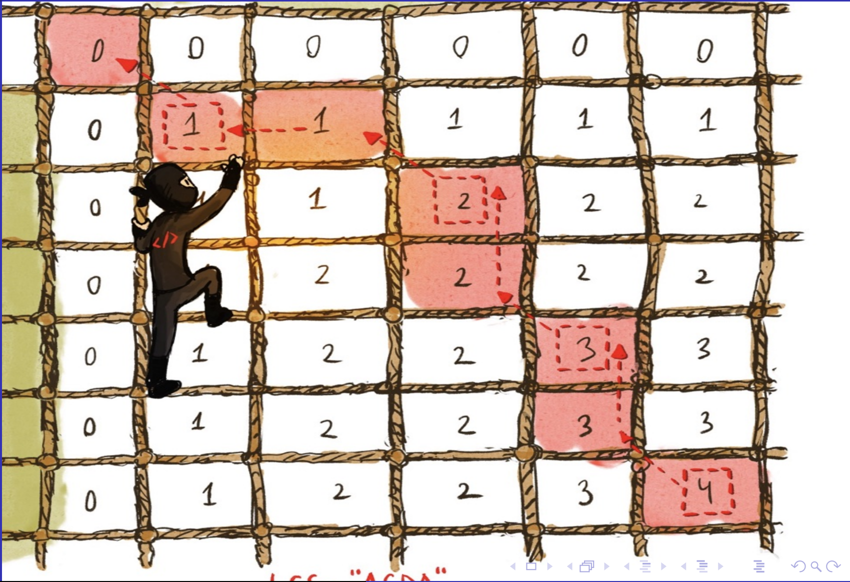
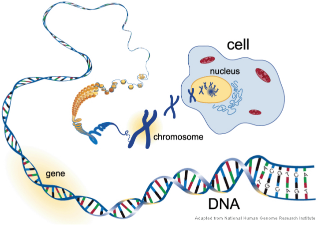
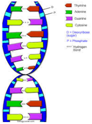
Dynamic Programming: additional examples



Matching DNA sequences



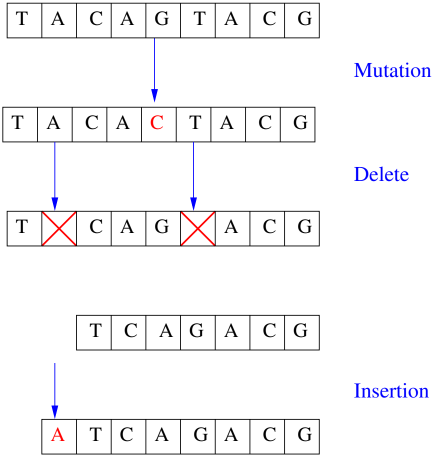


- DNA, is the hereditary material in almost all living organisms. They can reproduce by themselves.
- Its function is like a program unique to each individual organism that rules the working and evolution of the organism.
- Model as a string of 3×10^9 characters over $\{A, T, G, C\}$.

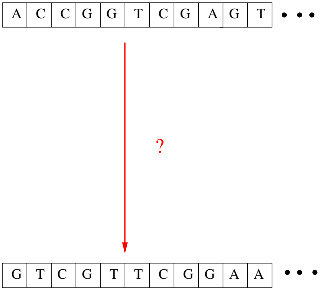
Computational genomics: Some questions

- When a new gene is discovered, one way to gain insight into its working, is to find well known genes (not necessarily in the same species) which match it closely. Biologists suggest a generalization of edit distance as a definition of approximately match.
- GenBank (`https://www.ncbi.nlm.nih.gov/genbank/`) has a collection of $> 10^{10}$ well studied genes, BLAST is a software to do fast searching for similarities between a genes a DB of genes.
- Sequencing DNA: consists in the determination of the order of DNA bases, in a short sequence of 500-700 characters of DNA. To get the global picture of the whole DNA chain, we generate a large amount of DNA sequences and try to assembled them into a coherent DNA sequence. This last part is usually a difficult one, as the position of each sequence is the global DNA chain is not know before hand.

