CSCI203

Algorithms and Data Structures

Dynamic Programming (II)

Lecturer: Dr. Xueqiao Liu

Room 3.117

Email: xueqiao@uow.edu.au

Dynamic Programming

- Previously we saw several "definitions" of Dynamic Programming:
  - Clever Brute Force;
  - Recursion + Memoization;
  - Tabular and bottom-up implementation
- Remember: dynamic programming is not an algorithm; it is a technique for constructing algorithms.

- This lecture we will examine some more problems that can be solved using dynamic programming techniques.

Dynamic Programming (DP)

- Typically applied to optimization problems
- It is used when the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*).
- Algorithm finds solutions to subproblems and
- Stores them in memory, i.e. a table, for later use.
- More efficient than "brute-force methods", which solve the same subproblems over and over again.

Dynamic Programming (DP)

- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

#1: Knapsack problem (Review)

- Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W. So we must consider weights of items as well as their value.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5

Knapsack problem

- There are two versions of the problem:
  1. "0-1 knapsack problem" and
  2. "Fractional knapsack problem"

1. Items are indivisible; you either take an item or not. Solved with dynamic programming

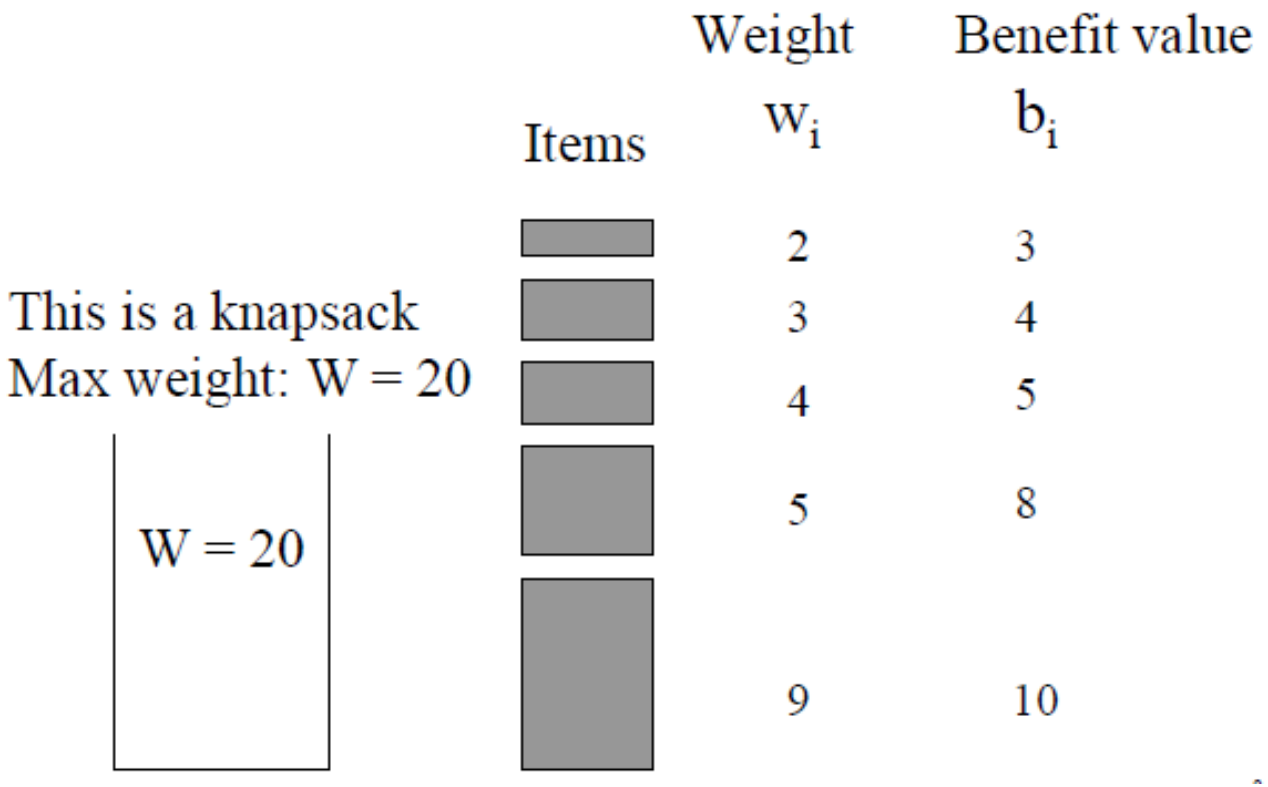
2. Items are divisible: you can take any fraction of an item. Solved with a greedy algorithm.

