3510
Lecture

Longest Sequences

Longest sequences is a class of common problems with efficient dynamic programming solutions. You are given a pair of arrays/strings/sequences and asked to find the longest common substring or subsequence. A substring is contiguous while a subsequence need not be. There is also the ed.f-distance problem and longest palindromic substring/subsequence. All use 2D DP tables

problem: Longest common substring: Given two strings, find the longest substring of each which is equal. For example, for gator, elgato, it is "gat" of length 3. We solve this slightly in an odd way. What is a substring, if not just a suffix of all possible prefixes. We solve the problem of longest common suffix, but do so for all prefixes, so the max will be the longest common substring.

solution: array: $dp[n][m]$ for strings of length n, m. $dp[i][j]$ contains the value for the longest common suffix for prefixes $a_1 \ldots a_i$ and $b_1 \ldots b_j$. The max of the table is then the longest common substring.

base cases: for prefixes of length zero, the longest common suffix can only be the string of length zero, so one row and column of all zeroes.

recurrence: for strings $a_1 \ldots a_i y$ and $b_1 \ldots b_j y$ with y the longest common suffix, add the next symbol to each, $a_1 \ldots a_i y a_{i+1}$, $b_1 \ldots b_j y b_{j+1}$. Notice that $y a_i$ is the longest common suffix of both only if $a_i = b_j$. Given some current shared longest suffix, the next suffixes are common if its the same letter. So our recurrence follows

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1 & \text{if } a_i = b_j \\ 0 & \text{if } a_i \neq b_j \end{cases}$$

if $a_i \neq b_j$, they cannot share any suffix.

implementation:

```
def lcsubstring(a_1 ... a_n, b_1 ... b_m):
    dp = [0..n][0..m]
    max = 0
    max pos = (0,0)
    for i in 1..n:
        for j in 1..m:
            if a_i = b_j
                dp[i][j] = dp[i-1][j-1] + 1
            else
                dp[i][j] = 0
            if max < dp[i][j]:
                max = dp[i][j]
                max pos = (i,j)
```

example

		e	l	g	a	t	o
	0	0	0	0	0	0	0
g	0			1			
a	0				2		
t	0					3	
e	0	1					
r	0						

so max substring is of length 3
notice the position also returns not just the length but the max substring itself.

runtime: nxm table with O(1) work for each so O(nm)

What about longest common subsequence? A subsequence need not be contiguous. for example, lcsubsequence(b d c a b a b c b d a b) = 4 bcba

solution

array: let $dp[0..n][0..m]$ with $dp[i][j]$ = longest common subsequence of $a_1 .. a_i$, $b_1 .. b_j$. $dp[n][m]$ is then the length of our longest common subsequence

base cases: if the sequences a or b are empty then there is no longest common subsequence save the zero

so $\forall i \; 0 \le i \le n \quad d[i][0] = 0$
$\forall j \; 0 \le j \le m \quad d[0][a_j] = 0$.

recurrence. suppose $a_1 .. a_{i-1}$ and $b_1 .. b_{j-1}$ have some longest common subsequence of $k = dp[i-1][j-1]$. If you increase the knowledge of both to $a_1 .. a_i$, $b_1 .. b_j$ then the lagest common subsequence of this is $k+1$ if $a_i = b_j$. If it doesn't, then its either the lcs subsequence of $a_1 .. a_i$ $b_1 .. b_{j-1}$ or $a_1 .. a_{i-1}$, $b_1 .. b_j$, which ever is greater. so this is our recurrence.