Evolution DNA

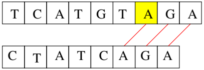


How to compare sequences

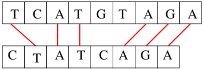


The basic problem

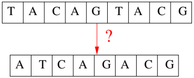
Longest common substring: Substring = consecutive characters in the string.



Longest common subsequence: Subsequence = ordered chain of characters (might have gaps).



Edit distance: Convert one string into another one using a given set of operations.



The Longest Common Subsequence

(Section 15.4 in CormenLRS' book.)

LCS Given sequences $X =< x_1 \cdots x_m >$ and $Y =< y_1 \cdots y_n >$, compute the longest common subsequence.

- $Z =< z_1 \cdots z_k >$ is a subsequence of X if there is a subsequence of integers $1 \leq i_1 < i_2 < \ldots < i_k \leq m$ such that $z_j = x_{i_j}$.

  TTT is a subsequence of $ATATAT$.
- If Z is a subsequence of X and Y, the Z is a common subsequence of X and Y.

DP approach: Characterization of optimal solution

Let $X =< x_1 \cdots x_n >$ and $Y =< y_1 \cdots y_m >$ and let Z be a longest common subsequence.

- $Z =< x_{i_1} \ldots x_{i_k} >=< y_{j_1} \ldots y_{j_k} >$
- There are no i, j, $i > i_k$ and $j > j_k$, s.t. $x_i = y_j$. If so, Z will not be optimal.
- $a = x_{i_k}$ might appear after i_k in X, but not after j_k in Y, or viceversa.
- There is an optimal solution in which i_k and j_k are the last occurrence of a in X and Y respectively.

DP approach: Characterization of optimal solution

Let $X =< x_1 \cdots x_n >$ and $Y =< y_1 \cdots y_m >$ and let $Z =< x_{i_1} \ldots x_{i_k} >=< y_{j_1} \ldots y_{j_k} >$ a lcs s.t. the index of the final common symbol in Z is its last occurrence in X, Y.

Let $X^- =< x_1 \cdots x_{n-1} >$ and $Y^- =< y_1 \cdots y_{m-1} >$

- Let us look at x_n and y_m.
- If $x_n = y_m$,
  $i_k = n$ and $j_k = m$ so, $< x_{i_1} \ldots x_{i_{k-1}} >$ is a lcs of X^- and Y^-.

DP approach: Characterization of optimal solution

Let $X =< x_1 \cdots x_n >$ and $Y =< y_1 \cdots y_m >$ and let $Z =< x_{i_1} \ldots x_{i_k} >=< y_{j_1} \ldots y_{j_k} >$ a lcs s.t. the index of the final common symbol in Z is its last occurrence in X, Y.

Let $X^- =< x_1 \cdots x_{n-1} >$ and $Y^- =< y_1 \cdots y_{m-1} >$

- Let us look at x_n and y_m.
- If $x_n \neq y_m$,
  - If $i_k < n$ and $j_k < m$, Z is a lcs of X^- and Y^-.
  - If $i_k = n$ and $j_k < m$, Z is a lcs of X and Y^-.
  - If $i_k <$ and $j_k = m$, Z is a lcs of X^- and Y.
  - Not that the last two include the first one.

DP approach: Supproblems

Subproblems = lcs of prefixes of the initial strings one string into other.

Notation:

- $X(i) = < x_1 \ldots x_i >$, for $0 \leq i \leq n$
- $Y(j) = < y_1 \ldots y_j >$, for $0 \leq j \leq m$
- $c[i, j]$ = length of the LCS of $X(i)$ and $Y(j)$.
- Want $c[n, m]$ i.e. length of the LCS for X and Y.

