CompSci 330 Design and Analysis of Algorithms
Assignment 7, Fall 2024 Duke University

TODO: Add your name(s) here

Due Date: November 11, 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 (High Score!). The *score* of a sequence of integers $b_1, b_2, \ldots, b_\ell$ is defined as

$$\sum_{1 \le i < \ell} (b_i - b_{i+1})^2.$$

Describe and analyze an $O(n^2 k)$-time dynamic programming algorithm to compute the highest possible score over all subsequences of length at most k in a given array $A[1..n]$ of integers, where k is a given parameter. For example, $1, 9, 3$ is a subsequence of $[3, \underline{1}, 5, 6, \underline{9}, 7, 4, \underline{3}, 8]$ of length 3 with score $(1-9)^2 + (9-3)^2 = 64 + 36 = 100$, which is the highest-scoring 3-length subsequence in the array. Note that the items in a subsequence need not be contiguous in the original array.

Solution 1. Let $HCS(i, j, \ell)$ be the score of the highest-scoring subsequence of $A[j..n]$ of length at most $\ell \ge 0$ that begins with $A[j]$ and its second element (if any) is in $A[i..n]$, where $i > j$. $\max_{1 \le i \le n} HCS(i+1, i, k)$ solves the given problem.

$$HCS(i,j,\ell) = \begin{cases} 0 & \text{if } i > n \text{ or } \ell \le 1 \\ \max \begin{cases} (A[i] - A[j])^2 + HCS(i+1, i, \ell-1), \\ HCS(i+1, j, \ell) \end{cases} & \text{otherwise} \end{cases}$$

We next use dynamic programming to obtain an efficient algorithm. We memoize the function in a $n \times n \times (k+1)$ array M, where $M[i][j][\ell]$ stores $HCS(i, j, \ell)$. The evaluation order is given by the following nested for-loops:

```
for j from n downto 1:
   for i from n downto j+1:
      for l from 0 to k:
         <eval. M[i][j][l]>
```

There are $O(n^2 k)$ subproblems and each is computed in $O(1)$ time with memoization, so the overall runtime is $O(n^2 k)$.

Alternate Solution 1. Let $HCS(i, \ell)$ be the score of the highest-scoring subsequence of $A[i..n]$ of length at most $\ell \ge 0$ that begins with $A[i]$. $\max_{1 \le i \le n} HCS(i, k)$ solves the given problem.

$$HCS(i,\ell) = \begin{cases} 0 & \text{if } i \ge n \text{ or } \ell \le 1 \\ \max_{i < j \le n} \left\{ (A[i] - A[j])^2 + HCS(j, \ell - 1) \right. & \text{otherwise} \end{cases}$$

We next use dynamic programming to obtain an efficient algorithm. We memoize the function in a $n \times (k+1)$ array M, where $M[i][\ell]$ stores $HCS(i, \ell)$. The evaluation order is given by the following nested for-loops:

```
for i from n downto 1:
   for l from 0 to k:
      <eval. M[i][l]>
```

There are $O(nk)$ subproblems and each is computed in $O(n)$ time with memoization, so the overall runtime is $O(n^2 k)$.

Problem 2 (The Inevitable Board-Game-Related Question) The sequel to the worldwide classic board game *Snakes and Ladders* is here: *Just Ladders*! The game of *Just Ladders* is not a race to the finish nor does it involve any randomness by way of dice rolling or spinner spinning; instead, it is a game about collecting as many points as possible by the time your token reaches the end of the board.

A game of *Just Ladders* consists of a $\sqrt{k} \times \sqrt{k}$ board, where $k > 0$ is a perfect square, that has $n \geq 1$ "ladders." The spaces are numbered so that there is a path along adjacent spaces starting from space 1 to space k, and each space i may have at most one ladder leading up from that space to a higher space j, $j > i$. A ladder leading up from a space i is associated with a number of points, $p_i > 0$. Note that a space may have many ladders leading into it but at most one leading up from it.

As a player of the game, you begin with your token on space 1. Then you repeat the following until your token reaches the last space on the board, space n:

1. You can always move your token from its current space i to a later space $j > i$, for no gain in points.
2. Additionally, if your current space i has a ladder *leading up* from that space to another space $j > i$, you can use the ladder to move your token to space j and gain p_i points for using it.

