Introduction to Algorithms

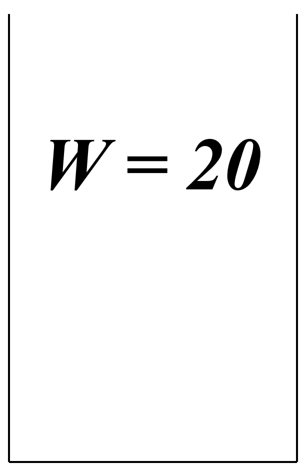
Dynamic Programming

0-1 Knapsack problem

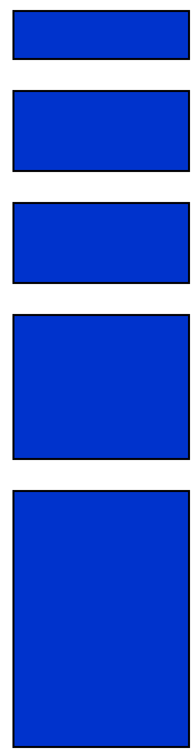
- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i , b_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem: a picture

This is a knapsack
Max weight: $W = 20$



Items



Weight w_i	***Benefit value*** b_i
2	3
3	4
4	5
5	8
9	10

0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

  - *The problem is called a "0-1" problem, because each item must be entirely accepted or rejected.*
  - *Just another version of this problem is the "Fractional Knapsack Problem", where we can take fractions of items.*

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to W
- Running time will be $O(2^n)$

0-1 Knapsack problem: brute-force approach

- Can we do better?
- Yes, with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Let's try this:

If items are labeled 1…n, then a subproblem would be to find an optimal solution for S_k = {items labeled 1, 2, … k}

Defining a Subproblem

If items are labeled *1…n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, … k}*

- This is a valid subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we <u>can't</u> do that. Explanation follows….

Defining a Subproblem

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_4=3$ $b_4=4$	

?

Max weight: $W = 20$

For S_4:

Total weight: 14;

total benefit: 20

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_4=9$ $b_4=10$

For S_5:

Total weight: 20

total benefit: 26

S_4

S_5

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Solution for S_4 is not part of the solution for S_5!!!

Defining a Subproblem (continued)

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter: w, which will represent the exact weight for each subset of items
- The subproblem then will be to compute $B[k,w]$

Recursive Formula for subproblems

- ***Recursive formula for subproblems:***

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- It means, that the best subset of S_k that has total weight w is one of the two:

1) the best subset of S_{k-1} that has total weight w,   **or**

2) the best subset of S_{k-1} that has total weight w-w_k plus the item k

Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.
- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be > w, which is unacceptable
- Second case: $w_k <= w$. Then the item k can be in the solution, and we choose the case with greater value

0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
    for w = 0 to W
        if w_i <= w // item i can be part of the solution
            if b_i + B[i-1,w-w_i] > B[i-1,w]
                B[i,w] = b_i + B[i-1,w- w_i]
            else
                B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // w_i > w
```

Running time

for w = 0 to W

    B[0,w] = 0

O(W)

for i = 0 to n

Repeat n times

    B[i,0] = 0

    for w = 0 to W

O(W)

        < the rest of the code >

What is the running time of this algorithm?

O(n*W)

Remember that the brute-force algorithm takes $O(2^n)$

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example (2)

W \ i	0	1	2	3	4
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				

for w = 0 to W

B[0,w] = 0

Example (3)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
5	0				

for i = 0 to n

B[i,0] = 0

Example (4)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0 →	0			
2	0				
3	0				
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

if $w_i <= w$ *// item i can be part of the solution*
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w-w_i]$
  else
    $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ *//* $w_i > w$

Example (5)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0				
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

```
if w_i <= w // item i can be part of the solution
   if b_i + B[i-1,w-w_i] > B[i-1,w]
      B[i,w] = b_i + B[i-1,w- w_i]
   else
      B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (6)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

if $w_i <= w$ // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Example (7)

Items:

1: (2,3)

2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

if $w_i <= w$ *// item i can be part of the solution*

  if $b_i + B[i-1,w-w_i] > B[i-1,w]$

    $B[i,w] = b_i + B[i-1,w-w_i]$

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ *//* $w_i > w$

Example (8)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0	3			

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=2$

if $w_i <= w$ *// item i can be part of the solution*
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w- w_i]$
  else
    $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ *//* $w_i > w$

Example (9)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0 →	0		
2	0	3			
3	0	3			
4	0	3			
5	0	3			

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

if $w_i \le w$ *// item i can be part of the solution*
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w-w_i]$
  else
    $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ *//* $w_i > w$

Example (10)

Items:

1: (2,3)
2: (3,4)

3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	→ 3		
3	0	3			
4	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

if $w_i <= w$ *// item i can be part of the solution*

  if $b_i + B[i-1,w-w_i] > B[i-1,w]$

    $B[i,w] = b_i + B[i-1,w- w_i]$

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ *//* $w_i > w$

Example (11)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

if $w_i <= w$ *// item i can be part of the solution*

    if $b_i + B[i-1,w-w_i] > B[i-1,w]$

        $B[i,w] = b_i + B[i-1,w- w_i]$

    else

        $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ *//* $w_i > w$

Example (12)

Items:

1: (2,3)
2: (3,4)

3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3			

$i=2$
$b_i=4$
$w_i=3$
$w=4$
$w-w_i=1$

if $w_i <= w$ *// item i can be part of the solution*
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w- w_i]$
  else
    $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ *//* $w_i > w$

