CS 3000: Algorithms & Data — Spring 2024

Homework 3
Due Monday February 19 at 11:59pm via Gradescope

Name: Alexander Rosenthal
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 homework is due Monday February 19 at 11:59pm via Gradescope. No late assignments will be accepted. Make sure to submit something before the deadline.
- Solutions must be typeset. If you need to draw any diagrams, you may draw them by hand as long as they are embedded in the PDF. We recommend that you use LaTeX, in which case it would be best to use the source file for this assignment to get started.
- We encourage you to work with your classmates on the homework problems, but also urge you to attempt all of the problems by yourself first. 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. *(3 + 6 + 7 + 4 = 20 points) Stacking packages in a warehouse*

You are working in an Amazon warehouse and are designing a program to optimally store shipments, each of which is a rectangular prism (each face is a rectangle). One piece of this program is to stack the maximum number of packages in a single stack. All packages have the same height but may vary in length, breadth, and weight. So, your program takes as input three lists L, B, and W, each of length n, where $L[i]$, $B[i]$, and $W[i]$ denote the length, breadth, and weight, respectively, of the ith package.

Package i can be placed on top of package j if (i) $W[i] < W[j]$, and (ii) package i can be oriented so that it is strictly smaller than package j in each of the two dimensions. For example, even though the package i with $L[i] = 5$ and $B[i] = 1$ is longer than package j with $L[j] = 2$ and $B[j] = 6$, i can be stacked on j by switching its length and breadth.

(a) Describe an $O(1)$ time algorithm to determine whether one package can be stacked on top of another.

Solution: This function returns whether the box of index i can be placed on top of the box of index j. This assumes the lists L, B, and W are global variables and do not need to be passed as arguments.

```
Function IsStackable($j, i$):
    lighter ← $W[i] < W[j]$
    smaller1 ← $(L[i] < L[j])$ and $(B[i] < B[j])$
    smaller2 ← $(B[i] < L[j])$ and $(L[i] < B[j])$
    Return lighter and (smaller1 or smaller2)
```

(b) Suppose the packages are labeled 1 through n in order of increasing weight (assume all weights are distinct). Let $\text{OPT}(i)$ denote the maximum number of packages you can stack using packages 1 through i, with package i at the bottom of the stack. Give a recurrence to compute $\text{OPT}(i)$, and write the base case for this recurrence. Write a few sentences explaining why your recurrence is correct.

Solution: Let p(i) denote all packages with label k such that IsStackable(i, k) is true. Then:

$$\text{OPT}(i) = \max_{k \text{ in } p(i)} \{\text{OPT}(k)\} + 1$$

OPT(1) = 1

We ensure that OPT(i) is the size of the tallest stack by finding the maximum stack size among the packages that can be stacked on top of i and adding 1.

(c) Using your recurrence, design a dynamic programming algorithm with pseudocode to output the optimal set of packages to stack. You may use either a top-down or bottom-up approach. Remember that your algorithm needs to output the optimal set of packages, not only their count. Your algorithm may output the set of packages as a list of indices. The running time of your algorithm must be polynomial in n.

Solution:

Algorithm 1: Stacking Packages in a Warehouse

```
Function TallestStack(L, B, W, n):
    sortByWeight(L, B, W)
    OPT[1] = 1, OPT[0] = 0
    \\ BoxAbove[i] = the box stacked directly on top of box i in the optimal stack, 0
      means no box above
    BoxAbove ← a list of length n with all 0
    tallest = 1 \\ index of tallest stack in OPT
    For i from 2 to n
        OPT[i] = 1
        For k from 1 to i-1
            If IsStackable(i, k) and (OPT[k] > OPT[BoxAbove[i]]) :
                OPT[i] = OPT[k] + 1
                BoxAbove[i] = k
        If OPT[i] > OPT[tallest] :
            tallest = i
    packages = []
    While tallest > 0
        optimal.add(tallest)
        tallest = BoxAbove[tallest]
    Return packages
```

(d) Analyze the running time and space usage of your algorithm.

