CS 3510 — Algorithms, Spring 2022

# Lecture 11: Chain Matrix Multiplication

February 22, 2022

*Lecturer: Frederic Faulkner* *Scribe: Aditya Diwakar*

Exam 2 is next Thursday. It will cover Dynamic Programming (Lectures 7-11). The review lecture will be next Tuesday and HW2B is due Friday.

## 1 Chain Matrix Multiplication

If we have a multiplication problem such as $2 \cdot 3 \cdot 7 \cdot 3 = 126$, how can we compute this? We can compute it through a various of different ways by paranthesizing and computing numbers of 2 digits.

For example, $((2 \cdot 3) \cdot 7) \cdot 3$ or $(2 \cdot 3) \cdot (7 \cdot 3)$ or any number of ways (total of $5$ ways here, grows exponentially).

For normal multiplication between digits, this is fine because the order of multiplication does not matter. However, that is not the case for <u>chain matrix multiplication</u>.

- Don't fret: no linear algebra here!
- Every matrix has 2 dimensions ($m \times n$)
- Multiply $m \times n$ matrix with an $n \times p$ matrix to get a matrix with size $m \times p$
- In general, $(m \times n) \cdot (k \times j)$ cannot be done if $n \neq k$, else yields $m \times j$.

How long does it take to multiply two matrices? Assume that all integer operations are $O(1)$. Takes $m \cdot n \cdot p$ operations to multiply a matrix of size $m \times n$ by $n \times p$.

We design an algorithm to multiply a series of matrices together, let's say we have some function that takes an input of lists $m_0, m_1, m_2, \ldots, m_n$ corresponding to matrices $A_1, A_2, A_3, \ldots, A_n$ where $A_1$ has size $m_0 \times m_1$, $A_1$ has size $m_1 \times m_2$, and $A_n$ has size $m_{n-1} \times m_n$.

The output of this algorithm should be the number of operations it takes to multiply all these matrices together. This is not very easy as we would get a different output depending on the order we choose to multiply matrices.

For example, say we have matrices $(5 \times 2) \times (2 \times 7) \times (7 \times 2)$, then we could:

- Do $(2 \times 7) \times (7 \times 2)$ first (28 ops) and then multiply $(5 \times 2) \times (2 \times 2)$ (20 ops) for a total of 48 operations
- Do $(5 \times 2) \times (2 \times 7)$ first (70 ops) and then multiply $(5 \times 7) \times (7 \times 2)$ (70 ops) for a total of 140 operations

Let's try to use dynamic programming! We know that we are trying to multiply $A_1, \ldots, A_n$ and to get smaller subproblems, we have to split and divide what we are going to multiply. For example, ...

We can define our dynamic programming table as $T[i, j]$ representing the number of operations to multiply $A_i, \ldots, A_j$, then we can multiply from $i$ to $j$ by splitting at some $d$:

$$(A_i \cdots A_d)(A_{d+1} \cdots A_j)$$

The cost of this is whatever the cost of the left hand side is and whatever the right hand side is. This is simply $T[i, d] + T[d + 1, j]$. We also need to multiply these two together. The left matrix has resultant size of $m_{i-1} \times m_d$ and the right has size of $m_d \times m_j$ meaning that it takes $m_{i-1} \times m_d \times m_j$ total operations. Hence, $T[i, j]$ (for some value $d$) can be:

$$T[i, j] = T[i, d] + T[d + 1, j] + (m_{i-1} \cdot m_d \cdot m_j)$$

However, we want to find the minimum amount of time for multiplying these two matrices and since our variable was the split point, then we can minimize over all possible split points:

$$T[i, j] = \min_{i \le d < j} (T[i, d] + T[d + 1, j] + (m_{i-1} \cdot m_d \cdot m_j))$$

The base cases are $T[i, i] = 0$ and the answer is stored in $T[1, n]$.

How do we fill this table? We only care about the upper triangular part and the recurrence relation relies on value with a smaller column value (to the left) or larger row values (down).

Hence, we must fill this table diagonally starting with the main diagonally and moving towards the top right. Since $T[i, i]$ is a base case, then the main diagonal is already filled and we can continue from there.

In order to iterate diagonally, we care about the difference between the row and column values. Why?

Notice for the main diagonal, we have $(1, 1), (2, 2), \dots, (n, n)$ where the row ($r$) minus the column ($c$) is 0. Hence, we can say $\Delta = c - r = 0$.

- Second diagonal: $(1, 2), (2, 3), \dots, (n - 1, n) \implies \Delta = 1$
- Third diagonal: $(1, 3), (2, 4), \dots (n - 2, n) \implies \Delta = 2$
- Last Diagonal: $(1, n) = (n - (n - 1), n) \implies \Delta = n - 1$

Hence, we can iterate in this fashion using a $\Delta$ from $1$ to $n - 1$ (we can start at $1$ since $\Delta = 0$ is our base case).

```
Function ChainMatrixMultiplication($m_0, m_1, \dots, m_{n-1}, m_n$):
    /* let $T$ be an empty $n \times n$ DP Table */
    $T \leftarrow [][]$
    for $i \in 1 \rightarrow n$ do
        $T[i][i] \leftarrow 0$
    for $\Delta \in 1 \rightarrow n - 1$ do
        for $i \in 1 \rightarrow n - \Delta$ do
            $j \leftarrow i + \Delta$
            $T[i, j] \leftarrow \min_{i \le d < j} (T[i, d] + T[d + 1, j] + (m_{i-1} \cdot m_d \cdot m_j))$
    return $T[1, n]$
```

The runtime of this algorithm is $O(n^3)$ due to the nested loops for $i$, $\Delta$, and $d$.

