Lecture 13: Dynamic Programming 3

CS 341: Algorithms

Thursday, Feb 28th 2019

Outline For Today

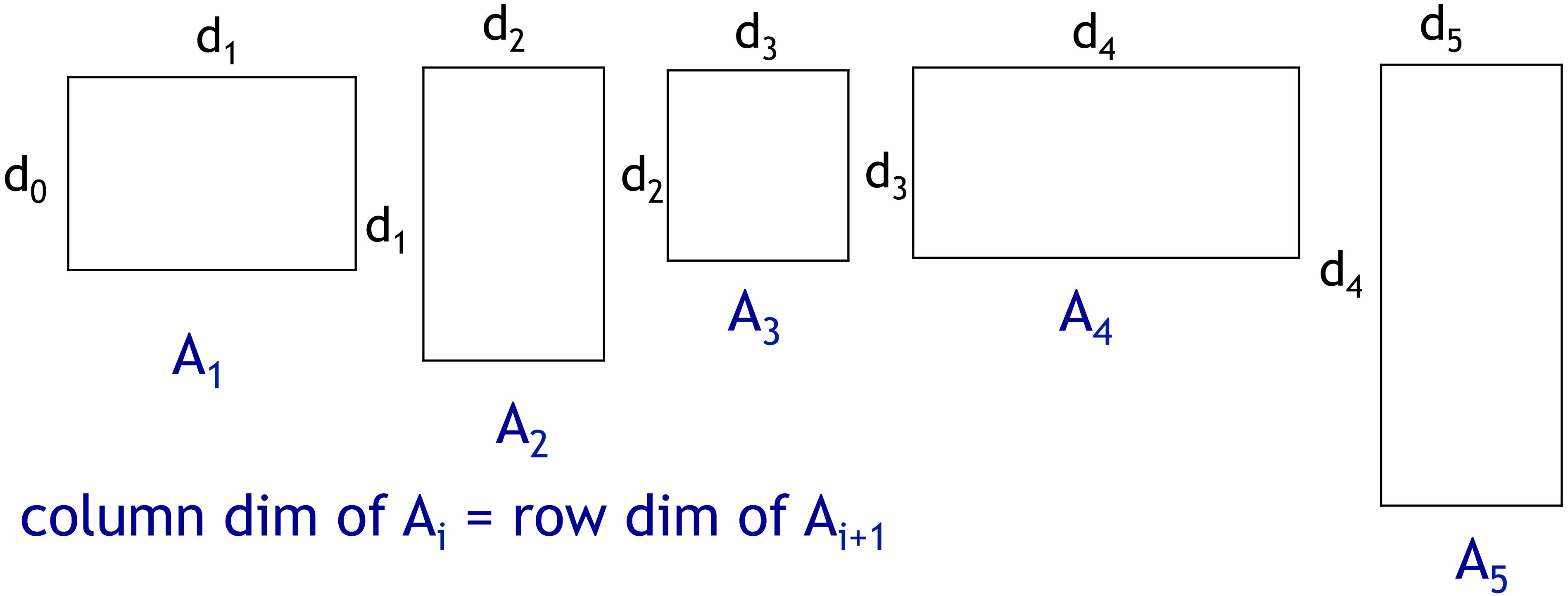
1. Matrix Multiplication Order
2. 0/1 Integer Weight Knapsack

Outline For Today

1. Matrix Multiplication Order
2. 0/1 Integer Weight Knapsack

Matrix-chain Multiplication (Ch 15.2)

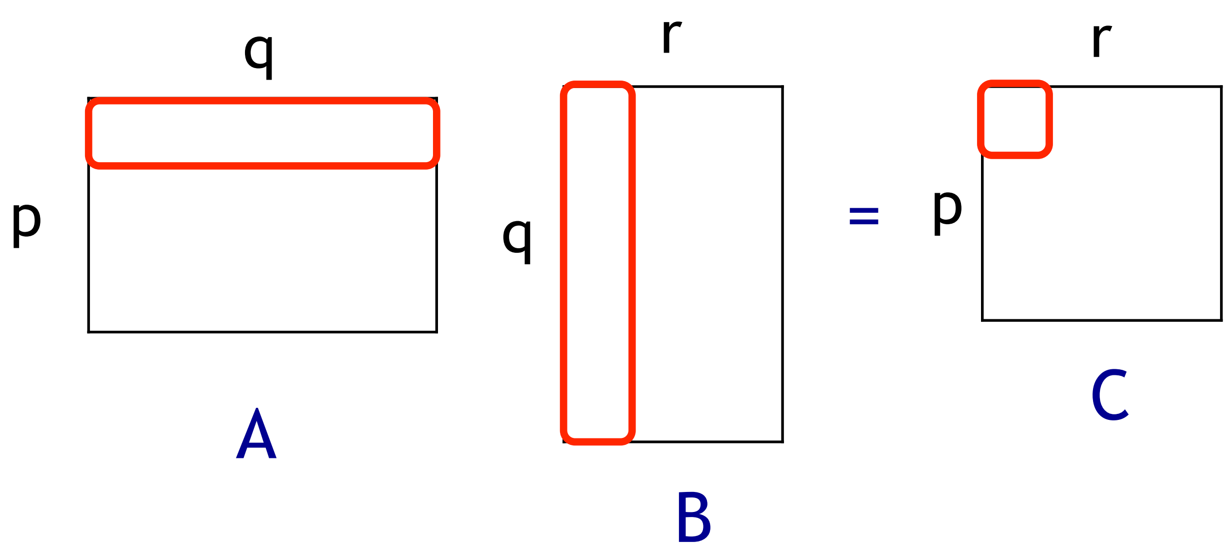
- Input: dimensions of n rectangular matrices



column dim of A_i = row dim of A_{i+1}

- Output: Find order of multiplying matrices with min cost using standard matrix multiplication

Cost of AxB By Standard MM Algorithm



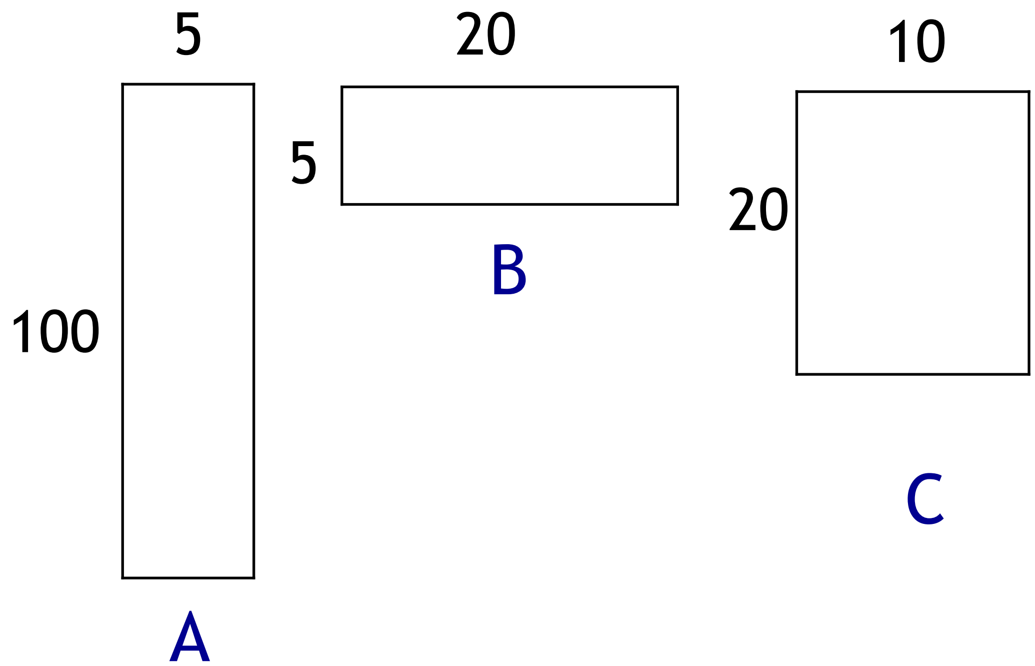
Q: How many operations will be done for each cell of C?

