Dynamic Programming

Matrix Traversal and Counting
Edit Distance,
Longest Common Subsequence,
Longest Increasing Subsequence

CSE 3318 – Algorithms and Data Structures
University of Texas at Arlington

Alexandra Stefan
(Includes images, formulas and examples from CLRS, Dr. Bob Weems, wikipedia)

Dynamic Programming (DP) - CLRS

- Dynamic programming (DP) applies when a problem has both of these properties:
  1. **Optimal substructure**: "optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently".
  2. **Overlapping subproblems**: "a recursive algorithm revisits the same problem repeatedly".
- Dynamic programming is typically used to:
  - Solve *optimization problems* that have the above properties.
  - Solve *counting problems* –e.g. Stair Climbing or Matrix Traversal.
  - *Speed up* existing recursive implementations of problems that have overlapping subproblems (property 2) – e.g. Fibonacci.
- Compare **dynamic programming** with **divide and conquer**, if covered.

Iterative or Bottom-Up
Dynamic Programming

- Main type of solution for DP problems
- Define the problems size and solve problems from size 0 going up to the size we need.
- “Iterative” – because it uses a loop
- “Bottom-up” because you solve problems from the bottom (the smallest problem size) up to the original problem size.

Steps for iterative (bottom up) solution

1. Identify trivial problems
    1. typically where the size is 0
2. Look at the **last step/choice in an optimal solution**:
    1. Assuming an optimal solution, what is the last action in completing it?
    2. Are there more than one options for that last action?
    3. If you consider each action, what is the smaller problem that you would combine with that last action?
    4. Generate all these answers
    5. Compute the value (gain or cost) for each of these answers.
    6. Keep the optimal one (max or min based on problem)
3. Make a 1D or 2D array and start filling in answers from smallest to largest problems.

Other types of solutions:

1. Brute force solution
2. Recursive solution (most likely exponential and inefficient)
3. Memoized solution

   ("memorized", not "memorized")

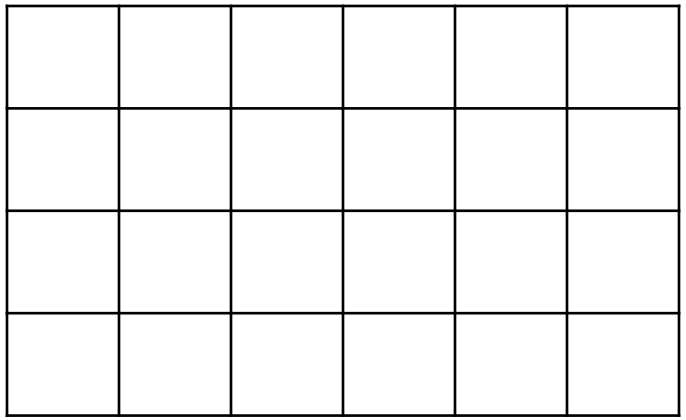
2D Matrix Traversal

P1. **Count** all possible ways to traverse a 2D matrix.

- Start from top left corner and reach bottom right corner.
- You can only move: 1 step to the right or one step down at a time. (No diagonal moves).
- **62. Unique Paths**
- Variation: Add obstacles (cannot travel through certain cells): **63. Unique Paths II**
- Variation: Allow to move in the diagonal direction as well.

P2. Add fish of various gains. Take path that gives **the most gain**.

- Variation: Add obstacles.
- Variation: minimization pb: **64. Minimum Path Sum**
- How about this? **174. Dungeon Game**



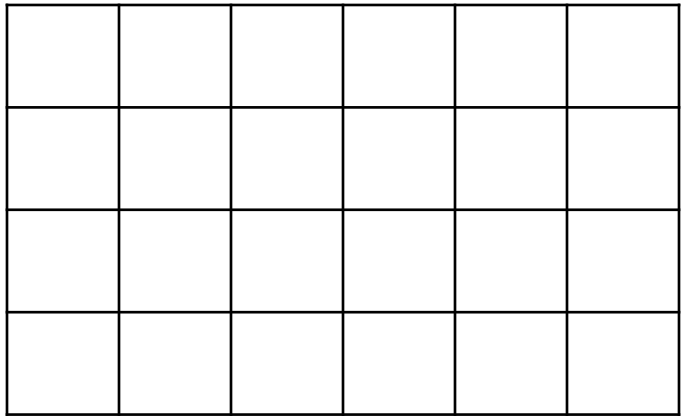
2D Matrix Traversal

P1. **Count** all possible ways to traverse a 2D matrix.

- Start from top left corner and reach bottom right corner.
- You can only move: 1 step to the right or one step down at a time. (No diagonal moves).
- **62. Unique Paths**
- Variation: Add obstacles (cannot travel through certain cells): **63. Unique Paths II**
- Variation: Allow to move in the diagonal direction as well.

P2. Add fish of various gains. Take path that gives **the most gain**.

- Variation: Add obstacles.
- Variation: minimization pb: **64. Minimum Path Sum**
- How about this? **174. Dungeon Game**



2D Matrix Traversal

P1. **Count** all possible ways to traverse a 2D matrix.

- Start from top left corner and reach bottom right corner.
- You can only move: 1 step to the right or one step down at a time. (No diagonal moves).

1	1	1	1	1	1
1	2	3	4	5	6
1	3	6	10	15	21
1	4	10	20	35	56

64. Minimum Path Sum

Longest Common Subsequence (LCS)

Longest Common Subsequence (LCS)

- Application: compute similarity of DNA strands
- Given 2 sequences, find the longest common subsequence (LCS).
  - a subsequence is a sequence that appears in the same order, but not necessarily in consecutive positions.
- Example:
  - A B C B D A B
  - B D C A B A
- Examples of subsequences of the above sequences:
  - BCBA (length 4)
  - BDAB
  - CBA (length 3)
  - CAB
  - BB (length 2)

Show the components of the solution. Can you show a solution similar to that of an Edit distance problem?