Formally, a board for *Just Ladders* consists of a perfect square integer k, denoting the size of the board, and three arrays $B[1..n]$, $T[1..n]$, and $P[1..n]$, that represent the $n \geq 1$ ladders: B is a sorted (potentially non-contiguous) subsequence of $[1, 2, \ldots, n]$ such that there is a ladder leading up from space $B[i]$ to space $T[i]$, $B[i] < T[i]$, that awards $P[i]$ points for using the ladder, for each i from 1 to n.

Design and analyze an efficient dynamic programming to compute the maximum number of points that is possible to obtain given board size k and arrays B, T, and P. The target runtime is $O(n \log n)$. *[Hint: Don't aim for the target runtime from the start. First obtain a* ***correct*** *solution, and* ***then*** *consider how to improve the runtime.*

Solution 2. Note that there is no opportunity to gain any points before the space $B[1]$, starting from the first space (which may be $B[1]$). Let $MS(i)$ denote the maximum score obtainable in *Just Ladders* starting at board space $B[i]$. We want to find $MS(1)$.

We can either take the i-th ladder or not. If we choose not to take it, we want the best score possible using the remaining ladders $n-i$ ladders, and the next opportunity to possibly gain points is when we reach space $B[i+1]$, the bottom of the next ladder. If we take the i-th ladder, we gain $P[i]$ points then want the best score possible using the ladders whose bottom space is at or after the top space of the i-th ladder, $T[i]$. That is, if $X(i)$ is the minimum index such that $B[X(i)] \geq T[i]$ (if there is no such index, set $X(i) = n+1$), we want the maximum score starting from space $B[X(i)]$, the next space at which there is an opportunity to gain points (by taking ladder $B[X(i)]$ or moving past it). Thus, MS is described by the following recurrence:

$$MS(i) = \begin{cases} 0 & \text{if } i > n \\ \max\{MS(i+1), P[i] + MS(B[X(i)]))\} & \text{otherwise} \end{cases}$$

We compute $X(i)$ for each $i \in \{1, \ldots, n\}$ using binary search on the array B, as its entries are sorted. This takes $O(n \log n)$ time in total.

We now use dynamic programming. We use a 1D array $M[0..n]$ as our memoization data structure, where we store $MS(i)$ in $M[i]$ for each i. We fill M in decreasing order, from $i = n$ to 1. There are $O(k)$ subproblems in total, and evaluating each one (after preprocessing the $X(i)$'s) takes $O(1)$ time with memoization. Therefore the overall runtime is $O(n \log n)$, dominated by the time to compute the $X(i)$'s. As mentioned above, to solve the original problem, we compute $MS(1)$.

Problem 3 (Reluctant Employees) The corporate structure at $\mathbb{Y}$, a recently-renamed social media company, can be represented by a rooted tree, where the employees are nodes and the child nodes of an employee are other employees that directly report to them. The CEO, at the root of the tree, does not report to anyone. The CEO wishes to pick a subset of employees to attend a publicity event so that, for each employee, either that employee or to whom that employee reports must attend the event. However, no employee wants to attend the event, and they are only willing to go if they are paid extra. Each employee has a price (\$) they are willing to be paid in order to attend the event (including the CEO). Your task is to select a subset of employees to attend the event so that, for any employee, either that employee attends or the employee to whom they report attends (possibly both), while paying the minimum total dollars.

Formally, let T be a rooted tree with n nodes that represents the corporate structure, where the CEO correspond to the root node, $c \in T$. Each employee (node) $v \in T$ stores the price $p(v) \geq 0$ that the employee is willing to accept to attend the event, and the list $\ell(v)$ of $r(v)$ employees (nodes) that report directly to employee v. Any employee v without other employees that report to them has $r(v) = 0$ and empty list $\ell(v)$. Describe an efficient dynamic programming algorithm that computes a suitable subset of employees to attend the event that has minimum total price. The target runtime is $O(n)$.

Solution 3. For a vertex $v \in T$, let $g(u)$ be the set of grandchildren of u in T; that is,

$$g(u) = \bigcup_{e \in r(u)} r(e).$$

Let $MPE(u)$ be the minimum total price of a suitable guest list among the employees in the subtree rooted at employee u, *where, if u is not the CEO, then the parent of u is already included in the guest list.* The function is described by the following recurrence:

$$MPE(u) = \begin{cases} 0 & u \text{ is a leaf} \\ \min \begin{cases} p(u) + \sum_{v \in r(u)} MPE(v) \\ \sum_{v \in r(u)} p(u) + \sum_{w \in g(u)} MPE(w) \end{cases} & \text{otherwise} \end{cases}$$

The top case in the min function corresponds to the decision to include u in the event list, so their children may or may not be included. The bottom case corresponds to the decision not to include u in the event list, which requires that all of its children are included, and as a result any of its grandchildren may or may not be included.