Solution: The algorithm first sorts the 3 lists by weight, which takes O($n \lg n$) time. The inner for loop makes n-k comparisons of constant time, and the outer for loop iterates n times, which we know sums to n(n+1)/2. Therefore the for loops take O(n^2) time. The while loop at the end iterates O(n) times. The overall run time is dominated by the runtime of the for loops, so overal it's O(n^2).
The OPT and BoxAbove lists have a size of n, which together is 2n, so the overall space complexity is simply O(n).

Problem 2. *(5 + 6 + 9 = 20 points) Nicest photos*

Alice would like to take nice photos of the Boston skyline. She records the height of the buildings in an array $h[1], \dots, h[n]$, where h_1 is the height of the leftmost building and h_n is the height of the rightmost building. Each photo cannot stray beyond the recorded skyline and it captures k consecutive buildings. The score of a photo is the total height of all the building in the photo plus the tallest height. For example, if the heights are $5, 2, 9, 1, 7$ and $k = 3$ then the best photo would capture $9, 1, 7$ with score $(9 + 1 + 7) + 9 = 26$. Alice would like to find the set of non-overlapping photos with the maximum total score. For example, if the heights are $1, 1, 1, 6, 6, 1, 1, 1, 1$ and $k = 3$ then the best set of photos consist of $1, 1, 6$ and $6, 1, 1$ for the total score of 28.

(a) Let $score[i]$ denote the score of the photo capturing buildings from $i - k + 1$ to i. Give an algorithm to compute $score[i]$ for all i that runs in $O(nk)$ time.

Solution: The first k-1 buildings do not have enough buildings to the left of them to take an entire photo, so they have a score of 0.

```
Function Scores(h, k):
    scores[1 to k-1] = 0
    For i from k to n
        heights ← empty list
        For j from i down to i-k+1
            heights.add(h[j])
        scores[i] = max(heights) + sum(heights)
    Return score
```

(b) Let $\text{OPT}[i]$ denote the maximum score for non-overlapping photos contained entirely between buildings from 1 to i, inclusively. Give a recurrence for computing $\text{OPT}[i]$ and the base case(s).

Solution: Find the max score between either including a picture of building i (first part), or not including it (second part).
OPT(i) = max{OPT(i-k) + score[i], OPT(i-1)}
OPT(i ≤ 0) = 0

(c) Use your recurrence to design a dynamic programming algorithm with pseudocode, which takes as input the h array and k, and computes the best total score as well as the photo locations. Give the running time of your algorithm.

Solution: This algorithm returns the max score of non-overlapping photos and the indices of the end of the photos included in the max. In the example provided above, it would return a score of 28 and the indices 4 and 7.

```
Function NicestPhotos(h, k):
    scores ← Scores(h, k)
    OPT[0] = 0
    For i from 1 to n
        If $i - k < 0$ :
            OPT[i] = 0
        Else
            OPT[i] = max(scores[i] + OPT[i-k], OPT[i-1])
    photos ← list to store indices
    j ← n
    While $j > 0$
        If $OPT[j] > OPT[j-1]$ :
            photos.add(j)
            $j \leftarrow j - k$
        Else
            $j \leftarrow j - 1$
    Return OPT[n], photos
```

First the scores[i] list is calculated. In the Scores function, the outer for loop iterates n times and the inner iterates k times and makes O(1) calculations, which gives us O(nk) running time. Back in the NicestPhotos fucntion, the first for loop iterates n times, and each iteration runs in O(1) time giving us O(n). The while loop at the end iterates in O(n) time as well. Overall this gives us O(nk) + O(n) + O(n) which is O(n(k+2)) time.

(d) **(0 bonus points)** Give an algorithm for part (a) with $O(n)$ runtime. Hint: a viable approach is divide and conquer.

Solution:

Problem 3. *(5 + 6 + 5 + 4 = 20 points) Elegant subsequence*

We are given an array $A[1,\dots,n]$ of n distinct integers sorted in increasing order. A subsequence of A is a sequence that can be derived from A by deleting some (or none) of the elements without changing the order of the remaining elements. A sequence is elegant if all the differences of the consecutive elements are all the same. For example, $1, 4, 7, 10$ is elegant because the differences between consecutive elements are all 3. We would like to find the longest elegant subsequence of A. For example, if $A = (1, 3, 4, 7, 8, 10)$ then the longest elegant subsequence is $(1, 4, 7, 10)$. For full credit, your algorithm should run in $O(n^2 \log n)$ time ($O(n^2)$ is possible).