Example (13)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3	7		

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

if $w_i \le w$ *// item i can be part of the solution*

  if $b_i + B[i-1,w-w_i] > B[i-1,w]$

    $B[i,w] = b_i + B[i-1,w-w_i]$

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ *//* $w_i > w$

Example (14)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4		
5	0	3	7		

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=3$

$b_i=5$

$w_i=4$

$w=1..3$

if $w_i <= w$ *// item i can be part of the solution*
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ *//* $w_i > w$

Example (15)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7		

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w - w_i=0$

if $w_i <= w$ // item i can be part of the solution
if $b_i + B[i-1,w-w_i] > B[i-1,w]$
$B[i,w] = b_i + B[i-1,w- w_i]$
else
$B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Example (15)

Items:

1: (2,3)
2: (3,4)
3: (4,5)

4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7 →	7	

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w- w_i=1$

if $w_i <= w$ *// item i can be part of the solution*

if $b_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = b_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ *//* $w_i > w$

Example (16)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0 →	0
2	0	3	3	3 →	3
3	0	3	4	4 →	4
4	0	3	4	5 →	5
5	0	3	7	7	

$i=3$

$b_i=5$

$w_i=4$

$w=1..4$

if $w_i <= w$ *// item i can be part of the solution*

    if $b_i + B[i-1,w-w_i] > B[i-1,w]$

        $B[i,w] = b_i + B[i-1,w-w_i]$

    else

        $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ *//* $w_i > w$

Example (17)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	4	5	5
5	0	3	7	7 →	7

$i=3$

$b_i=5$

$w_i=4$

$w=5$

if w_i <= w // item i can be part of the solution

if $b_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = b_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
- To know the items that make this maximum value, an addition to this algorithm is necessary
- Please see LCS algorithm from the previous lecture for the example how to extract this data from the table we built

Dynamic programming

Dynamic programming is distinct from divide-and-conquer, as the divide-and-conquer approach works well if the sub-problems are essentially unique

- Storing intermediate results would only waste memory

If sub-problems re-occur, the problem is said to have *overlapping sub-problems*

Matrix chain multiplication

Suppose $\mathbf{A}$ is $k \times m$ and $\mathbf{B}$ is $m \times n$

- Then $\mathbf{AB}$ is $k \times n$ and calculating $\mathbf{AB}$ is $\Theta(kmn)$
- The number of multiplications is given exactly kmn

Suppose we are multiplying three matrices $\mathbf{ABC}$

- Matrix multiplication is associative so we may choose $(\mathbf{AB})\mathbf{C}$ or $\mathbf{A}(\mathbf{BC})$
- The order of the multiplications may significantly affect the run time

For example, if $\mathbf{A}$ and $\mathbf{B}$ are $n \times n$ matrices and $\mathbf{v}$ is an n-dimensional column vector:

- Calculating $(\mathbf{AB})\mathbf{v}$ is $\Theta(n^3)$
- Calculating $\mathbf{A}(\mathbf{Bv})$ is $\Theta(n^2)$

Matrix chain multiplication

Suppose we want to multiply four matrices $\mathbf{ABCD}$

- There are may ways of parenthesizing this product:

  $((\mathbf{AB})\mathbf{C})\mathbf{D}$

  $(\mathbf{AB})(\mathbf{CD})$

  $(\mathbf{A}(\mathbf{BC}))\mathbf{D}$

  $\mathbf{A}((\mathbf{BC})\mathbf{D})$

  $\mathbf{A}(\mathbf{B}(\mathbf{CD}))$

- Which has the least number of operations?

Matrix chain multiplication

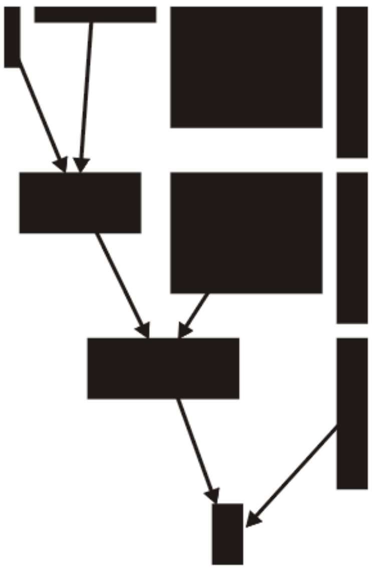
For example, consider these four:

Matrix	Dimensions
A	20×5
B	5×40
C	40×50
D	50×10

Matrix chain multiplication

Considering each order:

$((\mathbf{AB})\mathbf{C})\mathbf{D}$



The required number of multiplications is:

$\mathbf{AB}$	$20 \times 5 \times 40 = 4000$
$(\mathbf{AB})\mathbf{C}$	$20 \times 40 \times 50 = 40000$
$((\mathbf{AB})\mathbf{C})\mathbf{D}$	$20 \times 50 \times 10 = 10000$

This totals to 54000 multiplications

Matrix chain multiplication

Considering the next order:

$$(\mathbf{AB})(\mathbf{CD})$$

The required number of multiplications is:

$$\begin{array}{lr} \mathbf{AB} & 20 \times \ \ 5 \times 40 = \ \ 4000 \\ \mathbf{CD} & 40 \times 50 \times 10 = 20000 \\ (\mathbf{AB})(\mathbf{CD}) & 20 \times 40 \times 10 = \ \ 8000 \end{array}$$

This totals to 32000 multiplications

Matrix chain multiplication

Considering the next order:

$$(\mathbf{A}(\mathbf{BC}))\mathbf{D}$$

The required number of multiplications is:

$\mathbf{BC}$	$5 \times 40 \times 50 = 10000$
$\mathbf{A}(\mathbf{BC})$	$20 \times 5 \times 50 = 5000$
$(\mathbf{A}(\mathbf{BC}))\mathbf{D}$	$20 \times 50 \times 10 = 10000$

