Compsci 330 Design and Analysis of Algorithms
Assignment 4, Spring 2024 Duke University

TODO: Add your name(s) here

Due Date: Thursday, February 15, 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 large language models is unlikely to help you prepare for exams. See the course policies webpage for more details.

Grading. Theory problems will be graded by TAs on an S/U scale (for each sub-problem). Applied problems typically have a separate autograder where you can see your score. The lowest scoring problem is dropped. See the course assignments webpage for more details.

[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 (Parentheses). Suppose you are given a sequence of $n+1$ nonnegative real numbers in an array $A[\,]$ of size $n+1$ separated by n arithmetic operations $+$ and $\times$ in array $OP[\,]$ of size n where $OP[i]$ comes between $A[i]$ and $A[i+1]$. For example, the input arrays $A = [0.5, 2, 3, 5, 4]$ and $OP = [\times, +, +, \times]$ represent the following expression:

$$0.5 \times 2 + 3 + 5 \times 4$$

You can change the value of this expression by placing parentheses. For examples:

$$0.5 \times (2 + 3 + 5) \times 4 = 20$$
$$(0.5 \times 2) + 3 + (5 \times 4) = 24$$
$$(0.5 \times 2) + ((3 + 5) \times 4) = 33$$

(a) Let $V(i, j)$ return the maximum value obtained by placing parentheses between $A[i]$ and $A[j]$ (inclusive). For the previous example, $V(2, 4) = (3 + 5) \times 4 = 32$. Give a recurrence to compute $V(i, j)$. Also briefly explain each case of your recurrence in words.

(b) Describe an iterative dynamic programming algorithm to compute the maximum possible value the given expression can take by placing parentheses. Analyze the algorithm's runtime complexity and space complexity (that is, memory).

Solution 1.

(a)

$$V(i,j) = \begin{cases} A[i] & \text{if } i = j \\ \max_{i \le k < j} \left(V(i,k) \;\; OP[k] \;\; V(k+1,j) \right) & \text{otherwise.} \end{cases}$$

In the the base case ($i = j$) there are no operations to perform so just return the value of the given element. Otherwise, note that some operation must be performed last. We can search over all possible last operations, in each case using the maximum value obtainable by any placement of parentheses to the left and right of the last operation.

(b) We can solve these subproblems iteratively, noting that $V(i, j)$ depends on subproblems $V(i', j')$ where $i' \ge i$ and $j' \le j$, that is, smaller ranges.

```
1:  procedure MaxExp(A, OP, n)
2:      Initialize n + 1 by n + 1 array V[ , ] to 0s
3:      for i = 0 to n do
4:          V[i, i] = A[i]
5:      for g = 1 to n do
6:          for i = 0 to n − g do
7:              j = i + g
8:              for k = i to j − 1 do
9:                  V[i, j] = max(V[i, j], V[i, k]  OP[k]  V[k + 1, j])
10:     return V[0, n]
```

The runtime complexity of the algorithm is $O(n^3)$ from the three nested for looops on lines 5, 6, and 8. The space complexity, however, is only $O(n^2)$ as the V array only has size $O(n^2)$.

Problem 2 (Vertex Cover). You are given an undirected tree T rooted at r with n vertices. The tree may not be binary, but by definition it will be connected and will not have any cycles, implying it will have $n-1$ edges. A vertex cover is a subset of vertices such that for every edge (u, v) of the graph, either u or v is in the vertex cover. An example is drawn in Figure 1. Note that every edge has at least one red endpoint.

You may assume the following:

(i) Given a vertex v, it's unique "parent" is the vertex connected to v and closer to the root. The root r is the unique node with no parent.

(ii) Given a vertex v, it's "child" vertices are all those connected to v and not its parent. You can access the child nodes directly, for example, you can write a for loop over the child nodes of v. A vertex is a "leaf" node if it has no children.

(iii) You can store additional information at a vertex in the graph if you wish while traversing it: i.e., you are allowed auxiliary memory at each node.

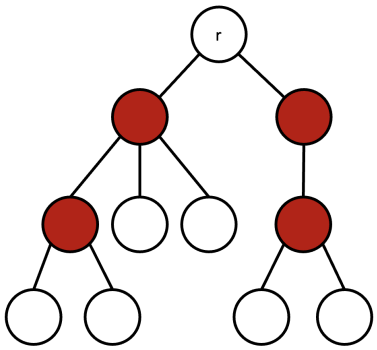

Figure 1: Example tree and vetex cover shown in shaded red.

Describe an algorithm to compute the size of a minimum vertex cover in $O(n)$ runtime. Briefly explain the runtime complexity. Prove the correctness of the algorithm.

