Dynamic programming

(0-1 Knapsack problem)

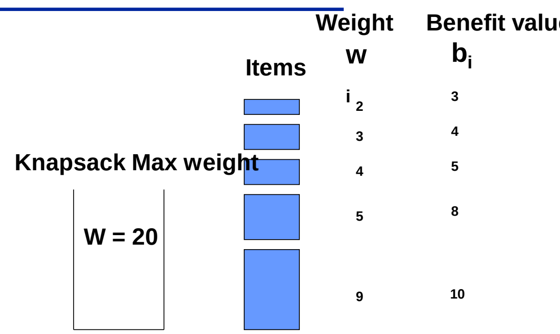
Properties of a problem that can be solved with dynamic programming

- Simple Subproblems
  - » We should be able to break the original problem to smaller subproblems that have the same structure
- Optimal Substructure of the problems
  - » The solution to the problem must be a composition of subproblem solutions
- Subproblem Overlap
  - » Optimal subproblems to unrelated problems can contain subproblems in common

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)
- <u>Problem</u>: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem



0-1 Knapsack problem

- Problem is to find

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

5

0-1 Knapsack problem: Brute-force approach

- 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 **$O(2^n)$**

6

0-1 Knapsack problem: Brute-force approach

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

7

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.
- Can we describe the final solution (S_n) in terms of subproblems (S_k)?

8

Defining a Subproblem

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

Max weight: W = 20
For S_4:
Total weight: 14;
total benefit: 20

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

S_4

S_5

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

For S_5:
Total weight: 20
total benefit: 26

Solution for S_4 is not part of the solution for S_5

9

Defining a Subproblem (continued)

- The solution for S_4 is **not** part of the solution for S_5
- **Definition of a subproblem is flawed → need another one.**
- Let's add another parameter: **w**, which will represent the <u>**exact**</u> **weight for each subset of items**
- **The subproblem then will be to compute B[k,w]**

10

Recursive Formula

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

- The best subset of S_k that has the total weight $\leq w$, 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: $w_k \leq w$. Then the item k <u>can</u> be in the solution, and we choose *the case with greater value*.

0-1 Knapsack Algorithm

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

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
    V[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
    V[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 + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else **V[i,w] = V[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 + V[i-1,w-w_i] > V[i-1,w]
    V[i,w] = b_i + V[i-1,w- w_i]
  else
    V[i,w] = V[i-1,w]
else V[i,w] = V[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 + V[i-1,w-w_i] > V[i-1,w]
    V[i,w] = b_i + V[i-1,w- w_i]
  else
    V[i,w] = V[i-1,w]
else V[i,w] = V[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 + V[i-1,w-w_i] > V[i-1,w]
    V[i,w] = b_i + V[i-1,w- w_i]
  else
    V[i,w] = V[i-1,w]
else V[i,w] = V[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 + V[i-1,w-w_i] > V[i-1,w]
    V[i,w] = b_i + V[i-1,w- w_i]
  else
    V[i,w] = V[i-1,w]
else V[i,w] = V[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 + V[i-1,w-w_i] > V[i-1,w]
V[i,w] = b_i + V[i-1,w- w_i]
else
V[i,w] = V[i-1,w]
else **V[i,w] = V[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 + V[i-1,w-w_i] > V[i-1,w]
V[i,w] = b_i + V[i-1,w- w_i]
else
V[i,w] = V[i-1,w]
else **V[i,w] = V[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 + V[i-1,w-w_i] > V[i-1,w]
V[i,w] = b_i + V[i-1,w- w_i]
else
V[i,w] = V[i-1,w]
else V[i,w] = V[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 + V[i-1,w-w_i] > V[i-1,w]
V[i,w] = b_i + V[i-1,w- w_i]
else
V[i,w] = V[i-1,w]
else V[i,w] = V[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 + V[i-1,w-w_i] > V[i-1,w]
    V[i,w] = b_i + V[i-1,w- w_i]
  else
    V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

25

Example (14)

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**		
4	0					

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 + V[i-1,w-w_i] > V[i-1,w]
    V[i,w] = b_i + V[i-1,w- w_i]
  else
    V[i,w] = V[i-1,w]
else **V[i,w] = V[i-1,w]** // w_i > w

26

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 + V[i-1,w-w_i] > V[i-1,w]
    V[i,w] = b_i + V[i-1,w- w_i]
  else
    V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

27

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 + V[i-1,w-w_i] > V[i-1,w]
    V[i,w] = b_i + V[i-1,w- w_i]
  else
    V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

28

Example (17)

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
b_i=6
w_i=5
w= 1..4

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

29

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 + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // w_i > w
```

30

Running time

```
for w = 0 to W
    V[0,w] = 0
for i = 1 to n
    V[i,0] = 0
for i = 1 to n
    for w = 0 to W
        < the rest of the code >
```

O(W)

Repeat *n* times

O(W)

What is the running time of this algorithm?

O(n*W)

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

31

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
  - » i.e., the value in V[n,W]
- To know the items that make this maximum value, an addition to this algorithm is necessary

32

How to find actual Knapsack Items

- All of the information we need is in the table.
- $V[n,W]$ is the maximal value of items that can be placed in the Knapsack.
- Let i=n and k=W

if $V[i,k] \neq V[i-1,k]$ 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 <u>not</u> 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
$V[i,k] = 7$
$V[i-1,k] = 7$

i=n, k=W
while i,k > 0
 if $V[i,k] \neq V[i-1,k]$ 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
$V[i,k] = 7$
$V[i-1,k] = 7$

i=n, k=W
while i,k > 0
 if $V[i,k] \neq V[i-1,k]$ 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=5
w_i=4
$V[i,k] = 7$
$V[i-1,k] = 7$

i=n, k=W
while i,k > 0
    if $V[i,k] \neq V[i-1,k]$ 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
$V[i,k] = 7$
$V[i-1,k] = 3$
$k - w_i$=2

i=n, k=W
while i,k > 0
    if $V[i,k] \neq V[i-1,k]$ 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
$V[i,k] = 3$
$V[i-1,k] = 0$
$k - w_i$=0

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

41

Memorization (Memory Function Method)

- *Goal:*
  - » *Solve only subproblems that are necessary and solve it only once*
- *Memorization* is another way to deal with overlapping subproblems in dynamic programming
- With memorization, we implement the algorithm ***recursively***:
  - » If we encounter a new subproblem, we compute and store the solution.
  - » If we encounter a subproblem we have seen, we look up the answer
- Most useful when the algorithm is easiest to implement recursively
  - » Especially if we do not need solutions to all subproblems.

42

0-1 Knapsack Memory Function Algorithm

```
for i = 1 to n
    for w = 1 to W
        V[i,w] = -1

for w = 0 to W
    V[0,w] = 0
for i = 1 to n
    V[i,0] = 0
```

```
MFKnapsack(i, w)
  if V[i,w] < 0
     if w < w_i
        value = MFKnapsack(i-1, w)
     else
        value = max(MFKnapsack(i-1, w),
                    b_i + MFKnapsack(i-1, w-w_i))
     V[i,w] = value
  return V[i,w]
```

43

Conclusion

- 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 (memorization)
- Running time of dynamic programming algorithm vs. naïve algorithm:
  - » 0-1 Knapsack problem: **O(W*n)** vs. **O(2^n)**

44