CompSci 330 Design and Analysis of Algorithms
Assignment 5, Fall 2024 Duke University

TODO: Add your name(s) here

Due Date: October 21, 2024

How to Do Homework. We recommend the following three step process for homework to help you learn and prepare for exams.

1. Give yourself 15-20 minutes per problem to try to solve on your own, without help or external materials, as if you were taking an exam. Try to brainstorm and sketch the algorithm for applied problems. Don't try to type anything yet.
2. After a break, review your answers. Lookup resources or get help (from peers, office hours, Ed discussion, etc.) about problems you weren't sure about.
3. Rework the problems, fill in the details, and typeset your final solutions.

Typesetting and Submission. Your solutions should be typed and submitted as a single pdf on Gradescope. Handwritten solutions or pdf files that cannot be opened will not be graded. LaTeX[1] is preferred but not required. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Any applied problems will request that you submit code separately on Gradescope to be autograded.

Writing Expectations. If you are asked to provide an algorithm, you should clearly and unambiguously define every step of the procedure as a combination of precise sentences in plain English or pseudocode. If you are asked to explain your algorithm, its runtime complexity, or argue for its correctness, your written answers should be clear, concise, and should show your work. Do not skip details but do not write paragraphs where a sentence suffices.

Collaboration and Internet. If you wish, you can work with a single partner (that is, in groups of 2), in which case you should submit a single solution as a group on Gradescope. You can use the internet, but looking up solutions or using LLMs is unlikely to help you prepare for exams. See the homework webpage for more details.

Grading. Theory problems will be graded by TAs on an S/I/U scale. Applied problems typically have a separate autograder where you can see your score. See the course policy webpage for details about homework grading.

[1] If you are new to LaTeX, you can download it for free at latex-project.org or you can use the popular and free (for a personal account) cloud-editor overleaf.com. We also recommend overleaf.com/learn for tutorials and reference.

Problem 1 (Pottery Firing). Phoebe is an artisan potter that specializes in terracotta. She offers a vast catalog of vases with various sizes, colors, and shapes, and she has just received an immensely large and complicated order for n different designs of vases, $V_1, V_2, \ldots, V_n$. Each design V_i takes a distinct number of minutes, s_i, for Phoebe to first sculpt by hand, and then another distinct number of minutes, f_i, for her to finish by firing it in her kiln. For purposes of this problem, suppose her kiln can fit any number of sculpted vases to fire and that it is OK for fully-fired vases to remain in the kiln until all vases are done firing.

Phoebe can only sculpt one design at a time, but immediately after she finishes sculpting a vase she adds it to the others in the kiln (if any) so that she can begin sculpting another vase. Once all vases are sculpted and fired in the kiln for at least as long as they all require, she pulls them all out at once and her job is done. Your goal is to help Phoebe determine which order she should sculpt the n designs in order to minimize the time when her job is done.

(a) Prove that ordering the designs in increasing order of s_i is not optimal by providing a counterexample by providing an input for which the ordering does not yield the best possible end time.

(b) Same as the first part, but ordering by increasing $s_i + f_i$.

(c) Same as the first part, but ordering by decreasing $s_i + f_i$.

(d) State an ordering for Phoebe to use and prove that it is optimal. You may find the following proof template useful[2]:

  (i) Compare your ordering to that of an arbitrary optimal ordering: If the orderings are not identical, then there are some pairs of designs in the optimal ordering that are out-of-order with respect to your ordering.

  (ii) Consider the optimal ordering *with the fewest such pairs.*

  (iii) Show that swapping two adjacent out-of-order designs in that ordering reduces the number of out-of-order pairs without increasing the overall end time.

  (iv) Conclude that, because the resulting order has fewer out-of-order pairs and optimal end time, your ordering is indeed optimal by contradiction.

Solution 1. .

