Data Structures & Algorithms

0-1 Knapsack problem

Knapsack problem

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 values.

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"
   - Items are indivisible; you either take an item or not. Some special instances can be solved with *dynamic programming*

2. "Fractional knapsack problem"
   - Items are divisible: you can take any fraction of an item

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 and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem

- Problem, in other words, is to find

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

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

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

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

0-1 Knapsack problem: dynamic programming approach

- We can do better with an algorithm based on dynamic programming

- We need to carefully identify the subproblems

Defining a Subproblem

- 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 and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

Defining a Subproblem

- Let's add parameter: *w*, which will represent the maximum weight for each subset of items

- The subproblem then will be to compute $V[k,w]$, *i.e.*, to find an optimal solution for S_k = *{items labeled 1, 2, .. k} in a knapsack of size w*

Recursive Formula for subproblems

- The subproblem will then be to compute *V[k,w]*, *i.e.*, to find an optimal solution for S_k = *{items labeled 1, 2, .. k} in a knapsack of size w*

- Assuming knowing V[i, j], where i=0,1, 2, … k-1, j=0,1,2, …w, how to derive V[k,w]?

Recursive Formula for subproblems (continued)

Recursive formula for subproblems:

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

It 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 $\leq w$, **or**

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

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 can 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 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
```

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 $O(2^n)$

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 $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $\mathbf{V[i,w] = b_i + V[i-1,w- w_i]}$
    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 $b_i + V[i-1,w-w_i] > V[i-1,w]$

        $\mathbf{V[i,w] = b_i + V[i-1,w- w_i]}$

    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

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]** // $w_i > w$

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

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] // $w_i > w$

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]** // $w_i > w$

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

Exercise

- P303 8.2.1 (a).

1. a. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

- How to find out which items are in the optimal subset?

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

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]* ≠ *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=0
k= 0

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
        $i = i-1$

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] ≠ V[i–1,k] then
        mark the nth item as in the knapsack
        i = i–1, k = k-w_i
    else
        i = i–1
```

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.

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 $V[i,w]$

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

In-Class Exercise

A *contiguous subsequence* of a list S is a subsequence made up of consecutive elements of S. For instance, if S is

$$5, 15, -30, 10, -5, 40, 10,$$

then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Give a linear-time algorithm for the following task:

Input: A list of numbers, $a_1, a_2, \ldots, a_n$.

Output: The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero).

For the preceding example, the answer would be $10, -5, 40, 10$, with a sum of 55.

(*Hint:* For each $j \in \{1, 2, \ldots, n\}$, consider contiguous subsequences ending exactly at position j.)

Brute-Force Approach

```
public static int MCSS(int [] a) {

    int max = 0, sum = 0, start = 0, end = 0;

    // Cycle through all possible values of start and end indexes
    // for the sum.
    for (i = 0; i < a.length; i++) {
        for (j = i; j < a.length; j++) {
            sum = 0;

            // Find sum A[i] to A[j].
            for (k = i; k <= j; k++)
                sum += a[k];
            if (sum > max) {
                max = sum;
                start = i; // Although method doesn't return these
                end = j;  // they can be computed.
            }
        }
    }
    return max;
}
```

Dynamic-Programming Approach

- (1) SMaxV(0) = 0
- (2) MaxV(0) = 0
- (3) for i=1 to n
- (4) SMaxV(i) = max(SmaxV(i-1)+x_i, 0)
- (5) MaxV(i) = max(MaxV(i-1), SMaxV(i))
- (6) return MaxV(n)

- Run the algorithm on the following example instance:
  - » 30, 40, -100, 10, 20, 50, -60, 90, -180, 100