CS3000: Algorithms & Data — Summer I '21 — Drew van der Poel

Homework 4
Due Saturday, June 12 at 11:59pm via Gradescope

Name: Gabriel Peter
Collaborators:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- This assignment is due Saturday, June 12 at 11:59pm via Gradescope. No late assignments will be accepted. Make sure to submit something before the deadline.
- Solutions must be typeset in LaTeX. If you need to draw any diagrams, you may draw them by hand as long as they are embedded in the PDF. I recommend using the source file for this assignment to get started.
- I encourage you to work with your classmates on the homework problems. *If you do collaborate, you must write all solutions by yourself, in your own words.* Do not submit anything you cannot explain. Please list all your collaborators in your solution for each problem by filling in the `yourcollaborators` command.
- Finding solutions to homework problems on the web, or by asking students not enrolled in the class is strictly forbidden.

Problem 1. *Huffman Code Example (*10 points*)*

Given the following alphabet $\Sigma = \{a, b, c, d, e, f, g\}$ and corresponding frequencies, use Huffman's algorithm to compute an optimal prefix-free code. Represent the prefix-free code as a binary tree, with either 0 or 1 on each edge. Compute the expected encoding length.

Letter	a	b	c	d	e	f	g
Frequency	0.25	0.21	0.2	0.1	0.08	0.05	0.11

Solution:

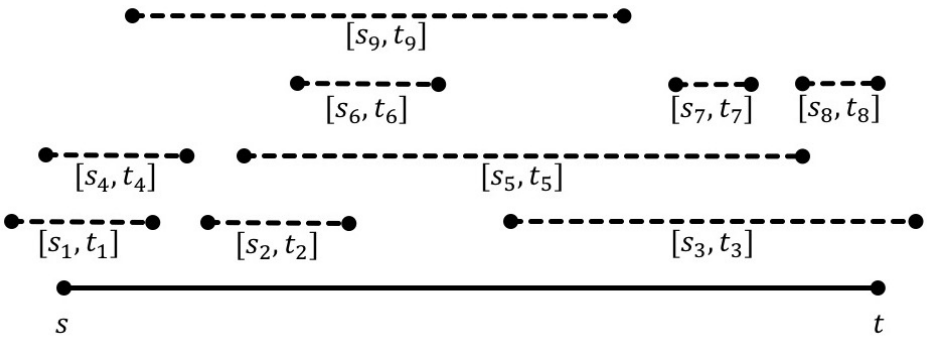
ATTACHED AS IMG 1.

Problem 2. *I wish I was a baller (*32+4 points*)*

As president of the Northeastern chapter of the Big Ballers Club, you need to staff a booth at this year's open house. The open house runs for the interval $[s, t]$. There are n volunteers, each of which is able to cover the booth for the interval $[s_i, t_i]$ ($i \in \{1, \ldots, n\}$). You need to select a set of volunteers $S \subseteq \{1, \ldots, n\}$ to *cover* the entire open house, meaning that $\bigcup_{i \in S} [s_i, t_i] \supseteq [s, t]$. Equivalently, for every time $z \in [s, t]$, there is some volunteer $i \in S$ such that $z \in [s_i, t_i]$. Each member will be paid for their services, but ironically the Big Ballers Club has very limited funds, so you need to ensure $|S|$ is as small as possible.

In this problem you will design an efficient greedy algorithm that takes as input the numbers $s, t, s_1, t_1, \ldots, s_n, t_n$ and outputs a set S that covers the open house and uses the minimum number of staffers. The running time of your algorithm should be at most $O(n^2)$. You may assume that a solution exists.

The following is an example input with 9 volunteers. One optimal solution is $S = \{1, 3, 9\}$.



(a) (**8 points**) Describe your algorithm in pseudocode. Provide a few sentences describing your algorithm.

