Dynamic programming (part 2)

CS 146 - Spring 2017

$$\text{dist}(i, j, \varnothing) = \begin{cases} w(i \to j) & \text{if } i \to j \\ \infty & \text{else} \end{cases}$$

for k = 0 to V-1

$$\text{dist}(i, j, \{0, \ldots, k\}) = \min \begin{cases} \text{dist}(i, j, \{0, \ldots, k-1\}) \\ \text{dist}(i, k, \{0, \ldots, k-1\}) + \text{dist}(k, j, \{0, \ldots, k-1\}) \end{cases}$$

$$\text{dist}(i, j, \varnothing) = \begin{cases} w(i \to j) & \text{if } i \to j \\ \infty & \text{else} \end{cases}$$

$$\text{dist}(i, j, \{0, \ldots, k\}) = \min \begin{cases} \text{dist}(i, j, \{0, \ldots, k-1\}) \\ \text{dist}(i, k, \{0, \ldots, k-1\}) + \text{dist}(k, j, \{0, \ldots, k-1\}) \end{cases}$$

for k = 0 to V-1

represented in 3d array

```
dist[i][j][k] = distance from i to j
 using vertices 0 up to k-1, where k > 0
```

(shift k by 1 to have room for empty set)

Making recursion more time efficient

(by using more space efficient)

	memoization (top-down)	**dynamic programming (bottom up)**
store subproblems in …	dictionary	array (2D if subpb has 2 params)
subproblem input	dictionary key	array index
subproblem output	dictionary value	entry at given index
code structure	recursive	for-loops

