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:

Constraint

The sum of all taken quantities must not exceed 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	

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	

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	

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

C = 1-1 = 0

Total value taken:

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* depends of the details of the implementation, e.g., Python 3.12.6 on Win 11 and AMD Ryzen 9
- steps to sort the elements
- up to 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 : 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: and

an integer C

Solution:

An assignment: for each object says whether it’s taken or not:

Constraint

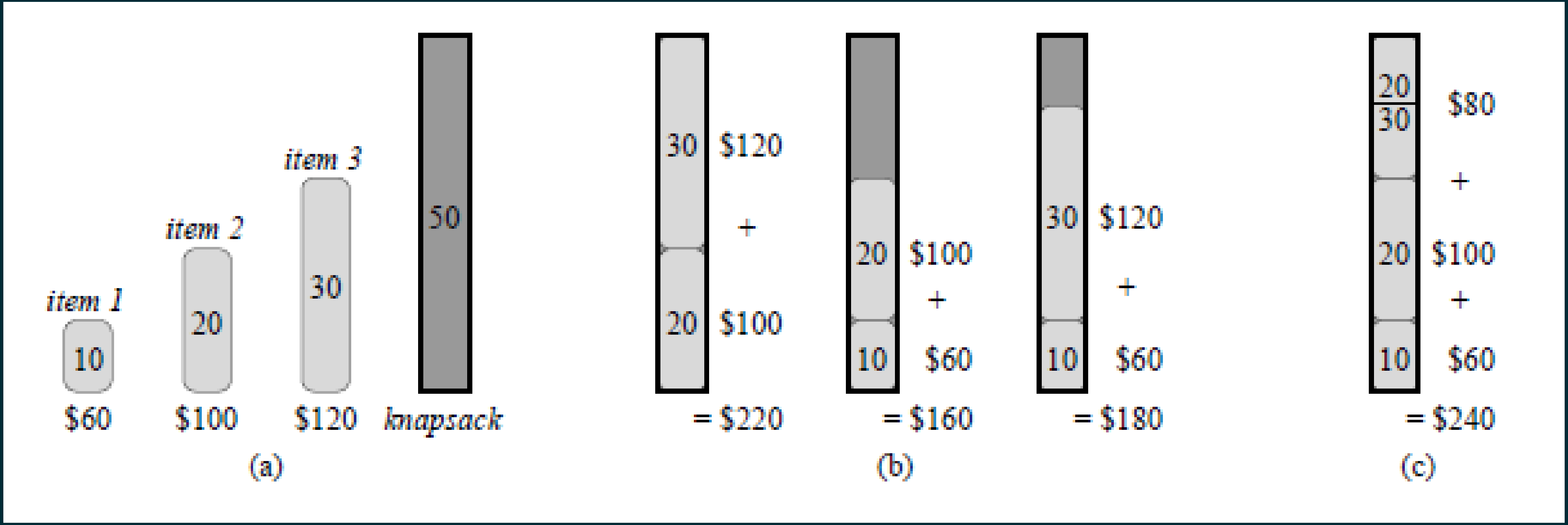
The sum of all taken weights C:

SURPRISE WITH KNAPSACK 0-1

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

Metal	Weight	Value
Ring	10	60
Earrings	20	100
Necklace	30	120

C = 50



The returend solution (1/1/0) is not even second-best

best solution is 0/1/1 and 1/0/1 is second-best

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 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

