Exercise 6.1 (teacher demo)

Let A, B, C, D be matrices with dimensions 5×4, 4×6, 6×2 and 2×5, respectively. Using dynamic programming, determine the multiplication sequence that minimizes the number of scalar multiplications in computing $ABCD$.

Solution

Matrix multiplication is an associative but not a commutative operation. This means that we are free to parenthesize the multiplication however we like, but we are not free to rearrange the order of the matrices. Also recall that when two (nonsquare) matrices are being multiplied, there are restrictions on the dimensions. A $p \times q$ matrix has p rows and q columns. You can multiply a $p \times q$ matrix X times a $q \times r$ matrix Y, and the result will be a $p \times r$ matrix Z. (The number of columns of X must be equal the number of rows of Y.) Observe that there are pr total entries in Z and each takes $O(q)$ time to compute, thus the total time to multiply these two matrices is proportional to the product of the dimensions, pqr.

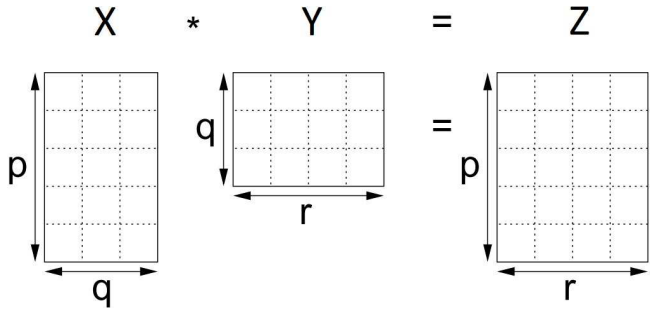

Figure 1: Matrix Multiplication.

Motivate to yourself why the problem is important (consider the simple case where we decide in which order to perform the multiplications for the sequence ABC).

This problem, like other dynamic programming problems involves determining a structure (in this case, a parenthesization). We want to break the problem into subproblems, whose solutions can be combined to solve the global problem. Here, a subproblem consists of determining the parenthesization that minimizes the number of scalar multiplications given a subsequence of the sequence $ABCD$. Notice, also that this problem involves optimal substructure.

We will solve the problem with a bottom-up approach, i.e., starting from sequences with one matrix, and building up to matrix sequences with four matrices. The DP-algorithm actually reduces the complexity of the problem from $O(2^n)$ to $O(n^3)$, compared to a brute-force-approach. This is because, the DP-algorithm does not solve the overlapping subproblems multiple times, and utilizes the optimal substructure.

The cost-to-go function for different matrix sequences calculates the minimal amount of scalar multiplications required.

$J(A) = 0$
$J(B) = 0$
$J(C) = 0$
$J(D) = 0$

$J(AB) = 5 \cdot 4 \cdot 6 = 120$
$J(BC) = 4 \cdot 6 \cdot 2 = 48$
$J(CD) = 6 \cdot 2 \cdot 5 = 60$

$$\begin{aligned} J(ABC) &= \min\{J(A) + J(BC) + 5 \cdot 4 \cdot 2, J(AB) + J(C) + 5 \cdot 6 \cdot 2\} \\ &= \min\{0 + 48 + 40, 120 + 0 + 60\} = \min\{88, 180\} = 88. \\ J(BCD) &= \min\{J(B) + J(CD) + 4 \cdot 6 \cdot 5, J(BC) + J(D) + 4 \cdot 2\} \\ &= \min\{0 + 60 + 120, 48 + 0 + 40\} = \min\{180, 88\} = 88. \end{aligned}$$

$$\begin{aligned} J(ABCD) = \min\{&J(A) + J(BCD) + 5 \cdot 4 \cdot 5, J(AB) + J(CD) + 5 \cdot 6 \cdot 5, \\ &J(ABC) + J(D) + 5 \cdot 2 \cdot 5\} = \min\{0 + 88 + 100, 120 + 48 + 150, 88 + 0 + 50\} \\ &= \min\{188, 318, 138\} = 138. \end{aligned}$$

The optimal parenthesization is $((A)(BC))(D)$ and requires 138 scalar multiplications. On the other hand, the worst parenthesization $(AB)(CD)$ requires 318 scalar multiplications. Even for this small example, considerable savings can be achieved by reordering the evaluation sequence. This general class of problem is important in compiler design for code optimization and in databases for query optimization.

Note that we could have as well solved the problem with a top-down approach, where we start with $J(ABCD) = \min\{...\}$ and use recursive calls, to arrive to the same optimal solution.

Exercise 6.2 (solved in class)

The graph of Fig. 2 depicts a flexible production line. The initial product A has to go through three phases to become the final product J. The cost from each node to another is given next to the corresponding arc.

Calculate the optimal cost-to-go function for each node and determine the steps that have to be taken for to minimize the cost.

