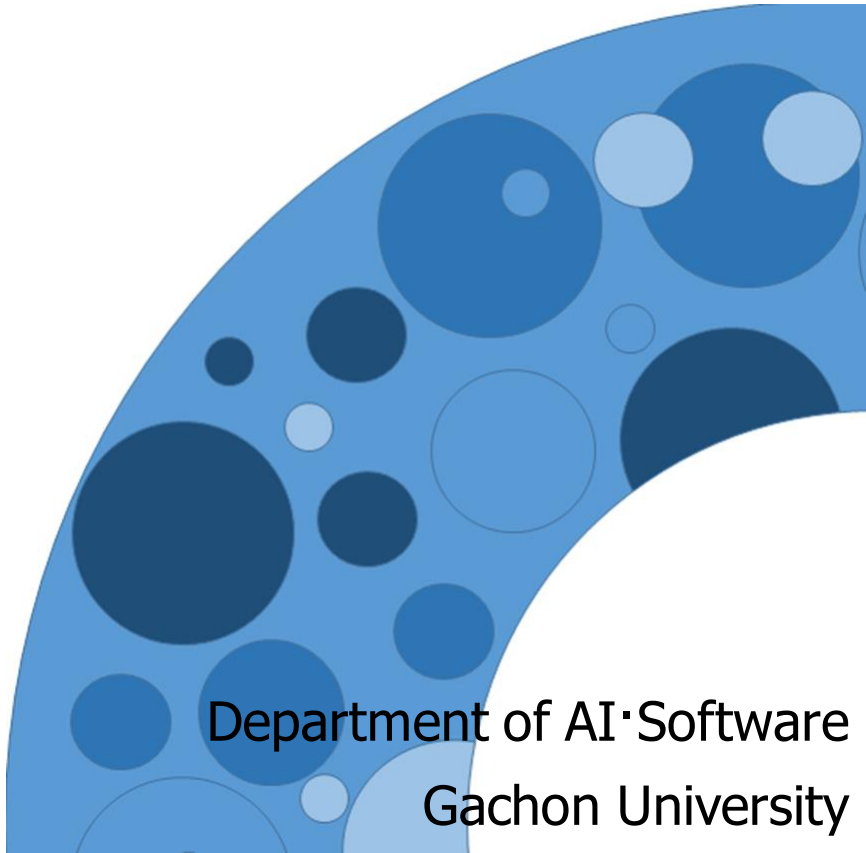
Algorithms

Kiho Choi

Fall, 2022



Department of AI·Software
Gachon University

3. Dynamic Programming II

Contents

The Edit Distance

Example : “algorithm” →”alligator”

What is the cheapest way to transform one word (the source) into another word(the output)?

Source: a l g o r i t h m
Output:

Operations: (none)

Example : “algorithm” →”alligator”

What is the cheapest way to transform one word (the source) into another word(the output)?

Source: a l g o r i t h m
Output:

Operations: (none)

Example : “algorithm” →”alligator”

Source: a l g o r i t h m

Output:

Operations: (none)

Example : “algorithm” →”alligator”

Source: a l g o r i t h m

Output: a

Operations: (none) Copy

Example : “algorithm” →”alligator”

Operations: (none) Copy, Copy, **Insert(l)**

Example : “algorithm” →”alligator”

Operations: Copy, Copy, Insert(l), Insert(i), Copy
Delete, Delete, Delete, Insert(a), Copy, Delete,
Delete, **Insert(o), Insert(r)**

The Edit Distance problem

- Problem: what is the cheapest way to transform one word (the **source**) into another word (the **output**)?
- At any point, you can:
  - **Delete** the current character of the source.
  - **Insert** a new character into the output word.
  - **Copy** the current character of the source into the output.
- **Each operation has a cost associated with it.**
- **The cost of a transformation is the sum of the costs of each operation in the sequence.**

Example : “algorithm” →”alligator”

- Operations: Copy, Copy, Insert(l), Insert(i), Copy
  Delete, Delete, Delete, Insert(a), Copy, Delete,
  Delete, Insert(o), Insert(r)
- Assume that:
  - Copying costs 3
  - Inserting costs 5
  - Deleting costs 2
- The cost of the above transformation is: 47
  - This is just 3 + 3 + 5 + 5 + 3 + 2 + 2 + …

The Edit Distance problem

- Problem: what is the cheapest way to transform one word (the **source**) into another word (the **output**)?
- At any point, you can:
  - **Delete** the current character of the source.
  - **Insert** a new character into the output word.
  - **Copy** the current character of the source into the output.
- **By "cheapest", we mean the transformation with the least cost.**

How to solve this problem?

- Will trying out all possible transformations work?
- Answer: It can, but it would take **way** too long.

Will dynamic programming work?

- Depends on our problem.
- Can our problem be solved based on the solution of some subproblems?
- What are the subproblems, if there any?
- **As is typical when trying out dynamic programming, it helps to find some "key observation" or insight.**

Will dynamic programming work?

- Depends on our problem.
- Can our problem be solved based on the solution of some subproblems?
- What are the subproblems, if there any?
- **As is typical when trying out dynamic programming, it helps to find some "key observation" or insight.**
  - We will need three observations for this problem.

Notation

- (s1, s2, s3, …, sn) stands for a sequence of n elements.
- Used for a list of operations.
  - Example: (Copy, Copy, Insert(c), Delete).
- Also used for strings.
  - Example: “help” would correspond to (h, e, l ,p).

Key Observation #1

- Given: Strings **X = (x1, ..., xn)** and **Y = (y1, ..., yt)**, and a sequence **S** of operations **(s1, s2, ..., sm)**.

- Let **S' = (s1, ..., s(m-1))**, **X' = (x1, ..., x(n-1))**, and **Y' = (y1, ..., y(t-1))**.

- Suppose that **S** is the cheapest sequence of operations to transform **X** into **Y**, and that **sm** is a Copy operation.

Key Observation #2

- Given: Strings **X = (x1, ..., xn)** and **Y = (y1, ..., yt)**, and a sequence **S** of operations **(s1, s2, ..., sm)**.
- Let **S' = (s1, ..., s(m-1))** and **X' = (x1, ..., x(n-1))**.
- Suppose that **S** is the cheapest sequence of operations to transform **X** into **Y**, and that **sm** is a Delete operation.

Key Observation #3

- Given: Strings **X = (x1, ..., xn)** and **Y = (y1, ..., yt)**, and a sequence **S** of operations **(s1, s2, ..., sm)**.
- Let **S' = (s1, ..., s(m-1))** and **Y' = (y1, ..., y(t-1))**.
- Suppose that **S** is the cheapest sequence of operations to transform **X** into **Y**, and that **sm** is an Insert(yn) operation.

Key Observation #3 (cont.)

- Claim: then, **S'** is the cheapest sequence of operations to transform **X** to **Y'**.
- Proof: by contradiction. Suppose that **T = (t1, ..., tk)** is cheaper than **S'** and tranforms **X** to **Y'**.
- Then **(t1, ..., tk, sm)** is cheaper than **S** and transforms **X** to **Y**. This is a contradiction, as **S** was supposed to be the cheapest way to transform **X** to **Y**.

An observation about the observations…

- Given **X**, **Y**, and **S**, we have covered all possible cases for **sm** (the last operation).

Putting the observations to use

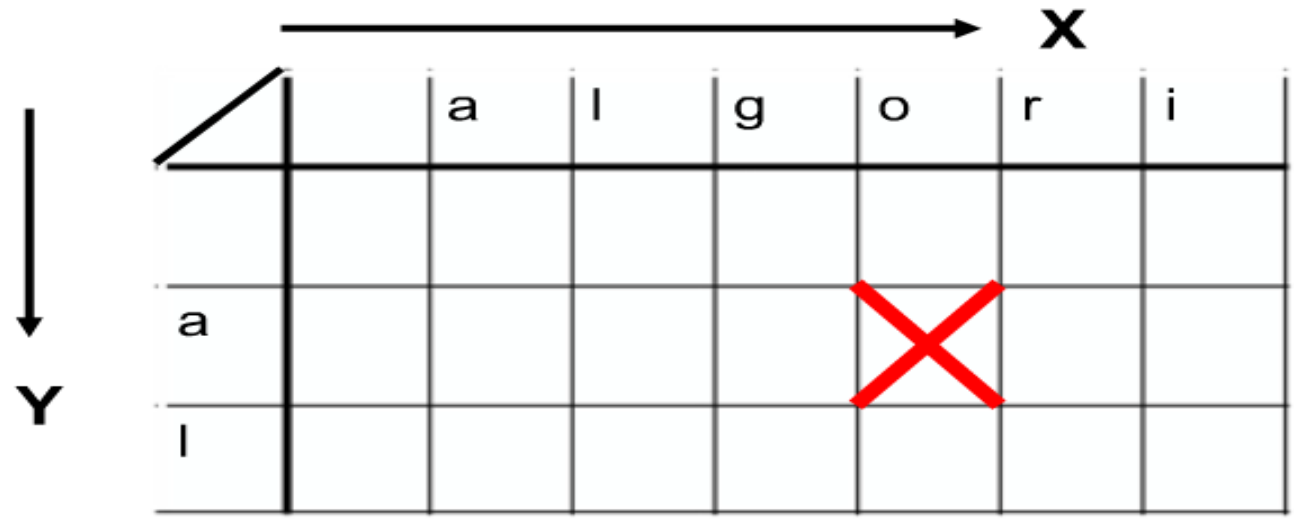
- The best sequence of operations to transform **X** to **Y** depended on one of the following:
  - (1) The best way to transform **X'** to **Y'**.
  - (2) The best way to transform **X** to **Y'**.
  - (3) The best way to transform **X'** to **Y**.
- The three cases become the subproblems to consider.
  - Given the solution to all three, we can find the solution to our actual problem (transforming **X** to **Y**).
  - Why? The best solution to transforming **X** to **Y** must contain a solution to one of the three cases by the three observations.