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



0-1 Knapsack problem

- Problem, in other words, is to find

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

- The problem is called a "0-1" problem, because each item must be entirely accepted or rejected.

- In the "*Fractional Knapsack Problem*," we can take fractions of items.

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 maximum value and with total weight less or equal to W

- Complexity will be $O(2^n)$

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 labelled $1,2,\cdots,n$, then a subproblem may be to find an optimal solution for
  - $S_k = \{items\ labeled\ 1,2,..k\}$

Defining a Subproblem

- If items are labelled $1, \cdots, n$, then a subproblem may be to find an optimal solution for

$$S_k = \{items\ labelled\ 1, 2, .. k\}$$

- This is a reasonable subproblem definition.

- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we can't do that.

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;

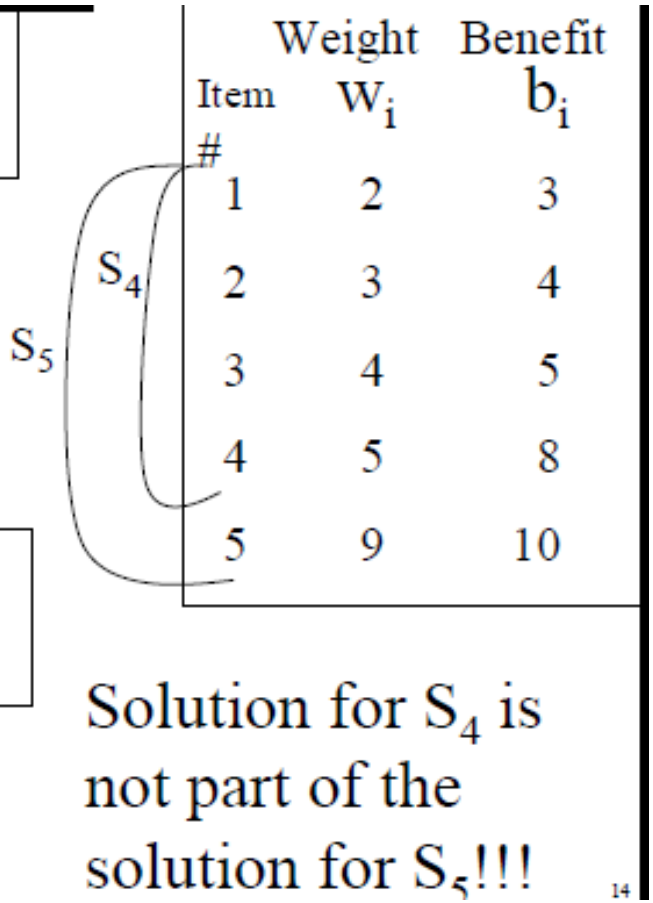
Maximum benefit: 20

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_5=9$ $b_5=10$

For S_5:

Total weight: 20

Maximum benefit: 26



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

14

Defining a Subproblem

- 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 $B[k, w]$

Defining a Subproblem

- $B[k, w]$ be the value of an optimal solution to the instance
  - The value of the most valuable subset of the first k items that fits into the knapsack of capacity w

- We can divide all the subsets of the first k items that the knapsack of capacity w into two categories
  - Those that do not include the $k'th$ item, and
  - Those that do

Defining a Subproblem

- Among the subsets that do not include the $k'th$ item, the value of an optimal subset is, by definition, $B[k-1, w]$

- Among the subsets that do include the $k'th$ item (hence, $w - w_k \geq 0$ or $w \geq w_k$), an optimal subset is made up of this item and an optimal subset of the first $k-1$ items that fits into the knapsack of capacity $w - w_k$, the value of such an optimal subset is $B[k-1, w-w_k] + b_k$

- Thus, the value of an optimal solution among all feasible subsets of the first k items is the maximum of these two values

Recursive Forumula

- Recursive formula for subproblems:

$$B[k,w] = \begin{cases} B[k-1,w] & if\ (w - w_k) < 0 \\ max\{B[k-1,w], B[k-1,w-w_k] + b_k & if\ (w - w_k) \geq 0 \end{cases}$$

