

CS 577: Introduction to Algorithms**Homework 4****Out: 02/23/21****Due: 03/02/21**Name: Joseph O'ConnellWisc ID: Jpoconnell2 9075895301**Ground Rules**

- Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document. **Since students have had issues fitting their answers into the boxes, we are increasing the size of the boxes for this problem set. However, do NOT feel obligated to fill the entire solution box. The size of the box does NOT correspond to the intended solution length.**
- The homework is to be done and submitted individually. You may discuss the homework with others in either section but you must write up the solution *on your own*.
- You are not allowed to consult any material outside of assigned textbooks and material the instructors post on the course websites. In particular, consulting the internet will be considered plagiarism and penalized appropriately.
- The homework is due at 11:59 PM CST on the due date. No extensions to the due date will be given under any circumstances.
- Homework must be submitted electronically on Gradescope.

Problem 1:

1. In the year 2048 (when teleportation is already invented), you have opened the first delivery company (*FastAlgo Express*) that uses a teleportation machine to deliver the packages. Suppose on a certain day, n customers give you packages to deliver. Each delivery i should be made within t_i days and the customer pays you a fixed amount of p_i dollars for doing it on time (if you don't do it on time, you get paid 0 dollars). On-time delivery means that if package i is due within $t_i = k$ days, we should deliver it on one of the days $1, 2, \dots, k$, to be on time. Unfortunately, your teleportation machine is able to do at most **one** delivery to **one** destination in one day (you cannot deliver packages to multiple destinations within 1 day). Your goal is to figure out which deliveries to make and when.

Input: A set of n deliveries with due dates $t_i \in \mathbb{N}, t_i \geq 1$ and payments $p_i > 0$ for each delivery $i \in \{1, \dots, n\}$.

Output: A delivery order such that your company's profit (the sum of p_i for the deliveries made on time) is maximized.

Example: You have 4 deliveries with due dates: $t_1 = 2, t_2 = 3, t_3 = 1, t_4 = 2$. You cannot deliver all of them on time, but you can make the third delivery on day 1, then the first on day 2 and the second on day 3.

We call a subset S of the deliveries *feasible* if there is a way to order the deliveries in S so that each one is made on time. You may assume the following lemma holds without proving it: *S is feasible if and only if for all days t that $t \leq n$, the number of deliveries in S due within t days is no more than t .*

Describe and analyze an efficient algorithm to determine the order in which the deliveries should be made so that you maximize your profit. You should aim for a runtime $O(n^2)$ or better.

(Hint: Build up a feasible set of deliveries by considering them in a particular order and adding each delivery to your schedule if the set of selected deliveries continues to remain feasible. Use the lemma to check for feasibility. What order should you consider deliveries in to maximize the profit? Use an "exchange" argument to prove the correctness of your algorithm.)

Ordering: we have two deliveries with values p_1 and p_2 , in order to maximize profit, we would choose the delivery with the higher value. This means we should look at orders in terms of decreasing values and prioritize based on this ordering.

```
Money(t[1,2,3,...n], p[1,2,3,...n])
    Sort the deliveries from highest to lowest values
    Define an empty set S to hold deliveries
    Define an array A of size n to use for determining feasibility
    For each delivery in the sorted order:
        // check if feasible
        If  $t[i] < A[i]$ 
            // by using the given lemma, the delivery is feasible
            Update array A
            Add the current delivery to S

    Return S
```


Running Time:

$O(n \log n)$ for sorting from highest to lowest

$O(n)$ n many times within the loop so $O(n^2)$ Checking if the delivery is feasible and updating A.

$O(n^2)$

Prove Correctness: S is the schedule output by the algorithm above. T is another optimal schedule to maximize profit. We want to prove $S = T$.

Assuming $S \neq T$ then we will find the first delivery i in decreasing order where S has delivery i , but T does not include i .

Suppose we create T' by adding i to T . T' could no longer be feasible so a delivery needs to be removed in order to make it feasible. Because of the decreasing value of deliveries, we want to remove a delivery with a higher index than i .

Since T was feasible and we added an item to create T' , if T' is infeasible then there is a day t at some point in T' where there are $t + 1$ deliveries on day t .

Since S is feasible and T' and S are the same until delivery i , then there must be a delivery greater than i with a delivery date t or less. We will select this item to remove from T' . Removing this item creates T'' .

Therefore, T'' is feasible and the maximum profit of T'' is not worse than T .

We can simply repeat these steps and change a feasible schedule of T into another feasible schedule S without worsening the maximum profit.

Therefore, S is an optimal schedule.

Problem 2:






2. There are m doors numbered 1 to m which are all initially closed. Your goal is to open them by pressing n switches numbered from 1 to n in an order of your choice. The behavior of the switches is described by an $n \times m$ matrix M . When switch i is pressed door j behaves as follows:

- If the (i, j) th entry, M_{ij} , is 1, door j **opens** no matter whether it was open or closed before.
- If the (i, j) th entry, M_{ij} , is -1 , door j **closes** no matter whether it was open or closed before.
- If the (i, j) th entry, M_{ij} , is 0, door j **remains at its previous state** (open or closed).

In this problem, you will design a greedy algorithm for determining an order to press the switches assuming such an order exists. **Each switch must be pressed exactly once.** Your algorithm is given the matrix M as input.

Example Suppose there are $m = 5$ doors and $n = 3$ switches with matrix M shown on the right. The optimal way to open all doors is to press switches in the order 2,3,1.

