Algorithms and Data Structures

A3. Sorting I: Selection and Insertion Sort

Gabriele Röger

University of Basel

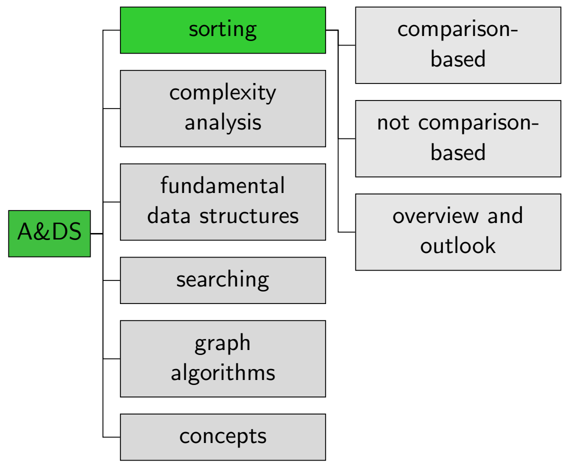
February 29, 2024

Algorithms and Data Structures

February 29, 2024 — A3. Sorting I: Selection and Insertion Sort

A3.1 Sorting

Content of the Course



Relevance

sorting data important for many applications, such as

- sorted presentation (e.g. on website)
  - products sorted by price, rating, ...
  - account transactions sorted by transaction date
- preprocessing for many efficient search algorithms
  - How quickly can you find a number in a (physical) telephone book? How quickly could you do so if the entries were not sorted?
- subroutine of many other algorithms
  - e.g. a program that renders layered graphical objects might sort them to determine where objects are covered by other objects

Journal "Computing in Science & Engineering" lists Quicksort as one of the 10 most important algorithms of the 20th century.

Sorting Problem

Sorting Problem

Input

- sequence of n elements $e_1, \dots, e_n$
- each element e_i has key $k_i = key(e_i)$
- partial order $\leq$ on the keys
  reflexive: $k \leq k$
  transitive: $k \leq k'$ and $k' \leq k'' \Rightarrow k \leq k''$
  antisymmetric: $k \leq k'$ and $k' \leq k \Rightarrow k = k'$

Output

- Sequence of the same elements sorted according to the ordering relation on its keys

Notation: also $e \leq e'$ for $key(e) \leq key(e')$

Sorting Problem: Examples

Example

Input: $\langle 3, 6, 2, 3, 1 \rangle$, $key(e) = e$, $\leq$ on the integers
Output: $\langle 1, 2, 3, 3, 6 \rangle$

Example

Input: list of all students of the Univ. of Basel,
$key(e) = \langle$place of residence of $e\rangle$, lexicographic order
Output: list of all students, sorted by their place of residence

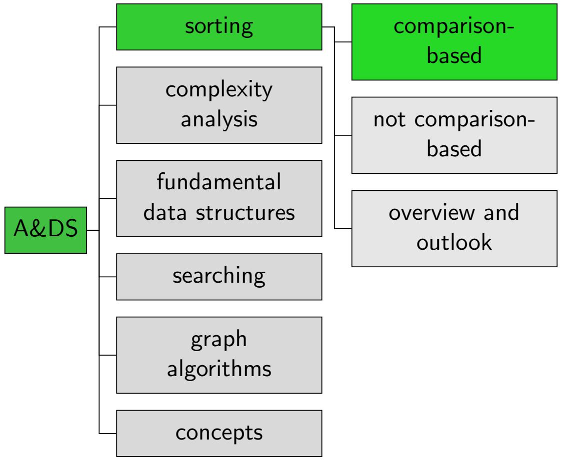
Is the output uniquely defined?