## 2 Challenges

### 2.1 Splitting Game

Let's play a game! You are given an $n$ long 1D array. During each turn, you split the array into two smaller arrays and add the sum of the outer elements to your score. For example, if you are given the array

$$[1 \quad 3 \quad -1 \quad 4 \quad 2 \quad 8]$$

then you can split it in the following way

$$[1 \quad 3 \quad -1] \qquad [4 \quad 2 \quad 8]$$

and add $1 + (-1) + 4 + 8 = 12$ to your score. You would keep doing this for each turn. How can you design a DP algorithm to find the maximum score you can achieve? Solution is on the next page.

### 2.2 Longest Palindromic Subsequence

Design an algorithm to return the longest palindromic sequence. A palindrome is a string that is the same forwards or backwards. For example, racecar and 0110110.

*Hint:* You have two options here. One option is to re-use LCS on the string (what would your second string be?) or by re-deriving a new recurrence relation that is inspired from LCS.

Solution is on the next page. Work through both of these solutions before looking.

# 3 Challenge Solutions

## 3.1 Spliting Game

This problem is very similar to the chain matrix multiplication question. We are curious about the most optimal split. Define our table entries as $T[i,j]$ denoting the maximum score achievable from the subarray $A[i:j]$.

We know that if we split the array at index $d$, then it yields two subarrays, $A[i:d]$ and $A[d+1:j]$ and this turn will yield a score of $A[i]+A[d]+A[d+1]+A[j]$. Since we want to maximize the score, we can define $T[i,j]$ as:

$$T[i,j] = \max_{i \le d < j} \left(T[i,d] + T[d+1,j] + A[i] + A[d] + A[d+1] + A[j]\right)$$

The base case is $T[i,i]=0$ since any individual element cannot be split anymore meaning there is no additional score we could get.

An implementation of this recurrence looks like:

```
Function SplittingGame($A$):
    /* let $T$ be an empty $n \times n$ DP Table */
    $T \leftarrow [][]$
    for $i \in 1 \rightarrow n$ do
        $T[i][i] = 0$
    for $\Delta \in 1 \rightarrow n-1$ do
        for $i \in 1 \rightarrow n - \Delta$ do
            $j \leftarrow i + \Delta$
            $T[i,j] =$
              $\max_{i \le d < j} (T[i,d] + T[d+1,j] + A[i] + A[d] + A[d+1] + A[j])$
    return $T[1,n]$
```

The runtime of this algorithm is $O(n^3)$ due to the nested loop for $i, \Delta, d$.

## 3.2 Longest Palindromic Subsequence

One approach is to define your table entries as $T[i, j]$ representing the longest palindromic subsequence (LPS) from $A[i : j]$ which may not include $A[i]$ or $A[j]$.

Then, we know that either $A[i] = A[j]$ or $A[i] \neq A[j]$. In the case that $A[i] = A[j]$, then we know $T[i, j] = T[i-1, j-1] + 2$ since we are adding two characters to the palindrome. Otherwise, we can either exclude $A[i]$, exclude $A[j]$, or exclude both. If we exclude both, we know this is more minimal than excluding $A[i]$ or $A[j]$. Therefore, the relation for when $A[i] \neq A[j]$ is $\max(T[i+1, j], T[i, j-1])$ Therefore, the overall recurrence relation is:

$$T[i, j] = \begin{cases} T[i+1, j-1] + 2 & A[i] = A[j] \\ \max(T[i+1, j], T[i, j-1]) & A[i] \neq A[j] \end{cases}$$

The base cases are $T[i, i] = 1$ since every string of length 1 is palindromic and $T[i, j] = 2$ if $A[i] = A[j]$ and $j = i + 1$ (length 2 base case). Another question: how do we fill in the tables? Since we require lower rows and leftward columns, we iterate diagonally.

```
Function LPS(A):
    /* let T be an empty n × n DP Table */
    T ← [][]
    for $i \in 1 \to n$ do
        $T[i][i] = 0$
    for $\Delta \in 1 \to n-1$ do
        for $i \in 1 \to n-\Delta$ do
            $j \leftarrow i + \Delta$
            if $A[i] = A[j]$ then
                if $j = i + 1$ then
                    $T[i, j] \leftarrow 2$
                $T[i, j] \leftarrow T[i+1, j-1] + 2$
            else
                $T[i, j] \leftarrow \max(T[i+1, j], T[i, j-1])$
    return $T[1, n]$
```