- It means, that the best subset of S_k that has total weight w is:
  - the best subset of S_{k-1} that has total weight w, or
  - the best subset of S_{k-1} that has total weight $w - w_k$ plus the item k

Recursive Formula

	0	$w - w_k$	w	W
0	0	0	0	0
$k-1$		$B[k-1, w-w_k]$	$B[k-1, w]$	
w_k, b_k, k			$B[k, w]$	
n				goal

$B[0, w] = 0 \; for \; w \geq 0$, and $B[k, 0] = 0 \; for \; k \geq 0$

Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & if\ (w - w_k) < 0 \\ max\{B[k-1,w], B[k-1,w-w_k] + b_k & if\ (w - w_k) \geq 0 \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.
- First case: $(w - w_k) < 0$ Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- Second case: $(w - w_k) \geq 0$. Then the item k can be in the solution, and we choose the case with greater value.

0-1 Knapsack Algorithm

```
Knapsack (W, n, b[1…n])
  for w=0 to W
    B[0,w]=0
  for i=1 to n
    B[i,0]=0
  for i=1 to n
    for w=1 to W
      if ((w − w_i) ≥ 0)    // 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 − w_i) < 0
    rof
  rof
```

Complexity

```
for w = 0 to W                    O(W)
    B[0,w] = 0
for i = 1 to n
    B[i,0] = 0
for i = 1 to n                    Repeat n times
    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
  - $O(2^n)$

An 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)

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

for w = 0 to W

B[0,w] = 0

Example (3)

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

```
for i = 1 to n
        B[i,0] = 0
```

Example (4)

Items:

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

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

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

i=1

b_i=3

w_i=2

w=2

w-w_i =0

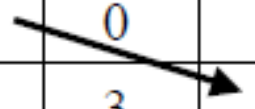
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (6)

Items:

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

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

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

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

i=1

b_i=3

w_i=2

w=5

w-w_i =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 (9)

Items:

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

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

i=2

b_i=4

w_i=3

w=1

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 (10)

Items:

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

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

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

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)

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

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

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)

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

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)

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

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

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 (14)

Items:

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

$i=3$

$b_i=5$

$w_i=4$

$w= 1..3$

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

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

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 (16)

Items:

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

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

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 (17)

Items:

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

$i=4$

$b_i=6$

$w_i=5$

$w= 1..4$

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

Example (18)

Items:

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

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

$i=4$

$b_i=6$

$w_i=5$

$w= 5$

$w- w_i=0$

```
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
```

Note

- This algorithm only finds the max possible value that can be carried in the knapsack
  - i.e., the value in $B[n, W]$

- To know the items that make this maximum value, an addition to this algorithm is necessary.

Finding Knapsack Items

- All of the information we need is in the table.
- $B[n, W]$ is the maximal value of items that can be placed in the Knapsack.

