CS4102 - ALGORITHMS - SPRING 2020

DUE SUNDAY, MARCH 5 AT 11P VIA COLLAB

H6(YL4DF)

Collaboration Policy: You are encouraged to collaborate with up to 4 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the comments at the top of the tex file. You **are** permitted to collaborate through online tools such as Google Docs, interactive whiteboards, Google Meet, Google Hangouts, Zoom, Skype, etc, however you **must** limit written/typed details to high-level algorithm design. Each person is responsible for taking those ideas and turning them into pesudocode and a writeup. Do **NOT** copy and paste from shared documents, which includes re-typing verbatim or trying to disguise text that you are essentially copying. Over-collaboration of that form is fairly easy to detect with plagiarism tools. **Do not seek published or online solutions, including pseudocode, for this assignment.** If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite them. 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: mw5ew

Sources: list any sources, e.g. Cormen, et al, Introduction to Algorithms

PROBLEM 1 *Covid Neutralizer*

The world is being overrun by a coronavirus! Stark industries has developed a Covid Defense System (CDS) to help protect us. The CDS continuously monitors all molecular particles across the globe to perfectly predict all possible Covid-19 infections.

Specifically, the CDS allows Stark Industries to predict that i days from now x_i people will become infected. The CDS has a covid-neutralizer that is able to stop an infection from entering a person (saving that person), but the device takes a lot of time to charge up. In general, charging the neutralizer for j days will allow it to protect d_j persons from infection.

Example: Suppose $(x_1, x_2, x_3, x_4) = (1, 10, 10, 1)$ and $(d_1, d_2, d_3, d_4) = (1, 2, 4, 8)$. The best solution is to use the neutralizer at times 3, 4 in order to save 5 people from being infected.

1. Construct an instance of the problem on which the following "greedy" algorithm returns the wrong answer:

   BADNEUTRALIZE$((x_1, x_2, x_3, \ldots, x_n), (d_1, d_2, d_3, , \ldots, d_n))$:

   Compute the smallest j such that $d_j \geq x_n$, Set $j = n$ if no such j exists

   Use the neutralizer at time n

   if $n > j$ then BADNEUTRALIZE$((x_1, \ldots, x_{n-j}), (d_1, ..., d_{n-j}))$

   Intuitively, the algorithm figures out how many days (j) are needed to save the most people in the last time slot. It neutralizes during that last time slot, and then accounts for the j days required to recharge for that last slot, and recursively considers the best solution for the smaller problem of size $n - j$.

   Suppose $(x_1, x_2, x_3, x_4) = (10, 10, 10, 2)$ and $(d_1, d_2, d_3, d_4) = (1, 2, 4, 8)$. The smallest j such that $d_j \geq x_n$ is $j = 2$. We use the neutralizer at time 4. Then we call BADNEUTRALIZE$((x_1, x_2), (d_1, d_2))$. Then such j does not exists, so we set $j = n = 2$ and use it at time 2. This means that the greedy algorithm tells use to use the neutralizer at time 2 and 4 to save 4 people. However, the optimal solution is to use the neutralizer at time 3 and 4 to save 5 people.

2. Given an array holding x_i and d_j, devise a dynamic programming algorithm that maximizes the number of people saved. Analyze the running time of your solution.

The pseudo-code for the dynamic programming is given below:
Initialize Memory S, days.
`DPNEUTRALIZE`(x, d):
    $S[0] = 0$
    For $i = 1$ to n:
        saved$= 0$
        For $j = 1$ to i:
            If saved $< S[i-j]$+min$(d[j], x[i])$:
                saved $= S[i-j]$+min$(d[j], x[i])$
                days$[i] = j$
        $S[i] =$saved
    return $S[n]$
The dynamic programming algorithm applies bottom-up strategy, in which we iteratively solve smallest to largest. We use S to store the solution for subproblems, i.e. the number of people saved, and use "days" for back tracking. The recursive structure is given by

$$S(n) = max[S(n-1) + min(d[1], x[n]), S(n-2) + min(d[2], x[n]), ..., S(0) + min(d[n], x[n])] \tag{1}$$

Here min$(d[j], x[n])$ means that we charge up j days to cure people from day n, and the number of people saved on day n depends on which number is smaller. We can get the total number of saved people by adding the number saved on day n with the number saved on previous days (already stored in S) and finding the maximum. In order to know which dates to use the neutralizer, we use the following pseudo-code:
`backTrack`$(days)$:
    dayList$=$ []
    $i = n$
    daysList.append(i)
    While $i > 0$:
        dayList.append$(i-$days$[i])$
        $i = i-$days$[i]$
    return daysList.reverse()
There are two for-loops in DP algorithm and both depends on n, so the running time is $\Theta(n^2)$. The backtrack process takes linear time. Thus, the total running time is $\Theta(n^2)$.

