Lecture 12

Dynamic programming I

Chinmay Nirkhe | CSE 421 Winter 2026

A new algorithmic paradigm

- **Greedy algorithms**:
  - Identify a “local” property to optimize
  - Generating a “global” solution by combining individual decisions
- **Divide and conquer**:
  - Recursively solve computational task by identifying **independent** subtasks
  - Each independent subtask is smaller than the original $n \to 0.9n$
  - Combine solutions to subtasks to solve original problem
  - The subtasks are different from each other and repeat substacks don’t occur

A new algorithmic paradigm

Dynamic programming

- **Optimal substructure:**
  - The optimal value of the problem can easily be obtained given the optimal values of subproblems.
  - In other words, there is a recursive algorithm for the problem which would be fast if we could just skip the recursive steps.
- **Overlapping subproblems:**
  - The subproblems share sub-subproblems.
  - In other words, if you actually ran that naïve recursive algorithm, it would waste a lot of time solving the same problems over and over again.

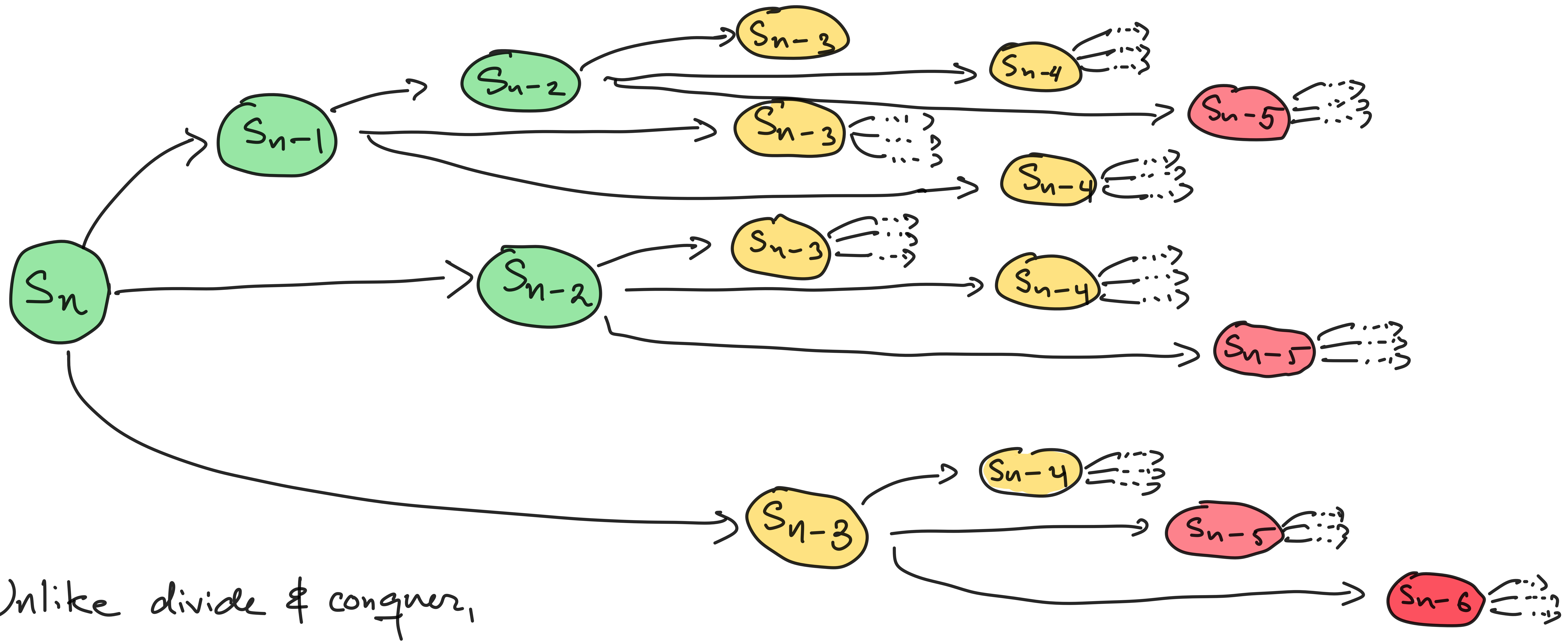
Tribonacci numbers

- **Input**: Integer n
- **Output**: Tribonacci number s_n defined recursively $s_1 = s_2 = s_3 = 1$ and

  $s_n = s_{n-1} + s_{n-2} + s_{n-3}$.

- There is a canonical recursive algorithm. But it's not very efficient.

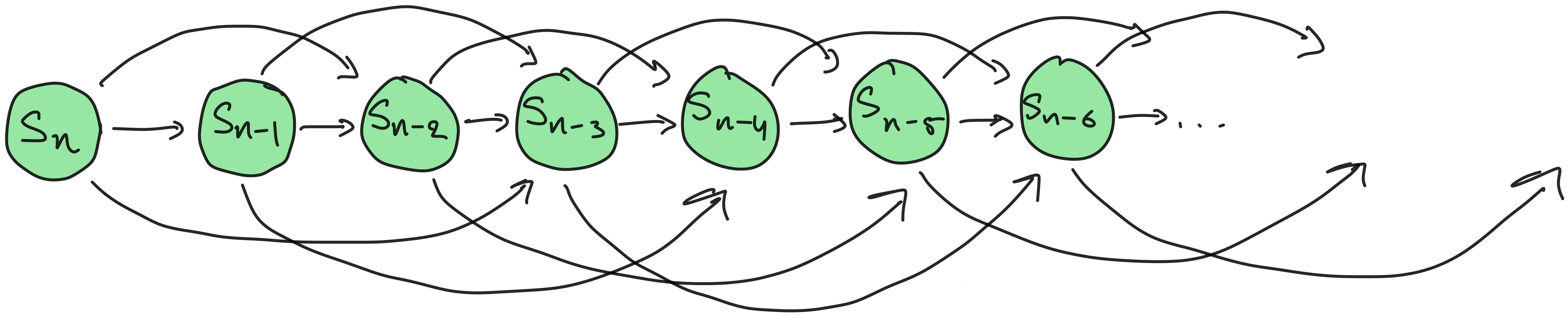
Overlapping subproblems



Unlike divide & conquer,

there are many repeated subproblems...

Overlapping subproblems



Unlike divide & conquer,

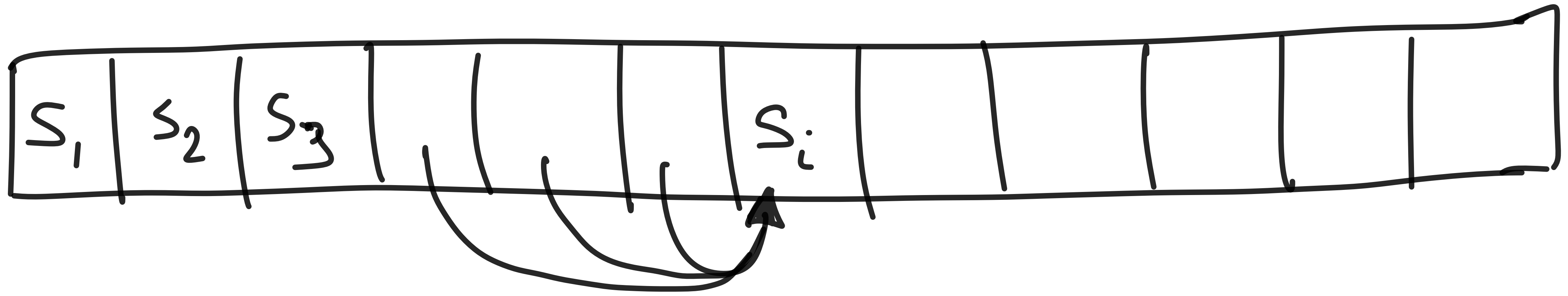
there are many repeated subproblems...

