Contents

0/1 knapsack problem

Definition (0/1 knapsack problem)

Given a set S of n items, such that each item i has a positive benefit v_i and a positive weight w_i, the goal is to find the maximum-benefit subset that does not exceed a given weight W.

0/1 knapsack problem

Definition (0/1 knapsack problem)

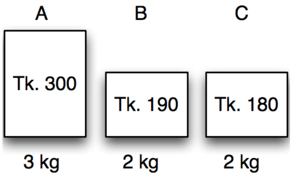
Given a set S of n items, such that each item i has a positive benefit v_i and a positive weight w_i, the goal is to find the maximum-benefit subset that does not exceed a given weight W. Formally, we wish to determine a subset of S that maximizes $\sum_{i \in S} v_i$, subject to $\sum_{i \in S} w_i \leq W$.

 Licensed under CC BY-NC-SA

0/1 knapsack problem

Definition (0/1 knapsack problem)

Given a set S of n items, such that each item i has a positive benefit v_i and a positive weight w_i, the goal is to find the maximum-benefit subset that does not exceed a given weight W. Formally, we wish to determine a subset of S that maximizes $\sum_{i \in S} v_i$, subject to $\sum_{i \in S} w_i \leq W$.

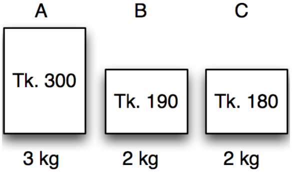


Maximum weight: $\boxed{W = 4 \text{ kg}}$

0/1 knapsack problem

Definition (0/1 knapsack problem)

Given a set S of n items, such that each item i has a positive benefit v_i and a positive weight w_i, the goal is to find the maximum-benefit subset that does not exceed a given weight W. Formally, we wish to determine a subset of S that maximizes $\sum_{i \in S} v_i$, subject to $\sum_{i \in S} w_i \leq W$.



Maximum weight: $\boxed{W = 4 \text{ kg}}$

Optimal solution: items B and C

Benefit: $\boxed{370}$

Licensed under

Developing a recursive solution

- Let S be an instance of a $0/1$ Knapsack problem, and ϑ be an optimal solution (even if we have no idea what it is yet).

Developing a recursive solution

- Let S be an instance of a $0/1$ Knapsack problem, and ϑ be an optimal solution (even if we have no idea what it is yet).
- Note that the presence of an item i in ϑ does not preclude any other item $j \neq i$ in ϑ.

 Licensed under CC BY-NC-SA

Developing a recursive solution

- Let S be an instance of a $0/1$ Knapsack problem, and ϑ be an optimal solution (even if we have no idea what it is yet).
- Note that the presence of an item i in ϑ does not preclude any other item $j \neq i$ in ϑ.
- If item n weighs more than the maximum allowed weight, it will not be in ϑ.

Developing a recursive solution

- Let S be an instance of a 0/1 Knapsack problem, and ϑ be an optimal solution (even if we have no idea what it is yet).
- Note that the presence of an item i in ϑ does not preclude any other item $j \neq i$ in ϑ.
- If item n weighs more than the maximum allowed weight, it will not be in ϑ.
- Otherwise, all we can say about ϑ is the following: item n (the last one) either belongs to ϑ, or it doesn't.

Developing a recursive solution

- Let S be an instance of a 0/1 Knapsack problem, and ϑ be an optimal solution (even if we have no idea what it is yet).
- Note that the presence of an item i in ϑ does not preclude any other item $j \neq i$ in ϑ.
- If item n weighs more than the maximum allowed weight, it will not be in ϑ.
- Otherwise, all we can say about ϑ is the following: item n (the last one) either belongs to ϑ, or it doesn't.

  If $n \in \vartheta$ Then the optimal solution contains n, plus an optimal solution for the other $n - 1$ items, but with a reduced maximum weight of $W - w_n$.

Developing a recursive solution

- Let S be an instance of a $0/1$ Knapsack problem, and ϑ be an optimal solution (even if we have no idea what it is yet).
- Note that the presence of an item i in ϑ does not preclude any other item $j \neq i$ in ϑ.
- If item n weighs more than the maximum allowed weight, it will not be in ϑ.
- Otherwise, all we can say about ϑ is the following: item n (the last one) either belongs to ϑ, or it doesn't.
  - If $n \in \vartheta$ Then the optimal solution contains n, plus an optimal solution for the other $n-1$ items, but with a reduced maximum weight of $W - w_n$.
  - If $n \notin \vartheta$ Then ϑ simply contains an optimal solution for the first $n-1$ items, with the maximum allowed weight W remaining unchanged.