This totals to 25000 multiplications

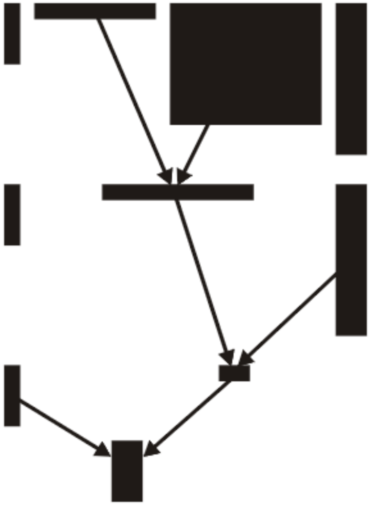
Matrix chain multiplication

And the the next order:

$$\mathbf{A((BC)D)}$$

The required number of multiplications is:

$\mathbf{BC}$	$5 \times 40 \times 50 = 10000$
$\mathbf{(BC)D}$	$5 \times 50 \times 10 = 2500$
$\mathbf{A((BC)D)}$	$20 \times 5 \times 10 = 1000$



This totals to 13500 multiplications

Matrix chain multiplication

Repeating this for the last, we get the following table:

Order	Multiplications
((**AB**)**C**)**D**	54000
(**AB**)(**CD**)	32000
(**A**(**BC**))**D**	25000
A((**BC**)**D**)	13500
A(**B**(**CD**))	23000

The optimal run time uses **A**((**BC**)**D**)

Matrix chain multiplication

Thus, the optimal run time may be found by calculating the product in the order

$$\mathbf{A}((\mathbf{BC})\mathbf{D})$$

Problem: What if we are multiply n matrices?

Matrix chain multiplication

Can we generate a greedy algorithm to achieve this?

- Greedy by the smallest number of operations?
  - Unfortunately, $2 \times 1 \times 2 + 2 \times 2 \times 3 = 16 > 12 = 1 \times 2 \times 3 + 2 \times 1 \times 3$

    even though $2 \times 1 \times 2 = 4 < 6 = 1 \times 2 \times 3$

$$\begin{pmatrix} 3 \\ 2 \end{pmatrix} \begin{pmatrix} 6 & 4 \end{pmatrix} \begin{pmatrix} 3 & 8 & 9 \\ 1 & 0 & 5 \end{pmatrix}$$

- Greedy by generating the *smallest* matrices (sum of dimensions)?
  - Unfortunately, $2 \times 1 \times 2 + 2 \times 2 \times 4 = 20 > 16 = 1 \times 2 \times 4 + 2 \times 1 \times 4$

    even though $2 + 2 = 4 < 5 = 1 + 4$

$$\begin{pmatrix} 3 \\ 2 \end{pmatrix} \begin{pmatrix} 6 & 4 \end{pmatrix} \begin{pmatrix} 2 & 7 & 9 & 4 \\ 0 & 8 & 1 & 5 \end{pmatrix}$$

Matrix chain multiplication

If we are multiplying $\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4\cdots\mathbf{A}_n$, starting top-down, there are $n-1$ different ways of parenthesizing this sequence:

$$(\mathbf{A}_1)(\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4\cdots\mathbf{A}_n)$$
$$(\mathbf{A}_1\mathbf{A}_2)(\mathbf{A}_3\mathbf{A}_4\cdots\mathbf{A}_n)$$
$$(\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3)(\mathbf{A}_4\cdots\mathbf{A}_n)$$
$$\cdots$$
$$(\mathbf{A}_1\cdots\mathbf{A}_{n-2})(\mathbf{A}_{n-1}\mathbf{A}_n)$$
$$(\mathbf{A}_1\cdots\mathbf{A}_{n-2}\mathbf{A}_{n-1})(\mathbf{A}_n)$$

For each one we must ask:

- What is the work required to perform this multiplication
- What is the minimal amount of work required to perform both of the other products?

Matrix chain multiplication

For example, in finding the best product of

$$(\mathbf{A}_1 \cdots \mathbf{A}_i)(\mathbf{A}_{i+1} \cdots \mathbf{A}_n)$$

the work required is:

- The product $\text{columns}(\mathbf{A}_1)\ \text{rows}(\mathbf{A}_i)\ \text{rows}(\mathbf{A}_n)$
  - Note that $\text{rows}(\mathbf{A}_i)$ and $\text{columns}(\mathbf{A}_{i+1})$ must be equal
- The minimal work required to multiply $\mathbf{A}_1 \cdots \mathbf{A}_i$
- The minimal work required to multiply $\mathbf{A}_{i+1} \cdots \mathbf{A}_n$

Matrix chain multiplication

```
int matrix_chain( Matrix *ms, int i, int j ) {
    // There is only one matrix
    if ( i + 1 == j ) {
            return 0;
    } else if ( i + 2 == j ) {
            assert( ms[i].columns() == ms[i + 1].rows() );
            return ms[i].rows() * ms[i].columns() * ms[i + 1].columns();
    }

    // We are multiplying at least three matrices
    // Start with calculating the work for M[i] * (M[i + 1] * ... * M[j - 1])
    assert( ms[i].columns() == ms[i + 1].rows() );
    int minimum = matrix_chain( ms, i + 1, j ) + ms[i].rows() * ms[i].columns() * ms[j - 1].columns();

    for ( int k = i + 2; k < j; ++k ) {
            // Find the work for (M[i] * ... * M[k - 1]) * (M[k] * ... M[j - 1]) and update if it is less
            assert( ms[k - 1].columns() == ms[k].rows() );
            int current = matrix_chain( ms, i, k ) + matrix_chain( ms, k, j ) +
                        + ms[i].rows() * ms[k].rows() * ms[j - 1].columns();

            if ( current < minimum ) {
                    minimum = current;
            }
    }

    return minimum;
}
```