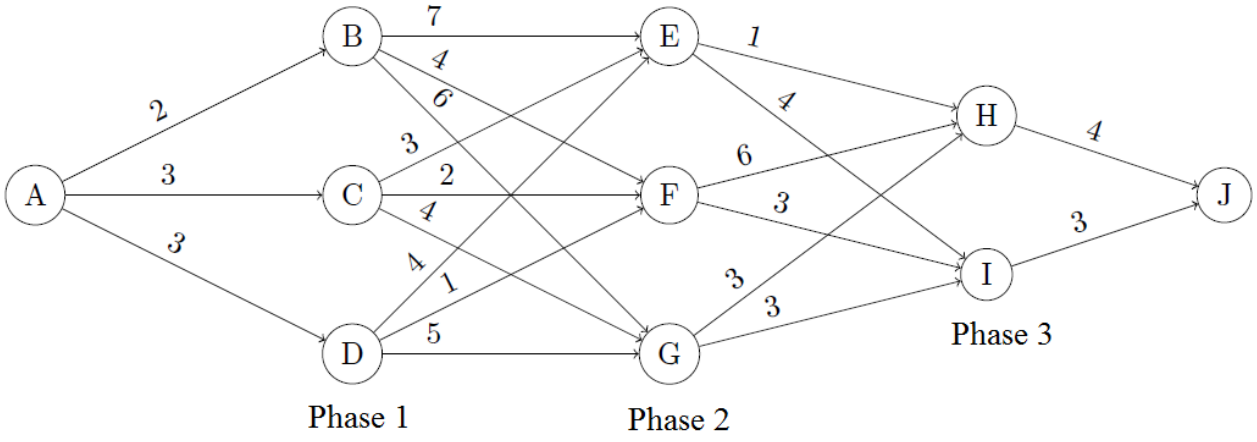

Figure 2: Graph of Exercise 6.2

Solution

backward iteration

Phase 4:

$J_4(J) = 0$

Phase 3:

$J_3(H) = 4$
$J_3(I) = 3$

Phase 2:

$J_2(E) = \min\{1 + J_3(H), 4 + J_3(I)\} = 5$
$J_2(F) = \min\{6 + J_3(H), 3 + J_3(I)\} = 6$
$J_2(G) = \min\{3 + J_3(H), 3 + J_3(I)\} = 6$

Phase 1:

$J_1(B) = \min\{7+J_2(E), 4+J_2(F), 6+J_2(G)\} = 10$
$J_1(C) = \min\{3+J_2(E), 2+J_2(F), 4+J_2(G)\} = 8$
$J_1(D) = \min\{4+J_2(E), 1+J_2(F), 5+J_2(G)\} = 7$

Phase 0:

$J_0(A) = \min\{2+J_1(B), 3+J_1(C), 3+J_1(D)\} = 10.$

forward iteration

Phase 0:

$J_0(A) = 0$

Phase 1:

$J_1(B) = 2$
$J_1(C) = 3$
$J_1(D) = 3$

Phase 2:

$J_2(E) = \min\{7+J_1(B), 3+J_1(C), 4+J_1(D)\} = 6$
$J_2(F) = \min\{4+J_1(B), 2+J_1(C), 1+J_1(D)\} = 4$
$J_2(G) = \min\{6+J_1(B), 4+J_1(C), 5+J_1(D)\} = 7$

Phase 3:

$J_3(H) = \min\{1+J_2(E), 6+J_2(F), 3+J_2(G)\} = 7$
$J_3(I) = \min\{4+J_2(E), 3+J_2(F), 3+J_2(G)\} = 7$

Phase 4:

$J_4(J) = \min\{4 + J_3(H), 3 + J_3(I)\} = 10.$

Optimal production order: A, D, F, I, J.

Exercise 6.3 (student presents)

You want to plant tomatoes, potatoes and peas on a 180 cm wide allotment. A row of tomatoes or potatoes takes up 40 cm of space, and a row of peas 20 cm. The utility is 10 for a row of tomatoes, 7 for a row of potatoes and 3 for a row of peas. Additionally, according to an EU-directive the maximum number of allowed rows of tomatoes is two. Give a dynamic programming algorithm that could be used to solve this problem.

Solution

Let l be the remaining width of the allotment, k the number of tomato rows planted, x_1, x_2, x_3 the required spaces and u_1, u_2, u_3 the utilities associated with peas, potatoes and tomatoes, respectively. Let $J(l,k)$ be the maximal utility one can get from an allotment, where there is l amount of remaining space, and k number of tomato rows planted. With the following algorithm the maximal utility of the allotment can be calculated.

$$J(l,k) = \begin{cases} 0 & \text{if } l < x_1 \\ u_1 + J(l - x_1, k) & \text{if } x_1 \leq l < x_2 \\ \max\limits_{i=1,2} \{u_i + J(l - x_i, k)\} & \text{if } k \geq 2 \text{ and } l \geq x_2 \\ \max \begin{cases} \max\limits_{i=1,2} \{u_i + J(l - x_i, k)\} \\ u_3 + J(l - x_3, k+1) \end{cases} & \text{else.} \end{cases}$$

Initially, $l = 180$ and $k = 0$.