LCS
Smaller Problems

- Original problem:

  A B C B D A B

  B D C A B A

- Smaller problems:

- Smaller problems that can be base cases:

Base cases and smaller problems

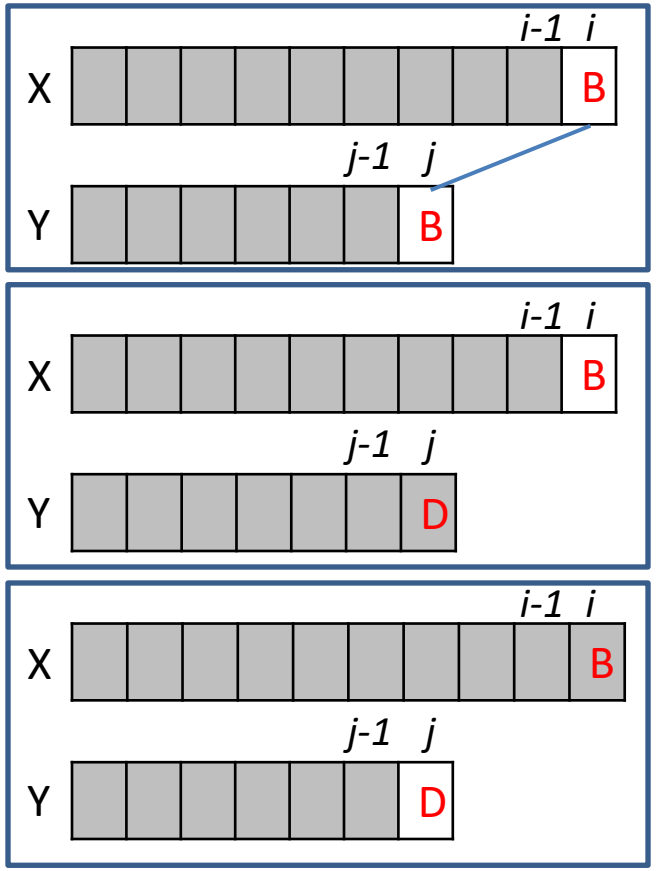
Original problem (LCS length)	A B C B D A B B D C A B A (4)				
Smaller problems (LCS length)	"ABCB" "BD" (1)	"AB" "DC" (0)			
Smaller problems that can be base cases (LCS length)	"" "" (0)	"" "B" (0)	"" "BDCABA" (0)	"A" "" (0)	"ACBDAB" "" (0)

Dependence on Subproblems (recursive case)

c(i,j) – depends on

c(i-1,j-1), c(i-1, j), c(i,j-1)

(grayed areas show solved subproblems)



The function below clearly shows the dependence on the smaller problems and that the optimal value of all possibilities is kept. I would use this one!

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

The function below is equivalent, but one should prove that before using it (or verify that it was proved).

Textbook version:

$$c(i,j) = \begin{cases} 0, & i = 0 \text{ or } j = 0 \\ c(i-1,j-1)+1, & x_{i-1} = y_{j-1}, i,j > 0 \\ \max\{c(i-1,j), c(i,j-1)\}, & x_{i-1} \neq y_{j-1}, i,j > 0 \end{cases}$$

⇒ *c(i-1,j-1) + 1, if* $x_{i-1} = y_{j-1}$
This case makes the solution grow
(finds an element of the subsequence)

⇒ *c(i-1,j)*
x_{i-1} *is ignored*

⇒ *c(i,j-1)*
y_{j-1} *is ignored*

Here indexes start from 1

Longest Common Subsequence – textbook version

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0								
1	A							
2	B							
3	C							
4	B							
5	D							
6	A							
7	B							

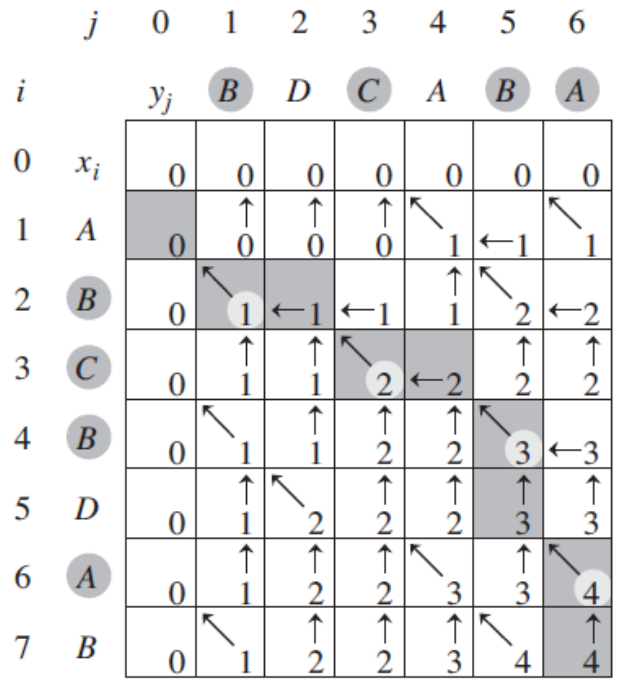
$$c(i,j) = \begin{cases} 0, & i = 0 \; or \; j = 0 \\ c(i-1,j-1)+1, & x_{i-1} = y_{j-1}, i,j > 0 \\ \max\{c(i-1,j), c(i,j-1)\}, & x_{i-1} \neq y_{j-1}, i,j > 0 \end{cases}$$

CLRS – table and formula

For a visualization go to [Data Structure Visualization Longest Common Subsequence]
And enter the words BDCABA and ABCBDAB.

N=strlen(X), P = strlen(Y)
Time Complexity: O( )
Space Complexity: O( )

Longest Common Subsequence – textbook version



i		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	←1	↖ 1
2	B		0	↖ 1	←1	←1	↑ 1	↖ 2	←2
3	C		0	↑ 1	↑ 1	↖ 2	←2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	←3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

$$c(i,j) = \begin{cases} 0, & i = 0 \text{ or } j = 0 \\ c(i-1,j-1)+1, & x_{i-1} = y_{j-1}, i,j > 0 \\ \max\{c(i-1,j), c(i,j-1)\}, & x_{i-1} \neq y_{j-1}, i,j > 0 \end{cases}$$

CLRS – table and formula

For a visualization go to [Data Structure Visualization Longest Common Subsequence]
And enter the words BDCABA and ABCBDAB.

