# LEARN CODING

ale66

github.com/ale66/learn-coding

# PRACTICE ALGORITHMICS

# FORMULATION, III

**Instance:**

a sequence of *n* `weight/value` pairs: $w_1, \ldots w_m$ and $v_1, \ldots v_n$

an integer C

**Solution:**

An allocation: for each bar says what quantity is taken: $\sigma : i \rightarrow [0..w_i]$

**Constraint**

The sum of all taken quantities must not exceed C: $\sum_{i=1}^{n} \sigma(i) \leq C$

# ALGORITHM

Think of a step-by-step process that, for any input combination, will take the most value out of the vault

Write it down in English with some Maths

Test it on some examples, e.g.

| Metal | Weight avail. | Total Val. |
|---|---:|---:|
| Gold | 3 | 162 |
| Palladium | 1 | 72 |
| Silver | 12 | 48 |

$C = 5$

# IDEA

- Consider the *unit* (per Kg) value: total value / available quantity
- sort the elements by descending unit value
- start from the top, take all there is, continue below, until capacity C is reached
- the last element, and only the last, may have to be cut to measure in order not to exceed C

This is the algorithm

- Consider the *unit* (per Kg) value: total value / available quantity

| Metal | Weight avail. | Total Val. | Unit val. |
|---|---:|---:|---:|
| Gold | 3 | 162 | 54 |
| Palladium | 1 | 72 | 72 |
| Silver | 12 | 48 | 4 |

- sort the elements by descending unit value

| Metal | Weight avail. | Total Val. | Unit val. | Pos. |
|---|---:|---:|---:|---:|
| Palladium | 1 | 72 | 72 | 1 |
| Gold | 3 | 162 | 54 | 2 |
| Silver | 12 | 48 | 48 | 3 |

- start from the top, take all there is, [...]

| Metal | Weight avail. | Total Val. | Unit val. | Pos. | Take |
|---|---|---|---|---|---|
| Palladium | 0 | 72 | 72 | 1 | 1 |
| Gold | 3 | 162 | 54 | 2 | |
| Silver | 12 | 48 | 4 | 3 | |

$$\sigma(Palladium) = 1$$

C = 5-1 = 4

- [...] continue below, [...]

| Metal | Weight avail. | Total Val. | Unit val. | Pos. | Take |
|---|---:|---:|---:|---:|---:|
| Palladium | 0 | 72 | 72 | 1 | 1 |
| Gold | 0 | 162 | 54 | 2 | 3 |
| Silver | 12 | 48 | 4 | 3 | |

$\sigma(Palladium) = 1 \, \sigma(Gold) = 3$

C = 4-3 = 1

- [...] until capacity C is reached

| Metal | Weight avail. | Total Val. | Unit val. | Pos. | Take |
|---|---|---|---|---|---|
| Palladium | 0 | 72 | 72 | 1 | 1 |
| Gold | 0 | 162 | 54 | 2 | 3 |
| Silver | 12 | 48 | 4 | 3 | |

$\sigma(Palladium) = 1\, \sigma(Gold) = 3$

C = 1

- the last element, and only the last, may have to be cut to measure in order not to exceed C

| Metal | Weight avail. | Total Val. | Unit val. | Pos. | Take |
|---|---|---|---|---|---|
| Palladium | 0 | 72 | 72 | 1 | 1 |
| Gold | 0 | 162 | 54 | 2 | 3 |
| Silver | 11 | 48 | 4 | 3 | 1 |

$\sigma(Palladium) = 1\,\sigma(Gold) = 3\,\sigma(Silver) = 1$

C = 1-1 = 0

Total value taken: $1 \times 72 + 3 \times 54 + 1 \times 4 \ = 72 + 162 + 4 = 238$ github.com/ale66/learn-coding

# OBERVATIONS

Theorem: our algorithm is *optimal*: it always finds the best solution

Computational cost: the costly step is sorting the table of metals

Fact: the number of basic computer steps depends on the number n of metals with law $K(n\log_2 n + n)$

- *K* depends of the details of the implementation, e.g., Python 3.12.6 on Win 11 and AMD Ryzen 9
- $n\log_2 n$ steps to sort the elements
- up to $n$ step to select the bars and decrease the residual capacity

This problem is generally scalable to the web

# INTRACTABLE PROBLEMS

Problems for which no scalable algorithm is known:

Their cost (no. of operations to complete) is expressed as $K2^n$: exponential

Only approximate solutions exist that can find a *good enough* solution with a low-growth cost function

# PROBLEM VARIATION

This time the Fourty thieves’ vault contains precious artesanal objects, e.g., gold watches.

