PPS-3

↳ Khushi Agarwal

↳ 21BRS1529

2.)

Input: Chain of matrices $\langle A_1, A_2, \ldots, A_n \rangle$ where A_i is a matrix of size $P_{i-1} \times P_i$. $(i = 1, 2, \ldots, n)$

Output: full parenthization of $\langle A_1, A_2, \ldots, A_n \rangle$ (which have minimum Scalar Multiplication) and minimum Value of Scalar Multiplication

Logic

Recursive approach can be used combined with memoization.

1) Define a recursive function, MCM, that takes the chain of matrices ~~and~~, start and end indices of the chain.

2) If start and end indices are same, return 0 scalar multiplication and an empty paranthesization.

3.) Iterate over all possible split points within the subchain.

4) For each split point, recursively call the MCM function for the left and right subchains.

5) Compute the number of scalar multiplication required for the current split by multiplying the dimensions of the matrices

6) compute total no of multiplication by summing the multiplication in left subchain right subchain, and current split

7) if total number of multiplication for the current split is less than the current minimum, update the minimum and store the optimal parenthesization

Illustration

chain of matrices A1, A2, A3 with dimension.

A1(10 X 30), A2(30 X 5), A3(5 X 60).

we start

A1 * A2 * A3

(A1)(A2 * A3) ①   ((A1 * A2) * A3) ②   (A1 * (A2 * A3)) ③

10 X 30 30 X 5 5 X 60

~~30 X 5 X 60~~

① Split at A1: (A1) * (A2 * A3) = (10 X 30) * ((30 * 5) * (5 * 60)) = (10 X 30) * (30 * 60) = (10 * 60).

② Split at A2: ((A1 * A2) * A3) =
= ((10 X 30) * (30 * 5)) * (5 X 60)
= (10 X 5) * (5 X 60) = (10 * 60)

③ Split at A3

(A1 * (A2 * A3)) = ((10X30) * ((30 X5) *
(5 X60)) = ((10 * 30) * (30 * 60) = (10 X 60)

In this case, all possible split yields the same dimension (10X60).

∴ The no of scalar multiplications required will be same for all parenthization.
We can choose any valid paranthization

<u>Pseudo Code.</u>

```
Start function MCM (chain, start, end):
    if start = end :
        return 0, " "
    minM = INFINITY
    Optimal P = " "

  for i from start to end-1:
leftM , leftP = MCM (chain, start, i)
    rightm , rightP = MCM(chain, i+1, end)

    currentM = chain[start-1][0] * chain[i][1]
    * chain [end][1]
  totalM = leftM + rightM + currentM

    if totalM < minM :
        minM = totalM
    Optimal P = "(" + leftP + right P + ")"
    return minM, optimalP
```

Proof of Correctness (POC)

~~Running~~

1) Base Case: When start and end indices are the same, we have a single matrix, and the f^n correctly return 0 scalar multiplications and an empty parenthesis.

2) Memoization: We use memoization to avoid redundant calculations. By storing and reusing previously computed results, we ensure that the algorithm runs efficiently.

3) Optimal Soln: At the end of algorithm, the optimal parenthization and the minimum number of scalar multiplication are returned.

Running Time

$T(n) = O(n^2)$

Best Case

When all possible splits have the same number of scalar multiplications.

$O(n^2)$

Average Case

$O(n^2)$

Worst Case

$O(n^3)$ when no memoization is used.

3)

Input: Chain of matrices A1, A2 ..., A_n
where A_i is a matrix of size $p_{i-1} \times p_i$

Output: Parenthized product of A1 * A2.

Logic:

Dynamic Programming is used to solve this problem. The key idea is to break down the problem into subproblems and find the optimal solution for each subproblem.

To compute optimal solution, we use bottom-up Approach.

M [i] [j] → represents minimum number of scalar multiplications needed to multiply matrices from A_i to A_j

~~Algorithm~~

S [i] [j] → stores the split point that gives the minimum cost.

To obtain full parenthization of optimal solution, we can use a recursive approach.
The printP (Parenthesis) function takes the S array and recursively prints the parenthesis based on split points.

Illustration

lets consider a three matrices.

A1 → 2X3 size.