Matrix chain multiplication

Because of the recursive nature, we will on numerous occasions be asking for the optimal behaviour of a given subsequence

- We will asked the optimal way to multiply $\mathbf{A}_3 \cdots \mathbf{A}_{n-2}$ when we ask the optimal way to multiply any of the following:

$$\mathbf{A}_1 (\mathbf{A}_2 (((\mathbf{A}_3 \cdots \mathbf{A}_{n-2}) \mathbf{A}_{n-1})\mathbf{A}_n))$$
$$\mathbf{A}_1 (\mathbf{A}_2 ((\mathbf{A}_3 \cdots \mathbf{A}_{n-2}) (\mathbf{A}_{n-1} \mathbf{A}_n)))$$
$$\mathbf{A}_1 ((\mathbf{A}_2 (\mathbf{A}_3 \cdots \mathbf{A}_{n-2})) (\mathbf{A}_{n-1}\mathbf{A}_n))$$
$$\mathbf{A}_1 ((\mathbf{A}_2 ((\mathbf{A}_3 \cdots \mathbf{A}_{n-2}) \mathbf{A}_{n-1})) \mathbf{A}_n)$$
$$\mathbf{A}_1 ((\mathbf{A}_2 (\mathbf{A}_3 \cdots \mathbf{A}_{n-2})) \mathbf{A}_{n-1}) \mathbf{A}_n)$$
$$(\mathbf{A}_1 \mathbf{A}_2) (((\mathbf{A}_3 \cdots \mathbf{A}_{n-2}) \mathbf{A}_{n-1}) \mathbf{A}_n)$$
$$(\mathbf{A}_1 \mathbf{A}_2) ((\mathbf{A}_3 \cdots \mathbf{A}_{n-2}) (\mathbf{A}_{n-1} \mathbf{A}_n))$$
$$(\mathbf{A}_1 (\mathbf{A}_2 ((\mathbf{A}_3 \cdots \mathbf{A}_{n-2}))) (\mathbf{A}_{n-1} \mathbf{A}_n)$$
$$((\mathbf{A}_1 \mathbf{A}_2) (\mathbf{A}_3 \cdots \mathbf{A}_{n-2}))(\mathbf{A}_{n-1} \mathbf{A}_n)$$
$$(\mathbf{A}_1 (\mathbf{A}_2 (\mathbf{A}_3 \cdots \mathbf{A}_{n-2}) \mathbf{A}_{n-1}))) \mathbf{A}_n$$
$$(\mathbf{A}_1 ((\mathbf{A}_2 (\mathbf{A}_3 \cdots \mathbf{A}_{n-2}))\mathbf{A}_{n-1})) \mathbf{A}_n$$
$$((\mathbf{A}_1 \mathbf{A}_2) ((\mathbf{A}_3 \cdots \mathbf{A}_{n-2}) \mathbf{A}_{n-1}) \mathbf{A}_n$$
$$((\mathbf{A}_1 (\mathbf{A}_2 (\mathbf{A}_3 \cdots \mathbf{A}_{n-2}))) \mathbf{A}_{n-1}) \mathbf{A}_n$$
$$(((\mathbf{A}_1 \mathbf{A}_2) (\mathbf{A}_3 \cdots \mathbf{A}_{n-2}))\mathbf{A}_{n-1}) \mathbf{A}_n$$

Matrix chain multiplication

The actual number of possible orderings is given by the following recurrence relation:

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(n)P(n-k) & n > 1 \end{cases}$$

Without proof, this recurrence relation is solved by $P(n) =$ $C(n-1)$ where $C(n-1)$ is the $(n-1)^{\text{th}}$ Catalan number

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Theta\left(\frac{4^n}{n^{3/2}}\right)$$

$C(1) = 1$
$C(2) = 2$
$C(3) = 5$
$C(4) = 14$
$C(5) = 42$
$C(6) = 132$
$C(7) = 429$
$C(8) = 1430$
$C(9) = 4862$
$C(10) = 16796$

Matrix chain multiplication

The number of function calls of the implementation is given by

$$T(n) = \begin{cases} 1 & n = 1,2 \\ 4 \times 3^{n-3} & n \geq 3 \end{cases} = \Theta(3^n)$$

Can we speed this up?

- Memoization: once we've found the optimal number of solutions for a given sequence, store it

Matrix chain multiplication

Associate a pair (i, j) with an integer

```
#include <map>
#include <utility>

int matrix_chain_memo( Matrix *ms, int i, int j, bool clear = true ) {
    static std::map< std::pair< int, int >, int > memo;

    if ( clear ) {
        memo.clear();
    }

    if ( i + 1 == j ) {
        return 0;
    } else if ( memo[std::pair<int, int>(i, j)] == 0 ) {
        if ( i + 2 == j ) {
            memo[std::pair<int, int>(i, j)] = ms[i].rows() * ms[i].columns() * ms[i + 1].columns();
        } else {
            int minimum = matrix_chain_memo( ms, i + 1, j, false )
                        + ms[i].rows() * ms[i].columns() * ms[j - 1].columns();

            for ( int k = i + 2; k < j; ++k ) {
                int current = matrix_chain_memo( ms, i, k, false ) + matrix_chain_memo( ms, k, j, false )
                            + ms[i].rows() * ms[k].rows() * ms[j - 1].columns();

                if ( current < minimum ) {
                    minimum = current;
                }
            }

            memo[std::pair<int, int>(i, j)] = minimum;
        }
    }

    return memo[std::pair<int, int>(i, j)];
}
```

Matrix chain multiplication

Our memoized version now runs in

$$T(n) = \left\{ \begin{matrix} 1 & n = 1 \\ (n-1)(n-1) & n \geq 2 \end{matrix} \right\} = \Theta(n^2)$$

This is a top-down implementation

- Can we implement a bottom-up version?

