

# CS375 Weekly Assignment 8

Due April 22nd 10PM ET

A general note for CS375: When writing up your homework, please write neatly and explain your answers clearly, giving all details needed to make your answers easy to understand. Graders may not award full credit to incomplete or illegible solutions.

You are encouraged to discuss the problems with your classmates, **but your submission should be your own**. When writing your solutions up, you should not be looking at any other student's submission.

Clear communication **is the point**, on every assignment. In general in CS375, unless explicitly specified otherwise, answers should be accompanied by explanations. Answers without explanations may not receive full credit. Please feel free to ask me any questions about explanations that might come up!

Necessary acknowledgements: many of these questions are reproductions or adaptations of questions of Eric Aaron's devising. Others are from our course texts as listed on the syllabus.

## 1. Pizza Delivery Problems:

- (a) Let's imagine we're the pizza company Sonimod and our business offers delivery. Luckily all our orders happen to be placed on the same road we're on: to model this, we'll imagine our input as points on the real line  $x_m, x_{m-1}, \dots, x_1, y_1, y_2, \dots, y_{n-m}$  where:
- our points are given in sorted order  
(so  $x_m < x_{m-1} < \dots < x_1 < y_1 < y_2 < \dots < y_{n-m}$ )
  - all  $x_i < 0$  and all  $y_j > 0$
  - our business is at the origin (0)

Every order also has a deadline that we have to get the pizza there by in order to get paid. Assume that we only have one driver who always travel at unit speed (so it takes  $t$  time to travel  $t$  distance) and all deliveries are instantaneous (they just throw the pizza out the window). Our goal is to make every delivery before the deadline. Suppose we made deliveries by order of deadline (ie, we sorted the orders by the their deadline and made the deliveries in that order). Show that this algorithm is incorrect.

**Solution:** Suppose our input were

| Point | Location | Deadline |
|-------|----------|----------|
| $x_1$ | -1       | 6        |
| $y_1$ | 3        | 4        |
| $y_2$ | 4        | 5        |

Our algorithm would try to deliver in the order  $(y_1, y_2, x_1)$ , which would meet both the  $y$ 's deadlines but would arrive at  $x_1$  at time 9, past its deadline. However, had it taken the path  $(x_1, y_1, y_2)$ , it would have arrived at  $x_1$  at time 1 ( $< 6$ ), at  $y_1$  at time 4, and at  $y_2$  at time 5.

- (b) Next, design an algorithm with runtime  $\Theta(n \times n!)$  (its correctness should be obvious) that determines whether you can make all the deliveries before their deadlines.

**Solution:** Just generate all possible delivery schedules (which are just permutations of the deliveries) and try each one: if we find one that makes it to every deadline, great! If not, then we're certain that no possible ordering exists. There are  $n!$  possible schedules (each delivery schedule is just a permutation over the input) and checking that each deadline is met per schedule just involves iterating over the schedule (of length  $n$ ) and confirming that the total distance traveled stays below the deadlines for each delivery. Since our first step would be to generate all possible permutations, this algorithm takes  $\Theta(n \times n!)$  total time.

- (c) Next, design a dynamic programming algorithm to determine whether we can make all the deliveries before their deadlines.

**Solution:** The difficulty of this problem is that determining the right recursive formula to compute is *quite* difficult. One would presumably first try something like

$$\text{OPT}(i) = \begin{cases} \text{TRUE} & \text{if we can make the first } i \text{ deliveries before their deadlines} \\ \text{FALSE} & \text{otherwise.} \end{cases}$$

But the problem here is that recursing is impossible - we know from above that going to each delivery in order won't work, which means that it might be correct to make the last ( $i$ th) delivery before the first  $i - 1$ , and suddenly the sentence would require starting from the  $i$ th delivery location instead of the origin... It becomes a mess. Fundamentally, what I try to do when I think about problems like this is to ask "what completely describes all the information at a given state of a solution"? Some of the information might not be necessary and the algorithm can then be improved by trimming stuff, but getting "all" the information seems like a good starting place. So what is all the information of a particular state in a solution? I would say I need to know:

- How much time has passed up to that state
- What locations have been delivered to
- Where the driver is (what delivery they just made)

