Comp 160: Algorithms **Recitation 8**

1. You are given two strings A and B with m and n characters, respectively, defined on a fixed alphabet, and three operations: insert (insert a character), delete (delete a character) and replace (replace a character with another character). The edit distance between A and B, $d(A, B)$, is defined as the **minimum** number of edits (operations) required to convert A into B.

   For example, if $A =$"algorithms", and $B =$"al gore rhythms", the edit distance is 6:

   (a) insert ("algorithms"→"al gorithms")
   (b) replace ("al gorithms"→"al gorethms")
   (c) insert ("al gorethms"→"al gore thms")
   (d) insert ("al gore thms"→"al gore rthms")
   (e) insert ("al gore rthms"→"al gore rhthms"),
   (f) insert ("al gore rhthms"→"al gore rhythms").

   This is a fundamental problem in computational biology and natural language processing.

   Describe and analyze an efficient **dynamic programming** algorithm that computes the edit distance.

   - **Define the subproblem:** Let $D[i, j]$ be the edit distance between prefixes $A[1 \ldots i]$ and $B[1 \ldots j]$.
   - **Base cases:** The base case is when one of the strings is empty. In this case, the cost is the length of the other string since we need to insert/delete all of its characters to make them equal. So $D[i, 0] = i$ and $D[0, j] = j$.
   - **Recursion:**

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

   - **Explanation of recursion:** When the last characters match, there exists an optimal solution that matches them as in LCS. This can be proven by contradiction (practice proving this). Then the last characters do not affect edit distance so we can ignore them and recurse to what's left.
     Otherwise (if the last characters do not match), we either need to insert, delete, or replace, which costs 1 edit and then leaves us with one of the subproblems $D[i, j-1]$, $D[i-1, j]$, or $D[i-1, j-1]$, respectively.
   - **Space and runtime:** We have $\Theta(mn)$ subproblems since i ranges from 0 to m and j ranges from 0 to n. Each subproblem can be computed in $O(1)$ time given that smaller subproblems are already computed. The total time and space needed becomes $\Theta(mn)$.

2. Use dynamic programming to find the length of the LIS (longest increasing subsequence) of any list of numbers. For example, in the list (13, 7, 25, 1, 60, 34, 100, 82, 91) the longest LIS we can find is length 5, e.g., (7, 25, 34, 82, 91). There are several increasing sequences of this length.

   Hint: When you define your problem, you may want to use one slightly different than the one we are ultimately answering.

   The trickiest part of LIS is defining the subproblem. Given an array $A[1, n]$, it is tempting to apply DP to $LIS[i]$ = the length of the longest LIS in $A[1, i]$. But it is actually tricky to relate $LIS[i]$ to $LIS[i-1]$. Consider $A = (1, 2, 6, 7, 3, 4, 5)$. The optimal solution for $A[1, 5]$ does not use $A[5]$...but the optimal solution for $A[1, 7]$ does use $A[5]$.

   Instead we will use a technique that gives us greater information about the subproblem solution.

   - **Define the subproblem:** Let $LIS\_with[i]$ = the length of the longest LIS in $A[1, i]$ *which contains* $A[i]$. Note this might NOT be the same as the actual $LIS[i]$ that we ultimately want to find.
   - **Base cases:** $LIS\_with[0] = 0$ since the empty list only allows the empty list as a subsequence.
   - **Explanation of recursion:** Now it is easier to recurse because we know $A[i]$ must be included. All we want is the longest subsequence of $A[1, i-1]$ that can viably be followed by $A[i]$, which happens if the last element is less than $A[i]$. This last element could be any of $A[1]$ through $A[i-1]$ so we consider them all and take the max.
   - **Recursion:** $LIS\_with[i] = 1 + \max_{j \text{ s.t. } S[j]<S[i]} \{LIS\_with[j]\}$
   - **Space and runtime**. There are n subproblems, so the space is $O(n)$. However, the recursion in this case takes $O(n)$ to compute , so overall runtime is $O(n^2)$.

   Now, we have to answer the original problem! The LIS will certainly end at one of the elements of A, so

$$LIS[i] = \max_{1 \leq j \leq i} \{LIS\_with[j]\}$$

   .

Additional practice questions:

3. You are given an array A of n real numbers (note that values could be positive and/or negative). Given two indices i, j such that $i \leq j$, their *score* is the sum of all values stored between A[i] and A[j] (inclusive). For example, The score of 3 and 5 is A[3] + A[4] + A[5]. Your task is to design an algorithm that, given an array A, returns the pair of indices with highest possible score.

