DP III: Parenthesization, Edit Distance, Knapsack

Overview

- Subproblems for strings
- Parenthesization
- Edit distance (& longest common subseq.)
- Knapsack
- Pseudo polynomial Time

Review

5 easy steps to dynamic programming

(a) define subproblems — count # subproblems

(b) guess (part of solution) — count # choices

(c) relate subproblem solutions — compute time/subproblem

(d) recurse + memoize — time = time/subproblem · # subproblems

OR build DP table bottom-up

check subproblems acyclic/topological order

(e) solve original problem: = a subproblem

OR by combining subproblem solutions ==⇒ extra time

* problems from L20 (text justification, Blackjack) are on sequences (words, cards)

* useful problems for strings/sequences x:

suffixes x[i :] / prefixes x[: i] — Θ(|x|) ← cheaper ==⇒ use if possible

substrings x[i : j] — Θ(x^2)

Parenthesization

Optimal evaluation of associative expression A[0] · A[1] · · · A[n − 1] — e.g., multiplying rectangular matrices

A, of dimensions 2×10

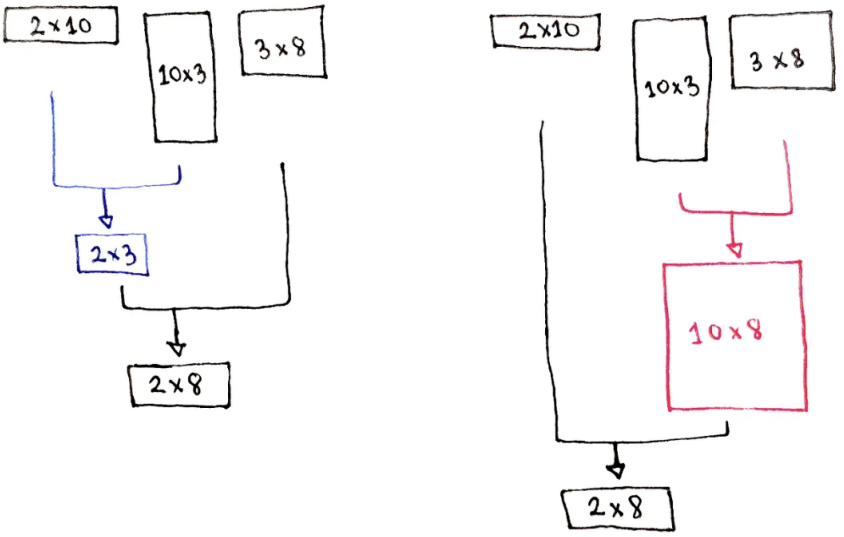
B, of dimensions 10×3

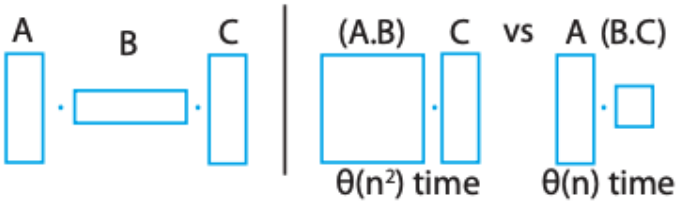
C, of dimensions 3×8

The following computations yield the same result, but require different number of operations:

(AB)C. The first multiplication generates a 2×3 matrix, which is then multiplied by C. This requires (2×10×3)+(2×3×8)=108 operations.

A(BC). The first multiplication generates a 10×8 matrix, which is then multiplied by A. This requires (10×3×8)+(2×10×8)=400 operations.





2. guessing = outermost multiplication (· · ·) (· · ·)

↑k−1 ↑k ⟹ # choices = O(n)

1. subproblems = prefixes & suffixes? NO

= cost of substring A[i : j] ⟹ # subproblems = Θ(n^2)

3. recurrence:

• DP[i, j] = min(DP [i, k] + DP[k, j]+ cost of multiplying (A[i] · · · A[k − 1]) by
(A[k] · · · A[j − 1]) for k in range(i + 1, j))

• DP[i, i + 1] = 0 ⟹ cost per subproblem = O(j − i) = O(n)

4. topological order: increasing substring size. Total time = O(n^3)

5. original problem = DP[0, n] (& use parent pointers to recover parents.)

* Above DP is not the shortest path in the subproblem DAG! Two dependencies ⇒ not path!

** 참고
https://medium.com/@avik.das/dynamic-programming-deep-dive-chain-matrix-multiplication-a3b8e207b201

https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/

Matrices Multiply: https://www.mathsisfun.com/algebra/matrix-multiplying.html

Edit Distance

두 개의 문자열이 같아지기 위해 몇 번의 Add/Delete/Replace가 필요한지 구하는 알고리즘

사용예: 오디오 유사도, 문서의 유사도, 스펠 체크 등 (표절 체크?)

(문제) Given two strings str1 and str2 and below operations that can be performed on str1. Find the minimum number of edits (operations) required to convert 'str1' into 'str2'.

Insert / **Remove** / **Replace**, All of the above operations are of equal cost.

(예) Input: str1 = "s**un**day", str2 = "s**atur**day"

Output: 3, Explanation: Last three and first characters are the same. We basically need to convert "un" to "atur". This can be done using below three operations. Replace 'n' with 'r', insert t, insert a

Definition

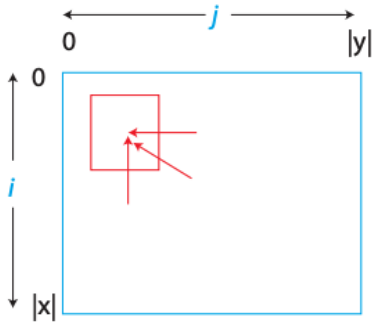
c(i, j) = edit-distance(x[i :], y[j :]) for 0 ≤ i < |x|, 0 ≤ j < |y| ⇒ Θ(|x| · |y|) subproblems

n = len(x), m = len(y), c(i, j) = edit distance between x[i..n] and y[j..m] => c(0, 0)

dp(i,j) = min

- cost(delete x[i]) + c(i + 1, j) if i < |x|,

- cost(insert y[j]) + c(i, j + 1) if j < |y|,
- cost(replace x[i] → y[j]) + c(i + 1, j + 1) if i < |x| & j < |y|



** 3개의 인접 셀로부터 영향을 받는다. bottom-up/right to left

** 참고

https://www.geeksforgeeks.org/edit-distance-dp-5/
https://blog.naver.com/ndb796/220870218783

https://www.youtube.com/watch?v=We3YDTzNXEk(실제 문제 어떻게 푸는지)

Knapsack

Knapsack of size S you want to pack

• item i has integer size si & real value vi

• goal: choose subset of items of maximum total value subject to total size ≤ S

• DP[i, X] = max(DP[i + 1, X], vi + DP[i + 1, X − si] if si ≤ X)

• DP[n, X] = 0

=⇒ time per subproblem = O(1)

Pseudo-polynomial

알고리즘의 복잡도를 산정할 시 최악의 경우에 대한 시간 복잡도가 input의 크기(갯수)가 아닌 input의 값(size of value)으로 결정되는 알고리즘을 Pseudo-polynomial algorithm이라 부른다.