Matrix chain multiplication

For a bottom-up implementation, we need a matrix that stores our best current solution to $\mathbf{A}_i$ to $\mathbf{A}_j$:

	$j = 1$	2	3	4	$\cdots$	n
$i = 1$	0					
2		0			$\cdots$	
3			0		$\cdots$	
4				0	$\cdots$	
					$\ddots$	
n						0

Matrix chain multiplication

As we calculate the minimum number of multiplications required for a specific sequence $\mathbf{A}_i \cdots \mathbf{A}_j$, we fill the entry a_{ij} in the table

	$j = 1$	2	3	4	$\cdots$	n
$i = 1$	0					
2		0			$\cdots$	
3			0		$\cdots$	
4				0	$\cdots$	
					$\ddots$	
n						0

Matrix chain multiplication

This table has n^2 entries, and therefore our run time must be at least $\mathbf{\Omega}(n^2)$

However, at each step, the actual run time is $\mathbf{\Theta}(n^3)$

Matrix chain multiplication

For example, given the previous example

Matrix	Dimensions
$\mathbf{A}_1$	20×5
$\mathbf{A}_2$	5×40
$\mathbf{A}_3$	40×50
$\mathbf{A}_4$	50×10

we can calculate the table as follows...

Matrix chain multiplication

We can calculate the off-diagonal easily:

Matrix	Dimensions
$\mathbf{A}_1$	20×5
$\mathbf{A}_2$	5×40
$\mathbf{A}_3$	40×50
$\mathbf{A}_4$	50×10

	1	2	3	4
1	0 20×5	4000 20×40		
2		0 5×40	10000 5×50	
3			0 40×50	20000 40×10
4				0 50×10

Matrix chain multiplication

Next, we may calculate either:

$\mathbf{A}_1(\mathbf{A}_2\mathbf{A}_3)$ $\quad 20 \times 5 \times 50 + 10000 = 15000$

$(\mathbf{A}_1\mathbf{A}_2)\mathbf{A}_3$ $\quad 4000 + 20 \times 40 \times 50 = 44000$

Matrix	Dimensions
$\mathbf{A}_1$	20×5
$\mathbf{A}_2$	5×40
$\mathbf{A}_3$	40×50
$\mathbf{A}_4$	50×10

	1	2	3	4
1	0 20×5	4000 20×40	15000 20×50	
2		0 5×40	10000 5×50	
3			0 40×50	24000 40×10
4				0 50×10

Matrix chain multiplication

We continue:

$\mathbf{A}_2(\mathbf{A}_3\mathbf{A}_4)$ $\quad 5 \times 40 \times 10 + 20000 = 22000$

$(\mathbf{A}_2\mathbf{A}_3)\mathbf{A}_4$ $\quad 10000 + 5 \times 50 \times 10 = 12500$

Matrix	Dimensions
$\mathbf{A}_1$	20×5
$\mathbf{A}_2$	5×40
$\mathbf{A}_3$	40×50
$\mathbf{A}_4$	50×10

	1	2	3	4
1	0 20×5	4000 20×40	15000 20×50	
2		0 5×40	10000 5×50	12500 5×10
3			0 40×50	20000 40×10
4				0 50×10

Matrix chain multiplication

Finally we calculate:

$A_1((A_2A_3)A_4)$	$20 \times 5 \times 10 + 12500 \quad = 13500$
$(A_1A_2)(A_3A_4)$	$4000 + 20 \times 40 \times 10 + 20000 = 32000$
$(A_1(A_2A_3))A_4$	$15000 + 20 \times 50 \times 10 = 25000$

Matrix	Dimensions
A_1	20×5
A_2	5×40
A_3	40×50
A_4	50×10

	1	2	3	4
1	0 20×5	4000 20×40	15000 20×50	13500 20×10
2		0 5×40	10000 5×50	12500 5×10
3			0 40×50	20000 40×10
4				0 50×10

Matrix chain multiplication

Thus, counting the number of calculations required (each $\Theta(1)$):

$$n-1$$
$$(n-2)\ 2$$
$$(n-3)\ 3$$

which suggests the sum

$$\sum_{i=1}^{n-1}(n-i)i = n\sum_{i=1}^{n-1} i - \sum_{i=1}^{n-1} i^2 = \frac{n^3-n^2}{2} - \frac{n(n-1)(2n-1)}{6} = \frac{n^3-n}{6}$$