N=strlen(X), P = strlen(Y)
Time Complexity: O( )
Space Complexity: O( )

Iterative solution

$$c(i,j) = \begin{cases} 0, & i = 0 \; or \; j = 0 \\ c(i-1,j-1)+1, & x_{i-1} = y_{j-1}, i,j > 0 \\ \max\{c(i-1,j), c(i,j-1)\}, & x_{i-1} \neq y_{j-1}, i,j > 0 \end{cases}$$

```
LCS_length(X,Y)
  N = length(X)
  P = length(Y)
  b = 2D array of N+1 rows, P+1 columns
  c = 2D array of N+1 rows, P+1 columns
  for i = 0 to N
    c[i,0] = 0
  for j = 0 to P
    c[0,j] = 0
  for i = 1 to N
    for j = 1 to P
      if x_{i-1} == y_{j-1}
        c[i,j] = c[i-1,j-1]+1
        b[i,j] = \   // diagonal
      else if c[i-1,j]≥c[i,j-1]
        c[i,j] = c[i-1,j]
        b[i,j] = ^   // up arrow
      else
        c[i,j] = c[i,j-1]
        b[i,j] = <   // left arrow
```

CLRS – pseudocode

N=strlen(X), P = strlen(Y)
Time Complexity: O(NP)
Space Complexity: O(NP)

Recover the subsequence

CLRS pseudcode

// to start it: print_LCS(b,X, ___, ____)

```
print_LCS(b,X,i,j)
  if i==0 or j==0
    return
  if b[i,j]==\    // diagonal arrow
    print_LCS(b,X,i-1, j-1)
    print(x_{i-1})
  else if (b[i,j]==^)  // up arrow
    print_LCS(b,X,i-1, j)
  else                 //left arrow
    print_LCS(b,X,i, j-1)
```

N=strlen(X), P = strlen(Y)

Time Complexity: O( )

Space Complexity: O( )

Longest Increasing Subsequence (LIS)

Longest Increasing Subsequence

Given an array of values, find the longest increasing subsequence.
Example: A = { 3,6,3,1,4,3,4}

Variations:

Repetitions allowed: increasing subsequence. E.g.: 3,3,4,4 (also ok 3,3,3,4)

Repetitions **not** allowed: strictly increasing subsequence. E.g.: 1,3,4

Simple solution: ***reduce it to a LCS problem.***
(For a more efficient solution tailored for the LIS problem see Dr. Weems notes.)

A = { 3,6,3,1,4,3,4}

Repetition allowed:

X = {1,3,3,3,4,4,6}

	3	6	3	1	4	3	4

Repetition not allowed:

X = {1,3,4,6} (sorted and unique)

3	6	3	1	4	3	4

Time and space of LCS(X,A) (dominates that of sorting) for both methods:

TC: ____

SC: ____ (depends on if only length needed or also subsequence needed and on LCS implementation)

A = {3,6,3,1,4,3,4}

Repetition allowed: X = {1,3,3,3,4,4,6} (A sorted)

LCS({1,3,3,3,4,4,6}, {3,6,3,1,4,3,4})

length ***4***, subsequence: {***3,3,3,4***}

		3	6	3	1	4	3	4
	0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1	1
3	0	1	1	1	1	1	2	2
3	0	1	1	2	2	2	2	2
3	0	1	1	2	2	2	3	3
4	0	1	1	2	2	3	3	4
4	0	1	1	2	2	3	3	4
6	0	1	2	2	2	3	3	4

Repetition not allowed: X = {1,3,4,6} (A sorted and unique)

LCS({1,3,4,6}, {3,6,3,1,4,3,4})

length: ***3***, subsequence: {***1,3,4***}

		3	6	3	1	4	3	4
	0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1	1
3	0	1	1	1	1	1	2	1
4	0	1	1	1	1	2	2	3
6	0	1	2	2	2	2	2	3

Time and space of LCS(X,A) (dominates that of sorting) for both methods:

TC: $O(n^2)$

SC: $O(n^2)$

LIS to LCS reduction

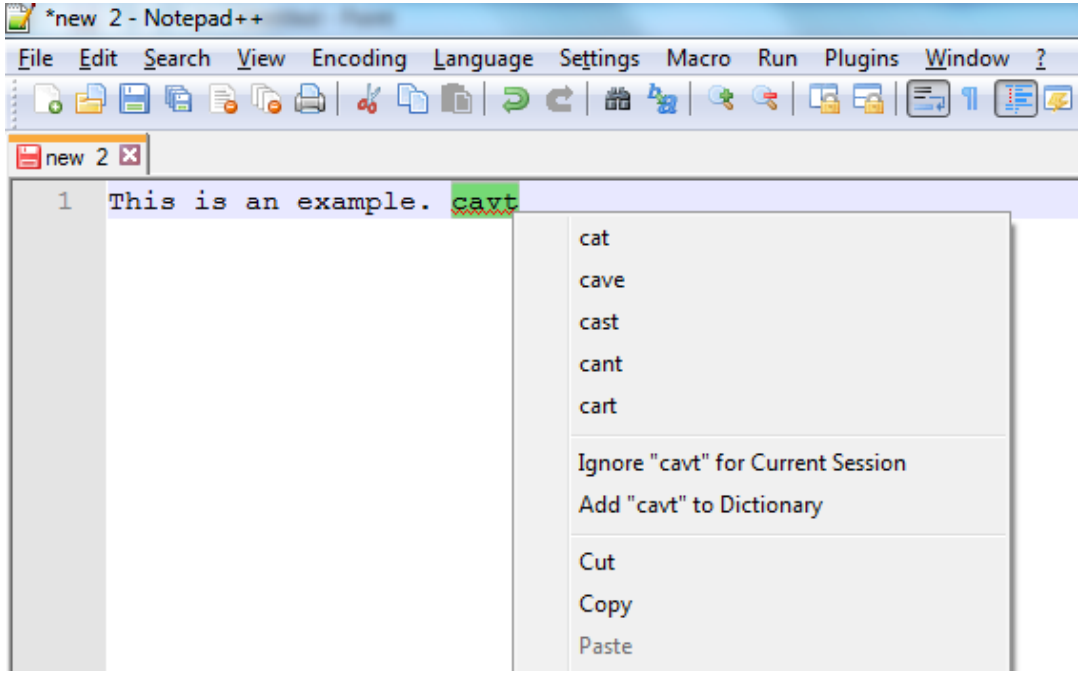
- A = {3,6,3,1,4,3,4} of size n
- Time and space of LCS(X,A) (dominates that of sorting) for both methods below
- LIS with repetitions:
  - produce sorted copy of A: X = {1,3,3,3,4,4,6}
  - LIS(A) = LCS(X,A)
  - Time complexity: $\Theta(n^2)$ (copy A and sort in nlgn + solve LCS in $\Theta(n^2)$)
  - Space complexity: $\Omega(n)$, $O(n^2)$
    - $O(n^2)$ if saving 2D table for LCS or
    - O(n) if saving only 2 rows for LCS. (This can be used if only the length is needed)
- LIS with NO repetitions:
  - produce sorted copy of unique elements in A: X = {1,3,4,6}
  - LIS(A) = LCS(X,A)
  - Time complexity: $O(n^2)$ (copy A and sort in nlgn + solve LCS in $O(n^2)$)
  - Space complexity: $\Omega(n)$, $O(n^2)$
    - $O(n^2)$ if saving 2D table for LCS or
    - O(n) if saving only 2 rows for LCS. (This can be used if only the length is needed)

Edit Distance/Levenshtein Distance

- Problem : given two strings, produce a number that reflects how different they are
- Applications:
  - auto-grade fill-in the blank questions in Canvas
  - Spell checker
- Intuition – alignment, pairs and pair cost
- Method:
  - Create table, fill in top row, fill in leftmost column, fill in remaining cells top-down and left-right
  - Time complexity
  - Space complexity
  - ***Dynamic Programming type of solution***
    - *Solution to current size problem is computed from solutions to smaller size problems*
    - *Smallest problems -> easy solution*
    - *Solve all problems from smallest size to current size*
  - Space improvement
- Recover the alignment - covered if time permits

The Edit Distance

Application: Spell-checker



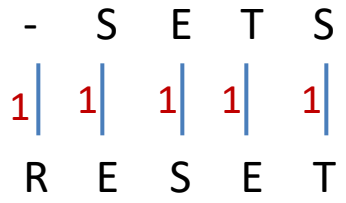
Edit distance

- ***Minimum cost*** of all possible alignments between two words.
- Identical words have distance 0
- Examples:
  - Dist("cat", "cat") -> 0
  - Dist("cat", "bat") -> 1
  - Dist("cat", "cats") -> 1 (insertion in word2)
  - Dist("cat", "at") -> 1 (insertion in word 1)
  - Dist("cat", "dogs") -> 4
  - Dist("cat", "set") -> 2

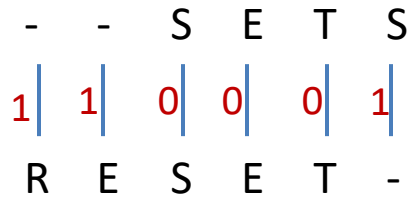
- Spell checker - computes the "edit distance" between the words. The smaller distance, the more similar the words. Returns all dictionary words that are at the smallest distance from misspelled word.
- Other applications: autograding (match answer), search by title
- This is a specific case of a more general problem: time series alignment.
- A related problem is Subsequence Search (find if a smaller string is part of a long one; not exact match)

Alignments

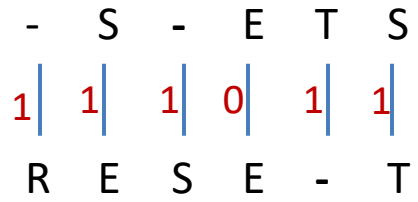
Examples of different alignments for the same words



Cost/distance: 5



Cost/distance: 3



Cost/distance: 5

- No cross-overs: The letters must be in the order in which they appear in the string.

Incorrect alignment

Pair cost:
- Same letters: 0
- Different letters: 1
- Letter-to-dash: 1 (a dash indicates insertion in that word)

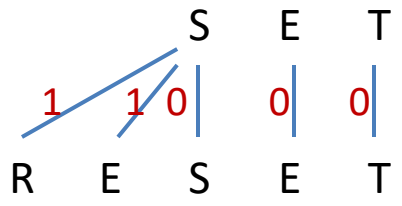
Alignment cost: sum of costs of all pairs in the alignment.

Edit distance: minimum alignment cost over all possible alignments.

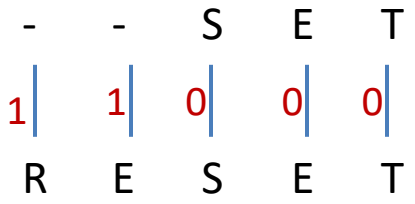
(We will compute the cost/distance, without explicitly generating the alignments)

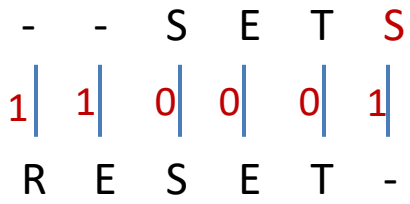
The Edit Distance

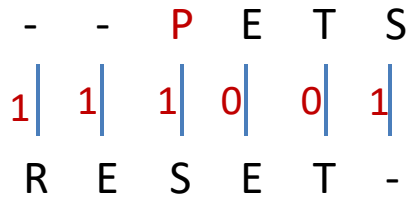
- Edit distance – the cost of the best alignment
  - Minimum cost of all possible alignments between two words.
  - (The smaller distance, the more similar the words)