Today

- edit distance problem

- pretty printing

Edit distance problem

- **edit:** insert, delete or replace a character

ALGORITHM

ALG**A**RITHM replace O with A

_LGARITHM delete A

L**O**GARITHM insert O

edit(ALGORITHM, LOGARITHM) = 3

Edit distance problem

- **edit:** insert, delete or replace a character

What is the edit distance between …

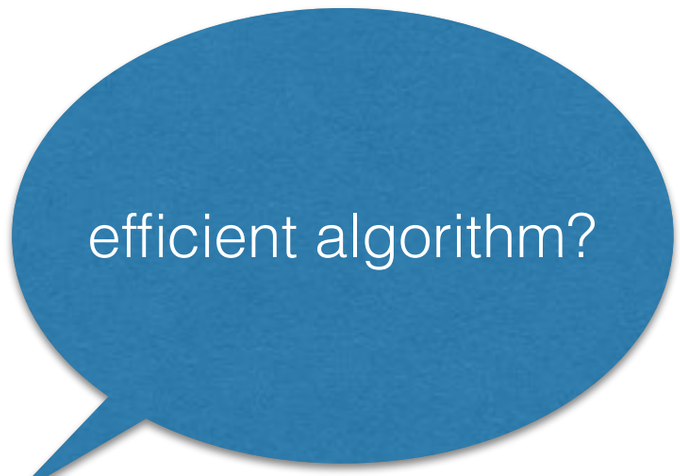
POLYNOMIAL EXPONENTIAL

Edit distance problem

- **edit:** insert, delete or replace a character