Each watch is described by weight and value

For each object, **it’s either you take it or leave it.**

# KNAPSACK 0-1

**Instance:**

a sequence of *n* `weight/value` pairs: $w_1, \ldots w_m$ and $v_1, \ldots v_n$

an integer C

**Solution:**

An assignment: for each object says whether it's taken or not: $\sigma : i \rightarrow \{0, 1\}$

**Constraint**

The sum of all taken weights C: $\sum_{i=1}^{n} w_i \sigma(i) \leq C$

# SURPRISE WITH KNAPSACK 0-1

The algorithm seen above stops working: the computed solution can be very suboptimal

# KNAPSACK 0-1 IS INTRACTABLE

No algorithm is known that can solve it in acceptable times as n grows

The take-it-or-leave-it nature of the problem forces us to consider up to $2^n$ alternative solutions

this will require spending an exponential amount of operations before we could be sure that the solution at hand is the best

It is believed that no scalable algorithm will ever be found: the P vs. NP conjecture

# SORTING

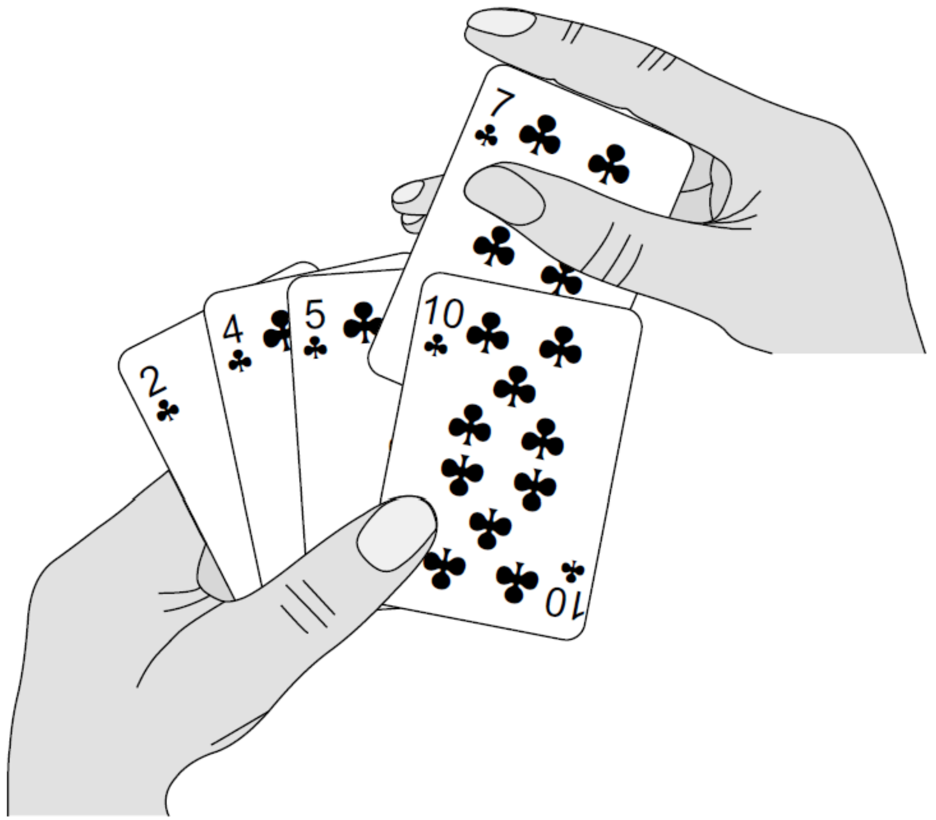


**Figure 2.1** Sorting a hand of cards using insertion sort.

# STATEMENT

**Instance:**

a sequence of *n* integers: $A = a_1, a_2 \ldots a_n$

**Solution:**

A permutation of the values $\pi : [1..n] \rightarrow [1..n]$

**Constraint**

values never decrease: $a_1 \leq a_2 \ldots a_n$

Let's assume that there are no repeated values

A = [11, 6, 8, 2, 22, 16]

sort(A) = [2, 6, 8, 11, 16, 22]

$$\pi(1) = 4$$

$$\pi(2) = 2$$

$$\pi(3) = 3$$

$$\pi(4) = 1$$

...

# GOOD NEWS ABOUT SORTING

Solvable within $Kn \log_2 n$ steps

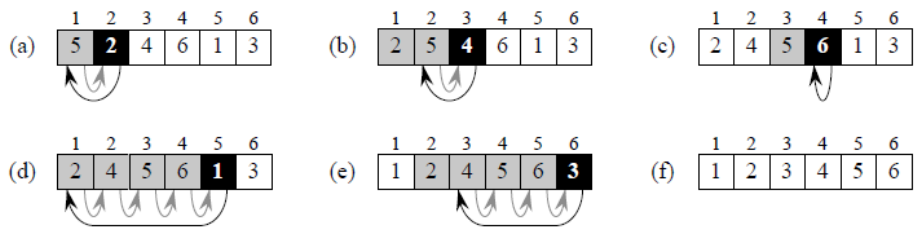


**Figure 2.2** The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

Modern algorithms are even quicker when A is already half-sorted

Python runs `powersort()` an optimised version of TimSort

```
a = [11, 6, 8, 2, 22, 16]

a.sort()
```