How to organize the subproblems

- Each subproblem is characterized by some initial part of the original strings **X** and **Y**.
- So use a matrix.

How to organize the subproblems (cont.)

- The marked location contains the score for the cheapest way to transform “algo” to “a”.

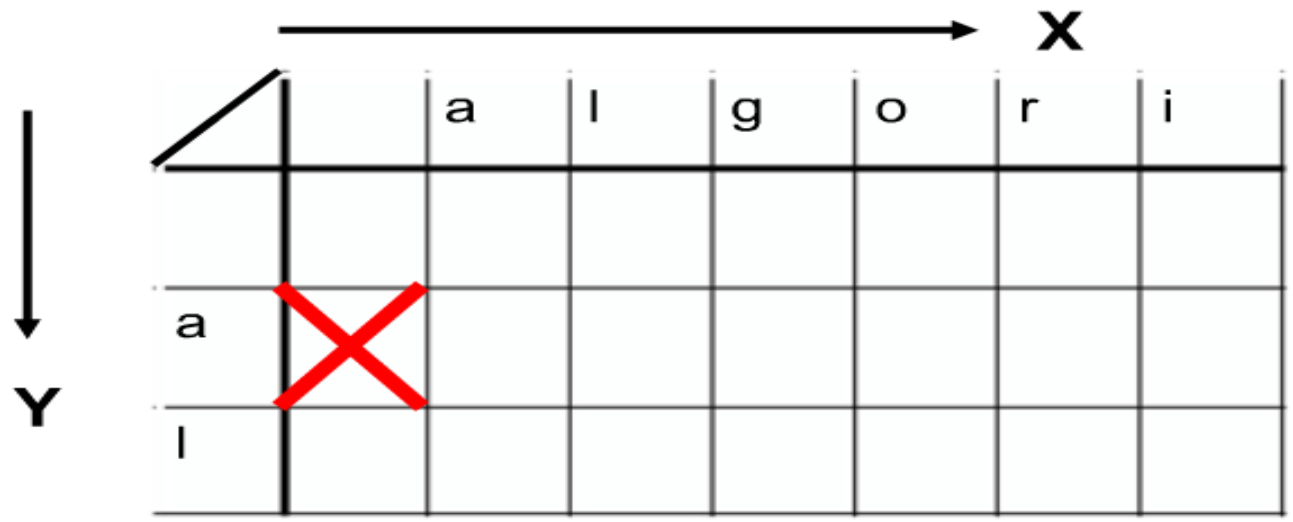


How to organize the subproblems (cont.)

- The marked location contains the score for the cheapest way to transform “alg” to “”.

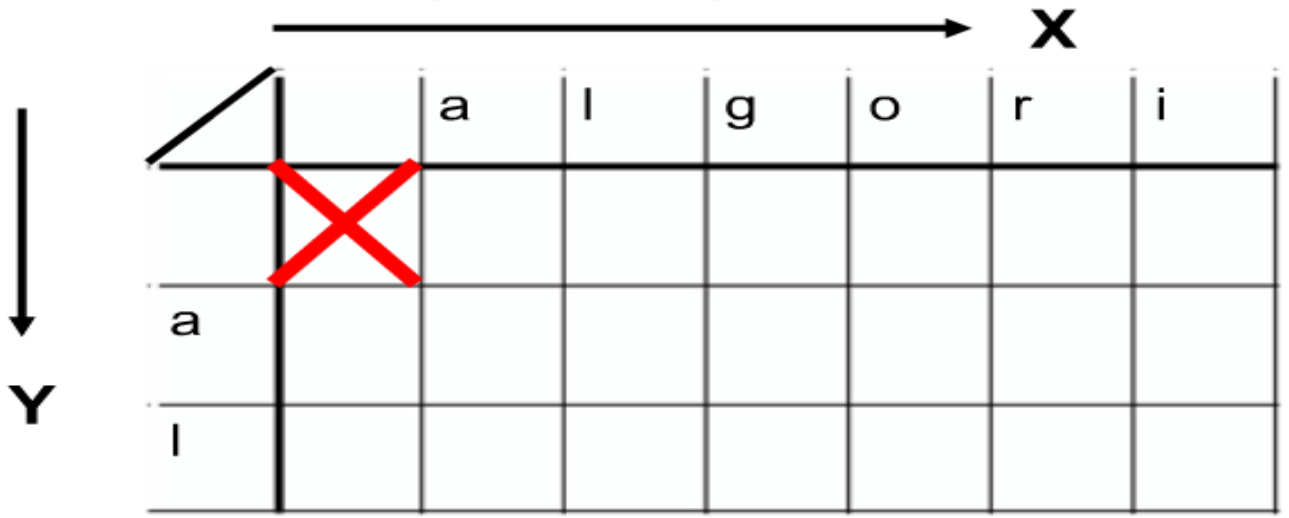
How to organize the subproblems (cont.)

- The marked location contains the score for the cheapest way to transform “” to “a”.



How to organize the subproblems (cont.)

- The marked location contains the score for the cheapest way to transform “” to “”.
- This is the smallest problem possible.



Additional information to store

- Each location should store the **last** operation in the cheapest sequence of operations used to get there.

Additional information to store (cont.)

- Example: the marked location would contain a score, and possibly a Copy operation.

How to fill out the matrix

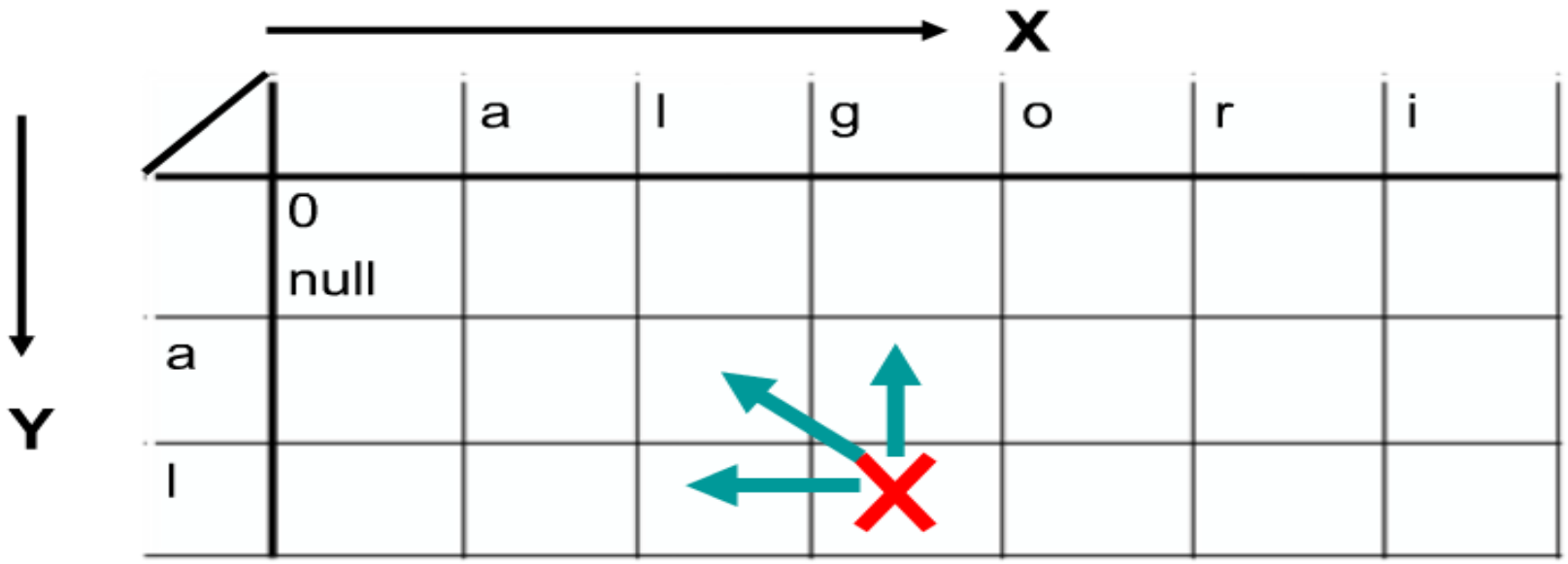
- For concreteness, assume that Copy costs 5, Insert(c) costs 10, and Delete costs 10.

How to fill out the matrix (cont.)

- Start at the upper left.
- This is trivial: score is zero, operation is null.

How to fill out the matrix (cont.)