Solution:

```
# Variable init(s)
s = None, t = None, n = None
S =[], staff = []

# Preprocessing step allows for original index
# to stay consistent regardless of array manipulations.
for i in range(n):
    m = stdin[i+3].split('␣')
     staff.append([i+1, int(m[0]), int(m[1])])

# Loops until solution is made
# (i.e. a final shift is added to extend past 't')
while s < t:
    best_m = [0,0,0]
```

```
    i = 0
    # Iterated through all staff members still
    # under consideration
    while i < len(staff):
        member = staff[i]
        start_time = member[1]
        end_time = member[2]

        # Rule 1: the solution can only be considered
        # if it doesn't allow a gap with the previous
        if start_time <= s:
            # Rule 2: The longest that prevents overlap
            # is the prioritized shift in the Solution.
            if end_time >= best_m[2]:
                best_m = member.copy()
            # Crucial! Removes elements that are included
            # or will never be included in the Solution.
            staff.pop(i) # A!!
            i -=1
        i += 1
    S.append(best_m[0])
    # new interation now considers the overlap
    # with best found shift's end time...
    s = best_m[2]

S.sort()
for i in range(len(S)): # printing formatics of the array
    print(str(S[i]), end=' ')
```

The solution is rather simple:

First, there is a preprocessing step which map's the staff array elements to also consist of there original index. This is done to ensure regardless of the array manipulation (foreshadow) the start and end times for a shift correlate to the original index which is also their label.
Next, the code enters a loop which iterates over until a shift's endtime is past or equal to the overall booth's endtime. Therefore, each iteration of this outer loop guarantees that we find a new shift to add to the optimal solution.
How do we find the next optimal shift? Well the internal code the iterates through all the members of the "still-available" staff and uses a two part greedy rule to consider if they are an optimal addition:
Part 1: The next segment of this schedule must overlap with the previous, meaning that the new shift in consideration must have a start -time before or equal to the previous shift's endtime. This was a strict rule in the problem as the station should never be unmanned.
Part 2: The solution should have the most distant end time if it supports Part 1. During part c,

it is explained why this is essential.

After all available staff shifts are considered it adds the best one found to the solution and makes the new "latest start time possible" the newly found shift's endtime. The cycle stops once a segment's shift surpasses the endtime needed for the booth, and the solution is sorted and printed...

(b) (**4 points**) Analyze the running time of your algorithm.

Solution:

The solution has a $o(n^2)$ because for each shift found in N iterations, it removes 1 element from the array, which means that only n-1 iterations are needed to find the next consecutive shift. We can remove the optimal shift because it can only be used once per solution, and can no longer be considered again.

$n + (n-1) + (n-2) + ... + (n-n) = n^2 - \sum_i^n i = n^2 - \frac{n^2+n}{2}$ (gauss) $= \frac{n^2-n}{2} < O(n^2) = o(n^2)$

Keep in mind that the preprocessing step is only o(n) which is insignificant to the meat of the algothihm of n^2

(c) (**12 points**) Prove that your algorithm finds a set of volunteers S of minimum size to cover the open house.

Solution:
In order to prove the correctness of this algorithm, I will base my explanation off the in-class proof of "Greedy stays ahead." in conjunction with the "exchange argument".

You need to take an optimal solution S. You can tell that this solution can grant $|S|$ positions, where $|S|$ is the minimum shifts you needs for a contigious schedule.

Then you take your solution G, which is greedy.

Now you say the first job you chose in G, this shift is 'i' and the time spent is $[s_i, t_i]$. There is two possibility, either 'i' is in S or it's not. If it's not by replacing one job in S by 'i' you get a better solution. BUT S is optimal so you can conclude that 'i' is in S.

Then you do that for every element. And you'll conclude that the amount of shifts required earned is the same in G and in S. So G is optimal.

In a more informal view, part 1 of the greedy rule ensures shift overlap; therefore, we need simply need the longest segments with a shift duration of s-t. Longer segments will allow for less small shifts as their domain takes more of the total booth time. In the inner loop, we only exchange if the resulting solution is better, by the exchange argument if we continue this pattern the solution we receive that can no longer be exchanged for an even better one is now in the optimal solution.

(d) **[8 points]** In order to better your familiarity with greedy algorithms, we have created a hackerrank challenge (www.hackerrank.com/cs3000-summer1-2021-programming-assignment-4).

Please implement your greedy strategy and submit it to the challenge. Your grade for this part will depend on (a) how many test cases your implementation passes and (b) actually implementing a greedy strategy (we will check!).

