# TREES, INDUCTION, AND DYNAMIC PROGRAMMING

RECITATION WEEK 7

# INDUCTION

- By finding a way of parametrizing and organizing structures (e.g., trees, graphs, lists) we may perform induction on them

- Show that any non-empty rooted binary tree with $n$ vertices has *exactly* $n - 1$ edges.

# EDGES IN A BINARY TREE

- How to organize the trees? Number of leaves/nodes/height?
- We'll work with the number of nodes
- Base case: n = 1
- There is exactly one tree with a single node and this tree has no edges.
- Inductive assumption: Assume that for all $0 < k < n$ and any binary tree $T$ with $k$ nodes, the number of edges in $T$ is $k-1$.
- Take any rooted binary tree $T'$ with $n$ nodes. Remove a leaf and its incident edges from $T'$ to generate another tree $T''$. $T''$ is a rooted binary tree with $n-1$ nodes. Therefore, by our inductive hypothesis, $T''$ has $n-2$ edges. But since we removed a leaf from $T'$ to generate $T''$ and $T'$ is a tree we removed exactly one incident edge from $T'$. Therefore, $T'$ has $n-1$ edges.

# ALTERNATIVE PROOF

- Show that any non-empty rooted binary tree with $n$ vertices has *exactly* $n - 1$ edges.
- We can approach this problem without induction using definitions and a charging argument
- A charging argument assigns costs (charges) to a source of payments ensuring that at the end of the process we do not have a negative amount of "money".
- In this problem we may charge edges to vertices.
- Since we are in a rooted binary tree, every child has exactly one parent (and therefore one "incoming" edge). Starting from the leaves, charge the edges incident on the leaves to the leaves and remove them. At the end of one phase of this process, we have a balance of 0 and have generated new leaves. We may repeat this process and continually end up with a balance of 0. This process will continue until there is only one vertex remaining, the root of the tree. This means that there is exactly one more vertex than there are edges in a rooted binary tree.

# INDUCTION

- Using induction, show that the following sorting algorithm is correct.

1. Given a list A with n elements, let x = A[1] and B=A[2...n]
2. If B is not empty, recursively sort B
3. Loop over the elements of B and insert x immediately to the left of the first element that exceeds it or at the end if no such element exists.
4. Return B

- Base case: A has length 1. Our algorithm just recreates A which is sorted.
- Inductive hypothesis: Assume that our algorithm correctly sorts all lists of length $1 \leq k < n$
- Suppose A is a list of *n* elements. Our algorithm first splits A into the first element, x, and a list of the remaining elements, B. B has *n-1* elements and our algorithm recursively sorts *B* which, by assumption, it does correctly. Since our algorithm then finds the first element in *B* greater than *x* and places *x* to its left, *x* must exceed everything to its left. Since *B* is sorted, *x* is also less than everything to its right. Therefore, *A* must be sorted and our algorithm works correctly.

# DYNAMIC PROGRAMMING – LONGEST INCREASING SUBSEQUENCE

- Dynamic programming is “recursion with some memory”
- Given a problem, we propose a recursive solution and then see if we are recalculating the same thing (overlapping subproblems) multiple times
- If we are, we use some additional memory to store the intermediate results or calculate the solutions to the recursive problems in some bottom-up fashion

- Given a sequence of $n$ non-negative numbers, output the length of the longest increasing subsequence. That is, given an unsorted array *A* of numbers, find the size of the largest set of indices *S* such that the array produced by including only elements of *A* at the indices in *S* is monotonically increasing
- Example: [5,0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15]
- Solution: 6 (<0,2,6,9,11,15>)

# DYNAMIC PROGRAMMING – LONGEST INCREASING SUBSEQUENCE

- Given a sequence of $n$ non-negative numbers, output the length of the longest increasing subsequence (LIS). That is, given an unsorted array *A* of numbers, find the size of the largest set of indices *S* such that the array produced by including only elements of *A* at the indices in *S* is monotonically increasing
- How would we solve this using recursion?
  - Hint: The longest increasing subsequence either contains the first element or it doesn't
- LIS(array, start, stop, end value)
- LIS(A, 1, n, -1) = max(LIS(A, 2, n, -1), 1+LIS(A, 2, n, A[1])
- LIS(A, i, n, prev) = max(LIS(A, i+1, n, prev), 1+LIS(A, i+1, n, A[i])
- This is exponential time: $T(n) \geq 2T(n-1) + \Theta(1)$
- However, there appears to be a lot of overlapping subproblems

# LONGEST INCREASING SUBSEQUENCE

- Given a sequence of $n$ non-negative numbers, output the length of the longest increasing subsequence (LIS). That is, given an unsorted array *A* of numbers, find the size of the largest set of indices *S* such that the array produced by including only elements of *A* at the indices in *S* is monotonically increasing
- Using dynamic programming (and only O(n) auxiliary memory), propose a polynomial time algorithm solving this problem.
  - Hint: Rather than considering whether or not index $i$ begins a subsequence, we can consider whether or not it continues a subsequence
  - Hint: What is the length of the longest increasing subsequence *ending* at position 1? Position i?
- Dynamic programming algorithm:
  - Declare an auxiliary memory array *B* of length *n*. B[i] will store the length of the longest subsequence ending at A[i].
  - Beginning at the left hand side of A, compute the length of the longest increasing subsequence ending at A[i] by passing over all A[k] for k < i. If A[i] > A[k] then A[i] can extend the longest subsequence ending at k. Select the largest such subsequence by looking at B. Output max(B).
  - Note: By maintaining a third array C of *n* back pointers, we may find the longest increasing subsequence as well.

# DYNAMIC PROGRAMMING – LONGEST COMMON SUBSEQUENCE

- In computational biology, it is often important to compare the DNA, RNA, or polypeptide chains present in two different organisms to measure biological similarity or diversity.
- A fast means of doing so has become so important that the research papers which proposed popular current algorithms BLAST and Clustal have been each cited over 50,000 times.
- We will consider an easier related question and find an algorithm which, given two sequences of characters, A and B, finds the longest common subsequence between them.
- Say length(A) = *n* and length(B) = *m.* A very naïve algorithm may look at all subsequences of A and check if they are present in B.
  - What would the running time of this be? How many subsequences of A are there?