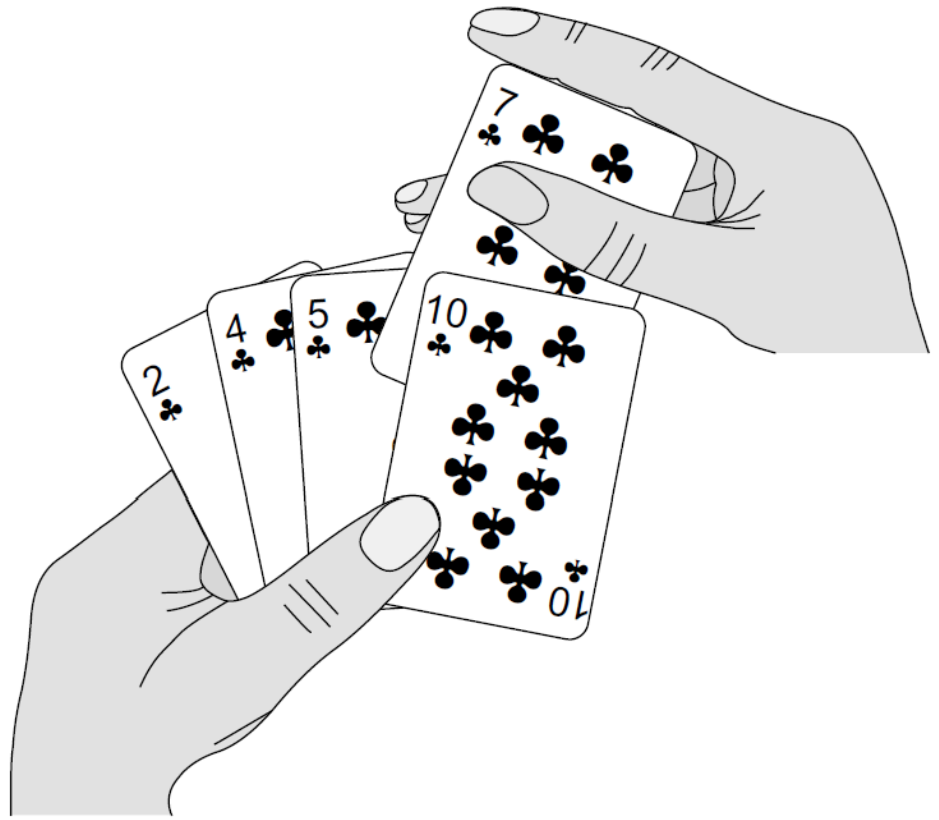


Figure 2.1 Sorting a hand of cards using insertion sort.

STATEMENT

Instance:

a sequence of *n* integers:

Solution:

A permutation of the values

Constraint

values never decrease:

Let’s assume that there are no repeated values

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

sort(A) = [?, ?, ?, ?, ?, ?]

…

Let's assume that there are no repeated values

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

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

...

GOOD NEWS ABOUT SORTING

Solvable within steps

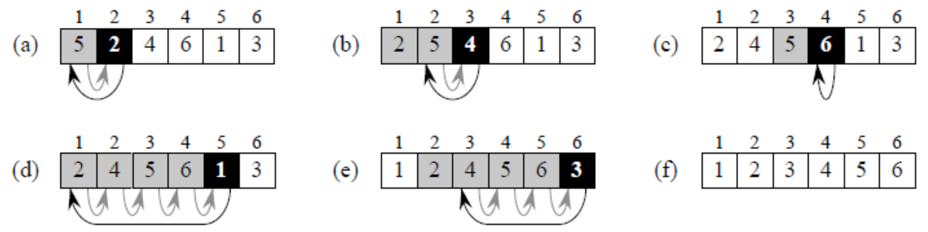


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle 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.

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()
```

SEARCHING

Istance:

- a collection of *n* integers
- an integer

Question: does k belong to the collection?

OBSERVATIONS

It can be generalised to data types that are ordered (strings have alphanumerical ordering)

There is a simple algorithm that will answer after at most comparisons

this is a very basic problem which is used elsewhere: it is important that the implementation is quick and well-tested

CASE STUDY: EGO NETWORKS

- while less than N profiles collected
  - generate random FB ids (a fixed-lenght, 32-digits integer)
  - test the random id: does it land on an open FB profile?
    - yes: expand the visit to the neighborood
    - no: go back to generating random ids

Cost: up to comparisons

SIMPLE SOLUTION

```python
def search(a, k):
  '''Vanilla search'''

  found = False
  n = len(a)

  for elem in a:
    if k == elem:
      found = True

  return found
```

EXERCISE

Apply `while` instead of `for` to stop operations as soon as the key value is found

ORDERED SEQUENCES

Special case: the input sequence A is already sorted, either in increasing or decreasing

a much more efficient algorithm is available

Binary_search, which is correct only for sorted sequences, will take *at most* comparisons before we stop

n	Vanilla	BS
1,000	1,000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30

```python
def bs(a, k):
  '''Binary search'''

  found = False
  n = len(a)

  # the boundaries of our search
  l = 0
  r = n

  mid = int((r-l)/2)
```

```python
  while (found == False) and (l<r):

    mid = int((r-l)/2)

    if k < a[mid]:
      r = mid
    elif k > a[mid]:
      l = mid
    else:
      found = True

  return found
```

github.com/ale66/learn-coding