Introduction to Algorithms

0-1 Knapsack Problem

Xiang-Yang Li and Haisheng Tan

School of Computer Science and Technology
University of Science and Technology of China (USTC)

Fall Semester 2024

Outline

Contents

Knapsack Problem

- The knapsack problem is a NP-complete problem of combinatorial optimization. Similar problems often appear in the fields of business, mathematics, computational complexity theory, cryptography, and applied mathematics.
- The knapsack problem has been studied for more than a century, with early works dating as far back as 1897.
- Application: find the least wasteful way to cut raw materials, choose investment and portfolio, choose asset-backed asset securitization, generate keys for Merkle-Hellman and other backpack cryptosystems.

Knapsack Problem

- Suppose we are planning a hiking trip; and we are, therefore, interested in filling a knapsack with items that are considered necessary for the trip.
- There are n different item types that are deemed desirable; these could include bottle of water, apple, orange, sandwich, and so forth. Each item type has a given set of two attributes, namely a weight (or volume) and a value that quantifies the level of importance associated with each unit of that type of item.
- Since the knapsack has a limited weight (or volume) capacity, the problem of interest is to figure out how to load the knapsack with a combination of units of the specified types of items that yields the greatest total value.

Knapsack Problem

Problem Definition(Knapsack):

- Input: Knapsack takes a set S of n items, each with benefit b_i and weight w_i, and a knapsack with weight bound W (for simplicity we assume that all elements have $w_i \leq W$).
- Output: Find a subset of items $I \subseteq S$ that maximizes $\sum_{i \in I} b_i$, and satisfies the constraint $\sum_{i \in I} w_i \leq W$.

Knapsack Problem

There are two versions of the problem:

- Fractional knapsack problem: Items are divisible; you can take any fraction of an item.
- 0-1 knapsack problem: Items are indivisible; you either take an item or not.

Contents

Greedy Algorithm for Knapsack

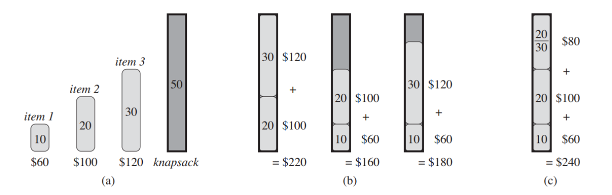
Greedy-Algorithm()

1: Sort items in non-increasing order of $\frac{b_i}{w_i}$.
2: Greedily pick items in the above order.

- To solve the fractional problem, we first compute the benefit per weight b_i/w_i for each item;
- Obeying a greedy strategy, we begins by taking as much as possible of the item with the greatest value per pound;
- Then we takes the next greatest valuable item, and so forth until he fills the knapsack;
- Thus, by sorting the items by value per pound, the greedy algorithm runs in $O(n \lg n)$ time.
- The fractional knapsack problem has the greedy-choice property.

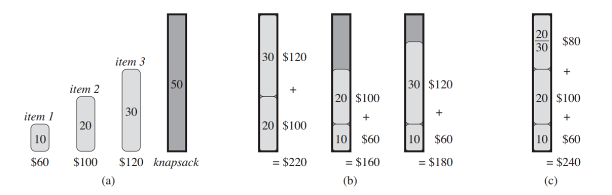
Greedy Algorithm for Knapsack

- But this greedy strategy does not work for the $0-1$ knapsack problem. To see the reason, consider the problem instance illustrated in Figure 16.2(a).
- The benefit per weight of item 1 is 6 per weight, which is greater than that of either item 2 (5 per weight) or item 3 (4 per weight).
- However, the optimal solution takes items 2 and 3, leaving 1 behind. The two possible solutions that involve item 1 are both suboptimal.



Greedy Algorithm for Knapsack

- The reason is that taking item 1 we are unable to fill the knapsack to capacity, and the empty space lowers the effective profit per size of our load.
- But for the comparable fractional problem, the greedy strategy, which takes item 1 first, does yield an optimal solution, as shown in Figure 16.2(c).



Greedy Algorithm for Knapsack: Very Bad

Greedy performs arbitraruly bad in the worst case.

Assume that there are two items. The first one has weight $\varepsilon > 0$ and benefit 2ε, and the second one has weight B and benefit B. The capacity of the knapsack is B.