PROBLEM 2 *Crossing the Bridge*

n people need to cross a narrow rope bridge as quickly as possible, and each respective person crosses at speeds $s_1, s_2, ..., s_n$ (you can assume these are integers and are sorted in descending order). You must also follow these additional constraints:

1. It is nighttime and you only have a single flashlight. One requires the flashlight to cross the bridge.
2. A max of two people can cross the bridge together at one time (and they must have the flashlight).
3. The flashlight must be walked back and forth, it cannot be thrown, mailed, raven'd, etc.
4. A pair walking across together crosses at the speed of the slowest individual. They must stay together!

Describe a greedy algorithm that solves this problem optimally. State the runtime of your algorithm and prove your algorithm always returns the optimal solution. *NOTE: The obvious greedy algorithm does NOT work here. Be careful! This is more complicated than it appears.*
Algorithm:
Calculate each person's time by $t_i = \frac{1}{s_i}$. This time list should be already in increasing order.
Base Case: If $n = 2$, let them cross the bridge and no return is needed. If $n = 3$, let the fastest and slowest pair up and cross the bridge. The fastest returns and crosses with another person.
If $n \geq 4$, suppose $t_1 \leq t_2 \leq ... \leq t_{n-1} \leq t_n$, we compare $t_n + t_{n-1} + 2t_1$ with $t_n + 2t_2 + t_1$.

1. If $t_n + t_{n-1} + 2t_1 \geq t_n + 2t_2 + t_1$, we ask t_1 and t_2 to cross the bridge and let t_2 return with flashlight. Then we ask t_n and t_{n-1} cross and let t_1 return with flashlight. The total time is $t_n + 2t_2 + t_1$.
2. If $t_n + t_{n-1} + 2t_1 < t_n + 2t_2 + t_1$, we ask t_n cross with t_1 and t_1 returns. Then ask t_{n-1} cross with t_1 and t_1 returns. The total time is $t_n + t_{n-1} + 2t_1$.

Then we can recursively solve $n - 2$ person problem.
In each recursion we have 4 crossings and we call recursion $\frac{n}{2}$ times, so the running time is $\Theta(2n) = \Theta(n)$.
Proof. If $n = 2$, nothing need to be changed.
If $n = 3$, it is obvious that $2t_1 + t_2 + t_3$ is optimal, because exchanging will cause total time increase to $t_1 + 2t_2 + t_3$ and $t_1 + t_2 + 2t_3$.
If $n = 4$, our goal for each recursion is to take the smallest time to move t_n and t_{n-1} across the bridge. We still consider the two cases.

1. $t_n + t_{n-1} + 2t_1 \geq t_n + 2t_2 + t_1$. If we exchange t_1, t_2 for other pairs, the total time will only increase because $2t_2 + t_1$ is smallest already and thus $t_n + 2t_2 + t_1$ is optimal.
2. $t_n + t_{n-1} + 2t_1 < t_n + 2t_2 + t_1$. If we exchange t_1 with any other person, the time will only increase because $2t_1$ is already smallest and thus $t_n + t_{n-1} + 2t_1$ is optimal.

Since we guarantee each recursion case/step to be optimal, the algorithm is optimal. □

PROBLEM 3 *Frozen 3: Ubering in Arendelle*

After all of their adventures and in order to pick up some extra cash, Kristoff has decided to moonlight as the sole Uber driver in Arendelle with the help of his trusty reindeer Sven. They usually work after the large kingdom-wide festivities at the palace and take everyone home after the final dance. Unfortunately, since a reindeer can only carry one person at a time, they must take each guest home and then return to the palace to pick up the next guest.

There are n guests at the party, guests $1, 2, ..., n$. Since it's a small kingdom, Kristoff 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 Kristoff will receive from guest i when they are safely at home. Assume that guests are willing to wait after the party for Kristoff, and that Kristoff and Sven can take guests home in any order they want. Based on the order they choose to fulfill the Uber requests, let D_i be the time they return from dropping off guest i. Devise a greedy algorithm that helps Kristoff and Sven pick an Uber schedule that minimizes the quantity:

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

In other words, they want 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.)
Algorithm:
Sort $\frac{T_i}{t_i}$ by descending order. Kristoff and Sven pick up guest based on this order, i.e. pick up the guest with larger weighted tips as soon as possible.