Memoization

- **Input**: Integer n
- **Output**: Tribonacci number s_n

This is the "memo" in memoization

- **Algorithm**:
  - Initialize an array s of length n
  - Set $s_1, s_2, s_3 \leftarrow 1$
  - For $i \leftarrow 4$ to n, set $s_i \leftarrow s_{i-1} + s_{i-2} + s_{i-3}$



Tribonacci runtime analysis

- **Theorem:** $s_n \leq 2^n$.
- **Proof:** By induction. Base cases are $s_1 = s_2 = s_3 = 1$. For induction

  $s_n = s_{n-1} + s_{n-2} + s_{n-3} \leq 2^{n-1} + 2^{n-2} + 2^{n-3} \leq 7 \cdot 2^{n-3} \leq 2^n$.
- **Corollary**: Each s_n can be expressed using n-bits.

Tribonacci runtime analysis

- Computing each entry s_i of the array takes 3 additions: $O(n)$ time
- Total time: $O(n^2)$, total space: $O(n^2)$

- Could we have done better?
  - Better time analysis: $O(1) + \sum_{i=4}^{n} O(i) = O(n^2)$ (only constant factor)
  - Better space: Use only $O(1)$ registers $= O(n)$ bits by recycling old terms in array

A note on runtime

- Runtime is often nebulously expressed
- **Example 1**: Sorting a list of n integers
  - The runtime is often expressed as $O(n \log n)$ time
  - But this is misleading — recall, it is really $O(n \log n)$ **arithmetic operations**
  - If each arithmetic is on k-bit integers (between 0 and $2^k - 1$), then this takes $O(n \log n \cdot k)$.
  - Input length is $\Theta(n)$ numbers or $\Theta(nk)$ bits.

A note on runtime

- Runtime is often nebulously expressed
- **Example 2**: Dealing with a graph $G = (V, E)$
    - The runtime is often expressed in terms of $n = |V|, m = |E|$
    - We are implicitly assuming the graph is expressed as an adjacency list

      Input: $\langle V = (1,\ldots,6), N_1 = (2,3,4), N_2 = (1,5), N_3 = (1), N_4 = (1,5), N_5 = (2,4), N_6 = () \rangle$
    - Length of input is $\Theta(n + m)$
    - If runtime is $f(n + m)$ then the runtime is also $O(f(|\text{input}|))$
    - We aren't really losing much by expressing the runtime in terms of n and m

A note on runtime

- Runtime is often nebulously expressed
- **Example 2**: Dealing with a graph $G = (V, E)$
  - Sometimes a graph is expressed as an adjacency matrix $M \in \{0,1\}^{n \times n}$ where $M_{ij} = 1$ if $(i, j) \in E$ and $= 0$ otherwise.
  - Input length is now $\Theta(n^2)$
  - So a runtime of $f(n)$ is equal to $O(f(\sqrt{|\text{input}|}))$

A note on runtime

- Runtime is often nebulously expressed
- **Example 3**: The input is an integer $n \in \mathbb{N}$
  - An integer can be expressed in unary $\underbrace{111\ldots1}_{n \text{ ones}}$ or in binary in $O(\log n)$ bits
  - The runtime can depend on how the input is expressed

Tribonacci runtime analysis

- **Unary input**
  - Runtime is $O(n^2)$ where $n = |\text{input}|$
- **Binary input**
  - Runtime is $O(4^\ell)$ where $\ell = |\text{input}|$

- Best possible runtime is $O(n \log^2 n)$ using explicit formula:

  $s_n = a_1 r_1^n + a_2 r_2^n + a_3 r_3^n$ for some algebraic numbers $a_1, a_2, a_3, r_1, r_2, r_3$ and using optimal algorithm for integer multiplication

Edit distance

- **Input:** Two strings $X = (x_1 \ldots x_m)$ and $Y = (y_1 \ldots y_n)$
- **Output:** A minimal sequence of edit operations converting X into Y with allowed transformations being Delete, Insert, or Substitute (one character)

M I S C H E V I O U S

M I S C H I E V O U S

Edit distance

- **Input**: Two strings $X = (x_1 \ldots x_m)$ and $Y = (y_1 \ldots y_n)$
- **Output**: A minimal sequence of edit operations converting X into Y with allowed transformations being Delete, Insert, or Substitute (one character)

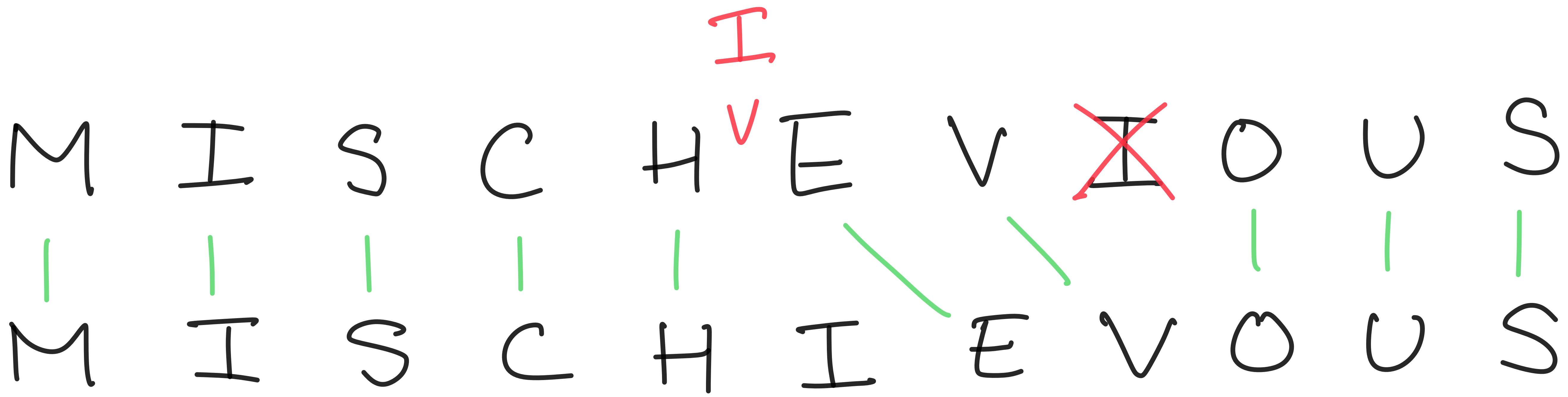
M I S C H E V I O U S

M I S C H I E V O U S

3 Edits

Edit distance

- **Input**: Two strings $X = (x_1 \ldots x_m)$ and $Y = (y_1 \ldots y_n)$
- **Output**: A minimal sequence of edit operations converting X into Y with allowed transformations being Delete, Insert, or Substitute (one character)