With that in mind, there's really two kinds of solutions: solutions that finish on the left ( $x$ ) side, and solutions that finish on the right ( $y$ ) side. The easiest way to do this is calculate them separately! To do so, we're going to make one super important observation: if you're going to drive past a delivery location, there is no reason to not chuck the delivery out the window as you go, as the problem says that it takes 0 time to make a delivery.<sup>1</sup>

$\text{OPT}_L(i, j) =$  The minimal time needed to make  $i$  deliveries where the last delivery is to  $x_j$  and all deliveries are made prior to their deadline, otherwise  $\infty$

$\text{OPT}_R(i, j) =$  The minimal time needed to make  $i$  deliveries where the last delivery is to  $y_j$  and all deliveries are made prior to their deadline, otherwise  $\infty$

To determine the recursive formulae for each of these, I'm going to focus on the calculation of  $\text{OPT}_L$ , since  $\text{OPT}_R$  is similarly determined. As a notational convenience, I'm going to use the function  $d(\ )$  to denote the deadline of a given delivery location (so  $d(x_1)$  is the delivery deadline of the delivery to  $x_1$ ). First, notice that it doesn't make sense to talk about cases where  $i < j$ : because I have made the rule that I always make deliveries as I drive past a delivery location, I can't get to  $x_j$  without making at least  $j$  deliveries. Note that if I have made  $i > 0$  total deliveries and the last one is to  $x_j > 0$ , then:

- I must have delivered to  $x_1, \dots, x_{j-1}$  too, since I drove past them
- I must not have delivered to any  $x_k$  for  $k > j$ , because otherwise I would have driven past  $x_j$  earlier and therefore would have already delivered to it
- The remaining  $i - j$  deliveries have all been to points on the right hand side: more particularly, I've made deliveries to  $y_1, \dots, y_{i-j}$
- If  $i = j$ , then I just drove from the origin to  $x_1$  and then  $x_2$  and so on until I got to  $x_j$ , so the last delivery was to  $x_{j-1}$ . For  $j < i$ , the last delivery (the  $i - 1$ th delivery) could have been to one of two places:  $x_{j-1}$  or  $y_{i-j}$ . Because of the rule that we do not drive past delivery locations without making deliveries, any other locations would be invalid.

Because we are recording  $\infty$  if it's impossible to make all the deliveries before their deadline, I'm going to make a little helper function for our definitions:

$$f(x, d) = \begin{cases} x & \text{if } x \leq d \\ \infty & \text{otherwise} \end{cases}$$

---

<sup>1</sup>Actually, it changes the problem in a pretty non trivial way if making a delivery takes some positive amount of time, even if that time is small

The above observations and this helper function gives the following recursive formulae:

$$\text{OPT}_L(i, j) = \begin{cases} 0 & \text{if } i = j = 0 \\ f(\text{OPT}_L(i-1, j-1) + x_{j-1} - x_j, d(x_j)) & \text{else if } i = j > 0 \\ f(\min[\text{OPT}_L(i-1, j-1) + x_{j-1} - x_j, \\ \text{OPT}_R(i-1, i-j) + y_{i-j} - x_j], d(x_j)) & \text{otherwise.} \end{cases}$$

$$\text{OPT}_R(i, j) = \begin{cases} 0 & \text{if } i = j = 0 \\ f(\text{OPT}_R(i-1, j-1) + x_{j-1} - x_j, d(x_j)) & \text{else if } i = j > 0 \\ f(\min[\text{OPT}_R(i-1, j-1) + y_j - y_{j-1}, \\ \text{OPT}_L(i-1, i-j) + y_j - x_{i-j}], d(y_j)) & \text{otherwise.} \end{cases}$$

Let's compute this dynamically.

```
def f(x, d):
    return x if x ≤ d else ∞

L = new array of size (n + 1) × (m + 1)
R = new array of size (n + 1) × (n - m + 1)

L[0, 0] = R[0, 0] = 0
for i in range(n + 1):
    for j in range(1, i):
        if i ≤ m:
            L[i, j] = L[i - 1, j - 1] + xj-1 - xj
            // try option 1
            L[i, j] = min(L[i, j], R[i - 1, i - j] + yi-j - xj)
            // take min with option 2
            L[i, j] = f(L[i, j], d(xj))
            // apply f incase we're past the deadline

        if i ≤ n - m:
            R[i, j] = R[i - 1, j - 1] + yj - yj-1
            R[i, j] = min(R[i, j], L[i - 1, i - j] + yj - xi-j)
            R[i, j] = f(R[i, j], d(yj))

    if i ≤ m:
        L[i, i] = L[i - 1, i - 1] + xj-1 - xj
        L[i, i] = f(L[i, i], d(xi))

    if i ≤ n - m:
        R[i, i] = R[i - 1, i - 1] + yj - yj-1
        R[i, i] = f(R[i, i], d(yj))
```