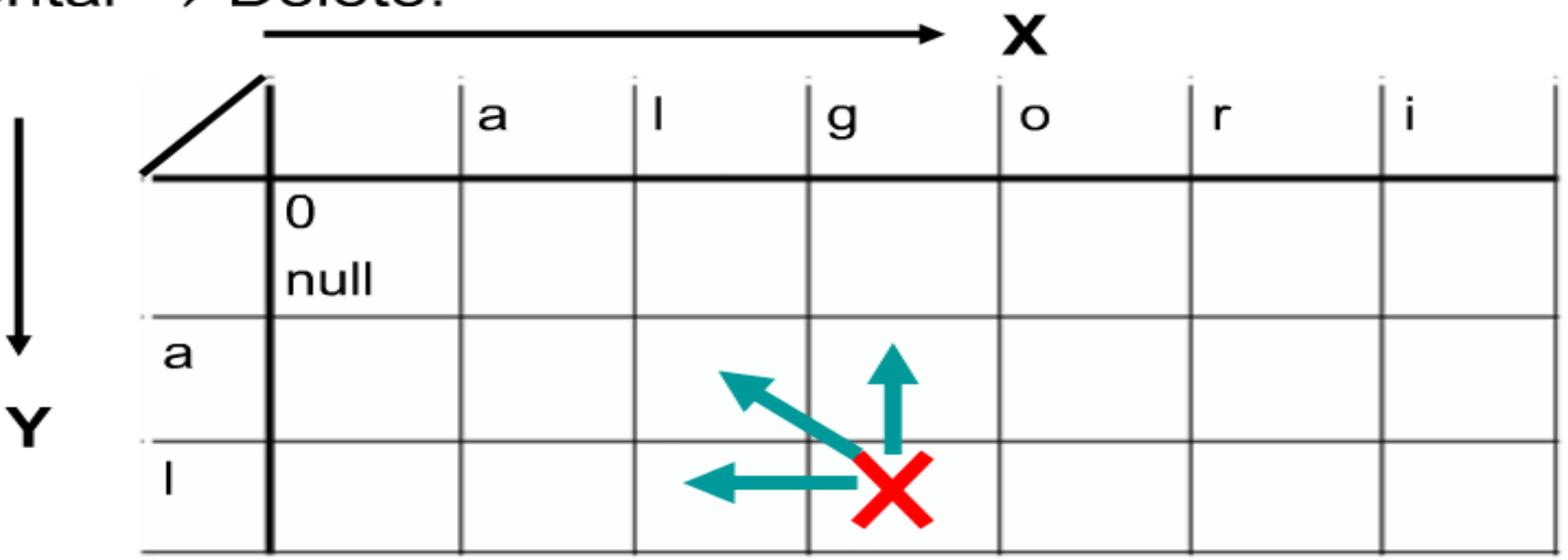
- All other locations depend on the values in up to three other places.



How to fill out the matrix (cont.)

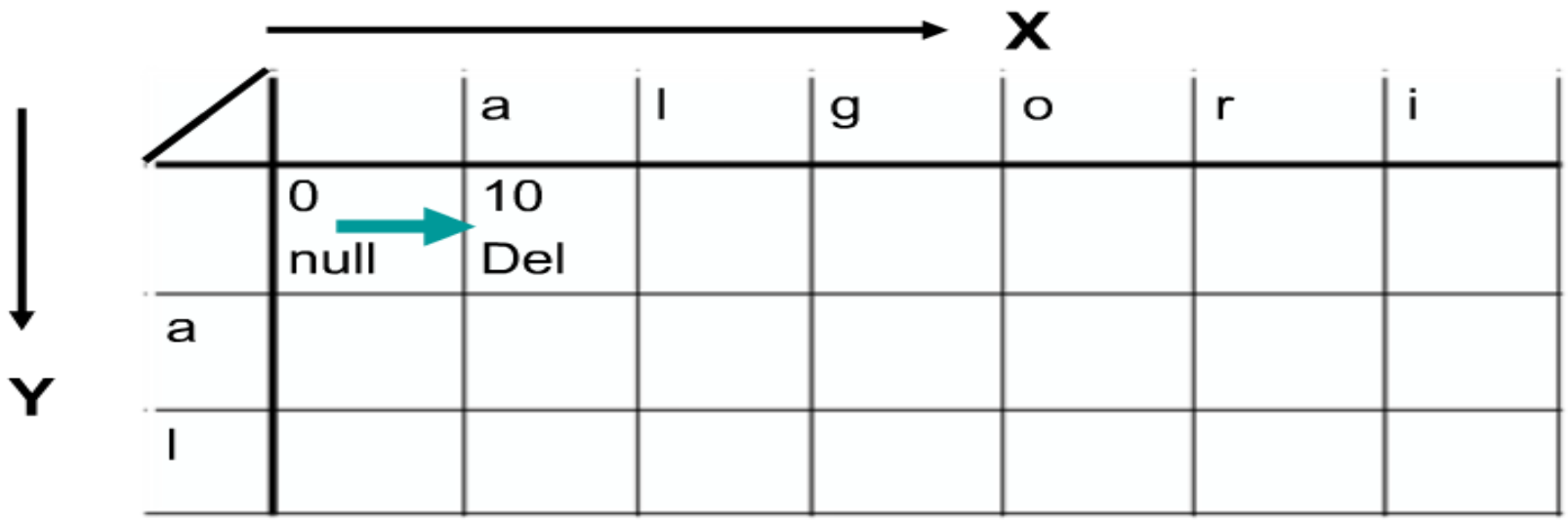
- Diagonal movement corresponds to a Copy operation.
- Vertical → Insert(c).
- Horizontal → Delete.

가로의



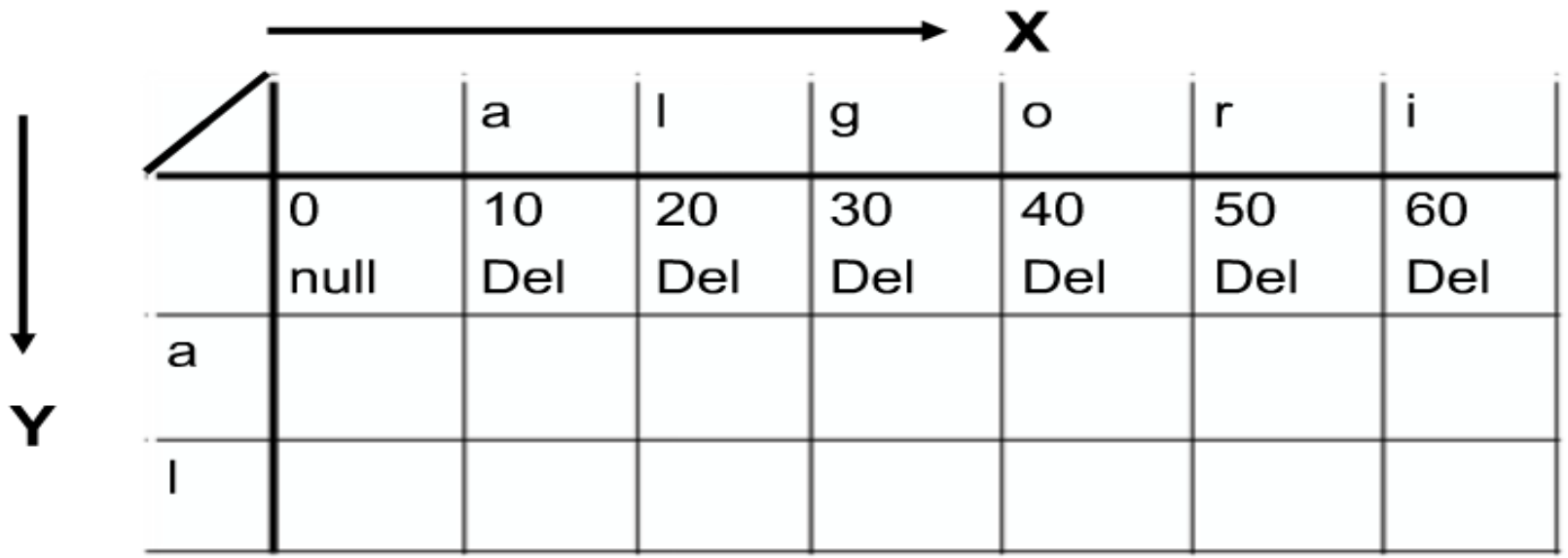
How to fill out the matrix (cont.)

- Fill out each row in turn.
- Only one option for the first row…



How to fill out the matrix (cont.)

