Web Algorithms

Eng. Fabio Persia, PhD

Design of dynamic programming algorithms.

1. Provide the recursive decomposition of the subproblems

1. Compute the subsolutions bottom-up, that is starting from the subproblems of smallest size
    1. use a table to store the results of subproblems
    2. avoid computation of the same solutions exploiting the table

2. Combine le solutions of already solved subproblems to construct the ones of subproblems of bigger size, till solving the original problem

Complexity dyn. progr. algorithms

- Let us consider the solutions table:

Subproblems parameters

Subproblems size

	p_1	p_2	p_3	…	p_k
1					
2					
3					
…					
n					

Table size =
Number of subproblems =
$n \cdot k$

Subsolutions combination

- Complexity:
  - (table size) x (time of combining subsolutions)
  - time of combining subsolutions always trivially polynomial
  - polynomial if table of polynomial size, that is only if polynomial number of different subproblems

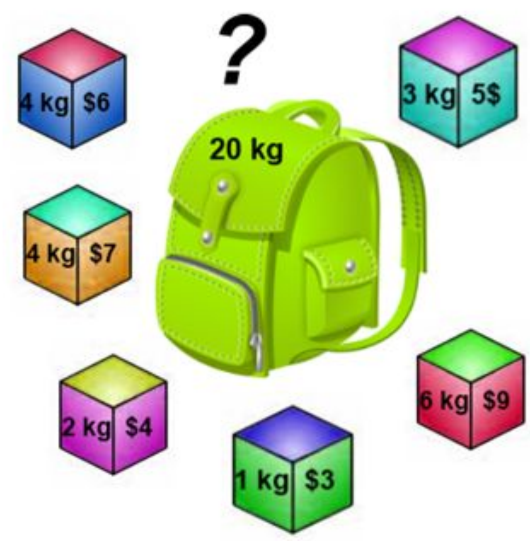
MAX 0-1 Knapsack

- INPUT: Finite set of objects O, an integral profit p_i and an integral weight w_i for every $o_i \in O$, a positive integer b
- SOLUTION: A subset of objects $Q \subseteq O$ such that $\sum_{o_i \in O} w_i \leq b$
- MEASURE: Total profit of the chosen objects, that is $\sum_{o_i \in Q} p_i$

Without loss of generality in the sequel we will always assume that $w_i \leq b$ and $p_i > 0$ for every object $o_i \in O$

Brute Force Algorithm

- Simple algorithm enumerating all the possible 2^n subsets of the n elements
- It chooses the best combination (best profit when satisfying the weight constraint)
- The Dynamic-Programming algorithm usually performs much better



Designing the algorithm …

Def. $OPT(i, w)$ = max profit subset of objects 1, ..., i with weight limit w

FACT: $OPT(n, b)$ = optimal solution initial problem

The following alternatives can occur for *OPT*:

- Case 1. *OPT* does not select object i
  - *OPT* selects best of { 1, 2, ..., $i - 1$ } using weight limit w

- Case 2. *OPT* selects object i
  - *OPT* selects best of { 1, 2, ..., $i - 1$ } using weight limit $w - w_i$

Designing the algorithm …

- Let us assume OPT(k,w) to be the optimal solution for the items {o1, …, ok}
- REMARK: The optimal solution OPT(k+1,w) could not correspond to OPT(k,w)
- Also because OPT(k+1,w) could not be a superset of OPT(k,w)

Example

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

- **The maximum weight of the knapsack is 20**

- The best solution for {I0, I1, I2} is {I0, I1, I2}
- But the best solution for {I0, I1, I2, I3} is {I0, I2, I3}
- In this example, the optimal solution exploits the partial solution {I0, I2}, which is the optimal solution of {I0, I1, I2}, when the weight of the knapsack is 12

We can thus provide the following recursive definition for *OPT:*

- $OPT(i,w)$ =
  - Empty set if $i=0$
  - $OPT(i-1,w)$ if $w_i > w$
  - Best choice between $OPT(i-1,w)$ and $OPT(i-1,w-w_i) \cup \{ o_i \}$ otherwise