(a) Let $L[i, j]$ be the length of the longest elegant subsequence whose last two indices are i and j where $i < j$. Describe a recurrence to compute $L[i, j]$ using $L[k, i]\ \forall k < i$. Hint: Define $p[i, j]$ to be the index of the element in A such that $A[p[i, j]] = 2A[i] - A[j]$ i.e. $A[p[i, j]] - A[i] = A[i] - A[j]$ (if such an element exists).

Solution:
p[i,j] is the index of an element in A such that A[p[i,j]] = 2A[i] - A[j].

$$L[i,j] = \begin{cases} 1 + L[p[i,j], i] & \text{if p[i,j] exists} \\ 2 & \text{if p[i,j] does not exist} \end{cases}$$

(b) Describe a dynamic programming algorithm with pseudocode to compute all $L[i, j]$.

Solution: This algorithm implements the recurrence above using 2 nested for loops and a binary search which returns the index of the element equal to the given target value.

```
Function Lengths(A):
    For i from 1 to n-1
        For j from i+1 to n
            target ← $2A[i] - A[j]$
            p ← BinarySearchIndex(A, target)
            If p == "not found" :
                L[i, j] = 2
            Else
                L[i, j] = 1 + L[p, i]
    Return L
```

(c) Describe how to find the longest elegant subsequence from the $L[i, j]$ you computed.

Solution: First, find the indices i and j of the maximum value within L. This can be done concurrently with the computation of all L[i,j] by keeping track of the largest L[i,j] and the indices of i and j, updating as you go. Then, you can use a for loop that counts backwards from the ith element and counts the number of elements that have a constant difference from the one previously counted. This is described in pseudocode below.

```
Function LongestElegantSubsequence:
    diff ← A[maxj] - A[maxi]
    last ← A[maxi]
    \\ .push adds an element to the beginning of the list
    sequence ← empty list
    sequence.push(A[maxj]); sequence.push(A[maxi])
    For from maxi-1 down to 1
        If last - A[k] == diff :
            sequence.push(A[k])
            last ← A[k]
    Return sequence
```

(d) Analyze the running time and space usage of your algorithm.

Solution: To compute all L[i,j] we use a series of nested for loops. The outer one iterates n times, and the inner iterates n - i times, which sums to O(n^2). For each iteration of the inner loop, a call to BinarySearchIndex is made, which runs in O($\lg n$) time, and O(1) other comparisons and calculations are made. Therefore, computing all L[i,j] takes O($n^2 \lg n$) time. In order to get the subsequence, we simply use the L[i,j] we already calculated and a for loop that iterates n times and makes O(1) calculations. This O(n) term is dominated by the larger one, so we have a total running time of O($n^2 \lg n$).
L[i,j] is a 2 dimension array of size n x n, so our space usage is O(n^2)

Problem 4. *(6 + 7 + 3 + 4 = 20 points) Resource reservation in video transmission*

Consider the following resource reservation problem arising in video transmission. Suppose we have a video that is a sequence of n frames. We are given an array $s[0 \dots n-1]$: frame i requires the reservation of at least $s[i]$ units of bandwidth along the transmission link. Since reserving resources separately for each frame may incur a significant overhead, we would like to partition the video into at most k $\underline{\text{segments}}$ (where k is usually much smaller than n), and then reserve bandwidth for each segment. Note that each segment is simply an interval of contiguous frames and that the segments may be of different lengths.

The amount of bandwidth that we need to reserve for a segment is the maximum, over all frames in the segment, of the bandwidth required for the frame. Formally put, for a given interval of frames I, the bandwidth $B(I)$ required for the segment equals $\max_{i \in I} s[i]$.

