

Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 08
Lecture - 04
Matrix multiplication

This is a final example to illustrate dynamic programming. We look at the problem of matrix multiplication.

(Refer Slide Time: 00:02)

The slide is titled "Multiplying matrices" in blue. To the right of the title, there are three hand-drawn boxes representing matrices. The first box is labeled 'm' on the left and 'n' on the top. The second box is labeled 'n' on the top and 'p' on the right. The third box is labeled 'm' on the left and 'p' on the bottom. A green circle with 'mp' inside is drawn to the right of the second box. Below the title, there is a bulleted list:

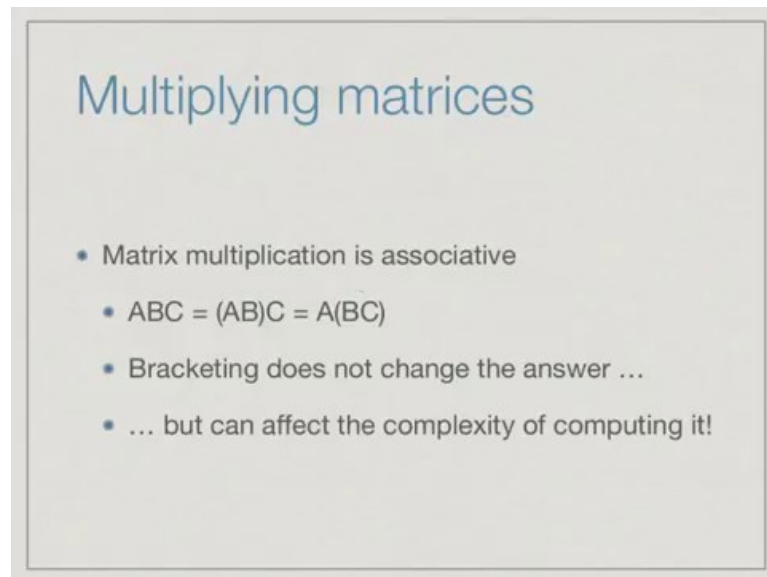
- To multiply matrices A and B, need compatible dimensions
- A of dimension $m \times n$, B of dimension $n \times p$
- AB has dimension mp
- Each entry in AB take $O(n)$ steps to compute
- $AB[i,j]$ is $A[i,1]B[1,j] + A[i,2]B[2,j] + \dots + A[i,n]B[n,j]$
- Overall, computing AB is $O(mnp)$

If **you** remember how matrix multiplication works, you have to take two rows, two matrices with compatible entries, and then we compute a new matrix in which for each entry there, we have to multiply a row here by a column of the same length and add up the values. If we have a matrix which is m by n then this must have n times p , so that we have a final answer which is m times p . In the final entry, we have to make mp entries in the final product matrix; and each of these entries, require us to compute this sum of these n entries, so that takes **us** order n time. With total work is, usually easy to compute us m times n times **p**.

So, $AB[i,j]$ is this long sum and this sum **has** 1, 2, 3 up to n entries. Computing matrix multiplication for two matrices **has** a very straight forward algorithm which is a product **triple** nested loop which takes order m times n times p , if all of the dimension **are** the

same this is an order n^3 cubed algorithm. Now there are more clever ways of doing it, but that is not the purpose of this lecture, but the naive straightforward, way of multiplying two matrices is m times n times p . Our interest is when we have a sequence of such multiplications to do.

(Refer Slide Time: 01:34)



The slide is titled "Multiplying matrices" in a blue font. It contains a bulleted list of points:

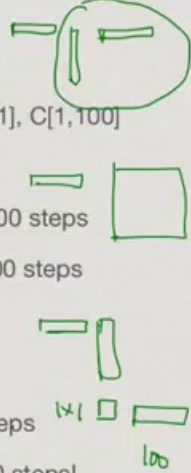
- Matrix multiplication is associative
 - $ABC = (AB)C = A(BC)$
 - Bracketing does not change the answer ...
 - ... but can affect the complexity of computing it!