Edit distance: minimum alignment cost over all possible alignments.
Alignment cost: sum of costs of all pairs in the alignment.
Pair cost:
Same letters: 0
Different letters: 1
Letter to dash: 1







Notations, Subproblems

- Notation:
  - $X = x_0,x_1,x_2,\ldots,x_n$
  - $Y = y_0,y_1,y_2,\ldots,y_p$
  - Dist(i,j) = the smallest cost of all possible alignments between substrings $x_0,x_1,x_2,\ldots,x_i$ and $y_0,y_1,y_2,\ldots,y_j$.
  - Dist(i,j) will be recorded in a matrix at cell [i,j].
- Subproblems of ("SETS", "RESET"):
  - Problem size can change by changing either X or Y (from two places):
  -
  -
  -
  -
- What is Dist for all of the above problems?

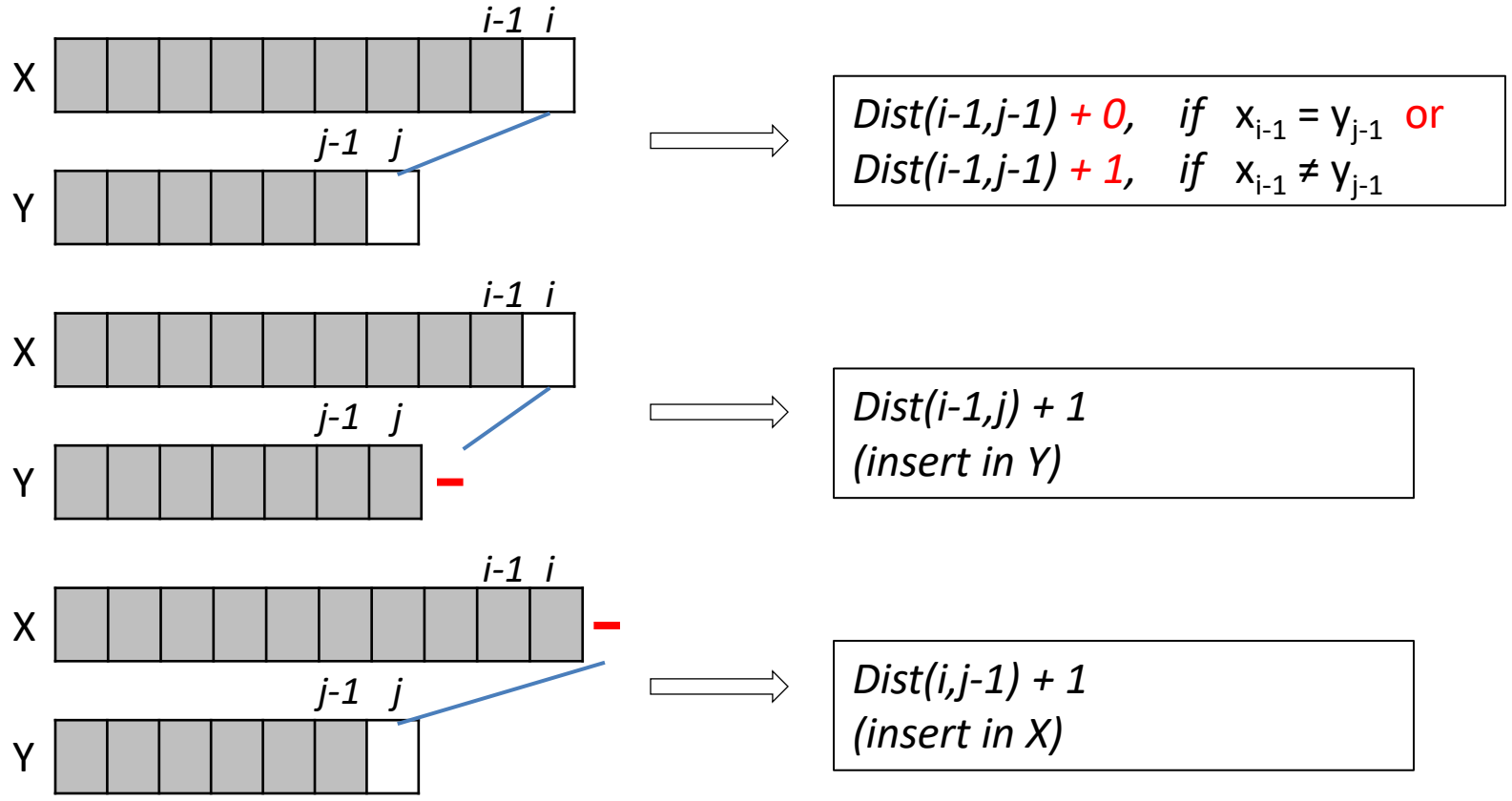
Notations, Subproblems

- Subproblems of ("SETS", "RESET"):
  - Problem size can change by changing either X or Y (from two places):
  - ("S", "RES")
  - ("", "R"), ("", "RE"), ("", "RES"), …, ("", "RESET")
  - ("S", ""), ("SE", ""), ("SET", ""), ("SETS", "")
  - ("" , "")
- What is Dist for all of the above problems?
- Notation:
  - $X = x_0,x_1,x_2,\ldots,x_n$
  - $Y = y_0,y_1,y_2,\ldots,y_p$
  - Dist(i,j) = the smallest cost of all possible alignments between substrings $x_0,x_1,x_2,\ldots,x_i$ and $y_0,y_1,y_2,\ldots,y_j$.
  - Dist(i,j) will be recorded in a matrix at cell [i][j].