# LONGEST COMMON SUBSEQUENCE

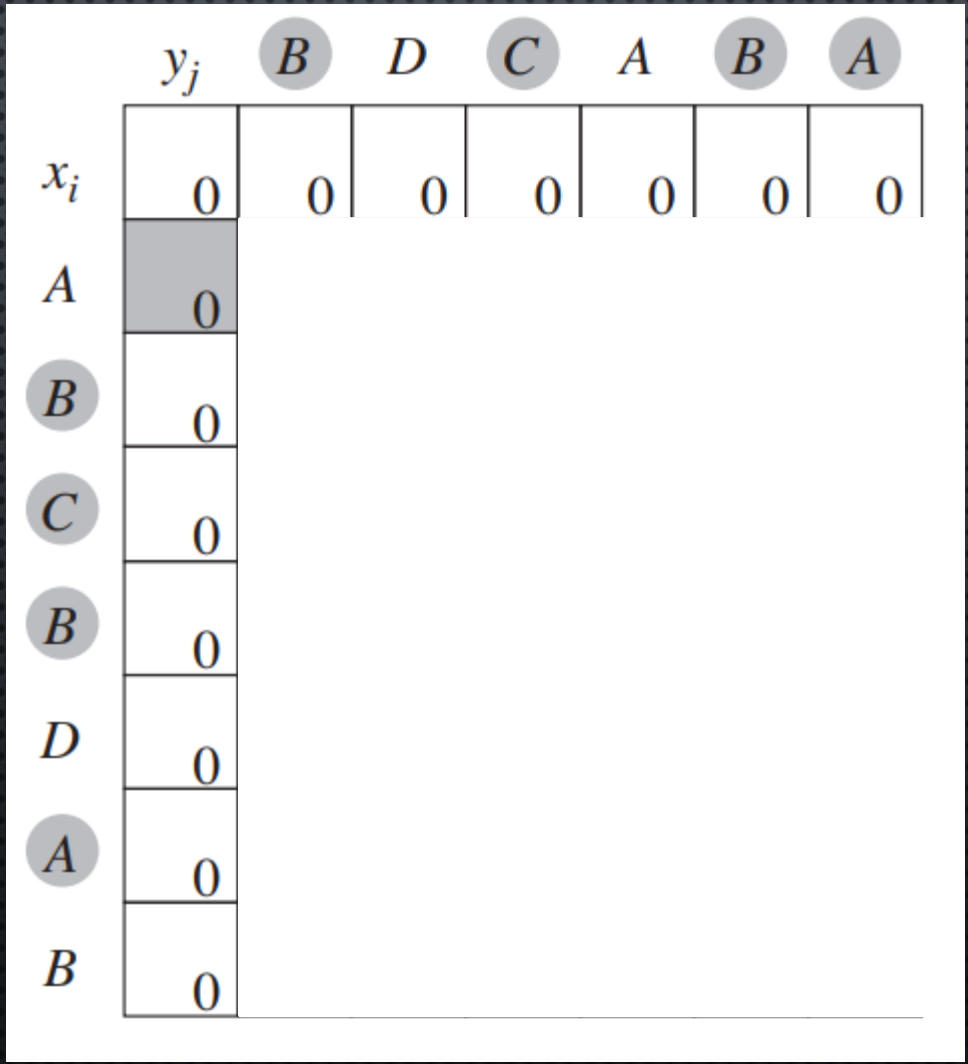
- We will consider an easier related question and find an algorithm which, given two sequences of characters, A and B, finds the longest common subsequence between them.
- Say length(A) = *n* and length(B) = *m.* Propose an alternative recursive algorithm to solve this problem.
  - Hint: If A[n] = B[m] what is the length of the longest common subsequence of A and B? What if A[n] and B[m] are different?
- LCS(A, B, i, j)
- LCS(A,B,0,j) = LCS(A,B,i,0) = 0
- LCS(A,B, n,m) = 1 + LCS(A,B, n-1, m-1) if A[n] = B[m]
- LCS(A,B,n,m) = max(LCS(A,B,n,m-1),LCS(A,B,m-1,n))
- There is a lot of overlapping problems in this as well, so we will try a bottom-up dynamic programming solution again.

# LONGEST COMMON SUBSEQUENCE

- We'll use our recursive equations as a guide for how to proceed
  - LCS(A, B, i, j)
  - LCS(A,B,0,j) = LCS(A,B,i,0) = 0
  - LCS(A,B, n,m) = 1 + LCS(A,B, n-1, m-1) if A[n] = B[m]
  - LCS(A,B,n,m) = max(LCS(A,B,n,m-1),LCS(A,B,m-1,n))
- Let's create a table of (m+1)(n+1) entries, this will be used to store the intermediate solutions for our problem. In the first row and column of this table we'll place zeros. Then, in row major order, we'll calculate the entries of our table using our recursive equations. We'll also place an arrow indicating which "direction" we are maximized from (since we will move backward in only one sequence). If A[i] and B[j] match, we'll use a diagonal arrow to indicate that we need to move backward in both sequences.

# LONGEST COMMON SUBSEQUENCE

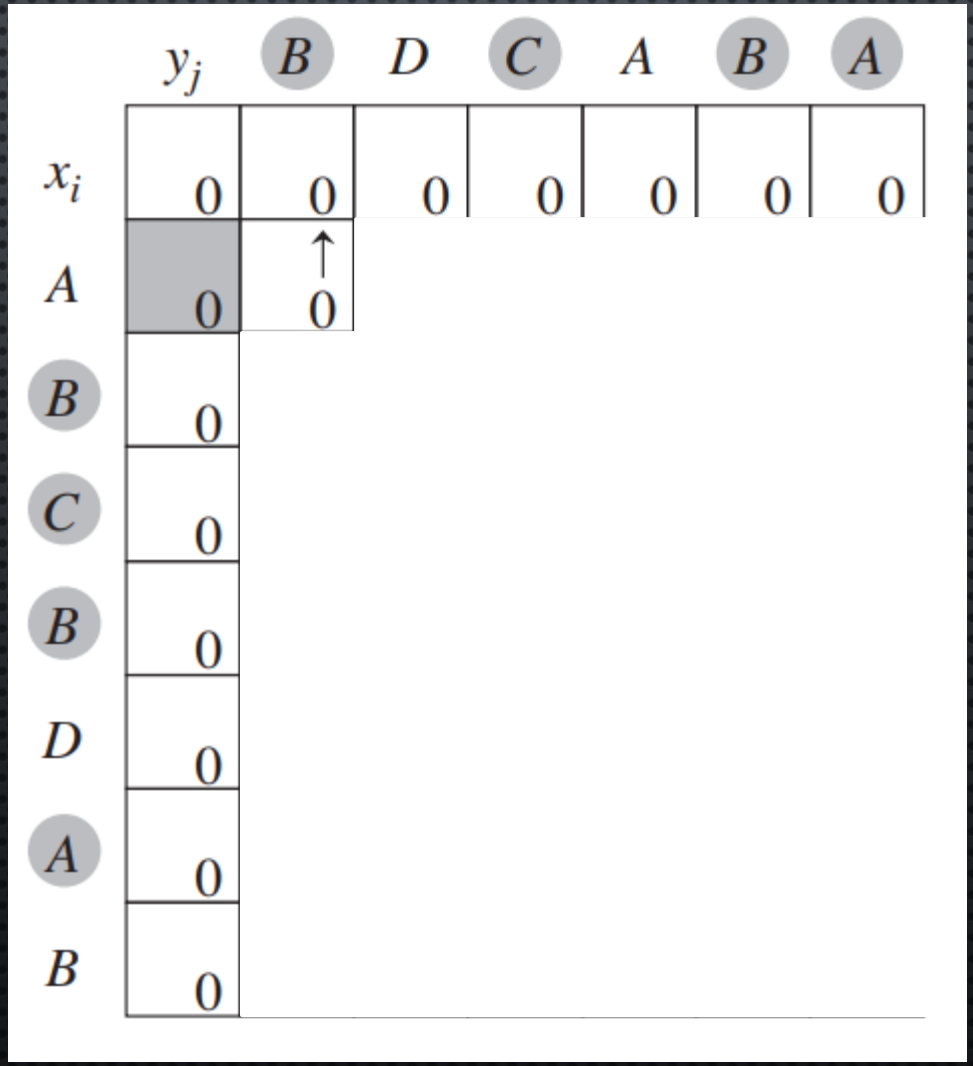
- An example, A = (abcdbab), B = (bdcaba)