2 Edits

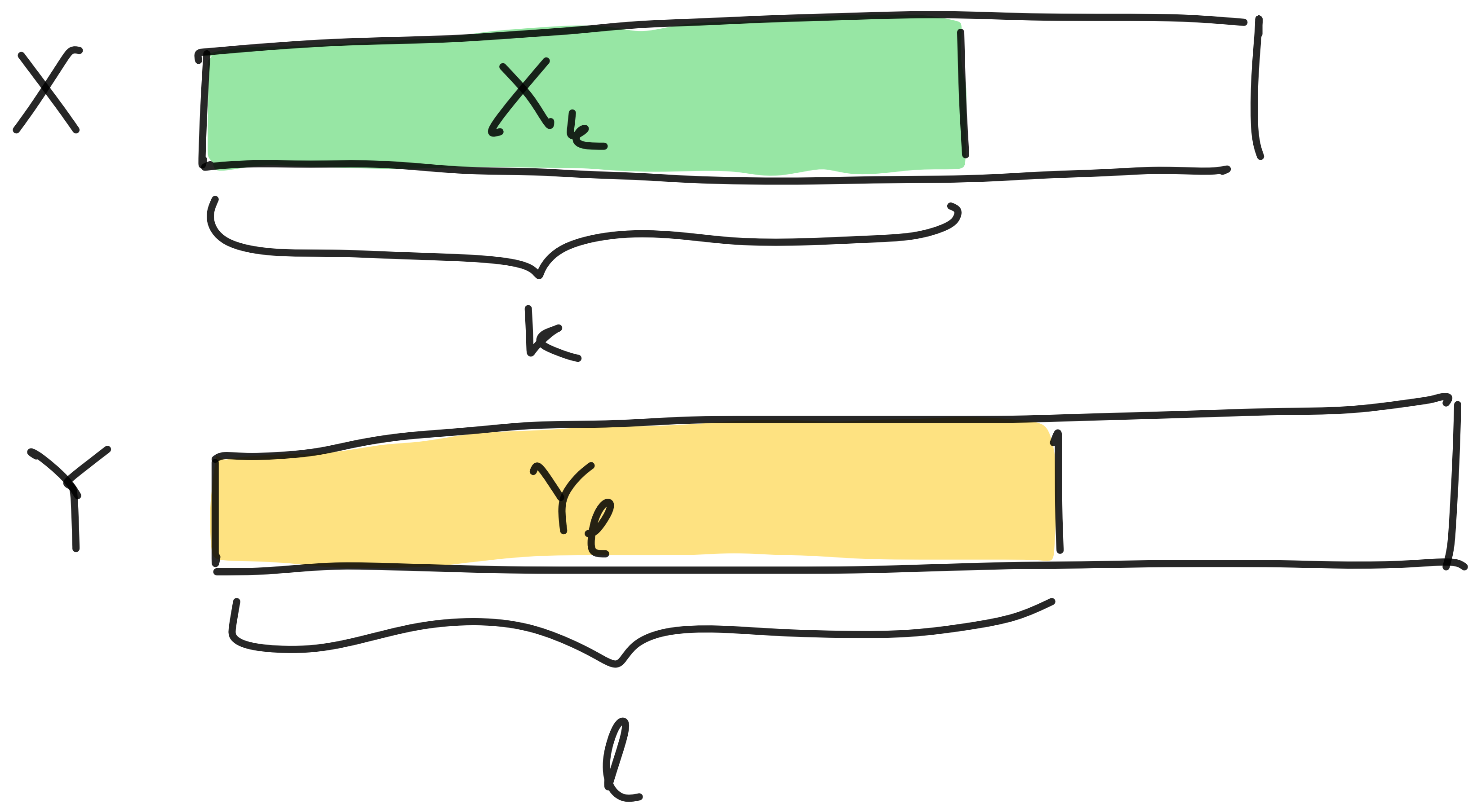
Edit distance

- **Input:** Two strings $X = (x_1 \ldots x_m)$ and $Y = (y_1 \ldots y_n)$
- **Output:** A minimal sequence of edit operations converting X into Y with allowed transformations being Delete, Insert, or Substitute (one character)

- To find a dynamic programming algorithm, we need to reframe the problem as a **special case** of a general problem which is recursively defined

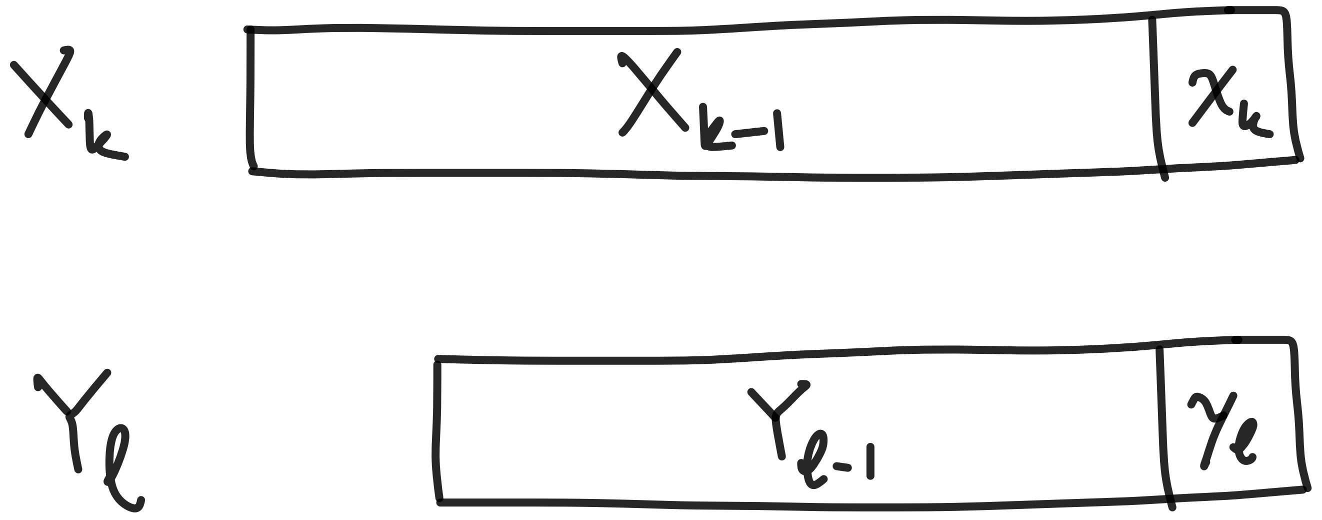
Edit distance

- **Input:** Two strings $X = (x_1 \ldots x_m)$ and $Y = (y_1 \ldots y_n)$
- **Definitions:**
  - Let X_k be the prefix of the first k characters of X
  - Let Y_ℓ be the prefix of the first ℓ characters of Y
  - Let $d(k, \ell)$ be the minimal edit distance between X_k and Y_ℓ
- **Base case:** $d(0, \ell) = \ell$, need to insert all characters
- **Base case:** $d(k, 0) = k$, need to delete all characters
- **Observation:** The order in which edits are made is irrelevant.



Recursive definition

Observation: The last character must change from x_k to y_ℓ if they differ.

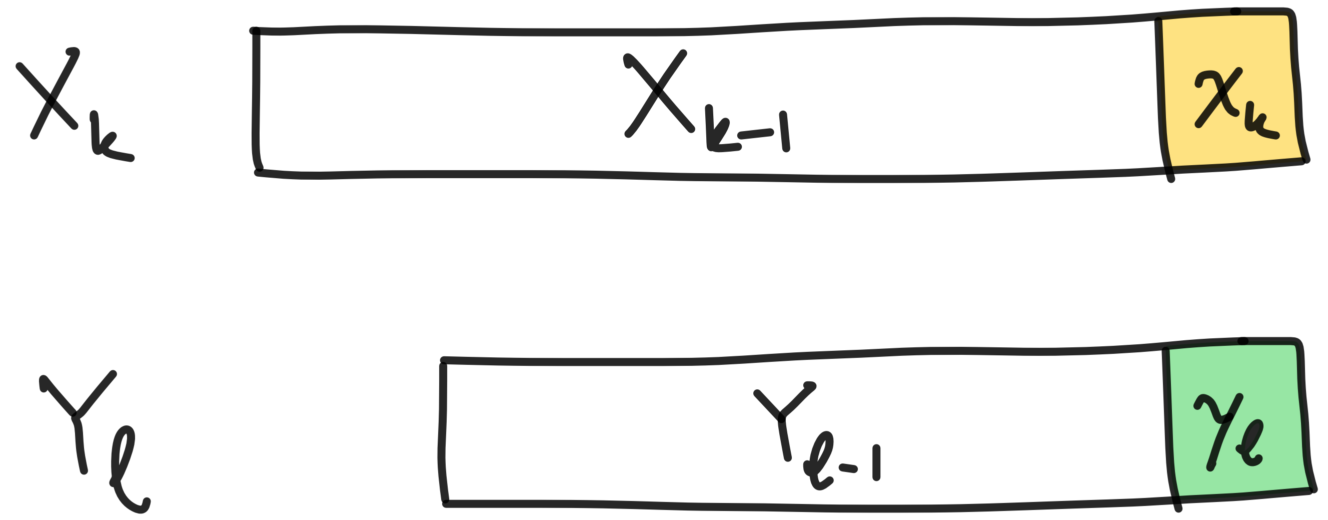


If $x_k = y_\ell$, this simplifies to computing the edit distance between X_{k-1} and $Y_{\ell-1}$, i.e.

