CS4102 - ALGORITHMS - FALL 2019

DUE TUESDAY, NOVEMBER 5, 2019 AT 11PM VIA COLLAB

H6:ER6QT

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 `collabs` command at the top of the tex file. 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 them. Do not submit a solution that you are unable to explain orally to a member of the course staff. **Please remember that you are not allowed to share any written notes or documents (these include but are not limited to Overleaf documents, LaTeX source code, homework PDFs, group discussion notes, etc.). 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: zz9ek, zh2yn

Sources: Cormen, et al, Introduction to Algorithms

PROBLEM 1 *Let's Make a Millionaire*

You are a contestant on a new game show, on which you are presented with a row of n doors. Each door has a sign that displays the quantity of prize money behind that door. The goal is to choose the best combination of doors in order to take home the most prize money possible. The only catch is that you cannot choose adjacent doors.

More specifically, given an array `doors` of size n, where `doors[i]` is the amount of money behind door i, write a **dynamic programming** algorithm to find the maximum amount of money you could win by choosing non-adjacent doors (i.e., you cannot open both door i and door $i+1$). Prove the correctness and running time of your algorithm.

As an example, given input `doors = [1,4,2,1,8]`, your algorithm should return 12. That is, the optimal solution would be to open door 1 (`doors[1]` $= 4$) and then open door 4 (`doors[4]` $=$ 8), giving a total winnings of 12.

Note: This problem appeared on the Final Exam from Spring 2019.

Algorithm

Given an array `doors` of size n that holds the input door numbers, we will iterate through each element(door) in the array. As a dynamic programming algorithm, we need to create a new array `res` which will hold the results of the previously solved sub-problems, which in this case would be the maximum amount of money. Then, we are going to get the first element at index 0 (`doors`$[0]$) and assign it into the new array `res` as this is the only and current maximum amount of money. We then check the second element by accessing the first index in the array (`doors`$[1]$) and compare its value against the first element's value. In other words, we compare the value of the second element with the value of the previous element in array `res`. If the second element's value is larger than the first element's value, we will put the second element into array `res`. If the second element's value is smaller than the first element's value, then we will not put it in array `res`. This way, we are always choosing the door(element) that holds the higher quantity of prize money between the two doors being compared. Then, for the third element and onwards, until we have gone through all input n doors in the row, we will continue the process of comparing two doors' values against each other and choosing the door with the larger value of the two to put into the `res` array. We will continue to go through all input doors values in the array until the maximum money money value is found (the door that is at the last index in the `res` array will hold the maximum amount of money).

Proof of Correctness by Induction

Goal: $\forall k \in \mathbb{N}, P(k)$ holds

For all array array sizes n, or k in this instance, in all natural numbers, the property of finding the

maximum amount of money holds

Base cases: $P(1)$ and $P(2)$
P(1) - When the array size n is 1, that means there there is only one element(door) left in the array. No comparisons can be done, and therefore, the algorithm is done, with that door holding the maximum amount of money behind it. This is because the current nth door's value(i) + res[$i-2$] or res[$i-1$], where n is the number of doors, is undefined for $P1$.
P(2) - When the array `doors` of size n is 2, you cannot do the current nth door's value [i] + res[$i-2$] and res[$i-1$] because you only have 2 elements in the array, so you cannot check for 2 elements previous and one. It would be undefined for $P(i)$. Therefore, we compare the two elements we have, and if the second element(door #) is greater than the first element in memory, we add that element to memory. However, if it is smaller, we add the previous element in memory.
Hypothesis: $\forall_x \leq x_0, P(x)$ holds
Assume the array size is $x_0 = n-1$. For any array size that is smaller than $n-1$, this algorithm will give the correct result.

Inductive Step: show $P(1)$,...,$P(x_0) \Rightarrow P(x_0+1)$

When you add an element(door) to the array `res`, the maximum amount of money you could win can be either from the sum of values of the current element(ith door's value) plus the two prior elements in memory ([i] and [$i-2$]) or the value of the previous element in memory ([$i-1$]). Therefore, using the aforementioned equations for this algorithm, you can accurately calculate the maximum amount of money you could win and so we are guaranteed that this algorithm proves correctness.