| | $y_j$ | B | D | C | A | B | A |
|---|---|---|---|---|---|---|---|
| $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | | | | | | |
| B | 0 | | | | | | |
| C | 0 | | | | | | |
| B | 0 | | | | | | |
| D | 0 | | | | | | |
| A | 0 | | | | | | |
| B | 0 | | | | | | |

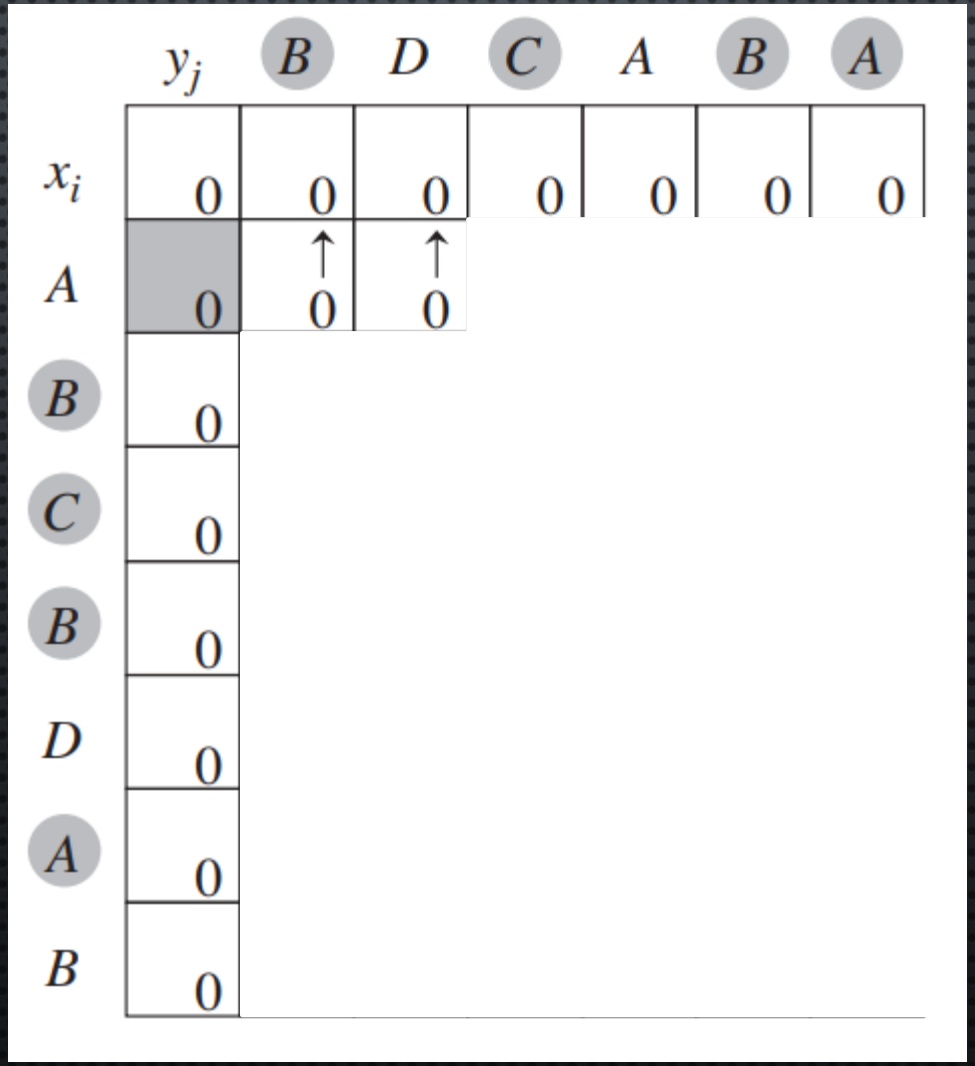
# LONGEST COMMON SUBSEQUENCE

- An example, A = (abcdbab), B = (bdcaba)