$$d(k, \ell) = d(k-1, \ell-1).$$

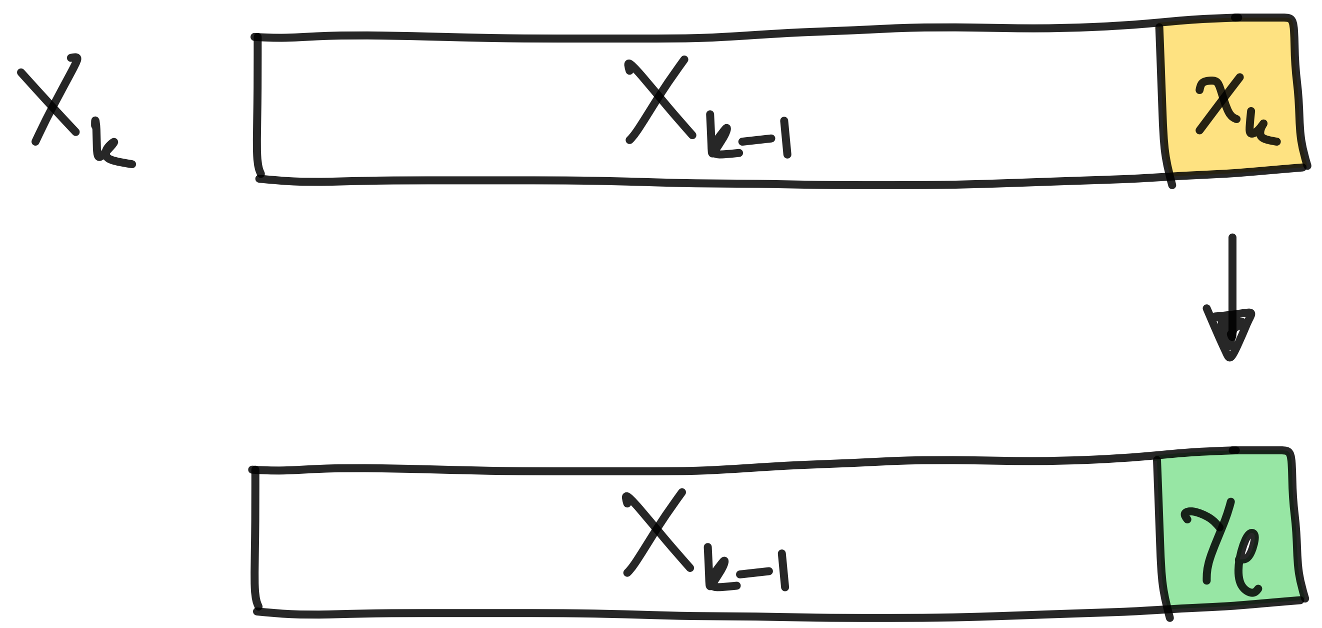
Recursive definition

If $x_k \neq y_\ell$,



there are 3 ways the last character will get set.

Case 1: Substitution

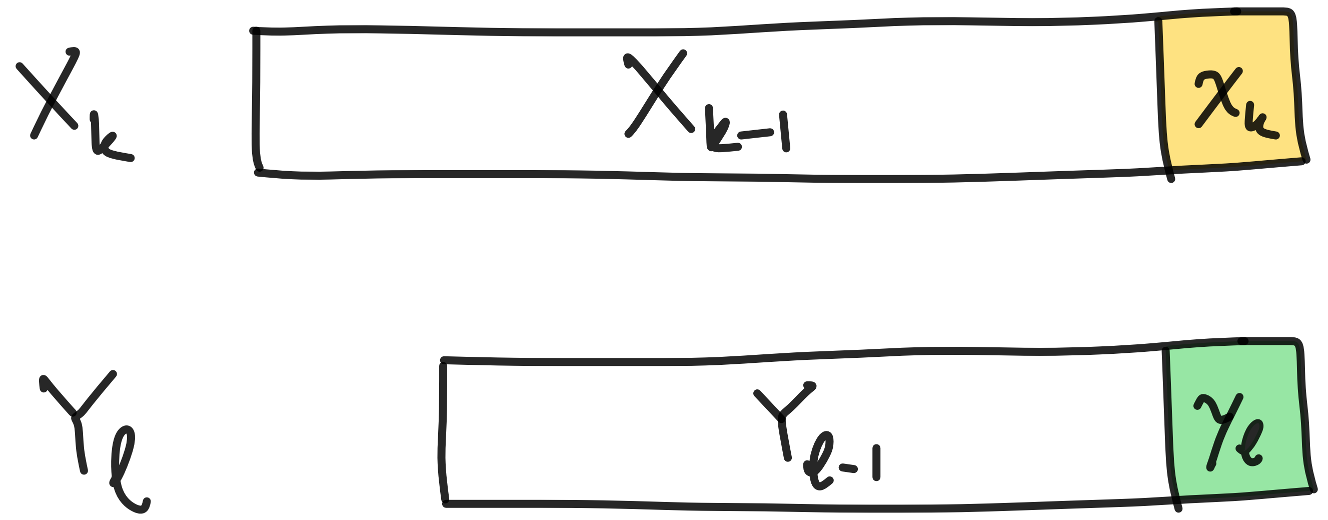


Problem simplifies to editing X_{k-1} to $Y_{\ell-1}$.

So $d(k,\ell) \leq d(k-1,\ell-1) + 1$.