Hint. Read Section 3.10 in our text by Erickson on the minimum independent set on trees problem.

Solution 2. Define the following subproblems:

$$\mathsf{DP}(v, 0) = \text{Size of the minimum vertex cover of the subtree rooted at } v, \text{ excluding } v.$$
$$\mathsf{DP}(v, 1) = \text{Size of the minimum vertex cover of the subtree rooted at } v, \text{ including } v.$$

Our goal is to compute and return $\min(\mathsf{DP}(r, 0), \mathsf{DP}(r, 1))$ where r is the root. We consider the following cases in developing a recurrence relation.

- If ℓ is a leaf node,

$$\mathsf{DP}(\ell, 0) = 0,$$
$$\mathsf{DP}(\ell, 1) = 1.$$

  That is, the size of a minimum vertex of a subtree rooted at a leaf node just depends on whether or not we select the leaf node.

- For another vertex v, not a leaf, let For each vertex v, let $\mathcal{C}_v$ be the set of child nodes of v.

$$\mathsf{DP}(v, 0) = \sum_{c \in \mathcal{C}_v} \mathsf{DP}(c, 1)$$

  This is because, if we do not select vertex v, the only way for us to cover the edges going out of v is to select all of the child nodes of that vertex. Also,

$$\mathsf{DP}(v,1) = 1 + \sum_{c \in \mathcal{C}_v} \min(\mathsf{DP}(c,0), \mathsf{DP}(c,1))$$

This is becauze, by selecting the root node v, we cover all edges (v,u), where $u \in \mathcal{C}_v$. This allows us to have the choice of whether we select a child node in our vertex cover or not. This is equivalent to considering separately the optimal vertex cover for each subtree stemming from a child node of v. The minimum size of a vertex cover for $u \in \mathcal{C}_v$ is simply $\min(\mathsf{DP}(u,0), \mathsf{DP}(u,1))$. We then sum over all child nodes to obtain the above formula for $\mathsf{DP}(v,1)$.

To compute $\mathsf{DP}(v,0)$ and $\mathsf{DP}(v,1)$, we need to already have the DP values for all the children of vertex v. To ensure this, we compute subproblems in a post-order traversal of the tree (meaning we do not visit a vertex until we have visited all nodes in subtrees rooted at its children). For the sake of simplicity here we assume that subproblem solutions $\mathsf{DP}(v,0)$ and $\mathsf{DP}(v,1)$ are stored as global arrays with the vertices labeled by integers $1...n$. It is also fine to store the values as attributes on the nodes of the tree directly.

The pseudocode for the recursive procedure is given below. To obtain the final result, we call TreeMVC(r) and return $\min(\mathsf{DP}(r,0), \mathsf{DP}(r,1))$.

```
1: procedure TreeMVC(v)
2:     DP(v, 0) = 0
3:     DP(v, 1) = 1
4:     for u ∈ C_v do
5:         TreeMVC(u)              ▷ To make sure we have computed DP(u, 0) and DP(u, 1))
6:         DP(v, 0) = DP(v, 0) + DP(u, 1)
7:         DP(v, 1) = DP(v, 1) + min(DP(u, 0), DP(u, 1))
8:     return min(DP(v, 0), DP(v, 1))
```

Runtime and Correctness. The algorithm runtime is $O(n)$ because a given recursive call takes $O(1)$ time and we recurse on each node in the tree just once.

We want to argue that after executing TreeMVC(v), $\mathsf{DP}(u,0)$ and $\mathsf{DP}(u,1)$ are the correct sizes of the minimum vertex cover of the subtree rooted at u excluding u and including u respectively in the vertex cover, for all nodes u in the subtree rooted at v. We argue correctness by induction on n, the size of a subtree. In the base case of a single leaf node, there are no edges to cover, so we simply have $\mathsf{DP}(v,0) = 0$ and $\mathsf{DP}(v,1) = 1$, counting only the selection of the single vertex or not.

For the inductive hypothesis (IH), suppose that after calling TreeMVC(u) for a child node u of v correctly computes the subproblems for all nodes in the subtree rooted at u. Consider the subtree rooted at v. There are two cases:

(i) If the minimum vertex cover rooted at v *does not include* v, then it must include every child of v to be a valid vertex cover. The minimum such cover would then have size equal to the sum of the minimum vertex cover of the subtrees rooted at each child node, including those child nodes. These are summed into $\mathsf{DP}(v,0)$ (see line 6) and each is correctly computed by the IH.