A2 → 3X4

A3 → 4X2

Matrix size [2,3,4,2]

1. Initialize the arrays M and S.
2. Calculate min cost for each subproblem and update M & S.
3. Print full parenthization based on split points stored in S.

Matrix M: [[0,24,20], [0,0,24], [0,0,0]]

Matrix S: [[0,1,1], [0,0,2], [0,0,0]]

Parenthization ((A1 (A2A3))

Pseudocode

```
function MCM (size):
    n = length (size) -1
    M = create 2D array of size nXn.
    S = create 2D array of size nXn

    for i=1 to n:
        M [i][i] = 0

    for chainlength = 2 to n:
        for i=1 to n - chainlength +1:
            j = i to chainlength -1
```

M [i][j] = infinity

for k = i to j-1 :

cost = M [i][k] + M [k+1] [j] +

Matrix sizes [i-1] * Matrix sizes [k] *

Matrix sizes [j]

if cost < M [i][j]:

M [i][j] = cost

s [i][j] = k.

print P (s, 1, n)

function print P (S, i, j) :

if i == j :

print "A" to i

else :

print "C".

~~print P~~

print P (S, i , S[i][j])

print P (S, S[i][j] +1, j)

print ")"

Running Time

$$T(n) = \frac{(n-1)(n-2)}{2}.$$

Best-case, Avg-case and worst-case

In terms of Matrix Chain Multiplicat, the best case average-case and worst-case running times are Same. The algorithm uses bottom-up approach

∴ Running Time is deterministic and solely depends on size of input.

Time Complexity

$O(n^3)$ $n \rightarrow$ no of matrices in chain.

This is because the algorithm involves 3 nested loops, and each loop iterates up to n-1.

(6)

Input: chain of matrices A1, A2, ..., An

Output: Parenthized product of A1 * A2 * ... An. that involve the minimum number of Scalar multiplications

Logic

We can extend the existing MCM pseudocode by introducing modifications to store all possible parenthizations when the cost is equal to minimum cost.

Illustration

let's consider sequence of Matrices. A1, A2, A3, A4, A5. where their dimensions given as [3,2,4,2,5]

step 1) call MCM(p,1,5)

MCM(p,1,1) MCM(p,2,5)

step 2) call MCM(p,1,1)

Since i = j, the cost is 0, and the parenthization is "A1".

Step 3) call MCM(p, 2, 5)

MCM(p, 2, 2)   MCM(p, 3, 5)

Step 4) call MCM(p, 2, 2)

i = j } cost = 0   parenthization is "A2"

Step 5) call MCM(p, 3, 5)

MCM(p, 3, 3)   MCM(p, 4, 5)

Step 6) call MCM(p, 3, 3)

i = j } cost = 0 } parenthization is "A3"

Step 7: call MCM(p, 4, 4)

i = j } cost = 0, parenthization is "A4"

Modified Pseudocode

```
function MCM(p, i, j):
    if i == j:
        return (0, ["A" + i])
    else:
        min_cost = infinite
        Optimal P = []

    for k = i to j-1:
    cost_left, p_left = MCM(p, i, k)
    cost_right, p_right = MCM(p, k+1, j)

    total_cost = cost_left + cost_right + p[i-1] *
                 p[k] * p[j]
```

if total_cost < min_cost:

min_cost = total_cost

~~optimal_p~~

Optimal P = []

for ~~left~~ left_p in p_left:

for right_p in p_right:

OptimalP.append("(" + left_p + ")" +
"(" + right_p + ")")

else if

total_cost == min_cost:

for left_p in p_left:

for right_p in p_right:

OptimalP.append("(" + left_p + ")" +
"(" + right_p + ")")

return (min_cost, optimal P)

Running Time T(n)

T(n) → O(n^3)

Best case: Occurs when given sequence of matrices is already fully parenthesized

Running Time → $O(n^3)$

Average case: depends on distribution of matrix dimension and probability of different parenthesizations

Worst-case : when all parenthizations need to be considered.

$O(n^3)$.

Time complexity : $O(n^3)$ where n → number of matrices in sequence. This complexity arises from nested loops in algorithm, which iterate over range of matrix positions and involve computations that takes $O(1)$ Time.