What is the edit distance between …

POLYNOMIAL EXPONENTIAL



Step 1: understand the problem

Work out some examples

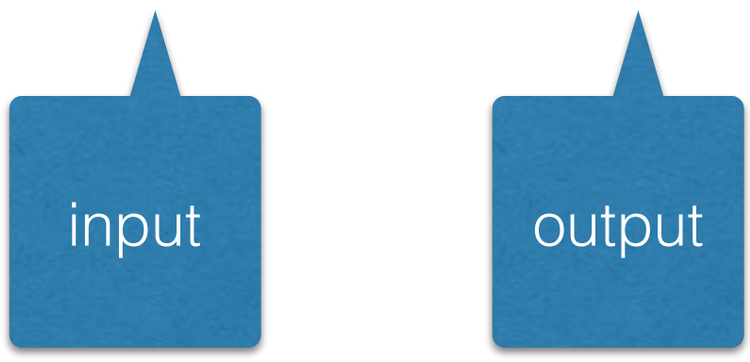
Understand the general input and output

input: 2 strings w1 and w2

output: edit distance between w1 and w2

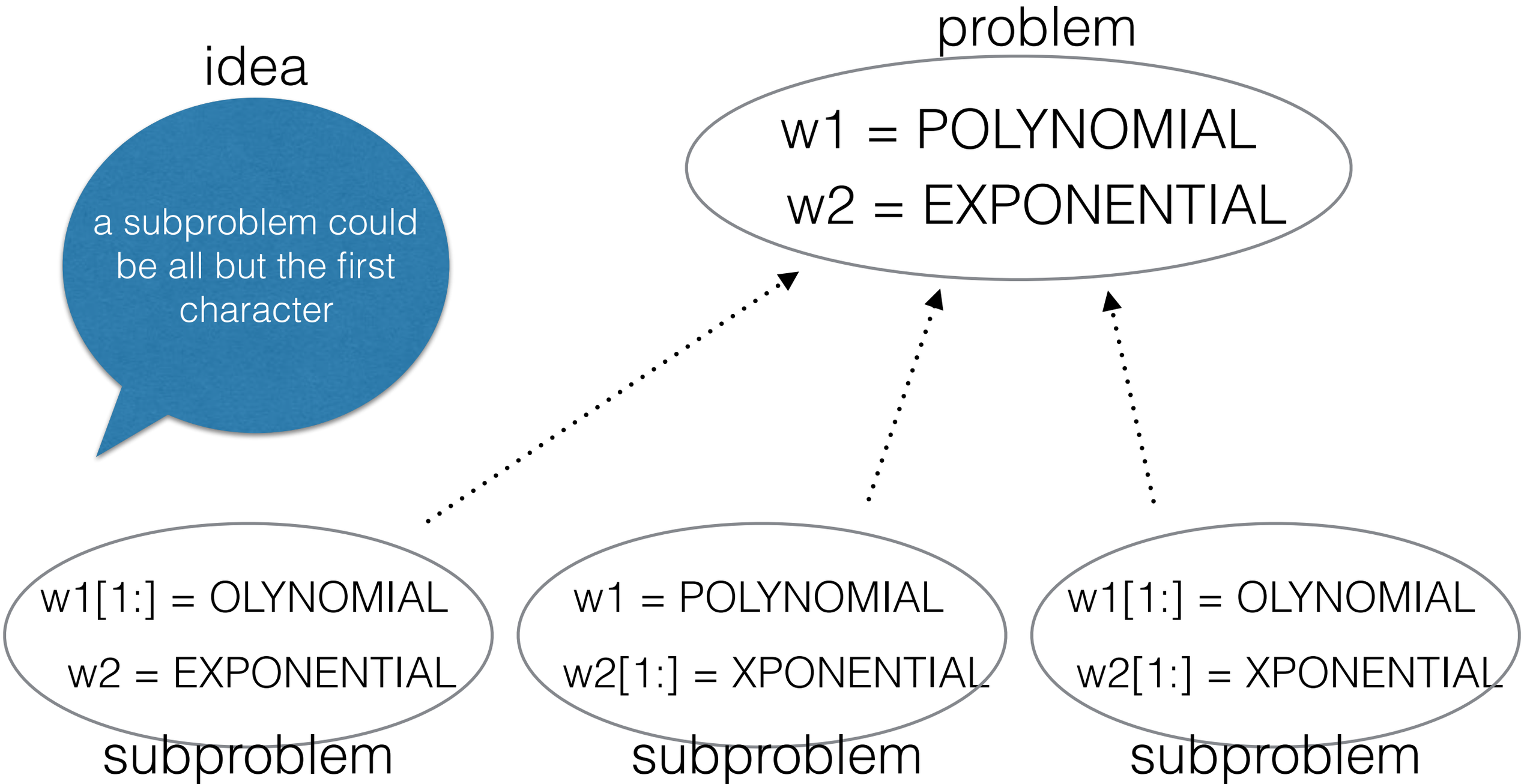
Decide on notation

edit(w1, w2) = an integer



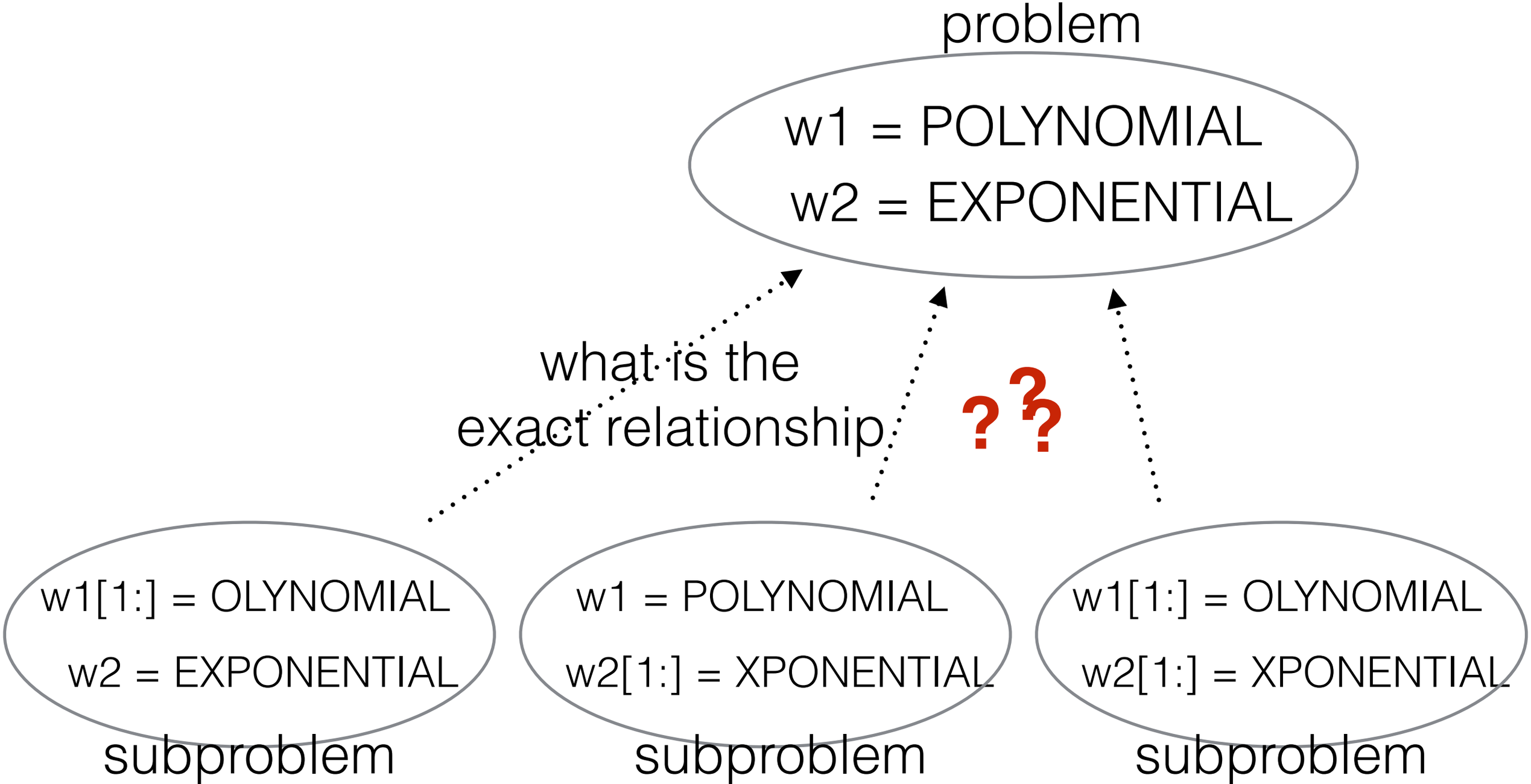
Step 2: come up with a brute force alg with **recursion**