A: q

Q: How many operations in total?

A:pqr

Example of 2 Different Orders (1)



2 Different Orders you can multiply

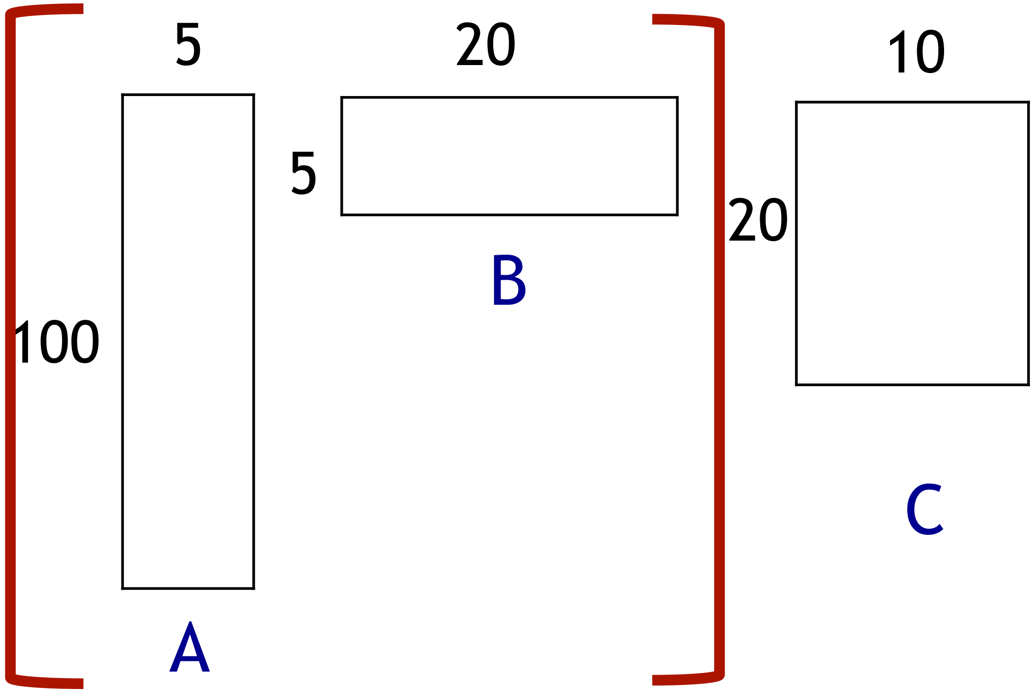
(1)

(AxB)xC

(2)

Ax(BxC)

Example of 2 Different Orders (2)

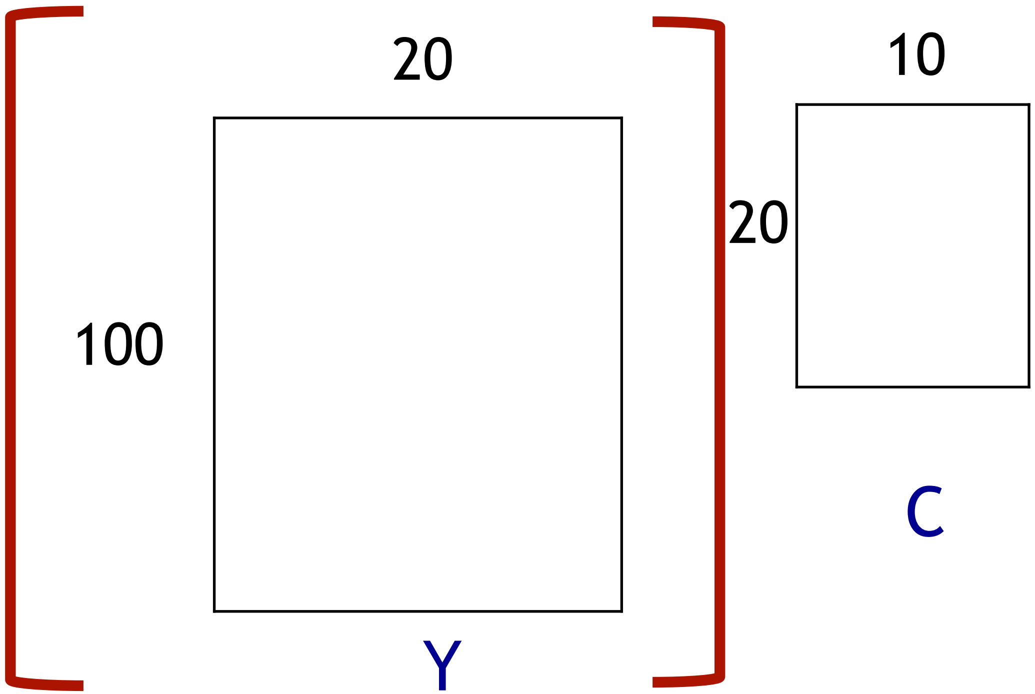


(1)

(AxB)xC

Cost of (AxB) = 100 x 5 x 20 = 10000

Example of 2 Different Orders (2)



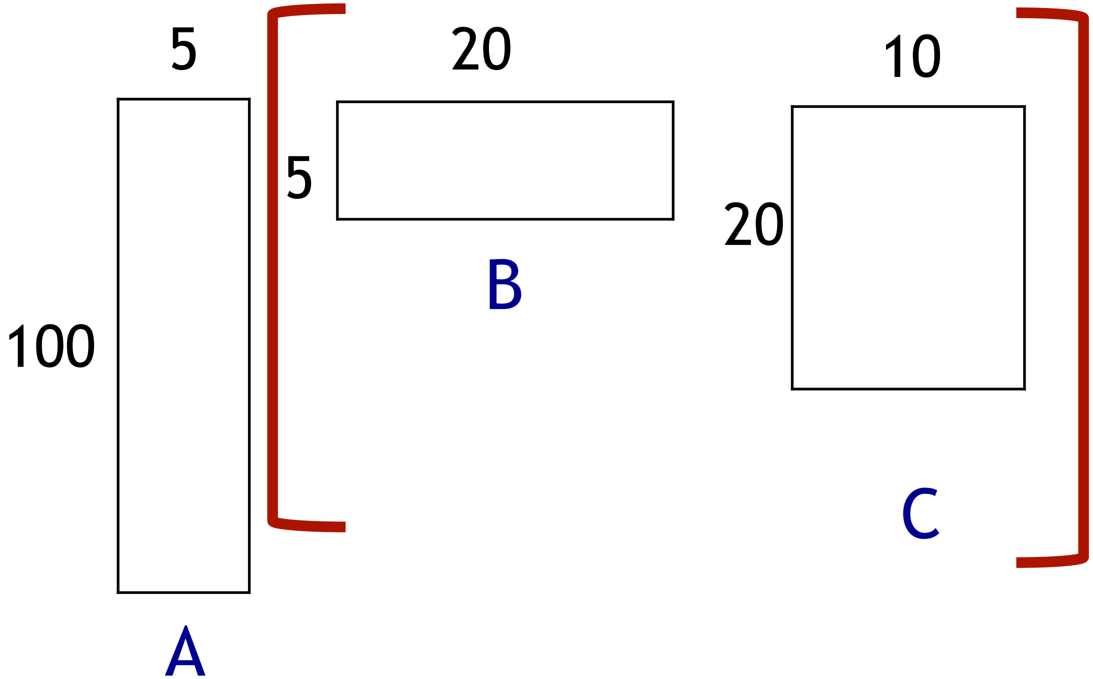
(1) Cost of (AxB) = 100 x 5 x 20 = 10000

(AxB)xC Let Y = AxB (of dims 100x20)

Cost of YxC = 100x20x10 = 20000

Total Cost: 10000 + 20000 = 30000

Example of 2 Different Orders (3)