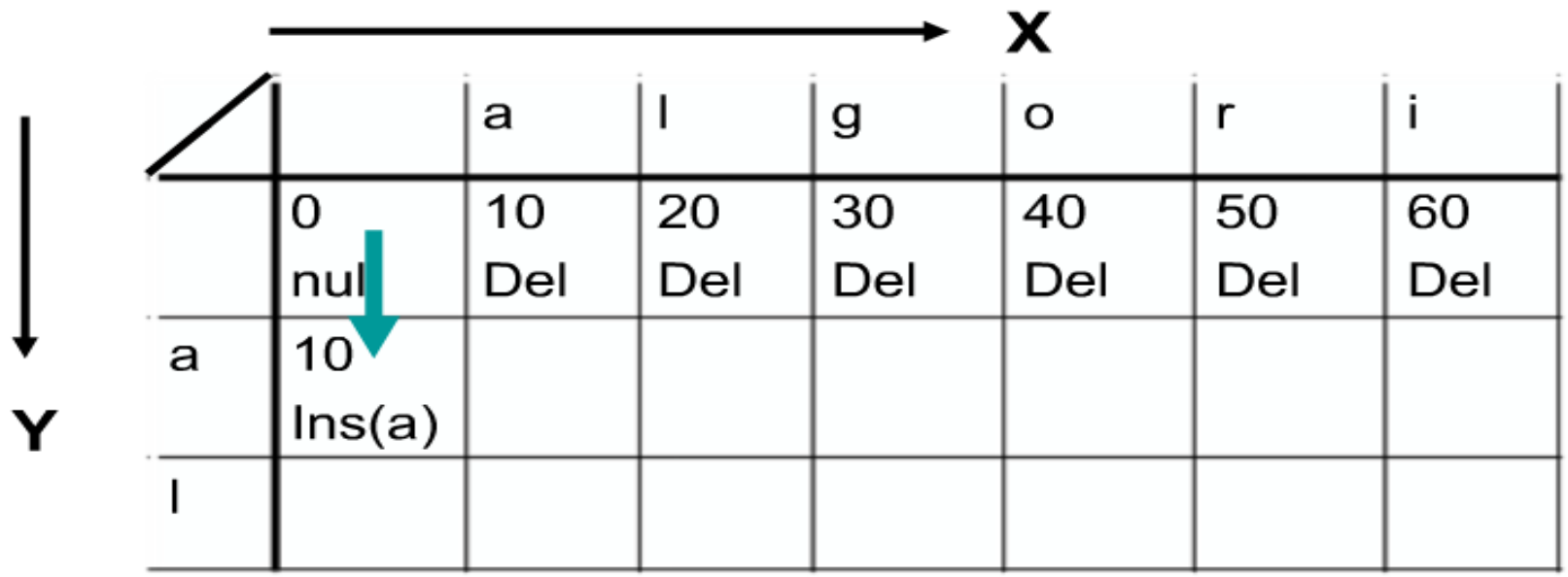
- Fill out each row in turn.
- Only one option for the first row…



		a	l	g	o	r	i
	0 null	10 Del	20 Del	30 Del	40 Del	50 Del	60 Del
a							
l							

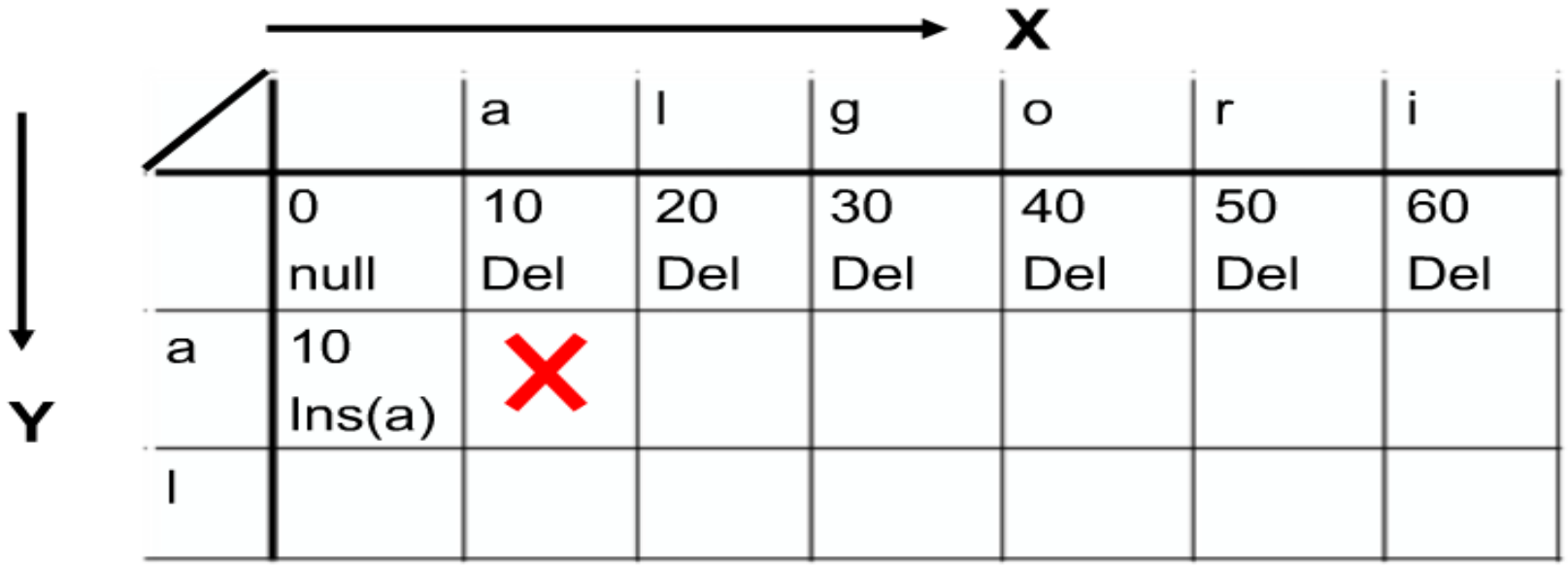
How to fill out the matrix (cont.)

- Fill out each row in turn.



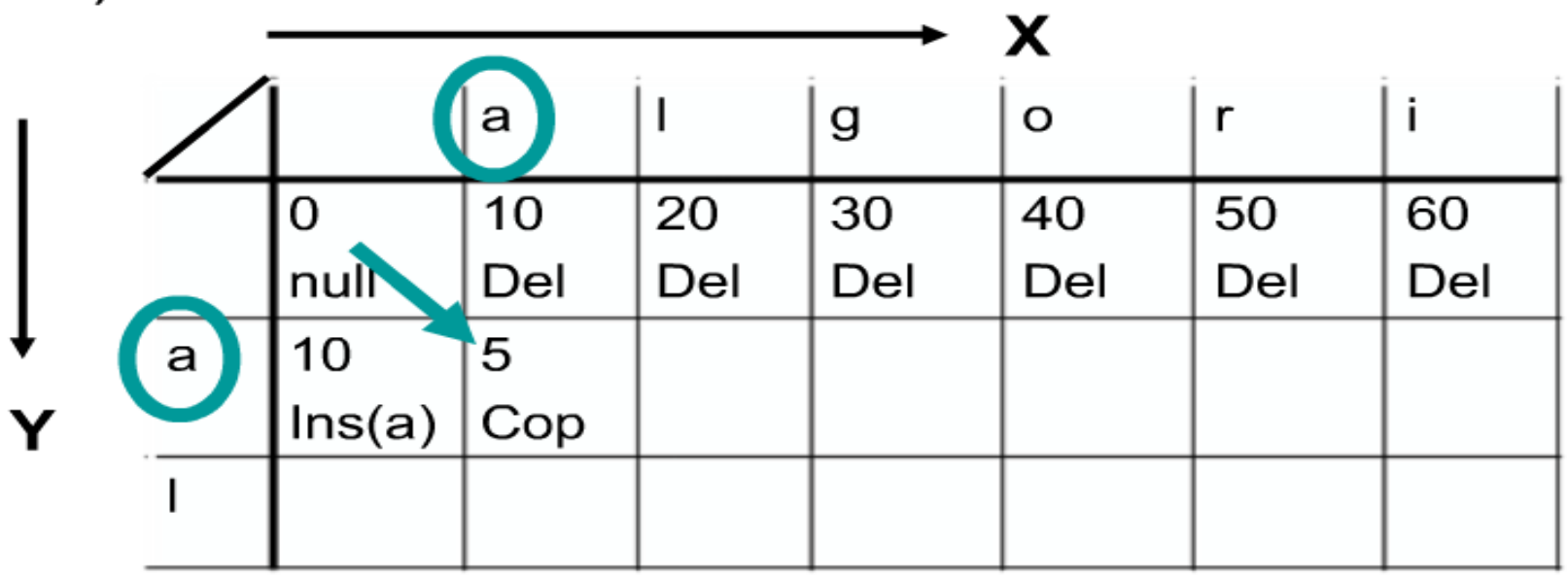
How to fill out the matrix (cont.)

- Three possibilities to consider for the marked square.
- Clear that Copy is cheapest. (5 versus 20 for Insert(a) or Delete).