Express the problem in terms of subproblems



Step 2: come up with a brute force alg with **recursion**

Express the problem in terms of subproblems



Step 2: come up with a brute force alg with **recursion**

Express the problem in terms of subproblems

problem

w1 = POLYNOMIAL

w2 = EXPONENTIAL

if first characters are a match

no editing on 1st chars

else

there must be an edit

to make the 1st characters match

w1[1:] = OLYNOMIAL

w2 = EXPONENTIAL

subproblem

w1 = POLYNOMIAL

w2[1:] = XPONENTIAL

subproblem

w1[1:] = OLYNOMIAL

w2[1:] = XPONENTIAL

subproblem

Step 2: come up with a brute force alg with **recursion**

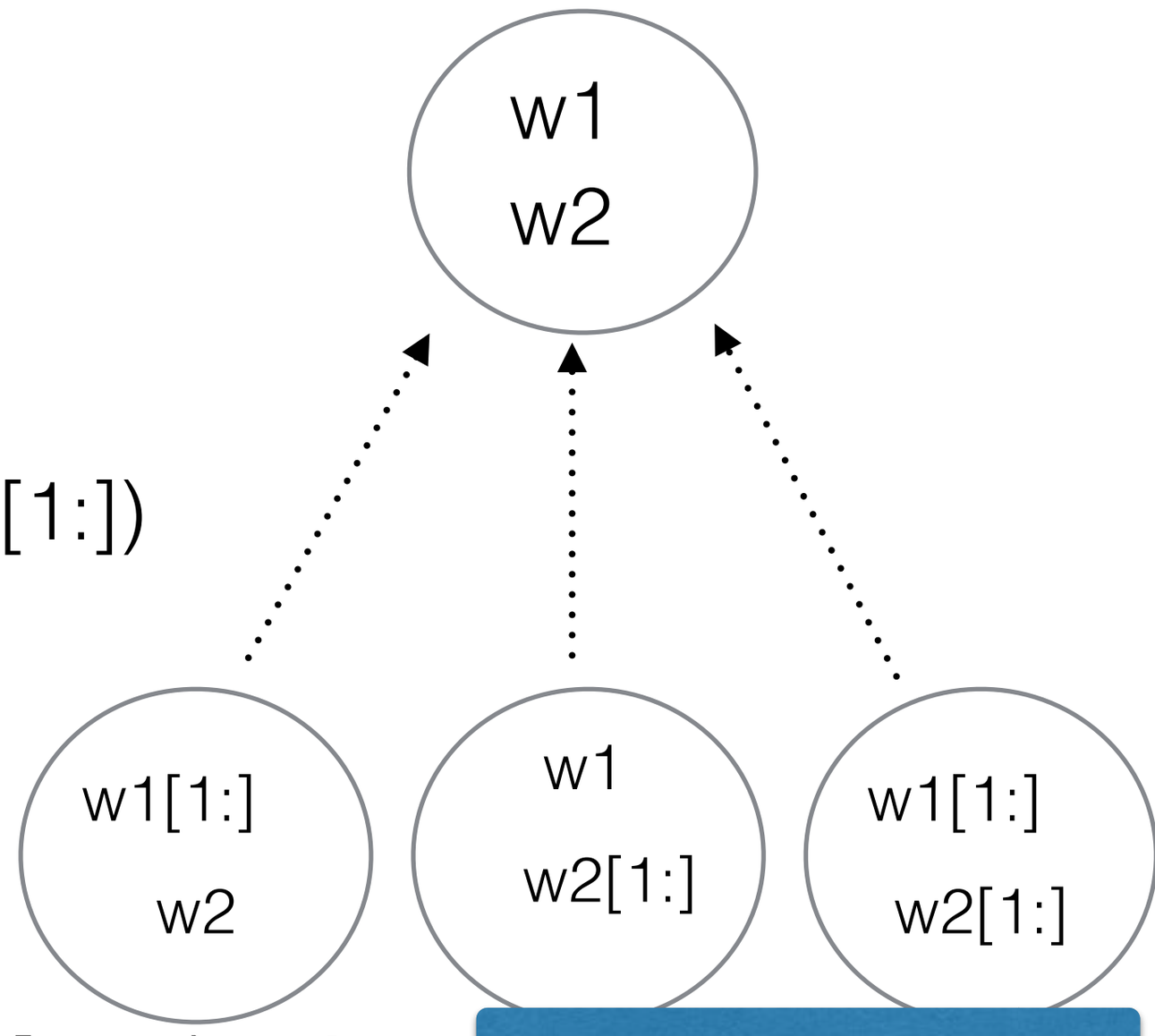
Express the problem in terms of subproblems