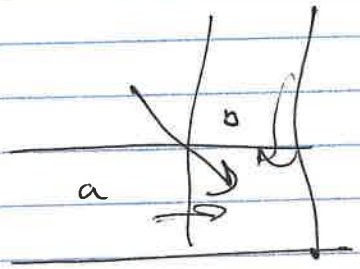
$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1 & \text{if } a_i = b_j \\ \max(dp[i-1][j], dp[i][j-1]) & \text{if } a_i \neq b_j \end{cases}$$

if $a_i = b_j$, then the current lcs subsequence is increased by appending $a_i = b_j$. Otherwise, its the longest of the two lcs subsequences $a_1 .. a_i$, $b_1 .. b_{j-1}$ or $a_1 .. a_{i-1}$, $b_1 .. b_j$.

we will implement this to not only give us the length, but the sequence itself in a creative way. we maintain two tables this time.

Implementation:

```
def lcsubsequence(a_1 .. a_n, b_1 .. b_m):
  dp = [0..n][0..m] all zeroes
  bt = [0..n][0..m] all zeroes
  for i in (1..n):
    for j in (1..m):
      if a_i == b_j:
        dp[i][j] = dp[i-1][j-1] + 1
        bt[i][j] = "↖"
      elif dp[i-1][j] < dp[i][j-1]:
        dp[i][j] = dp[i][j-1]
        bt[i][j] = "↑"
      else
        dp[i][j] = dp[i-1][j]
        bt[i][j] = "←"
```



mck(a,b) otr

		B	D	C	A	B	A	
	0	0	0	0	0	0	0	
A	0	0	0	0	1	1	1	A ↑
B	0	1	1	1	1	2	2	B ↑
C	0	1	1	2	2	2	2	C ↑
B	0	1	1	2	2	2	2	B ↖
D	0	1	2	2	2	2	2	D ↑ ←
A	0	1	2	2	3	3	3	A ↖ ←
B	0	1	2	2	3	4	4	B ↖ ←

B D C A B A

The table of arrows helps us reconstruct the actual subsequence which is the path of "↖", BDAB.

```
...a...b...c...c | d
..a...b...c..... | c
```

$a_i \neq b_j$ so must be a b c c instead.

runtime: n×m table with O(1) work done at each, so $O(nm)$.

problem: suppose you have as input one string $a_1 \dots a_n$ and want to find the longest palindromic subsequence. Notice quickly that this is lc subsequence $(a_1 \dots a_n, a_n \dots a_1)$. We do it manually

solution array: $dp[1..n][1..n]$ with $dp[i][j]$ = longest palindromic subsequence from $a_i \dots a_j$. Will return $dp[1][n]$

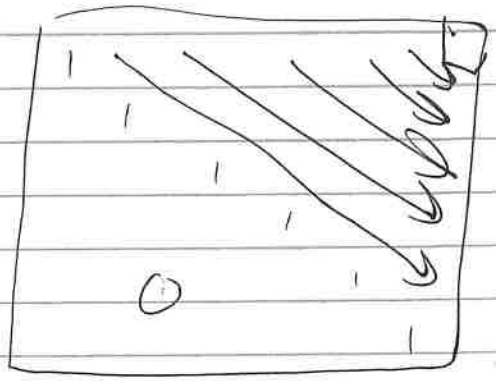
base cases: single characters and empty strings are palindromic, so $\forall i \; dp[i][i] = 1$
also note $dp[i][i+1] = 2$ if $a_i = a_{i+1}$

recurrence. If $a_i \dots a_j$ has the fact that $a_i = b_j$ then this could be the ends of a palindrome, but it depends on $a_i (a_{i+1} \dots a_{j-1}) a_j$. so

$$dp[i][j] = \begin{cases} 2 + dp[i+1][j-1] & \text{if } a_i = a_j \\ \max(dp[i+1,j], dp[i,j-1]) & \text{if } a_i \neq a_j \end{cases}$$

implementation:

```
def lc palindromic subsequence (a_1 ... a_n):
    dp[1..n][1..n] all zeroes
    for i in range n: dp[i][i] = 1
    for s in range 1..n:
        for i in range n-s:
            j = i+s
            (recurrence here)
```



runtime: $O(n^2)$ for similar reasons.