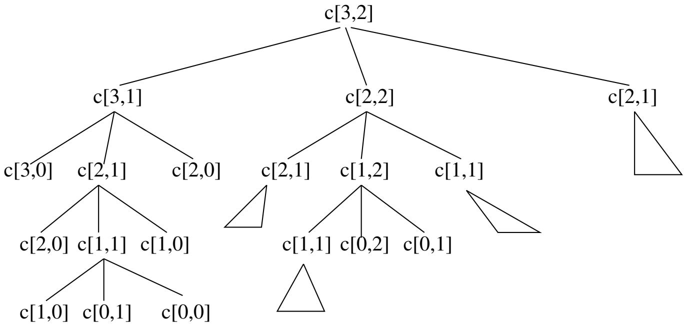
DP approach: Recursion

Therefore, given X and Y

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

Recursion tree

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$



The recursive algorithm

LCS(X, Y)
if $m = 0$ or $n = 0$ **then**
  return 0
else if $x_n = y_m$ **then**
  return $1+\textbf{LCS}(X^-, Y^-)$
else
  return $\max\{\textbf{LCS}(X, Y^-), \textbf{LCS}(X^-, Y)\}$

The algorithm makes 3 recursive calls and explores a tree of depth $O(n+m)$, therefore the time complexity is $3^{O(n+m)}$.

DP: tabulating

Avoid the exponential running time, by tabulating the subproblems and not repeating their computation, we need to find the correct traversal of the table holding the $c[i,j]$ values.

- Base case is $c[0,j] = 0$, for $0 \leq j \leq m$, and $c[i,0] = 0$, for $0 \leq i \leq n$.
- To compute $c[i,j]$, we have to access

$c[i-1,j-1]$	$c[i-1,j]$
$c[i,j-1]$	$c[i,j]$

  A row traversal provides a correct ordering.
- To being able to recover a solution we use a table b, to indicate which one of the three options provided the value $c[i,j]$.

Tabulating

LCS(X, Y)
for $i = 0$ **to** n **do**
  $c[i, 0] = 0$
for $j = 1$ **to** m **do**
  $c[0, j] = 0$
for $i = 1$ **to** n **do**
  for $j = 1$ **to** m **do**
    if $x_i = y_j$ **then**
      $c[i, j] = c[i-1, j-1] + 1, \; b[i.j] = \nwarrow$
    else if $c[i-1, j] \geq c[i, j-1]$ **then**
      $c[i, j] = c[i-1, j], \; b[i, j] = \leftarrow$
    else
      $c[i, j] = c[i, j-1], \; b[i, j] = \uparrow.$

complexity:
$T = O(nm)$.

Example.

X=(ATCTGAT); Y=(TGCATA). Therefore, $m = 6, n = 7$

		0	1 T	2 G	3 C	4 A	5 T	6 A
0		0	0	0	0	0	0	0
1	A	0	$\uparrow 0$	$\uparrow 0$	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\nwarrow 1$
2	T	0	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$
3	C	0	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\uparrow 2$	$\uparrow 2$
4	T	0	$\nwarrow 1$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\leftarrow 3$
5	G	0	$\uparrow 1$	$\nwarrow 2$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\uparrow 3$
6	A	0	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 3$	$\nwarrow 4$
7	T	0	$\nwarrow 1$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	4	$\uparrow 4$

Following the arrows: TCTA

Construct the solution

Access the tables c and d.

The first call to the algorithm is **sol-LCS**(n, m)

sol-LCS(i, j)
if $i = 0$ or $j = 0$ **then**
  STOP.
else if $b[i, j] = \nwarrow$ **then**
  sol-LCS$(i - 1, j - 1)$
  return x_i
else if $b[i, j] = \uparrow$ **then**
  sol-LCS$(i - 1, j)$
else
  sol-LCS$(i, j - 1)$

The algorithm has time complexity $O(n + m)$.