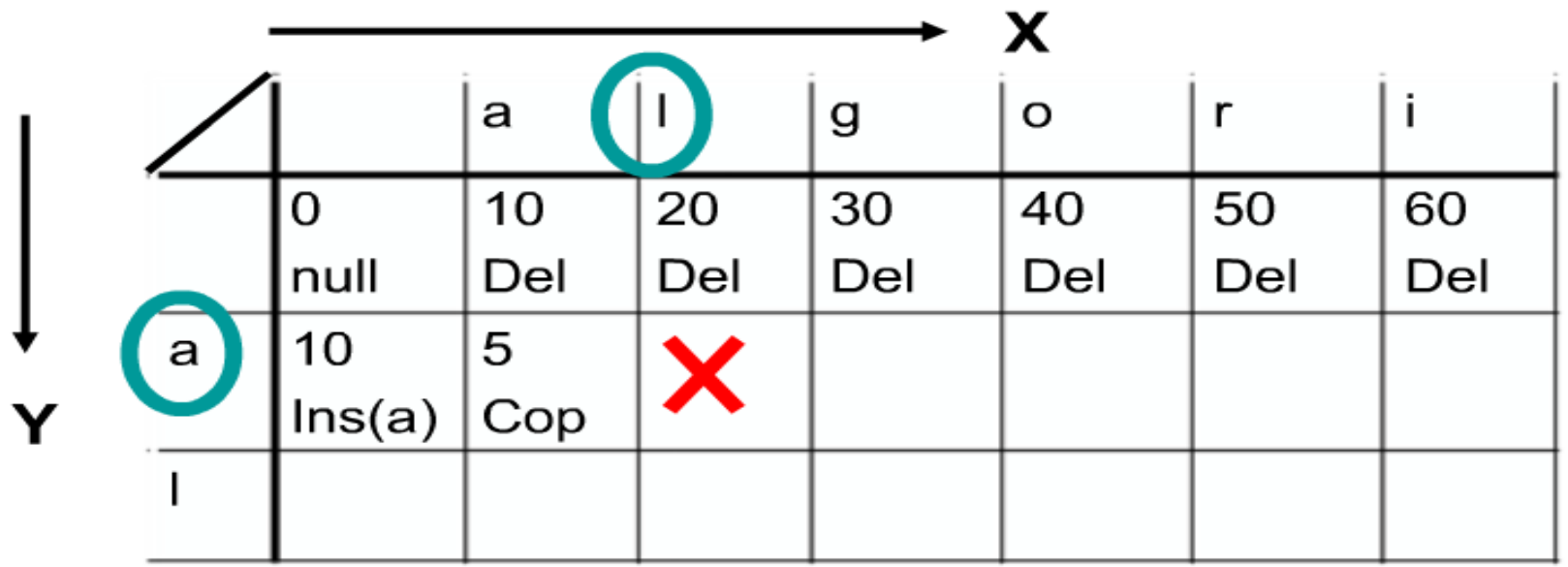
How to fill out the matrix (cont.)

- Three possibilities to consider for the marked square.
- Clear that Copy is cheapest. (5 versus 20 for Insert(a) or Delete).



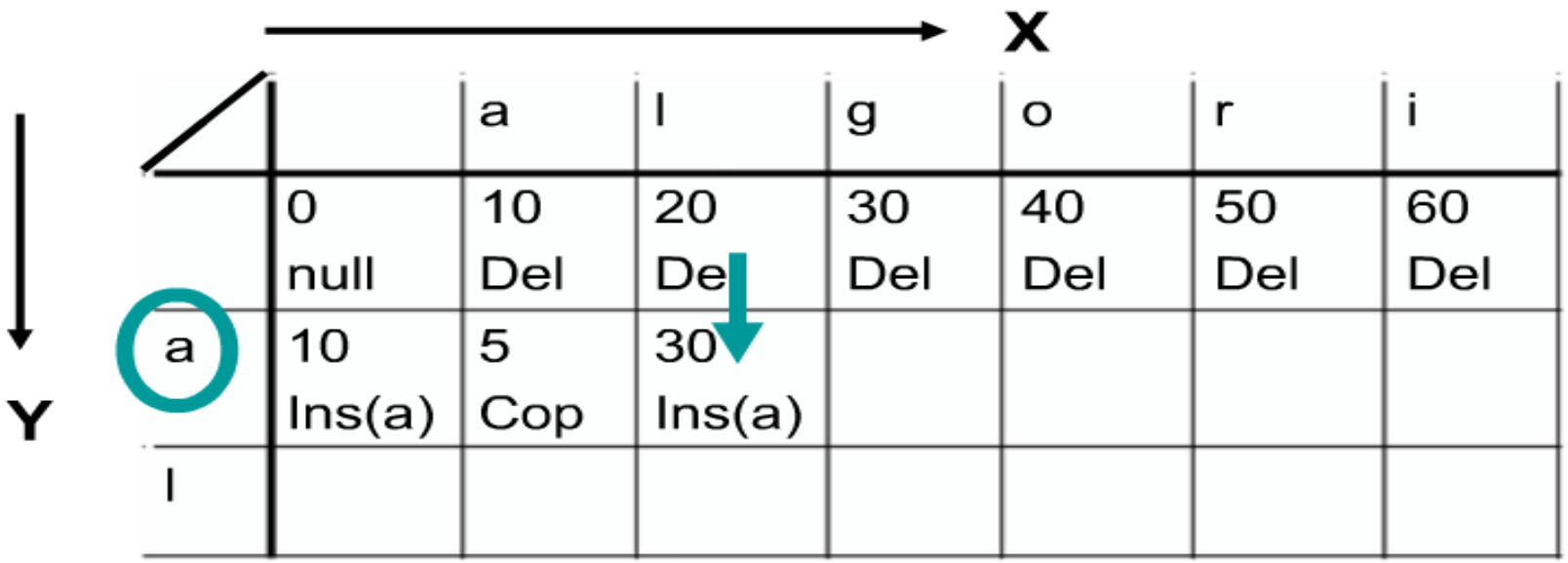
How to fill out the matrix (cont.)

- **Two** possibilities to consider for the marked square.
- Copy is **not** possible since “a” != “l”.



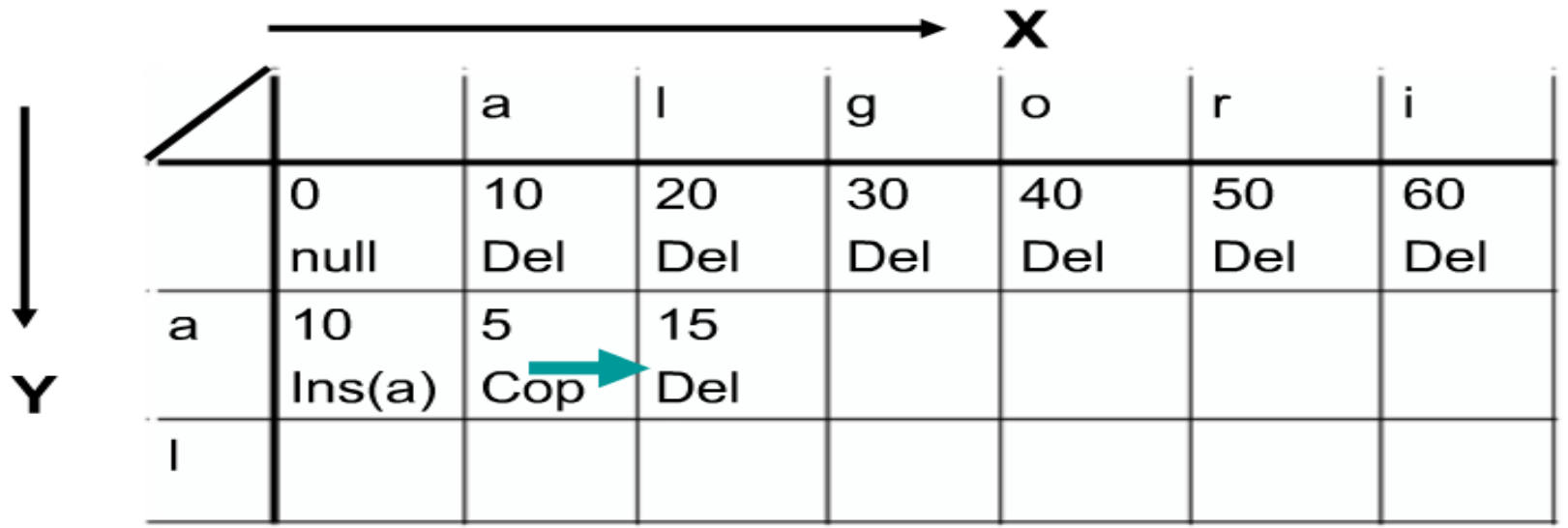
How to fill out the matrix (cont.)

- If we chose Insert(a)….



How to fill out the matrix (cont.)