Developing a recursive solution

- Let S be an instance of a 0/1 Knapsack problem, and ϑ be an optimal solution (even if we have no idea what it is yet).
- Note that the presence of an item i in ϑ does not preclude any other item $j \neq i$ in ϑ.
- If item n weighs more than the maximum allowed weight, it will not be in ϑ.
- Otherwise, all we can say about ϑ is the following: item n (the last one) either belongs to ϑ, or it doesn't.

  If $n \in \vartheta$ Then the optimal solution contains n, plus an optimal solution for the other $n-1$ items, but with a reduced maximum weight of $W - w_n$.

  If $n \notin \vartheta$ Then ϑ simply contains an optimal solution for the first $n-1$ items, with the maximum allowed weight W remaining unchanged.
- We have two parameters for each subproblem – the items S, and the maximum allowed weight W.

 Licensed under CC BY-NC-SA

Developing a recursive solution (continued)

- $w_n > W \implies n \notin \vartheta$.
  - $\vartheta(n, W) = \vartheta(n-1, W)$

Licensed under CC BY-NC-SA

Developing a recursive solution (continued)

- $w_n > W \implies n \notin \vartheta$.
  $\triangleright$ $\vartheta(n, W) = \vartheta(n-1, W)$
- Otherwise, n is either $\in \vartheta$ or $\notin \vartheta$.
  - If $n \in \vartheta$, then $\vartheta(n, W)$ is an optimal solution to the subproblem for items $\{1, 2, \ldots, n\}$:
    $\triangleright$ $\vartheta(n, W) = v_n + \vartheta(n-1, W - w_n)$

Licensed under CC BY-NC-SA

Developing a recursive solution (continued)

- $w_n > W \implies n \notin \vartheta$.
  $\triangleright\ \vartheta(n, W) = \vartheta(n-1, W)$
- Otherwise, n is either $\in \vartheta$ or $\notin \vartheta$.
  - If $n \in \vartheta$, then $\vartheta(n, W)$ is an optimal solution to the subproblem for items $\{1, 2, \ldots, n\}$:
    $\triangleright\ \vartheta(n, W) = v_n + \vartheta(n-1, W - w_n)$
  - If $n \notin \vartheta$, then $\vartheta(n, W)$ simply contains an optimal solution to the subproblem consisting of the intervals $\{1, 2, \ldots, n-1\}$:
    $\triangleright\ \vartheta(n, W) = \vartheta(n-1, W)$

Developing a recursive solution (continued)

- $w_n > W \implies n \notin \vartheta$.
  $\triangleright\ \vartheta(n, W) = \vartheta(n-1, W)$
- Otherwise, n is either $\in \vartheta$ or $\notin \vartheta$.
  - If $n \in \vartheta$, then $\vartheta(n, W)$ is an optimal solution to the subproblem for items $\{1, 2, \ldots, n\}$:
    $\triangleright\ \vartheta(n, W) = v_n + \vartheta(n-1, W - w_n)$
  - If $n \notin \vartheta$, then $\vartheta(n, W)$ simply contains an optimal solution to the subproblem consisting of the intervals $\{1, 2, \ldots, n-1\}$:
    $\triangleright\ \vartheta(n, W) = \vartheta(n-1, W)$
  - Since an optimal solution must maximize the sum of the weights in the intervals it contains, we accept the larger of the two.

Developing a recursive solution (continued)

- $w_n > W \implies n \notin \vartheta$.
  $\triangleright$ $\vartheta(n, W) = \vartheta(n-1, W)$
- Otherwise, n is either $\in \vartheta$ or $\notin \vartheta$.
  - If $n \in \vartheta$, then $\vartheta(n, W)$ is an optimal solution to the subproblem for items $\{1, 2, \ldots, n\}$:
    $\triangleright$ $\vartheta(n, W) = v_n + \vartheta(n-1, W - w_n)$
  - If $n \notin \vartheta$, then $\vartheta(n, W)$ simply contains an optimal solution to the subproblem consisting of the intervals $\{1, 2, \ldots, n-1\}$:
    $\triangleright$ $\vartheta(n, W) = \vartheta(n-1, W)$
  - Since an optimal solution must maximize the sum of the weights in the intervals it contains, we accept the larger of the two.
    $\triangleright$ $\boxed{\vartheta(n, W) = \textsc{max}(v_n + \vartheta(n-1, W - w_n), \vartheta(n-1, W))}$