In this course: mostly integers, $key(e) = e$ and $\leq$ on integers

Interesting Properties of Sorting Algorithms

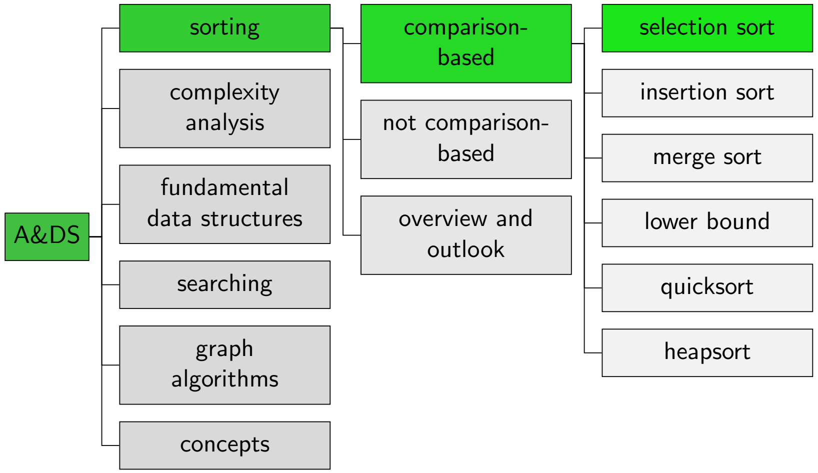
- running time: how many key comparisons and swaps of elements are executed? adaptive: algorithms faster if input already (partially) sorted
- space consumption: how much space is used in addition to the space occupied by the input sequence (explicitly or in call stack)?
  in-place: needs no additional storage beyond the input array and a constant amount of space
  (independent of the input size)
- stable: elements with the same value appear in the output sequence in the same order as they do in the input sequence
- comparison-based: uses only key comparisons and swaps of elements

Content of the Course

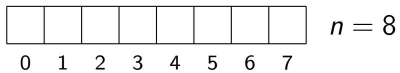


A3.2 Selection Sort

Content of the Course



Selection Sort: Informally



- identify smallest element at positions $0, \dots, n-1$ and swap it to position 0
- identify smallest element at positions $1, \dots, n-1$ and swap it to position 1
- ...
- identify smallest element at positions $n-2, n-1$ and swap it to position n - 2

Selection Sort: Example

3	7	2	9	7	1	4	5
1	7	2	9	7	3	4	5
1	2	7	9	7	3	4	5
1	2	3	9	7	7	4	5
1	2	3	4	7	7	9	5
1	2	3	4	5	7	9	7
1	2	3	4	5	7	9	7
1	2	3	4	5	7	7	9

Selection Sort: Algorithm

```python
def selection_sort(array):
    n = len(array)
    for i in range(n - 1):  # i = 0, ..., n-2
        # find index of minimum element at positions i, ..., n-1
        min_index = i
        for j in range(i + 1, n):  # j = i+1, ..., n-1
            if array[j] < array[min_index]:
                min_index = j
        # swap element at position i with minimum element
        array[i], array[min_index] = array[min_index], array[i]
```


Selection Sort: Example

i	min_ind.	0	1	2	3	4	5	6	7
		3	7	2	9	7	1	4	5
0	5	3	7	2	9	7	1	4	5
1	2	1	7	2	9	7	3	4	5
2	5	1	2	7	9	7	3	4	5
3	6	1	2	3	9	7	7	4	5
4	7	1	2	3	4	7	7	9	5
5	5	1	2	3	4	5	7	9	7
6	7	1	2	3	4	5	7	9	7
		1	2	3	4	5	7	7	9

looking for minimum among dark entries

red entry is found minimum

gray entries already sorted

Correctness

Correctness of an algorithm

An algorithm for a computational problem is correct if for every problem instance provided as input, it

- halts, i.e. it finishes its computation in finite time, and
- determines a correct solution to the problem instance.