In terms of measure $m(i,w)$ of the optimal solution $OPT(i,w)$

- $m(i,w)$ =
  - 0 if $i=0$
  - $m(i-1,w)$ if $w_i > w$
  - $Max \{ m(i-1,w) , m(i-1,w-w_i) + p_i \}$ otherwise

Clearly, $m^*=m(n,b)$

As a result, this means that the best subset of k objects with weight constraint w is (mutual exclusion):

- The best subset of (k-1) objects with total weight w
- The best subset of (k-1) objects with total weight $w-w_k$, plus the contribution (i.e., the weight) of the k-th object

Thus, as regards the following recursive formula

- $OPT(i,w)$ =
  - Empty set if $i=0$
  - $OPT(i-1,w)$ if $w_i > w$
  - Best choice between $OPT(i-1,w)$ and $OPT(i-1,w-w_i) \cup \{ o_i \}$ otherwise

  - the k-th object cannot be part of the solution (since just its weight is so big that the object itself does not fit in the knapsack)
- otherwise, we choose the best solution between
  - the solution including the new object
  - the best solution which does not include the new object

Algorithm Progr-Dyn-Knapsack

```
Begin
    For w=0 to b do
        M[0,w]=0

    For i=1 to n do
        For w=0 to b do
            if ( w_i>w)  M[i,w] = M[i-1,w]
            else         M[i,w] = max {M[i-1,w], M[i−1,w−w_i] + p_i}

    Return M[n,b]
End
```

Exercise: modify the pseudo-code in order to return the optimal subset of objects, and not just its measure (in the next slides)

Knapsack 0-1: example

- Let us consider the **following instance** of the problem:
- n = 4 (# of objects)
- b = W = 5 (maximum weight)
- Objects (weight, profit)
- (2,3), (3,4), (4,5), (5,6)
- [The object 1 weighs 2 and has profit 3]

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

Initialization (base case scenario):

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

- In the next slides, the blue color shows the instruction currently being executed
- The blue arrow in the table shows the cell of the matrix which is exploited to compute the new value

Example

Knapsack 0-: esempio

elementi:

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					

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

if $w_i <= w$ // oggetto i può essere inserito nella soluzione
    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-1,w]** // $w_i > w$ //massimo: da stessa colonna

Example

Knapsack 0-1: esempio

elementi:

$\boxed{1: (2,3)}$

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			
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 2}$

$w-w_i = 0$

if $w_i <= w$ // oggetto i può essere inserito nella soluzione

  if $v_i + B[i-1,w-w_i] > B[i-1,w]$

    $B[i,w] = v_i + B[i-1,w- w_i]$ // massimo: da B[0,0]

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Example

Knapsack 0-1: esempio

elementi:

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					

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

if $w_i <= w$ // oggetto i può essere inserito nella soluzione
    if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
        $B[i,w] = v_i + B[i\text{-}1,w\text{-} w_i]$ // massimo: da B[0,1]
    else
        $B[i,w] = B[i\text{-}1,w]$
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Example

Knapsack 0-1: esempio

elementi:
- 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					

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

```
if  w_i <= w   // oggetto i può essere inserito nella soluzione
        if v_i + B[i-1,w-w_i] > B[i-1,w]
            B[i,w] = v_i + B[i-1,w- w_i] // massimo: da B[0,2]
        else
            B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w
```

Example

Knapsack 0-1: esempio

elementi:
$\boxed{1: (2,3)}$
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	3
2	0					
3	0					
4	0					

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

if $w_i <= w$ // oggetto i può essere inserito nella soluzione
    if $v_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = v_i + B[i-1,w- w_i]$ // massimo: da B[0,3]
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Example

Knapsack 0-1: esempio

elementi:
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					

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

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 **$B[i,w] = B[i-1,w]$** // $w_i > w$

Example

Knapsack 0-1: esempio

elementi:
1: (2,3)
$\boxed{\text{2: (3,4)}}$
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					

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

if $w_i <= w$ // oggetto i può essere inserito nella soluzione
    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 **$B[i,w] = B[i-1,w]$** // $w_i > w$