Developing a recursive solution (continued)

Recursive algorithm for an optimal value

If $OPT(j, w)$ is an optimal solution to the subproblem for items $\{1, 2, \ldots, j\}$, for any $j \in \{1, 2, \ldots, n\}$, and with a maximum allowed weight of w, then:

$$OPT(j, w) = \begin{cases} OPT(j-1, w) & \text{if } w_j > w, \\ \textsc{max}(v_j + OPT(j-1, w - w_j), & \\ \qquad OPT(j-1, w)) & \text{otherwise.} \end{cases}$$

Licensed under CC BY-NC-SA

Developing a recursive solution (continued)

Recursive algorithm for an optimal value

If $OPT(j, w)$ is an optimal solution to the subproblem for items $\{1, 2, \ldots, j\}$, for any $j \in \{1, 2, \ldots, n\}$, and with a maximum allowed weight of w, then:

$$\boxed{OPT(j, w) = \begin{cases} OPT(j-1, w) & \text{if } w_j > w, \\ \text{MAX}(v_j + OPT(j-1, w - w_j), \\ \quad OPT(j-1, w)) & \text{otherwise.} \end{cases}}$$

Extracting the items in an optimal solution

The item j is in an optimal solution $OPT(j, w)$ **if and only if** the first of the two options is larger than the second.

$$\boxed{v_j + OPT(j-1, w - w_j) \geq OPT(j-1, w)}$$

A recursive algorithm

$\textsc{Knapsack}(j, w)$

if $j = 0$ or $w = 0$
  then return 0
elseif $w_j > w$
  then return $\textsc{Knapsack}(j-1, w)$
else return $\textsc{max}(v_j + \textsc{Knapsack}(j-1, w - w_j),$
  $\textsc{Knapsack}(j-1, w))$

A recursive algorithm

$\textsc{Knapsack}(j, w)$

```
if j = 0 or w = 0
    then return 0
elseif w_j > w
    then return KNAPSACK(j − 1, w))
else return MAX(v_j + KNAPSACK(j − 1, w − w_j),
                KNAPSACK(j − 1, w))
```

- The initial call is $\textsc{Knapsack}(n, W)$.

 Licensed under

A recursive algorithm

$\text{KNAPSACK}(j, w)$

if $j = 0$ or $w = 0$
    then return 0
elseif $w_j > w$
    then return $\text{KNAPSACK}(j - 1, w)$
else return $\text{MAX}(v_j + \text{KNAPSACK}(j - 1, w - w_j),$
$\qquad\qquad\text{KNAPSACK}(j - 1, w))$

- The initial call is $\text{KNAPSACK}(n, W)$.
- The tree grows very rapidly, leading to exponential running time.

Licensed under

A recursive algorithm

$\textsc{Knapsack}(j, w)$

if $j = 0$ or $w = 0$
  then return 0
elseif $w_j > w$
  then return $\textsc{Knapsack}(j-1, w)$
else return $\textsc{max}(v_j + \textsc{Knapsack}(j-1, w - w_j),$
    $\textsc{Knapsack}(j-1, w))$

- The initial call is $\textsc{Knapsack}(n, W)$.
- The tree grows very rapidly, leading to exponential running time.
- There are many overlapping subproblems, so the obvious choice is to memoize the recursion.

Licensed under

Memoizing the recursion

M-KNAPSACK(j, w)

if $j = 0$ or $w = 0$
  then return 0
elseif $M[j, w]$ is empty
  then $M[j, w] \leftarrow \textsc{max}(v_j + \text{M-Knapsack}(j - 1, w - w_j),$
    $\text{M-Knapsack}(j - 1, w))$
return $M[j, w]$

Licensed under

Memoizing the recursion

$\textsc{M-Knapsack}(j, w)$

if $j = 0$ or $w = 0$
    then return 0
elseif $M[j, w]$ is empty
    then $M[j, w] \leftarrow \textsc{max}(v_j + \textsc{M-Knapsack}(j - 1, w - w_j),$
$\qquad\qquad\qquad\qquad\textsc{M-Knapsack}(j - 1, w))$
return $M[j, w]$

- Each entry in $M[j, w]$ gets filled in only once at $\Theta(1)$ time, and there are $n + 1 \times W + 1$ entries, so $\textsc{M-Knapsack}(n, W)$ takes $\boxed{\Theta(nW)}$ time.