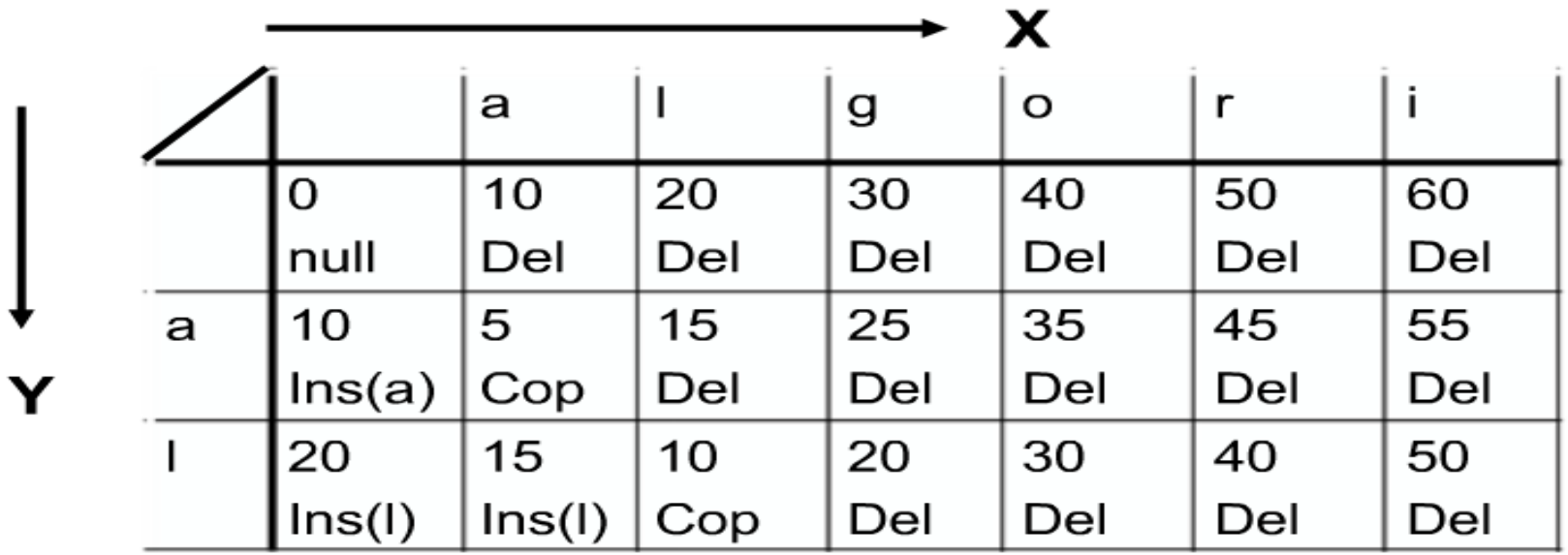
- But Delete is cheaper!



		a	l	g	o	r	i
	0 null	10 Del	20 Del	30 Del	40 Del	50 Del	60 Del
a	10 Ins(a)	5 Cop	15 Del				
l							

How to fill out the matrix (cont.)

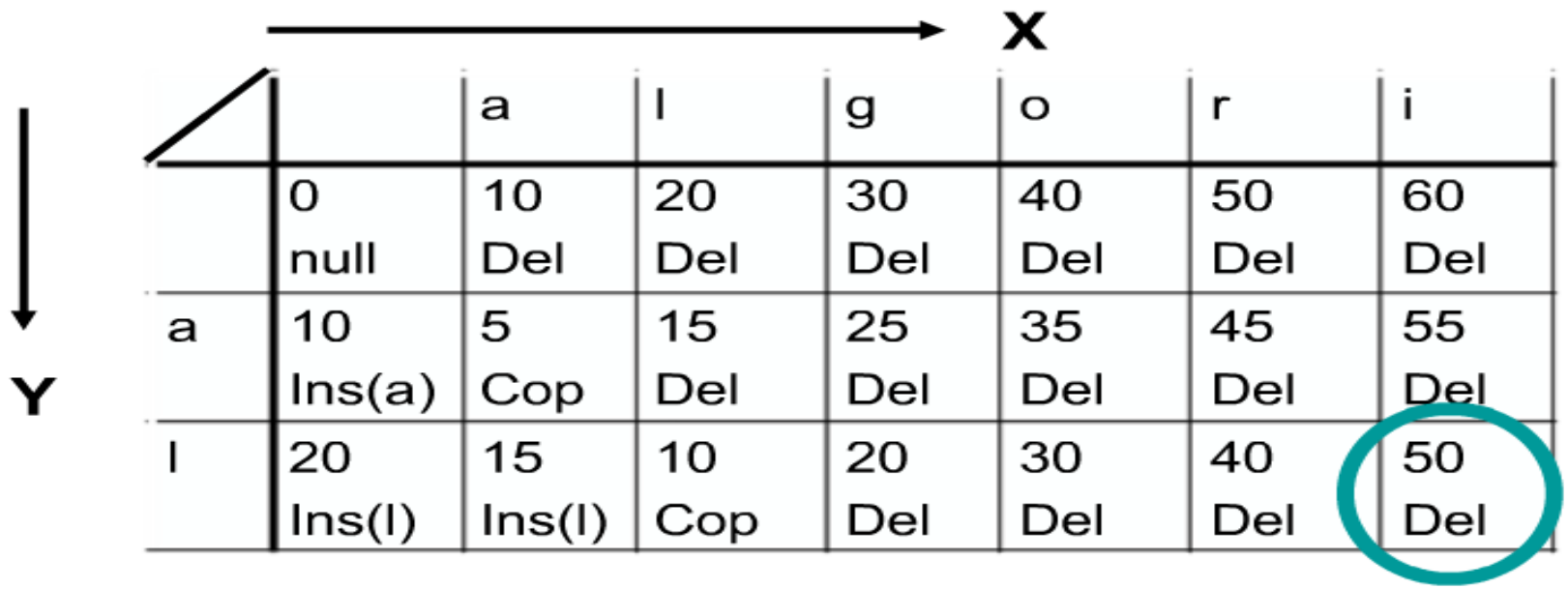
- Fill the rest of the matrix out in a similar fashion.



		a	l	g	o	r	i
	0 null	10 Del	20 Del	30 Del	40 Del	50 Del	60 Del
a	10 Ins(a)	5 Cop	15 Del	25 Del	35 Del	45 Del	55 Del
l	20 Ins(l)	15 Ins(l)	10 Cop	20 Del	30 Del	40 Del	50 Del

Transforming “algori” to “al”

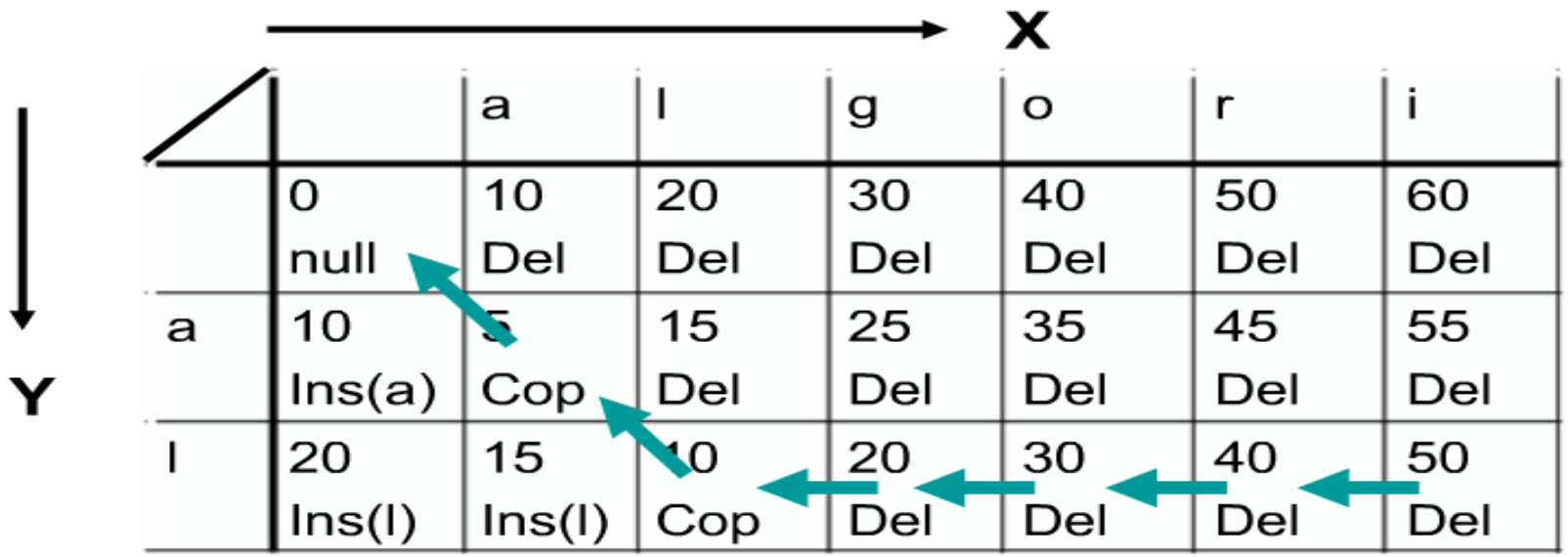
- The cheapest sequence of operations has cost: 50



		a	l	g	o	r	i
	0 null	10 Del	20 Del	30 Del	40 Del	50 Del	60 Del
a	10 Ins(a)	5 Cop	15 Del	25 Del	35 Del	45 Del	55 Del
l	20 Ins(l)	15 Ins(l)	10 Cop	20 Del	30 Del	40 Del	50 Del

Transforming “algori” to “al” (cont.)