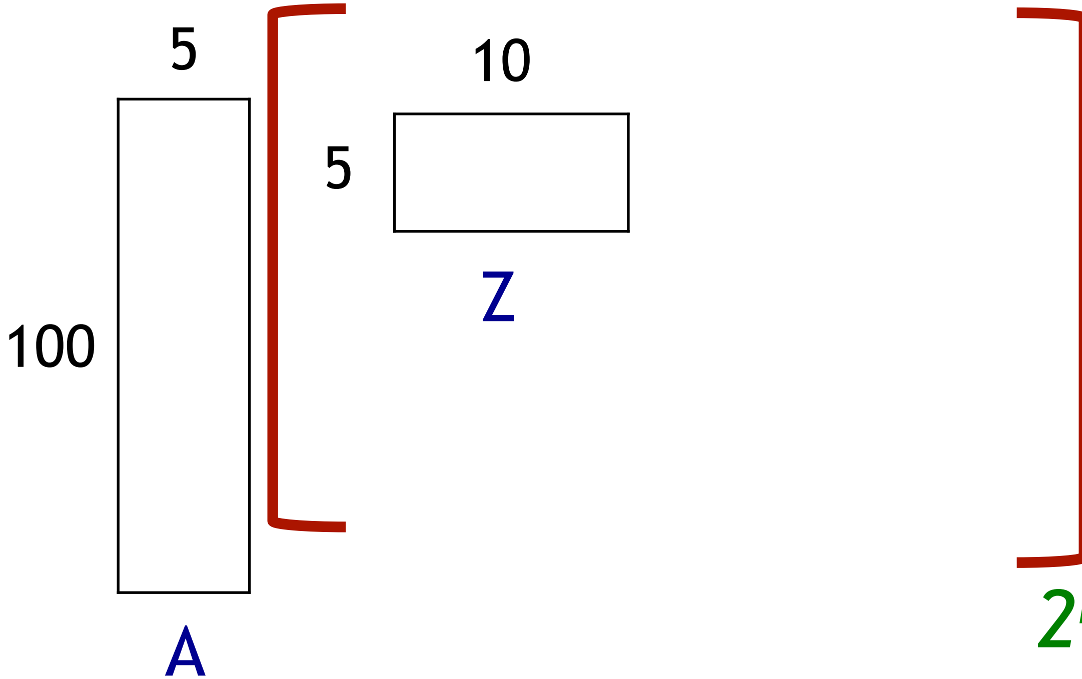


(2)

Cost of (BxC) = 5 x 20 x 10 = 1000

Ax(BxC)

Example of 2 Different Orders (3)



2^{nd} order is 5x faster!

(2) Cost of (BxC) = 5 x 20 x 10 = 1000

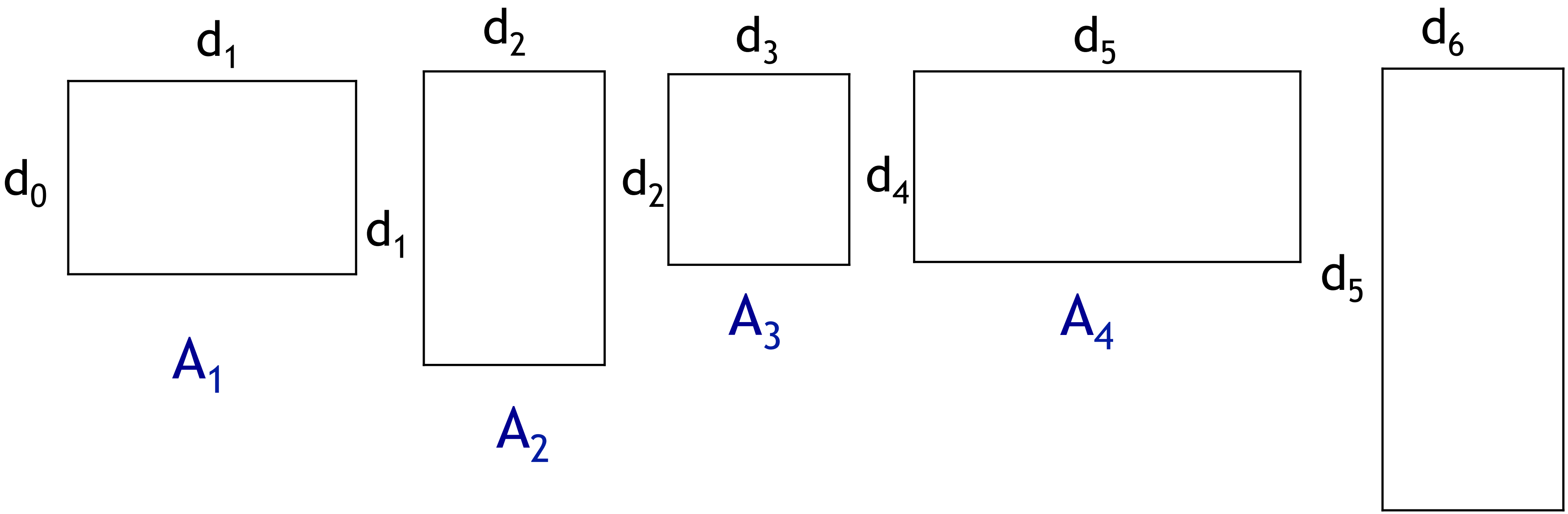
Ax(BxC) Let Z = BxC (of dims 5x10)

Cost of AxZ = 100x5x10 = 5000

Total Cost: 1000 + 5000 = 6000

Recall Our Input & Output

- Input: dimensions of n rectangular matrices



- Output: Find order of multiplying with min cost

Our input are NOT the matrices, just their dimensions.

Our goal is NOT to multiply the n matrices

Our goal is to find the min cost order of multiplications

Different Ways of Multiplying n Matrices?

Let's think about the final multiplication of an order M_1 x M_2

$$\underbrace{A_1}_{M_1} \times \underbrace{(A_2 \times A_3 \times \ldots \times A_n)}_{M_2}$$

Recursively: M_2 can be computed in many different ways as well.

$$\underbrace{(A_1 \times A_2)}_{M_1} \times \underbrace{(A_3 \times \ldots \times A_n)}_{M_2}$$

...

$$\underbrace{(A_1 \times A_2 \times A_3 \times \ldots A_{n-1})}_{M_1} \times \underbrace{A_n}_{M_2}$$

Let T(i): # ways i matrices can be multiplied

Then:

$T(n) = T(1)T(n-1) +$
$T(2)T(n-2) +$
...
$T(n-1)T(1)$

$$T(n) = \sum_{k=1}^{n-1} T(k)T(n-k)$$

Different Ways of Multiplying n Matrices?