Supposing, we want to multiply three matrices together A times B times C , then it turns out it does not matter whether we first multiply AB and then multiply C or we first multiply A and then multiply BC , A times BC , because this is stated as the associative the order in which we group the multiplication does not matter just for normal numbers. If we do 6 times 3 times 2 , it does not matter whether you do 6 times 3 first, or 3 times 2 first finally, the product is going to be the same. So, the bracketing does not change the answer the final value is the same, but it turns out that it can have dramatic effects on the complexity of computing the answer. Why is this the case.

(Refer Slide Time: 02:18)

Multiplying matrices

- Suppose dimensions are $A[1,100]$, $B[100,1]$, $C[1,100]$
- Computing $A(BC)$
 - BC is $[100,100]$, $100 \times 1 \times 100 = 10000$ steps
 - $A(BC)$ is $[1,100]$, $1 \times 100 \times 100 = 10000$ steps
- Computing $(AB)C$
 - AB is $[1,1]$, $1 \times 100 \times 1 = 100$ steps
 - $(AB)C$ is $[1,100]$, $1 \times 1 \times 100 = 100$ steps
- $A(BC)$ takes 20000 steps, $(AB)C$ takes 200 steps!



Suppose, we have these matrices A, B and C, and A and B have these columns and rows. A is basically got 1 by 100, A just has 1 row and 100 columns and B has a 100 rows and 1 column. And C has again 1 row and 100 columns. These are matrices which look like this. Now what happens is that when I multiply d times C then I get something which is 100 into 1 into 100. So, we will get an output which is a 100 by 100 matrix, so that has 10000 entries, so it is going to take us 10,000 steps.

Now when I multiply A by this I am going to get 1 into 100 into 100 that is another 10000 step. If I do A, after I do BC and I do 10000 plus 10000, so I do 20000 steps. Now if I do it the other way, if I take A times B first then this whole thing collapses into a 1 by 1 single entry. So, I get 1 into 100 into 1 in 100 steps, I just collapse this row and this column into a single entry that is like computing one entry in a matrix multiplication with the resulting thing is exactly that one entry.

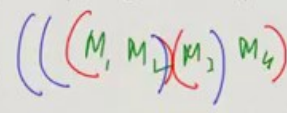
Now I have this one entry, and again I have to multiply it by this thing and that will take me for each of the columns in this I will get one entry, so that will take me 100 steps. I take 100 steps to collapse this A into B into a single cell, and another 100 steps after that to compute that product into C. So instead of 20000 steps, I have done it in 200 steps. This is the way in which the sequence of multiplications though multiplication is associative and it does not matter what you do you will get the same answer; the

sequence in which you do the associative steps can dramatically improve or worsen the amount of time you spend doing this.

(Refer Slide Time: 04:09)

Multiplying matrices

- Given matrices M_1, M_2, \dots, M_n of dimensions $[r_1, c_1], [r_2, c_2], \dots, [r_n, c_n]$
- Dimensions match, so $M_1 \times M_2 \times \dots \times M_n$ can be computed
- $c_i = r_{i+1}$ for $1 \leq i < n$
- Find an optimal order to compute the product
- That is, bracket the expression optimally



In general, we have a sequence M_1 to M_n and each of them has some rows and columns, and what we are guaranteed is that each adjacent pair can be multiplied. So, r_1, c_1, r_2, c_2 the first two are such then c_1 is equal to r_2 , the number of columns in the first matrix is equal to the number rows in second matrix, similarly, c_2 is equal to r_3 and so on. These dimensions are guaranteed to match, so the matrix multiplication is always possible.

Our target is to find out in what order we would do it. So, we can at best do two matrices at a time, we only know how to multiply A times B , we cannot take three matrices and directly multiply them. If we have to do three matrices, we have to do in two steps A times B and then C , or B times C and then A . Now same way with n matrices, we have to do two at a time, but those two at a time could be a complicated thing. I could do M_1, M_2 then I can combine that and do that combination with M_3 or I can do M_1, M_2, M_3, M_4 and then do combination of M_1, M_2 multiplied by M_3, M_4 and so on.