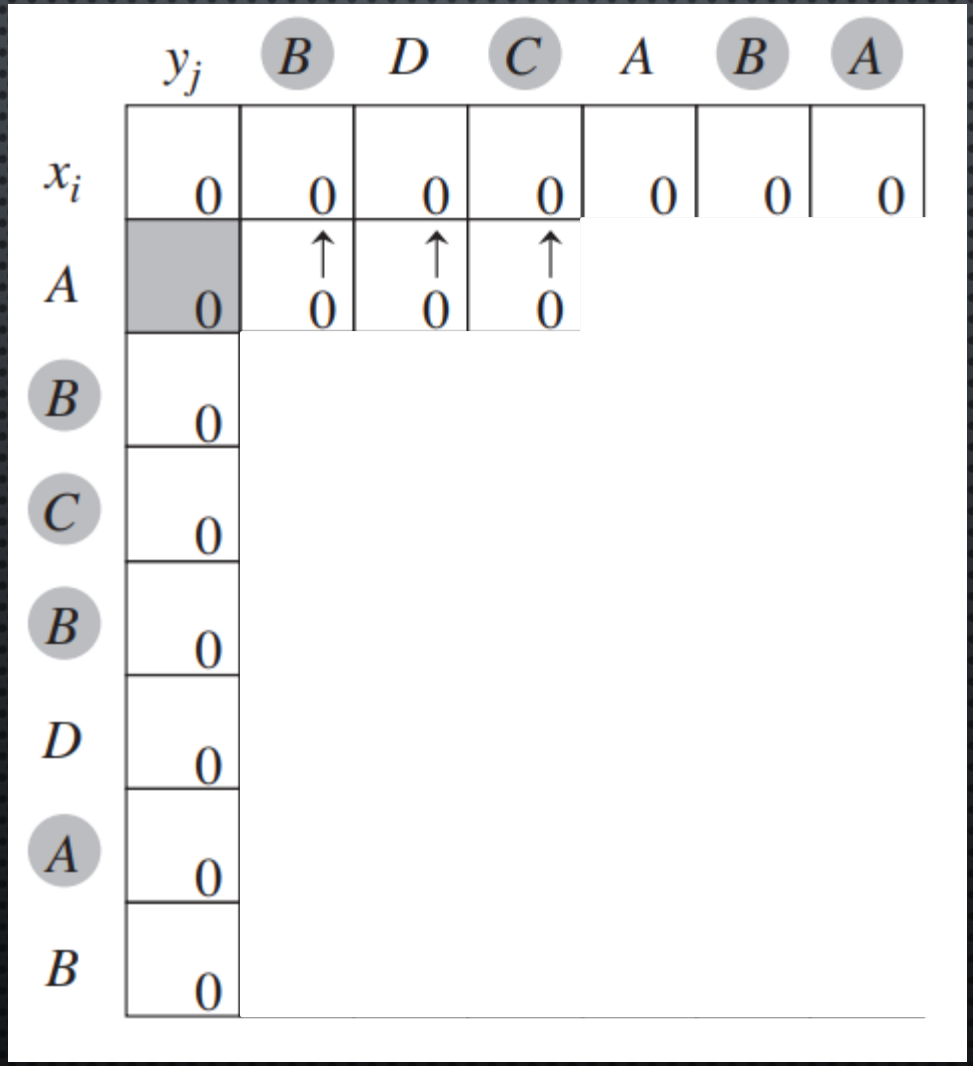
# LONGEST COMMON SUBSEQUENCE

- An example, A = (abcdbab), B = (bdcaba)



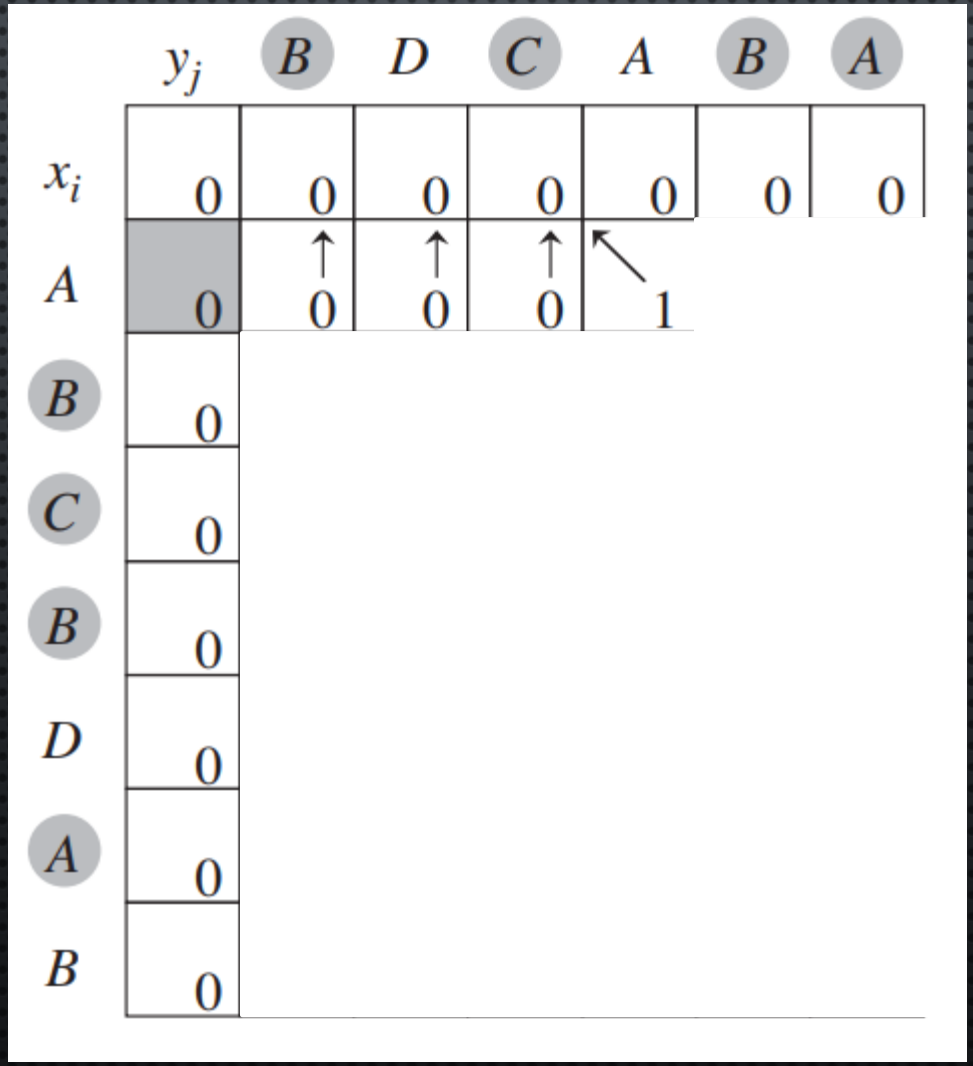
# LONGEST COMMON SUBSEQUENCE

- An example, A = (abcdbab), B = (bdcaba)