Dependence on Subproblems

Dist(i,j) – depends on *Dist(i-1,j-1), Dist (i-1, j), Dist(i,j-1)*

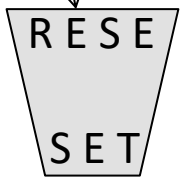
- *Below, grayed areas show the solved subproblems*
- *We have 3 options for the **last pair** in an alignment up to index i (in X) and j (in Y)*



Edit Distance: Filling out the distance matrix

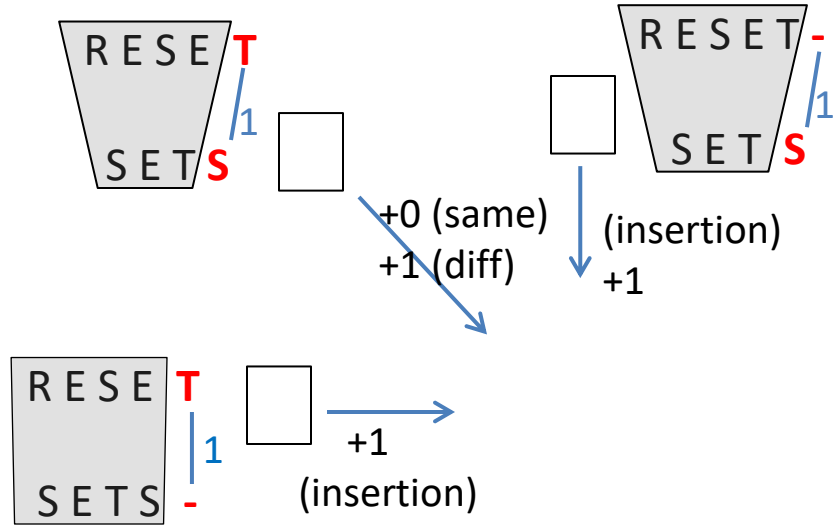
- Each cell will have the answer for a specific subproblem.
- Special cases:
  - Dist(0,0) =
  - Dist(0,j) =
  - Dist(i,0) =
  - Dist(i,j) =

Represents an optimal alignment between "RESE" and "SET"



- Complexity (where: |X| = n, |Y| = p): Time: Space:

		0 ""	1 R	2 E	3 S	4 E	5 T
0	""						
1	S						
2	E						
3	T						
4	S						

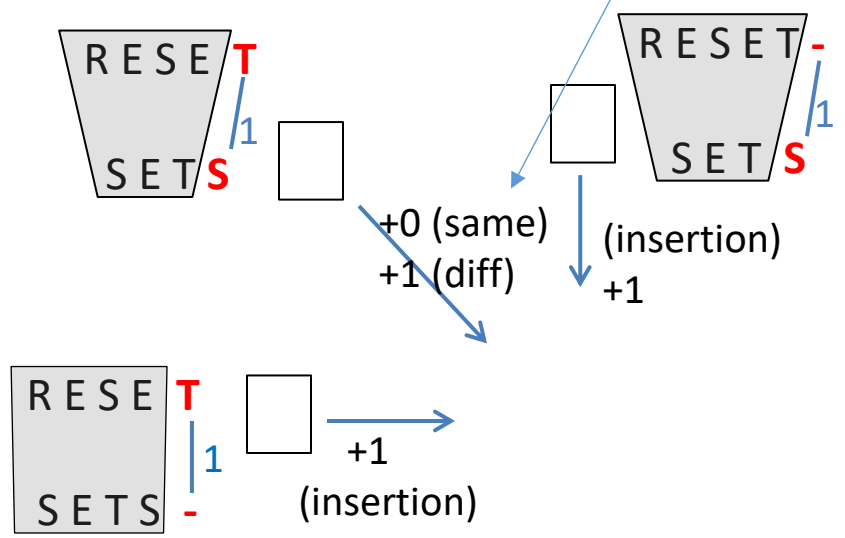


Edit Distance – Cost function

- Each cell will have the answer for a specific subproblem.
- Special cases:
  - Dist(0,0) = 0
  - Dist(0,j) = 1 + Dist(0,j-1)
  - Dist(i,0) = 1 + Dist(i-1,0)
  - Dist(i,j) = $\begin{cases} \min\{ \text{Dist}(i-1,j)+1,\ \text{Dist}(i,j-1)+1,\ \text{Dist}(i-1,j-1) \} & \text{if } x_{i-1} = y_{j-1} \text{ or} \\ \min\{ \text{Dist}(i-1,j)+1,\ \text{Dist}(i,j-1)+1,\ \text{Dist}(i-1,j-1)+1 \} & \text{if } x_{i-1} \neq y_{j-1} \end{cases}$
- Complexity (where: |X| = n, |Y| = p): Time: O(n*p) Space: O(n*p)