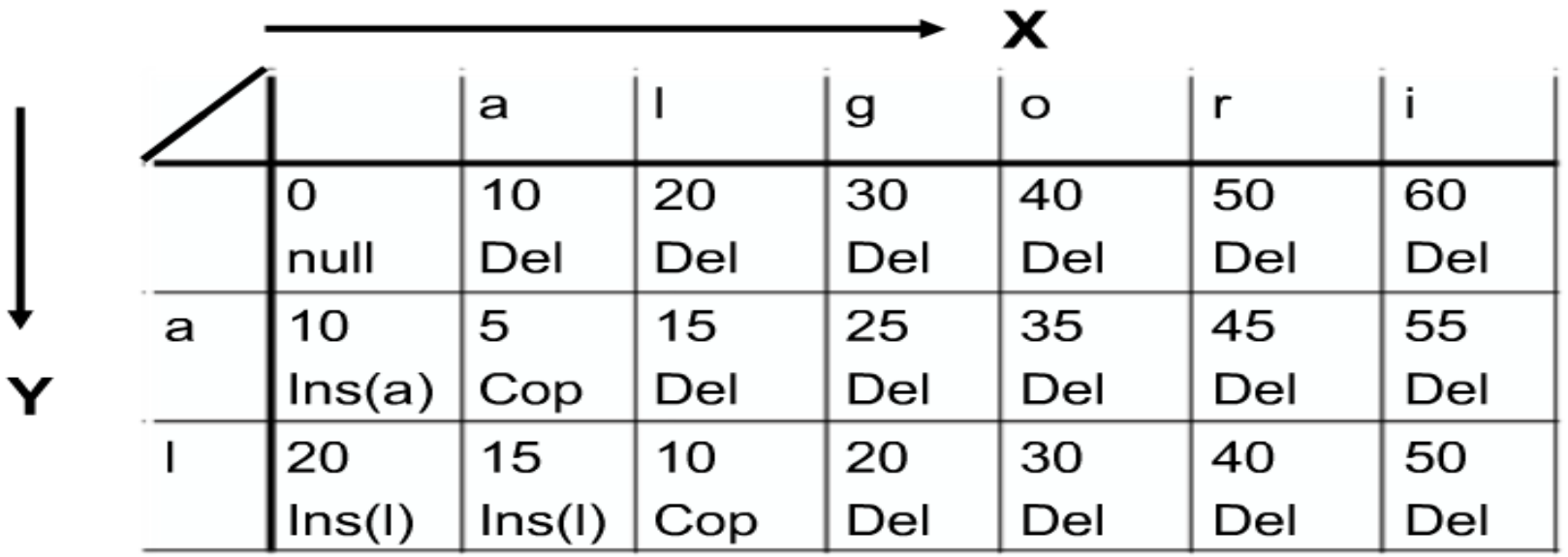
- Recover the sequence by working backwards.
- We get: Copy, Copy, Delete, Delete, Delete, Delete.



		a	l	g	o	r	i
	0 null	10 Del	20 Del	30 Del	40 Del	50 Del	60 Del
a	10 Ins(a)	5 Cop	15 Del	25 Del	35 Del	45 Del	55 Del
l	20 Ins(l)	15 Ins(l)	10 Cop	20 Del	30 Del	40 Del	50 Del

Why this worked

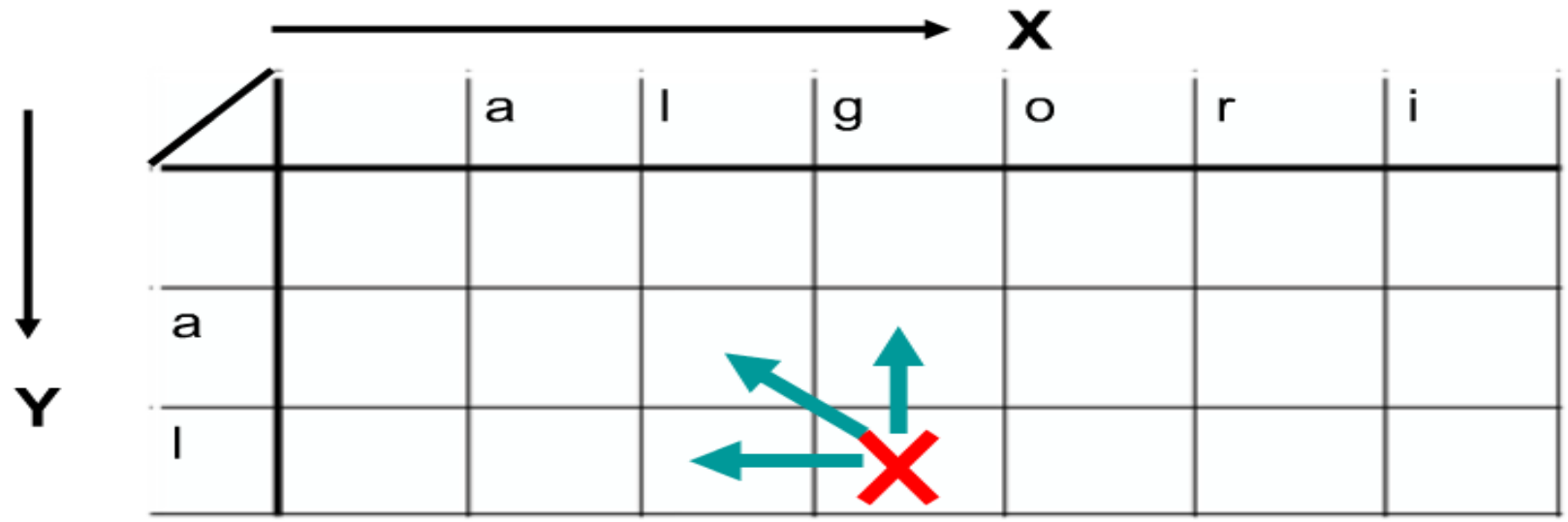
- This worked becuase of the three observations we made earlier.



		a	l	g	o	r	i
	0 null	10 Del	20 Del	30 Del	40 Del	50 Del	60 Del
a	10 Ins(a)	5 Cop	15 Del	25 Del	35 Del	45 Del	55 Del
l	20 Ins(l)	15 Ins(l)	10 Cop	20 Del	30 Del	40 Del	50 Del

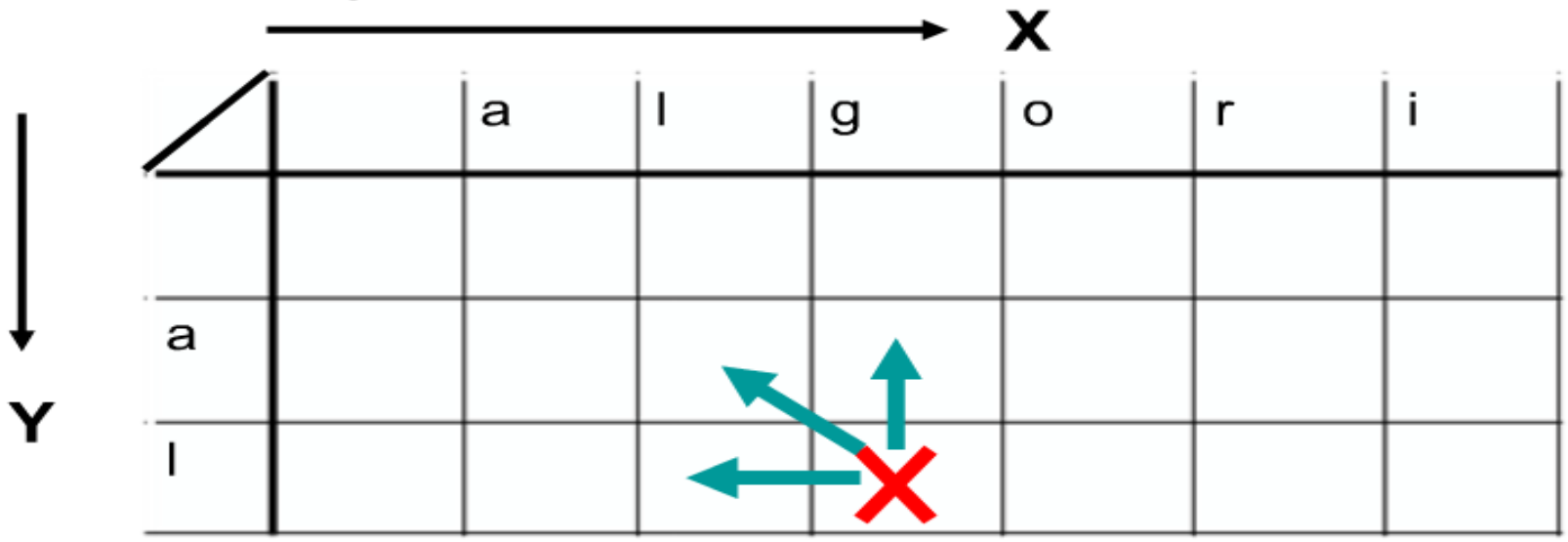
Why this worked (cont.)

- The best answer to put in a location must use the best solution to one of three possible subproblems.