Proof of Runtime
The runtime of this algorithm is $O(n)$ as we iterate through n number of doors in a row and store the doors holding the larger monetary values in memory, or array `res` / in the worst case, all n values will be checked.

PROBLEM 2 *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.

Algorithm
For this algorithm, we will start off by calculating the weight of bag1 for person1. For this, we will initialize j to be the first j items in bag1 for person1, k to be the count/number of j items assigned to $bag1$ for $person1$, and x to be the difference in the two bag weights between the two people, which can be written as such: $W(bag1) - W(bag2)$.

We can initialize the algorithm as such: $S(j,k,x)$. There are two conditions where $S(j,k,x)$ can hold true. We will find each of these two conditions by adding the last item j to either bag1 or bag2.

When bag1 holds $k-1$ items, and bag2 holds $j-k$ items, we add $(+1)$ the last item j to bag1 so that we can keep the number of items that each person has equal. Bag1 then has k number of items, which causes the difference in weights to be $x + W(j)$.

When bag1 holds k items, and bag2 holds $j-1-k$ items, we add the next item j to bag2. Bag2 then has $j-k$ items, which causes the difference in weights to be $x - W(j)$

The pseudocode is as follows:

Initialize $S(0,0,0)$ = True
for $j = 1, ..., n$:
  for $k = 1, ..., \lceil \frac{n}{2} \rceil$:
    for $x = 0, ..., nM$:
      $S(j,k,x) = S(j-1,k,x+W(j))$ or $S(j-1,k-1,x-W(j))$

In the pseudocode above, we start off by initializing $S(0,0,0)$ to true. $S(0,0,0)$ is true if of the first j items, k number are assigned to bag1 and it's weight(x). Then we check for all possible combinations of weights for the items and see if the possibility is valid or invalid. Validness of $S(j,k,x)$ is dependent on whether one of the two conditions mentioned above are met. $k = \lceil \frac{n}{2} \rceil$ is the max value for k so that we can satisfy the requirement that the maximum difference in the total number of items that each person can carry in bag1 and bag2 is 1. The weight x can range from when all the items have the maximum weight M and are assigned in bag1/(0) (all in one bag), to when all the items are assigned in bag2/(nM). Some weight differences x are unable to be obtained by definition of the problem and how the weights need to be distributed. Therefore, some states are invalid.

Finding minimum weight difference/Backtracking to find list of items each person will carry

In order to divide the items between the two people/bags such that the weight difference is as small as possible, we will, first, search for the x, the minimum weight difference by setting/going through all items, $j = n$, and setting $k = \lceil \frac{n}{2} \rceil$, where half of the items are in bag1, and then find/search for the minimum weight diff. x. Once we find this minimum x, we will proceed to find the list of items that each person will carry/which bag the item will go in by backtracking. In order to come to the state where where we have minimum x by recursively checking both conditions $S(j,k,x) = S(j-1,k,x+W(j))$ and $S(j-1,k-1,x-W(j))$ to see if they are true for a true $S(j,k,x)$. This allows us to know which bag/person holds item n accordingly. For example, if the first condition $S(j,k,x) = S(j-1,k,x+W(j))$ is true, than we assign item n to bag1/person1, and if the second condition $S(j-1,k-1,x-W(j))$ is true, we assign item n to bag2/person2. If both are true, then consistently choose one of the two bags/people to assign the item to. This process is continued until $S(0,0,0)$, or all the items have been assigned to a bag/person.

Runtime
The run time of this algorithm can be evaluated to $O(n^3M)$ The first for loop loops through all n items, the second for loop iterates till $\lceil \frac{n}{2} \rceil$ number of items, which is n (asymptotically), and the third loop iterates through all the weight differences (range of weight differences), which is nM asymptotically. Thus, taking everything into consideration, our overall runtime is $(n + n + nM)$, which can be simplified to $O(n^3M)$.

PROBLEM 3 *Course Scheduling*

The university registrar needs your help in assigning classrooms to courses for the spring 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.