Therefore, the run time is $\Theta(n^3)$

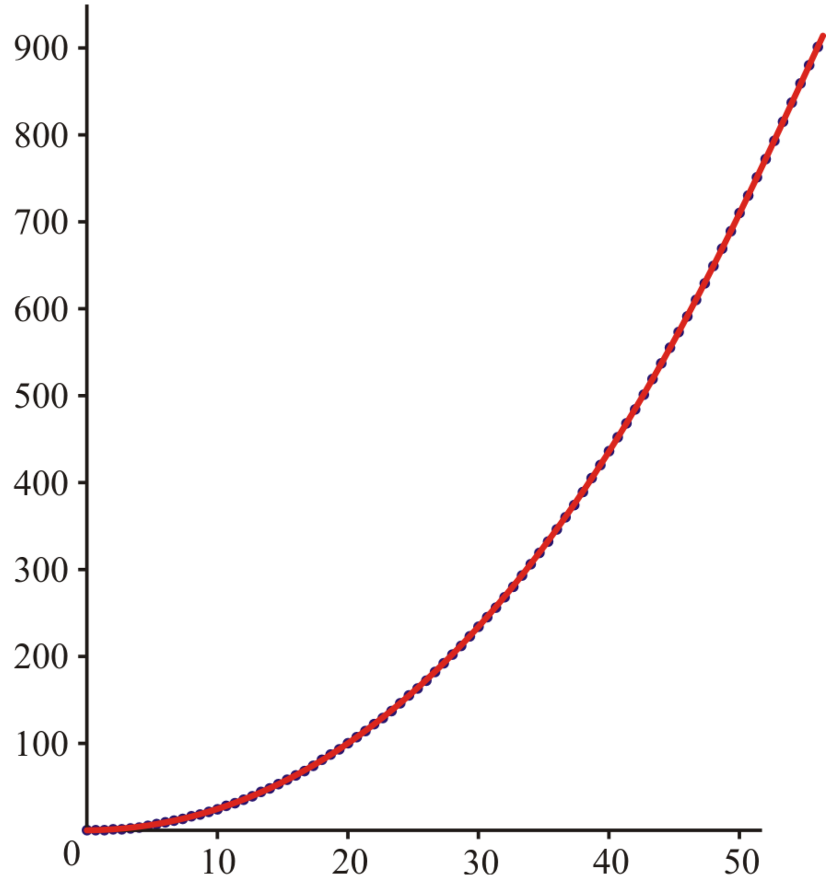
Matrix chain multiplication

```
int matrix_chain_iterative( Matrix *ms, int n ) {
    int array[n][n];

    for ( int i = 0; i < n; i++ ) {
       array[i][i] = 0;
    }

    for ( int i = 1; i < n; i++ ) {
        for ( int j = 0; j < n - i; j++ ) {
            array[j][j + i] = array[j][j] + array[j + 1][j + i]
                            + ms[j].rows()*ms[j + 1].rows()*ms[j + i].columns();

            for ( int k = j + 1; k < j + i; k++) {
                int current = array[j][k] + array[k + 1][j + i]
                            + ms[j].rows()*ms[k + 1].rows()*ms[j + i].columns();

                if ( current < array[j][j + i] ) {
                    array[j][j + i] = current;
                }
            }
        }
    }

    return array[0][n - 1];
}
```

Matrix chain multiplication

How can you estimate run times?

- Which is faster—the top-down implementation with memoization or the bottom-up implementation?
- You could plot run times…
- Question: what is the growth of this plotted data?
  - Quadratic?
  - Cubic?
  - $\Theta(n^2 \ln(n))$



Matrix chain multiplication

If a function grows in polynomial time, it is of the form:

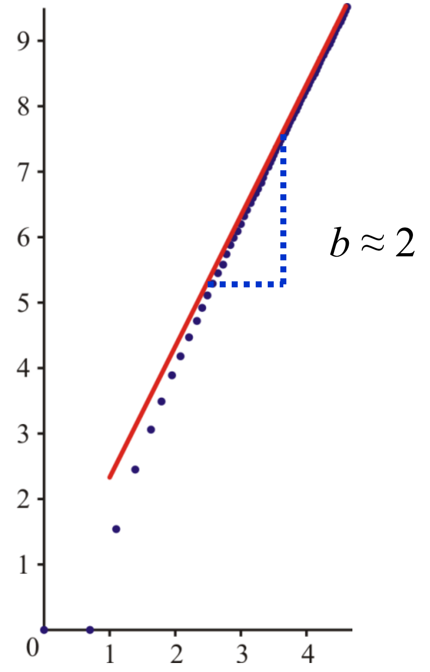
$$T(n) = an^b$$

Take the logarithm of both sides:

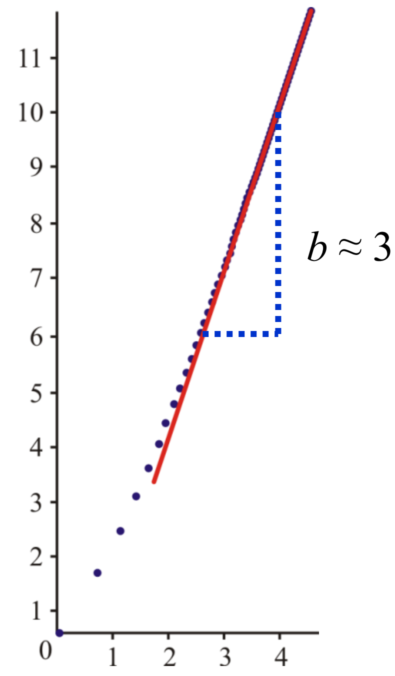
$$\ln(T(n)) = \ln(an^b) = \ln(a) + b\ln(n)$$

