CSC3160: Design and Analysis of Algorithms

Week 5: Dynamic Programming

Instructor: Shengyu Zhang

About midterm

- Time: Mar 3, 2:50pm – 4:50pm.
- Place: This lecture room.
- Open book, open lecture notes.
  - But no Internet allowed.
- Scope: First 6 lectures

Dynamic Programming

- A simple but non-trivial method for designing algorithms
- Achieve much better efficiency than naïve ones.

- A couple of examples will be exhibited and analyzed.

Problem 1: Chain matrix multiplication

Suppose we want to multiply four matrices

- We want to multiply four matrices: $A \times B \times C \times D$.
- Dimensions: $A_{50\times20}$, $B_{20\times1}$, $C_{1\times10}$, $D_{10\times100}$
- Assume: cost $(X_{m\times n} \times Y_{n\times l}) = mnl$.
  - $A \times \big((B \times C) \times D\big)$: $20 \times 1 \times 10 + 20 \times 10 \times 100 + 50 \times 20 \times 100 = 120{,}200$
  - $A \times \big(B \times (C \times D)\big)$: $1 \times 10 \times 100 + 20 \times 1 \times 100 + 50 \times 20 \times 100 = 103{,}000$
  - $(A \times B) \times (C \times D)$: $50 \times 20 \times 1 + 1 \times 10 \times 100 + 50 \times 1 \times 100 = 7{,}000$
  - $\big((A \times B) \times C\big) \times D$: $50 \times 20 \times 1 + 50 \times 1 \times 10 + 50 \times 10 \times 100 = 51{,}500$
  - $\big(A \times (B \times C)\big) \times D$: $20 \times 1 \times 10 + 50 \times 20 \times 10 + 50 \times 10 \times 100 = 60{,}200$

The order matters!

- Question: In what order should we multiply them?

Key property

- General question: We have matrices $A_1, \ldots, A_n$, we want to find the best order for $A_1 \times \cdots \times A_n$
  - Dimension of A_i: $m_{i-1} \times m_i$
- One way to find the optimum: Consider the last step.
  - Suppose: $(A_1 \times \cdots \times A_i) \times (A_{i+1} \times \cdots \times A_n)$ for some $i \in \{1, \ldots, n-1\}$.
- $\text{cost}(1, n) = \text{cost}(1, i) + \text{cost}(i+1, n) + m_0 m_i m_n$