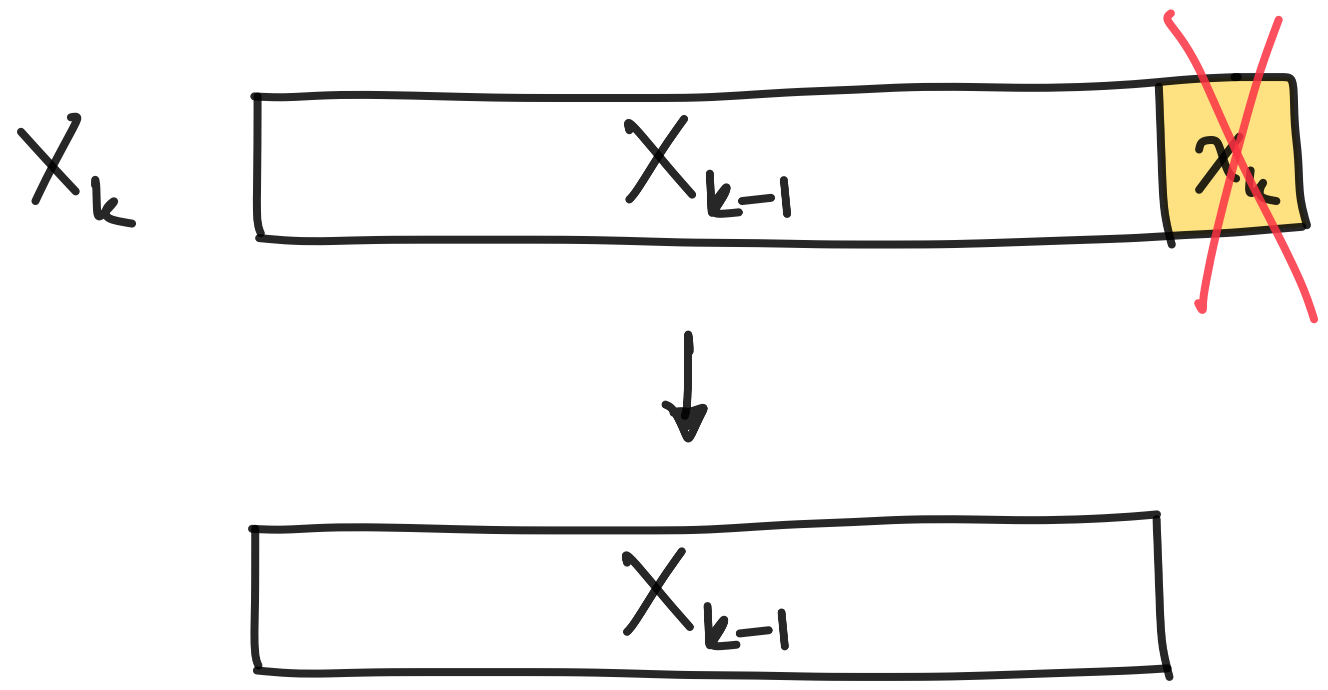
Recursive definition

If $x_k \neq y_\ell$,



there are 3 ways the last character will get set.

Case 2: Deletion

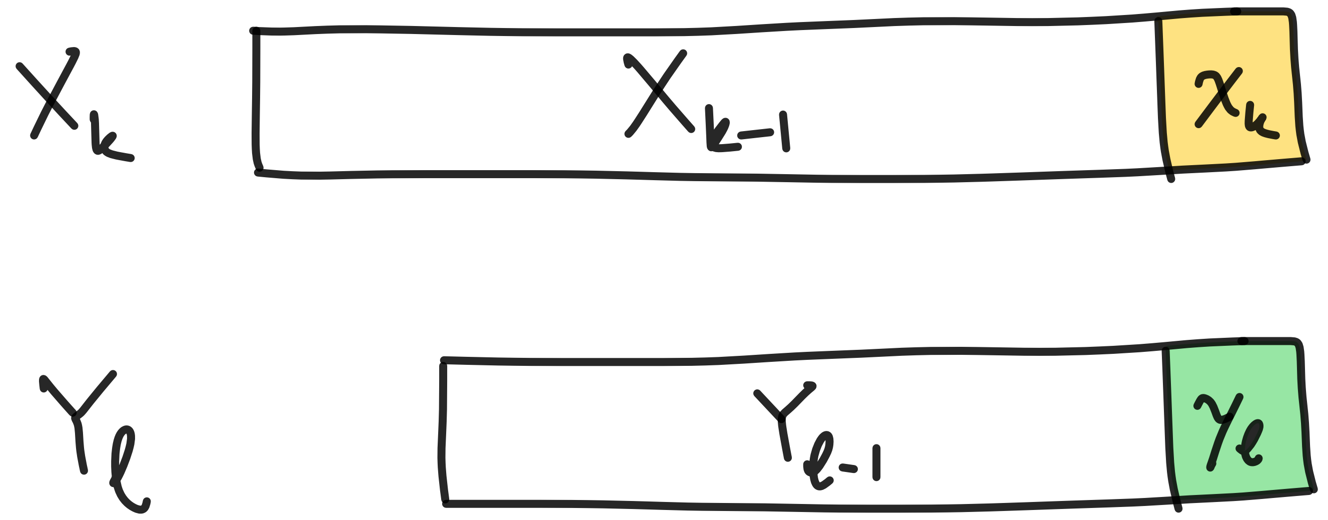


Problem simplifies to editing X_{k-1} to Y_ℓ.

So $d(k,\ell) \leq d(k-1,\ell) + 1$.

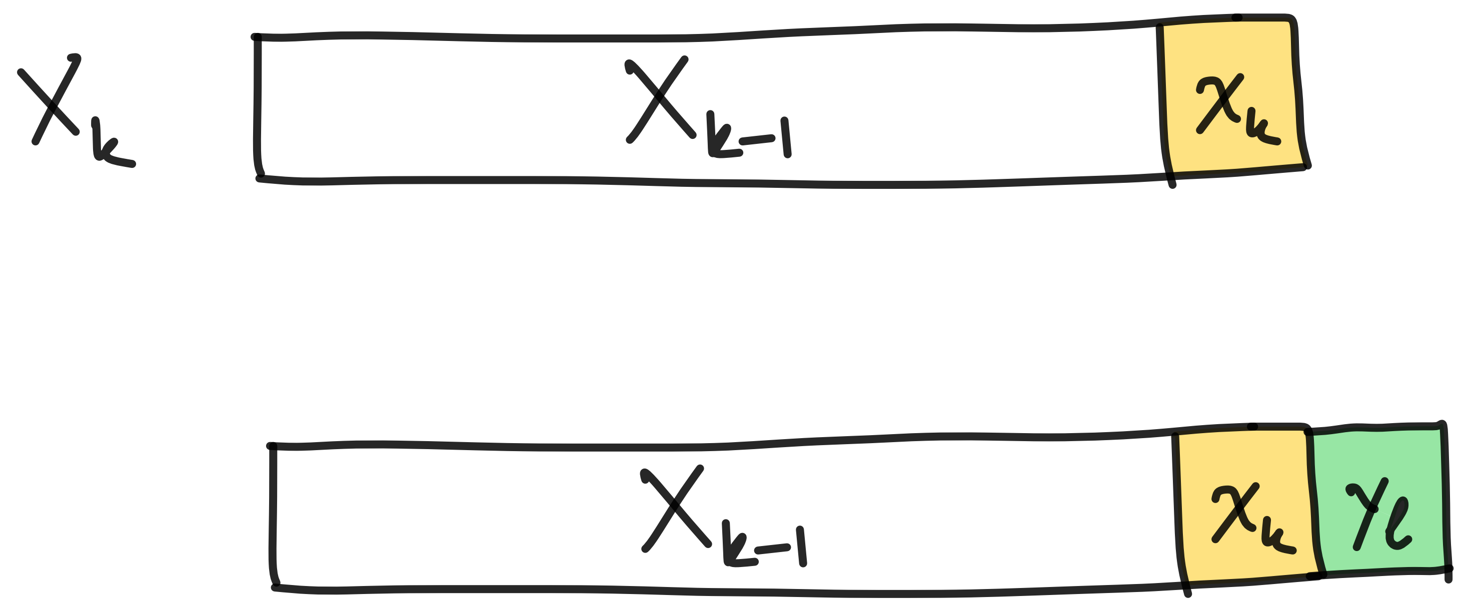
Recursive definition

If $x_k \neq y_\ell$,



there are 3 ways the last character will get set.

Case 3: Insertion

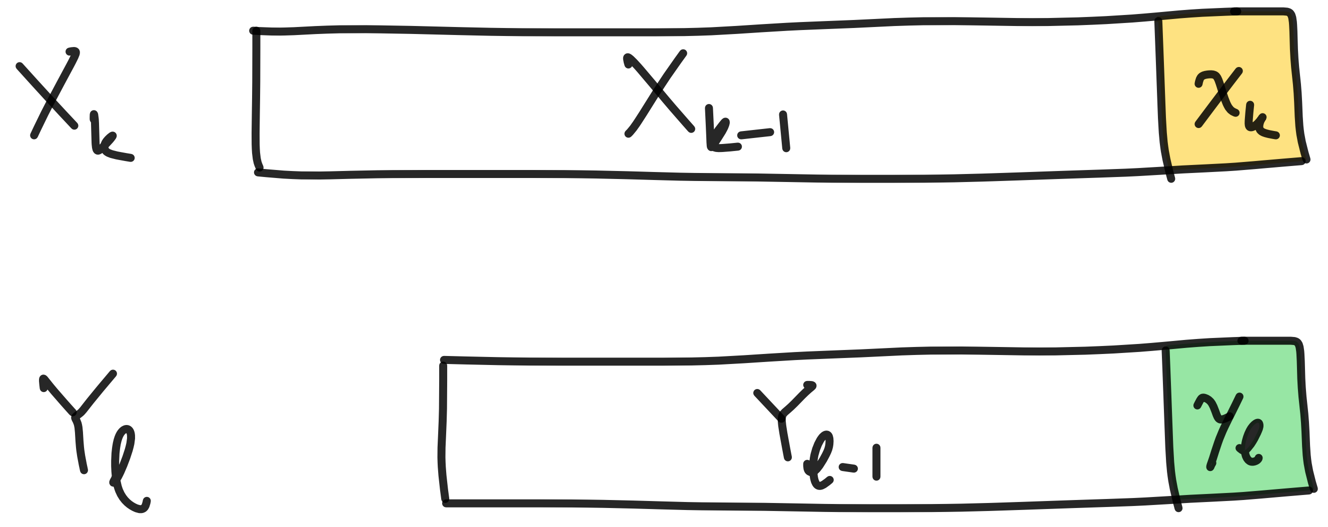


Problem simplifies to editing X_k to $Y_{\ell-1}$.