Example

Knapsack 0-1: esempio

elementi:
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		
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 3}$
$w\text{-}w_i = 0$

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 $B[i,w] = B[i-1,w]$ // $w_i > w$

Example

Knapsack 0-1: esempio

elementi:
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					

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

if $w_i <= w$ //oggetto i può essere inserito nella soluzione
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 $B[i,w] = B[i-1,w]$ // $w_i > w$

Example

Knapsack 0-1: esempio

elementi:
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					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 5}$
$w\text{-}w_i = 2$

if w_i <= w // oggetto i può essere inserito nella soluzione

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 $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Example

Knapsack 0-1: esempio

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

$i = 3$
$v_i = 5$
$w_i = 4$
$\mathbf{w = 1..3}$
$w - w_i = -3..-1$

if $w_i <= w$ // oggetto i può essere inserito nella soluzione
  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 $B[i,w] = B[i-1,w]$ // $w_i > w$

Example

Knapsack 0-1: esempio

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

$i = 3$
$v_i = 5$
$w_i = 4$
$\mathbf{w = 4}$
$w\text{-}w_i = 0$

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 $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Example

Knapsack 0-1: esempio

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

$i = 3$
$v_i = 5$
$w_i = 4$
$\mathbf{w = 5}$
$w\text{-}w_i = 1$

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 $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Example

Knapsack 0-1: esempio

elementi:
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	

$i = 4$
$v_i = 6$
$w_i = 5$
$\mathbf{w = 1..4}$
$w-w_i = -4..-1$

if $w_i \le w$ //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]** // $w_i > w$

Example

Knapsack 0-1: esempio

elementi:

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

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

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

Example

Knapsack 0-1: esempio

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

elementi:

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

DONE! The maximum profit possible that can be inserted into the knapsack is **7**

KNAPSACK 0-1 Algorithm

- The algorithm finds the maximum value which can be inserted into the knapsack
- This value is memorized in B[n,W] at the end
- In order to discover the objects which have been inserted in the optimal solution, we need to come back to the table
  - We need to store somehow each added object

KNAPSACK 0-1 Algorithm

Knapsack 0-1 Algoritmo:
trovare gli elementi nella soluzione ottima

- Sia $i = n$ e $k = W$

  if $B[i, k] \neq B[i-1, k]$ then

      marca oggetto i "dentro lo zaino"

      $i = i-1$, $k = k-w_i$

  else

      $i = i-1$ // Assumi che oggetto i^{th} non sia nello zaino

      // Potrebbbe essere inserito nello zaino

      // in una soluzione ottima?

KNAPSACK 0-1 Algorithm

Knapsack 0-1: costruire la soluzione

elementi:
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 $B[i, k] \neq B[i-1, k]$ then
        marca elemento i nello zaino
        $i = i-1,\ k = k-w_i$
    else
        $i = i-1$