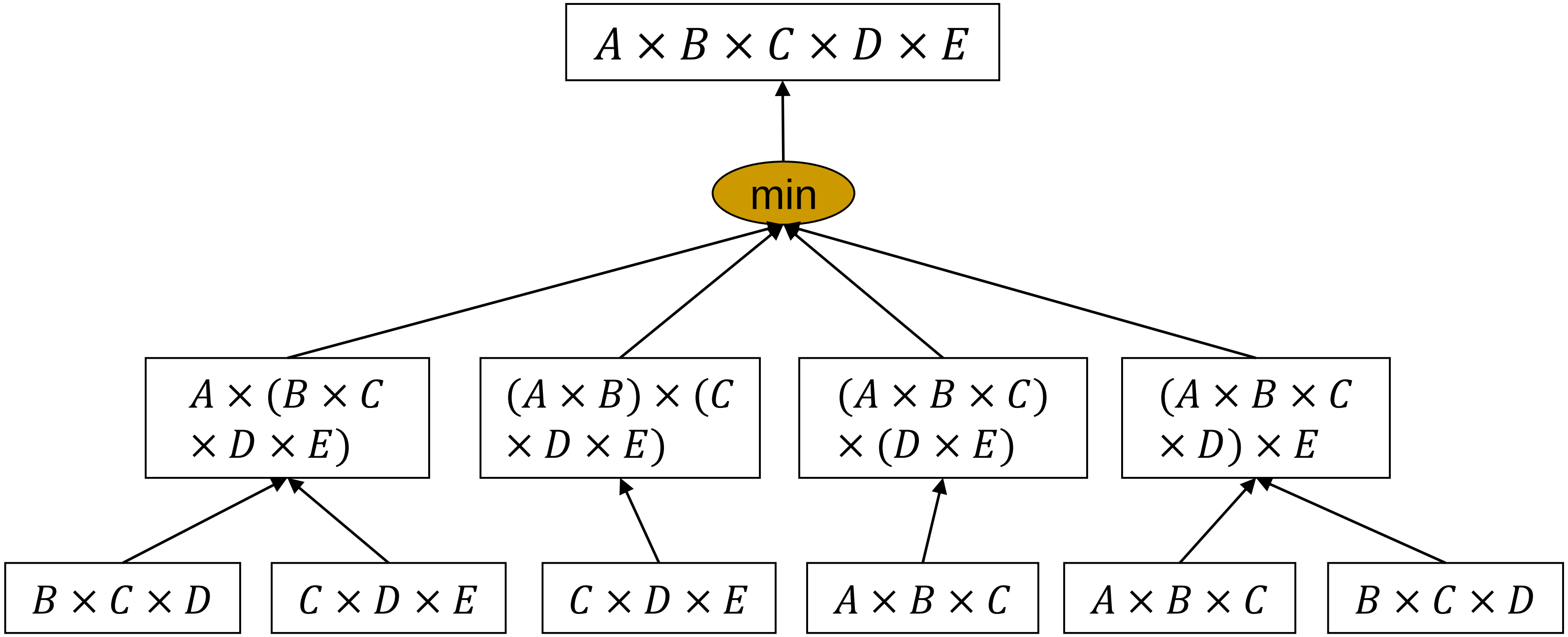
Algorithm

- But what is a best i?
- We don't know… Try all and take the min.

$$\mathrm{bestcost}(1,n)$$
$$= \min_i \mathrm{bestcost}(1,i) + \mathrm{bestcost}(i+1,n) + m_0 m_i m_n$$

  - $\mathrm{bestcost}(i,j)$: the min cost of computing $\left(A_i \times \cdots \times A_j\right)$
- How to solve $(A_1 \times \cdots \times A_i)$ and $(A_{i+1} \times \cdots \times A_n)$?
- Attempt: Same way, i.e. a recursion
- Complexity:
  - $T(1,n) = \sum_i (T(1,i) + T(i+1,n) + O(1))$
  - Exponential!

$A_{50\times 20}, B_{20\times 1}, C_{1\times 10}, D_{10\times 100}, E_{100\times 30}$