- It grows linearly with a slope b

Top-down with memoization



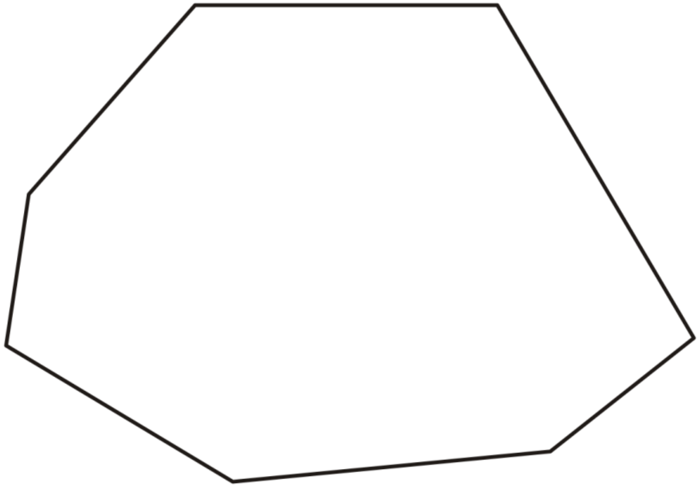
Bottom-up



Optimal polygon triangulation

In graphics and geometry, convex polygons are a basic unit

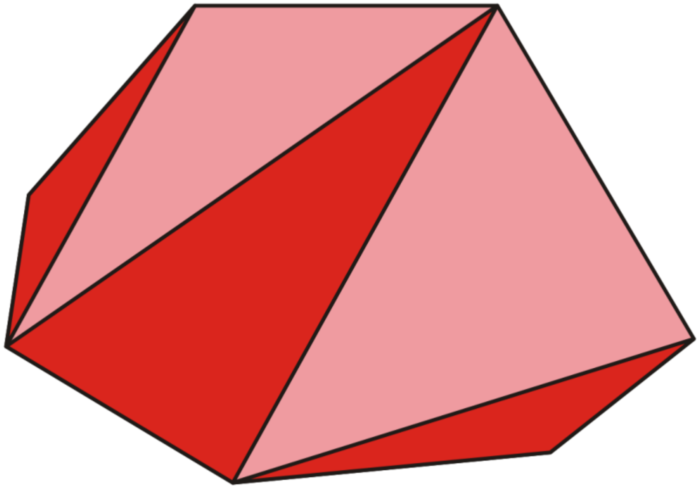
- Applications in graphics and finite-element methods



Optimal polygon triangulation

In graphics and geometry, convex polygons are a basic unit

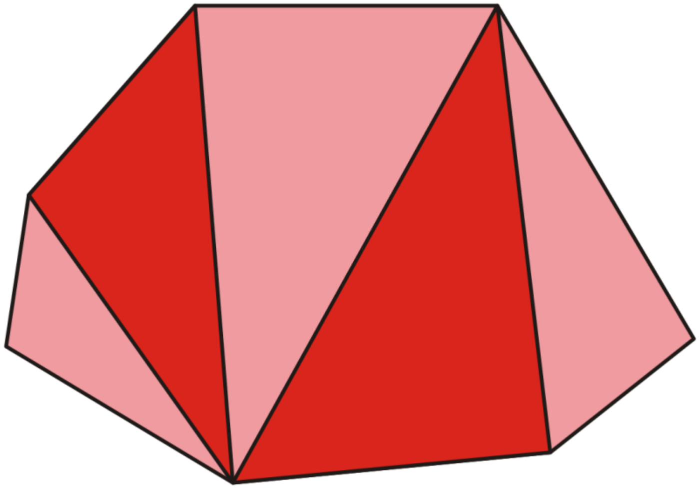
- Dividing such a polygon into simpler triangles is a common operation



Optimal polygon triangulation

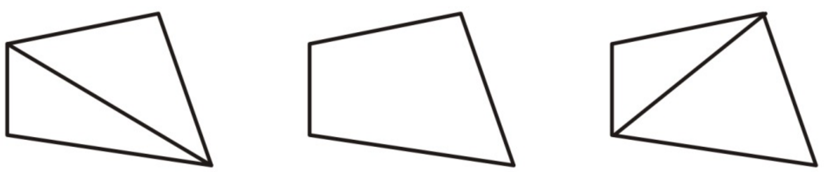
In graphics and geometry, convex polygons are a basic unit

- Some triangulations may be *better* than others
- For example,



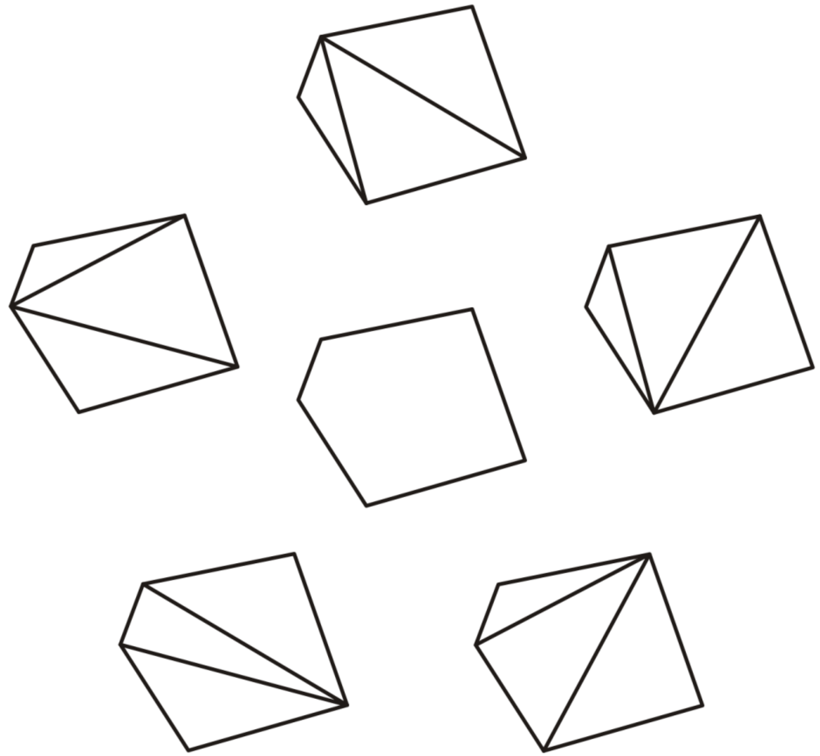
Optimal polygon triangulation

Now, a convex quadrilateral (tetragon) can only be triangulated in two different ways