$$T(n) = \sum_{k=1}^{n-1} T(k)T(n-k)$$

This is the recurrence for the Catalan number n = $\Theta(4^n/n^{3/2})$

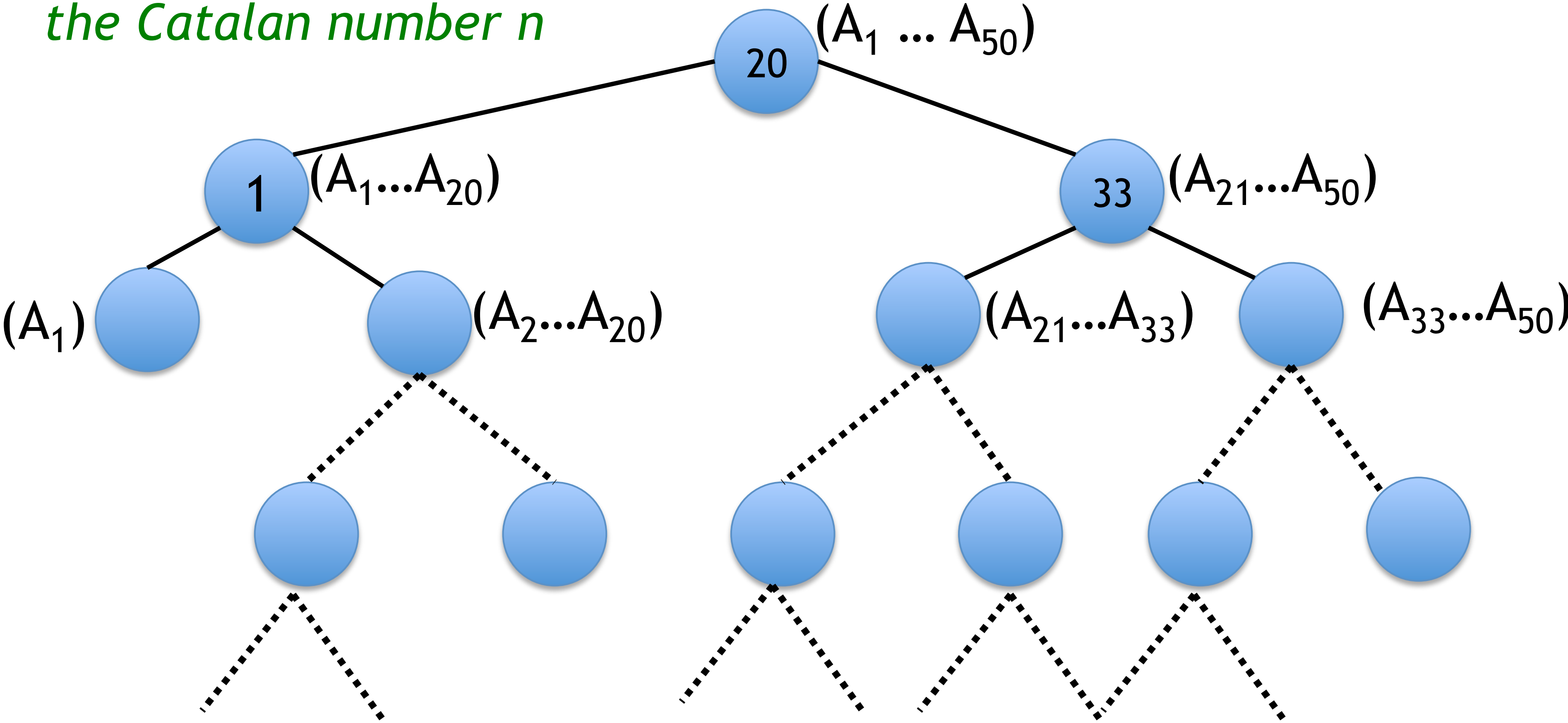
Exercise: Using the substitution method, show it's $\Omega(2^n)$

So there are actually exponential # orderings of n matrices.

Binary Tree Interpretation of Orders

- Any ordering can be thought as a binary tree of numbers 1,…,n (and vice versa)

Fact: # of dif. binary trees of 1…n is the Catalan number n



Recall Recipe of a DP Algorithm

1: Identify small # of subproblems

2: quickly + correctly solve "larger" subproblems given solutions to smaller ones

3: After solving all subproblems, can quickly compute final solution

Question: What subproblems should we be thinking of?

Subproblems

Let OPT be the opt. ordering for multiplying A_1 x A_2 x … x $A_{n.}$

Let $A_{i…j}$ be the result of multiplying A_i x A_{i+1} x … A_j

Claim that doesn't require a proof:

OPT multiplies (or splits):

Case 1: A_1 x $A_{2…n}$

subproblems

Case 2: $A_{1…2,}$ x $A_{3…n}$

…

Case k: $\boxed{A_{1…k}}$ x $\boxed{A_{k+1…n}}$

…

Case n-1: $A_{1…n-1}$ x A_n

Case k

Suppose we are in Case k

I.e. OPT's final multiplication is $A_{1\ldots k} \times A_{k+1\ldots n}$

Claim: Then OPT 's ordering for $A_{1\ldots k}$, call OPT_{1k}, is the min cost ordering for multiplying $A_1 \times A_2 \times \ldots \times A_k$

Proof: Break OPT into OPT_{1k} and $OPT_{k+1,n}$

Suppose $\exists$ O_{1k}^* for multiplying $A_1 \times A_2 \times \ldots \times A_k$ s.t

$$\text{cost}(O_{1k}^*) < \text{cost}(OPT_{1k})$$

$$\text{then cost}(O_{1k}^* \cup OPT_{k+1,n}) + d_0 d_k d_n < \text{cost}(OPT)$$

contradicting OPT's optimality.

Q.E.D.

OPT In terms of Solutions to Subproblems

Let $OPT_{i,j}$ be the optimal ordering to $A_i \times A_{i+1} \times \ldots \times A_j$

$OPT_{i,i}$ is simply A_i

$$OPT_{1,n} = \min \begin{cases} OPT_{1,1} + OPT_{2,n} + d_0 d_1 d_n \\ OPT_{1,2} + OPT_{3,n} + d_0 d_2 d_n \\ \ldots \\ OPT_{1,k} + OPT_{k+1,n} + d_0 d_k d_n \\ \ldots \\ OPT_{1,n-1} + OPT_{n,n} + d_0 d_{n-1} d_n \end{cases}$$

More Generally

$$OPT_{i,j} = \min \begin{cases} OPT_{i,i} + OPT_{i+1,j} + d_{i-1}d_id_j \\ OPT_{i,i+1} + OPT_{i+2,j} + d_{i-1}d_{i+1}d_j \\ \dots \\ OPT_{i,k} + OPT_{k+1,j} + d_{i-1}d_kd_j \\ \dots \\ OPT_{i,j-1} + OPT_{j,j} + d_{i-1}d_{j-1}d_j \end{cases}$$

A Possible Recursive Algorithm

Recurse on all cases and return the best solution.

```
Recursive-Matrix-Ordering: (dim of Ai,…, Aj)
```

$O_1 = RMO(A_i) + RMO(A_{i+1} \dots A_j) + d_{i-1} d_i d_j$

$O_2 = RMO(A_i A_{i+1}) + RMO(A_{i+2} \dots A_j) + d_{i-1} d_{i+1} d_j$

…

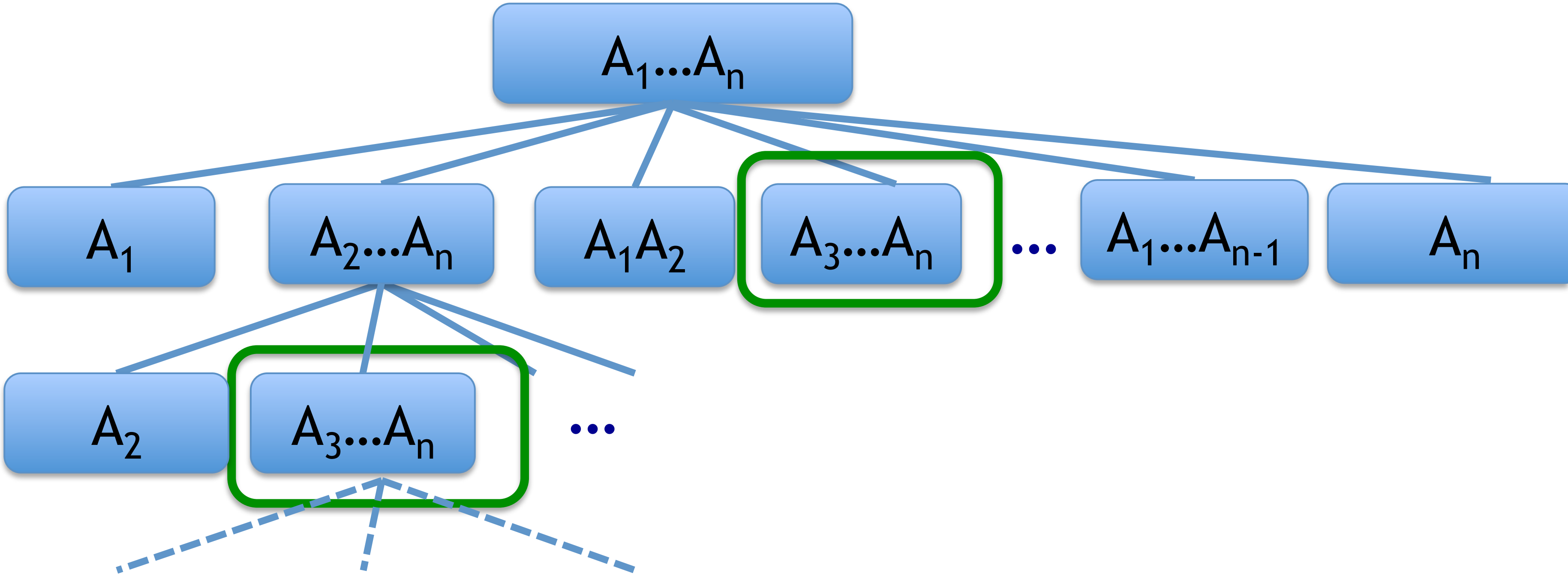
$O_{j-i-1} = RMO(A_i \dots A_{j-1}) + RMO(A_j) + d_{i-1} d_{j-1} d_j$