Longest common substring

- A slightly different problem with a similar solution
- LCSt Given two strings $X =< x_1 \dots x_m >$ and $Y =< y_1 \dots y_n >$, compute their longest common substring Z, i.e., corresponding to the largest k for which there are indices i and j with $x_i x_{i+1} \dots x_{i+k} = y_j y_{j+1} \dots y_{j+k}$.
- For example:
  X : DEADBEEF
  Y : EATBEEF
  Z :

Longest common substring

- A slightly different problem with a similar solution
- LCSt Given two strings $X =< x_1 \ldots x_m >$ and $Y =< y_1 \ldots y_n >$, compute their longest common substring Z, i.e., corresponding to the largest k for which there are indices i and j with $x_i x_{i+1} \ldots x_{i+k} = y_j y_{j+1} \ldots y_{j+k}$.
- For example:
  X : DEADBBEEF
  Y : EATBEEF
  Z :

Longest common substring

- A slightly different problem with a similar solution
- LCSt Given two strings $X =< x_1 \dots x_m >$ and $Y =< y_1 \dots y_n >$, compute their longest common substring Z, i.e., corresponding to the largest k for which there are indices i and j with $x_i x_{i+1} \dots x_{i+k} = y_j y_{j+1} \dots y_{j+k}$.
- For example:
  X : DEADBBEEF
  Y : EATBEEF
  Z : BEEF pick the longest substring

Characterization of optimal solution

- Let $X =< x_1 \cdots x_n >$ and $Y =< y_1 \cdots y_m >$ and let Z be a longest common substring.
  - $Z =< x_i \ldots x_{i+k} >=< y_j \ldots y_{j+k} >$

Characterization of optimal solution

- Let $X =< x_1 \cdots x_n >$ and $Y =< y_1 \cdots y_m >$ and let Z be a longest common substring.
  - $Z =< x_i \ldots x_{i+k} >=< y_j \ldots y_{j+k} >$
  - Z is the longest common suffix of $X(i+k)$ and $Y(j+k)$.

Characterization of optimal solution

- Let $X =< x_1 \cdots x_n >$ and $Y =< y_1 \cdots y_m >$ and let Z be a longest common substring.
  - $Z =< x_i \ldots x_{i+k} >=< y_j \ldots y_{j+k} >$
  - Z is the longest common suffix of $X(i+k)$ and $Y(j+k)$.
- We can consider the subproblems $LCSf(i,j)$: compute the longest common suffix of $X(i)$ and $Y(j)$.

Characterization of optimal solution

- Let $X =< x_1 \cdots x_n >$ and $Y =< y_1 \cdots y_m >$ and let Z be a longest common substring.
  - $Z =< x_i \ldots x_{i+k} >=< y_j \ldots y_{j+k} >$
  - Z is the longest common suffix of $X(i+k)$ and $Y(j+k)$.
- We can consider the subproblems $LCSf(i,j)$: compute the longest common suffix of $X(i)$ and $Y(j)$.
- The $LCSt(X, Y)$ is the longest of such common suffixes.

Computing the LC Suffixes

- To solve $LCSf(i, j)$ it is enough to go backward from position i in X and j in Y until we find two different characters.
- This has cost $O(n + m)$ per subproblem.

Computing the LC Suffixes

- To solve $LCSf(i, j)$ it is enough to go backward from position i in X and j in Y until we find two different characters.
- This has cost $O(n + m)$ per subproblem.
- We get a $O(nm(n + m))$ algorithm for LCSt

Computing the LC Suffixes

- To solve $LCSf(i,j)$ it is enough to go backward from position i in X and j in Y until we find two different characters.
- This has cost $O(n+m)$ per subproblem.
- We get a $O(nm(n+m))$ algorithm for LCSt
- Can we do it faster?

Computing the LC Suffixes

- To solve $LCSf(i, j)$ it is enough to go backward from position i in X and j in Y until we find two different characters.
- This has cost $O(n + m)$ per subproblem.
- We get a $O(nm(n + m))$ algorithm for LCSt
- Can we do it faster? Let us use DP!

A recursive solution for LC Suffixes

Notation:

- $X(i) =< x_1 \ldots x_i >$, for $0 \leq i \leq n$
- $Y(j) =< y_1 \ldots y_j >$, for $0 \leq j \leq m$
- $s[i,j]$ = the length of the LC Suffix of $X(i)$ and $Y(j)$.
- Want $\max_{i,j} s[i,j]$ i.e., the length of the LCSt of X, Y.