if w1[0] = w2[0]

edit(w1, w2) = edit(w1[1:], w2[1:])

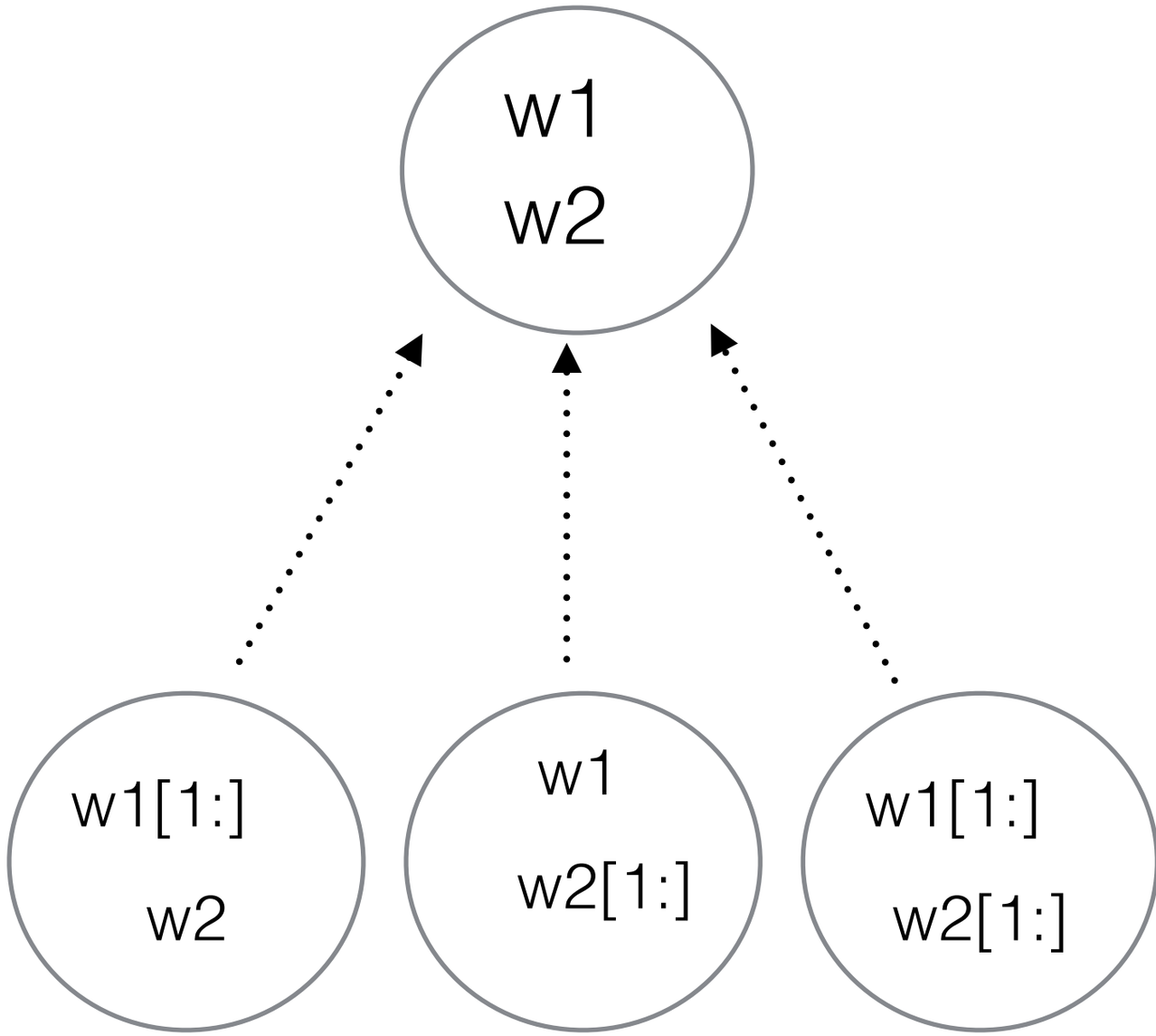
else

edit(w1, w2) = min(edit(w1[1:], w2) **+ 1**,
edit(w1, w2[1:]) **+ 1**,
edit(w1[1:], w2[1:]) **+ 1**)



Step 2: come up with a brute force alg with **recursion**

Express the problem in terms of subproblems



base case: w1 = empty string

edit(w1, w2) = length(w2)

cost of inserting all the letters of w2 into empty string

base case: w2 = empty string

edit(w1, w2) = length(w1)

cost of deleting all the letters of w1 until it becomes an empty string

Step 2: come up with a brute force alg with **recursion**

Express the problem in terms of subproblems

base cases

edit(“”, w2) = length(w2)

edit(w1, “”) = length(w1)

if w1[0] = w2[0]

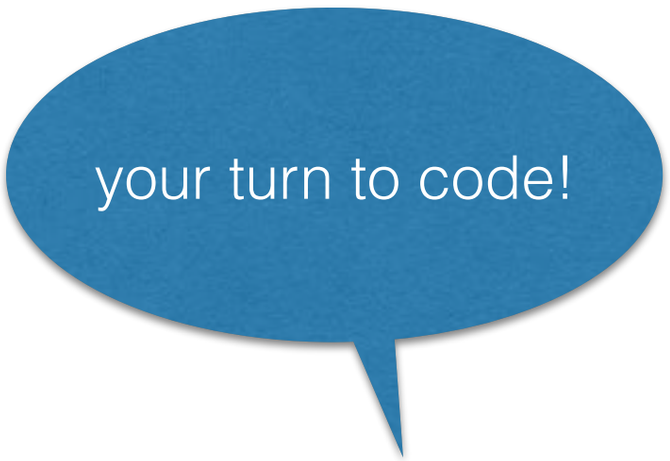
edit(w1, w2) = edit(w1[1:], w2[1:])

else

edit(w1, w2) = min(edit(w1[1:], w2) **+ 1**,
edit(w1, w2[1:]) **+ 1**,
edit(w1[1:], w2[1:]) **+ 1**)

Step 3: optimize!

- option 1: memoization
- option 2: dynamic programming



https://github.com/jnylam/SJSU-cs146-s17

-> Dynamic programming

-> Edit distance

Time and space complexity

- **brute force:** time? space?
- **dynamic programming** (easy to analyze):
  - time: ?
  - space: ?
  - Note: time and and space not necessarily the same
- **memoized** (messier to analyze directly)

Time and space complexity

- **memoized** (messier to analyze directly b/c recursive)
  - every call either does actual work or retrieves from cache
  - # calls that do work = # subproblems = size of DP table, each doing (branching factor) work
  - # calls that retrieve from cache <= #subproblems * branching factor, each doing O(1)
  - **memoize time** = table size * branching factor = **DP time**
  - **memoize space** = map size = #subproblems = table size = **DP space**