Pertemuan 9
Dynamic Programming

1

Dynamic Programming (DP)

- Like divide-and-conquer, solve problem by combining the solutions to sub-problems.
- Differences between divide-and-conquer and DP:
  - Independent sub-problems, solve sub-problems independently and recursively, (so same sub(sub)problems solved repeatedly)
  - Sub-problems are dependent, i.e., sub-problems share sub-sub-problems, every sub(sub)problem solved just once, solutions to sub(sub)problems are stored in a table and used for solving higher level sub-problems.

2

Example: Fibonacci numbers

- **Recall definition of Fibonacci numbers:**

$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$
$$F(0) = 0$$
$$F(1) = 1$$

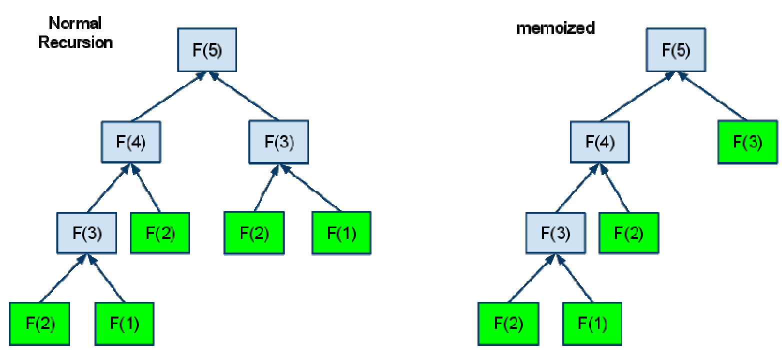
- **Computing the n^{th} Fibonacci number recursively (top-down):**

$$F(n)$$
$$F(n\text{-}1) \quad + \quad F(n\text{-}2)$$
$$F(n\text{-}2) \quad + \quad F(n\text{-}3) \qquad F(n\text{-}3) \quad + \quad F(n\text{-}4)$$

…

Top down approach + memoization

- Always remember the past…



4

Bottom up approach

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$F(0) = 0$
$F(1) = 1$
$F(2) = 1+0 = 1$
…
$F(n-2) =$
$F(n-1) =$
$F(n) = F(n-1) + F(n-2)$

0	**1**	**1**	**. . .**	***F*(*n*-2)**	***F*(*n*-1)**	***F*(*n*)**

Efficiency:
- time n
- space n

Application domain of DP

- Optimization problem: find a solution with optimal (maximum or minimum) value.
- *An* optimal solution, not *the* optimal solution, since may more than one optimal solution, any one is OK.

6

Typical steps of DP

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution in a bottom-up fashion.
- Compute an optimal solution from computed/stored information.

7

Elements of DP

- Optimal (sub)structure
  - An optimal solution to the problem contains within it optimal solutions to subproblems.
- Overlapping subproblems
  - The space of subproblems is "small" in that a recursive algorithm for the problem solves the same subproblems over and over. Total number of distinct subproblems is typically polynomial in input size.
- (Reconstruction an optimal solution)

8

Matrix-chain multiplication (MCM)

- Problem: given <A_1, A_2, …,A_n>, compute the product: $A_1 \times A_2 \times \ldots \times A_n$, find the fastest way (i.e., minimum number of multiplications) to compute it.
- Suppose two matrices $A(p,q)$ and $B(q,r)$, compute their product $C(p,r)$ in $p \times q \times r$ multiplications
  - **for** a=1 **to** p
    - **for** b=1 **to** r
      - **for** c=1 **to** q
        - » $C[a,b] = C[a,b] + A[a,c]B[c,b]$

9

Matrix-chain multiplication

- Different parenthesizations will have different number of multiplications for product of multiple matrices
- Example: **A**(10,100), **B**(100,5), **C**(5,50)
  - If ((**A** ×**B**) ×**C**): 10 ×100 ×5 +10 ×5 ×50 =7500
  - If (**A** ×(**B** ×**C**)): 10 ×100 ×50+100 ×5 ×50=75000
- The first way is ten times faster than the second !!!
- Denote <A_1, A_2, …,A_n> by < $p_0,p_1,p_2,\ldots,p_n$>
  - i.e, $A_1(p_0,p_1)$, $A_2(p_1,p_2)$, …, $A_i(p_{i-1},p_i)$,… $A_n(p_{n-1},p_n)$