return min $O_1, O_2, \dots, O_{j-i-1}$

Good news: The algorithm is correct!

Problem: This is brute-force search!

Q: How many distinct recursive calls?



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

for each i, j s.t i < j, there is one recursive call

Dynamic Programming Solution

Let S[][] be an n by n matrix,

S[i][j] is min cost ordering for multiplying A_i, …, A_j

```
procedure DP-Matrix-Ordering(d_0,d_1,…,d_n):
  Base Cases: S[i][i]=0;S[i][i+1]=d_{i-1}d_i d_{i+1}
  for i = 1 … n
    for j = 1 … n
      min_{i,j} = +∞
      for k = i, … j-1
        min_{i,j}=min(min_{i,j},S[i][k]+S[k+1][j]+d_{i-1}d_k d_j)
      S[i][j] = min_{i,j}
  return S[1][n]
```

Looks wrong!

Ex: i=1, j = n, k =2

We access S[2][n] => not yet computed

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

$$S[2, 6] = \begin{cases} S[2, 2] + S[3,6] + d_1 d_2 d_6 \\ \\ \\ \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

$$S[2,\ 6] = \begin{cases} S[2,\ 2] + S[3,6] + d_1 d_2 d_6 \\ S[2,\ 3] + S[4,6] + d_1 d_3 d_6 \\ \\ \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

$$S[2, 6] = \begin{cases} S[2, 2] + S[3,6] + d_1 d_2 d_6 \\ S[2, 3] + S[4,6] + d_1 d_3 d_6 \\ S[2, 4] + S[5,6] + d_1 d_4 d_6 \\ \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

$$S[2,6] = \begin{cases} S[2,2] + S[3,6] + d_1 d_2 d_6 \\ S[2,3] + S[4,6] + d_1 d_3 d_6 \\ S[2,4] + S[5,6] + d_1 d_4 d_6 \\ S[2,5] + S[6,6] + d_1 d_5 d_6 \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

$$S[2,\ 6] = \begin{cases} S[2,\ 2] + S[3,6] + d_1 d_2 d_6 \\ S[2,\ 3] + S[4,6] + d_1 d_3 d_6 \\ S[2,\ 4] + S[5,6] + d_1 d_4 d_6 \\ S[2,\ 5] + S[6,6] + d_1 d_5 d_6 \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

$$S[3, 7] = \begin{cases} S[3, 3] + S[4,7] + d_2 d_3 d_7 \\ \\ \\ \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

$$S[3, 7] = \begin{cases} S[3, 3] + S[4,7] + d_2 d_3 d_7 \\ S[3, 4] + S[5,7] + d_2 d_4 d_7 \\ \\ \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

$$S[3,\ 7]= \begin{cases} S[3,\ 3] + S[4,7] + d_2d_3d_7 \\ S[3,\ 4] + S[5,7] + d_2d_4d_7 \\ S[3,\ 5] + S[6,7] + d_2d_5d_7 \\ \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

$$S[3, 7] = \begin{cases} S[3, 3] + S[4,7] + d_2 d_3 d_7 \\ S[3, 4] + S[5,7] + d_2 d_4 d_7 \\ S[3, 5] + S[6,7] + d_2 d_5 d_7 \\ S[3, 6] + S[7,7] + d_2 d_6 d_7 \end{cases}$$

Which Cells Do We Need to Access?

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

$$S[3,\ 7] = \begin{cases} S[3,\ 3] + S[4,7] + d_2 d_3 d_7 \\ S[3,\ 4] + S[5,7] + d_2 d_4 d_7 \\ S[3,\ 5] + S[6,7] + d_2 d_5 d_7 \\ S[3,\ 6] + S[7,7] + d_2 d_6 d_7 \end{cases}$$

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200	220			
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200	220			
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200	220			
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200	220			
3			0	40	100		
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200	220			
3			0	40	100		
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200	220			
3			0	40	100		
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200	220			
3			0	40	100		
4				0	80	150	
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200	220			
3			0	40	100		
4				0	80	150	
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200	220			
3			0	40	100		
4				0	80	150	
5					0	120	
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200	220			
3			0	40	100		
4				0	80	150	
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200	220			
3			0	40	100		
4				0	80	150	
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200	220			
3			0	40	100		
4				0	80	150	
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400				
2		0	200	220			
3			0	40	100		
4				0	80	150	
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600			
2		0	200	220			
3			0	40	100		
4				0	80	150	
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600			
2		0	200	220	310		
3			0	40	100		
4				0	80	150	
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600			
2		0	200	220	310		
3			0	40	100	205	
4				0	80	150	
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600			
2		0	200	220	310		
3			0	40	100	205	
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600	540		
2		0	200	220	310		
3			0	40	100	205	
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600	540		
2		0	200	220	310	380	
3			0	40	100	205	
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600	540		
2		0	200	220	310	380	
3			0	40	100	205	540
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600	540	660	
2		0	200	220	310	380	
3			0	40	100	205	540
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600	540	660	
2		0	200	220	310	380	700
3			0	40	100	205	540
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600	540	660	
2		0	200	220	310	380	700
3			0	40	100	205	540
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600	540	660	
2		0	200	220	310	380	700
3			0	40	100	205	540
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600	540	660	
2		0	200	220	310	380	700
3			0	40	100	205	540
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600	540	660	
2		0	200	220	310	380	700
3			0	40	100	205	540
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600	540	660	
2		0	200	220	310	380	700
3			0	40	100	205	540
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600	540	660	
2		0	200	220	310	380	700
3			0	40	100	205	540
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (1)

	1	2	3	4	5	6	7
1	0	500	400	600	540	660	740
2		0	200	220	310	380	700
3			0	40	100	205	540
4				0	80	150	500
5					0	120	300
6						0	400
7							0

Note, this is the final solution

(not the ordering but the cost)

Dynamic Programming Solution

Let S[][] be an n by n matrix,

S[i][j] is min cost ordering for multiplying A_i, …, A_j

```
procedure DP-Matrix-Ordering(d_0,d_1,…,d_n):
  Base Cases: S[i][i]=0;S[i][i+1]=d_{i-1}d_i d_{i+1}
   for len = 3 … n
     for r = 1 … n-len
        c = r + len - 1;
        min_{c,r} = +∞
        for k = 1,…,len
           j = r+k-1;
           min_{c,r}=min(min_{c,r},S[r][j]+S[j+1][c]+d_{r-1}d_k d_c)
        S[c][r] = min_{c,r}
        c++;
  return S[1][n]
```

Runtime: $O(n^3)$
(check as an exercise)

Note: Text book has another fix.