```
i=n; k=W
while i,k >0
    if B[i,k] ≠ B[i-1,k] then
        mark the i'th item as in the knapsack
        i = i-1, k = k-w_i
    else
        i = i-1 // Assume the i'th item is not in the knapsack
                // Could it be in the optimally packed knapsack?
```

Finding the Items

Items:

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

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

i=4
k= 5
b_i=6
w_i=5
$B[i,k] = 7$
$B[i-1,k] = 7$

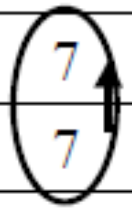
i=n, k=W
while i,k > 0
    if $B[i,k] \neq B[i-1,k]$ then
        mark the i^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Finding the Items (2)

Items:

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

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



i=4
k= 5
b_i=6
w_i=5
$B[i,k] = 7$
$B[i-1,k] = 7$

i=n, k=W
while i,k > 0
    if $B[i,k] \neq B[i-1,k]$ then
        mark the i^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Finding the Items (3)

Items:

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

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

i=3
k= 5
b_i=6
w_i=4
$B[i,k] = 7$
$B[i-1,k] = 7$

i=n, k=W
while i,k > 0
    if $B[i,k] \neq B[i-1,k]$ then
        mark the i^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Finding the Items (4)

Items:

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

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

i=2
k= 5
b_i=4
w_i=3
$B[i,k] = 7$
$B[i-1,k]$ =3
k – w_i=2

i=n, k=W
while i,k > 0
    if $B[i,k] \neq B[i-1,k]$ then
        mark the i^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Finding the Items (5)

Items:

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

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

i=1
k= 2
b_i=3
w_i=2
$B[i,k] = 3$
$B[i-1,k] = 0$
$k - w_i = 0$

i=n, k=W
while i,k > 0
    if $B[i,k] \neq B[i-1,k]$ then
        mark the i^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Finding the Items (6)

Items:

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

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

i=0
k= 0

The optimal knapsack should contain {1, 2}

i=n, k=W
while i,k > 0
    if $B[i,k] \neq B[i-1,k]$ then
        mark the n^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Finding the Items (7)

Items:

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

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

The optimal knapsack should contain {1, 2}

i=n, k=W
while i,k > 0
    if $B[i,k] \neq B[i-1,k]$ then
        mark the n^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Note

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary (memoization)