In order to allow for efficient grading, please write your hackerrank account below.

Solution:
peter_g

(e) **[+4 points]** Describe a greedy algorithm which solves this problem in $o(n^2)$ time. State and justify your runtime.

Solution:

#A allows for $o(n^2)$ because it also remove solutions that will now never be valid after finding the best consecutive segment. This is because if we find a certain shift to have a valid start time, but end up not being the best solution with the longest endtime, the shift will never be considered again since the new start-time is beyond this stale shift's start-time. A shift will never be considered twice if it got beat by a superior shift in length. This leaves a guaranteed runtime of $\leq O(n^2) = o(n^2)$

In fact, the previous explanation on 2b) already supports the algo to be $o(n^2)$ since $\frac{n^2-n}{2}$ satifies that asymptotic runtime. However, taking out even more segments allows for even more optimization, in a real-world example.

Problem 3. *Total Weighted Finish Time (*24 points*)*

A scheduler needs to determine an order in which a set of n processes will be assigned to a processor. The i-th process requires t_i units of time and has weight w_i. For a given schedule, define the finish time of process i to be the time at which process i is completed by the processor. (Assume that the processor starts processing the tasks at time 0.)

For example, consider 4 processes with $t_1 = 5, t_2 = 2, t_3 = 7, t_4 = 4$. And weights $w_1 = 1, w_2 = 3, w_3 = 2$, and $w_4 = 2$. Consider the schedule 1,3,2,4. The finish time for process 1 is 5, for process 2 is $t_1 + t_3 + t_2 = 5 + 7 + 2 = 14$, for process 3 is $t_1 + t_3 = 5 + 7 = 12$, and for process 4 is $t_1 + t_3 + t_2 + t_4 = 5 + 7 + 2 + 4 = 18$. The total weighted finish time of the schedule is then $5 * 1 + 14 * 3 + 12 * 2 + 18 * 2 = 5 + 42 + 24 + 36 = 107$.

(a) **[10 points]** Give a greedy algorithm to determine a schedule that minimizes the total weighted finish time. (**Hint:** If you are stuck, trying figuring out a rule that *always* works when $n = 2$).

Solution:

```
def OptS(w, t, n):
    X = [[i, t[i], w[i]] for i in range(n)] #1
    S = []
    while len(S) < len(X): #2
        max_w = X[0][2] #3
        sub_sols = [X[0]]
        for i in range(n): #4
            if i+1 in S: continue
            x = X[i]
            w = x[2]
            if w >= max_w: #5
                if w == max_w:
                    sub_sols.append(x)
                else:
                    max_w = w
                    sub_sols = [x]
        best = min(sub_sols, key=lambda x: x[1]) # 6
        S.append(best[0]+1)

    return S
```

For this greedy algorithm we state a precedence of Maximum weights then minimum times. This is based on intuition, then later proof, that multiplicative weights are significant more expensive than additive times and should be placed first in the queue such that a high-weight process does not multiply the entire summation of previous process times. However, if the multiplicate times are the same for two processes, then the one which takes less time should be placed first (explained in 3b)

#1, As a pre-processing step, the labels, times, and weights of the processes are a mapped to a single n-sized list, X.

#2-#5This lays the groud work for our algorithm. For each search of the next S schedule, we interate through the availble processes and find one with max_w in order to queue that first. In order to consider ties of max weight. The sub_solution is an array and holds all processes with the maximum weight so far. If a process is found with an even higher weight, it demotes all the other proccesses and becomes the sole sub_solution... etc.

#6: The minimum time of all tied-weight processes is found, then added to the solution. This ends the inner for-loop and a new iteration for S_{i+1} is run until $|S| = n$ where all entries are unique.

(b) **[10 points]** Prove the correctness of your algorithm.

Solution:

Let a be a constant. Informally we can quickly conclude that when $n > 1 \rightarrow n * a \geq n + a$.
This asserts the multiplicative precedence in our greedy algorightm. Where S is the current time summation of the schedule, $S_i * n \leq S_{i+1,2,3,...} * n$.