(a) We will prove this greedy policy is not optimal by counter-example. Suppose $s_1 = 1$, $f_1 = 1$, $s_2 = 2$, $f_2 = 2$. Then, we sculpt V_1 before V_2 because $s_1 < s_2$. The total sculpting and firing time in this ordering is: $1 + 2 + 2 = 5$. However, if we sculpt V_2 before V_1, then the total sculpting and firing time is: $2 + 2 = 4$. This takes less time than the greedy choice, so the greedy choice is not optimal.

(b) We will prove this greedy policy is not optimal by counter-example. Suppose $s_1 = 1$, $f_1 = 1$, $s_1 + f_1 = 2$, $s_2 = 1$, $f_2 = 2$, $s_2 + f_2 = 3$. Then, we sculpt V_1 before V_2. The total sculpting and firing time in this ordering is: $1 + 1 + 2 = 4$. However, if we sculpt V_2 before V_1, then the

[2]This is a *proof by smallest counterexample* instead of induction, which is functionally equivalent but perhaps more intuitive here.

total sculpting and firing time is: $1 + 2 = 3$. This takes less time than the greedy choice, so the greedy choice is not optimal.

(c) **Informal Heuristic/Intuition:**
When cooking a meal, you can categorize dishes based on their prep times and cook times. If a dish needs to be cooked in an oven for a long time, you should probably prep the ingredients for that dish and put it in the oven first.

Ordering:
We will order the vase designs in **decreasing** order of the firing time f_i (note that there are no ties, due to distinctness), and sculpt them one after the other. This means that we first sculpt the vase with the largest f_i, and sculpt the vase with the smallest f_i at the end. Let the ordering obtained in this fashion be ALG.

Proof of Correctness:
For any arbitrary ordering O, a pair of designs (V_i, V_j) is said to be **out of order** with respect to ALG if the order they appear in O is not the same as ALG. Observe that if no pair is out of order, then both O and ALG must be the same ordering.

Consider an optimal ordering $OPT = (V_{i_1}, V_{i_2}, \dots, V_{i_n})$ of the designs which has the **fewest** out of order pairs with respect to ALG.

- If the firing time for V_{i_j} is **greater** than the firing time for $V_{i_{j+1}}$ for each $1 \le j \le n-1$, then the designs in OPT are in order of decreasing firing times, which is the same as ALG so our algorithm correctly computes the optimal ordering.
- If not, there exists some j such such that the firing time for V_{i_j} is **lesser** than the firing time for $V_{i_{j+1}}$. In other words, we have 2 *adjacent* out of order pairs in OPT. We claim that swapping $(V_{i_j}, V_{i_{j+1}})$ produces a new ordering OPT^* that is at least as good as OPT, i,e., requires total time at most that of OPT. If this were the case, we have a **contradiction** as we have reduced the number of out of order pairs by exactly one, meaning that OPT wasn't the optimal solution with the fewest out of order pairs.

  Firstly, note that the designs V_{i_k} for $k < j$ and $k > j+1$ still start and finish at the same time as they did before, as the total sculpting time for $(V_{i_j}, V_{i_{j+1}})$ after swapping remains the same:
  $$s_{i_{j+1}} + s_{i_j} = s_{i_j} + s_{i_{j+1}}.$$

  Let us suppose V_{i_j} was started at time t_{i_j} in the original ordering OPT. Then, in OPT the designs $(V_{i_j}, V_{i_{j+1}})$ get finished at times
  $$t_{i_j} + s_{i_j} + f_{i_j}, \qquad t_{i_j} + s_{i_j} + s_{i_{j+1}} + f_{i_{j+1}}$$
  respectively. On the other hand, in OPT^* the design $V_{i_{j+1}}$ is done immediately before V_{i_j}, and they get finished at times
  $$t_{i_j} + s_{i_{j+1}} + s_{i_j} + f_{i_j}, \qquad t_{i_j} + s_{i_{j+1}} + f_{i_{j+1}}$$
  respectively. Since $f_{i_j} < f_{i_{j+1}}$ (the pair was out of order), we have the inequalities
  $$t_{i_j} + s_{i_{j+1}} + s_{i_j} + f_{i_j} \le t_{i_j} + s_{i_j} + s_{i_{j+1}} + f_{i_{j+1}}$$

$$t_{i_j} + s_{i_{j+1}} + f_{i_{j+1}} \leq t_{i_j} + s_{i_j} + s_{i_{j+1}} + f_{i_{j+1}}$$

where the RHS is the time at which design $V_{i_{j+1}}$ was finished in OPT. Thus, sculpting according to the ordering OPT^* takes total time at most that of OPT, which gives us a contradiction as discussed earlier.