$A \times B \times C \times D \times E$

min

$A \times (B \times C \times D \times E)$

$(A \times B) \times (C \times D \times E)$

$(A \times B \times C) \times (D \times E)$

$(A \times B \times C \times D) \times E$

$B \times C \times D$

$C \times D \times E$

$C \times D \times E$

$A \times B \times C$

$A \times B \times C$

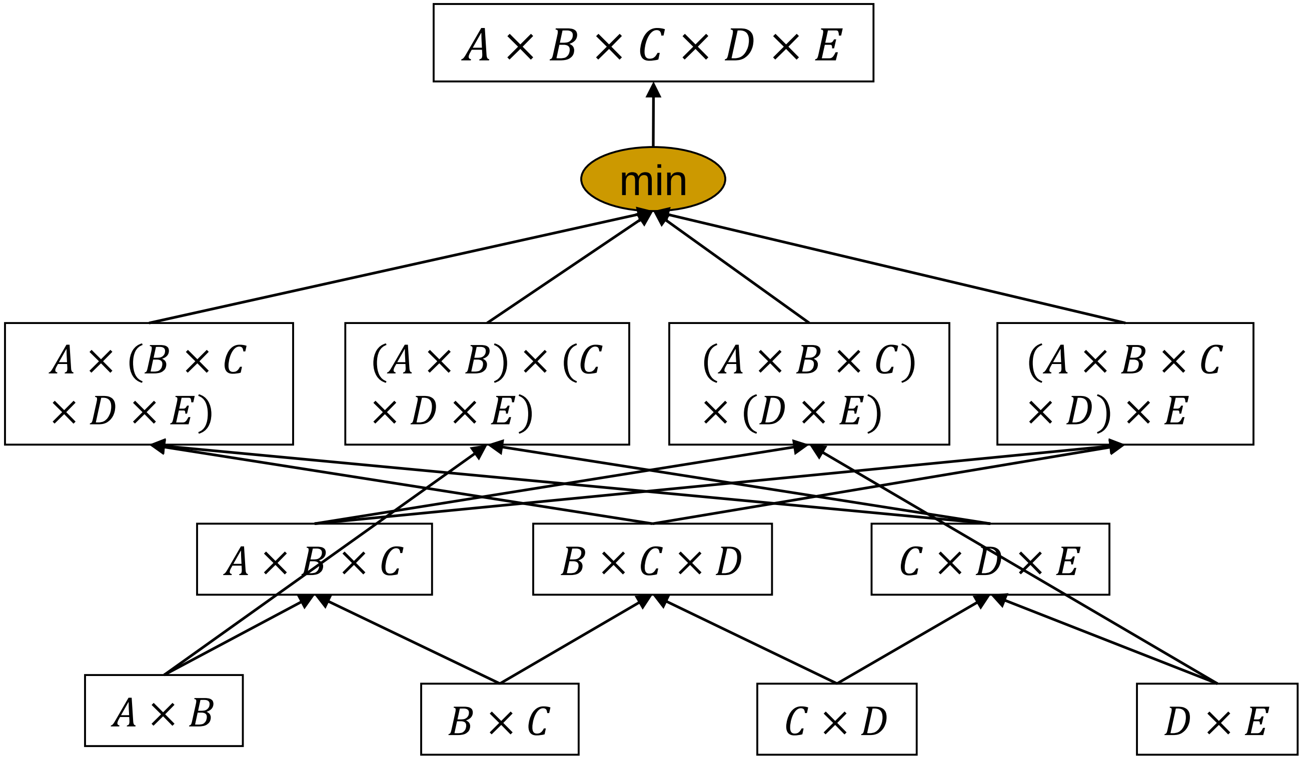
$B \times C \times D$

- Observation: small subproblems are calculated many times!

What did we observe?

- Why not just do it once and store the result for later reference?
  - When needed later: simply look up the stored result.
- That’s dynamic programming.
  - First compute the small problems and store the answers
  - Then compute the large problems using the stored results of smaller subproblems.

$A_{50\times20}, B_{20\times1}, C_{1\times10}, D_{10\times100}, E_{100\times30}$


$A \times B \times C \times D \times E$
min
$A \times (B \times C \times D \times E)$
$(A \times B) \times (C \times D \times E)$
$(A \times B \times C) \times (D \times E)$
$(A \times B \times C \times D) \times E$
$A \times B \times C$
$B \times C \times D$
$C \times D \times E$
$A \times B$
$B \times C$
$C \times D$
$D \times E$

- Now solve the problem this way.

Algorithm

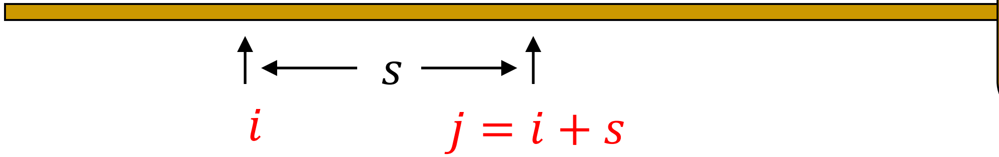
```
for $i = 1$ to $n$
    $C(i,i) = 0$
for $s = 1$ to $n - 1$   // $s$: step length
    for $i = 1$ to $n - s$
        $j = i + s$
        $C(i,j) = \min\{C(i,k) + C(k+1,j) + m_{i-1}m_k m_j : i \le k < j\}$
return $C(1,n)$
```

For the first example:
$s = 1$: {bestcost($A_1 \times A_2$), bestcost($A_2 \times A_3$), bestcost($A_3 \times A_4$)}
$s = 2$: {bestcost($A_1 \times A_2 \times A_3$), bestcost($A_2 \times A_3 \times A_4$)}
$s = 3$: {bestcost($\mathrm{A}_1 \times A_2 \times A_3 \times A_4$)}.