(ii) Otherwise the minimum vertex cover rooted at v does include v. In this case a child node could or could not be included in the vertex cover. The minimum such cover is the minimum

over both choices, added over every child node. These are summed into $\mathsf{DP}(v, 1)$ (see line 7) and each is correctly computed by the IH.

The smaller of the two values $\mathsf{DP}(v, 0)$ and $\mathsf{DP}(v, 1)$ is returned (line 8), so whichever case the minimum vertex cover falls in is returned.

Problem 3 (Palindrome Decomposition). A palindrome is any string that is exactly the same as its reversal, like I, or DEED, or RACECAR. Any string can be decomposed into a sequence of substrings that are each palindromes. For example, the string BUBBASEESABANANA ("Bubba sees a banana.") can be broken into palindromes in the following ways (and many others):

BUB · BASEESAB · ANANA

B · U · BB · ASEESA · B · ANANA

BUB · B · A · SEES · ABA · N · ANA

Describe an algorithm to find the smallest number of palindromes that make up a given input string s of length n in $O(n^2)$ time. For example, given the input string BUBBASEESABANANA, your algorithm should return 3. You do not need to prove correctness, but for any recurrence(s), be sure to clearly define the recurrence and briefly explain every case.

Hint. Divide the algorithm into two parts. First compute, for every substring, whether it is a palindrome, in $O(n^2)$ time using dynamic programming. Second, determine the optimal way to split s into substrings all of which are palindromes.

Solution 3. Suppose the input string is s of length n. Let $s[i : j]$ denote the substring of s starting at the i^{th} character and ending at the j^{th} character (one-indexed, inclusive). Create two arrays, PAL and DP. We solve this problem using dynamic programming twice. Each array is defined as follows:

- $\mathsf{PAL}(i, j) = 1$ if $s[i : j]$ is palindrome and $\mathsf{PAL}(i, j) = 0$ otherwise.
- $\mathsf{DP}(i)$ = The smallest number of palindromes that $s[1 : i]$ could be broken into.

We first show how to compute PAL using dynamic programming by developing a recurrence.

- Base Cases. For each $i \in \{1, \ldots, n\}$, let $\mathsf{PAL}(i, i) = 1$. For each $i \in \{1, \ldots, n-1\}$, if $s[i] = s[i+1]$, let $\mathsf{PAL}(i, i+1) = 1$; otherwise, let $\mathsf{PAL}(i, i+1) = 0$.
- General Cases: For each (i, j) such that $i + 2 \leq j$, $\mathsf{PAL}(i, j) = 1$ if $\mathsf{PAL}(i+1, j-1) = 1$ and $s[i] = s[j]$; otherwise, $\mathsf{PAL}(i, j) = 0$. This is because $s[i : j]$ is a palindrome if and only if $s[i] = s[j]$ and $s[i+1 : j-1]$ is a palindrome.

The pseudocode for computing PAL is given as follows.

```
1:  procedure ComputePAL(s)
2:      for i = 1 to n do
3:          PAL(i, i) = 1
4:      for i = 1 to n − 1 do
5:          if s[i] = s[i + 1] then
6:              PAL(i, i + 1) = 1
7:          else
8:              PAL(i, i + 1) = 0
9:      for l = 2 to n − 1 do
10:         for i = 1 to n − l do
11:             j = i + l
12:             if PAL(i + 1, j − 1) = 1 and s[i] = s[j] then
13:                 PAL(i, j) = 1
```

```
14:             else
15:                 PAL(i, j) = 0
```

Now we use PAL to help us efficiently compute DP by observing that:

$$\mathsf{DP}(i) = \min_{k:\mathsf{PAL}(k,i)=1} (1 + \mathsf{DP}(k-1)). \tag{1}$$

This recurrence determines the minimum number of palindromes into which we can decompose the string $s[1:i]$.

This recurrence relation comes from the following observations:

- We consider all potential starting positions k for a palindrome that concludes at position i.
- The condition $\mathsf{PAL}(k,i) = 1$ confirms that the substring $s[k:i]$ is a palindrome.
- The term $\mathsf{DP}(k-1)$ provides the optimal count of palindromes for the segment ending at $k-1$. Note that we will need to compute all DP terms before i prior to computing $\mathsf{DP}(i)$.
- Adding 1 to $\mathsf{DP}(k-1)$ effectively adds the palindrome $s[k:i]$ to our cumulative palindrome count.
- Among all valid last palindromes from k to i, we choose the one minimizing the total number of palindromes in the decomposition.