Problem 2 (Intervals). Let X be a set of n intervals on the real line labeled $1, \ldots, n$. We say that a subset of intervals $Y \subseteq X$ *covers* X if the union of all intervals in Y is equal to the union of all intervals in X. The *size* of a cover is just the number of intervals.

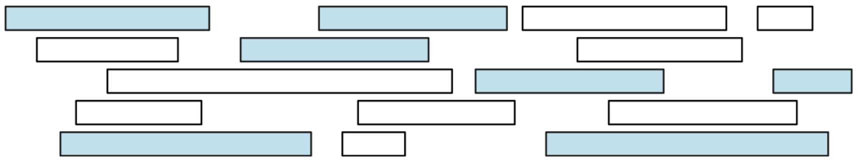

Figure 1: A set of intervals, with a cover (shaded) of size 7 (not necessarily the optimal cover).

Describe a greedy $O(n \log n)$-time algorithm to compute the smallest cover of X. Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, where $(L[i], R[i])$ is the i-th interval in X. Note that X as given is ordered arbitrarily. For simplicity, assume that no two endpoints of intervals are equal; that is, all elements of L, R are distinct. Prove the correctness of the algorithm (including an exchange argument) and analyze its runtime complexity.

Solution 2. We use a greedy algorithm that scans intervals from left to right. On encountering a left endpoint, if we have no chosen interval overlapping at this point, we must choose this interval. Otherwise, we search among all intervals with left endpoint before the right endpoint of the chosen interval overlapping this point and select such interval with farthest right endpoint.

```
procedure COVER($L, R, n$)
    Let $Q$ be a queue of $(L[i], R[i], i)$ tuples ordered from least to greatest $L[i]$
    Let $C$ be an empty list
    $r = -\infty$
    while $Q$ not empty do
        $(\ell_i, r_i, i) = Q$.remove()
        if $\ell_i > r$ then
            $C$.append($i$)
            $r = r_i$
        else if $r_i > r$ then
            while next in $Q$ with left endpoint at most $r$ do
                $(\ell_j, r_j, j) = Q$.remove()
                if $r_j > r_i$ then
                    $(\ell_i, r_i, i) = (\ell_j, r_j, j)$
            $C$.append($i$)
            $r = r_i$
        else
            Continue
    return $C$
```

Correctness. Let $x_1, x_2, \ldots, x_m$ be the intervals chosen by the greedy algorithm sorted from least to greatest left endpoint. We will argue by induction that for all $1 \leq k \leq m$ there is an optimal schedule that chooses $x_1, x_2, \ldots, x_k$.

For the base case, it is clearly necessary to choose the interval with the least left endpoint in order to cover the union of all intervals. For the inductive hypothesis (IH), suppose there is an optimal solution of the form $x_1, x_2, \dots, x_{k-1}, y_k, y_{k+1}, \dots, y_{m^*}$ where m^* is the number of intervals used in this optimal solution and again the intervals are ordered from least to greatest left endpoint. There are two cases.

1. If $L[x_k] > R[x_{k-1}]$ then $y_k = x_k$, otherwise the optimal solution does not cover all of x_k using only intervals with greater left endpoints.
2. Otherwise $L[x_k] < R[x_{k-1}]$, and we can further observe that $R[x_{k-1}] < R[x_k]$ from line 10 of the algorithm. In order to cover x_k, it must be that $L[y_k] < R[x_{k-1}]$. The algorithm chooses the interval with the greatest R, so $R[x_k] > R[y_k]$. Then $x_1, x_2, \dots, x_{k-1}, x_k, y_{k+1}, \dots, y_{m^*}$ must also be a valid optimal solution.