DP approach: Recursion

Therefore, given X and Y

$$s[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 0 & \text{if } x_i \neq y_j \\ s[i-1, j-1] + 1 & \text{if } x_i = y_j \end{cases}$$

DP approach: Recursion

Therefore, given X and Y

$$s[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 0 & \text{if } x_i \neq y_j \\ s[i-1,j-1]+1 & \text{if } x_i = y_j \end{cases}$$

Using the recurrence the cost per recursive call (or per element in the table) is constant

The recursive algorithm

LCSf(X, Y)
if $m = 0$ or $n = 0$ **then**
  return 0
else if $x_n = y_m$ **then**
  return 1+**LCSf**(X^-, Y^-)
else
  return 0

The algorithm makes 1 recursive calls and explores a tree of depth $O(n + m)$, therefore the time complexity is $O(nm(n + m))$.

Tabulating

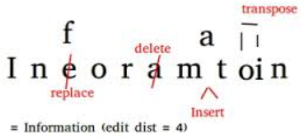
LCSf(X, Y)
for $i = 0$ **to** n **do**
    $s[i, 0] = 0$
for $j = 1$ **to** m **do**
    $s[0, j] = 0$
for $i = 1$ **to** n **do**
    for $j = 1$ **to** m **do**
        s[i,j]=0
        if $x_i = y_j$ **then**
            $s[i, j] = s[i-1, j-1] + 1$

complexity: $O(nm)$.

Which gives an algorithm with cost $O(nm)$ for LCSt

The EDIT DISTANCE problem

(Section 6.3 in Dasgupta, Papadimritriou, Vazirani's book.)



The edit distance between strings $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$ is defined to be the minimum number of *edit operations* needed to transform X into Y.

All the operations are done on X

Edit distance: Applications

- Computational genomics: evolution between generations, i.e. between strings on $\{A, T, G, C, -\}$.
- Natural Language Processing: distance, between strings on the alphabet.
- Text processor, suggested corrections

EDIT DISTANCE: Levenshtein distance

In the Levenshtein distance the set of operations are

- insert(X, i, a)= $x_1 \cdots x_i a x_{i+1} \cdots x_n$.
- delete(X, i)= $x_1 \cdots x_{i-1} x_{i+1} \cdots x_n$
- modify(X, i, a)= $x_1 \cdots x_{i-1} a x_{i+1} \cdots x_n$.

the cost of modify is 2, and the cost of insert/delete is 1.

To simplify, in the following we assume that *the cost of each operation is 1.*

For other operations and costs the structure of the DP will be similar.

Exemple-1

$X = aabab$ and $Y = babb$

$aabab = X$

$X' = \text{insert}(X, 0, b)$ $\quad baabab$

$X'' = \text{delete}(X', 2)$ $\quad babab$

$X'' = \text{delete}(X'', 4)$ $\quad babb$

$X = aabab \to Y = babb$

Exemple-1

$X = aabab$ and $Y = babb$
$aabab = X$
$X' =$insert$(X, 0, b)$ $baabab$
$X'' =$delete$(X', 2)$ $babab$
$X'' =$delete$(X'', 4)$ $babb$
$X = aabab \rightarrow Y = babb$

A shortest edit distance

$aabab = X$
$X' =$modify$(X, 1, b)$ $babab$
$Y =$delete$(X', 4)$ $babb$

Use dynamic programming.

The structure of an optimal solution

- As for the LCS, we look at what happens to the final symbol in X in an optimal solution to the problem.
- A solution is a sequence of operations on X.

The structure of an optimal solution

- In a solution O with minimum edit distance from $X = x_1 \cdots x_n$ to $Y = y_1 \cdots y_m$, we have three possible alignments for the last terms

(1)	(2)	(3)
x_n	$-$	x_n
$-$	y_m	y_m

- In (1), O performs delete x_n, and it transforms optimally, $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_m$.
- In (2), O performs insert y_m at the end x, and it transforms optimally, $x_1 \cdots x_n$ into $y_1 \cdots y_{m-1}$.
- In (3), if $x_n \neq y_m$, O performs modify x_n by y_m, otherwise O, aligns them without cost, and then it transforms optimally $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_{m-1}$.