Algorithm
For the implementation of course scheduling, we are using a greedy algorithm with the earliest start property. Our goal is to find the minimum number of classrooms to schedule all classes so that no two classes occur at the same time in the same classroom
You start by sorting the classes by their start time, so $s_1 \leq s_2 .... \leq s_n$, in ascending order so that we can find the minimum end time for all classrooms (used in priority queue). You then set the number of initial classrooms to 0. While a class still has to be placed in a classroom, take the first class not yet given a classroom and if it is compatible with the classroom in that it starts after the ending time of the last class (no overlapping), then schedule that class into that classroom. If not, then create/allocate a new classroom (number of classrooms + 1) and schedule that class into that classroom and increment the number of classrooms (number of classrooms = number of classrooms + 1). When the condition is no longer met (all classes have been assigned a classroom), return the number of classrooms.
This algorithm can be implemented using a min heap:
Note: Each classroom(node) in the heap stores the list of courses held within that classroom so we can check all assignments of courses to classrooms.
We can store the classrooms(nodes) in a min heap that can be keyed on the finish time of the last class. In the case of this algorithm, this implementation chooses a classroom whose last class has the earliest finish time (min/root node). We use the peek operation to get this root node. To decide whether a given class can be scheduled within a certain classroom, we compare s_1 to the key which holds the minimum end time of a classroom in the min heap. We increase the key of a classroom to e_i when a class is scheduled into a classroom to update the ending time of the course in the classroom interval. In a min heap, if the course in question is compatible with the root node classroom, we pop the root node off and put the course into it, and then reinsert the node(classroom) that holds the list of all courses thus far with the new course back into the heap, updating the end time of the course in that node classroom. If the next course in the list is not compatible with the root node classroom, do not pop the root node; add a new node, aka create a new classroom, to the heap and percolate up. After we have gone through all courses in the list, we will return the number of nodes in the heap which are the minimum number of classrooms we needed to assign all courses.

Prove Correctness (Greedy Exchange Argument)
The above greedy algorithm adds a classroom R when a course C cannot be assigned/is incompatible with the other classrooms (R - 1). Incompatibility of an assignment of a course to a classroom is based off of the rest of the classrooms (R - 1) as you check if they are holding courses that end after the current course's start time. In other words, we add a classroom R when a course C has a starting time that begins before any of the other classes have ended. Therefore, it can be inferred that all the classrooms from 1 to $R - 1$ were already occupied because they had courses ending after course C. As well, since the algorithm assigns courses by their start time in ascending order, the other classrooms' last course C had to have started before the current course and ended after the current course, which would cause two courses to overlap. This means that there would be a direct conflict in times and the current course C would not be able to be assigned into any of those classrooms. Therefore, for compatibility purposes, we add another classroom to assign that course into it. Thus, this algorithm returns T total number of classrooms(total number of nodes

in min heap) because this was minimum number of overlaps between courses. The minimum number of classrooms(nodes in the heap) returned/required ensures us that we will not have any overlapping courses assigned to any classroom at any one given time, and, as mentioned before, each classroom(node) in the heap stores the list of courses held within that classroom, thus we will find a complete schedule holding the assignment of courses to classrooms.

Prove Runtime
Overall, this algorithm runs in $O(nlogn)$ time.
Breakdown:
The initial sorting of all the n courses by their start time, s_i, takes $O(nlogn)$ time if we used a sort such as mergesort, which in the worst case runs $O(nlogn)$.
After that, in order to iterate/go through each n course in the given list of all n courses and determine if it is compatible with the current classroom, it takes $O(n)$ time, visiting each course exactly once and checking it's compatibility with the root node(classroom) in the min heap, giving a total of n operations.
Total runtime: $O(nlog(n)) + O(n)$, which is equivalent to $O(nlog(n))$.