The pseudocode for computing DP is given as follows.

```
1: procedure ComputeDP(s)
2:     for i = 0 to n do
3:         DP(i) = i
4:     for i = 1 to n do
5:         for k = 1 to i do
6:             if PAL(k, i) = 1 then
7:                 DP(i) = min(DP(i), 1 + DP(k − 1))
8:     return DP(n)
```

The dimension of PAL is $n \times n$. For each $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, n\}$, $\mathsf{PAL}(i,j)$ is visited exactly once and computed in $O(1)$ time. Therefore, the running time of ComputePAL is $O(n^2)$. The length of DP is n, and for each $i \in \{1, \ldots, n\}$, $\mathsf{DP}(i)$ is visited exactly once and computed in $O(i)$ time. Therefore, the running time of ComputeDP is $O(n^2)$ as well. Hence, the total running time is $O(n^2)$.

Problem 4 (Applied). You are working on a data analysis project for a large theme park. You have location trajectory data for paths that guests take through their park, represented as a sequence of (x, y)-coordinate pairs (for example, $(0.0, 0.0), (1.0, 0.0), (2.0, 1.0), (2.0, 2.0)$, etc.). You would like to characterize common trajectories through the park, but in order to do so you must first be able to measure how similar any given pair of trajectories are to each other.

The distance to be used between any pair of points is just the standard Euclidean distance. The problem is that not all of your trajectories have the same number of points. Further complicating things, not everyone walks through the park at the same speed, but you want to consider two trajectories as equivalent if they take the same exact path at different speeds.

Based on these observations, what you would like to calculate is the distance between two trajectories under an optimal alignment. Suppose we label the trajectories A and B. An alignment is a mapping between the two trajectories. To be a valid alignment, it must satisfy the following properties:

- $A[0]$ must map to $B[0]$ (the beginnings must be aligned)
- The last point of A must map to the last point of B (the ends must be aligned)
- Every point in A must map to one or more points in B, and vice versa (all points must be aligned somewhere)
- The alignment must be monotone in the sense that if $A[i]$ maps to $B[j]$ then for all $k > i$, $A[k]$ must map to $B[j']$ for some $j' \geq j$, and vice versa (the alignment must go forward in time for both trajectories)

Among valid alignments, the optimal alignment has the minimum total distance when the distance between all aligned pairs of points is added up. **You will need to design and implement an algorithm that returns the minimum distance between two trajectories under an optimal alignment.** Your solution will need to have an empirical runtime that is within constant factors of an reference solution with $O(|A||B|)$ runtime, where $|A|$ and $|B|$ are the number of points in trajectories A and B respectively.

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. **NOTE:** Unlike the theory problems, the applied problem grade **is the raw score shown on Gradescope**. See the course assignments webpage for more details.

- **Python.** You should submit a file called `align.py` to the Gradescope item "Assignment 4 - Applied (Python)." The file should define (at least) a top level function `align` that takes in two inputs, each being a list of tuples `(i:int, j:int)` and return a float number. The function header is as follows
  - `def align(seriesA: list[(int, int)], seriesB: list[(int, int)]):`
- **Java.** You should submit a file called `Align.java` to the Gradescope item "Assignment 4 - Applied (Java)." The file should define (at least) a top level function `align` that takes in two 2D double arrays: `seriesA` and `seriesB` and returns a double value. The header for the method is as follows
  - `public double align(double[][] seriesA, double[][] seriesB);`

  seriesA and seriesB are n by 2 2D arrays, where seriesA$[i]$ is the ith point in the trajectory

Solution 4. We don't provide full code implementations for applied problems, but will sketch the idea of the algorithm here.

Let $D(i, j)$ be the total distance under an optimal alignment of the first i points of A and the first j points of B. Consider the following cases:

- If $i = 1$ then we simply have the distance from $A[0]$ to each of the first j points in B. Similarly if $j = 1$.
- Otherwise $i > 1$ and $j > 1$. By the problem statement, the last element $A[i-1]$ must map to the last element $B[j-1]$. So there is a distance of $d(A[i-1], B[j-1])$. In addition, there are three posibilities:
  - Neither $A[i-1]$ nor $B[j-1]$ are aligned with any prior points in an optimal alignment, in which case we would add $D(i-1, j-1)$.
  - $A[i-1]$ could also be aligned with prior points in B, in which case we would add $D(i, j-1)$..
  - $B[j-1]$ could be mapped to by prior points in A, in which case we would add $D(i-1, j)$.

  It cannot be the case that both $A[i-1]$ is mapped to prior points in B and $B[j-1]$ is mapped to by prior points in A, as this would violate the monotonicity requirement. As we want the minimum distance alignment, we take the minimum of the three recursive cases.

The dynamic programming algorithm can be efficiently implemented to compute this recurrence either with memoized recursion or a bottom-up iterative solution of the subproblems.