- Running time of dynamic programming algorithm vs. naïve algorithm:
  - 0-1 Knapsack problem: $O(W * n)$ vs. $O(2n)$

Related References

- Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
  - Chapters 8.2

- Most slides are based on the slides prepared by Dr Steve Goddard@cse.unl.edu

#2:Matrix Multiplication…

$$c_{11}=a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31}+a_{14}b_{41}$$

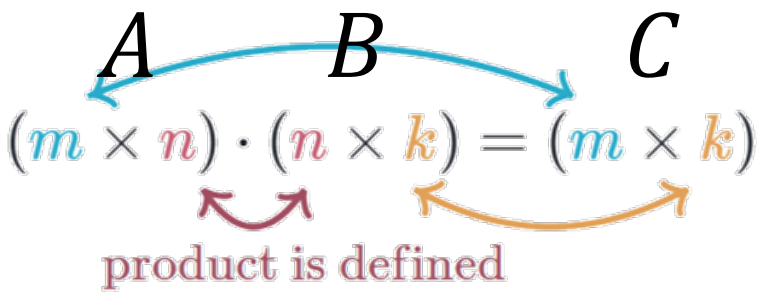
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

2 x 4 4 x 3 2 x 3

$$c_{22}=a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32}+a_{24}b_{42}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

$$\boldsymbol{A \times B = C}$$



Matrix Multiplication…

- An Example

- Let A and B be the following Matrices:
- $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$, $B = \begin{bmatrix} -1 & 2 \\ 2 & -1 \end{bmatrix}$
- Then C is calculated as follows:
- $c(1,1) = 1 \times -1 + 2 \times 2 = 3$
- $c(1,2) = 1 \times 2 + 2 \times -1 = 0$
- $c(2,1) = 3 \times -1 + 4 \times 2 = 5$
- $c(2,2) = 3 \times 2 + 4 \times -1 = 2$
- $c(3,1) = 5 \times -1 + 6 \times 2 = 7$
- $c(3,2) = 5 \times 2 + 6 \times -1 = 4$
- So:
- $C = \begin{bmatrix} 3 & 0 \\ 5 & 2 \\ 7 & 4 \end{bmatrix}$

Matrix Multiplication…

- If A has m rows and n columns and B has n rows and k columns then C will have m rows and k columns.
- Matrix multiplication is not commutative: $AB \neq BA$ (BA may not even exist).

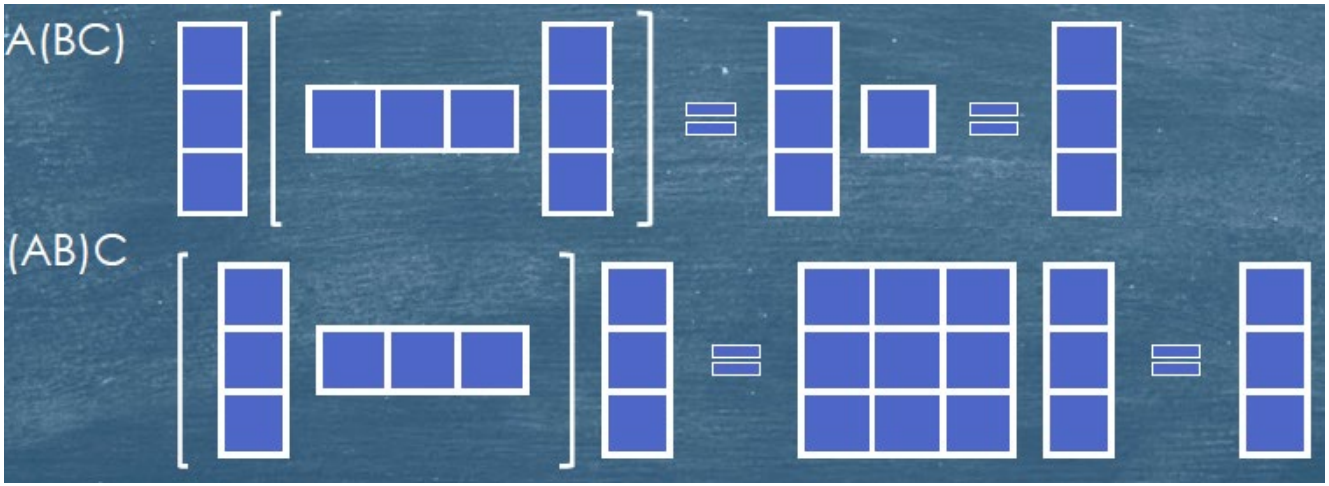
- The cost of a matrix multiplication depends on the sizes of A and B
- $A_{m\times n} \times B_{n\times k}$ will take $m \times n \times k$ multiplications and $m \times (n-1) \times k$ additions.
- The order in which multiple matrices are multiplied will effect the total cost.

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

Matrix-chain Multiplication

- Matrix multiplication is associative $A(BC)=(AB)C$.

- Let A, B and C be three matrices with sizes 3×1, 1×3 and 3×1 respectively:
- ABC can be computed as $A(BC)$ or $(AB)C$.



6 multiplications

18 multiplications

Matrix-chain multiplication

- Multiplication of four matrices
  - $< A_1, A_2, A_3, A_4 >$
- We can define five distinct ways to perform the calculation using parentheses:

$$\big(A_1(A_2(A_3A_4))\big), \big(A_1((A_2A_3)A_4)\big), ((A_1A_2)(A_3A_4)), \big((A_1(A_2A_3))A_4\big), \big(((A_1A_2)A_3)A_4\big)$$

- How we parenthesize a chain of matrices can have a dramatic impact on the cost of computing the product.

Generalization

- If we need to evaluate the product of n matrices:
  - $A_1 \times A_2 \times A_3 \times \cdots \times A_n$ or $A_1 A_2 A_3 \cdots A_n$
- We can perform this in $O(4^n)$ different ways, each has a potentially different cost.