Let c be the CEO (root of the tree). To solve the original problem **allowing the CEO not to be included**, then $MPE(c)$ is the solution. To solve the original problem **requiring the CEO to be included**, then $\sum_{e \in r(c)} MPE(e)$ is the solution.

We memoize using the vertices of the tree, storing $MPE(u)$ on vertex u itself, for each $u \in T$. We evaluate in a postorder traversal. There are $O(n)$ subproblems, and computing $MPE(u)$ takes $O(1 + |r(u)| + |g(u)|)$ time, where the $O(1)$ to account for work when $r(u) = g(u) = \varnothing$. Every employee is the child and grandchild of at most one vertex, so the total runtime is

$$O\Big(\sum_{u \in T} (1 + |r(u)| + |g(u)|)\Big) = O(n).$$

Problem 4 (Spelling Bee) Let $G = (V, E)$ be a directed, acyclic graph (DAG) where each edge $e \in E$ is associated with a letter $\ell(e)$. For any path P in G with edges, $e_1, e_2, \ldots, e_k$, P *spells* the string $\ell(e_1)\ell(e_2)\cdots\ell(e_k)$.

Describe and analyze an $O(V^2 + E^2)$-time algorithm that computes the length of the longest path in a given graph G, of the form described above, that spells a palindrome. Note that the endpoints of the path are not specified. **As part of your DP solution, include an explicit specification for the subproblems that your recurrence relation / pseudocode solves, and explain each of its cases in English.** Otherwise, the reader is unable to understand the more general problem you are solving and follow its correctness. Of course, also state how to solve the original problem. Then analyze its runtime.

[Hint: Compute a topological ordering of G at the start, then use it later to define a valid evaluation order of your subproblems.]

Solution 4. English specification:

Fix a topological ordering of G computed in $O(V + E)$ time. For simplicity, we refer to the i-th vertex in the ordering as i itself. Recall that a topological sort of a DAG ensures that for all edges $(i, j) \in E$, $i < j$.

Let $\textsc{LongestPalindrome}(i, j)$ be the length of the longest path from node i to node j in G that spells a palindrome. We solve the original problem by calling $\textsc{LongestPalindrome}(0, n)$.

Recurrence: We define the recurrence as follows:

$$\textsc{LongestPalindrome}(i,j) = \begin{cases} -\infty & \text{if } i > j \\ 0 & \text{if } i = j \\ \max \begin{cases} 1, \\ \displaystyle\max_{\substack{(i,h),(k,j)\in E: \\ \ell(i,h)=\ell(k,j)}} 2 + \textsc{LongestPalindrome}(h,k) \end{cases} & \text{if } (i,j) \in E \\ \displaystyle\max_{\substack{(i,h),(k,j)\in E: \\ \ell(i,h)=\ell(k,j)}} 2 + \textsc{LongestPalindrome}(h,k) & \text{otherwise} \end{cases}$$

where max over the empty set is defined as $-\infty$. To explain the recursive cases, the longest palindrome from i to j has multiple cases: if there is an edge from i to j then the path consisting of this edge spells a trivial length-1 palindrome, and otherwise, there may be a longer palindrome from i to j which begins with an outgoing $(i, h) \in E$ and ends with an incoming edge $(k, j) \in E$ provided the letters on those edges match.

The answer to the original problem is given by $\max_{i,j \in V} \textsc{LongestPalindrome}(i, j)$. We will now use **dynamic programming** with memoization to solve the problem.

Memoization structure: We define the memoization structure as a $|V| \times |V|$ array M. We initialize $M[i][i] = 0$, and all other entries to $-\infty$.

Evaluation order: We fill the rows in reverse topological order (for i from $n - 1$ down to 1), and fill each row in topological order (for j from $i + 1$ to n).

Runtime analysis: Computing the topological ordering of G takes $O(V + E)$ time.

Recall that the *in-degree* (resp., *out-degree*) a vertex i is the number of incoming (resp., outgoing) edges at i. There are $O(V^2)$ subproblems, and each subproblem $\textsc{LongestPalindrome}(i, j)$ takes $O(\text{out-degree}(i) \cdot \text{in-degree}(j))$ time in the worst-case, as there at most that many pairs of incident edges leaving node i and entering j when computing the max function in the recursive case. Since

$$\sum_{i,j \in V} \text{out-degree}(i) \cdot \text{in-degree}(j) = \sum_{i \in V} \text{out-degree}(i) \sum_{j \in V} \text{in-degree}(j) = |E| \sum_{i \in V} \text{out-degree}(i) = |E|^2$$

we have that the overall time to evaluate the subproblems is $O(V^2 + E^2)$.

