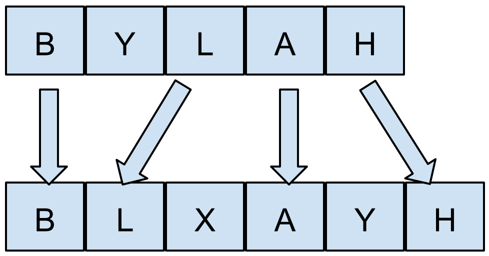
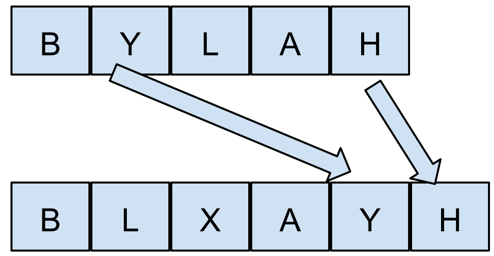
Longest Common Subsequence (LCS)

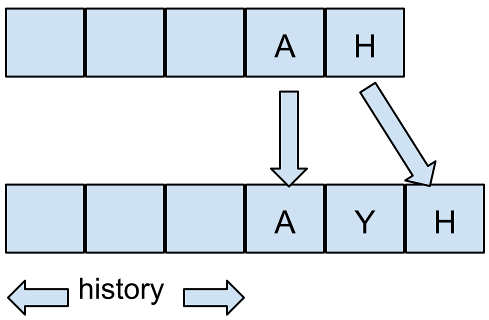


BLAH length = 4

We can state it as a choice problem. Choose a set of matching pairs.

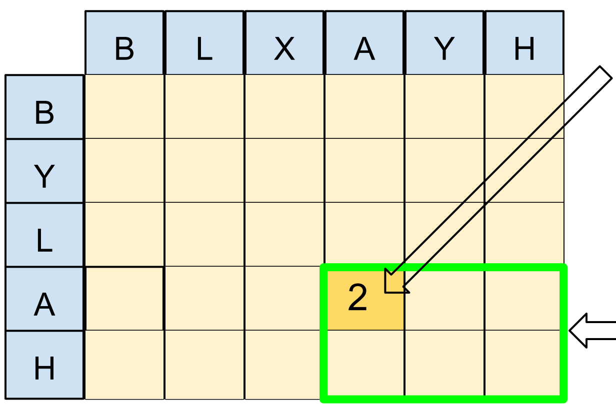


YH length = 2



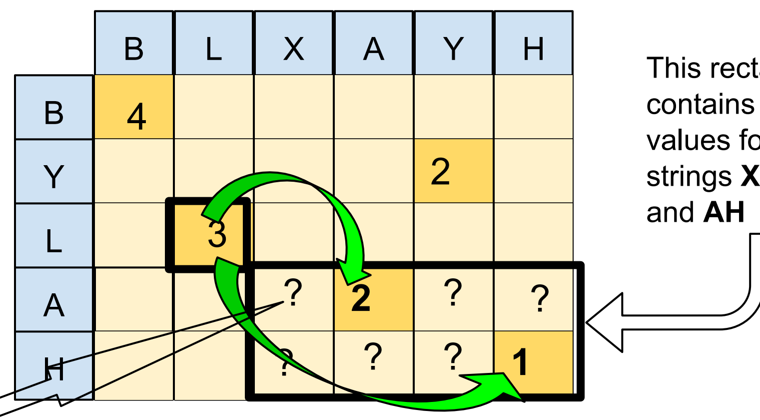
Obviously history can be ignored. The best result for strings **AH** and **AYH** is 2, and it is always =2 regardless of the history. The idea is to save this result to the table .

The choice is a pair of values , so we will use 2D table.



This cell corresponds to the **AA** pair.We want to save the best length here.

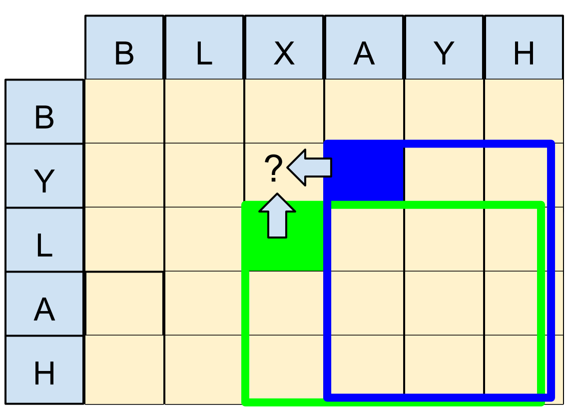
Green rectangle represents two substrings: **AYH** and **AH**



This rectangle contains all values for strings **XAYH** and **AH**

We want to have the max value (2) in this cell, so we will read it from here instead of scanning the whole rectangle.

Filling the table (or derive new value from already known values). Find some matching pair. (**LL** in the example above) . Scan all matching pairs in the corresponding rectangle and find the max (=2 in this example). Then add 1 because we increase the length. So the best length for strings **LXAYH** and **LAH** is 3 and it is derived from **XAYH** and **AH.** Scanning matching pairs creates additional complexity and can be avoided by filing not matching cells.



Filling not matching cells.
Two overlapping rectangles green and blue. Consider that left top corner keeps the max value for all cells that belongs to this rectangle. So ? = max(green,blue). In other words:
Tab[x][y] = max(Tab[x + 1][y],Tab[x][y + 1]);

```
if (pA[x] == pB[y]) { // matching pair
  Tab[x][y] = Tab[x+1][y+1] + 1;
}
else { // propagate max value
  Tab[x][y] = max(Tab[x + 1][y],Tab[x][y + 1]);
}
```