What is the optimal way of computing the product, what is the optimal way of putting brackets in other words? Brackets are what tell us, so when we say M_1, M_2, M_3, M_4 then one way of computing it just to do this right do M_1, M_2 , then M_3, M_4 . And other way of doing it would be to say do M_1, M_2 and then do that multiplied by M_3 and then

M 4 and so on. So, different ways of bracketing correspond to different evaluation orders for this multiplication. And what you want to do is kind of calculate without doing the actual computation which is the best sequence and which to do this calculation, so that we optimize the operations involved.

(Refer Slide Time: 05:48)

Inductive structure

- Product to be computed: $(M_1 \times M_2 \times \dots \times M_n)$
- Final step would have combined two subproducts
- $(M_1 \times M_2 \times \dots \times M_k) \times (M_{k+1} \times M_{k+2} \times \dots \times M_n)$, for some $1 \leq k < n$
- First factor has dimension (r_1, c_k) , second (r_{k+1}, c_n)
- Final multiplication step costs $O(r_1 c_k c_n)$
- Add cost of computing the two factors

What is the inductive structure? Finally, when we do this remember we **only** do two at a time. At the end, we must end with the multiplication which involves some group multiplied by some group it must look like this, and must have this whole thing collapsing to some M_1 prime, and this whole thing collapsing to M_2 prime and **finally** multiplying M_1 prime by M_2 prime. In other words M_1 prime is for some k it is from M_1 all the way up to M_k , and M_2 prime is from $k+1$ up to n . So, this is my M_1 prime and this is my M_2 prime, and this k is somewhere between 1 and n .

In the worst case I could be doing M_1 , and on the other side I can doing it I could have done M_2 up to M_n this whole thing. This whole thing is my, the other worst cases I could have done M_1 2 not worst, but extreme cases M_1 into M_{n-1} , I might have already computed, and now I want to finally, multiply it by M_n or it could be anywhere in between. If I just pick an arbitrary k then the first one is **has** r_1 rows c_k columns.

So, second one as r_{k+1} rows c_n column, but we know that c_k is equal to r_{k+1} . So this matrix will work. The final computation is going to be r_1 into c_k into c_n right, m into n into p - the rows into the column, common number of column row into the

final number of columns. So this final multiplication, we know how much it is going to cost. And to this, we have to recursively add the inductive cost of having computed the two factors; how much time it will take us to **do** M_1 prime how much time it will take us to **do** M_2 prime.

(Refer Slide Time: 07:31)

Subproblems


- Final step is
 $(M_1 \times M_2 \times \dots \times M_k) \times (M_{k+1} \times M_{k+2} \times \dots \times M_n)$
- Subproblems are $(M_1 \times M_2 \times \dots \times M_k)$ and $(M_{k+1} \times M_{k+2} \times \dots \times M_n)$
- Total cost is $\text{Cost}(M_1 \times M_2 \times \dots \times M_k) + \text{Cost}(M_{k+1} \times M_{k+2} \times \dots \times M_n) + r_1 c_k c_n$
- Which k should we choose?
- No idea! Try them all and choose the minimum!

We have that the cost of M_1 splitting at k is a cost of M_1 to M_k plus the cost of M_k plus **one** to M_n plus the last multiplication r_1 into c_k to c_n . So, clearly this cost will vary for different case, and there are many **number of cases**. We said there are n minus 1 choices of **k** , anything from 1 to n minus 1 we can choose as k . There are n minus 1 sub problems. So, **when we did** the longest common sub sequence problem, we had two sub problems. We could either drop the first letter a_i or the second letter b_j , and then we have to consider two sub problems. We had no way of knowing which is better, so we did them both and took the max.