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	
6						0	400
7							0

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80		
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80	150	
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80	150	
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80	150	
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80	150	
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40			
4				0	80	150	500
5					0	120	300
6						0	400
7							0

The Way We Should Traverse (2)

	1	2	3	4	5	6	7
1	0	500					
2		0	200				
3			0	40	100		
4				0	80	150	500
5					0	120	300
6						0	400
7							0

so and so forth…

Exercise: Write the code for the 2nd traversal.

Reconstructing The Optimal Ordering

Store with each cell, the split k that minimized the cost

Reconstructing The Optimal Solution

	1	2	3	4	5	6	7
1	1	1	2	2	2	3	5
2		2	2	3	4	4	3
3			3	3	4	3	5
4				4	4	4	4
5					5	5	5
6						6	6
7							7

Reconstructing The Optimal Solution

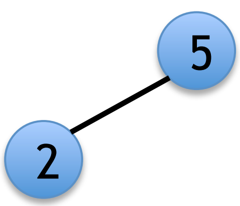
	1	2	3	4	5	6	7
1	1	1	2	2	2	3	5
2		2	2	3	4	4	3
3			3	3	4	3	5
4				4	4	4	4
5					5	5	5
6						6	6
7							7

5

$(A_1A_2A_3A_4A_5) \times (A_6A_7)$

Reconstructing The Optimal Solution

	1	2	3	4	5	6	7
1	1	1	2	2	2	3	5
2		2	2	3	4	4	3
3			3	3	4	3	5
4				4	4	4	4
5					5	5	5
6						6	6
7							7

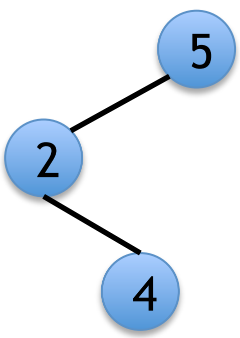


$$\boxed{(A_1A_2A_3A_4A_5)} \times (A_6A_7)$$

$$(A_1A_2) \times \boxed{(A_3A_4A_5)}$$

Reconstructing The Optimal Solution

	1	2	3	4	5	6	7
1	1	1	2	2	2	3	5
2		2	2	3	4	4	3
3			3	3	4	3	5
4				4	4	4	4
5					5	5	5
6						6	6
7							7



$(A_1A_2A_3A_4A_5) \times (A_6A_7)$

$(A_1A_2) \times \boxed{(A_3A_4A_5)}$

$(A_3A_4) \times A_5$

Final Order: $((A_1 \times A_2) \times ((A_3 \times A_4) \times A_5)) \times (A_6 \times A_7)$

Outline For Today

1. Matrix Multiplication Order
2. 0/1 Integer Weight Knapsack

0-1 Integer Weight Knapsack

- Input:
  - n items
  - values for items $v_1, \ldots, v_n \geq 0$ (not necessarily integer)
  - sizes for items $w_1, \ldots, w_n \geq 0$ (integers)
  - knapsack capacity $W \geq 0$ (integer)
- Output: subset $S \subseteq \{1, 2, \ldots, n\}$ items s.t.

$$\max \sum_{i \in S} v_i$$

$$s.t. \sum_{i \in S} w_i \leq W$$

Knapsack Example

$v_1 = 2.2$
$w_1 = 2$

$v_2 = 4.0$
$w_2 = 3$

$v_3 = 2.0$
$w_3 = 3$

$v_4 = 3.0$
$w_4 = 5$

$W=9$

Knapsack Example

$v_1 = 2.2$
$w_1 = 2$

$v_2 = 4.0$
$w_2 = 3$

$v_3 = 2.0$
$w_3 = 3$

$v_4 = 3.0$
$w_4 = 5$

$v_1 = 2.2$
$w_1 = 2$

$v_3 = 2.0$
$w_3 = 3$

$v_2 = 4.0$
$w_2 = 3$

W=9

OPT = 4+2+2.2=8.2

Greedy Algorithm 1 (Most valuable)

$v_1 = 2.2$
$w_1 = 2$

$v_2 = 4.0$
$w_2 = 3$

$v_3 = 2.0$
$w_3 = 3$

$v_4 = 3.0$
$w_4 = 5$

$v_4 = 3.0$
$w_4 = 5$

$v_2 = 4.0$
$w_2 = 3$

4+3=7

W=9

Clearly, not optimal

Greedy Algorithm 2 (Lightest first)

$v_1 = 2.2$
$w_1 = 2$

$v_2 = 4.0$
$w_2 = 3$

$v_3 = 2.0$
$w_3 = 3$

$v_4 = 3$
$w_4 = 5$

$v_3 = 2.0$
$w_3 = 3$

$v_2 = 4.0$
$w_2 = 3$

$v_1 = 2.2$
$w_1 = 2$

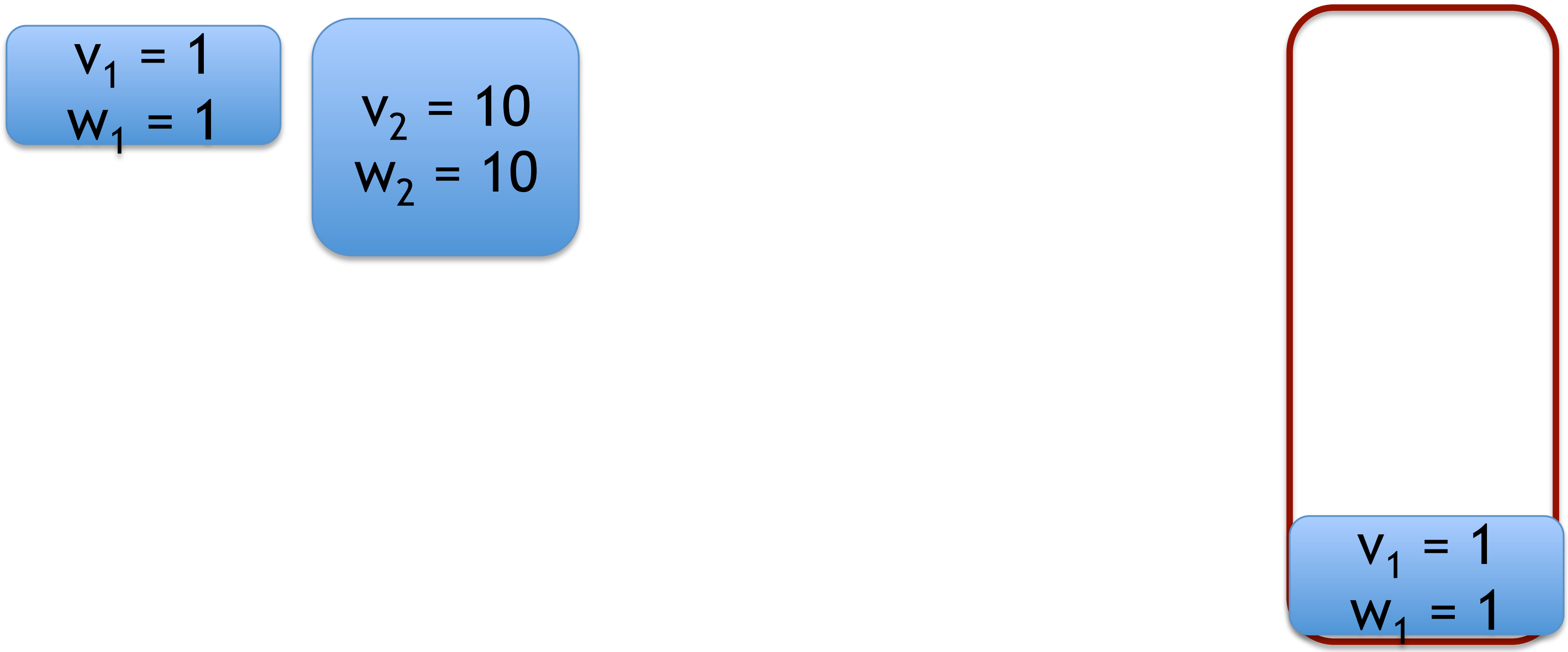
$W=9$

2.2 + 4+ 2=8.2

Looks optimal for this example

But what can go wrong?