Best cost of $A_i \times \cdots \times A_k$

Best cost of $A_{k+1} \times \cdots \times A_j$

Cost of $X \times Y$, where
$X = A_i \times \cdots \times A_k$,
$Y = A_{k+1} \times \cdots \times A_j$



Complexity

- for $i = 1$ to n
    - $C(i, i) = 0$
- for $s = 1$ to $n - 1$ // s: step length
    - for $i = 1$ to $n - s$

      $\Theta(n^2)$ iterations

        - $j = i + s$

          - $O(1)$

        - $C(i, j) = \min\{C(i, k) + C(k + 1, j) + m_{i-1}m_k m_j : i \leq k < j\}$

          - $O(n)$
- return $C(1, n)$

- Total: $O(n^2) \times O(n) = O(n^3)$
    - Much better than the exponential!

Optimal value vs. optimal solution

- We’ve seen how to compute the optimal value using dynamic programming.

- What if we want an optimal solution?
  - The order of matrix multiplication.

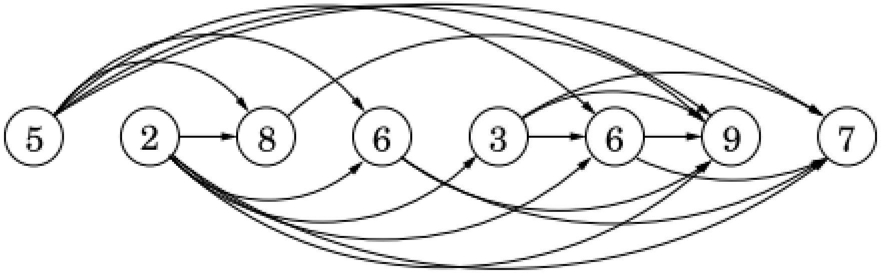
Problem 2: longest increasing subsequence

Problem 2: longest increasing subsequence

- A sequence of numbers $a_1, a_2, \dots, a_n$
  - Eg: 5, 2, 8, 6, 3, 6, 9, 7
- A subsequence: a subset of these numbers taken in order
  - $a_{i_1}, a_{i_2}, \dots, a_{i_j}$, where $1 \le i_1 < i_2 < \cdots < i_j \le n$
- An increasing subsequence: a subsequence in which the numbers are strictly increasing
  - Eg: 5, 2, 8, 6, 3, 6, 9, 7
- Problem: Find a longest increasing subsequence.

A good algorithm

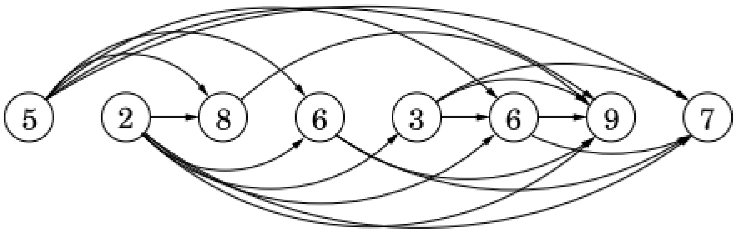
- Consider the following graph where
  - $V = \{a_1, \dots, a_n\}$
  - $E = \{(a_i, a_j)\colon i < j \text{ and } a_i < a_j\}$



longest increasing subsequence ↔ longest path

Attempt

- Consider the solution.
  - Suppose it ends at j.