The recurrence

Let $X[i] = x_1 \cdots x_i$, $Y[j] = y_1 \cdots y_j$.
$E[i,j] =$ edit distance from $X[i]$ to $Y[j]$ is the maximum of

- I *put* y_j *at the end* x: $E[i, j-1] + 1$
- D *delete* x_i: $E[i-1, j] + 1$
- if $x_i \neq y_j$, M *change* x_i *into* y_j: $E[i-1, j-1] + 1$, otherwise $E[i-1, j-1]$

Edit distance: Recurrence

Adding the base cases, we have the recurrence

$$E[i,j] = \begin{cases} i & \text{if } j = 0 \text{ (converting } \lambda \to y[j]) \\ j & \text{if } i = 0 \text{ (converting } X[i] \to \lambda) \\ \min \begin{cases} E[i-1,j]+1 & \text{if D} \\ E[i,j-1]+1, & \text{if I} \\ E[i-1,j-1]+\delta(x_i,y_j) & \text{otherwise} \end{cases} \end{cases}$$

where

$$\delta(x_i,y_j) = \begin{cases} 0 & \text{if } x_i = y_j \\ 1 & \text{otherwise} \end{cases}$$

Computing the optimal costs.

Edit(X, Y)
for $i = 0$ **to** n **do**
  $E[i, 0] = i$
for $j = 0$ **to** m **do**
  $E[0, j] = j$
for $i = 1$ **to** n **do**
  for $j = 1$ **to** m **do**
    $\delta = 0$
    if $x_i \neq y_j$ **then**
      $\delta = 1$
    $E[i, j] = E[i, j-1] + 1 \; b[i, j] = \uparrow$
    if $E[i-1, j-1] + \delta < E[i, j]$ **then**
      $E[i, j] = E[i-1, j-1] + \delta, \; b[i, j] := \nwarrow$
    if $E[i-1, j] + 1 < E[i, j]$ **then**
      $E[i, j] = E[i-1, j] + 1, \; b[i, j] := \leftarrow$

Space and time complexity: $O(nm)$.

$\leftarrow$ is a I operation, $\uparrow$ is a D operation, and $\nwarrow$ is either a M or a no-operation.

Computing the optimal costs: Example

X=aabab; Y=babb. Therefore, $n = 5, m = 4$

		0 λ	1 b	2 a	3 b	4 b
0	λ	0	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$
1	a	$\uparrow 1$	$\nwarrow 1$	$\nwarrow 1$	$\leftarrow 2$	$\leftarrow 3$
2	a	$\uparrow 2$	$\nwarrow 2$	$\nwarrow 1$	$\leftarrow 2$	$\leftarrow 3$
3	b	$\uparrow 3$	$\nwarrow 2$	$\uparrow 2$	$\nwarrow 1$	$\nwarrow 2$
4	a	$\uparrow 4$	$\uparrow 3$	$\nwarrow 2$	$\uparrow 2$	$\nwarrow 2$
5	b	$\uparrow 5$	$\nwarrow 4$	$\uparrow 3$	$\uparrow 2$	$\nwarrow 2$

$\leftarrow$ is a I operation, $\uparrow$ is a D operation, and
$\nwarrow$ is either a M or a no-operation.

Obtain Y in edit distance from X

Uses as input the arrays E and b.

The first call to the algorithm is **con-Edit** (n, m)

con-Edit(i, j)
if $i = 0$ or $j = 0$ **then**
  return IF $b[i, j] = \nwarrow$ and $x_i = y_j$
  change(X, i, y_j); **con-Edit**$(i - 1, j - 1)$
if $b[i, j] = \uparrow$ **then**
  delete(X, i); **con-Edit**$(i - 1, j)$
if $b[i, j] = \leftarrow$ **then**
  insert(X, i, y_j), **con-Edit**$(i, j - 1)$

This algorithm has time complexity $O(nm)$.