10

Matrix-chain multiplication –MCM

- Intuitive brute-force solution: Counting the number of parenthesizations by exhaustively checking all possible parenthesizations.
- Let P(n) denote the number of alternative parenthesizations of a sequence of n matrices:
  - $P(n) = \begin{cases} 1 & \text{if } n=1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n\geq 2 \end{cases}$
- The solution to the recursion is $\Omega(2^n)$.
- So brute-force will not work.

MCP Steps

- Step 1: structure of an optimal parenthesization
  - Let $A_{i..j}$ ($i\leq j$) denote the matrix resulting from $A_i\times A_{i+1}\times\ldots\times A_j$
  - Any parenthesization of $A_i\times A_{i+1}\times\ldots\times A_j$ must split the product between A_k and A_{k+1} for some k, ($i\leq k<j$). The cost = #computing $A_{i..k}$ + #computing $A_{k+1..j}$ + # $A_{i..k}\times A_{k+1..j}$.
  - $A_i\times A_{i+1}\times\ldots\times A_k \times A_{k+1}\times\ldots\times A_j$

MCM Steps

- Step 2: a recursive relation
  - Let m[i,j] be the minimum number of multiplications for $A_i \times A_{i+1} \times \ldots \times A_j$
  - m[1,n] will be the answer
  - $m[i,j] = \begin{cases} 0 \text{ if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} \text{ if } i<j \end{cases}$

13

A Recursive Algorithm for Matrix-Chain Multiplication

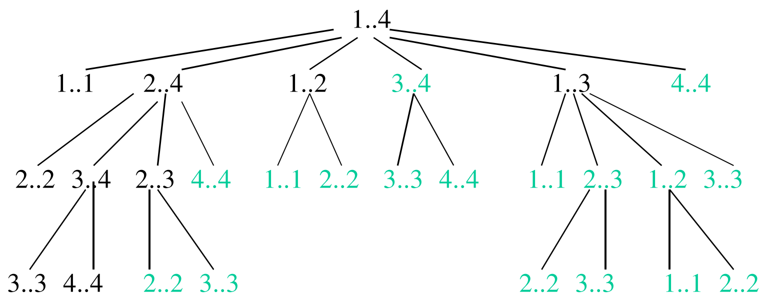
RECURSIVE-MATRIX-CHAIN(p,i,j) (called with(p,1,n))

1. **if** i=j **then return** 0
2. $m[i,j] \leftarrow \infty$
3. **for** $k \leftarrow i$ to j-1
4. **do** $q \leftarrow$ RECURSIVE-MATRIX-CHAIN(p,i,k)+ RECURSIVE-MATRIX-CHAIN(p,k+1,j)+$p_{i-1}p_kp_j$
5. **if** $q < m[i,j]$ **then** $m[i,j] \leftarrow q$
6. **return** $m[i,j]$

The running time of the algorithm is $O(2^n)$

14

Recursion tree for the computation of RECURSIVE-MATRIX-CHAIN(p,1,4)



This divide-and-conquer recursive algorithm solves the overlapping problems *over and over.*
In contrast, DP solves the same (overlapping) subproblems only once (at the first time), then store the result in a table, when the same subproblem is encountered later, just look up the table to get the result.
The computations in green color are replaced by table look up in MEMOIZED-MATRIX-CHAIN(p,1,4)
The divide-and-conquer is better for the problem which generates brand-new problems at each step of recursion.