Runtime. The algorithm takes $O(n \log(n))$ to sort the n elements initially with, for example, mergesort. Thereafter, every iteration of the (nested) while loop takes constant time and removes an element from Q which initially has n elements, so a total of $O(n)$ time. The runtime is therefore dominated by the sorting, $O(n \log(n))$.

Problem 3 (Main Street). Flatville, IL is a small town with a single street, Main Street, which runs through it from east to west. At the center of town there are n contiguous, rectangular buildings along the south side of Main Street, each exactly 30 feet wide and 80 feet deep, which overlook the town square on the north side of the street.

The mayor of Flatville thinks the shadows of the buildings make an unsightly, jagged shape on the lawn of the town square due to the varying heights of the buildings, especially when the sun is directly overhead at noon. For their first act as mayor, they have declared that the buildings along Main Street must be remodeled by either reducing or increasing the height of the buildings in order to make an aesthetically pleasing shadow. For example, the mayor would ideally make buildings exactly the same height, say, 52 feet tall. However, this would require drastic remodeling of much taller and much shorter buildings. Instead, the mayor wishes to minimize the pairs of adjacent buildings with different heights, allowing a maximum change of $\delta > 0$. That is, the height of each building may be reduced by at most δ or increased up to at most δ. See the figure below.

Let $H[1..n]$ be an array of integral heights of the buildings in their order on Main Street (that is, $H[i] > 0$ is the height of the i-th building from east to west, and let $\delta > 0$ be a parameter. Describe and analyze an $O(n)$-time algorithm that reports the fewest possible number of adjacent pairs of buildings with different heights after adjusting their heights by up to $\pm\delta$. See the figure below.

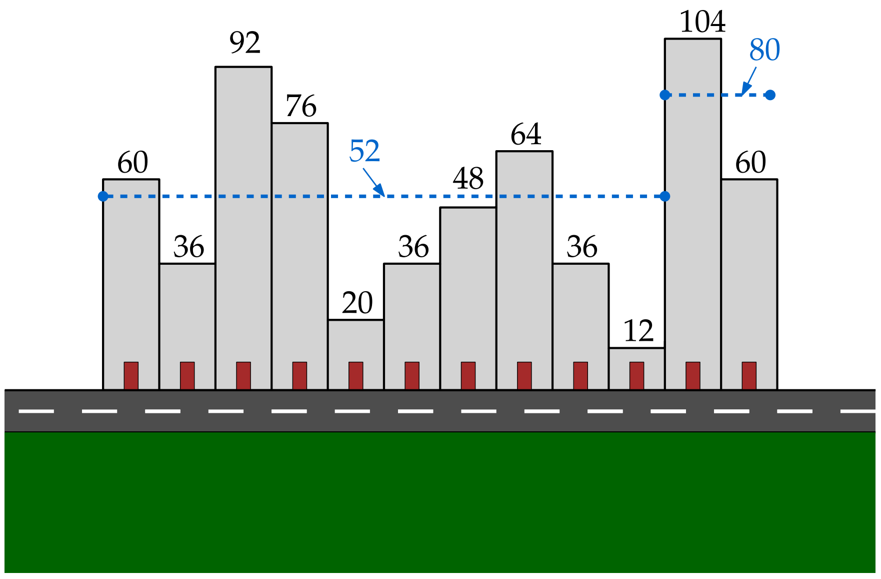

Figure 2: $n = 12$ buildings on Main Street overlooking the lawn of the town square (not drawn to scale). The original heights of the buildings are shown above in black, and a candidate solution for $\delta = 45$ is shown in blue, where the dotted line above/below the top of building denotes its new height. This candidate solution has $k = 1$ which is the best possible for $\delta = 45$.