Therefore, the total runtime is $O(V + E) + O(V^2 + E^2) = O(V^2 + E^2)$.

Problem 5 (Sorting in Expected Linear Time) For this problem, consider the following setting: We have n numbers coming from a universe $[m] = \{0, 1, ..., m-1\}$, where m is a multiple of n; that is, $m = kn$ for some integer $k \geq 1$. Unlike in lecture and recitation, we assume here that the n numbers come from a uniform distribution on $[m]$, that is, each number are chosen independently such that it has equal probabilities of becoming any one of $1, 2, ..., m$.

(a) Find a function $h : [m] \to [n]$ that satisfies both of the following:

- $\Pr[h(x) = a] = 1/n$ for all $a \in [n]$

- if $x \leq y$, then $h(x) \leq h(y)$

Note that x is the randomly chosen input, and the probability is over the choice of x.

Solution. We can see that exactly $k = m/n$ numbers should be mapped to each $a \in [n]$. Also, h should be nondecreasing, so the smallest k numbers should be assigned to 0, the second smallest k numbers should be assigned to 1, and so on. This gives us that $h(x) = \lfloor x/k \rfloor = \lfloor xn/m \rfloor$.

(b) Our strategy is to put the n input numbers $x_1, x_2, ..., x_n$ into n buckets, with h computing which bucket the number should belong to, and each bucket maintained as a linked list with each node containing the original value x_i. Then, we will do insertion sort on each bucket based on the x_i values contained in the nodes, and finish by concatenate all the buckets in order. Show the correctness of this algorithm, and prove that it has expected runtime $O(n)$.

Solution. If bucket i has n_i elements, insertion sort takes $\leq Cn_i^2$ time. To show that this algorithm runs in $O(n)$ total expected time, it suffices to show that $E[n_i^2]$ is $O(1)$.

Define indicator variable $x_{i,j} = \mathbf{1}[$ item j is assigned to bucket $i]$. We can express $n_i = \sum_{j=1}^{n} x_{i,j}$ and find the expectation for n_i^2:

$$\begin{aligned} E[n_i^2] &= E[(\sum_{j=1}^{n} x_{i,j})^2] = E[\sum_{j=1}^{n} x_{i,j}^2 + \sum_{j \neq j'} x_{i,j} x_{i,j'}] \\ &= \sum_{j=1}^{n} E[x_{i,j}^2] + \sum_{j \neq j'} E[x_{i,j} x_{i,j'}] = n\frac{1}{n} + n(n-1)\frac{1}{n^2} = 2 - \frac{1}{n} = O(1) \end{aligned}$$

The expectations explained: $x_{i,j}$ is 1 with probability $1/n$ because each input element is equally likely to be in any bucket; also $x_{i,j} = x_{i,j}^2$ since it is a 0-1 variable. Therefore, $E[x_{i,j}] = 1/n$.

$x_{i,j} x_{i,j'}$ is 1 when jth and j'th both map to bucket i. Since items are mapped independently, the probability that both are mapped to bucket i is $1/n^2$. Therefore, $E[x_{i,j}] = 1/n^2$.

Problem 6 (Etsy Empire) Freddie is the owner of Freddie's Patterned Mousepads (FPM), the premier vendor on Etsy for custom-sized neoprene mousepads. No matter how small or how large a mousepad you need (in whole units), FPM has you covered. Every one of their mousepads features the same world-renowned signature pattern: down-sloping stripes in Duke Navy Blue (hex code #012169) on solid grey.

To make the mousepads, Freddie begins with a single large rectangular $a \times b$ neoprene sheet, for some integers $a, b \geq 1$. Using his cutting machine, he can make either a vertical or horizontal cut across the sheet, dividing it into two smaller rectangular sheets. After a sequence of cuts, he has a pile of various rectangular sheets, which he finishes into mousepads by stiching up the sides then posting them for sale on his Etsy store. However, the prices his customers are willing to pay for different sized mousepads seem to be completely arbitrary; for example, a 1×7 mousepad may sell for \$37 and a 2×7 mousepad may only sell for \$4. In particular, because of the pattern, a 1×7 mousepad and 7×1 mousepad look different and may sell for wildly different prices. Overwhelmed, Freddie is asking for your help to figure out the most money he can make from a single $a \times b$ neoprene sheet.