- What is the minimum cost for the overall computation.
  - What is the optimum sequence of matrix multiplications to perform?
- Once again, we can solve this with dynamic programming.

DP: Parenthesization

- We can restate the problem as follows:
- Given a sequence of n matrices; find the optimal locations for $n-1$ pairs of balanced parentheses, such that each pair contains exactly two matrices or parenthesized sets of matrices.
- E.g. given matrices $ABCD$, possible parenthesizations are:
  - $A(B(CD)), A((BC)D), (AB)(CD), (A(BC))D, ((AB)C)D$
- Let us approach this problem in the same way as we have used with the other problems.

Notations

- Given a chain $< A_1, A_2, \cdots, A_n >$ of matrices, where for $i = 1,2, \cdots, n$ matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications

- $A_{i \cdots j}$ denotes $A_i A_{i+1} \cdots A_j$, $i \leq j$

- Note: *we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost.*

Step 1. Structure an optimal parenthesization

- Suppose to optimally parenthesize $A_i \cdots A_j$
- Let's say an optimal split is between A_k and A_{k+1}
- The problem becomes two sub-problems
  - $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$
- We must ensure that we search for the correct place to split the product
  - We have considered all possible places so that we are sure of having examined the optimal one

Step 2. A recursive solution

- Let $m[i, j]$ be the minimum number of scalar multiplications needed for $A_{i \ldots j}$
  - For the full problem $A_{1 \ldots n}$, it would be $m[1, n]$

- Lets' examine $m[i, j]$

Step 2. A recursive solution…

- If $i = j$, $A_{i \cdots i} = A_i$, no scalar multiplications, $m[i, i] = 0$, for $i = 1,2, \cdots, n$
- If $i < j$ and optimal split at k, i.e. $A_{i \cdots k}$ and $A_{k+1 \cdots j}$
  - Scalar multiplications for $A_{i \cdots k}$ is $m[i, k]$
  - Scalar multiplication for $A_{k+1 \cdots j}$ is $m[k+1, j]$
  - The dimension of $A_{i \cdots k}$ is $p_{i-1} \times p_k$
  - The dimension of $A_{k+1 \cdots j}$ is $p_k \times p_j$
  - Scalar multiplications for $A_{i \cdots k} A_{k+1 \cdots j}$ is $p_{i-1} p_k p_j$

Step 2. A recursive solution…

- Thus, we have

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$$

- This recursive assume we know the value of k, which we do not.
- There are $j - i$ possible values for k, i.e.
  - $k = i, i+1, \cdots, j-1$
- We need to check all of them and find the best

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j\ . \end{cases}$$

Step 2. A recursive solution…

- We need to check all of them and find the best

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j\ . \end{cases}$$