After this concludes, we'll have the  $L[i, j] = \text{OPT}_L(i, j)$  and  $R[i, j] = \text{OPT}_R(i, j)$ . Since we're building two arrays that are of sizes  $\Theta(nm)$  and  $\Theta(n(n-m))$ , we get a combined runtime of  $\Theta(n^2)$

Now, if both  $L[n, m]$  and  $R[n, n-m]$  are infinite, *then we know that we can't make all the deliveries*: if either are finite, then we can make a solution that ends on the corresponding side! So we just need to backtrack to determine the actual delivery schedule that works, if one works:

```

if L[n, m] == ∞ and R[n, n - m] == ∞:
    output that no solution is possible

elif L[n, m] == ∞:
    curArray = R
else:
    curArray = L

i = n
j = m if curArray == L else n - m
output = []

while i > 0:
    if curArray == L:
        add  $x_j$  to output
        if L[i, j] == L[i - 1, j - 1] +  $x_{j-1} - x_j$ :
            j -= 1
        else:
            curArray = R
            j = i - j
        i -= 1
    else:
        add  $y_j$  to output
        if R[i, j] == R[i - 1, j - 1] +  $y_j - y_{j-1}$ :
            j -= 1
        else:
            curArray = L
            j = i - j
        i -= 1

return the reverse of output

```

Backtracking here takes  $\Theta(n)$  time (assuming there's a way to make all the deadlines, otherwise it's  $\Theta(1)$  as the code immediately outputs in the first two lines), as the while loop decreases  $i$  by 1 each iteration and reversing a list of  $n$  items takes  $\Theta(n)$  time. With the original building of the arrays, this gives us a total runtime of  $\Theta(n^2) + \Theta(n) = \Theta(n^2)$ .

- (d) Next, suppose Sonimod hired a second driver and expanded their delivery radius beyond the road they are on: now we have a sequence of deliveries to make, where delivery  $i$  has a location  $p_i$  and a deadline  $d_i$ . Our manager, however, has gotten complaints from some of our perceptive customers who are upset that we aren't delivering by order of deadline (which is the same as delivering by order of deadline, since all deadlines are just a fixed amount of time after the original order placement) **initial order placement**. So we have some new rules:

- Every point must be assigned to a driver
- Each driver must visit the deliveries they've been assigned by order of ~~deadline (earliest first, then next earliest, etc.)~~ **submission (you can assume the original points were given in order of submission)**.
- ~~Our goal is still to make every delivery before its deadline~~ **Since our order of deliveries is now somewhat fixed, the manager just wants to minimize gas usage, so we need to minimize the total units of distance traveled.**

Design a brute-force algorithm to solve this problem. What is its runtime? **You should freely assume that you can calculate the time it takes to drive distance between any two points  $d(p_i, p_j)$  in constant time.**

**Solution:** Notice that every possible solution is just assigning some subset of the deliveries to

the first driver (who must make them in order) while the second driver handles the rest (also in order). So a brute force algorithm for this would be to just generate all subsets of deliveries, then for each such subset give them to the first driver and the rest to the second and check which such partition gives the minimum combined distance traveled. Of course, this means that for each subset (of which there are  $2^n$ ) we have to calculate the distance traveled by the induced partition, which just takes  $\Theta(n)$  time. This gives a total runtime of  $\Theta(n2^n)$ .

- (e) Design a dynamic programming algorithm to solve this problem. In your answer, include an argument of its correctness and an analysis of its runtime.

**Solution:** Again, let's try to figure out all the information that describes a state of a solution:

- Probably I want the number of deliveries made. Since they're being made in order this time, we can probably just parameterize by the first  $i$  deliveries of the problem.
- **Complete** info would be to record exactly who made each delivery - but that info isn't so easily encoded this time. In the previous problem, knowing the number of deliveries and the last delivery was sufficient to knowing which deliveries had been made. So we'll be a little more conservative here: we'll just ask what the last delivery is for each driver.
- But! If I told you we're making the first  $i$  deliveries, then I know that one of the drivers is at  $p_i$ : the only remaining info then to handle the above bullet point is just where the other driver is.