Counter Example: Greedy Lightest First



Optimal is 10. (put 2nd item)

W=10

Greedy-Lightest: 1

Clearly not optimal.

Recall Scheduling Problem

- Input: Each job$_i$ has length l_i AND weight w_i

Job 1 l_1, w_1

Job 2 l_2, w_2

…

Job n l_n,w_n

- Output: A schedule of the jobs on a processor

s.t: $$\sum_{i=1}^{n} w_i C_i$$ ← weighted completion time of job i

is minimum over all possible n! schedules.

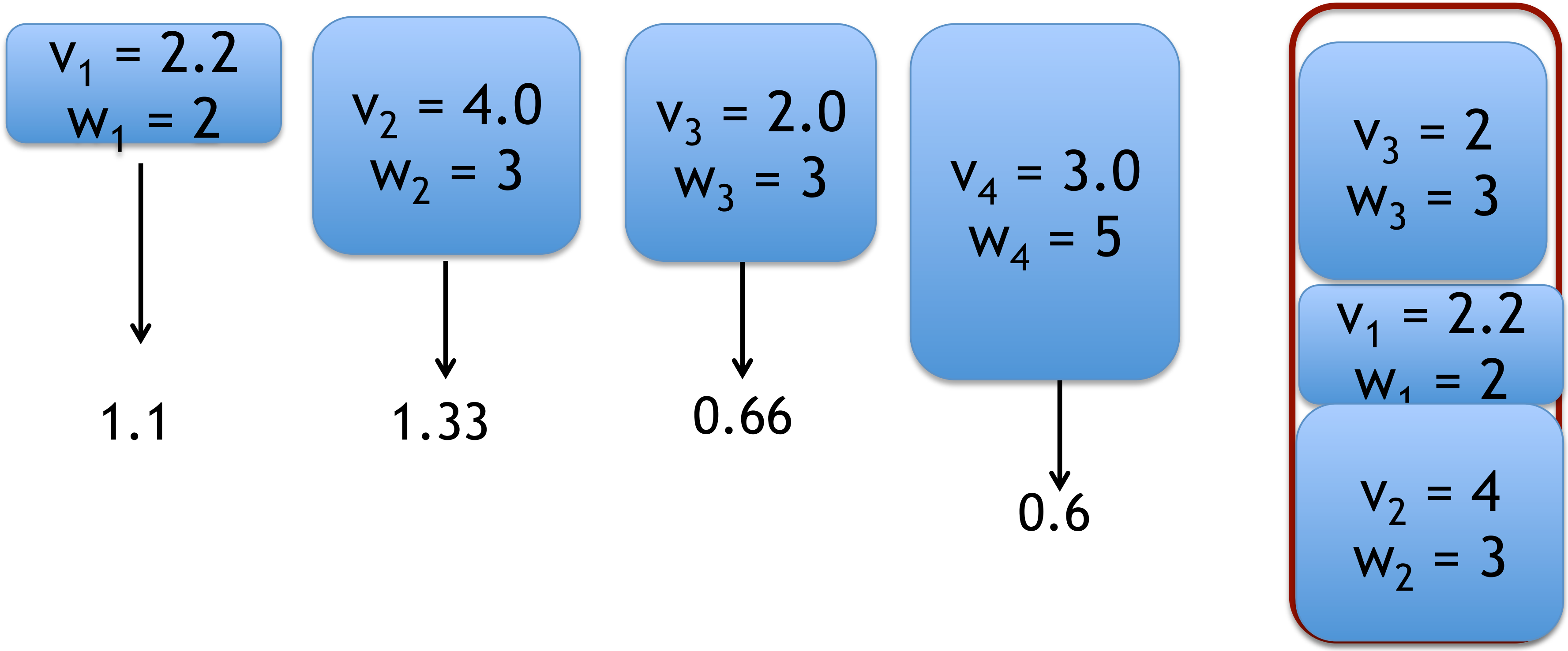
Greedy 3: Largest (Value/Weight)

- Similar to Greedy Weighted Scheduling
- If weights are the same, put higher value items first
- If values are the same, put lighter items first
- Greedy Algorithm:

1. Combine v_i and w_i into a single score v_i / $w_{i:}$
2. Sort items in increasing combined score

   Assume w.l.o.g.: $v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n$
3. Pack until can't pack anymore

Greedy 3: Largest (Value/Weight)



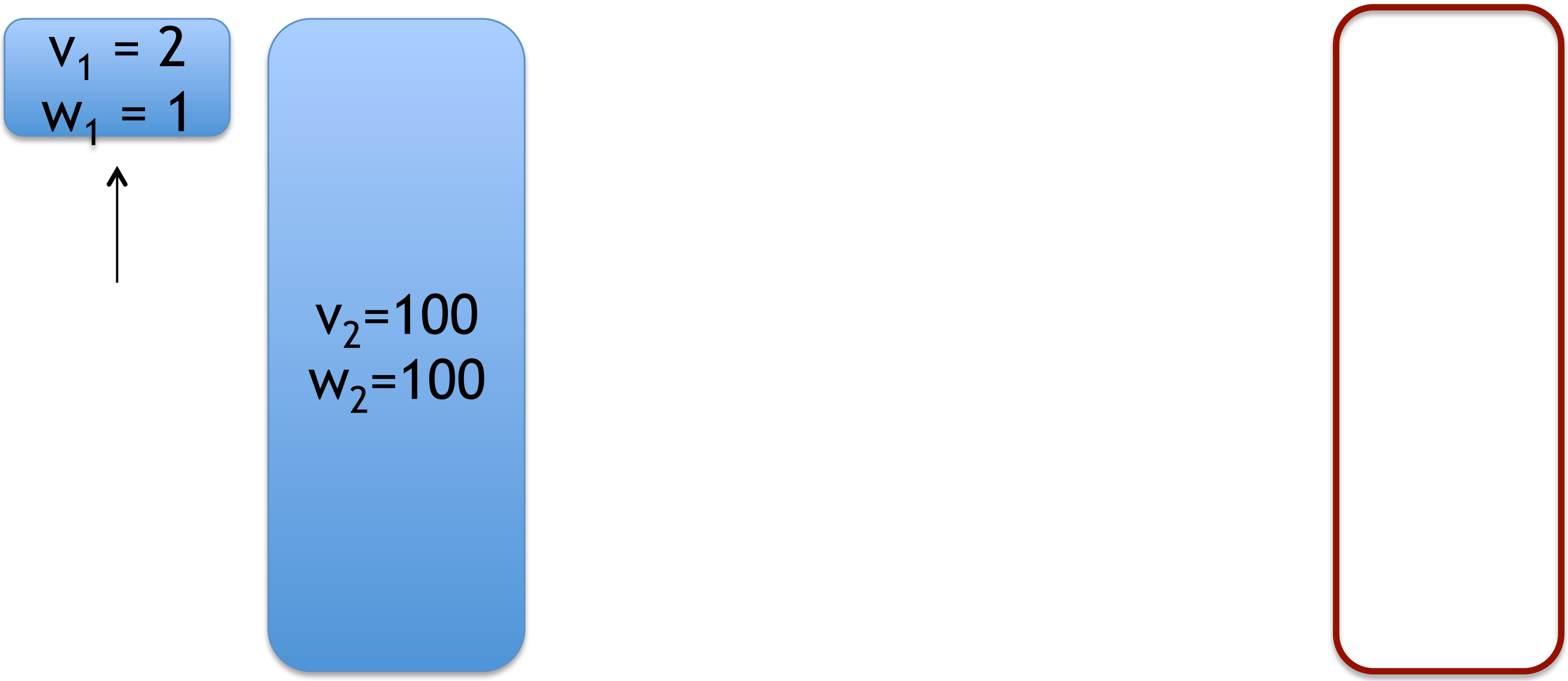
2.2 + 4+ 2=8.2

W=9

Looks optimal for this example

But what can go wrong?