Oggetto 4 NON è nello zaino

KNAPSACK 0-1 Algorithm

Knapsack 0-1: costruire la soluzione

elementi:
- 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
```

Oggetto 3 NON è nello zaino

KNAPSACK 0-1 Algorithm

Knapsack 0-1: costruire la soluzione

elementi:
- 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

$i = 2$
$k = 5$
$v_i = 4$
$w_i = 3$
$\mathbf{B[i,k] = 7}$
$B[i-1,k] = 3$
$k - w_i = 2$

i = n , k = W
while i, k > 0
    if *B[i, k] ≠ B[i-1, k]* then
        marca elemento i nello zaino
    $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Oggetto 2 È nello zaino

KNAPSACK 0-1 Algorithm

Knapsack 0-1: costruire la soluzione

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

elementi:

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

Knapsack:

Item 2
Item 1

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
$\mathbf{B[i,k] = 3}$
$B[i\text{-}1,k] = 0$
$k - w_i = 0$

$i = n$, $k = W$
while $i, k > 0$
  if $B[i, k] \neq B[i\text{-}1, k]$ then
    marca elemento i nello zaino
    $i = i\text{-}1,\ k = k\text{-}w_i$
  else
    $i = i\text{-}1$

Oggetto 1
È nello
zaino

KNAPSACK 0-1 Algorithm

Knapsack 0-1: costruire la soluzione

elementi:

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

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
$\mathbf{B[i,k] = 3}$
$B[i-1,k] = 0$
$k - w_i = 0$

DONE! (k=0)

The optimal solution contains items 1 and 2

Remark: besides modifying the pseudo-code, the optimal subset of objects can be easily derived from the table *M* of the optimal subsolution measures (typical in dyn. progr. algorithms)

i	v	w
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w)=\begin{cases}0 & \text{if } i=0\\ OPT(i-1,w) & \text{if } w_i>w\\ \max\{OPT(i-1,w),\; v_i+OPT(i-1,w-w_i)\} & \text{otherwise}\end{cases}$$

weight limit w

subset of items 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

Time complexity

Theorem. The time complexity of *Progr-Dyn-Knapsack* is $O(n \cdot b)$

Proof.

- The algorithm takes $O(1)$ for each table entry
- There are $O(n \cdot b)$ table entries
- After computing values, we can trace back to find the optimal solution: take item o_i in $OPT(i,w)$ iff $M[i-1,w] < M[i,w]$ □

Question: is the algorithm polynomial?

Hint: consider the case $b=2^n$

Answer: in order to be polynomial, the complexity should be polynomial in the logarithm of the values coded in the input instance, that is with respect to *log b!!!*

This complexity called is pseudo-polynomial, that is polynomial in the input size and values, but not only in the input size

Dual approach …

Def. $OPT(i, p)$ = min weight subset of objects 1, ..., i with profit **at least** p

QUESTION: which subproblem corresponds to the optimal solution?

The following alternatives can occur for $OPT(i, p)$:

- Case 1. OPT does not select object i
  - OPT selects best of { 1, 2, ..., i – 1 } using profit limit p

- Case 2. OPT selects object i
  - OPT selects best of { 1, 2, ..., i – 1 } using profit limit $p - p_i$

We can thus provide the following recursive definition for *OPT:*

- $OPT(i,p)$ =
  - *Undef* if $i=0$
  - Best choice between $OPT(i-1,p)$ and $\{ o_i \}$ if $p_i \geq p$
  - Best choice between $OPT(i-1,p)$ and $OPT(i-1,p-p_i) \cup \{ o_i \}$ otherwise (*Undef* if both *Undef*)

In terms of weight v(i,p) of the optimal solution $OPT(i,p)$

- $v(i,p)$ =
  - ∞ if $i=0$
  - $Min \{ v(i-1,p) , w_i \}$ if $p_i \geq p$
  - $Min \{ v(i-1,p) , v(i-1,p-p_i) + w_i \}$ otherwise

- *Algorithm Progr-Dyn-Knapsack-Dual*

Begin

For p=1 to P do

V[0,p]=∞

For i=1 to n do

For p=1 to P do

if $(p_i \geq p)$ $V[i,p] = \min\{V[i-1,p],\ w_i\}$

else $V[i,p] = \min\{V[i-1,p],\ V[i-1,p-p_i]+w_i\}$

Return max p such that V[n,p]≤b

End

Problem: how should we choose *P*?

Answer: large enough to include optimum, that is any upper bound to m^*, i.e. $P \geq m^*$.

Choice: $P = n\, p_{max} \geq$ $\geq m^*$ $(p_{max} = \max p_j)$

$$\sum_{i=1}^{n} p_i$$

Time complexity

Theorem. The complexity of *Progr-Dyn-Knapsack-Dual* is $O(n^2\, p_{max})$

Proof.

- The algorithm takes $O(1)$ for each table entry
- There are $O(n \cdot P) = O(n^2\, p_{max})$ table entries
- After computing values, we can trace back to find the optimal solution: take item o_i in $OPT(i,p)$ iff $V[i-1,p] > V[i,p]$ □

Exercise: modify the pseudo-code in order to return the optimal subset of objects, and not just the measure