This suggests the following formula:

$$\text{OPT}(i, j) = \begin{array}{l} \text{The minimum distance needed to be traveled to make the first } i \text{ deliveries} \\ \text{where the driver who doesn't make the delivery to } p_i \text{ instead finishes at } p_j \end{array}$$

Like before, it doesn't make sense to consider cases where  $i \leq j$ . Note that if  $i > j + 1$ , then the driver who didn't travel to  $p_i$  stopped at  $p_j$  and so the driver who made it to  $p_i$  also covered  $p_{j+1}, p_{j+2}, \dots, p_{i-1}$ , and  $p_i$ . This tells us that for  $j < i - 1$  we get  $\text{OPT}(i, j) = \text{OPT}(i - 1, j) + d(p_{i-1}, p_i)$ . The case where  $j = i - 1$  is more interesting: we don't know exactly where the driver who went to  $p_i$  came from, except that they didn't go to  $p_{j-1}$  (since the other driver handled it). But wherever they came from, let's call it  $p_k$ , the solution that brought them to  $p_k$  must have also been optimal, which tells us that  $\text{OPT}(i, i - 1) = \min_{0 \leq k < i-1} (\text{OPT}(i - 1, k) + d(p_k, p_i))$  (note that I'm letting  $p_0$  denote the origin, Sonimod's location). Putting it together:

$$\text{OPT}(i, j) = \begin{cases} 0 & \text{if } i = j = 0 \\ \text{OPT}(i - 1, j) + d(p_{i-1}, p_i) & \text{else if } j < i - 1 \\ \min_{0 \leq k < i-1} (\text{OPT}(i - 1, k) + d(p_k, p_i)) & \text{otherwise.} \end{cases}$$

To build it dynamically:

A = new array of size  $(n + 1) \times n$

A[0, 0] = 0

for i in range(1, n + 1):

    for j in range(i - 1):

        A[i, j] = A[i - 1, j] + d(p<sub>i-1</sub>, p<sub>i</sub>)

    A[i, i - 1] = min([A[i - 1, k] + d(p<sub>k</sub>, p<sub>i</sub>) for k in range(i - 1)])

    // did a little python list comprehension there...

    // I think its clear enough for pseudocode...

    // if anyone actually reads this and doesn't know what I'm doing,

    // please come ask!

At the end of this, we'll get that A[i, j] = OPT(i, j) for all i and j. The runtime of constructing this is  $\Theta(n^2)$ : each iteration i clearly takes linear time (linear for the for-loop on j plus another linear to take a minimization over a list of linear entries).

Similar to a previous problem last week, however, our answer is not necessarily at the "bottom right corner" of this array - A[n, n - 1] corresponds to the best way where the driver who doesn't handle  $p_n$  handles  $p_{n-1}$ , *but that's not necessary to be an optimal solution*. We technically need to first determine the best last delivery for that driver, then backtrack from there:

```

min_j = 0
for j in range(n):
    if A[i, j] < A[i, min_j]:
        min_j = j

i = n
j = min_j
driver1 = []
driver2 = []
while i > 0:
    add i to driver1
    if j == i - 1:
        k = 0
        while A[i, j] != A[i - 1, k] + d(p_k, p_i):
            k += 1
        // we know by construction that some value of k must work
        j = k

    // swap driver1 and driver2
    temp = driver1
    driver1 = driver2
    driver2 = temp
    i -= 1

return (reverse of driver1, reverse of driver2)

```

Note that the runtime of this portion is  $O(n^2)$ : although  $i$  decreases by 1 each iteration, the inner while loop could take at most another linear factor. This gives a combined runtime of  $\Theta(n^2) + O(n^2) = \Theta(n^2)$ .

2. (Erickson 3.5) This set of exercises asks you to develop efficient algorithms to find optimal subsequences of various kinds. A subsequence is anything obtained from a sequence by extracting a subset of elements, but keeping them in the same order; the elements of the subsequence need not be contiguous in the original sequence. For example, the strings **C**, **DAMN**, **YAIIOAI**, and **DYNAMICPROGRAMMING** are all subsequences of the string **DYNAMICPROGRAMMING**. **Each of these counts as one problem** [Hint: Exactly one of these problems can be solved in  $O(n)$  time using a greedy algorithm.]