Correctness of Selection Sort

- invariant: property that is true during the entire execution of the algorithm
- invariant 1: at the end of each iteration of the outer loop, all elements at positions $\leq i$ are sorted.
- Invariant 2: at the end of each iteration of the outer loop, none of the elements at positions $\leq i$ is (strictly) larger than an element at a position $> i$.
- correctness of invariants by (joint) induction
- after the last iteration, all elements except for the last one are in the correct order and the last one is not smaller than the second-last. $\rightarrow$ entire sequence sorted
- Termination: $n - 1$ iterations of outer loop, each with fewer than n iterations of inner loop $\rightarrow$ finite runtime

Properties of Selection Sort

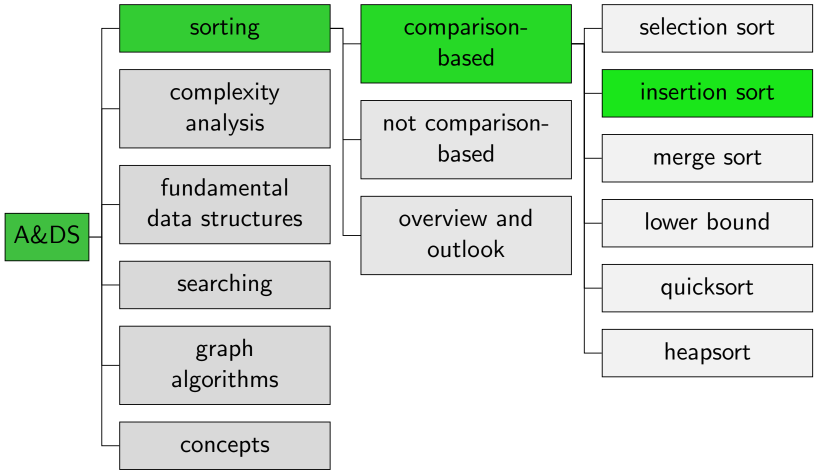
- in-place: additional storage does not depend on input size
- running time: does only depend on the size of the input (not adaptive)
  exact analysis: later chapter
- not stable: can swap the element at position i behind an element with an equal key, which will not be "repaired" later.

Jupyter Notebook

Jupyter notebook: `selection_sort.ipynb`

A3.3 Insertion Sort

Content of the Course



Insertion Sort: Informally

- similar to common method for sorting a hand of playing cards
- elements subsequently moved to correct position in the already sorted part of the sequence
- larger elements correspondingly moved to the right

Insertion Sort: Example

i	0	1	2	3	4	5	6	7
	3	7	2	9	7	1	4	5
1	3	7	2	9	7	1	4	5
2	2	3	7	9	7	1	4	5
3	2	3	7	9	7	1	4	5
4	2	3	7	7	9	1	4	5
5	1	2	3	7	7	9	4	5
6	1	2	3	4	7	7	9	5
7	1	2	3	4	5	7	7	9

gray entries not moved

red entry moved into sorted range

black entries moved one position to the right

Insertion Sort: Algorithm

```python
def insertion_sort(array):
    n = len(array)
    for i in range(1, n):  # i = 1, ..., n - 1
        # move array[i] to the left until it is
        # at the correct position.
        j = i
        while j > 0 and array[j - 1] > array[j]:
            # not yet at final position.
            # swap array[j] and array[j-1]
            array[j], array[j-1] = array[j-1], array[j]
            j -= 1
```

Insertion Sort: Algorithm (Slightly Faster)

previous variant: most assignments to `array[j-1]` unnecessary

```python
def insertion_sort(array):
    for i in range(1, len(array)):
        val = array[i]
        j = i
        while j > 0 and array[j - 1] > val:
            array[j] = array[j - 1]
            j -= 1
        array[j] = val
```

runtime analysis (later): no fundamental difference
nevertheless: preferable if direct assignment possible

Properties of Insertion Sort

- in-place: additional storage does not depend on input size
- running time: adaptive for partially sorted inputs
  - with already sorted input, immediate exit from inner loop
  - with reversely sorted input, every element moved step-by-step to the front

  exact analysis: later
- stable: elements only moved to the left as long it is swapped with a strictly larger element.
  $\rightarrow$ cannot change relative order with an equal element

A3.4 Summary

Summary

- selection sort and insertion sort are two simple sorting algorithms.
- selection sort builds the sorted sequence from left to right by successively swapping a minimal element from the unsorted range to the end of the sorted range.
- insertion sort considers the elements from left to right and moves them to the correct position in the already sorted range at the beginning of the sequence.