Now for why smaller t_i priority is used to break ties: We know that S_i has a smaller sum than S_{i+1}. This is because t_i is included in $S_i, ..S_n$ (based on how it is calculated) so if we put the larger values in further in the queue, those "sums" are then reduced as they contain less uses of the greate t_i value. However, as asserted before, the weights are more influential (multiplicative) than the times to process (additive), which is why they're not equal in greedy rule but sequential.

Now that we know this, our greedy rule will create an optimal solution because it will always choose the next consecutive solution that decreases the summation total where the highest w_i's and then lowest t_i's are always picked first. Given that we cannot invert the scheduling an expect a solution less than or equal to the value. We know that the greedy schedule is equal to the optimal schedule, and that greedy stay's ahead as it will always pick the best consecutive answer without ambiguity. This is essentially a rephrasing of the exchange argument which has already been formally proven in class and in #2! which can be applied here.

(c) **[4 points]** Analyze the worst-case running time of your algorithm.

Solution:

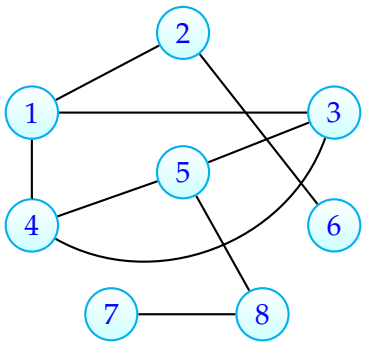
The runtime of this algorithm is $O(n^2)$:

#1 runs 'n' times, as one solution is found per interation and the Schedule needs to include all n processes. #4 run 'n' times in order to find the best next process to schedule as well as #6 to find the minimum item.
However, #4 and #6 are additive functions: so we simplify our expression $O(n(n + n)) = O(n(2n)) \rightarrow O(n^2)$.

Although unnessary, I can already see optimizations like adding all sub_sols elements first in sorted t_i order since they are all guaranteed to be the next elements of max weight. etc.

Problem 4. *Graph Representations and Exploration (15 points)*

This problem is about the following graph. **Tip:** Use the LaTeXfor rendering the graph as a starting point if your solutions involve drawing a graph.



(a) **[5 points]** Draw the adjacency matrix of this graph.[1]

Solution:

1	2	3	4	5	6	7	8
0	1	1	1	0	0	0	0
1	0	0	0	0	1	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
0	0	1	1	0	0	0	1
0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	1

(b) **[5 points]** Draw the adjacency list of this graph.

Solution:
$[1] = \{2, 3, 4\}$
$[2] = \{1, 6\}$
$[3] = \{1, 4, 5\}$
$[4] = \{1, 3, 5\}$
$[5] = \{3, 4, 8\}$
$[6] = \{2\}$
$[7] = \{8\}$
$[8] = \{5, 7\}$

(c) **[5 points]** BFS this graph starting from the node 1. Always choose the lowest-numbered node next. Draw the BFS tree and label each node with its distance from 1.

[1] **Hint:** If drawing the matrix in latex, I recommend using the `tabular` or `matrix` environments.

Solution:

ATTACHED AS IMG 2.

Problem 5. *Graph Properties (10 points)*

Consider an undirected graph $G = (V, E)$. The *degree* of a vertex v is the number of edges adjacent to v—that is, the number of edges of the form $(v, u) \in E$. Recall the standard notational convention that $n = |V|$ and $m = |E|$.

Prove by induction that the sum of the degrees of the vertices is equal to $2m$.

Solution:

Base Case(s):
If $m = 0$, $\sum degrees(v) = 0$
If $m = 1$, $\sum degrees(v) = 2$
Suppose we have a graph G with m edges where $m \geq 1$. By adding one edge by random, the resulting new graph now has $m + 1$ edges, we shall name it H.

We will assume that our Inductive Hypothesis to be that $\sum degrees(v) = 2(m + 1)$.

Through simple graph rules, we know that if we add a single edge, it must be connected by two vertices. If those two vertices each acquire a degree of +1, then we know that the new summation of degrees is $= 2m + 2$, which is equal to our IH through expression manipulation. $\sum degrees(v) = 2(m + 1) = 2m + 2$.

Conclusion of proof.