CS 3510 Algorithms 10/5/2023

Lecture 10: Dynamic Programming II

Lecturer: Abrahim Ladha *Scribe(s): Adam Zamlynny*

1 Longest Common Substring

Take the two strings “EL GATO” and “GATER”. They both share the letter E which is a common substring, but they also contain the string “GAT” which is the longest string. Brute-force here, while not that bad (still in P) can be improved by DP. We do this using two degrees of freedom where the first is the index of the first string, and the second is the index of the second string.

a_1, a_2 be any strings with a_1y, a_2y having the longest common substring y. What is the longest common substring a_1yb_1, a_2yb_2. We might be able to append letters here if y is the suffix and y will remain the longest common substring. Thinking about this, we come to the following. If $b_1 = b_2$, then $yb_1 = yb_2$ is the longest common suffix of a_1yb_1, a_2yb_2. If $b_1 \neq b_2$, then y is the longest common substring of a_1yb_1, a_2yb_2

Think about what the last step is and how that will effect your answer. If you can inductively come up with an answer from previous state you'll save a lot of time.

Let's define the elements of our table. We'll have a two dimensional table T, indexed by the indices i and j. Index i is the index into the first string and j the index into the second string.

$$T[i,j] = \left\{ \begin{array}{ll} T[i-1,j-1]+1, & \text{if } x_i = y_j \\ 0, & \text{if } x_i \neq y_j \end{array} \right\}$$

Now we can write the code to fill in the table. Here the maximum suffix over all prefixes is the maximum of the substrings, so instead of finding that at the end, we're going to do it as we fill in the table. The

```
def lcsubstring(x, y):
    initialize dp as a table of size |x| + 1 by |y| + 1 as 0s
    max = 0
    maxpos = (0, 0)

    for i in 1..(|x| + 1)
        for j in 1..(|y| + 1)
            if x[i - 1] = y[j - 1]
                dp[i, j] = dp[i - 1, j - 1] + 1
            else
                dp[i, j] = 0

            if max < dp[i, j]
                max = dp[i, j]
                maxpos = i, j
```

Figure 1: Longest common substring algorithm.

Considering it in terms of suffices is easier than solving this problem in terms of substrings. The table stores the suffix lengths, and the longest suffix of a prefix is of course the longest substring. We'll fill in the table.

	_	E	L	G	A	T	O
_	0	0	0	0	0	0	0
G	0	0	0	1	0	0	0
A	0	0	0	0	2	0	0
T	0	0	0	0	0	3	0
E	0	1	0	0	0	0	0
R	0	0	0	0	0	0	0

Figure 2: Longest common substring example table.

When filling in the table, we only look if the two letters in the strings are equal and if they are we add one to the element to the left and up.

Runtime: $n \times m$ table with $\mathcal{O}(1)$ work for each entry, so $\mathcal{O}(nm)$ overall runtime and space complexity, where $n = |x|$, and $m = |y|$.

2 Longest Common Subsequence

Take the example $x = a, b, c, b, d, a, b$ and $y = b, d, c, a, b, a$. Subsequences are not contiguous, so for example the subsequences b, c, b, a and b, d, a, b are both valid. Again let's think about the last letter that we add since this will be eaiser for the dp approach: if we add one character, how does that change our state and our answer.

Let $X = x_1, ..., x_m$, $Y = y_1, ..., y_n$, $Z = z_1, ...z_k$ where Z is the longest common subsequence. Then if $x_m = y_n$, then $z_k = x_m = y_n$ and $z_1, ..., z_{k-1}$ is the longest common subsequence. If $x_m \neq y_n$, then $Z_k \neq X_m$ or $Z_k \neq Y_n$ and Z is still the longest common subsequence.