Design and implement a dynamic programming algorithm that solves Freddie's problem for inputs a, b, P where a, b denote the dimensions $a \times b$ of the initial neoprene sheet and $P[0..a][0..b]$ is a 2D array such that $P[i, j]$ stores the price for which an $i \times j$ mousepad will sell, for any integers $0 \leq i \leq a$ and $1 \leq j \leq b$. P is such that $P[0][j] = P[i][0] = 0$ for all i, j–Freddie cannot sell a mousepad with zero width or height! The empirical running time of your solution will be compared to a reference solution with asymptotic runtime $O(ab(a + b))$.

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 `mousepads.py` to the Gradescope item "Homework 7 (Python)." The file should define (at least) a top level function `max_revenue` that takes in a, b as ints and 2D list P of floats, as defined above, and returns the maximum possible revenue.

  The function header is:

  ```
  def max_revenue(a:int, b:int, P:list[list[float]]):  -> float
  ```

- **Java.** You should submit a file called `Mosuepads.java` to the Gradescope item "Homework 7 (Java)." The file should define (at least) a top level function `maxRevenue` 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 double maxRevenue(int a, int b, double[][] P);
  ```

$$P = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 12 & 13 & 3 & 7 \\ 0 & 8 & 27 & 30 & 11 & 9 \\ 0 & 8 & 7 & 0 & 8 & 16 \end{bmatrix}$$

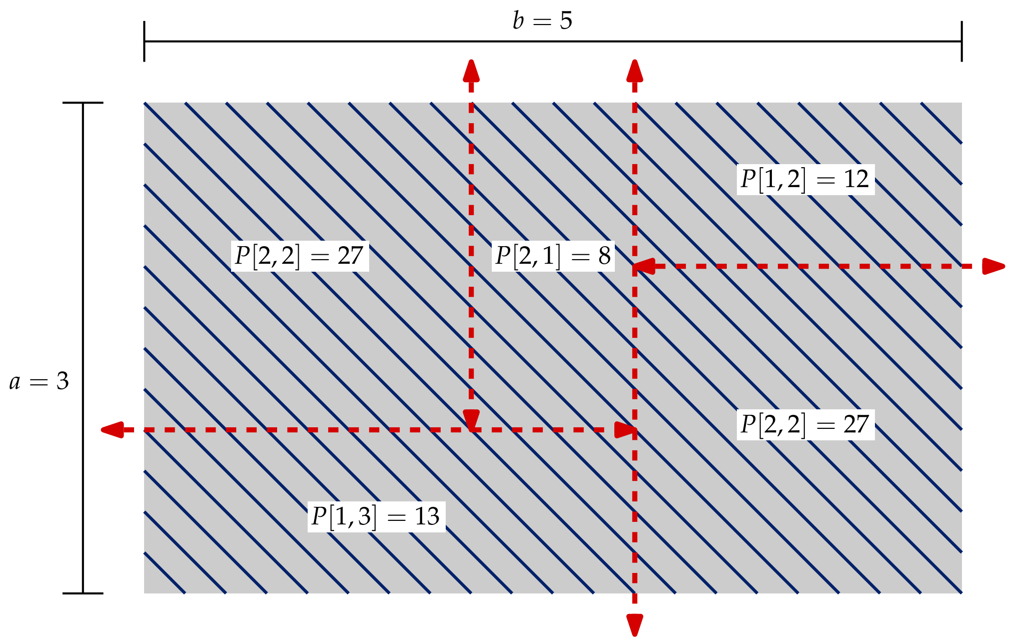

Figure 1: Example of four cuts on an initial 3×5 sheet, starting with the vertical cut across the entire sheet. The resulting five mousepads sell for $27 + 8 + 12 + 13 + 27 = 87$ dollars, following the given price array $P[0..3][0..5]$ below, which is the best possible for these prices.

Solution 6. For any sheet of size $i \times j$, there are $i-1$ choices for a first horizontal cut, $j-1$ choices for a first vertical cut, and the choice not to cut at all for a revenue of $P[i, j]$. Letting $MaxRev(i, j)$ be the maximum revenue possible on a sheet of dimensions $i \times j$, this leads to a simple recurrence relation. In the end, we wish to output $MaxRev(a, b)$ for the initial sheet $a \times b$.