- Initially: (Closed, Closed, Closed, Closed, Closed)
- After 2: (Open, Open, Open, Closed, Closed)
- After 3: (Open, Closed, Open, Closed, Open)
- Finally: (Open, Open, Open, Open, Open)

					
Switch ₁	0	1	1	1	0
Switch ₂	1	1	1	0	-1
Switch ₃	0	-1	0	0	1

- (a) Determine an ordering that works for the following matrix. No explanation is required.

Hint: Think about which switch should be pressed last.

$$\begin{bmatrix} 1 & -1 & 1 & -1 \\ 0 & 1 & 1 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & -1 & -1 & 1 \end{bmatrix}$$

Row

$$\begin{bmatrix} 1 & -1 & 1 & -1 \\ 0 & 1 & 1 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & -1 & -1 & 1 \end{bmatrix}$$

1, 4, 3, 2

Door #

row order	1	2	3	4
1	1	-1	1	-1
4	0	-1	-1	1
3	1	-1	0	0
2	0	1	1	0

result | 1 1 1 1

✓

- (b) Now consider a general instance with n switches and m doors. Assume that all the doors can open by pressing the switches in some order. State a greedy rule for determining which of the n switches should be pressed **last**.

I will generalize my greedy rule so it can be applied for all steps. Assume we have an array to keep track of "1"s following the current greedy step. We start from the last step.

Greedy Rule: For each column that does not have a following "1" (use array), iterate through all numbers and find all columns with only 1 remaining "1" and extract the row containing that 1. If none exist, iterate again with 2, then 3, ..., up to N .

Once we have a set of rows, pick an arbitrary switch(row) that does not violate the following rule: using the array, the current row must not insert a -1 unless there is a 1 following it within the column.

Finally, update the array according to the indexes of the row which contain a 1.

Since the array does not have any information for choosing the last switch to be pressed, there can be no "-1"s in the last switch row.

- (c) Briefly prove the correctness of your greedy rule. That is, show that if there exists an ordering of switches for opening all doors, then there exists one where the switch you chose in **part (b)** is pressed last.

Show: There exists one ordering of switches where the switch I chose in part b is pressed last.

There are possibly multiple possible solutions to this problem. Let's say my greedy rule outputs S and the optimal switch to press last is S^* .

Proof: Using the exchange argument, prove that $S = S^*$ or $S \equiv S^*$ for switch choice that is pressed last.

Invariant 1: For any sequence of 1s, -1s, and 0s, in order for a door to be open after the last switch is flipped, the last nonzero number in the sequence must be a 1.

Direct Proof: Following the properties of what each number does to the door, the only possible method to end with a door open is to have the last nonzero number be a 1. From this we can derive that the last switch choice must contain no -1s. S and S^* do not contain a -1. If S^* did contain a -1, it would not be an optimal solution.

Invariant 2: If there exists multiple rows not containing a -1, the optimal solution will take the row with the fewest remaining occurrences of 1 in each of its columns.

Proof: if the S^* index with the lowest occurrences count of 1 is $> S$ index with the lowest occurrences count of 1, there is a possibility that S^* will not produce a valid ordering of switches. Therefore, S^* would not be optimal. So, S^* must be the row containing the fewest occurrences of 1 at an index.

Final Proof:

S outputs a switch with no -1s its row and the fewest occurrences of 1 at some index.

Following invariant 1 and 2, S^* must follow the same properties.

There is a possibility of multiple rows following both conditions. In this case, the choice is arbitrary and multiple solutions are possible.

Therefore, $S \equiv S^*$ and the greedy rule in part b is correct.

- (d) Develop an algorithm for determining an ordering (in reverse order from last to first) based on your greedy rule from **part (b)**. Describe your algorithm by filling out details in the following pseudocode. Here S denotes the set of switches left to consider when some partial suffix of the ordering has been determined. Feel free to use any other notation your algorithm requires.

Algorithm 1: Order(M)

Returns an ordering over $1, \dots, n$ that opens all doors, in reverse order, given the matrix M .

1 Initialize $S = \{1, \dots, n\}$.

2 */*Add any extra bookkeeping*/*

Create an int array of size m named ONEF // initialized to zero

NumDoors = number of doors (column)

NumSwitches = number of switches (row)

3 **while** $S \neq \emptyset$ **do**

4 */*Insert greedy rule here to determine an index i .*/*

For(int $i = 1$; $i \leq \text{numSwitches}$; $i++$)

For each door (column) j where $\text{ONEF}[j] \neq 1$

Iterate through all switches and find columns with exactly i remaining "1"s and extract the row containing that "1".

If there is at least 1 row (switch) in this set, break out of the loop.

For each row in this set

For each index j in a Currentrow

If $\text{currentRow}[j] == -1$ and $\text{ONEF}[j] == 0$

break out of the current loop and go to next iteration of outer loop

//If we reach this point, we have a switch (row) i to add to the back of the ordering

5 Append i to the end of the ordering.

6 Set $S := S \setminus \{i\}$.

7 */*Add any extra bookkeeping*/*

Iterate through each index of i and if the value $== 1$, update the same index of ONEF to $== 1$.

8 Return ordering.

- (e) State the asymptotic runtime of your algorithm in **part (d)**. No explanation required.

While loop: n

First nested for loop: $n \cdot n \cdot m$

Second nested for loop: $n \cdot m$

Constant time

Constant time

The above algorithm runs in $O(m \cdot n^3)$ time.

