CS 3510 — Algorithms, Spring 2022

Lecture 8: More DP

February 3, 2022

Lecturer: Frederic Faulkner *Scribe: Aditya Diwakar*

The next lecture (Tuesday) is going to be a review sesion. The exam will be in the same lecture room. Also as a reminder, the notes must be handwritten on a singular side of a piece of paper.

Longest Increasing Subsequence

We are interested in the longest increasing subsequence. A subsequence is any number of elements in the same order as an original array with optional gaps between elements.

Increasing, in this context, means we want every next element in the subsequence to be greater than the previous element.

As a reminder, the steps to give a DP solution are as such:

1. What does each entry represent in the table?
2. What is the recurrence relation between table entries?
3. How do you implement your dynamic programming algorithm?
4. Perform a running time analysis.

A naive approach to this question involves computing every possible subsequenece, computing how many are strictly increasing, and then finding the subsequence with largest strictly increasing length. However, this requires computing 2^n subsequences (bad!).

Alternatively, let $T[i]$ represent the longest increasing subsequence from $a_1, \ldots, a_i$ (it may not include a_i, but it is the answer from the first to ith element).
Below is an example of the table filled in based on this entry definition:

array	5	8	-1	9	2	6	3	10	4	7
$T[i]$	1	2	2	3	3	3	3	4	4	5

We can fill in this table because we can see the small subsequences that exist in this array, but notice from the above that the subsequences we were tracking ($5, 8, 9, 10$ or $-1, 2, 6, 10$ or $-1, 2, 3, 10$) are none of the valid subsequences, as the $T[i]$ with 5 corresponds with the subsequence $-1, 2, 3, 4, 7$. Hence, this definition for entries does not work.

Instead, we can choose to let $T[i] =$ the length of the longest increasing subsequence in $a_1, \ldots, a_i$ that includes a_i. Now, in the same example as above, we have:

array	5	8	-1	9	2	6	3	10	4	7
$T[i]$	1	2	1	3	2	3	3	4	4	5

The recurrence relationship is given by:

$$T[i] = 1 + \max_{\forall j < i : a_j < a_i} (T[j])$$

unlike before where no recurrence relationship existed. This algorthim could be implemented like so (in pseudocode):

```
1  Function LIS(a_1, ..., a_n):
       /* let T be an empty n long list                          */
2      T ← []
3      for i ∈ 1 → n do
4          T[i] ← 1
5          for j ∈ 1 → i − 1 do
6              if a_j < a_i then
7                  T[i] ← max(T[i], T[j] + 1)
       /* answer is not simply T[n], rather it is max of table   */
8      ret ← 1
9      for i ∈ 1 → n do
10         ret ← max(ret, T[i])
11     return ret
```

The running time of this algorithm is $O(n^2)$ for the double nested loop and $O(n)$ for the max function, giving $O(n^2) + O(n) = O(n^2)$ as the total runtime.

Largest Sum Subarray

Let's say we have an input $a_1, \ldots, a_n$ (allowing for negative numbers), then we want a subarray with the maximum sum (a subarray is not the same as a subsequence, subarrays must be contiguous).

Let $T[i]$ be the maximum sum subarray in $a_1, \ldots, a_i$ that includes a_i, so what is our recurrence relation? Since we always include a_i, that is always part of the relation.

Now, if we use a_i, maybe we choose to use the highest possible from the previous (we can either use this or nothing at all). Since $T[i-1]$ is gives this maximum sum for the previous index, we can pick it if it makes our sum bigger (positive). Hence, the recurrence relation is given by:

$$T[i] = a_i + \max(T[i-1], 0)$$

An example of this could be:

array	5	-9	1	2	-1	5
$T[i]$	5	-4	1	3	2	7

In pseudocode, this would be implemeneted as:

```
1 Function MaxSumSubarray(a_1, ..., a_n):
    /* let T be an empty n long list                    */
2   T ← []
3   T[1] ← a_1
4   for i ∈ 2 → n do
5     T[i] ← a_i + max(T[i − 1], 0)
    /* answer is not simply T[n], rather it is max of table */
6   ret ← −∞
7   for i ∈ 1 → n do
8     ret ← max(ret, T[i])
9   return ret
```

Challenge Questions: Off by One Subsequence

We can represent $T[i]$ as the length of longest off by one subsequence from $a_i, \ldots, a_i$ that includes a_i. Hence, the recurrence relation is given by:

$$T[i] = 1 + \max_{\forall j \leq 1 : |a_j - a_i| = 1} (T[j])$$

where we can only pick a previous off by one subsequence the last element of that subsequence is off by one ($|a_j - a_i| = 1$).

The pseudocode and running time of this algorithm is the same as the LIS with a modified conditional, and so it is not presented.

Challenge Question: Picking Out Words

Let us say you have a block of text, but all of the spaces have been removed. For example, say we have `hellogeneralkenobi`. We want to figure out if we could split this block of text into multiple valid words.

You are given an algorithm `isValidWord()` that returns true or false in $O(1)$ time (for sake of the question). Don't read on until you have given this some thought. The solution is on the next page.

We represent $T[i]$ given by a boolean value determined by if the substring 0 to i can be broken into valid words. Using this, how can we get the recurrence relation?

The idea is to loop through all the earlier indices and see if the last few characters is a word and if the 0 to that index is also a series of words.

$$\underbrace{(c_1 \dots c_j)}_{\text{determined if word sequence by } T[j]} \qquad \overbrace{(c_{j+1} \dots c_i)}^{\text{check if this is a word using } \texttt{isValidWord}}$$

Hence, the recurrence relation can be given by:

$$T[i] = \bigvee_{j<i} (T[j] \wedge \texttt{isValidWord}(s_j, \dots, s_i)$$

The runtime of this algorithm is $O(n^2)$ as we perform $O(n)$ iterations to populate the table, and each entry requires $O(n)$ lookups giving a total runtime of $O(n^2)$.