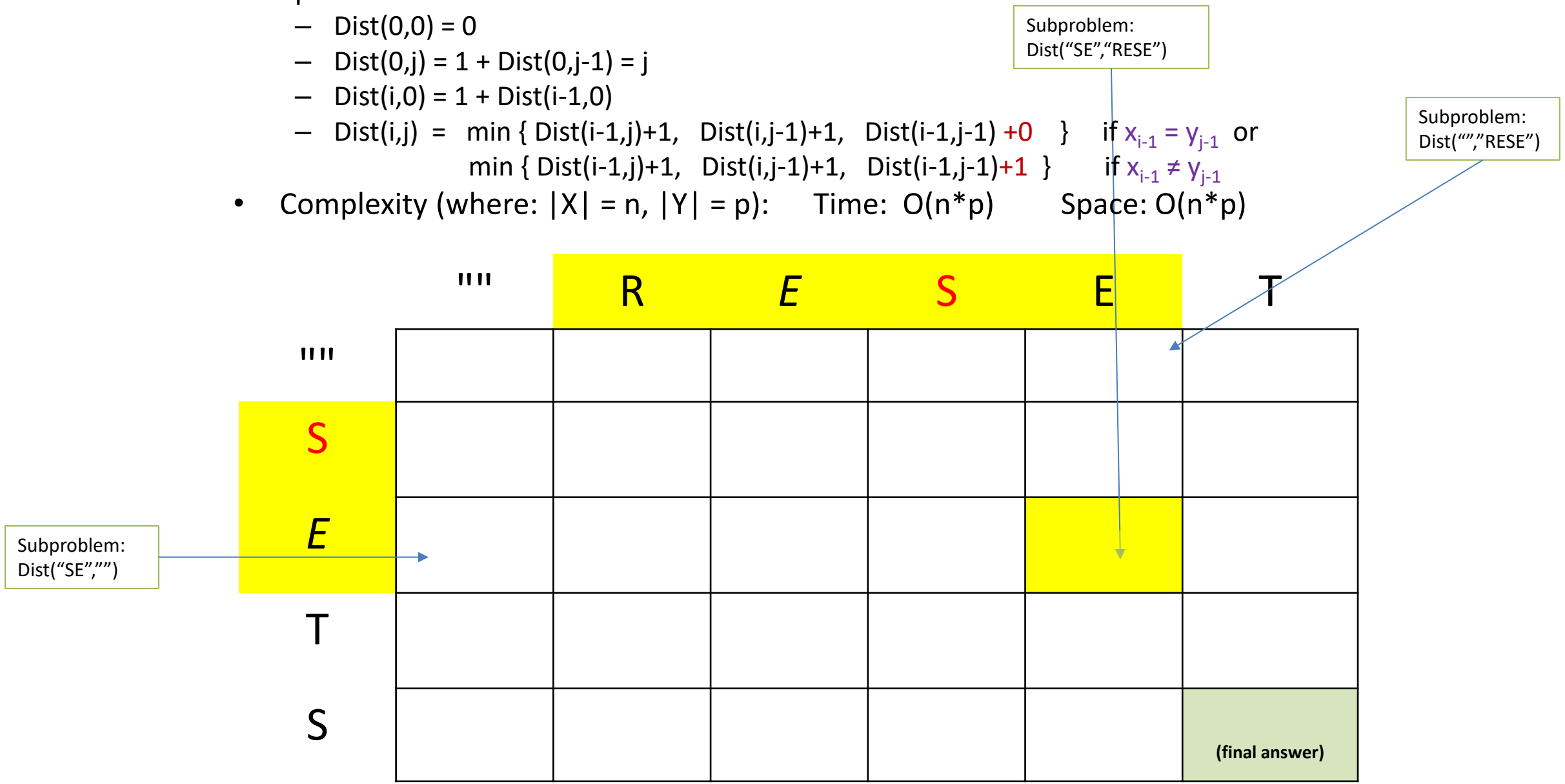
NOTE: Use this definition where for Dist(i,j) the min of the 3 possible smaller problems is used regardless of how letters x_{i-1} and y_{j-1} compare.

		0	1	2	3	4	5
		""	R	E	S	E	T
0	""	**0**	**1**	**2**	**3**	**4**	**5**
1	S	**1**	**1**	**2**	**2**	**3**	**4**
2	E	**2**	**2**	**1**	**2**	**2**	**3**
3	T	**3**	**3**	**2**	**2**	**3**	**2**
4	S	**4**	**4**	**3**	**2**	**3**	**3**



Worked out example

- Each cell will have the answer for a specific subproblem.
- Special cases:
  - Dist(0,0) = 0
  - Dist(0,j) = 1 + Dist(0,j-1) = j
  - Dist(i,0) = 1 + Dist(i-1,0)
  - Dist(i,j) = min { Dist(i-1,j)+1, Dist(i,j-1)+1, Dist(i-1,j-1) +0 } if $x_{i-1} = y_{j-1}$ or
    min { Dist(i-1,j)+1, Dist(i,j-1)+1, Dist(i-1,j-1)+1 } if $x_{i-1} \neq y_{j-1}$
- Complexity (where: |X| = n, |Y| = p): Time: O(n*p) Space: O(n*p)



Worked out example

- Each cell will have the answer for a specific subproblem.
- Special cases:
  - Dist(0,0) = 0
  - Dist(0,j) = 1 + Dist(0,j-1) = j
  - Dist(i,0) = 1 + Dist(i-1,0)
  - Dist(i,j) = min { Dist(i-1,j)+1, Dist(i,j-1)+1, Dist(i-1,j-1) +0 } if $x_{i-1} = y_{j-1}$ or
    min { Dist(i-1,j)+1, Dist(i,j-1)+1, Dist(i-1,j-1)+1 } if $x_{i-1} \neq y_{j-1}$
- Complexity (where: |X| = n, |Y| = p): Time: O(n*p) Space: O(n*p)

Subproblem: Dist("SE","RESE")

Subproblem: Dist("","RESE")

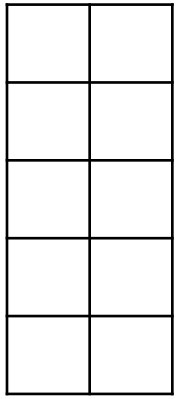
+0 because the corresponding letter is the same (here T)

Subproblem: Dist("SE","")

	""	R	*E*	S	E	T
""	0	0+1=1 1	1+1=2 2	2+1 3	3+1 4	4+1 5
S	0+1 =1 1	0+1=**1** 1+1=**2** 1+1 =**2** 1	1+1=**2** 2+1=**3** 1+1=**2** 2	2+**0** 3+1 2+1 2	3+1 4+1 2+1 3	4+1 5+1 3+1 4
E	1+1=2 2	1+1 1+1 2+1 2	1+**0** 2+1 2+1 1	2+1 2+1 1+1 2	2+**0**(E,E) 3+1 2+1 2	3+1 4+1 2+1 3
T	2+1 3	2+1 2+1 3+1 3	2+1 1+1 3+1 2	1+1 2+1 2+1 2	2+1(T,E) 2+1 2+1 3	2+0 (T,T) 3+1 3+1 2
S	3+1 4	3+1 3+1 4+1 4	3+1 2+1 4+1 3	2+**0** 2+1 3+1 2	2+1 3+1 2+1 3	3+1 2+1 3+1 **3**(final ans)

ED - Improving memory usage: Θ(min{p,n})

- Optimize the memory usage: store only smaller problems that are needed.
  - Store either 2 rows or 2 columns
  - the choices cannot be recovered anymore (i.e. cannot recover what items to pick to achieve the computed optimal value).