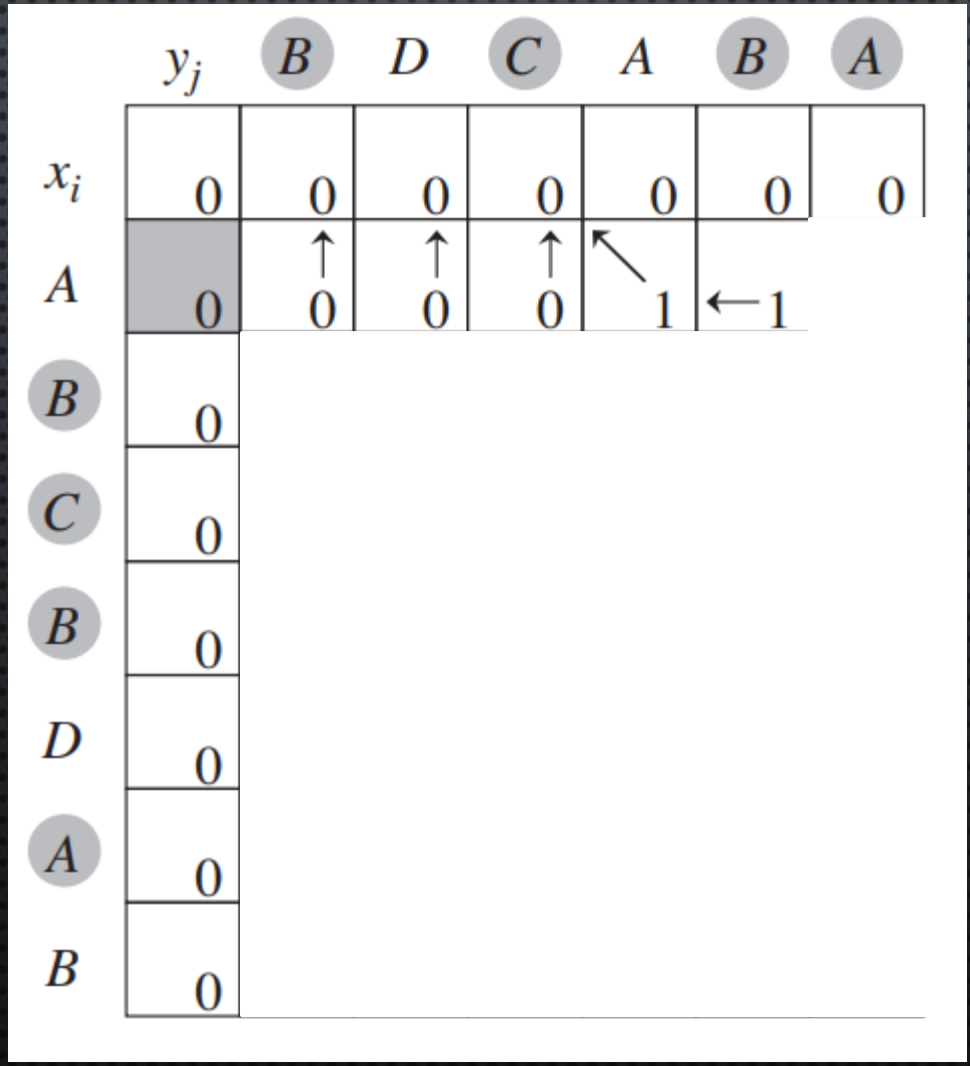
# LONGEST COMMON SUBSEQUENCE

- An example, A = (abcdbab), B = (bdcaba)



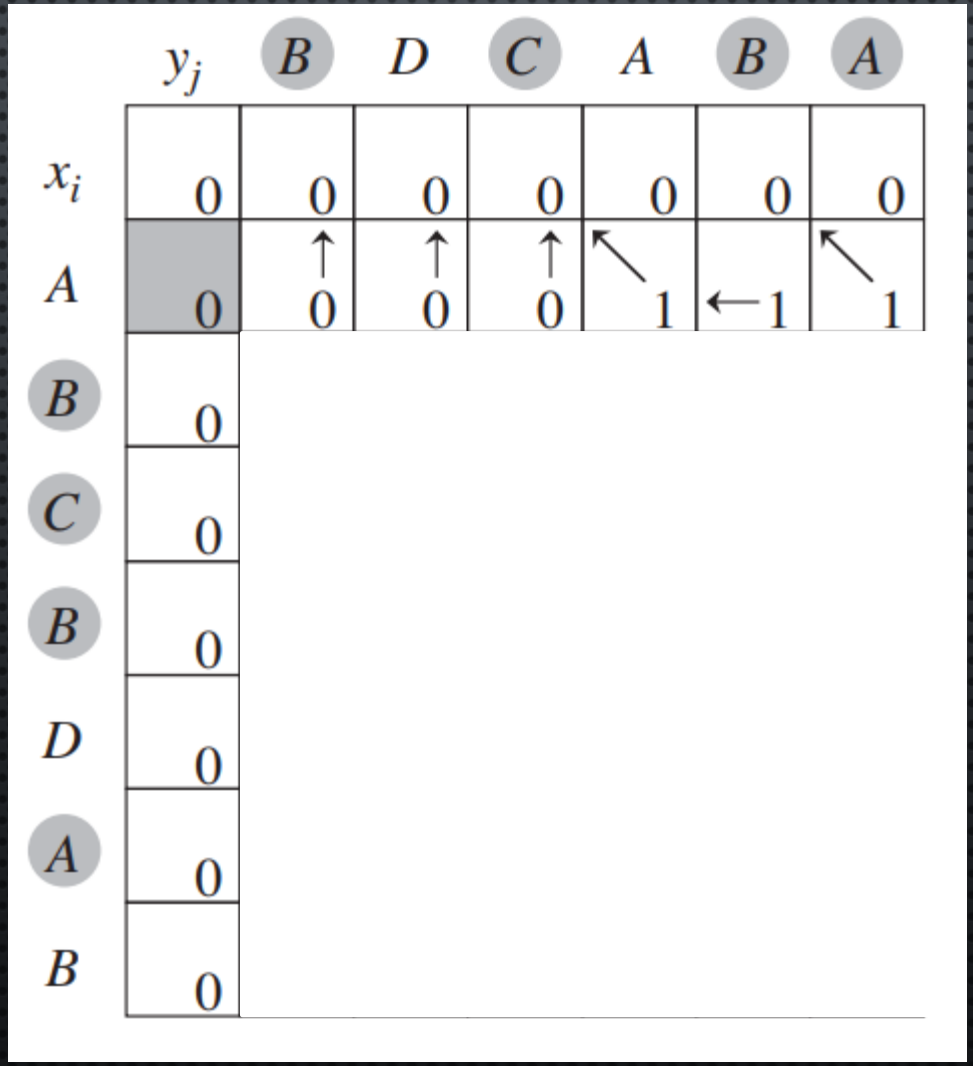
# LONGEST COMMON SUBSEQUENCE

- An example, A = (abcdbab), B = (bdcaba)



# LONGEST COMMON SUBSEQUENCE

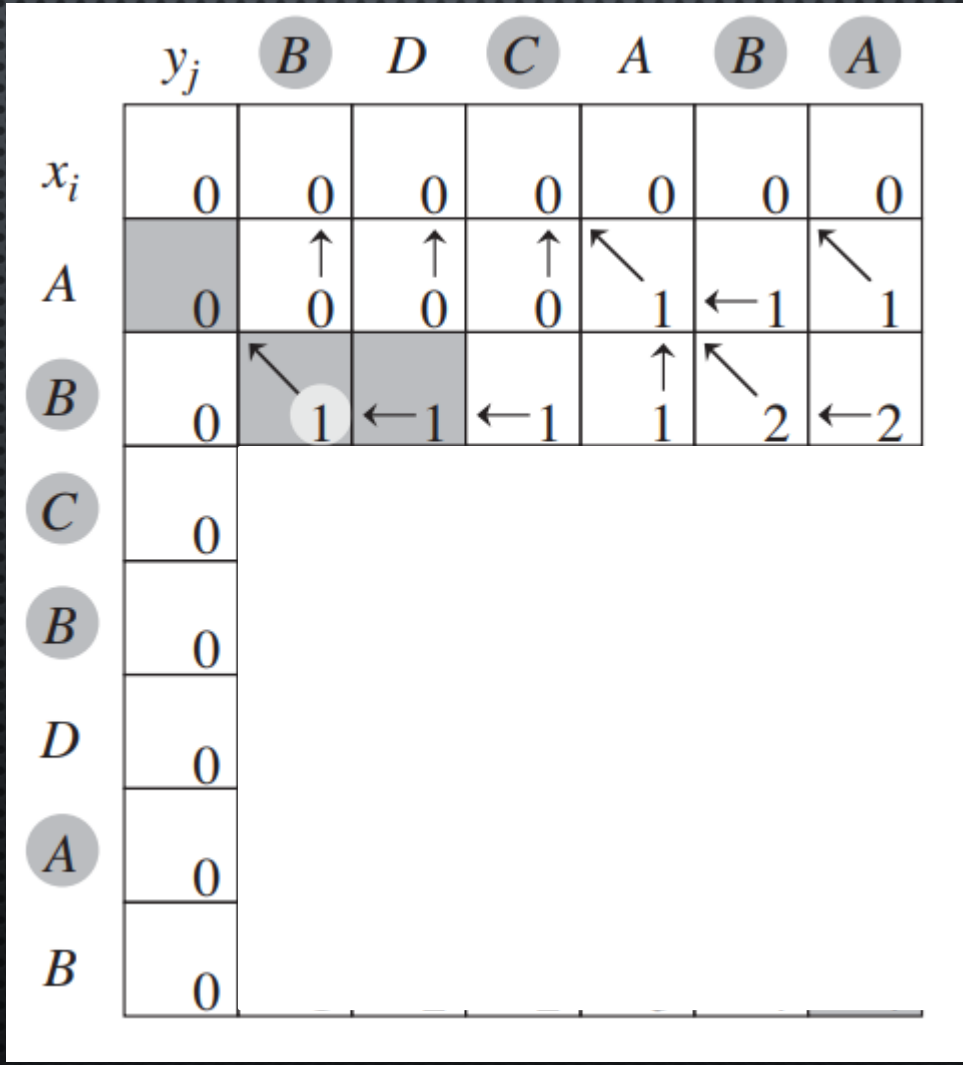
- An example, A = (abcdbab), B = (bdcaba)