Licensed under CC BY-NC-SA

Memoizing the recursion

$\textsc{M-Knapsack}(j, w)$

```
if j = 0 or w = 0
    then return 0
elseif M[j, w] is empty
    then M[j, w] ← MAX(v_j + M-KNAPSACK(j − 1, w − w_j),
                                M-KNAPSACK(j − 1, w))
return M[j, w]
```

- Each entry in $M[j, w]$ gets filled in only once at $\Theta(1)$ time, and there are $n + 1 \times W + 1$ entries, so $\textsc{M-Knapsack}(n, W)$ takes $\boxed{\Theta(nW)}$ time.
- Is this a linear-time algorithm?

Licensed under CC BY-NC-SA

Memoizing the recursion

$\text{M-Knapsack}(j, w)$

```
if j = 0 or w = 0
    then return 0
elseif M[j, w] is empty
    then M[j, w] ← MAX(v_j + M-KNAPSACK(j − 1, w − w_j),
                       M-KNAPSACK(j − 1, w))
return M[j, w]
```

- Each entry in $M[j, w]$ gets filled in only once at $\Theta(1)$ time, and there are $n + 1 \times W + 1$ entries, so $\text{M-Knapsack}(n, W)$ takes $\boxed{\Theta(nW)}$ time.
- Is this a linear-time algorithm?
- This is an example of a pseudo-polynomial problem, since it depends on another parameter W that is independent of the problem size.

Licensed under

Developing a Dynamic Programming algorithm

```
KNAPSACK(n, W)
for i ← 0 to n                ▷ no remaining capacity
      do M[i, 0] ← 0
for w ← 0 to W                ▷ no item to choose from
      do M[0, w] ← 0
for j ← 1 to n
      do for w ← 1 to W
              do if w_j > w
                   then M[j] = M[j − 1, w]
                   else  M[j, w] ← MAX(v_j + M[j − 1, w − w_j],
                                       M[j − 1, w])
return M[n, W]
```

 CC BY NC SA

0/1 Knapsack recursive algorithm in action

Given the following (from M. H. Alsuwaiyel, ex. 7.6):

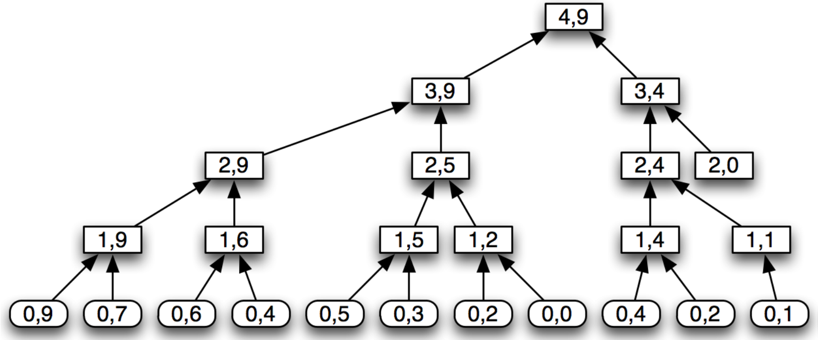
$$W = 9$$
$$w_i = \{2, 3, 4, 5\}$$
$$v_i = \{3, 4, 5, 7\}$$

Licensed under CC BY-NC-SA

0/1 Knapsack recursive algorithm in action

Given the following (from M. H. Alsuwaiyel, ex. 7.6):

$$W = 9$$
$$w_i = \{2, 3, 4, 5\}$$
$$v_i = \{3, 4, 5, 7\}$$



0/1 Knapsack DP algorithm in action

Given the following (from M. H. Alsuwaiyel, ex. 7.6):

$$W = 9$$
$$w_i = \{2, 3, 4, 5\}$$
$$v_i = \{3, 4, 5, 7\}$$

	0	1	2	3	4	5	6	7	8	9
4	-	-	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-	-	-
0	-	-	-	-	-	-	-	-	-	-

0/1 Knapsack DP algorithm in action

Given the following (from M. H. Alsuwaiyel, ex. 7.6):

$$W = 9$$
$$w_i = \{2, 3, 4, 5\}$$
$$v_i = \{3, 4, 5, 7\}$$

	0	1	2	3	4	5	6	7	8	9
4	0	0	3	4	5	7	8	10	11	12
3	0	0	3	4	4	7	8	9	9	12
2	0	0	3	4	4	7	7	7	7	7
1	0	0	3	3	3	3	3	3	3	3
0	0	0	0	0	0	0	0	0	0	0

Licensed under CC BY-NC-SA