

Course «Algorithms» · Lecture 7

The Knapsack Problem and Memory Functions

Textbook: A. Levitin, Section 8.2

Today’s goal

Dynamic programming with **two parameters**, and a top-down way to do it.

- the 0/1 knapsack problem and its recurrence
- a full worked example with traceback
- why $\Theta(nW)$ is only **pseudo-polynomial**
- memory functions (memoization)

The 0/1 knapsack problem

n items with weights w_i and values v_i, a knapsack of capacity *W*. Take the most valuable set that fits.

$$\max \sum_{i=1}^{n} v_i x_i \quad \text{subject to} \quad \sum_{i=1}^{n} w_i x_i \le W, \qquad x_i \in \{0, 1\}$$

- Each item is taken whole or not at all
- Brute force: 2^n subsets, hopeless at n ≈ 40
- NP-hard; used for budgets, cargo, ads, capital planning

Why not be greedy? W = 50

Item	Weight	Value	v / w
1	10	60	6
2	20	100	5
3	30	120	4

Greedy by ratio: items 1, 2
weight 30, value **160**

Optimum: items 2, 3
weight 50, value **220**

The subproblem has two parameters

F(i, j) = the best value from the **first i items** that fits into capacity **j**

- $0 \le i \le n$, $0 \le j \le W$
- We want F(n, W)
- Last decision: do we take item *i* or not?

The recurrence

Skip item i
best value F(i − 1, j)

Take item i (if j ≥ w_i)
v_i + F(i − 1, j − w_i)

$$F(i, j) = \max\left\{ F(i-1, j), \quad v_i + F(i-1,\ j - w_i) \right\}, \qquad j \geq w_i$$

$$F(i, j) = F(i-1, j), \qquad j < w_i$$

$$F(0, j) = 0, \qquad F(i, 0) = 0$$

Bottom-up algorithm

```
Knapsack(w[1..n], v[1..n], W)
  for j ← 0 to W do F[0, j] ← 0
  for i ← 1 to n do
    F[i, 0] ← 0
    for j ← 1 to W do
      if j ≥ w[i]
        F[i, j] ← max(F[i−1, j], v[i] + F[i−1, j−w[i]])
      else F[i, j] ← F[i−1, j]
  return F[n, W]
```

Row by row: each entry needs only the previous row.

Example: W = 5, four items

Item	Weight w_i	Value v_i
1	2	12
2	1	10
3	3	20
4	2	15

The filled table F(i, j)

i	w_i	v_i	j=0	1	2	3	4	5
0			0	0	0	0	0	0
1	2	12	0	0	12	12	12	12
2	1	10	0	10	12	22	22	22
3	3	20	0	10	12	22	30	32
4	2	15	0	10	15	25	30	37

Skip it or take it?

$$F(2,3) = \max\left\{F(1,3),\ \ 10 + F(1,2)\right\} = \max\left\{12,\ \ 22\right\} = 22$$

$$F(3,5) = \max\left\{F(2,5),\ \ 20 + F(2,2)\right\} = \max\left\{22,\ \ 32\right\} = 32$$

$$F(4,5) = \max\left\{F(3,5),\ \ 15 + F(3,3)\right\} = \max\left\{32,\ \ 37\right\} = 37$$

The best value we can carry: **F(4, 5) = 37**.

Which items give 37? Walk back

- F(4, 5) = 37 ≠ F(3, 5) = 32 → item 4 **taken**, capacity 5 − 2 = 3
- F(3, 3) = 22 = F(2, 3) → item 3 **not taken**
- F(2, 3) = 22 ≠ F(1, 3) = 12 → item 2 **taken**, capacity 3 − 1 = 2
- F(1, 2) = 12 ≠ F(0, 2) = 0 → item 1 **taken**, capacity 0

Optimal set {1, 2, 4}

$$\text{weight} = 2 + 1 + 2 = 5, \qquad \text{value} = 12 + 10 + 15 = 37$$

Compare an entry with the one above it: equal means “skipped”.

Efficiency and saving memory

- (n + 1)(W + 1) entries, each O(1): time **Θ(nW)**
- Space **Θ(nW)** for the whole table; traceback Θ(n)

Need only the value? Keep **one row** and go **downward**:

$$F[j] \leftarrow \max\left\{F[j],\ v_i + F[j - w_i]\right\}, \qquad j = W,\ W-1,\ \ldots,\ w_i$$

Space Θ(W). An upward loop would reuse an item: that solves the *unbounded* knapsack.

Is Θ(nW) polynomial?

No. W is written in binary, so it takes only about **$\log_2 W$ bits**.

$$W = 2^{\log_2 W}$$

- Polynomial in the **value** of W, exponential in its **length**
- W from 20 to 40 bits: the table grows about a million times
- Such algorithms are called **pseudo-polynomial**

This does not contradict NP-hardness: it is not polynomial in the input length.

Memory functions: top-down with a table

Follow the recurrence recursively, but **check the table first** and store every result.

```
MFKnapsack(i, j)
 // F[0, j] = F[i, 0] = 0; other entries = −1
 if F[i, j] < 0
  if j < w[i]
   value ← MFKnapsack(i − 1, j)
  else
   value ← max(MFKnapsack(i − 1, j),
     v[i] + MFKnapsack(i − 1, j − w[i]))
  F[i, j] ← value
 return F[i, j]
```

Same example, top-down

11/20 entries computed. The other nine, such as F(2, 1), F(3, 2), F(4, 1)…F(4, 4), are never needed for F(4, 5) = 37.

- Worst case still **Θ(nW)**
- Costs: recursion depth up to n, function-call overhead

Three ways to code one recurrence

	Plain recursion	Memory function	Bottom-up
Each subproblem once	no	yes	yes
Only needed subproblems	—	yes	no
Recursion overhead	yes	yes	no
Easy space saving	no	no	yes
Worst-case time	exponential	$\Theta(nW)$	$\Theta(nW)$

Regular dependencies: bottom-up. Sparse or irregular ones: memoization.

Variations worth knowing

Unbounded knapsack (any number of copies):

$$F(j) = \max_{i:\ w_i \le j} \left\{ v_i + F(j - w_i) \right\}$$

The same shape as change making from Lecture 6.

- **Subset sum, partition:** value = weight; also Θ(nW)
- **Several constraints:** one more table dimension each
- **Small values:** index the table by value, store the minimum weight

Summary

- Brute force 2^n; greedy by ratio can be badly wrong
- F(i, j): best value from the first i items with capacity j
- Table row by row: **Θ(nW)**; traceback gives the items
- Θ(nW) is **pseudo-polynomial**
- Memory function: top-down, each entry at most once; here 11 of 20

Review questions

1. What does F(i, j) mean, and why are two parameters needed?
2. Fill the table for W = 6 and items (3, 25), (2, 20), (1, 15), (4, 40), (5, 50). Find the optimal subset.
3. Why must the one-row version loop from W down to w_i?
4. Why is Θ(nW) not a polynomial bound?
5. When would you prefer memoization to the bottom-up table?