| | | $y_j$ | B | D | C | A | B | A |
|---|---|---|---|---|---|---|---|---|
| | $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 | ← 1 | ↖ 1 |
| B | | 0 | | | | | | |
| C | | 0 | | | | | | |
| B | | 0 | | | | | | |
| D | | 0 | | | | | | |
| A | | 0 | | | | | | |
| B | | 0 | | | | | | |

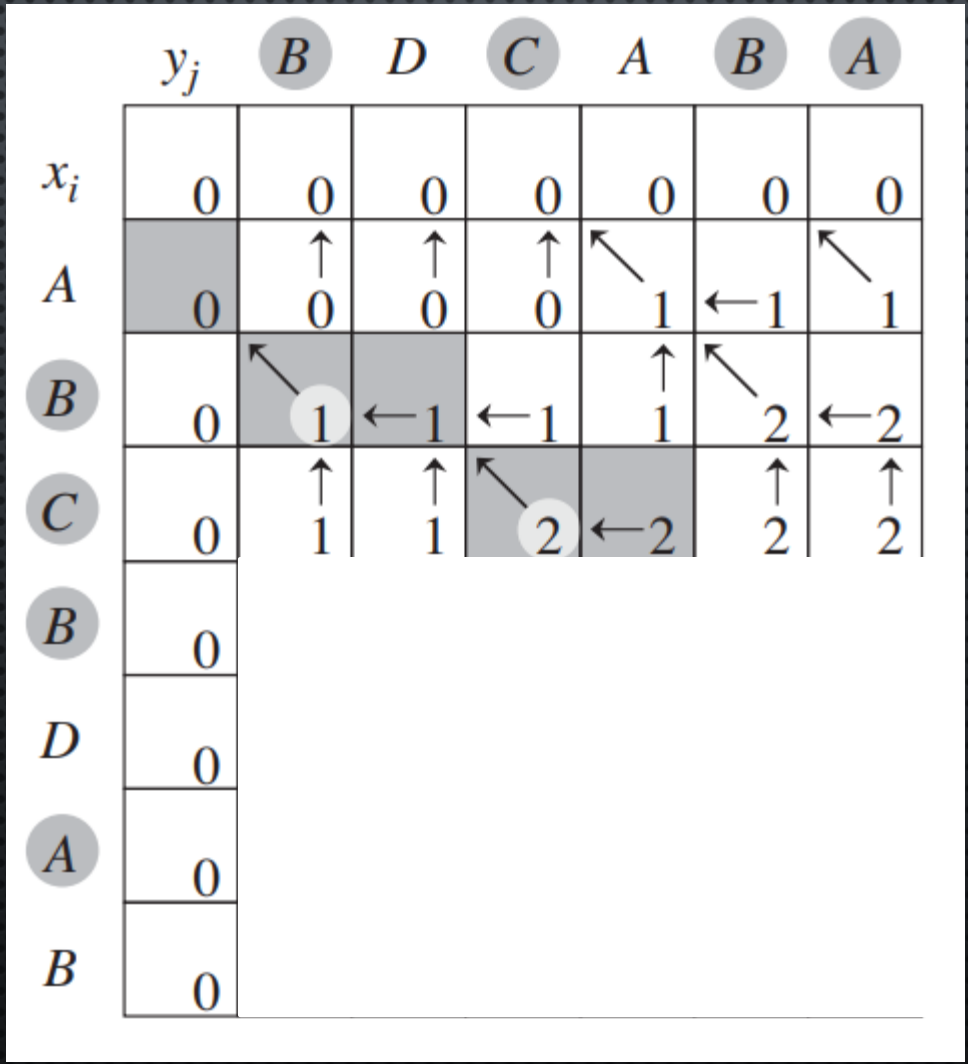
# LONGEST COMMON SUBSEQUENCE

- An example, A = (abcdbab), B = (bdcaba)