So $d(k,\ell) \leq d(k,\ell-1) + 1$.

Recursive definition

If $x_k \neq y_\ell$,



there are 3 ways the last character will get set.

One of these 3 cases must occur.

So, if $x_k \neq y_\ell$,

$$d(k,\ell) = 1 + \min \begin{cases} d(k-1,\ell-1) \\ d(k-1,\ell) \\ d(k,\ell-1) \end{cases}$$

Recursive algorithm

- **Recursive algorithm** $d(k, \ell)$**:**
  - If $k = 0$, then return ℓ
  - If $\ell = 0$, then return k
  - If $x_k = y_\ell$,
    - Return $d(k-1, \ell-1)$
    - Else, return $1 + \min \left\{ \begin{matrix} d(k-1, \ell-1), \\ d(k, \ell-1), \\ d(k-1, \ell) \end{matrix} \right\}$.

The edit distance of the original problem is $d(n, m)$.

There are many repeated subproblems.

Memoization

Table of $d(k, \ell)$:

n					$d(n,m)$
3					
2			$d(k,\ell)$		
1					
0	1	2	3	4	m

Memoization

Table of $d(k, \ell)$:

n					$d(n,m)$
3					
2		$d(k-1,\ell-1)$	$d(k,\ell)$		
1		$d(k-1,\ell-1)$	$d(k,\ell-1)$		
0	1	2	3	4	m

Note that the value of $d(k,\ell)$ only depends on

1. if $x_k = y_\ell$
2. the 3 squares of one fewer Hamming weight.

Memoization

Table of $d(k, \ell)$:

n					$d(n,m)$
3					
2		$d(k-1,\ell-1)$	$d(k,\ell)$		
1		$d(k-1,\ell-1)$	$d(k,\ell-1)$		
0	1	2	3	4	m

Algorithm overview:

Fill table column by column, left to right, bottom to top using recursive def.

Output $d(n, m)$.

Edit distance algorithm

- Create a table $(n+1) \times (m+1)$ table d.
- Fill the base row and column to 0: (I.e., set $d(k,0) \leftarrow k$, $d(0,\ell) \leftarrow \ell$ for $k \in [n], \ell \in [m]$) ← $O(n+m)$ time
- Going left to right, bottom to top
- (I.e., For $k \leftarrow 1$ to n and for $\ell \leftarrow 1$ to m)

nm loops

  - If $x_k = y_\ell$, then set $d(k,\ell) \leftarrow d(k-1,\ell-1)$
  - Else, set $d(k,\ell) \leftarrow 1 + \min \left\{ \begin{array}{c} d(k-1,\ell-1), \\ d(k,\ell-1), \\ d(k-1,\ell) \end{array} \right\}.$

$O(1)$ computations per loop

- Return $d(n,m)$.

Total time $= O(nm)$

Finding the set of edits

- This algorithm only computes the edit distance.
- How do we also calculate the collection of edits that need to be made?
- Recall we set $d(k, \ell)$ based on a local **optimization** of subproblems
- **Solution:** Also keep track of which subproblem achieved the optimization
- Create a tree with $V = [n+1] \times [m+1]$ (the squares of the table) and a edge point from (k, ℓ) to the subproblem that solved the optimization

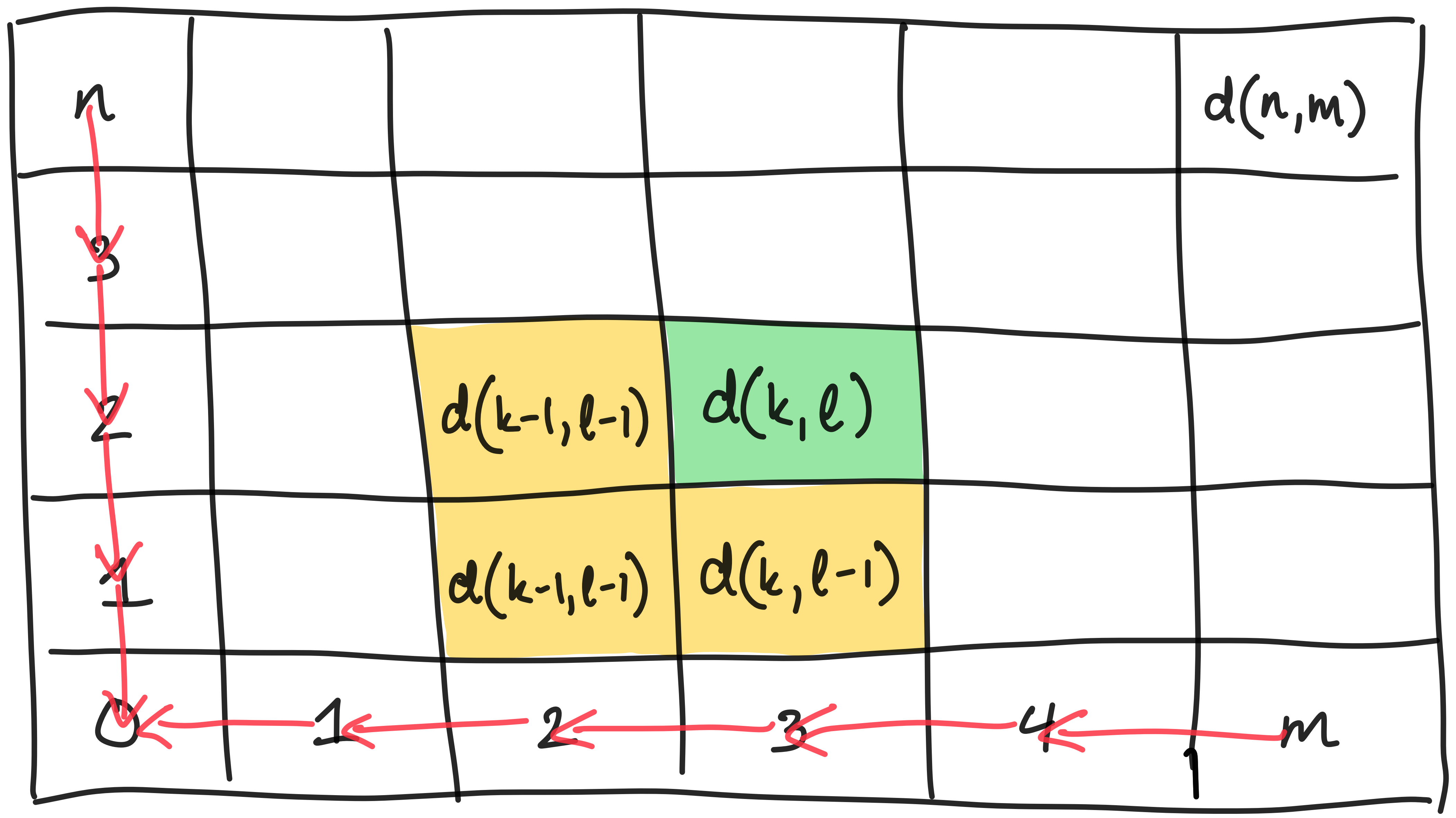
Finding the set of edits

Table of $d(k, \ell)$:

n					$d(n,m)$
3					
2		$d(k-1,\ell-1)$	$d(k,\ell)$		
1		$d(k-1,\ell-1)$	$d(k,\ell-1)$		
0	1	2	3	4	m