(a) (warm-up) Assume all values of the array are non-negative. What algorithm would you use? What is the runtime? (This is a warm-up question. Solution and explanation should take 3 lines)

(b) (warm-up) Repeat the same question for the case in which values of the array are negative.

(c) (warm-up) Describe a cubic time algorithm to solve this problem (the algorithm should be very simple, not using any techniques from 160)

(d) Ok, now let's use a dynamic programing algorithm to improve the runtime. Make sure to explicitly list the base case(s), the recurrence that you use to compute other cases, justify why the recurrence works, and then discuss and time complexity and space required by your algorithm.

Note: make sure to do a well structured description of your strategy. If this were a graded exercise, for full credit your solution should both (*i*) run in subquadratic time and (*ii*) be well-structured.

This is known as the *maximum subarray problem*, among other names.

(a) If all values are non-negative, we want to sum the whole array. Thus, the solution is to return the indices $(1, n)$. This can be done in constant time (or linear if you actually want to report the sum)

(b) If all values are negative, we want the reverse: to sum as little as possible. The least we can do is sum one entry, so we return the index i whose $A[i]$ is largest (in other words, we return $\arg\max_i\{A[i]\}$ twice). This is done in linear time in a single scan of the array.

(c) Brute force approach: for every pair of indices i, j, let $S[i,j] = \sum_{k=i}^{j} A[k]$. It takes $O(n)$ time to compute S for a single pair of indices and there are $O(n^2)$ pairs of indices, giving a total of $O(n^3)$ time and $O(n^2)$ space to explicitly compute and store S. Now this is very similar to the previous case, just that we return the pair (i,j) that maximize $S[i,j]$.

(d) If we want to use DP, an easy approach is to do like brute force, but reuse information in terms of smaller subproblems: for any $i < j$, it holds that $S[i,j] = S[i, j-1] + A[j]$. Using dynamic programing we still have $O(n^2)$ distinct subproblems, but now each one takes $O(1)$ time to be computed. This reduces the runtime to $O(n^2)$ (and keeps the same space bounds).

Instead, we introduce a linear time solution: let $P[j]$ represent the maximum possible score over all subarrays **ending** at index j. Note that this does not mean *any* range defined by some indices between 1 and j. It means a range that specifically *ends* at j.

Lemma 1. *For all $j > 0$ it holds that:*

$$P[j] = \begin{cases} A[j] & \text{if } P[j-1] \leq 0 \\ A[j] + P[j-1] & \text{otherwise.} \end{cases}$$

Proof. Regardless of what happens, any subarray ending at position j must include $A[j]$. The question is if is it advantageous to include other indices before it (and if so, how many). Any additional indices that we add will be $A[j-1]$, $A[j-2]$, and so on until some index i. Specifically, we are looking for the index i such that the score of $(i, j-1)$ is as large as possible. This value is, by definition, equal to If $P[j-1]$.

Notice that we will only add the chain from i to $j-1$ if it increases the score, and the score only increases if that chain has a positive score. Thus, if $P[j-1]$ is positive, the solution we are looking for is the whole chain from i to $j-1$ (and $A[j]$). Otherwise, $P[j-1]$ is negative and we drop it completely. Thus our solution consists of only $A[j]$. □

Note that the table $P[j]$ has size $O(n)$ and, using the previous Lemma, each entry can be computed in $O(1)$ time (it just looks at one other position in the array). Thus, we can build the table in $O(n)$ time. Once the table has been built we need to sweep the array, find the largest score, and return it. Both operations can be combined in one loop, but the runtime is unaffected. Space is also linear, since we only need to store an array of length n.

Note: with the approach described above we only return the highest score. How would you modify this to return the pair (i, j) that defines the highest score? There are several ways to do so, but one is very cool. Can you find it? (solution in next page).

By the way we built the array, we can easily find j (it will be the index whose position $P[j]$ is largest). To find i we could trace the Dynamic programming choices. However, there is a cooler method: iteratively add $A[j] + A[j-1] + \dots$ until for some index i the sum is equal to $P[j]$. We know it is feasible to do so, hence our algorithm must end and in at most $O(n)$ operations.

4. Remember the flip problem from Recitation 6? Give a dynamic programming approach to solve this problem

5. Still want more exercises? As of this writing, there are over 200 problems on Leetcode! (see https://leetcode.com/tag/dynamic-programming/). And this is just one of the many sites to practice your coding skills.