Now here we have n minus 1 different choices of k , we **have** no way of knowing which of this case is better. So, again we try all of them and take the minimum. There we **were** doing the maximum because we want the longest **common** subsequence, here we want the minimum cost so we choose that k which minimizes this split, and recursively each of those things would minimize their split and so on. So, that is the inductive structure.

(Refer Slide Time: 08:37)

Inductive formulation



- $\text{Cost}(M_1 \times M_2 \times \dots \times M_n) =$
minimum value, for $1 \leq k < n$, of
 $\text{Cost}(M_1 \times M_2 \times \dots \times M_k) +$
 $\text{Cost}(M_{k+1} \times M_{k+2} \times \dots \times M_n) +$
 $r_1 c_k c_n$
- When we compute $\text{Cost}(M_1 \times M_2 \times \dots \times M_k)$ we will get subproblems of the form $M_j \times M_{j+1} \times \dots \times M_k$

Finally, we say that the cost of multiplying M_1 to M_n is the minimum for all choices of k of the cost of multiplying M_1 to M_k plus the cost of multiplying M_{k+1} to M_n plus. Of course, for that choice of k , we have to do one final multiplication which is to take these resulting sub matrices, so that is r_1 into c_k into c_n . So, when we take this, so we have M_1 to M_n , so we have picked M_1 to M_k .

Then as before what will happen is that we will have to split this somewhere, so we will we will now end up having some segment which is neither M_1 nor M_n , it starts at some M_j and goes to M_k . In general, we need to express this quantity for arbitrary left and right point, we cannot assume that the left hand point is 1; we cannot assume the right hand point is n right.

(Refer Slide Time: 09:25)

In general ...

$\downarrow k$

- $\text{Cost}(M_i \times M_{i+1} \times \dots \times M_j) =$
minimum value, for $i \leq k < j$, of
 $\text{Cost}(M_i \times M_{i+1} \times \dots \times M_k) +$
 $\text{Cost}(M_{k+1} \times M_{k+2} \times \dots \times M_j) +$
 $r_i c_k c_j$
- Write $\text{Cost}(i, j)$ to denote $\text{Cost}(M_i \times M_{i+1} \times \dots \times M_j)$

In general, if we have a segment from M_i to M_j , then we want the smallest value of among all the values of k from i to j minus 1, we want the minimum cost which occurs from computing the cost of M_i to M_k , and M_{k+1} to M_j , and the cost of this final multiplication which is r_i into c_k into c_j . This quantity we will write as $\text{cost } i \ j$. $\text{cost } i \ j$ is a cost of computing the segment from M_i to M_j which involves picking the best k . So, M_i to M_j is called $\text{cost } i \ j$, and we use this same recursive inductive definition choose the best.

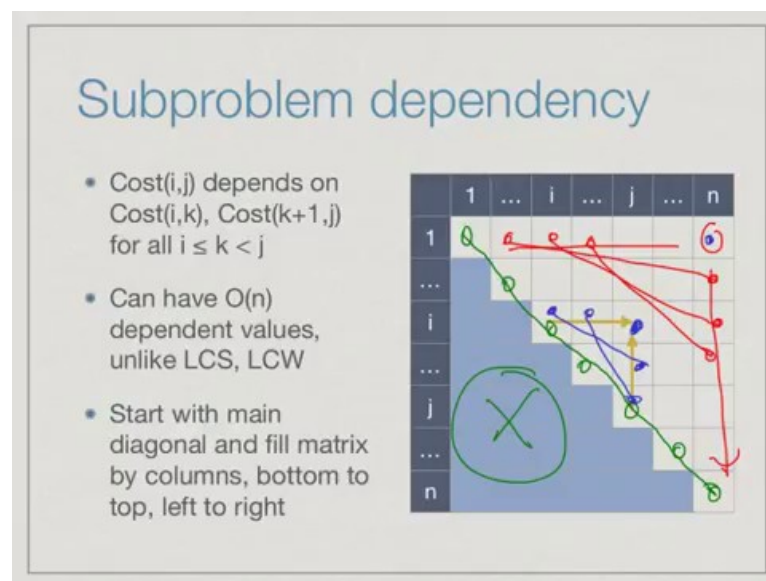
(Refer Slide Time: 10:07)