PROBLEM 4 *As You Wish*

Buttercup has given Westley a set of n tasks $t_1, \ldots, t_n$ to complete on the farm. Each task $t_i = (d_i, w_i)$ is associated with a deadline d_i and an estimated amount of time w_i needed to complete the task. To express his undying love to Buttercup, Westley strives to complete all the assigned tasks as early as possible. However, some deadlines might be a bit too demanding, so it may not be possible for him to finish a task by its deadline; some tasks may need extra time and therefore will be completed late. Your goal (inconceivable!) is to help Westley minimize the deadline overruns of any task; if he starts task t_i at time s_i, he will finish at time $f_i = s_i + w_i$. The deadline overrun (or lateness) of tasks—denoted L_i—for t_i is the value

$$L_i = \begin{cases} f_i - d_i & \text{if } f_i > d_i \\ 0 & \text{otherwise.} \end{cases}$$

Give a polynomial-time algorithm that computes the optimal order T for Westley to complete Buttercup's tasks so as to minimize the maximum L_i across all tasks. That is, your algorithm should compute T that minimizes

$$\min_T \max_{i=1,\ldots,n} L_i$$

In other words, you do not want Westley to complete any task too late, so you minimize the deadline overrun of the task completed that is most past its deadline. Prove that your algorithm produces an optimal schedule and analyze its running time.

For the implementation of As You Wish, we are using a greedy algorithm with the earliest deadline property ~~with the earliest end property~~. This property will allow Wesley to do tasks with the earliest deadline first, and then complete tasks of later deadlines afterwards (in order of descending order). We do this so that Wesley can choose which tasks to take care of the earliest to minimize the deadline overruns (or lateness) of tasks/minimize the maximum L_i across all tasks. Therefore, it is essential that Wesley works on tasks with the earliest deadlines in ascending order (start with the task of earliest deadline with task of second earliest deadline and so on).

We will use the exchange theory to prove correctness (that our algorithm produces an optimal schedule) for this greedy algorithm.
Say we have two tasks, $T1$ and $T2$. Task $T1$ is completed longest past its deadline (largest lateness). Therefore, task $T2$ has a deadline d_i either earlier or equal to the deadline d_i for task $T1$. This can

be written as such:

$T1 = T1_i$
$T2 = T2_{i+1}$

$T1_i = (d_i, w_i)$
$T2_{i+1} = (d_{i+1}, w_{i+1})$

where d_i is the deadline and w_i is the amount of time needed to complete a given task.

The lateness of task $T1$ and task $T2$ can be denoted, respectively, as such:

$L_{T1} = (s_i + w_i) - d_i$
$L_{T2} = (s_i + w_i + w_{i+1}) - d_{i+1}$

where s_i denotes the start time of a given.

An optimal solution exists when tasks $T1$ and $T2$ have the same amount of lateness/deadline overrun and same order.
If we swap the lateness values of tasks $T1$, denoted L_{T1}, and $T2$, denoted L_{T2}, from above, we will denote the lateness values of these two tasks after being swapped as:

$T1 = J1 = L_{J1}$

$T2 = J2 = L_{J2}$

$L_{J1} = s_i + w_{i+1} + w_i - d_i$
$L_{J2} = s_i + w_{i+1} - d_{i+1}$

We then compare the lateness values of the tasks before swapping (L_{T1}, L_{T2}) and after swapping (L_{J1}, L_{J2})

$L_{T1} = (s_i + w_i) - d_i \leq L_{J1} = s_i + w_{i+1} + w_i - d_i$
$L_{T2} = (s_i + w_i + w_{i+1}) - d_{i+1} \leq L_{J1} = s_i + w_{i+1} + w_i - d_i$

Due to the ascending order of the algorithm, d_i is greater than d_{i+1} Thus, $L_{T1} \leq L_{J1}$ and $L_{T2} \leq L_{J1}$. These comparisons show that we can achieve an equally optimal solution when we swap our task with another task in an arbitrary solution. As well, these comparisons prove that after swapping values, the lateness values of the two tasks did not increase more than the arbitrary optimal solution. So, in essence, according to the exchange argument, the lateness value of our algorithm will be better than or equal to that of an optimal solution. Therefore, because the maximum lateness value of our algorithm is $\leq$ the max lateness value of an optimal algorithm, we have proved that our algorithm produces an optimal schedule.

Runtime
The runtime of this algorithm is $O(nlogn)$ as we are sorting by earliest deadline.