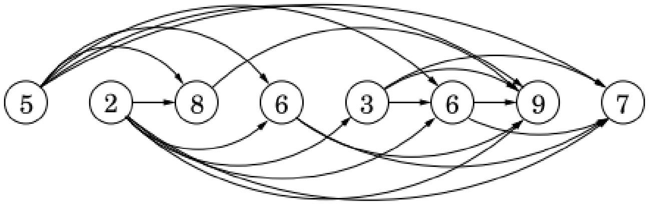


- The path must come from some edge (i, j) as the last step.
- If we do this recursively
  - $L(j) = \max_{i:(i,j)\in E} L(i) + 1$
    - $L(j)$ = length of the longest path ending at j
    - Length: # of nodes on the path.
  - Simple recursion: exponential.

Again…

- We observe that subproblems are calculated over and over again.
- So we record the answers to them.
- And use them for later computation.

Algorithm



- for $j = 1, 2, \ldots, n$
  - $L(j) = 1 + \max\{L(i): (i, j) \in E\}$
- return $\max_j L(j)$

- Run this algorithm on the example

  5, 2, 8, 6, 3, 6, 9, 7
- What's $\{L(j): j = 1, \ldots, 8\}$?

Correctness

- $L(j)$ = length of the longest path ending at j
  - Length here: number of nodes on the path
- $L(j) = 1 + \max\{L(i): (i, j) \in E\}$
- Any path ending at j must go through an edge (i, j) from some i
- Where is the best i?
  - It's taken care of by the max operation.
- By induction, property proved.

Complexity

- Obtaining the graph -$O(n^2)$
- for $j = 1, 2, \dots, n$
  - $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$ -$O(|N(j)|)$
- return $\max_j L(j)$

- Total: $O(n^2) + \sum_j O(|N(j)|) = O(n^2 + m) = O(n^2)$
  - $n = |V|, m = |E|$.
  - $N(j)$: set of incoming neighbours of vertex j

What's the strategy used?

- We break the problem into smaller ones.
- We find an order of the problems s.t. *easy* problems appear ahead of *hard* ones.
- We solve the problems in the order of their difficulty, and write down answers along the way.
- When we need to compute a hard problem, we use the previously stored answers (to the easy problems) to help.

Optimal value vs. optimal solution

- We’ve seen how to compute the optimal value using dynamic programming.
  - The length of the longest increasing subsequence.
- What if we want an optimal solution?
  - A longest increasing subsequence.

More questions to think about

- We’ve learned two problems using dynamic programming.
  - Chain matrix multiplication: solve problem(i, j) from $j - i = 1$ to $n - 1$
  - Longest increasing subsequence: solve problem(i) from $i = 1$ to n.
- Questions: Why different?
  - What happens if we compute chain matrix multiplication by solving problem(i) from $i = 1$ to n?
  - What happens if we compute longest increasing subsequence by solving problem(i, j) from $j - i = 1$ to $n - 1$?

In general

- Think about whether you can use algorithm methods A, B, C on problems X, Y, Z…

- That'll help you to understand both the algorithms and the problems.

Problem 3: All-pairs Shortest Path

Recap of shortest path problems

- We’ve learned how to find distance and a shortest path on a given graph.
  - st-Shortest Path: from vertex s to another vertex t
  - Single-Source Shortest Paths: s → all other vertices t.
- There is yet another shortest part problem:
  - All-Pairs Shortest Paths: all vertices $s \longrightarrow$ all other vertices t.

Naive algorithms and a new one

- Suppose that a given graph has negative edges but no negative cycles.
- If we use Bellman-Ford n times, each time for a different starting vertex s, then it takes time

$$O(|V| \cdot |E|) \cdot |V| = O(|E| \cdot |V|^2)$$

  - Recall: Bellman-Form takes times $O(|V| \cdot |E|)$.
- Now we give an algorithm with running time $O(|V|^3)$, using dynamic programming.