Final equation

- $\text{Cost}(i, i) = 0$ — No multiplication to be done
- $\text{Cost}(i, j) = \min \text{ over } i \leq k < j$
 $[\text{Cost}(i, k) + \text{Cost}(k+1, j) + r_i c_k c_j]$
- Note that we only require $\text{Cost}(i, j)$ when $i \leq j$

The base case well if **we are** just looking at a segment of length 1, supposing we just want to multiply one matrix from 1 to 1, or 3 to 3, or 7 to 7 nothing is to be done. It is a just matrix there is no multiplication, so the cost is 0. We can then write out this cost i, j equation saying the minimum over k of cost i, k plus cost $k+1, j$ plus $r_i \cdot c_k \cdot c_j$ which is the actual multiplication. And of course, i is always going to be less than or equal to j , because we are doing it from left to right, so we can assume that the segment is **given** to us with two end points where i is less than or equal to j .

(Refer Slide Time: 10:49)



So, i is less than or equal to j , and we look at cost i, j as a kind of matrix then this whole area where i is greater than j is ruled out. So, we only look at this diagonal. And the entries along the diagonal are the **ones** which are of the form i, i , so all these entries are initially zero right. There is no cost involved with doing any multiplication from position j to position j along this diagonal. Now, in order to compute i, j , I need to pick a k , and I need to compute for that k all the values I mean I have to compute i, k , and so it turns out that this corresponds to saying that if I want to compute a particular entry i, j , then I need to choose a good k .

And in order to choose a good k , I need so this I can express in many different ways. I can say pick, so for example, supposing I want to compute this particular thing then I have to say pick this entry i to i and then I want the entry $i+1$ to j . So, I have $i+1$

to j this entry. These two entries I have to sum up; otherwise, I have to take this entry and sum it up with this entry.

In general, if I have a thing there, I will say if I choose this entry as my k point then I must add this entry to get it up, and take this sum or I have to take this entry and add this entry and then add this and so on. In general, we could have order n values I need to compute for this I need to compute, I need all the values here and I need all the values here.

(Refer Slide Time: 12:40)

Subproblem dependency

- $\text{Cost}(i,j)$ depends on $\text{Cost}(i,k)$, $\text{Cost}(k+1,j)$ for all $i \leq k < j$
- Can have $O(n)$ dependent values, unlike LCS, LCW
- Start with main diagonal and fill matrix by columns, bottom to top, left to right

I need something to the left and to the bottom, but if I start in the diagonal then it becomes easy, because I can actually say that if I have initially these values filled in, this is the base case. Then to the left and below, I can fill in this because I know it is values to the left, I know this value to the left and below, so I can fill up this diagonal. In the next step, I can fill up this diagonal, because I have all the values to left below. Now at this entry, I have values to the left and below, so I can fill up this diagonal. So, I can fill it up diagonal by diagonal. This is the order in which I can fill up this table using this inductive definition because this is the way in which the dependency is stored.

(Refer Slide Time: 13:14)

```
MMCost(M1,...,Mn), DP

def MMC(R,C):
    # R[0..n-1],C[0..n-1] have row/column sizes

    for r in range(len(R)):
        MMC[r][r] = 0

    for c in range(1,len(R)): # c = 1,2,...,n-1
        for r in range(c-1,-1,-1): # r = c,c-1,...,0
            MMC[r][c] = infinity # Something large
            for k in range(r,c): # k = r,r+1,...,c-1
                subprob = MMC[r][k] + MMC[k][c] +
                           R[r]C[k]C[c]
            if subprob < MMC[r][c]:
                MMC[r][c] = subprob
```

This is the code for this particular thing. So, you can go through it and just check the only thing that you need to notice that we have used some.