- $m[i,j]$ values gives the costs of optimal solutions to a subproblems, does not provide the information to construct an optimal solution, i.e. where to split
- $s[i,j]$ to be a value of k which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization, i.e. $s[i,j] =$k

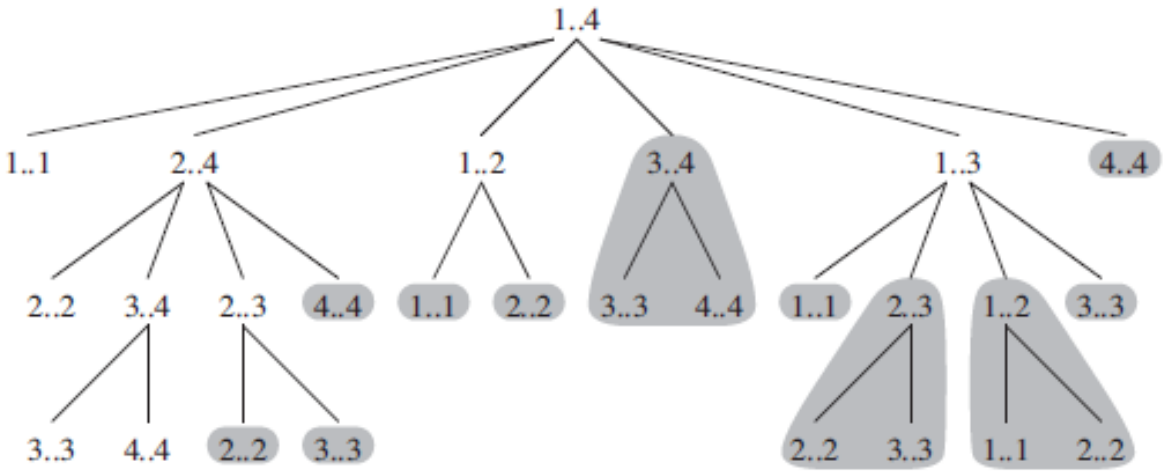
Step 3. Computing the optimal costs

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j\ . \end{cases}$$

- Given the above cost formulae, a recursive algorithm can be written to calculate the minimum cost $m[1,n]$ for $A_1 A_2 \cdots A_n$
- $p =< p_{i-1}, p_i, \cdots, p_n >$ dimensions of the matrices in the chain

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
           + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
           + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```

Step 3. Computing the optimal costs



RECURSIVE-MATRIX-CHAIN (p, 1,4)

- The recursive takes exponential time.
- It is no better than the brute-force method.

Step 3. Computing the optimal costs…

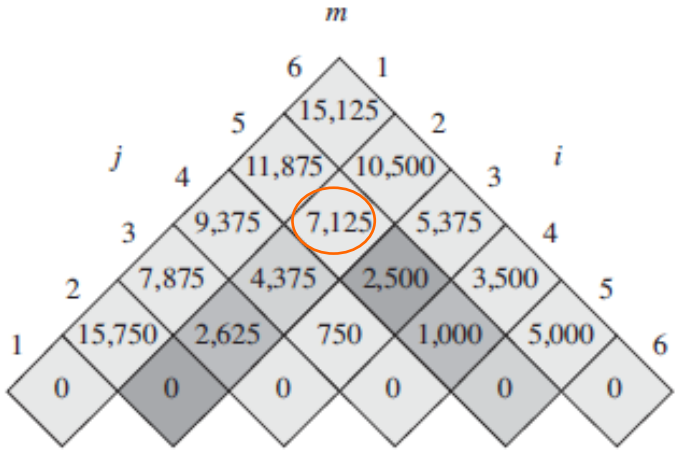
$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j\ . \end{cases}$$

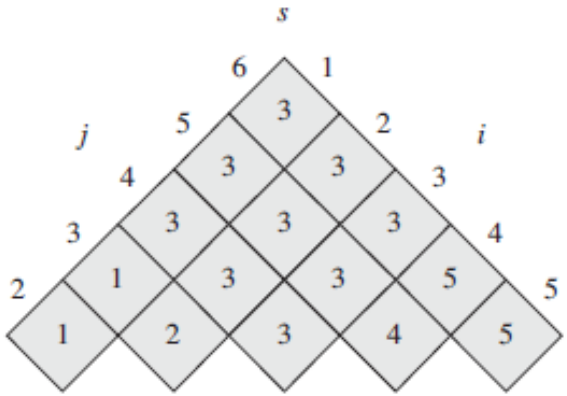
- Compute the optimal cost by using a tabular, bottom-up approach
  - Dimension of A_i is $p_{i-1} \times p_i$,
  - $p =< p_0, p_1, \cdots, p_n >$ and $p.length = n + 1$

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

Complexity $O(n^3)$

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25





$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$
$$= 7125\,.$$

- The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1,6] = 15{,}125$.

Step 4. Constructing an optimal solution

- The table $s[1 \ldots n-1, 2 \ldots n]$ gives us the information we need to do so.
- Each entry $s[i, j]$ records a value of k such that an optimal parenthesization of $A_{i \ldots j}$ splits the product between A_k and A_{k+1}.
  - $A_{1 \cdots n} \rightarrow A_{1 \cdots s[1,n]} A_{s[1,n]+1, \ldots n}$ split at $s[1, n]$
  - $A_{1 \ldots s[1,n]} \rightarrow$ split at $s[1, s[1, n]]$
  - $A_{s[1,n]+1 \ldots n} \rightarrow$ split at $s[s[1, n] + 1, n]$
  - .....

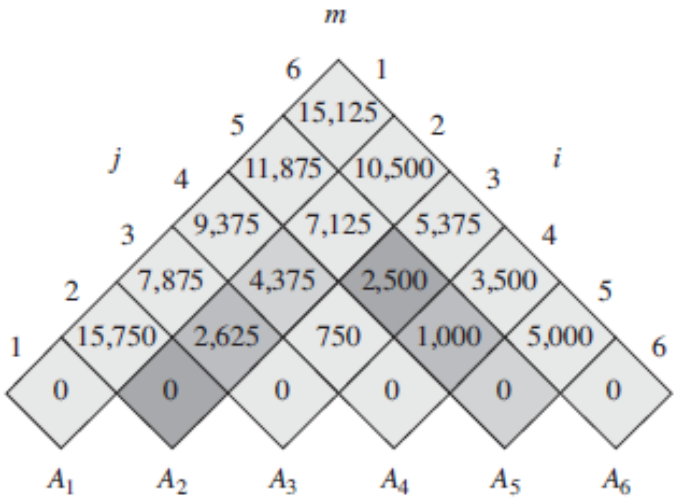
Step 4. Constructing an optimal solution

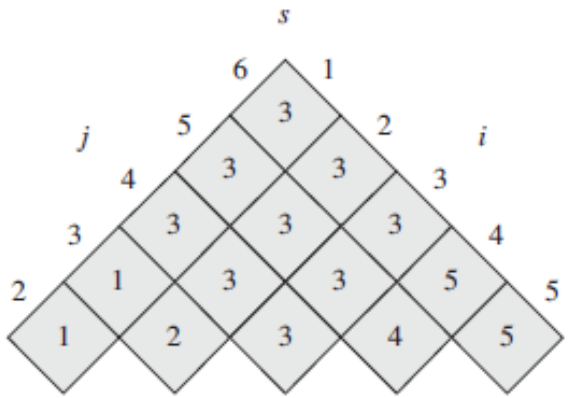
- An optimal parenthesization of $A_1 A_2 \cdots A_n$ is

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2       print "A"_i
3  else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2       print "A"_i
3  else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25





$$((A_1(A_2A_3))((A_4A_5)A_6))$$

Related References

- Introduction to *Algorithms*, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 15.2