subproblems

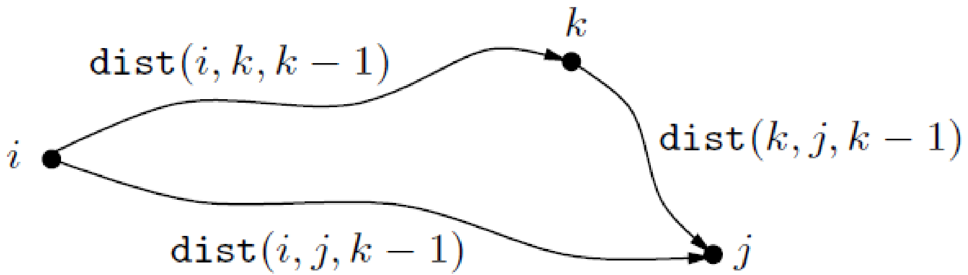
- Subproblem

  $\text{dist}(i, j, k) = \text{distance from } i \text{ to } j$

  $\text{using only vertices } \{1,2, \ldots, k\}$
- For each k, compute $\text{dist}(i, j, k)$ for all (i, j).
- We need to know whether using vertex k gives a shorter path
  - compared to using only vertices $\{1,2, \ldots, k-1\}$.
- What’s the update rule?

Updating rule

- Observation. If vertex k is used in a shortest path, it's used only once.
  - We assumed that there is no negative cycle.
- Comparison:



$$\begin{aligned}&\mathrm{dist}(i,j,k)\\ &= \min\{\underset{\substack{\downarrow\\ \text{shortest path}\\ \text{using vertex } k}}{\mathrm{dist}(i,k,k-1)+\mathrm{dist}(k,j,k-1)}, \underset{\substack{\downarrow\\ \text{shortest path}\\ \text{without using vertex } k}}{\mathrm{dist}(i,j,k-1)}\}\end{aligned}$$

Floyd-Warshall Algorithm

```
for $i = 1$ to $n$
    for $j = 1$ to $n$
        $\text{dist}(i, j, 0) = \infty$
for all $(i, j) \in E$
    $\text{dist}(i, j, 0) = w(i, j)$ // weight on edge $(i, j)$
for $k = 1$ to $n$
    for $i = 1$ to $n$
        for $j = 1$ to $n$
            $\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k-1) + \text{dist}(k, j, k-1),$
                                   $\text{dist}(i, j, k-1)\}$
Output $\text{dist}(i, j, n)$ for all $(i, j)$
```

Complexity

```
for $i = 1$ to $n$
    for $j = 1$ to $n$                                   $O(n^2)$
        $\text{dist}(i, j, 0) = \infty$
for all $(i, j) \in E$                                   $O(m)$
    $\text{dist}(i, j, 0) = w(i, j)$
for $k = 1$ to $n$
    for $i = 1$ to $n$                                   $O(n^3)$
        for $j = 1$ to $n$
            $\text{dist}(i, j, k) = \min\{\text{dist}(i, k, k-1) + \text{dist}(k, j, k-1),$
                                         $\text{dist}(i, j, k-1)\}$
Output $\text{dist}(i, j, n)$ for all $(i, j)$              → $O(n^2)$
```

- Total cost: $O(n^3)$

Problem 4: Edut dstamnce

Definition and applications

- Edut dstamnce

  → Edit distance
- $E(x, y)$: the minimal number of single-character *edits* needed to transform x to y.
  - *edit*: deletion, insertion, substitution
  - x and y don’t need to have the same length
- Applications:
  - Misspelling correction
  - Similarity search (for information retrieval, plagiarism catching, DNA variation)
  - …

What are subproblems now?

- It turns out that the edit distance between prefixes is a good one.
- We want to know $E(x_1 \ldots x_i, y_1 \ldots y_j)$. Suppose we already know
  - $E(x_1 \ldots x_{i-1}, y_1 \ldots y_{j-1}) = d_1$
  - $E(x_1 \ldots x_{i-1}, y_1 \ldots y_j) = d_2$
  - $E(x_1 \ldots x_i, y_1 \ldots y_{j-1}) = d_3$
- Express $E(x_1 \ldots x_i, y_1 \ldots y_j)$ as a function of d_1, d_2, d_3 and comparison of (x_i, y_j).