Finding the set of edits

Table of $d(k, \ell)$:

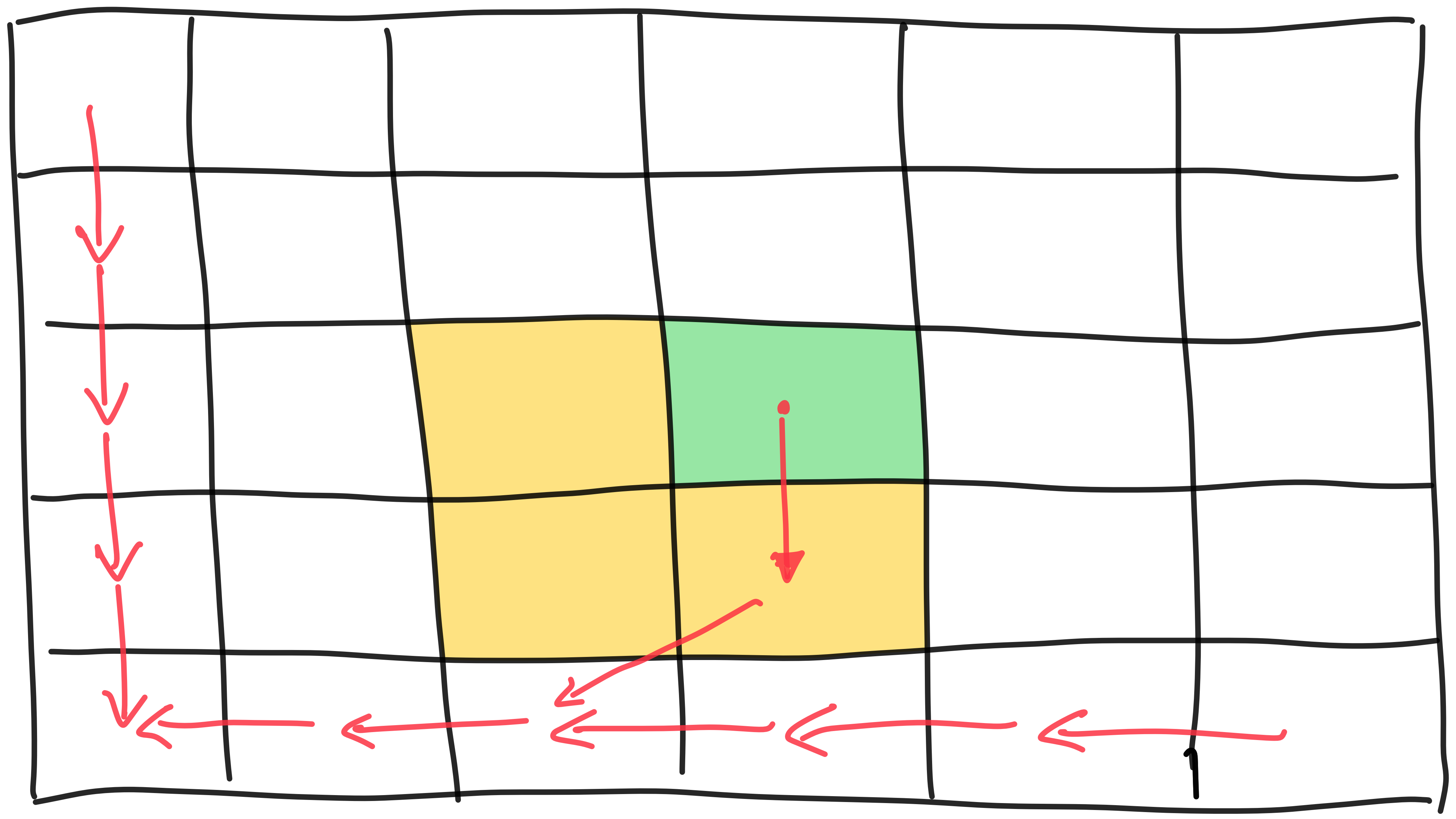


↓ arrow from $d(k, \ell)$ means "Insert y_ℓ"

← arrow from $d(k, \ell)$ means "Delete x_k"

Finding the set of edits

Table of $d(k, \ell)$:

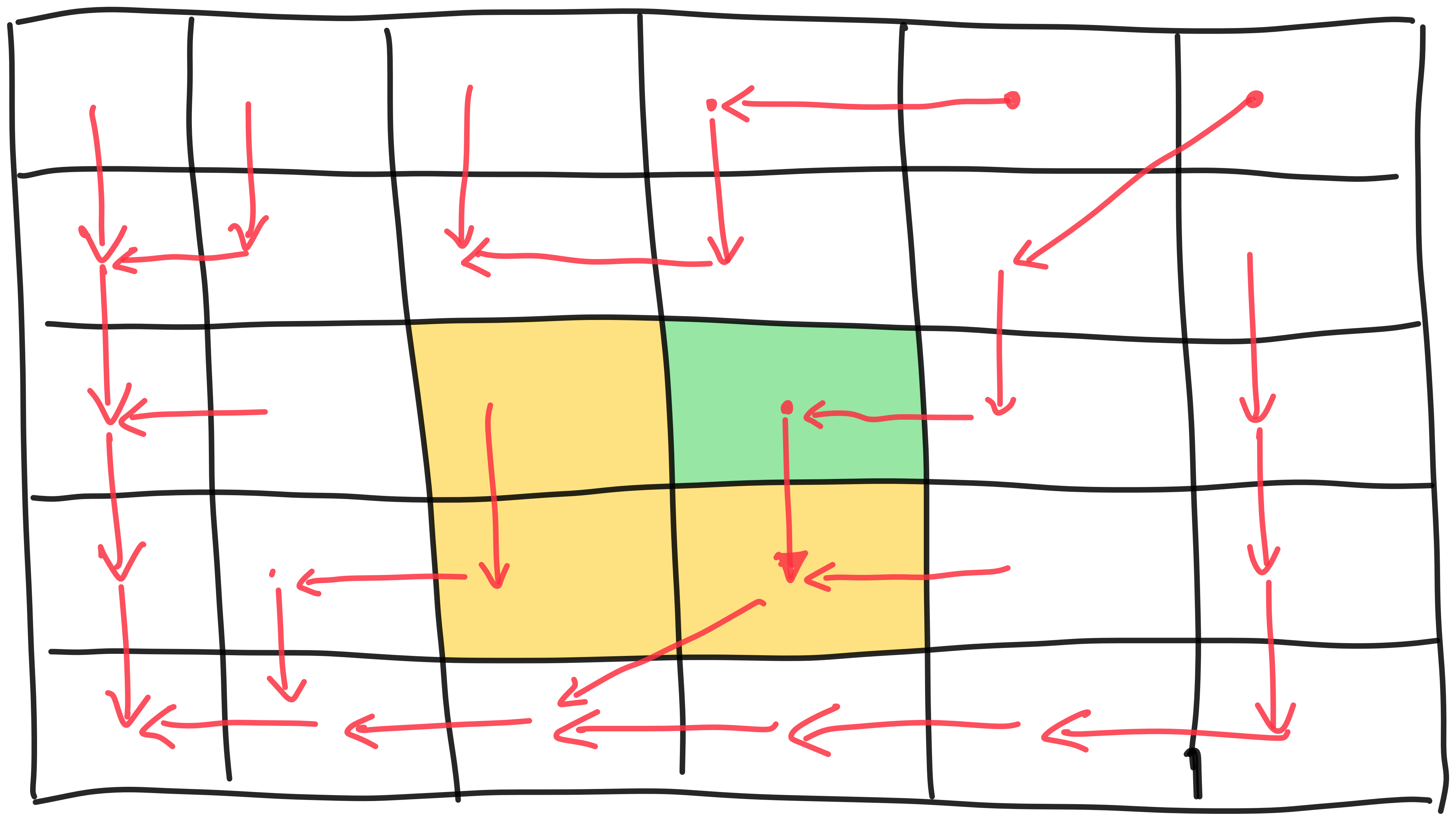


$\downarrow$ arrow from $d(k, \ell)$ means "Insert y_ℓ"