15

MCM Steps

- Step 3, Computing the optimal cost
  - If by recursive algorithm, exponential time $\Omega(2^n)$ (ref. to P.346 for the proof.), no better than brute-force.
  - Total number of subproblems: $\binom{n}{2}+n = \Theta(n^2)$
  - Recursive algorithm will encounter the same subproblem many times.
  - If tabling the answers for subproblems, each subproblem is only solved once.
  - The second hallmark of DP: overlapping subproblems and solve every subproblem just once.

16

MCM DP Steps

- Step 3, Algorithm,
  - array $m[1..n,1..n]$, with $m[i,j]$ records the optimal cost for $A_i \times A_{i+1} \times \ldots \times A_j$.
  - array $s[1..n,1..n]$, $s[i,j]$ records index k which achieved the optimal cost when computing $m[i,j]$.
  - Suppose the input to the algorithm is $p = < p_0, p_1, \ldots, p_n >$.

17

MCM DP Steps

```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
      do m[i, i] ← 0
for l ← 2 to n              ▷ l is the chain length.
      do for i ← 1 to n − l + 1
              do j ← i + l − 1
                 m[i, j] ← ∞
                 for k ← i to j − 1
                      do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                         if q < m[i, j]
                           then m[i, j] ← q
                                s[i, j] ← k
return m and s
```

18

MCM DP—order of matrix computations

$m(1,1)$ $m(1,2)$ $m(1,3)$ $m(1,4)$ $m(1,5)$ $m(1,6)$

$m(2,2)$ $m(2,3)$ $m(2,4)$ $m(2,5)$ $m(2,6)$

$m(3,3)$ $m(3,4)$ $m(3,5)$ $m(3,6)$

$m(4,4)$ $m(4,5)$ $m(4,6)$

$m(5,5)$ $m(5,6)$

$m(6,6)$

19

MCM DP Example

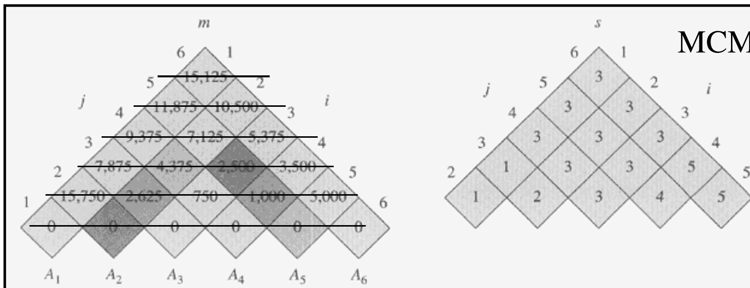


Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1,6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$
$$= 7125\,.$$

20

MCM DP Steps

- Step 4, constructing a parenthesization order for the optimal solution.
  - Since $s[1..n,1..n]$ is computed, and $s[i,j]$ is the split position for $A_iA_{i+1}\ldots A_j$, i.e, $A_i\ldots A_{s[i,j]}$ and $A_{s[i,j]+1}\ldots A_j$, thus, the parenthesization order can be obtained from $s[1..n,1..n]$ recursively, beginning from $s[1,n]$.

21

MCM DP Steps

- Step 4, algorithm

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i = j
   then print "A"_i
   else print "("
        PRINT-OPTIMAL-PARENS(s, i, s[i, j])
        PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
        print ")"
```

22

Finding Optimal substructures

- Show a solution to the problem consists of making a choice, which results in one or more subproblems to be solved.
- Suppose you are given a choice leading to an optimal solution.
  - Determine which subproblems follows and how to characterize the resulting space of subproblems.
- Show the solution to the subproblems used within the optimal solution to the problem must themselves be optimal by cut-and-paste technique.

23

Optimal Substructure Varies in Two Ways

- How many subproblems
  - In matrix-chain multiplication: two subproblems
- How many choices
  - In matrix-chain multiplication: j-i choices
- DP solve the problem in bottom-up manner.

24

Running Time for DP Programs

- #overall subproblems × #choices.
  - In matrix-chain multiplication, $O(n^2) \times O(n) = O(n^3)$
- The cost =costs of solving subproblems + cost of making choice.
  - In matrix-chain multiplication, choice cost is $p_{i-1}p_kp_j$.

25

Reconstructing an Optimal Solution

- An auxiliary table:
  - Store the choice of the subproblem in each step
  - Reconstructing the optimal steps from the table.

26

Memoization

- A variation of DP
- Keep the same efficiency as DP
- But in a top-down manner.
- Idea:
  - Each entry in table initially contains a value indicating the entry has yet to be filled in.
  - When a subproblem is first encountered, its solution needs to be solved and then is stored in the corresponding entry of the table.
  - If the subproblem is encountered again in the future, just look up the table to take the value.

27

Memoized Matrix Chain

