to returning our answer, we check and find the valid solution with minimum weight difference. Note that we use oddCheck to ensure that we are searching for an answer at the valid difference level (number of objects differs by at most 1).

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 is very similar to the interval scheduling problem except for the fact that instead of us finding one non-intersecting set, we need to find multiple non-intersecting sets. More specifically, we need to find horizontal sets of vectors such that each horizontal set consists of a classroom.***

If we assume that the data is given to us in the same format as in lecture, a series of pairs consisting of start and end times, then we can use the following algorithm. We begin by sorting each value in the array by the finish time and then adopt the first member of the list into one of our horizontal sets. We then add to this set the next class that doesn't intersect with it. We repeat this process until we are out of classes to select. The total number of sets is the number of classrooms we need.

An algorithm to find a horizontal set is below:

```
horizontalSet(startEndTimes)
```

```
    startEndTimes.sort(key = 1)
    A = startEndTimes[0];
    for i = 1 to startEndTimes.length
        if ( startEndTimes[i][0] > A[0] )
            A.append(startEndTimes[i])
            A = startEndTimes[i]

    return A
```

Notice that if we had to run horizontalSet for each classroom, we would take n^2 time because in the worst case, we would need to iterate over every member of the input n times. But, there is a way to count the classrooms in $O(nlog(n))$ time. The way that we do this is that we create a priority queue to hold the classrooms that is sorted by earliest end time of classes first. We keep checking the top of the priority queue for the end time of latest class that it has and add classes accordingly. An algorithm that counts the number of classrooms (horizontal sets) needed is below:

```
numberOfClassrooms(StartEndTimes)
    startEndTimes.sort(key = 0) //note that we sort by START times instead of finish times
    pqClassrooms // priority queue of classrooms
    classroomCounter = 1
    new classRoom
    classRoom.add(startEndTimes[0])

    pqClassrooms.add(classRoom, classRoom[end][1]) // we add a class with the key being its fini

    for i = 1 to startEndTimes.length
        if ( startEndTimes[i][0] > pqClassrooms[0] ) // if the start time of the next class is a
            classRoom = pqClassrooms.pop(0)
            classRoom.add(startEndTimes[i])
            pqClassrooms.insert(classRoom, classRoom[end][1]

        else // otherwise we create a new classroom but do the same thing

            new classRoom
            classRoomCounter += 1
            classRoom.add(pqClassrooms[i])
            pqClassrooms.insert(classRoom, classRoom[end][1]
    }

    return classroomCounter
```

The reason that this takes $\Theta(nlog(n))$ time is because the initial sort takes n log(n) time, a given value for sorts. Then, the for loop runs n times, once for each class and makes a decision. But for either decision, we end up adding a value to the priority queue. Both the pop+insert and normal insert take log(n) time on behalf of the priority queue to take the new value back into the fold. Therefore, we take $\Theta(nlog(n))$ time.

Now for a proof of correctness. I will proceed via proof by induction. Assume the base case, that we have a set of valid classrooms in the priority queue is held prior to the for loop. Now, we have two cases, the next class, the one with the earliest start time, is either compatible with

the classroom with the earliest end time (the classroom at the beginning of the priority queue) or it isn't. In the case that it is compatible, we can simply add the class that we are assessing to the classroom, changing the classrooms key (time of latest ending class) and re-adding the class to the priority queue. In the case that the class is incompatible (the interesting case) we know that no other classroom is compatible with this class. This is because we know that the top of the priority queue holds the class with the "earliest" key and all other classes will have later ending classes. Because of this, we need to add another classroom to our list of classrooms. This concludes the proof of correctness.

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: Ithink juggling around the summation should reveal some interesting facts to us. Namely, lets look at the first two "steps" of the summation, where i = 1 and i = 2:

$$\sum_{i=1}^{2} T_i \cdot D_i = \tag{1}$$

$$T_1 * D_1 + \tag{2}$$

$$T_2 * (D_1 + D_2) \tag{3}$$

Here, we can assume that D_i is equal to $d_i * 2$. Lets observe the effect on the total summation that swapping T_2 and T_1 would have:

$$\sum_{i=1}^{2} T_i \cdot D_i = \tag{4}$$

$$T_2 * D_2 + \tag{5}$$

$$T_1 * (D_1 + D_2) \tag{6}$$

Now we have two sums that we can compare:

$$T_1 * D_1 + T_2 * D_1 + T_2 * D_2$$

and

$$T_2 * D_2 + T_1 * D_1 + T_1 * D_2$$

If we eliminate the common terms between them, we find that we are left with the following "representative" values for both summations:

$$T_2 * D_1$$

and

$$T_1 * D_2$$

Namely, we find that the difference between Summation 1 and Summation 2 is equal to $(T_2 * D_1) - (T_1 * D_2)$**. This indicates that we should swap customer 2 to be served before customer 1 only if:**

$$T_2 * D_1 > T_1 * D_2$$

If we use reverse cross-multiplication, we can manipulate the above condition to something a little more palatable:

$$T_2/D_2 > T_1/D_1$$

Therefore, we should swap customer 2 to be served customer 1 only if the above condition holds. But how can this information be used to order the entire series of customers through a greedy algorithm? How about instead of considering the condition in terms of a "swapping" condition, we use the condition as a ordering condition such that as long as the ordering is followed, we have an optimal solution.

I will now detail the algorithm here. The first thing we do is perform a T_i/D_i **operation for all i. Then we sort such that the largest ratio is first and we have Inigo serve the customer with the largest** T_i/D_i **first. Note that this ordering is a greedy algorithm because we have a subset that is the optimal solution and then we subsequently add to this subsolution via our greedy property (selecting the job with largest** T_i/D_i**) until we have included all customers.**

Now I will prove that this algorithm is correct. Suppose that we have an ordering of customers that is an optimal solution already. Suppose that we make a swap that serves an earlier customer later and a later customer earlier. This later customer will be called customer 2 and this earlier customer will be called customer 1. After performing the same summation difference operations detailed above, we are left with:

$$T_2 * D_1$$

and

$$T_1 * D_2$$

Because the pre-swap summation is an optimal solution, we know that $T_2 * D_1 <= T_1 * D_2$**. But, does this hold with our original expectation? If we perform reverse cross multiplication, we find that:**

$$T_2/D_2 \leq T_1/D_1$$

This holds with our original ordering principle that customers with larger T_i/D_i **must come before customers with lower** T_i/D_i**. And because we stated that customer 1 would have a larger**

T_i/D_i, and that was how our customer service was ordered, we can prove that our ordering demonstrates an optimal solution.

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.