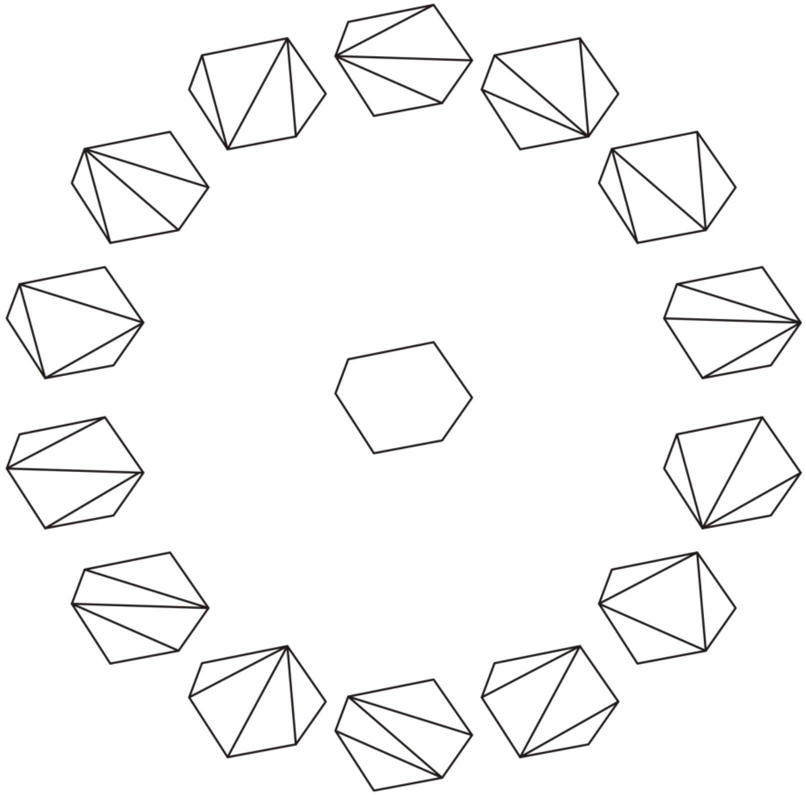
Optimal polygon triangulation

A convex pentagon can be triangulated in five different ways



Optimal polygon triangulation

And a convex hexagon can be triangulated in 14 different ways



Optimal polygon triangulation

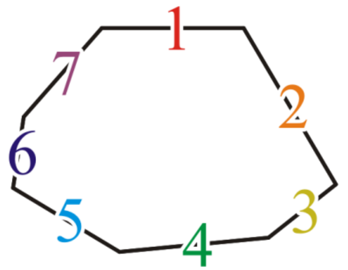
If we can put a weight on each generated triangle, can we find an optimal triangulation?

- Can we come up with a good algorithm?
- Consider the previous problem of finding an optimal order for multiplying matrices

Optimal polygon triangulation

Choose a side and begin numbering the sides in order

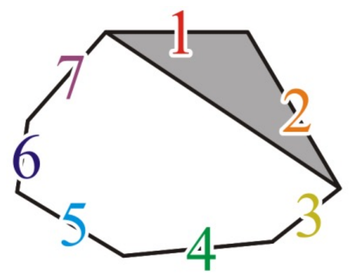
- Any two adjacent sides can be joined to create a triangle
- Any two adjacent matrices could be multiplied



$$\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4\mathbf{A}_5\mathbf{A}_6\mathbf{A}_7$$

Optimal polygon triangulation

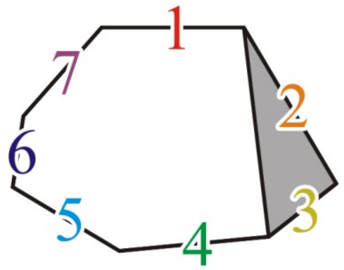
Taking two adjacent sides and creating a triangle is similar to bracketing



$(A_1A_2)A_3A_4A_5A_6A_7$

Optimal polygon triangulation

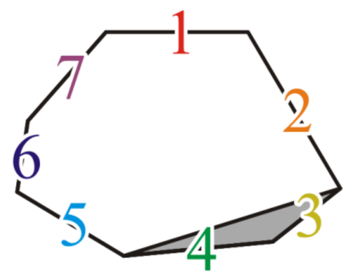
Instead of sides 1 and 2, we could choose 2 and 3



$A_1(A_2A_3)A_4A_5A_6A_7$

Optimal polygon triangulation

Or 3 and 4

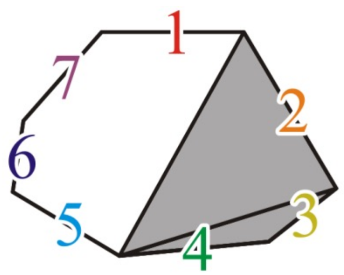


$A_1A_2(A_3A_4)A_5A_6A_7$

Optimal polygon triangulation

Suppose the triangle 3/4 was optimal

- Next, do we add the side 2?

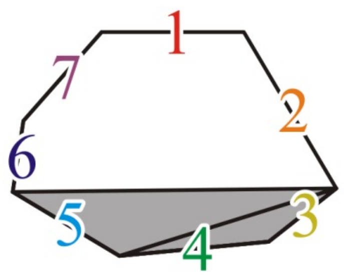


$A_1(A_2(A_3A_4))A_5A_6A_7$

Optimal polygon triangulation

Suppose the triangle 3/4 was optimal

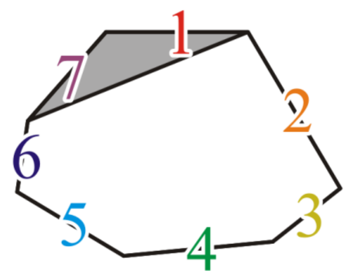
- Or do we add the side 5?



$A_1 A_2 ((A_3 A_4) A_5) A_6 A_7$

Optimal polygon triangulation

The analogy is not exact because there is no logic to bracketing and multiplying matrices $\mathbf{A}_1$ and $\mathbf{A}_7$



$\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4\mathbf{A}_5\mathbf{A}_6\mathbf{A}_7$

Never-the-less, this strongly suggests that there is an efficient algorithm based on dynamic programming that will find an optimal triangulation