- (a) Let  $A[1 \dots m]$  and  $B[1 \dots n]$  be two arbitrary arrays. A common subsequence of  $A$  and  $B$  is another sequence that is a subsequence of both  $A$  and  $B$ . Describe an efficient algorithm to compute the longest common subsequence of  $A$  and  $B$ .

**Solution:** Let

$\text{OPT}(i, j)$  = The length of the LCS of the first  $i$  characters of  $A$  and the first  $j$  characters of  $B$ .

Note that:

- if  $i$  or  $j$  is 0, then there is no string to take characters from, in which case  $\text{OPT}(i, j) = 0$ .
- if  $A[i] = B[j]$ , then the longest common subsequence over the first  $i$  characters of  $A$  and the first  $j$  of  $B$  ends in  $A[i]$ ; the rest of the string is just the longest common subsequence of the first  $i - 1$  characters of  $A$  and the first  $j - 1$  of  $B$ .
- otherwise (if  $A[i] \neq B[j]$ ), then the LCS of the first  $i$  characters of  $A$  and the first  $j$  of  $B$  can't end in both  $A[i]$  and  $B[j]$ , so at least one of the characters are unnecessary. If  $A[i]$  is unnecessary, then the LCS of the the first  $i$  characters of  $A$  and the first  $j$  of  $B$  is just the LCS of the first  $i - 1$  characters of  $A$  and the first  $j$  of  $B$ ; if  $B[j]$  is unnecessary, then the LCS of the the first  $i$  characters of  $A$  and the first  $j$  of  $B$  is just the LCS of the first  $i$  characters of  $A$  and the first  $j - 1$  of  $B$ .

This gives us the following recursive formula:

$$\text{OPT}(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + \text{OPT}(i - 1, j - 1) & \text{if } A[i] = B[j] \\ \max(\text{OPT}(i - 1, j), \text{OPT}(i, j - 1)) & \text{otherwise.} \end{cases}$$

To compute these values dynamically, we'll build an array  $C$  such that  $C[i, j] = \text{OPT}(i, j)$ :

```
C = new array of size (m + 1) × (n + 1)
for i in range(1, m + 1):
    C[i, 0] = 0

for j in range(1, n + 1):
    C[0, j] = 0

for i in range(1, m + 1):
    for j in range(1, n + 1):
        if A[i] == B[j]:
            C[i, j] = 1 + C[i - 1, j - 1]
        else:
            C[i, j] = max(C[i - 1, j], C[i, j - 1])
```

Note that the time it takes to compute  $C$  is  $\Theta(n, m)$ , since that nested for-loop has  $\Theta(n, m)$  iterations with a constant runtime per iteration.

After computing  $C$ , the length of the LCS over  $A$  and  $B$  is stored in  $C[m, n]$ : to work out the string itself, we just need to backtrack:

```
i = m
j = n
output = []
while i > 0 and j > 0:
    if A[i] == B[j]:
        add A[i] to output
        i -= 1
        j -= 1
    else:
        if C[i, j] == C[i - 1, j]:
            i -= 1
        else:
            j -= 1
return the reverse of output
```

The runtime of computing `output` is  $O(n + m)$  since at least one of  $i$  or  $j$  decreases by 1 per iteration of the while loop. Since this is dominated by construction of  $C$ , the total runtime of our algorithm is still  $\Theta(nm)$ .

- (b) Let  $A[1 \dots m]$  and  $B[1 \dots n]$  be two arbitrary arrays. A common supersequence of  $A$  and  $B$  is another sequence that contains both  $A$  and  $B$  as subsequences. Describe an efficient algorithm to compute the shortest common supersequence of  $A$  and  $B$ .

**Solution:** Let

$\text{OPT}(i, j)$  = The length of the SCS of the first  $i$  characters of  $A$  and the first  $j$  characters of  $B$ .

Note that:

- if  $i$  is 0, then the shortest common supersequence is just the first  $j$  characters of  $B$ .
- if  $j$  is 0, then the SCS is just the first  $i$  characters of  $A$ .