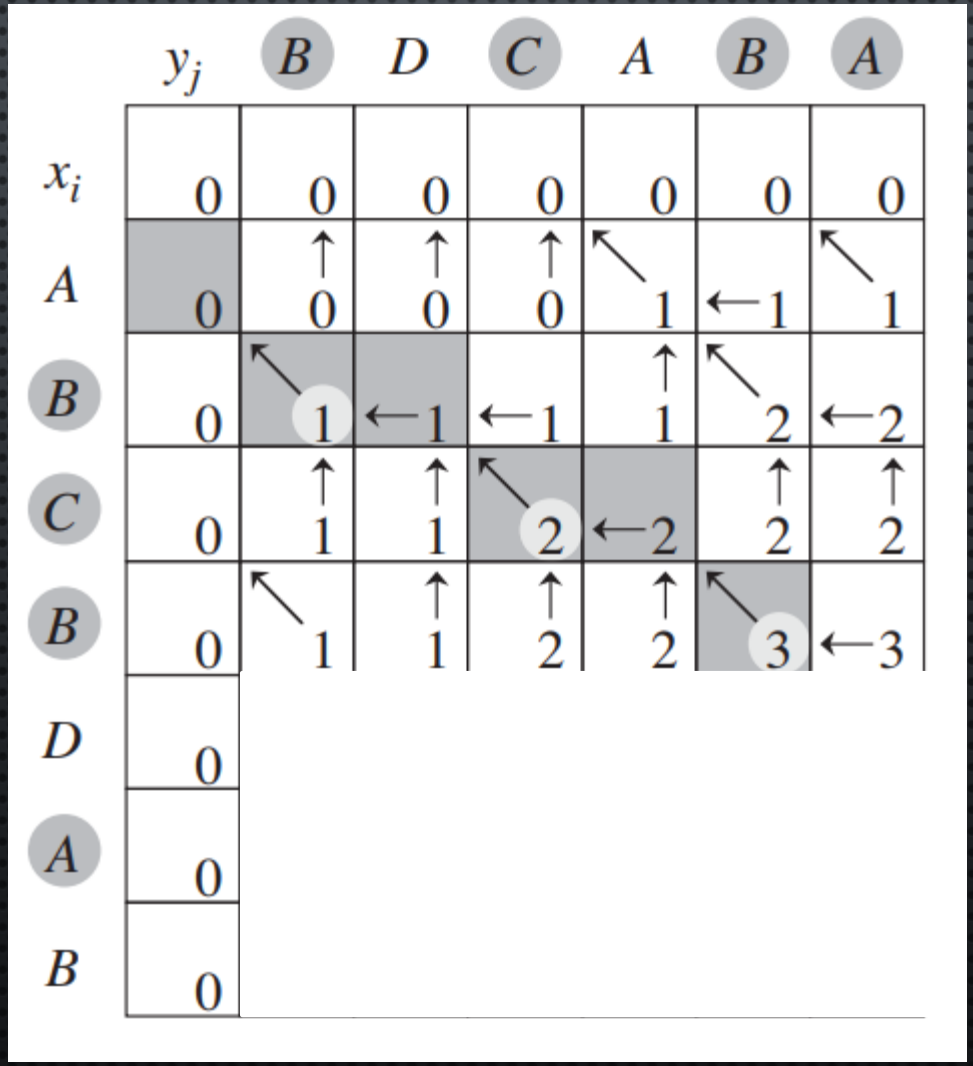
# LONGEST COMMON SUBSEQUENCE

- An example, A = (abcdbab), B = (bdcaba)



# LONGEST COMMON SUBSEQUENCE

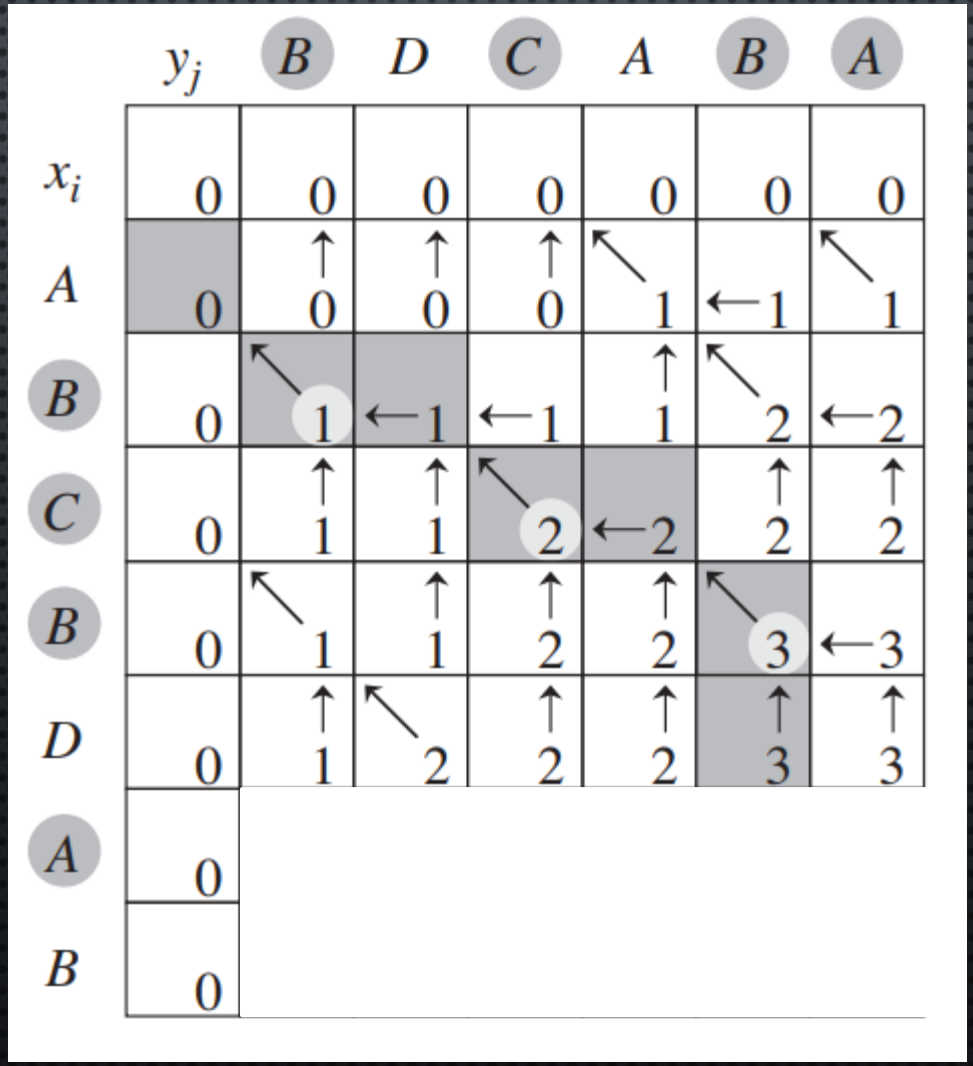
- An example, A = (abcdbab), B = (bdcaba)



| | | $y_j$ | B | D | C | A | B | A |
|---|---|---|---|---|---|---|---|---|
| | $x_i$ | 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 | ↖ 3 | ← 3 |
| D | | 0 | | | | | | |
| A | | 0 | | | | | | |
| B | | 0 | | | | | | |