$\leftarrow$ arrow from $d(k, \ell)$ means "Delete x_k"

Finding the set of edits

Table of $d(k, \ell)$:



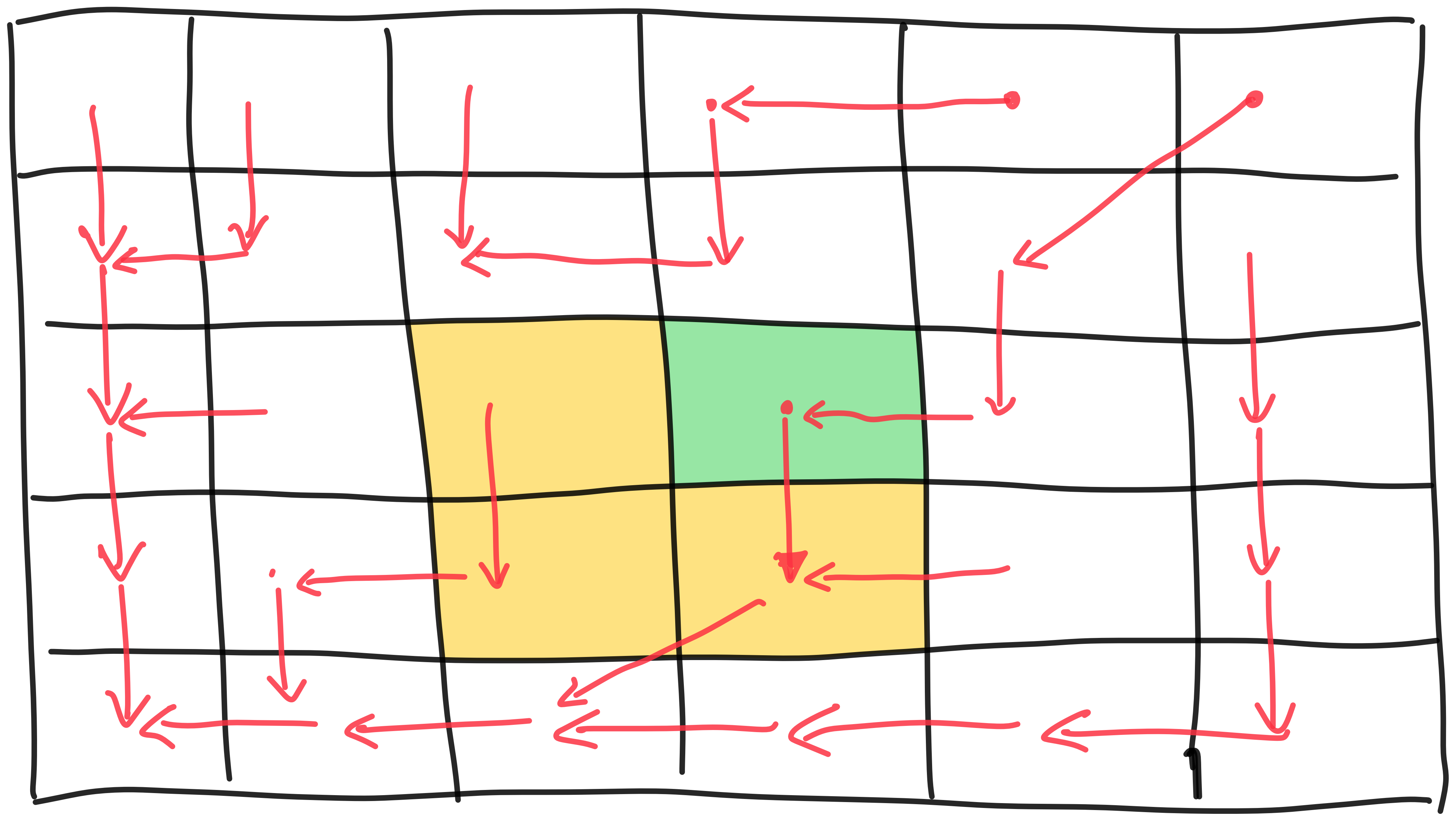
$\downarrow$ arrow from $d(k,\ell)$ means "Insert y_ℓ"

$\leftarrow$ arrow from $d(k,\ell)$ means "Delete x_k"

$\swarrow$ arrow from $d(k,\ell)$ means $\begin{cases} \text{if } x_k = y_\ell, \text{ do nothing} \\ \text{else, "Substitute } x_k \text{ for } y_\ell\text{"} \end{cases}$

Finding the set of edits

Table of $d(k, \ell)$:



Out-degree is 1 of every vertex.

Tree from all squares to the root (0,0)

consisting of $\leftarrow$, $\downarrow$, $\swarrow$ edges

Optimal edit path algorithm

- Generate tables:
  - Create $(n+1) \times (m+1)$ tables d, p.
  - Set $d(k,0) \leftarrow k, d(0,\ell) \leftarrow \ell$ and $p(k,0) \leftarrow (k-1,0), p(0,\ell) \leftarrow (0,\ell-1)$ for $k \in [n], \ell \in [m]$.
  - For $k \leftarrow 1$ to n and for $\ell \leftarrow 1$ to m
    - Compute $d(k,\ell)$ recursively and identify parent p of (k,ℓ).

Optimal edit path algorithm

- **Produce edit path:**
  - Set $(k, \ell) = \leftarrow (n, m)$
  - While $(k, \ell) \neq (0,0)$
    - If $p(k, \ell) = (k-1, \ell-1)$ and $x_k \neq y_\ell$, print "Substitute x_k for y_ℓ"
    - If $p(k, \ell) = (k-1, \ell)$, print "Delete x_k"
    - If $p(k, \ell) = (k, \ell-1)$, print "Insert y_ℓ"
    - Set $(k, \ell) \leftarrow p(k, \ell)$

Follow path from (n, m) back to $(0,0)$ and find the edits along the way

- If $p(k, l) = (k-1, l-1)$ and $x_k = y_l$, then no edit is required for the last character

Edit distance runtime

- Generating tables subroutine runs in $O(nm)$ time
- The path from (n, m) to $(0,0)$ has length at most $n + m$. Total time to print the edit distance is $O(n + m)$.
- Total runtime is still $O(nm)$.

General dynamic programming algorithm

- **Iterate through subproblems**: Starting from the “smallest” and building up to the “biggest.” For each one:
  - Find the optimal value, using the previously-computed optimal values to smaller subproblems.
  - Record the choices made to obtain this optimal value. (If many smaller subproblems were considered as candidates, record which one was chosen.)
- **Compute the solution**: We have the value of the optimal solution to this optimization problem but we don’t have the actual solution itself. Use the recorded information to actually reconstruct the optimal solution.

General dynamic programming runtime

$$\text{Runtime} = (\text{Total number of subproblems}) \times \begin{pmatrix} \text{Time it takes to solve problems} \\ \text{given solutions to subproblems} \end{pmatrix}$$