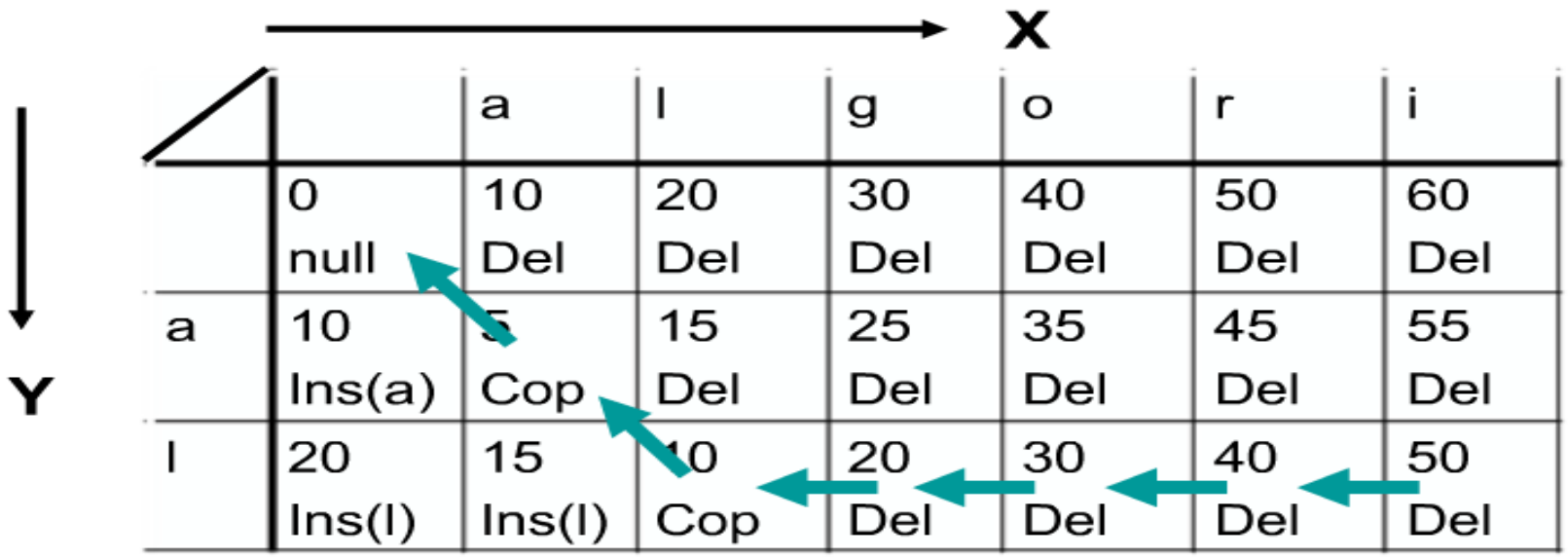
Why this worked (cont.)

- So we solved those subproblems first.
- Then we considered cases and figured out how best to solve our current problem.



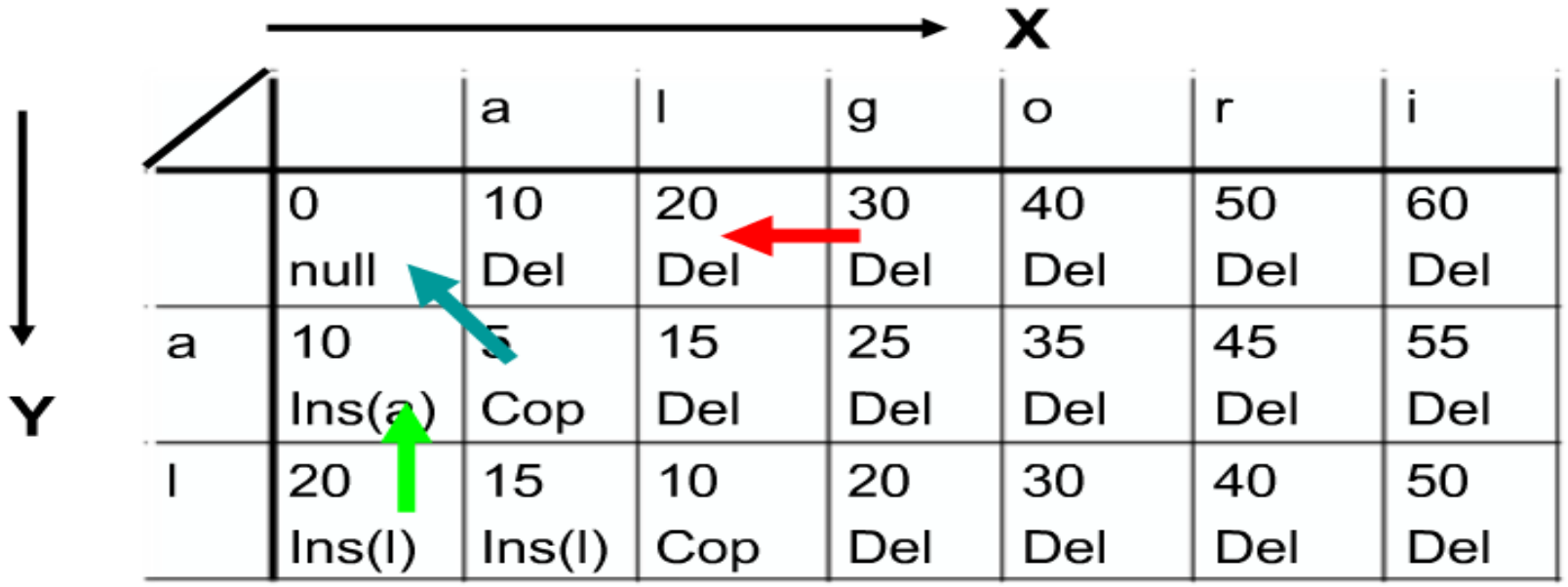
Why this worked (cont.)

- We could recover the cheapest sequence of operations since we stored operations at each step.



Why this worked (cont.)

- Each location's operation tells us where to look for the previous one in the sequence.



		a	l	g	o	r	i
	0 null	10 Del	20 Del	30 Del	40 Del	50 Del	60 Del
a	10 Ins(a)	5 Cop	15 Del	25 Del	35 Del	45 Del	55 Del
l	20 Ins(l)	15 Ins(l)	10 Cop	20 Del	30 Del	40 Del	50 Del

Defining Min Edit Distance (*Levenshtein)*

- Initialization

  D(i,0) = i

  D(0,j) = j

- Recurrence Relation:

  For each i = 1…M

  For each j = 1…N

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } X(i) \neq Y(j) \\ 0; & \text{if } X(i) = Y(j) \end{cases} \end{cases}$$

- Termination:

  D(N,M) is distance

Simple problems for edit distance

- $X=x_1, x_2, \ldots x_i, Y= y_1, y_2, \ldots, y_j$

$$D(i,j) = \min[D(i-1,j)+\delta_d,\ D(i, j-1)+ \delta_i,\ D(i-1, j-1)+0/2\ \delta_c]$$

$\delta_d=1$, $\delta_i= 1$, $\delta_c=0/2$ $\quad$ $[0/\ \delta_c$: 0 (if $x_i=y_j$), 2 /δ_c (if $x_i \neq y_j$)]

Q1. What is the cheapest way to transform "*monkey*" into "money"?

Q2. What is the cheapest way to transform "*Wednesday*" into "*Website*"?

		m	o	n	k	e	y
	0	1	2	3	4	5	6
m	1	0	1	2	3	4	5
o	2	1	0	1	2	3	4
n	3	2	1	0	1	2	3
e	4	3	2	1	1	1	2
y	5	4	3	2	2	2	1

Levenshtein Distance

Implementation

Implementation of edit distance

```
def editDistance(str1, str2, m, n):
    # empty case
    if m == 0:
       return n
    # empty case
    if n == 0:
       return m

    # last character is same to each other
    if str1[m-1] == str2[n-1]:
        return editDistance(str1, str2, m-1, n-1)

    # if the last character is not same, check the three operations
    return 1 + min(editDistance(str1, str2, m, n-1),    # Insert
                   editDistance(str1, str2, m-1, n),    # Remove
                   editDistance(str1, str2, m-1, n-1)    # Replace
                   )


# Driver code
str1 = "monkey"
str2 = "money"
print(editDistance(str1, str2, len(str1), len(str2)))
```

Implementation of edit distance

- Initialization

  D(i,0) = i

  D(0,j) = j

- Recurrence Relation:

  For each i = 1…M

  For each j = 1…N

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } X(i) \neq Y(j) \\ 0; & \text{if } X(i) = Y(j) \end{cases} \end{cases}$$

- Termination:

  D(N,M) is distance

```python
def editDistance(str1, str2, m, n):

    dp = [[0 for x in range(n + 1)] for x in range(m + 1)]

    for i in range(m + 1):
        for j in range(n + 1):

            # empty case
            if i == 0:
                dp[i][j] = j

            # empty case
            elif j == 0:
                dp[i][j] = i

            # last character is same to each other
            elif str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1]

            # If last character are different, consider three operations
            else:
                dp[i][j] = 1 + min(dp[i][j-1],        # Insert
                                   dp[i-1][j],        # Remove
                                   dp[i-1][j-1])    # Replace

    return dp[m][n]

# Driver code
str1 = "monkey"
str2 = "money"
print(editDistance(str1, str2, len(str1), len(str2)))
```

Example code test

- Code test: https://www.acmicpc.net/problem/15483
- Solving the problem using edit distance
- Example result of submission

THANK YOU