Answer

- $E(x_1 \ldots x_{i-1}, y_1 \ldots y_{j-1}) = d_1$
- $E(x_1 \ldots x_{i-1}, y_1 \ldots y_j) = d_2$
- $E(x_1 \ldots x_i, y_1 \ldots y_{j-1}) = d_3$
- $E(x_1 \ldots x_i, y_1 \ldots y_j) = \min\{\text{diff}(x_i, y_j) + d_1, 1 + d_2, 1 + d_3\}$
  - $\text{diff}(x_i, y_j) = \begin{cases} 1 & x_i \neq y_j \\ 0 & x_i = y_j \end{cases}$
- Two cases:
  - $x_i = y_j$
  - $x_i \neq y_j$

If $x_i = y_j$

- Option 1: delete x_i. Reduces to $E(x_1 \dots x_{i-1}, y_1 \dots y_j) = d_2$.
- Option 2: delete y_j. Reduces to $E(x_1 \dots x_i, y_1 \dots y_{j-1}) = d_3$.
- Option 3: Don’t delete x_i or y_j. Reduces to $E(x_1 \dots x_{i-1}, y_1 \dots y_{j-1}) = d_1$.
- So $E(x_1 \dots x_i, y_1 \dots y_j) = \min\{d_1, 1 + d_2, 1 + d_3\}$ in case of $x_i = y_j$
  - *“1”: the cost for the deletion.*
- **Exercise**. Show that the minimum is always achieved by d_1 in this case of $x_i = y_j$.

If $x_i \neq y_j$:

- To finally match the last character, we need to do at least one of the following three:
  - Delete x_i
  - Delete y_j
  - Substitute y_j for x_i

Each costs 1.

- Convince yourself that inserting letters after x_i or y_j doesn't help.
- It reduces to three subproblems:
  - Delete x_i: $E(x_1 \dots x_{i-1}, y_1 \dots y_j) = d_2$
  - Delete y_j: $E(x_1 \dots x_i, y_1 \dots y_{j-1}) = d_3$
  - Substitute y_j for x_i: $E(x_1 \dots x_{i-1}, y_1 \dots y_{j-1}) = d_1$
- We pick whichever is the best, so
  - $E(x_1 \dots x_i, y_1 \dots y_j) = \min\{1 + d_1, 1 + d_2, 1 + d_3\}$ in case of $x_i \neq y_j$

Now the algorithm

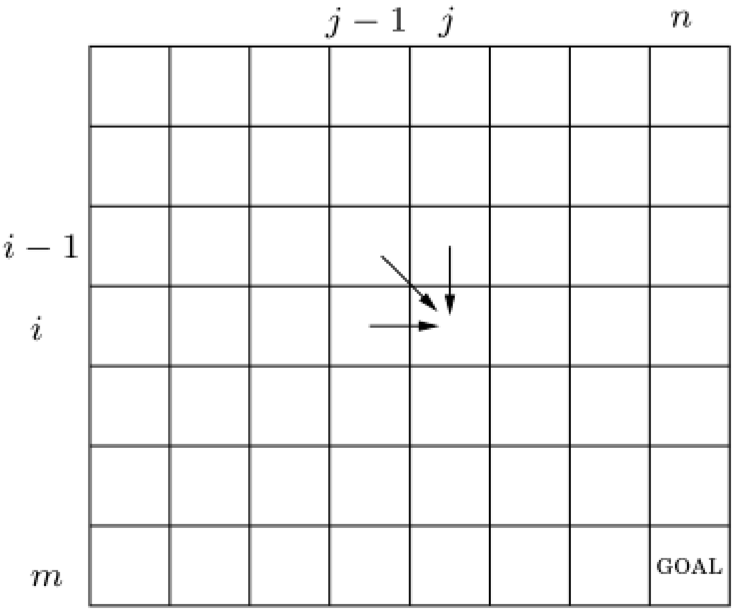
```
for $i = 0,1,2, \ldots, m$
    $E(i, 0) = i$
for $j = 1, 2, \ldots , n$:
    $E(0, j) = j$
for $i = 1,2, \ldots, m$:
    for $j = 1,2, \ldots, n$:
        $E(i, j) = \min\{E(i-1, j) + 1, E(i, j-1) + 1, E(i-1, j-1) + \mathrm{diff}(x_i, y_j)\}$
return $E(m, n)$
// recall:
```

$$\mathrm{diff}(x_i, y_j) = \begin{cases} 1 & x_i \neq y_j \\ 0 & x_i = y_j \end{cases}$$