```
MEMOIZED-MATRIX-CHAIN(p)
n ← length[p] − 1
for i ← 1 to n
    do for j ← i to n
           do m[i, j] ← ∞
return LOOKUP-CHAIN(p, 1, n)
```

```
LOOKUP-CHAIN(p,i,j)
if m[i,j]<∞ then return m[i,j]
if i=j then m[i,j] ←0
    else for k←i to j-1
            do q← LOOKUP-CHAIN(p,i,k)+
                   LOOKUP-CHAIN(p,k+1,j)+p_{i-1}p_k p_j
               if q< m[i,j] then m[i,j] ←q
return m[i,j]
```

28

DP VS. Memoization

- MCM can be solved by DP or Memoized algorithm, both in $O(n^3)$.
  - Total $\Theta(n^2)$ subproblems, with $O(n)$ for each.
- If all subproblems must be solved at least once, DP is better by a constant factor due to no recursive involvement as in Memoized algorithm.
- If some subproblems may not need to be solved, Memoized algorithm may be more efficient, since it only solve these subproblems which are definitely required.

29

DP VS. Memoization

- MCM can be solved by DP or Memoized algorithm, both in $O(n^3)$.
  - Total $\Theta(n^2)$ subproblems, with $O(n)$ for each.
- If all subproblems must be solved at least once, DP is better by a constant factor due to no recursive involvement as in Memoized algorithm.
- If some subproblems may not need to be solved, Memoized algorithm may be more efficient, since it only solve these subproblems which are definitely required.

30

Knapsack 0-1 Problem

- The goal is to **maximize the value of a knapsack** that can hold at most W units (i.e. lbs or kg) worth of goods from a list of items $I_0, I_1, \ldots I_{n-1}$.
  - Each item has 2 attributes:
    1) Value – let this be v_i for item I_i
    2) Weight – let this be w_i for item I_i

Knapsack 0-1 Problem

- The difference between this problem and the fractional knapsack one is that you CANNOT take a fraction of an item.
  - You can either take it or not.
  - Hence the name Knapsack 0-1 problem.

Knapsack 0-1 Problem

- Brute Force
  - The naïve way to solve this problem is to cycle through all 2^n subsets of the n items and pick the subset with a legal weight that maximizes the value of the knapsack.
  - We can come up with a dynamic programming algorithm that will USUALLY do better than this brute force technique.

Knapsack 0-1 Problem

- As we did before we are going to solve the problem in terms of sub-problems.
  - So let's try to do that…
- Our first attempt might be to characterize a sub-problem as follows:
  - Let S_k be the optimal subset of elements from $\{I_0, I_1, \ldots, I_k\}$.
    - What we find is that the optimal subset from the elements $\{I_0, I_1, \ldots, I_{k+1}\}$ may not correspond to the optimal subset of elements from $\{I_0, I_1, \ldots, I_k\}$ in any regular pattern.
  - Basically, the solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

Knapsack 0-1 Problem

- Let's illustrate that point with an example:

Item	Weight	Value
I_0	**3**	**10**
I_1	**8**	**4**
I_2	**9**	**9**
I_3	**8**	**11**

- **The maximum weight the knapsack can hold is 20.**

- The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$
- BUT the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$.
  - In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$.
    - (Instead it build's upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12 or less.)

Knapsack 0-1 problem

- So now we must re-work the way we build upon previous sub-problems…
  - Let **B[k, w]** represent the maximum total value of a subset S_k with weight w.
  - Our goal is to find **B[n, W],** where n is the total number of items and W is the maximal weight the knapsack can carry.

- So our recursive formula for subproblems:

$$\begin{aligned} B[k, w] &= B[k-1, w], \text{ if } w_k > w \\ &= \max\{B[k-1, w], B[k-1, w-w_k] + v_k\}, \text{ otherwise} \end{aligned}$$

- In English, this means that the best subset of S_k that has total weight w is:
  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

Knapsack 0-1 Problem – 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! If it was the total weight would be > w, which is unacceptable.
- **Second case:** $w_k \leq w$
  - Then the item *k* can be in the solution, and we choose the case with greater value.

Knapsack 0-1 Algorithm