Exercise 6.4 (solved in class)

A certain string-processing language offers a primitive operation which splits a string into two pieces. Since this operation involves copying the original string, it takes n units of time for a string of length n, regardless of the location of the cut. Suppose, now, that you want to break a string into many pieces. The order in which the breaks are made can affect the total running time. For example, if you want to cut a 20-character string at positions 3 and 10, then making the first cut at position 3 incurs a total cost of $20 + 17 = 37$, while doing position 10 first has a better cost of $20 + 10 = 30$.

Give a dynamic programming algorithm that, given the locations of m cuts in a string of length n, finds the minimum cost of breaking the string into $m + 1$ pieces.

Solution Let us denote the locations where the string can be cut by $x_0, ..., x_{m+1}$. Here $x_0 = 0$ and $x_{m+1} = n$. Let $J(i, j)$, where $i = 0...m$ and $j = 1...m + 1$, denote the minimum cost it takes to cut a string, which begins at x_i and ends at x_j. Now we can formulate the dynamic programming algorithm.

$$J(i,j) = \begin{cases} 0 & \text{for } j = i+1 \\ (x_j - x_i) + \min\limits_{i<k<j} \{J(i,k) + J(k+1,j)\} & \text{for } j > i+1. \end{cases}$$

Exercise 6.5 (self-study)

Initially there is A amount of a resource, and it has to be used until the end of N periods. The amount $u(k)$ of the resource used on period k produces the utility $\sqrt{u(k)}$.

a) Formulate the problem, i.e., define the state equation, utility function, control constraints, and boundary conditions for the state variable. Notice that the state and control are continuous!

b) Show that $J_k(x_k) = \sqrt{(N-k)x_k}$ is the optimal cost-to-go function.

Solution

a)

State variable: x_k (amount of resource in the start of period k).

State equation: $x_{k+1} = x_k - u_k$.

Utility function: $g(x_k, u_k) = \sqrt{u_k}$.

Control constraint: $0 \le u_k \le x_k, \quad \forall\, k.$

Boundary conditions: $x_0 = A, \quad x_N = 0.$

Objective function: $J = \sum_{k=0}^{N-1} g(x_k, u_k) \hookrightarrow \max!.$

b) The cost-to-go function of dynamic programming:

$$\begin{aligned} J_N^*(x_N) &:= 0 \\ J_{N-1}^*(x_{N-1}) &:= \sqrt{x_{N-1}} \\ J_k^*(x_k) &:= \max_{u_k \in \mathcal{U}_k} \Big[g_k(x_k, u_k) + J_{k+1}^*(x_{k+1}) \Big] \\ &= \max_{0 \le u_k \le x_k} \Big[\sqrt{u_k} + J_{k+1}^*(x_k - u_k) \Big], \quad k = 0, \ldots, N-2. \end{aligned}$$

Claim:

$$J_k^*(x_k) = \sqrt{(N-k)x_k}.$$

Lets prove it by induction. Initial step: OK, when $k = N$. Lets assume the claim holds for the cost-to-go function $J_{k+1}^*(x_{k+1})$:

$$\begin{aligned} J_k^*(x_k) &= \max_{0 \le u_k \le x_k} \Big[\sqrt{u_k} + J_{k+1}^*(x_k - u_k) \Big] \\ &= \max_{0 \le u_k \le x_k} \Big[\sqrt{u_k} + \sqrt{(N-k-1)x_{k+1}} \Big] \\ &= \max_{0 \le u_k \le x_k} \Big[\sqrt{u_k} + \sqrt{(N-k-1)[x_k - u_k]} \Big]. \end{aligned}$$

Derivate with regard to u_k and set the derivative to equal zero:

$$\begin{gathered} \frac{1}{2\sqrt{u_k^*}} - \frac{\sqrt{N-k-1}}{2\sqrt{x_k - u_k^*}} = 0 \\ \Downarrow \\ \frac{1}{u_k^*} = \frac{N-k-1}{x_k - u_k^*} \\ \Updownarrow \\ \boxed{u_k^* = \frac{1}{N-k} x_k} \end{gathered}$$

This is a feasible control for all $x_k > 0$. Insert the optimal control into the cost-to-go function:

$$\begin{aligned} J_k^*(x_k) &= \sqrt{\frac{x_k}{N-k}} + \sqrt{\frac{(N-k-1)^2 x_k}{N-k}} \\ &= \sqrt{\frac{x_k}{N-k}} + (N-k-1)\sqrt{\frac{x_k}{N-k}} \\ &= (N-k)\sqrt{\frac{x_k}{N-k}} \\ &= \sqrt{(N-k)x_k} \end{aligned}$$

and the claim has been proved. The initial condition $x_0 = A$ gives the control $u_0^* = A/N$. Then $x_1 = A(N-1)/N$, and the corresponding control is $u_1^* = A/N$. Notice, that as a matter of fact, the optimal control u^* is constant on each period.