(Refer Slide Time: 13:31)

Subproblem dependency

- Cost(i,j) depends on Cost(i,k), Cost(k+1,j) for all $i \leq k < j$
- Can have $O(n)$ dependent values, unlike LCS, LCW
- Start with main diagonal and fill matrix by columns, bottom to top, left to right

So what we are doing is, when we compute as we said one entry. Supposing, we are computing this one entry then we want to compute the minimum across many different pairs right this entry this entry and so on. Remember what we did when you computed the maximum longest **common** sub word, we assume that the maximum was zero, and every time we saw a bigger value we updated it. Here, we want the minimum entry. So,

what we do is we assume the minimum as some large number and every time we see an entry, if it is smaller than the minimum we reduce it.

(Refer Slide Time: 13:57)

```
MMCost(M1,...,Mn), DP

def MMC(R,C):
    # R[0..n-1],C[0..n-1] have row/column sizes

    for r in range(len(R)):
        MMC[r][r] = 0

    for c in range(1,len(R)): # c = 1,2,...,n-1
        for r in range(c-1,-1,-1): # r = c,c-1,...,0
            → MMC[r][c] = infinity # Something large
            for k in range(r,c) # k = r,r+1,...,c-1
                subprob = MMC[r][k] + MMC[k][c] +
                           R[r]C[k]C[c]
            if subprob < MMC[r][c]:
                MMC[r][c] = subprob
```

That is what it is happening here, when we start the loop we assume that the value for r c is actually infinity. Now what is infinity? well you can take infinity So that we can take for instance the product of all the dimensions that appear in this problem. You know that the total dimension will not be more than that. So, you can take the product of all the dimensions you can take a very large number, it does not matter something related to what your problem as. So, we have not defined it in the code. The important thing is we are computing minimum.

Instead of starting with 0, and updating it when you do maximum; you start with the large value and keep shrinking it. So every time, we find a sub problem which is smaller than the current value that we have seen then we replace that value as the new minimum, so that is all that is important here. Everything else is just a way to make sure that we go it, go through this table diagonal by diagonal, and for each diagonal we scan the row and the column and compute the minimum across all pairs in that row and column.

(Refer Slide Time: 14:50)

The slide is titled "Complexity" in blue. Below the title, the handwritten text "Complexity > Table Size" is written in black. To the right of this text is a red arrow pointing towards a green diagram. The diagram is a right-angled triangle with a green outline, representing a table of size $O(n^2)$. Below the diagram, there is a bulleted list of three points:

- As with LCS, ~~we~~, we to fill an $O(n^2)$ size table
- However, filling $MMC[i][j]$ could require examining $O(n)$ intermediate values
- Hence, overall complexity is $O(n^3)$

As with this LCS problem, we have to fill an order n squared size table, but the main difference between LCS and this is that in order to fill an entry in the LCS thing we have to look at only a constant number of entries. So, order mn or order n squared, but the point was each entry takes constant time, so the **effort involved is** the same as size of the table, $m n$ table or takes $m n$ time. Here, unfortunately **it** is not the case right. We saw that an entry will take time proportional to it is distance from the diagonal. In general, that will add an order n factor.

Though we have order n squared by two entries actually order n squared entries, we would have to account for order n work per entry because each entry **has** to scan the row to it is left in the column below not just one entry away from it. And so this whole thing becomes order n cubed and not order n square. So, it is this is some point to keep in mind that there could be problems, **where** the complexity is bigger than the table size. All the examples we saw before that is the reason to do this example.

In all the examples we saw before, the Fibonacci we had a table which is linear in **n** and it took time linear **in n to** fill it. For the grid path and for longest common subsequence, we had tables, which are m by n , but it took only n by m time or m by n **time** to fill that. Here, we have a table which is n by n , but it takes n cube time to fill it up because each entry it requires more than constant time, it actually takes time proportional to n .