- if  $A[i] = B[j]$ , then the SCS over the first  $i$  characters of  $A$  and the first  $j$  of  $B$  ends in  $A[i]$ ; the rest of the string is just the shortest common supersequence of the first  $i - 1$  characters of  $A$  and the first  $j - 1$  of  $B$ .
- otherwise (if  $A[i] \neq B[j]$ ), then the SCS of the first  $i$  characters of  $A$  and the first  $j$  of  $B$  must end in either  $A[i]$  or  $B[j]$ , since we need to cover both. If  $A[i]$  is the last, then the LCS of the first  $i$  characters of  $A$  and the first  $j$  of  $B$  is just the SCS of the first  $i - 1$  characters of  $A$  and the first  $j$  of  $B$  with  $A[i]$  tacked on at the end; if it ended in  $B[j]$ , then the SCS of the first  $i$  characters of  $A$  and the first  $j$  of  $B$  is just the SCS of the first  $i$  characters of  $A$  and the first  $j - 1$  of  $B$  tacked on with  $B[j]$  at the end.

This gives us the following recursive formula:

$$\text{OPT}(i, j) = \begin{cases} j & \text{if } i = 0 \\ i & \text{if } j = 0 \\ 1 + \text{OPT}(i - 1, j - 1) & \text{if } A[i] = B[j] \\ 1 + \min(\text{OPT}(i - 1, j), \text{OPT}(i, j - 1)) & \text{otherwise.} \end{cases}$$

To compute these values dynamically, we'll build an array  $C$  such that  $C[i, j] = \text{OPT}(i, j)$ :

```
C = new array of size (m + 1) × (n + 1)
for i in range(1, m + 1):
    C[i, 0] = i

for j in range(1, n + 1):
    C[0, j] = j

for i in range(1, m + 1):
    for j in range(1, n + 1):
        if A[i] == B[j]:
            C[i, j] = 1 + C[i - 1, j - 1]
        else:
            C[i, j] = 1 + min(C[i - 1, j], C[i, j - 1])
```

Note that the time it takes to compute  $C$  is  $\Theta(n, m)$ , since that nested for-loop has  $\Theta(n, m)$  iterations with a constant runtime per iteration.

After computing  $C$ , the length of the SCS over  $A$  and  $B$  is stored in  $C[m, n]$ : to work out the string itself, we just need to backtrack:

```
i = m
j = n
output = []
while i > 0 and j > 0:
    if A[i] == B[j]:
        add A[i] to output
        i -= 1
        j -= 1
    else:
        if C[i, j] == C[i - 1, j]:
            add A[i] to the output
            i -= 1
        else:
            add B[j] to the output
            j -= 1
return the reverse of output
```

The runtime of computing `output` is  $O(n + m)$  since at least one of  $i$  or  $j$  decreases by 1 per iteration of the while loop. Since this is dominated by construction of  $\mathbf{C}$ , the total runtime of our algorithm is still  $\Theta(nm)$ .

- (c) Call a sequence  $\mathbf{X}[1 \dots n]$  of numbers bitonic if there is an index  $i$  with  $1 < i < n$ , such that the prefix  $\mathbf{X}[1 \dots i]$  is increasing and the suffix  $\mathbf{X}[i \dots n]$  is decreasing. Describe an efficient algorithm to compute the longest bitonic subsequence of an arbitrary array  $\mathbf{A}$  of integers.

**Solution:** We'll compute the following two formulae:

$\text{Is}(i, j)$  = The smallest last number of all length  $j$   
increasing sequences over the first  $i$  values of  $\mathbf{X}$

$\text{Ds}(i, j)$  = The smallest first number of all length  $j$   
increasing sequences over  $\mathbf{X}[i:n + 1]$

The first of these we developed in class, so I'll just point out that the formula we came up with is:

$$\text{Is}(i, j) = \begin{cases} -\infty & \text{if } j = 0 \\ \infty & \text{else if } i < j \\ \text{Is}(i - 1, j) & \text{else if } \mathbf{X}[i] \leq \text{Is}(i - 1, j - 1) \\ \min(\mathbf{X}[i], \text{Is}(i - 1, j)) & \text{otherwise.} \end{cases}$$

To develop a recursive formula for  $\text{Ds}$ , note that:

- if  $j = 0$ , then we're talking about sequences of length 0, so there isn't really any 'smallest' last number applicable. But we want to be able to tack on any number to a length 0 sequence, so  $-\infty$  makes it so that we can tack anything on.
- for a given value of  $i$ , there are  $n - i + 1$  total values in  $\mathbf{X}[i : n + 1]$ , which limits us to sequences of length  $n - i + 1$ . So if  $j > n - i + 1$  then we can't even make a sequence of length  $j$ . In this case, we'll put a value of  $\infty$  to mark that it's impossible (and that we can't tack on values to make it work).
- if  $\text{Ds}(i, j) = \mathbf{X}[i]$ , then we know that the descending sequence of length  $j$  that starts on the smallest number is  $\mathbf{X}[i]$ , meaning the rest of the sequence is a descending sequence of length  $j - 1$  descending sequence over the numbers  $\mathbf{X}[i + 1 : n + 1]$ . Moreover,  $\mathbf{X}[i]$  must be larger than the *smallest* starting number of such sequences; had  $\mathbf{X}[i]$  been smaller than the smallest starting number of all length  $j - 1$  descending sequences over the numbers in  $\mathbf{X}[i + 1 : n + 1]$ , then we couldn't have even formed a length  $j$  descending sequence.

The above observations give us the following recursive formula:

$$Ds(i, j) = \begin{cases} -\infty & \text{if } j = 0 \\ \infty & \text{else if } n - i + 1 < j \\ Ds(i + 1, j) & \text{else if } X[i] \leq Ds(i + 1, j - 1) \\ \min(X[i], Ds(i + 1, j)) & \text{otherwise.} \end{cases}$$

Now that we have recursive definitions of the formulae, let's build them dynamically:

Initialize I and D as two new arrays of size  $(n + 1) \times (n + 1)$

```
for i in range(n + 1):
    I[i, 0] = D[i, 0] = -∞

for i in range(n + 1):
    for j in range(1, i + 1):
        I[i, j] = I[i - 1, j]
        if X[i] > I[i - 1, j - 1]:
            I[i, j] = min(I[i, j], X[i])
    for j in range(i + 1, n + 1):
        I[i, j] = ∞

for i in range(n, 0, -1): # i = n, n-1, ..., 1
    for j in range(1, n - i + 2):
        D[i, j] = D[i + 1, j]
        if X[i] > D[i + 1, j - 1]:
            D[i, j] = min(D[i, j], X[i])
    for j in range(n - i + 2, n + 1):
        D[i, j] = ∞
```

Note that the above code takes  $\Theta(n^2)$  time to compute: each array is of size  $\Theta(n^2)$ , and each entry takes  $\Theta(1)$  time to compute.

After the above code has been completed, we'll have that  $D[i, j] = DS(i, j)$  and  $I[i, n] = IS(i, j)$ . So you might be asking yourself: what was the point? It's not immediately obvious how this gives us the answer. But let's think about what the answer actually looks like: let  $X[i]$  be the 'peak' of the sequence; where it transitions from increasing to decreasing. Note that all elements of the sequence up to  $X[i]$  must be *the longest sequence of increasing numbers over the first  $i - 1$  numbers in  $X$  whose last number is smaller than  $X[i]$* . Simultaneously, all the elements over the remainder of the sequence form *the longest decreasing sequence of numbers from  $X[i + 1:]$  whose first number is smaller than  $X[i]$* . So let's use that to determine the index of the 'peak':

```
best_i = None
best_length = 0
length_up = 0
length_down = 0
for i in range(1, n + 1):
    j = i - 1
    while j > 0 and I[i - 1, j] > X[i]:
        j -= 1

    increasing_length = j

    j = n - i
    while j > 0 and D[i + 1, j] > X[i]:
        j -= 1

    decreasing_length = j

    total_length = increasing_length + decreasing_length + 1

    if total_length > best_length:
        best_i = i
        best_length = total_length
        length_up = increasing_length
        length_down = decreasing_length
```

Note that the above also takes  $O(n^2)$  time to determine: we have an outer loop of length  $n$ , and each iteration on  $i$  takes  $O(n)$  time.