Solution 3. **Informal Heuristic/Intuition:** The idea is find the longest prefix of the buildings which can all be made a same new height without incurring any absolute change in height greater than δ. Informally, we greedily take as many buildings from the left "as possible," make them the same height, then repeat starting with the leftmost of the remaining buildings.

Algorithm:

- We first initialize a counter that is set to 0. This will store the fewest possible number of adjacent pairs of buildings by the end of the algorithm.
- Then, we iterate through the array of heights (starting from the first building), dynamically updating the maximum h^+ and minimum heights h^- encountered until then.
- If $\lceil \frac{h^+ - h^-}{2} \rceil > \delta$, we reset h^+, h^- to the height of the current building, and increment the counter, starting a new segment.
- We output the value of counter after iterating through all the buildings.

To help illustrate the algorithm, here is the pseudocode.

```
1: procedure COVER($L, R, n$)
2:     $k \leftarrow 0$
3:     $h^- \leftarrow H[1]$
4:     $h^+ \leftarrow H[1]$
5:     Let $C$ be an empty list
6:     for $i$ from 2 to $n$ do
7:         $h^+ = \max(h^+, H[i])$
8:         $h^- = \min(h^-, H[i])$
9:         if $\lceil \frac{h^+ - h^-}{2} \rceil \} > \delta$ then
10:            $k \leftarrow k + 1$
11:            $h^+ = H[i]$
12:            $h^- = H[i]$
13:    return $k$
```

Proof of Correctness:

- Let $ALG = [a_1, a_2, \ldots, a_k]$ be the increasing sequence of indices where "switches" occurs in our solution, i.e., the sequence of values of variable i of the for-loop in which k is incremented. Similarly, let $OPT = [o_1, o_2, \ldots, a_m]$ be the increasing sequence of indices where such switches occur in a fixed, arbitrary optimal solution. ALG can only equal or more switches than any optimal solution, so $k \geq m$. For simplicity of the proof, we set $a_0 = o_0 = 0$, denoting an artificial "switch" that occurs before the first building, and set $a_{k+1} = o_{m+1} = n$, denoting an artificial "switch" that occurs after the last building. Lastly, we introduce the notation $B[i \ldots j]$ to be the sequence of buildings from indices i to $j - 1$, and $H[i \ldots j]$ to be their sequence of heights in H. If $i \geq j$, then these sequences are empty.

  Firstly, it is clear that our algorithm outputs a valid solution. This is because we can divide the buildings into $\alpha + 1$ segments, $B[(a_0 = 1)..a_1], B[a_1..a_2], \ldots, B[a_k..(a_{k+1} = n + 1)]$, with start/end points corresponding to when we incremented k, and the IF condition ensures that with h^+, h^- as the maximum, minimum building heights in that segment, then $h^+ - h^- \leq 2\delta$, and thus it is feasible to change the heights of buildings in an interval $B[a_i, a_{i+1}]$ to height $\frac{h^+ - h^-}{2}$.

  Next we prove ALG is optimal: $k = m$. If OPT contains all elements of ALG then ALG is optimal, since OPT cannot have more switches than our valid solution ALG. So suppose not, i.e., OPT does not contain all indices of ALG. Let a_i be the first index that is in ALG which is not in OPT. By assumption, $a_j = o_j$ for all $j < i$, and $a_i \neq o_i$. There are two cases to consider:

- Case $a_i < o_i$. Then the optimal solution puts all buildings in $B[a_{i-1}, o_i]$ at a common new height; in particular, this includes all buildings in $B[a_{i-1}, a_i]$. However, since the algorithm incurs a switch at index a_i, we have that the difference between the heights of the highest and shortest buildings in this range exceeds 2δ.; i.e., $\max B[a_{i-1}, a_i] - \min B[a_{i-1}, a_i] > 2\delta$. Thus, it is impossible for the optimal solution to put these buildings at a new height within δ of all these buildings, a contradiction.
- Case $a_i > o_i$. Consider the solution $OPT' = [a_1, a_2, \ldots, a_i, o_{i+1}, \ldots, o_m]$ obtained by moving the i-th switch of OPT at index o_i to instead occur at index a_i. We have argued above that all buildings in $B[a_{i-1}..a_i]$ can be made a new common height within δ of their original heights. Furthermore, all buildings in with indices o_{i+1} and greater remain able to be resized, per the original optimal solution. Thus, it remains to show that all buildings in $B[a_i..o_{i+1}]$ can be made a new common height within δ of their original heights. This is clearly true, as the optimal solution corresponding to OPT made all of the buildings in $B[o_i..o_{i+1}]$ to a common feasible height, which contains $B[a_i..o_{i+1}]$ as $a_i > o_i$.

  We conclude that, by the exchange argument above, there is an optimal solution whose switches occur at the indices of ALG, a valid solution, and thus the algorithm is optimal.

Runtime:
The runtime is $O(n)$, as we simply iterate through the array once and do $O(1)$ many operations at each iteration.

Problem 4 (Applied). **Autograder available 10/8 @ 5pm**
The NotUnited airline company from Homework 2 is expanding from its m *existing* locations with n *new* locations, by adding pairs of connecting flights. The input consists of two arrays, $A[1..m]$ and $B[1..n]$, of non-negative integers where $A[i] \geq 0$ is the number of new connections required by the existing i-th location and $B[j] \geq 0$ is the number of new connections required by the new j-th location.

The goal is to compute $m \times n$ matrix $F[1..m, 1..n]$ where $F[i, j] = 1$ if there is a new flight between the i-th existing location and j-th new location and $F[i, j] = 0$ indicates no such new flight, that meets the new flight constraints (or otherwise indicate no such matrix exists). In particular, F must meet the following constraints:

- For all i, $1 \leq i \leq m$, the i-th existing location has $\sum_{j=1}^{n} F[i, j] = A[i]$ connections, and
- For all j, $1 \leq j \leq n$, the j-th new location has $\sum_{i=1}^{n} F[i, j] = B[j]$ connections.

For example, with $A = [1, 3, 2]$ and $B = [2, 2, 2]$, a valid matrix F is:

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

However, with $A = [1, 3, 2]$ and before but $B = [0, 3, 3]$, there is no solution.

The empirical running time of your solution will be compared to an algorithm with worst-case asymptotic running time $O(mn \log(m + n))$. Consider what a greedy strategy could look like in the context of this problem in order to populate the matrix F appropriately. Language-specific details follow. You can use whichever of Python or Java you prefer. You will receive automatic feedback when submitting, and you can resubmit as many times as you like up to the deadline.

- **Python.** You should submit a file called `not_united.py` to the Gradescope item "Homework 5 (Python)." The file should define (at least) a top level function `notunited` that takes as input two lists of ints, `A` and `B`. The function returns a list of lists `F:list[list[int]]` that represents matrix F, or returns `None` if no such matrix exists. The function header is:

```
def not_united(A:list[int], A:list[int]):  -> list[list[int]]
```

- **Java.** You should submit a file called `NotUnited.java` to the Gradescope item "Homework 5 (Java)." The file should define (at least) a top level function `NotUnited` that takes in two `int[]` that represent arrays A and B. The function returns an `int[][]` that represents matrix F, or returns `null` if no such matrix exists. The method header is:

```
public int[][] notUnited(int[] a, int[] b);
```

Solution 4 . Each i-th row much receive exactly $A[i]$ 1's. We iterate through the rows, from row 1 to row m, and choose in which columns to place the 1s. We maintain the remaining number of 1s to make in each column by modifying B. To fill in the i-th row, we greedily choose the $A[i]$ columns with greatest values in B. If there are fewer than $A[i]$ with positive values, we return "impossible." Otherwise, we place 1s in the selected columns, decrement the counts for these columns in B, then proceed to the next row.