# LONGEST COMMON SUBSEQUENCE

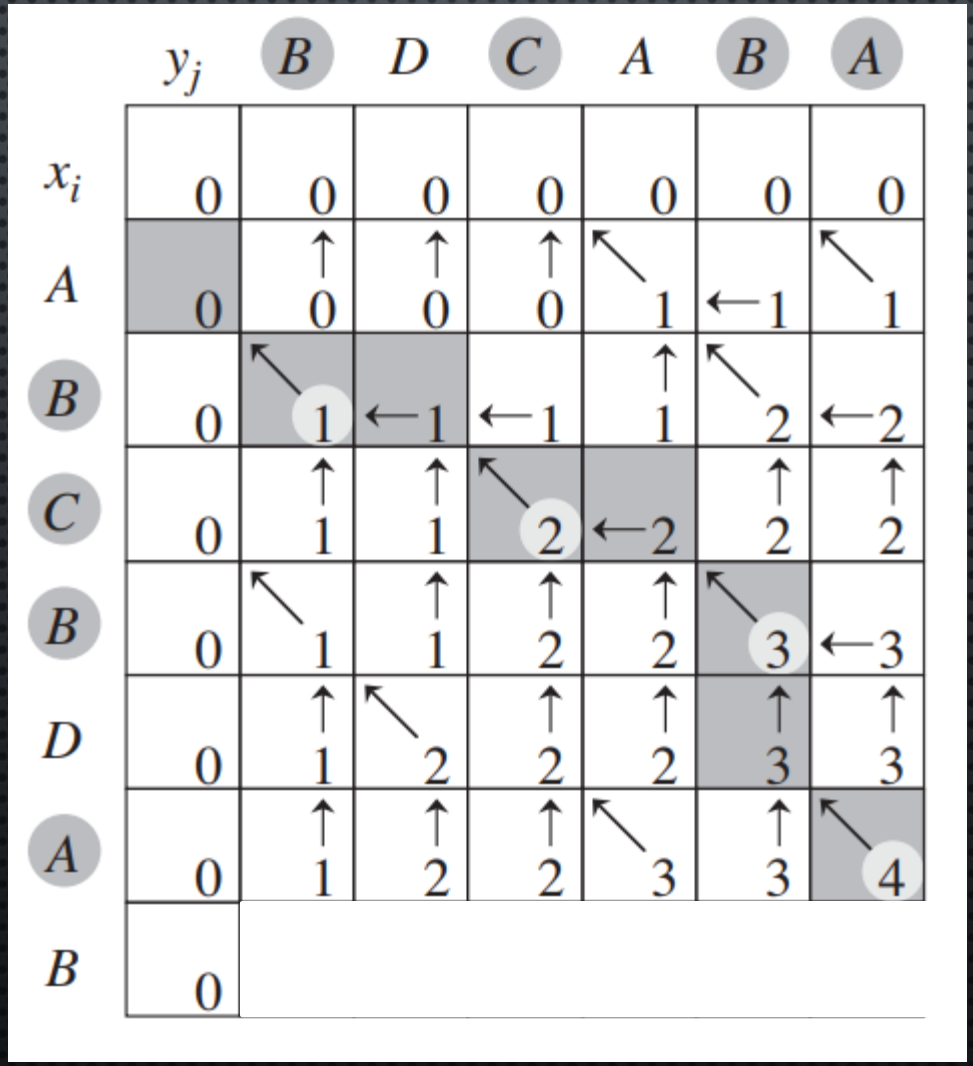
- An example, A = (abcdbab), B = (bdcaba)



| $x_i$ \ $y_j$ | | 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 | ↖ 3 | ← 3 |
| D | 0 | ↑ 1 | ↖ 2 | ↑ 2 | ↑ 2 | ↑ 3 | ↑ 3 |
| A | 0 | | | | | | |
| B | 0 | | | | | | |

# LONGEST COMMON SUBSEQUENCE

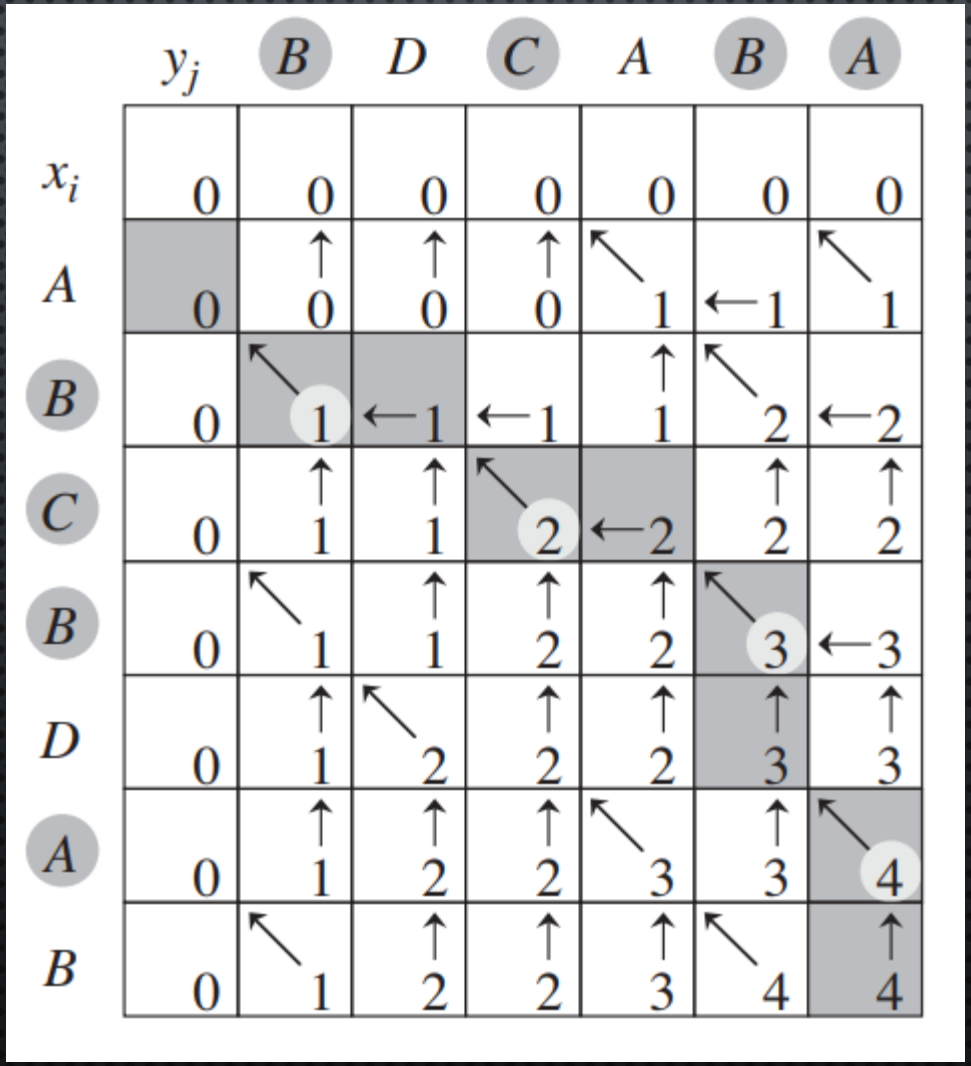
- An example, A = (abcdbab), B = (bdcaba)



| $x_i$ \ $y_j$ | | 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 | ↖ 3 | ← 3 |
| D | 0 | ↑ 1 | ↖ 2 | ↑ 2 | ↑ 2 | ↑ 3 | ↑ 3 |
| A | 0 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ↑ 3 | ↖ 4 |
| B | 0 | | | | | | |

# LONGEST COMMON SUBSEQUENCE

- An example, A = (abcdbab), B = (bdcaba)



| $x_i$ \ $y_j$ | | 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 | ↖ 3 | ← 3 |
| D | 0 | ↑ 1 | ↖ 2 | ↑ 2 | ↑ 2 | ↑ 3 | ↑ 3 |
| A | 0 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ↑ 3 | ↖ 4 |
| B | 0 | ↖ 1 | ↑ 2 | ↑ 2 | ↑ 3 | ↖ 4 | ↑ 4 |