- Space complexity: Θ(min{ p, n })

- Practice:
  - Can you implement this solution?

Motivation for Edit Distance

- The Edit Distance is a ***Time Series Alignment***
- Other examples of problems solved with Time Series Alignment:
  - Given observed temperatures, find location:
    - Collected in a database temperatures at different hours over one day in various places (labelled with the name). Given a query consisting of temperatures collected in an unknown place, find the place with the most similar temperatures. Issues:
      - Not same number of measurements for every place and query.
  - Measure similarity of signs in videos of sign language => find videos of similar signs
  - Find shapes in images (after image processing extracted relevant features)
- Find a substring in a string
  - E.g. swear words in Pokemon Names
  - Uses two additional sink states (at the beginning and end of the small query)

Sample Exam Problem

On the right is part of an edit distance table. CART is the complete second string. AL is the end of the first string (the first letters of this string are not shown).

a. (6 points) Fill-out the empty rows (finish the table).

b. (4 points) How many letters are missing from the first string (before AL)? Justify your answer.

c. (8 points) Using the table and the information from part b), for each of the letters **C and A** in the second string, CART, say if it could be one of the missing letters of the first string: **Yes** (it is one of the missing letters – 'proof'), **No** (it is not among the missing ones – 'proof'), **Maybe** (it may or may not be among the missing ones – give example of both cases).

   – C: Yes/No/Maybe. Justify:

   – A: Yes/No/Maybe. Justify:

		C	A	R	T
…	…	…	…	…	…
…	5	5	4	3	3
A					
L					

Recover the alignment for Edit Distance

Not covered

Edit Distance
Recover the alignment – Worksheet
(using the arrow information)

X = SETS
Y = RESET

Time complexity: O(……..)
(where: |X| = n, |Y| = p)

	Aligned Pair	Update
↖	x_{i-1} y_{j-1}	i = i-1 j = j-1
↑	x_{i-1} -	i = i-1
←	- y_{j-1}	j = j-1

i	j	0	1	2	3	4	5
		""	R	E	S	E	T
0	""	↖ 0	← 1	← 2	← 3	← 4	← 5
1	S	↑ 1	↖ 1	↖ 2	↖ 2	← 3	← 4
2	E	↑ 2	↖ 2	↖ 1	← 2	↖ 2	← 3
3	T	↑ 3	↖ 3	↑ 2	↖ 2	↖ 3	↖ 2
4	S	↑ 4	↖ 4	↑ 3	↖ 2	↖ 3	↑ 3

i							
j							
X							
Y							

Start at:
i = …….
j = …….

How big will the solution be (as num of pairs)?

Edit Distance

Recover the alignment

Here the pairs are filled in from the LEFT end to the RIGHT end and printed from RIGHT to LEFT.

X = SETS
Y = RESET

Time complexity: O(n+p)
(where: |X| = n, |Y| = p)

	j	0	1	2	3	4	5
i		""	R	E	S	E	T
0	""	↖ 0	← 1	← 2	← 3	← 4	← 5
1	S	↑ 1	↖ 1	↖ 2	↖ 2	← 3	← 4
2	E	↑ 2	↖ 2	↖ 1	← 2	↖ 2	← 3
3	T	↑ 3	↖ 3	↑ 2	↖ 2	↖ 3	↖ 2
4	S	↑ 4	↖ 4	↑ 3	↖ 2	↖ 3	↑ 3

	Aligned Pair	Update
↖	x_{i-1} y_{j-1}	i = i-1 j = j-1
↑	x_{i-1} -	i = i-1
←	- y_{j-1}	j = j-1

i	4	3	2	1	0	0	0
j	5	5	4	3	2	1	0
	↑	↖	↖	↖	←	←	
X	**S**	**T**	**E**	**S**	-	-	
Y	-	**T**	**E**	**S**	**E**	**R**	
	1	**0**	**0**	**0**	**1**	**1**	

Start at:
i = 4
j = 5

How big will the solution be (as num of pairs)?
n+p

Print from right to left.

Sum of costs of pairs in the alignment string is the same as table[4][5]: 1+0+0+0+1+1 = 3

- What is the best alignment between

abcdefghijk

cdXYZefgh

Edit Distance

Recover the alignment - Method 2:

(based only on distances)

Even if the choice was not recorded, we can backtrace based on the distances: see from what direction (cell) you could have gotten here.

		w	w	a	b	u	d	e	f
	0	1	2	3	4	5	6	7	8
a	1	1	2	2	3	4	5	6	7
b	2	2	2	3	2	3	4	5	6
c	3	3	3	3	3	3	4	5	6
d	4	4	4	4	4	4	3	4	5
e	5	5	5	5	5	5	4	3	4
f	6	6	6	6	6	6	5	4	3
y	7	7	7	7	7	7	6	5	4
y	8	8	8	8	8	8	7	6	5
y	9	9	9	9	9	9	8	7	6

first: abcdefyyy
second: wwabudef

edit distance:
Alignment:

Edit Distance

Recover the alignment - Method 2:

(based only on distances)

Even if the choice was not recorded, we can backtrace based on the distances: see from what direction (cell) you could have gotten here.

		w	w	a	b	u	d	e	f
	0	**1**	**2**	3	4	5	6	7	8
a	1	1	2	**2**	3	4	5	6	7
b	2	2	2	3	**2**	3	4	5	6
c	3	3	3	3	3	**3**	4	5	6
d	4	4	4	4	4	4	**3**	4	5
e	5	5	5	5	5	5	4	**3**	4
f	6	6	6	6	6	6	5	4	**3**
y	7	7	7	7	7	7	6	5	**4**
y	8	8	8	8	8	8	7	6	**5**
y	9	9	9	9	9	9	8	7	**6**

first: abcdefyyy
second: wwabudef

edit distance: 6
Alignment:
- - a b c d e f y y y
w w a b u d e f - - -
1 1 0 0 1 0 0 0 1 1 1