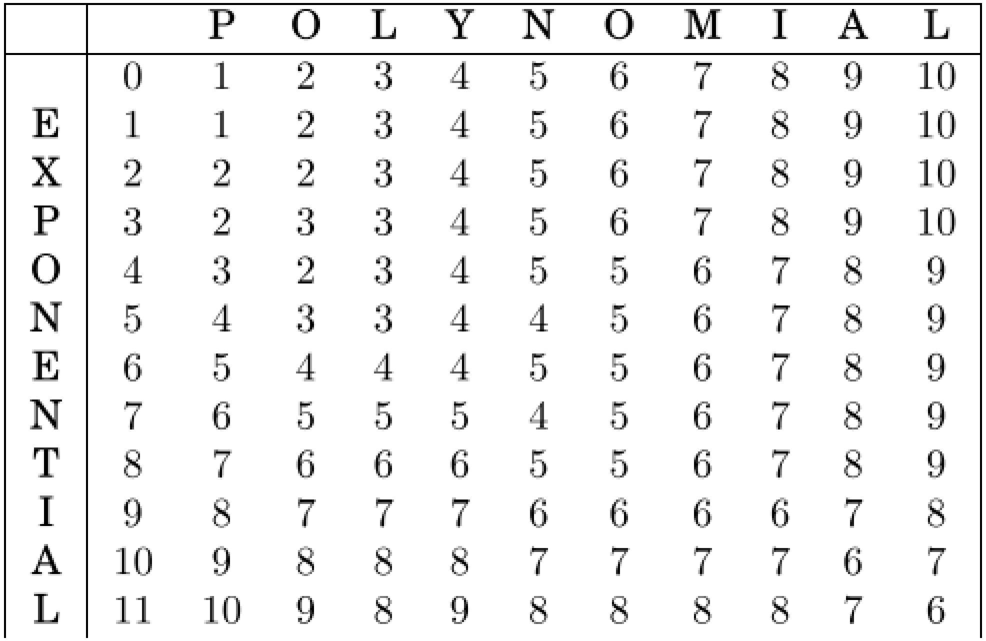
The initialization part corresponds to $E(\text{empty\_string}, y_1 \ldots y_j) = j$.

(The best way is simply insert $y_1 \ldots y_j$ one by one.)

And similarly $E(x_1 \ldots x_i, \text{empty\_string}) = i$.



Running it on (polynomial, exponential)



		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

$$E(i,j) = \min\{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+\mathrm{diff}(x_i,y_j)\}$$

Complexity

- for $i = 0,1,2, \ldots, m$
  - $E(i, 0) = i$
- for $j = 1, 2, \ldots , n$:
  - $E(0, j) = j$
- for $i = 1,2, \ldots, m$:

  for $j = 1,2, \ldots, n$:

  $E(i, j) = \min\{E(i-1, j) + 1, E(i, j-1) + 1, E(i-1, j-1) + \text{diff}(x_i, y_j)\}$
- return $E(m, n)$

- $O(1)$ time for each square, so clearly $O(mn)$ in total.

Optimal value vs. optimal solution

- We’ve seen how to compute the optimal value using dynamic programming.
  - The edit distance.

- What if we want an optimal solution?
  - A short sequence of insert/delete/substitution operations to change x to y.

Summary of dynamic programming

- Break the problem into smaller subproblems.
- Subproblems overlap
  - Some subproblems appear many times in different branches.
- Compute subproblems and store the answers.
- When later needed to solve these subproblems, just look up the stored answers.