Our greedy algorithm will only pick the small item, and the benefit is 2ε. The optimal solution is to pick the second item, with benefit B. This example makes this greedy method a pretty bad algorithm.

Greedy-Redux Algorithm for Knapsack: Small Twist

Therefore, we make the following small adjustment to our greedy algorithm:

Greedy-Algorithm Redux()

1: Sort items in non-increasing order of $\frac{b_i}{w_i}$ // we here denote each item as a_i, where $1 \leq i \leq n$.
2: Greedily add items until we hit an item a_i that is too big. $(\sum_{k=1}^{i} w_k > W \geq \sum_{k=1}^{i-1} w_k)$.
3: Pick the better of $\{a_1, a_2, ..., a_{i-1}\}$ and a_i.

Greedy-Redux Algorithm for Knapsack: Bounded Approximation Ratio

Theorem: Greedy Algorithm Redux is a 2-approximation for the knapsack problem.
Proof: We employed a greedy algorithm. Therefore we can say that if our solution is suboptimal, we must have some leftover space W_{rest} at the end. Imagine for a second that our algorithm was able to take a fraction of an item. Then, by adding $\frac{W_{rest}}{w_i} b_i$ to our knapsack value, we would either match or exceed OPT (remember that OPT is unable to take fractional items), i.e., $\sum_{k=1}^{i-1} b_k + \frac{W_{rest}}{w_i} b_i \geq OPT$.
Therefore, either $\sum_{k=1}^{i-1} b_k \geq \frac{1}{2}OPT$ or $b_i \geq \frac{W_{rest}}{w_i} b_i \geq \frac{1}{2}OPT$

Contents

Dynamic Programming

- We can do better with an algorithm based on dynamic programming.
- We need to carefully identify the subproblems.

Dynamic Programming

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 b_i (Here, we can assume all w_i and W are integer values.)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?
- Let's add another parameter: w, which will represent the weight of the knapsack for a subproblem.

Dynamic Programming

Defining a Subproblem

- 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 $i = 0, 1, 2, \ldots, k-1$, $j = 0, 1, 2, \ldots, w$, how to derive $V[k, w]$?

Dynamic Programming

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:

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

Dynamic Programming

```
DP for knapsack()
for w = 0 to W do
    V[0,w]=0
for i = 1 to n do
    V[i,0]=0
for i = 1 to n do
    for w = 0 to W do
        if w_i ≤ W then
            if b_i + V[i−1, w−w_i] > V[i−1, w] then
                V[i, w] = b_i + V[i−1, w−w_i]
        else
            V[i, w] = V[i−1, w]
```

Dynamic Programming

- What is the running time of this algorithm? O(nW)
- Let's run our algorithm on the following data:
  n = 4 (number of items)
  W = 5 (weight bound)
  Elements (weight, benefit):
  (2,3), (3,4), (4,5), (5,6)

Dynamic Programming Example

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
```

Dynamic Programming Example

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
```

Dynamic Programming Example

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
```

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

Dynamic Programming Example

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

Items:

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

Dynamic Programming Example

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

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

Dynamic Programming Example

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
```

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Dynamic Programming Example

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

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

Dynamic Programming Example

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
```

Items:

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

Dynamic Programming Example

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
```

Items:

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

Dynamic Programming Example

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
```

Items:

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

Dynamic Programming Example

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
```

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

Dynamic Programming Example

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
```

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

Dynamic Programming Example

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
```

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

Dynamic Programming Example

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
```

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

Dynamic Programming Example

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
```

Items:

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

Dynamic Programming Example

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

Items:

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

Dynamic Programming Example

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

Items:

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

Dynamic Programming

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 $k = W$.

find actual knapsacks items()

```
if i = n and k = W then
    mark the i-th item as in the knapsack
    i = i − 1, k = k − w_i
else
    i = i − 1
```

Dynamic Programming Example

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
        $i = i-1$

Items:

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

Dynamic Programming Example

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
        $i = i-1$

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

Dynamic Programming Example

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
    $i = i-1$

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

Dynamic Programming Example

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
$V[i,k] = 7$
$V[i-1,k] = 3$
$k - w_i = 2$

Items:

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

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
    $i = i-1$

Dynamic Programming Example

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
    $i = i-1$

Items:

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

Dynamic Programming Example

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] ≠ 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
```

Items:

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

Dynamic Programming Example

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
    $i = i-1$

Items:

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

Contents

Discussion : Pseudo-polynomial

Pseudo-polynomial time:

a numeric algorithm runs in pseudo-polynomial time if its running time is a polynomial in the numeric value of the input — but not necessarily in the length of the input (the number of bits required to represent it)

- The Running time of dynamic programming algorithm on 0-1 Knapsack problem is $O(W * n)$, the number W needs $\log W$ bits to describe, so it is pseudo-polynomial.
- Other pseudo-polynomial algorithm: Primality testing

Discussion: Another DP apprach, Pseudo-polynomial

- Let P be the profit of the most profitable object, i.e. $P = \max_{a \in S} p(a)$. From this, we can upper bound the profit that can be achieved as nP for the n objects. Here, we can assume the benefit of each item are interger values.
- For each $i \in \{1, ..., n\}$ and $p \in \{1, ..., nP\}$, let $S_{i,p}$ denote a subset of $\{a_1, ..., a_i\}$ that has a total profit of exactly p and takes up the least amount of sapce possible.
- Let $A(i,p)$ be the size of the set $S_{i,p}$, with a value of ∞ to denote no such subset.
- For $A(i,p)$, we have the base case $A(1,p)$ where $A(1,p(a_1))$ is $s(a_1)$ and all other values are ∞.

Discussion: Another DP apprach, Pseudo-polynomial

- We can use the following recurrence to caculate all values for $A(i, p)$:

$$A(i+1, p) = \begin{cases} \min\{A(i, p), s(a_{i+1}) + A(i, p - p(a_{i+1}))\}, & \text{if } p(a_{i+1}) \leq p \\ A(i, p), & \text{otherwise} \end{cases}$$

- The optimal subset then corresponds with the set $S_{n,p}$ for which p is maximized and $A(n, p) \leq B$. Since this iterates through at most n different values to caculate each $A(i, p)$ we get a total running time of $O(n^2 P)$ and thus a pseudo-polynomial algorithm for knapsack.
- It is easy to modify the above DP algorithm to achieve a full polynomial-time approximation scheme (FPTAS) for 0-1 knapsack.

Another Dynamic Programming for Knapsack

DP for knapsack()

1: Let P be the maximum benefit of all items.
2: Given $\varepsilon > 0$, let $K = \frac{\varepsilon \cdot P}{n}$.
3: for each object a_i do
4: define a new profit $p'(a_i) = \lfloor \frac{p(a_i)}{K} \rfloor$.
5: With these as profits of n items, using the dynamic programming algorithm presented in previous slide, find the most profitable set, say S'.
6: Output S' as the final solution for the original knapsack problem

Another Dynamic Programming for Knapsack

Theorem

The set S', output by the aforementioned algorithm, satisfies that

$$P(S') \geq (1-\varepsilon)\cdot OPT.$$

Here $P(S')$ denotes the profit (or benefit) from the set S', and OPT is the optimum benefit of the original problem.

Another Dynamic Programming for Knapsack

Proof.

Let O be the optimal set for the original problem, and let $P'(X)$ be the modified profit of set X with profit function $p'()$. Clearly,

$$p(a) - K \le K \cdot p'(a) \le p(a);$$

$$P(O) - K \cdot P'(O) \le n \cdot K.$$

Then we have

$$P(S') \ge K \cdot P'(S') \ge K \cdot P'(O) \ge P(O) - nK = OPT - \varepsilon \cdot P \ge (1 - \varepsilon)OPT$$

This finishes the proof. □

Discussions: Variations of Knapsack Problem

There are many variations of the knapsack problem that have arisen from the vast number of applications of the basic problem.

- Basic knapsack: n items, each with benefit b_i and weight w_i, and a knapsack with weight bound W.
- Unbounded knapsack problem: For each item a_i, it can be selected unlimited times, i.e., we do not put any upper bounds on the number of times an item may be selected.
- Bounded knapsack problem: For each item a_i, it can only be selected by at most k_i times in the final solution, i.e., there is an upper bound that an item may be selected.
- Multidimensional knapsack problem There are more than one constraints (for example, both a volume limit and a weight limit). This problem has 0-1, bounded, and unbounded etc. variants.