Let's define our recurrence $T[i,j]$ is the longest common subsequence of $x_1, ..., x_i$ and $y_1, ..., y_j$. If either of the i or j is zero, we want zero because there could not be a subsequence. If the two letters are equal we want to add that character so we take the longest common subsequence of both strings without that character and add one for that character. Otherwise, the longest common subsequence is the maximum of the longest common subsequences with one less character.

$$T[i,j] = \left\{ \begin{array}{ll} 0, & \text{if } i = 0 \text{ or } j = 0 \\ T[i-1, j-1] + 1, & \text{if } x_i = y_j \\ \max(T[i-1,j], T[i,j-1]), & \text{if } x_i \neq y_j \end{array} \right\}$$

```
def lcsubsequence(x, y):
    initialize dp as a table of size |x| + 1 by |y| + 1 as 0s
    initialize bt as a table of size |x| + 1 by |y| + 1 as 0s

    for i in 1..(|x| + 1)
        for j in 1..(|y| + 1)
            if x[i] = y[j]
                dp[i, j] = dp[i - 1, j - 1] + 1
                bt[i, j] = ↖
            else
                if dp[i, j - 1] < dp[j, i - 1]
                    dp[i, j] = dp[i - 1, j]
                    bt[i, j] = ←
                else
                    dp[i, j] = dp[i, j - 1]
                    bt[i, j] = ↑
```

Figure 3: Longest common subsequence algorithm.

	_	B	D	C	A	B	A
_	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	2	2
D	0	1	2	2	2	2	2
A	0	1	2	2	3	3	3
B	0	1	2	2	3	4	4

Figure 4: Longest common subsequence example dp table.

If you're trying to follow the notes, refer to both this table and the table below which shows how this table was made.

	_	B	D	C	A	B	A
_	·	←	←	←	←	←	←
A	↑	←	←	←	↖	←	↖
B	↑	↖	←	←	←	↖	←
C	↑	↑	←	↖	←	←	←
B	↑	↖	←	←	←	←	←
D	↑	↑	↖	←	←	←	←
A	↑	↑	←	←	↖	←	↖
B	↑	↖	←	←	↑	↖	↖

Figure 5: Longest common subsequence example backtracking table.

The arrows in the above table correspond to the backtracking which you could take to get an answer. Note that the arrows the algorithms produce give a strict solution.

Runtime: $n \times m$ table with $\mathcal{O}(1)$ work for each entry, so $\mathcal{O}(nm)$ overall runtime and space complexity, where $n = |x|$, and $m = |y|$.

3 Longest Palindromic Subsequence

Suppose you have as input one string $x = a_1, ..., a_n$ and we want to find the longest palindromic subsequence. Notice quickly that this is lcsubsequence($a_1, ..., a_n$, $a_n, ..., a_1$).

We can construct the dp array dp[n][n] with dp[i][j] = largest palindromic subsequence from $a_i ... a_j$. We have the base cases where empty strings and single characters are palindromic, so $\forall i$ dp[i][i] = 1.

If $a_i, ..., a_j$ has the fact that $a_i = a_j$, then it could be the ends of a palindrome, but it depends on $a_i(a_{i+1}...a_{j-1})a_j$, so we obtain the recurrence $T[i, j]$ below:

$$T[i,j] = \left\{ \begin{array}{ll} 2 + T[i+1, j-1], & \text{if } x_i = x_j \\ \max(T[i+1,j], T[i,j-1]), & \text{if } x_i \neq x_j \end{array} \right\}$$

```
def lpalindromesubsequence(x1...x_n):
    initialize dp as a table of size n by n as 0s

    for i in 1...(n)
        dp[i][i] = 1

    for s in range 1...(n)
        for i in range n-s
            j = i + s
            if x[i] = x[j]
                dp[i][j] = 2 + dp[i+1][j-1]
            else
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])
```

Figure 6: Longest palindromic subsequence algorithm.

Runtime: $n \times n$ table with $\mathcal{O}(1)$ work for each entry, so $\mathcal{O}(n^2)$ overall runtime and space complexity, where $n = |x|$.