A segmentation of the video into k segments is a sequence of $k-1$ integers $0 < p[0] < p[1] < \dots < p[k-2] < k$ indicating that the segments are given by the k intervals $I_0 = [0 \dots p[0]-1], I_1 = [p[0] \dots p[1]-1], \dots, I_{k-1} = [p[k-1] \dots k-1]$. The $\underline{\text{cost}}$ of the segmentation is the total bandwidth required, which is the following:

$$\sum_{0 \le i < k} |I_i| \cdot B(I_i)$$

where $|I_i|$ is the number of frames in segment S_i.
Example: Suppose $n = 10$, the array $s = [3,5,2,9,5,6,7,1,4,11]$, and $k = 3$. If we consider the segmentation $[0-2],[3-6],[7-9]$, the total bandwidth requirement would be

$$5 \times 3 + 9 \times 4 + 11 \times 3 = 84.$$

On the other hand, if we choose the segmentation $[0-3],[4-8],[9-9]$, the total bandwidth requirement would be

$$9 \times 4 + 7 \times 5 + 11 = 82.$$

Your goal is to design a dynamic programming algorithm that takes input the n values $s[0], s[1], \dots, s[n-1]$, and the positive integer $k \le n$ and computes a segmentation which has minimum cost.

(a) Let $\text{OPT}(i,m)$ denote the minimum cost for segmenting frames 0 through i using m segments. Give a recurrence to compute $\text{OPT}(i,m)$, and write the base case(s) for this recurrence. Write a few sentences explaining why your recurrence is correct.

Solution: Let cost(a,b) denote the cost of the segment from frame a to frame b. i = -1 indicates no frames.

$$\text{OPT}(i, m) = \min_{-1 \le j < i} \{\text{OPT}(j, m-1) + \text{cost}(j+1, i)\}$$

OPT(-1, m) = 0 and OPT(i, 1) = cost(0, i)

This is correct because we find the way to add frame i to a segment that results in the minimum overall cost. j = -1 means one segment containing all frames, j = 0 means frame i is in a segment with all other frames except for the first, this continues until j = i-1 which means frame i is in a segment by itself.

(b) Using your recurrence, design a dynamic programming algorithm with pseudocode, which takes as input the *s* array and *k*, and outputs the optimal cost for segmenting the video into *k* segments. You may use either a top-down or bottom-up approach. The output of your algorithm is a single number. The running time of your algorithm must be polynomial in *n*.

Solution:

```
Algorithm 2: Least Video Transmission Bandwidth

Function Cost(s, a, b):
    Return length(s[a,...,b]) · max(s[a,...,b])

Function MinCost(s, k):
    For i from 0 to n-1
        For m from 1 to k
            min ← ∞
            For j from -1 to i-1
                If (j == -1) or (m == 1) :
                    cost ← Cost(s, 0, i)
                Else
                    cost ← OPT[j][m-1] + Cost(s, j+1, i)
                If cost < min :
                    min ← cost
            OPT[i][m] ← min
    Return OPT[n-1][k]
```

(c) Analyze the running time and space usage of your algorithm.

Solution: The first two for loops run in n and k time respectively, and the innermost for loop runs in n - i time. We know the inner and outmost loops sum to O(n^2), and the middle loop multiplies each iteration of the outer loop by k. This gives us a total of O(kn^2) running time.
As for space usage, OPT is given a value for every i and m, which is a total of an n*k array. Therefore the space usage is O(nk).

(d) Enhance your algorithm of part (b) to return the optimal segmentation, not just the optimal cost. The output of your algorithm can be the array *p* giving the segmentation.

Solution:

Algorithm 3: Least Video Transmission Bandwidth With Segmentation

```
Function Cost(s, a, b):
    Return length(s[a,...,b]) · max(s[a,...,b])

Function MinSegment(s, k):
    For i from 0 to n-1
        For m from 1 to k
            min ← $-$inf
            For j from -1 to i-1
                If (j == -1) or (m == 1) :
                    cost ← Cost(s, 0, i)
                Else
                    cost ← OPT[j][m-1] + Cost(s, j+1, i)
                If cost < min :
                    min ← cost
                    FrameBefore[i][m] = j \\ Holds the last frame in the segment that
                      comes before the newest segment containing i
            OPT[i][m] ← min
    p ← empty list
    frame ← n - 1
    For seg from k down to 2
        p.push(FrameBefore[frame][seg] + 1) \\ add to beginning of list
        frame ← FrameBefore[frame][seg]
    Return p
```