By the time the above code finishes, we'll have determined the index `best_i` that gives us the peak of the longest bitonic sequence. Additionally, we've determined `length_up`, the length of the increasing sequence of numbers less than `X[i]`, as well as `length_down`, the length of the decreasing sequence of numbers less than `X[i]`. All that remains is to take this information and backtrack to determine the actual sequence itself!

```

output = []
add X[best_i] to output
i = best_i - 1
j = length_up
while j > 0:
    while X[i] != I[i, j]:
        i -= 1
    add X[i] to output
    i -= 1
    j -= 1

output = reverse(output)

i = best_i + 1
j = length_down
while j > 0:
    while X[i] != D[i, j]:
        i += 1
    add X[i] to output
    i += 1
    j -= 1

return output

```

This last step to calculate `output` takes  $O(n)$  time; this is a little subtle to see, but let's focus on the part that calculates the increasing part of the sequence. Clearly the outer while loop takes at most  $n$  iterations, since `j` starts at a value at most  $n$  and decreases by 1 each iteration. But across the whole running of the outer while loop, the number of total iterations of the inner while loop is *also*  $O(n)$ , since `i` also starts at a value of at most  $n$ , can't decrease past 0, and never increases across all the iterations of the outer while loop! The same holds for the second loop determining the decreasing part of the sequence. As a result, the total runtime of our algorithm takes  $\Theta(n^2) + O(n^2) + O(n)$ , giving us a total runtime of  $\Theta(n^2)$ .

- (d) Call a sequence `X[1 .. n]` of numbers oscillating if `X[i] < X[i + 1]` for all even `i`, and `X[i] > X[i + 1]` for all odd `i`. Describe an efficient algorithm to compute the longest oscillating subsequence of an arbitrary array `A` of integers.

**Solution:** This is the one that can actually be done by a greedy algorithm! The more 'brute-force'<sup>2</sup> dynamic programming algorithm for it looks a lot like the previous problem, so I won't bother doing that again.

The greedy algorithm says to maintain an two indices `i` and `j` where:

- `i` will denote the index we are inserting to in our output
- `j` will denote where we are as we are iterating through our input

and we will operate according to the following rule:

- if `i` is even, keep increasing `j` until `A[j] < A[j + 1]`
- if `i` is odd, keep increasing `j` until `A[j] > A[j + 1]`

---

<sup>2</sup>I'm really hesitant to write this here; the good news is that I doubt anyone will make it this far into the solution anyway, but I could see someone reading that sentence and thinking "is Bender trying to claim that dynamic programming is brute force? That's wild!". And that would be very unfortunate. But, if you're reading this, then it might be good for you to know that most people consider dynamic programming a 'pruned' brute-force algorithm.

This induces the following pseudocode:

```

output = []
i = 1 # note that this is 1-indexed
j = 1
while j < len(A):
    if i % 2 == 0:
        while j < len(A) and A[j] > A[j + 1]:
            j += 1
        add A[j] to output
        i += 1
        j += 1
    else:
        while j < len(A) and A[j] < A[j + 1]:
            j += 1
        add A[j] to output
        i += 1
        j += 1

```

To show the correctness of this, we'll use our good old exchange argument! Specifically, we'll consider some other valid oscillating sequence  $S$  and show that ours is at least as long.

To do so, look for the first point of disagreement: let that disagreement be on index  $i$  of our output.

- First, if  $i$  is even: note that in this case the sequence  $S[i - 1], S[i], S[i + 1]$  must decrease then increase. Let  $j$  be the index of  $A$  that we put at index  $i$  of our output and let  $k$  be the index of  $A$  that  $S$  has at index  $i$ .
  - if  $k < j$ , then we can make a new sequence  $T$  of the same length as  $S$  by simply replacing  $S[i]$  with  $\text{output}[i]$ . Note that we kept increasing  $j$  as long as  $A$  is decreasing: this means that  $A[j] < A[k]$ . It also means that the next index of  $A$  that  $S$  pulls from for  $S[i + 1]$  **must** be after  $j$ , because  $S[i] < S[i + 1]$ . But, altogether, this means that  $S[i - 1] > S[i] = A[k] > A[j] = \text{output}[i] = A[j] < A[k] = S[i] < S[i + 1]$ <sup>3</sup>, so I can swap  $\text{output}[i]$  in replacement of  $S[i]$  and still have a valid oscillating sequence of the same length!
  - On the other hand, if  $k > j$ ...
- On the other hand, if  $i$  is odd...

Finish the above proof for a hefty amount of extra credit :)

For full credit, it should follow basically the same logic as I started out in the first case. Just email it to me directly (I'm not putting it on Moodle to see how many people actually read these)

---

<sup>3</sup>obviously there's some redundancy in here, but the point is that I can swap  $\text{output}[i]$  in for  $S[i]$