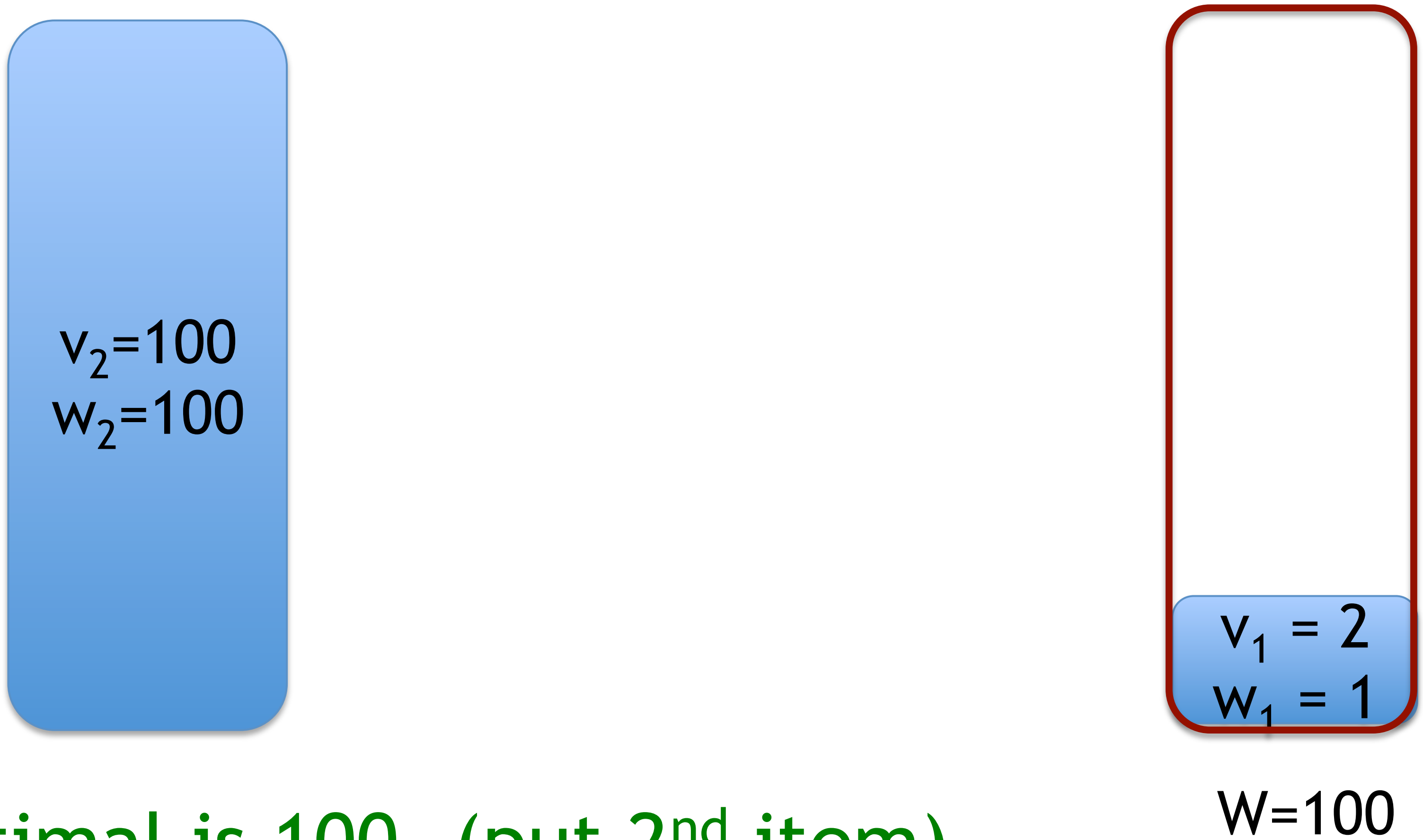
Counter Example: Greedy Largest V/W



W=100

Optimal is 100. (put 2nd item)

Counter Example: Greedy Largest V/W



Optimal is 100. (put 2nd item)

Greedy Largest V/W has a value of 2.

Can be arbitrarily bad.

Not Surprising That Greedy Algs Don't Work!

0-1 Knapsack is Intractable!

At a high-level, you can think of 0-1 Knapsack as the *hardest* (no exaggeration) computational problem to solve ***exactly*** in the "real world". More specifically: It is *as hard as every other computational problem* among problems that generally appear in practice.
(Other "harder" problems can be constructed but they don't appear often in the real world.)
Will make formal by NP-Completeness in 5 weeks.

Recap: Recipe of a DP Algorithms

1. Identify small # of subproblems
2. Quickly + correctly solve "larger" subproblems given solutions to smaller ones
3. After solving all subproblems, can quickly compute final solution

0-1 Knapsack DP Algorithm 1

- Order the n items in arbitrary order: {1, 2, …, n}.
- Consider the optimal solution S*

A Claim that Doesn't Require A Proof:

$$(1)\ n \notin S^* \text{ or } (2)\ n \in S^*$$

Case 1: $n \notin S^*$

Q: What can we assert about S^* for items {1, …, n-1}?

A: S^* is opt. for items {1, …, n-1} and capacity W.

Proof: Assume $\exists$ better S^{**} w/ cap. W for {1, …, n-1}

$\Rightarrow$ S^{**} is feasible for {1, …, n} and better than S^*

$\Rightarrow$ Which would contradict S^*'s optimality

Q.E.D.

Case 2: $n \in S^*$

Q: What can we assert about S^*-{n} for items {1, …, n-1}?

A: S^*-{n} is opt. for items {1, …, n-1} and cap. $W-w_n$.

Pf: Assume $\exists$ better S^{**} w/ cap $\leq W-w_n$ for {1, …, n-1}

$\Rightarrow S^{**} \cup \{n\}$ has capacity $\leq W$

$\Rightarrow S^{**} \cup \{n\}$ is feasible for {1, …, n} and better than S^*

$\Rightarrow$ Which would contradict S^*'s optimality

Q.E.D.

What Are The Subproblems?

$K_{(i,\ c)}$: opt. knapsack for the first i items and cap c.

Q: How many subproblems are there?

A: n*W

$$K_{(i,\ c)} = \max \begin{cases} K_{(i-1,\ c)} \\ K_{(i-1,\ c\ -\ w\_i)} + v_i \end{cases}$$

0-1 Knapsack DP Algorithm 1 Pseudocode

$$K_{(i,\,c)} = \max \begin{cases} K_{(i-1,\,c)} \\ K_{(i-1,\,c\,-\,w\_i)} \end{cases}$$

```
procedure DP-Knapsack-1(n, W):
 Base Cases: A[0][i] = 0
  for i = 1,2,…,n:
     for c = 1, …, W:
     A[i][c] = max{A[i-1][c], A[i-1][c-wᵢ]+vᵢ}
  return A[n][W]
```

Run-time

Runtime: $O(nW)$

Brute-Force Search: $\Omega(2^n)$

Observation: This is polynomial in n and W.

Q: Is Knapsack then tractable?!

A: No! B/c we're still exponential in input size.

Input

Input size: # bits (key strokes) to represent the problem

n weights, values (n * (log of max weight and value))

capacity => log(W) bits.

Note: W is exponential in log(W)

w_1	v_1	w_2	v_2	…	…	w_n	v_n	W

Pseudo-polynomial Time Algorithm

An algorithm that's polynomial in the numeric values of the inputs but not the # bits to represent it.

Ex: O(nW) is pseudo-polynomial

Interpretation: If we fix W to a (constant) integer value

=> Knapsack is tractable

Called "Fixed-Parameter Tractable" Problem