```
for w = 0 to W {  // Initialize 1st row to 0's
   B[0,w] = 0
}
for i = 1 to n {  // Initialize 1st column to 0's
   B[i,0] = 0
}
for i = 1 to n {
   for w = 0 to W {
        if wi <= w {  //item i can be in the solution
                if vi + B[i-1,w-wi] > B[i-1,w]
                        B[i,w] = vi + B[i-1,w- wi]
                else
                        B[i,w] = B[i-1,w]
        }
        else B[i,w] = B[i-1,w] // wi > w
   }
}
```

Knapsack 0-1 Problem

- Let's run our algorithm on the following data:
  - n = 4 (# of elements)
  - W = 5 (max weight)
  - Elements (weight, value):
    (2,3), (3,4), (4,5), (5,6)

Knapsack 0-1 Example

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

```
// Initialize the base cases
for w = 0 to W
        B[0,w] = 0

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

Knapsack 0-1 Example

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					

$i = 1$
$v_i = 3$
$w_i = 2$
$w = 1$
$w-w_i = -1$

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

Knapsack 0-1 Example

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					

$i = 1$
$v_i = 3$
$w_i = 2$
$w = 2$
$w-w_i = 0$

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

Knapsack 0-1 Example

Items:
$\boxed{1: (2,3)}$
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
$v_i = 3$
$w_i = 2$
w = 3
$w-w_i = 1$

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

Knapsack 0-1 Example

Items:
$\boxed{1: (2,3)}$
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
$v_i = 3$
$w_i = 2$
w = 4
$w-w_i = 2$

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

Knapsack 0-1 Example

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					

$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 5}$
$w\text{-}w_i = 3$

if $w_i <= w$ //item i can be in the solution
if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
$B[i,w] = v_i + B[i\text{-}1,w\text{-} w_i]$
else
$B[i,w] = B[i\text{-}1,w]$
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

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					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 1}$
$w\text{-}w_i = -2$

if $w_i <= w$ //item i can be in the solution
if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
$B[i,w] = v_i + B[i\text{-}1,w\text{-} w_i]$
else
$B[i,w] = B[i\text{-}1,w]$
else **$B[i,w] = B[i\text{-}1,w]$** // $w_i > w$

Knapsack 0-1 Example

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
$v_i = 4$
$w_i = 3$
w = 2
$w-w_i = -1$

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

Knapsack 0-1 Example

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
$v_i = 4$
$w_i = 3$
w = 3
$w-w_i = 0$

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

Knapsack 0-1 Example

Items:
1: (2,3)
$\boxed{2: (3,4)}$
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
$v_i = 4$
$w_i = 3$
w = 4
$w\text{-}w_i = 1$

if w_i <= w //item i can be in the solution
    if v_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = v_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

Knapsack 0-1 Example

Items:
1: (2,3)
$\boxed{2: (3,4)}$
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
$v_i = 4$
$w_i = 3$
w = 5
$w\text{-}w_i = 2$

if w_i <= w //item i can be in the solution
    if v_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = v_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

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
$\boxed{\text{3: (4,5)}}$
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		
4	0					

i = 3
$v_i = 5$
$w_i = 4$
w = 1..3
$w-w_i = -3..-1$

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

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
$\boxed{\text{3: (4,5)}}$
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
$v_i = 5$
$w_i = 4$
w = 4
$w-w_i = 0$