Proof. We prove it by Exchange Argument.
Suppose our greedy algorithm gives us schedule G and there is a optimal schedule O.
For ease of proof, we claim that if G is not optimal, then there will be at least one adjacent pair of guests, namely, i and j, that are not in the same order of O. This claim can be proved by rearranging optimal schedule $O = (1, 2, 3, 4, 5, 6, ..)$. We can witness adjacent non-ordered pair when we swap any two number in the list, so claim is proved.
Let us say that G order j before i, so we have $\frac{T_j}{t_j} \geq \frac{T_i}{t_i}$. Optimal schedule O order i before j.
Assume it takes C to complete all prior orders. In G, the cost of (j, i) is $T_j \cdot (C + t_j) + T_i \cdot (C + t_j + t_i)$. In O, the cost of (i, j) is $T_i \cdot (C + t_i) + T_j \cdot (C + t_i + t_j)$. By taking difference, we have

$$T_j \cdot (C + t_j) + T_i \cdot (C + t_j + t_i) - T_i \cdot (C + t_i) - T_j \cdot (C + t_i + t_j) = T_i * t_j - T_j * t_i \leq 0 \qquad (2)$$

Th last step is due to our greedy property. The difference implies that the cost in G is smaller than the cost in O. Then by exchanging order in O from G, the new solution becomes no worse. Thus, the greedy algorithm is optimal. □

EC1(YL4DF)

Extra Credit Instructions: For each of the next three homeworks (HW6, HW7, HW8), you will be given an extra credit problem, with the opportunity to replace one homework at the end of the course. Combined, the extra credit portions are worth 60 points—the same as one normal homework assignment. Each Extra Credit portion (EC1, EC2, EC3) will be combined into one *optional* replacement homework, meaning that the total scored out of all 60 extra credit points will replace your lowest homework grade for the semester. Note: this extra credit will **not** be applied to this homework (HW6). To make the most of this opportunity, you should commit to solving **all** extra credit portions for the next three homework assignments.

Since the extra credit is optional, no office hours will be provided to aid in solving the additional *optional* problems.

EXTRA CREDIT 1 *Social Distancing*

You are leaving town on a backpacking trip through Shenandoah National park with a close friend to engage in social distancing. 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 give its running time. Assume that M is the maximum weight of all the items, i.e., $\forall i,\ w_i \leq M$. The running time 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.

Algorithm:

Design a 3D Boolean structure divide(p, q, d), in which p denotes all items that should be assigned, q denotes the items assigned to me, and d denotes the difference. The cell returns True if such an assignment exists, and False if otherwise.

The recursive structure is given by whether the last item is assigned to me, i.e. divide$(p, q, d) =$ divide$(p-1, q, d+w_i)$ || divide$(p-1, q-1, d-w_i)$.

Base Case: Initialize divide$(0, 0, 0) =$ True, divide$(i, 0, 0) =$ False, divide$(0, j, 0) =$ False, divide$(0, 0, d)$ $=$ False.

We use bottom-up approach to fill up the 3D matrix:

For p from 1 to n:
  For q from 1 to p:
    For d from 0 to $p \cdot M$:
      divide$(p, q, d) =$ divide$(p-1, q, d+w_i)$ || divide$(p-1, q-1, d-w_i)$

In order to get the minimum difference, we find the smallest d satisfying divide$(p, q, d) =$ True for $p = n$ and $q \in [\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil]$. The range for q is given by the restriction that the number of items that each one carries differ at most 1.

In order to get the list of items for each one, start from the cell where we get the smallest d. If $(p-1, q, d+w_i)$ is True, then w_i is assigned to another person and recursively transverse back starting from $(p-1, q, d+w_i)$. If divide$(p-1, q-1, d-w_i)$ is True, then w_i is assigned to me and recursively transverse back starting from $(p-1, q-1, d-w_i)$. If both are True, we can go either direction. The procedure is stopped when we have assigned all items.

The running time for creating 3D matrix is $\Theta(n * n * nM)$, since range for p is from 0 to n, for q is from 0 to n, for d is from 0 to nM. It takes $\Theta(M)$ to get the smallest d and takes $\Theta(n)$ to get the lists. Thus, the total running time is $\Theta(n * n * nM + M + n) = \Theta(n^3 M)$.