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

Knapsack 0-1 Example

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					

$i = 3$
$v_i = 5$
$w_i = 4$
$w = 5$
$w-w_i = 1$

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

Knapsack 0-1 Example

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	

$i = 4$
$v_i = 6$
$w_i = 5$
$w = 1..4$
$w-w_i = -4..-1$

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

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
$\boxed{4: (5,6)}$

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

$i = 4$
$v_i = 6$
$w_i = 5$
$\mathbf{w = 5}$
$w\text{-}w_i = 0$

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

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
$\boxed{4: (5,6)}$

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

We're DONE!!
The max possible value that can be carried in this knapsack is ***$7***

Knapsack 0-1 Algorithm

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

- To know the ***items*** that make this maximum value, we need to trace back through the table.

Knapsack 0-1 Algorithm
Finding the Items

- Let i = n and k = W
  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 // Assume the i^{th} item is not in the knapsack
  // Could it be in the optimally packed knapsack?

Knapsack 0-1 Algorithm Finding the Items

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

Knapsack:

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

i = n , k = W
while i, k > 0
  if *B[i, k]* ≠ *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

Knapsack 0-1 Algorithm Finding the Items

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

Knapsack:

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

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

Knapsack 0-1 Algorithm Finding the Items

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

Knapsack:
Item 2

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
$v_i = 4$
$w_i = 3$
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]* ≠ *B[i-1, k]* then
        mark the i^{th} *item as in the knapsack*
        $i = i\text{-}1,\ k = k\text{-}w_i$
    else
        i = i-1

Knapsack 0-1 Algorithm Finding the Items

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

Knapsack:
Item 2
Item 1

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
$v_i = 3$
$w_i = 2$
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]* ≠ *B[i-1, k]* then
        mark the i^{th} *item as in the knapsack*
        $i = i\text{-}1,\ k = k\text{-}w_i$
    else
        i = i-1

Knapsack 0-1 Algorithm Finding the Items

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

Knapsack:
Item 2
Item 1

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

k = 0, so we're DONE!

The optimal knapsack should contain:
Item 1 and Item 2

Knapsack 0-1 Problem – Run Time

for w = 0 to W
  B[0,w] = 0 **O(*W*)**

for i = 1 to n
  B[i,0] = 0 **O(*n*)**

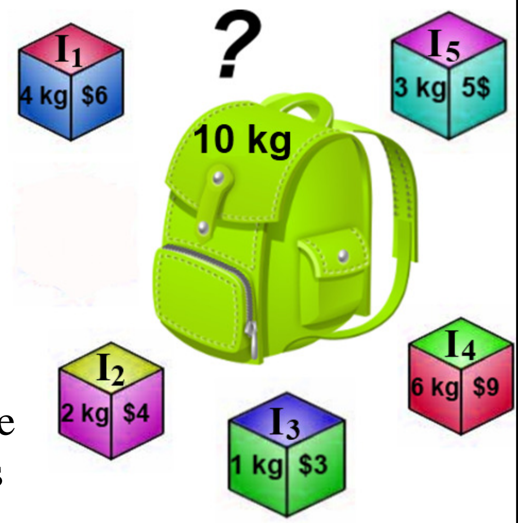
for i = 1 to n **Repeat *n* times**
  for w = 0 to W **O(*W*)**
    < the rest of the code >

What is the running time of this algorithm?
  O(*n*W*)

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

Knapsack Problem

1) Fill out the dynamic programming table for the knapsack problem to the right.
2) Trace back through the table to find the items in the knapsack.



References

- Slides adapted from Arup Guha's Computer Science II Lecture notes: http://www.cs.ucf.edu/~dmarino/ucf/cop3503/lectures/
- Additional material from the textbook:

  Data Structures and Algorithm Analysis in Java (Second Edition) by Mark Allen Weiss
- Additional images:

  www.wikipedia.com

  xkcd.com