Algorithms and Data Structures

A5. Runtime Analysis: Introduction and Selection Sort

Gabriele Röger

University of Basel

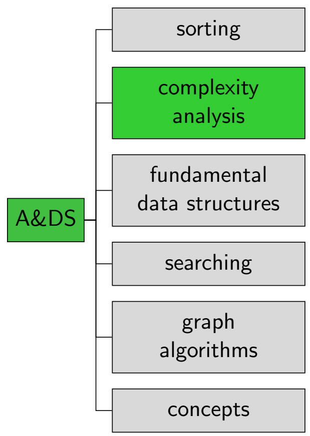
March 6, 2024

Algorithms and Data Structures

March 6, 2024 — A5. Runtime Analysis: Introduction and Selection Sort

A5.1 Runtime Analysis in General

Content of the Course



Exact Runtime Analysis Unrealistic

- Would be nice: formula that determines for a specific input how long the computation will take.
- Exact runtime prediction is hard because of too many influencing factors.
  - Speed and architecture of the computer
  - Programming language
  - Compiler version
  - Current load (what else is running?)
  - Caching behavior

We neither can nor want to consider all this in a formula.

Runtime Analysis: 1st Simplification

Don't measure time but count operations

What is an operation?

- Ideally: one line of machine code or – even more precisely – one processor cycle
- Instead: constant-time operations
  - Constant time: running time independent of input.
  - Ignore runtime differences of different operations.
  - E.g. addition, assignments, branching, function call.
  - Roughly: operation = one line of code.
  - But: also consider what's behind it e.g. steps inside the called function.

Running time roughly proportional to the number of operations

Runtime Analysis: 2nd Simplification

Don't count exactly but use bounds!

- Mostly considering upper bounds
  How many steps does it take at most?
- Sometimes also lower bound
  How many steps are at least executed?

„running time“ for bound on number of executed operations

Runtime Analysis: 3rd Simplification

Bounds only relative to the input size

- $T(n)$: running time for input of size n
- For adaptive algorithms we distinguish
  - Best case
    running time for best possible input of size n
  - Worst case
    running time for worst possible input of size n
  - Average case
    average running time over all inputs of size n

Cost Models

Sometimes: analysis wrt. cost model

- Identify fundamental operations for the algorithm class e.g. for sorting algorithms.
  - Key comparison
  - Swap of two elements or movement of an element
- Analyze number of these operations.

Example from C++ Reference

function template

std::sort <algorithm>

default (1)
```
template <class RandomAccessIterator>
  void sort (RandomAccessIterator first, RandomAccessIterator last);
```
custom (2)
```
template <class RandomAccessIterator, class Compare>
  void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Sort elements in range

Sorts the elements in the range `[first,last)` into ascending order.

The elements are compared using `operator<` for the first version, and *comp* for the second.

Equivalent elements are not guaranteed to keep their original relative order (see `stable_sort`).

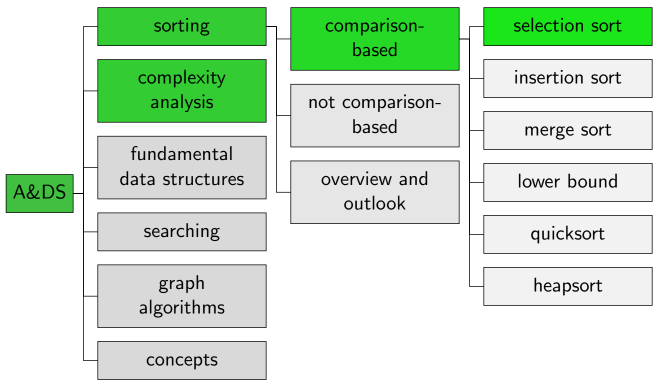
Complexity

On average, linearithmic in the distance between *first* and *last*: Performs approximately $N*\log_2(N)$ (where N is this distance) comparisons of elements, and up to that many element swaps (or moves).

http://www.cplusplus.com/reference/algorithm/sort/

A5.2 Example: Selection Sort

Content of the Course



Selection Sort: Algorithm

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


Selection Sort with Cost Model

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

→ n-1 swaps of two elements ("linear")
→ 0.5(n-1)n key comparisons ("quadratic")

Selection Sort: Analysis I

We show: $T(n) \leq c' \cdot n^2$ for $n \geq 1$ and some constant c'

- Outer loop (3-10) and inner loop (6-8)
- Number of operations for each iteration of the outer loop:
  - Constant a for no. of operations in lines 7 and 8
  - Constant b for no. of operations in lines 5 and 10

i	# operations
0	$a(n-1)+b$
1	$a(n-2)+b$
	...
n-2	$a \cdot 1 + b$

- Total: $T(n) = \sum_{i=0}^{n-2}(a(n-(i+1))+b)$

Selection Sort: Analysis II

$$\begin{aligned}
T(n) &= \sum_{i=0}^{n-2}(a(n-(i+1))+b) \\
&= \sum_{i=1}^{n-1}(a(n-i)+b) \\
&= a\sum_{i=1}^{n-1}(n-i) + b(n-1) \\
&= 0.5a(n-1)n + b(n-1) \\
&\leq 0.5an^2 + b(n-1) \\
&\leq 0.5an^2 + b(n-1)n \\
&\leq 0.5an^2 + bn^2 \\
&= (0.5a+b)n^2
\end{aligned}$$

$\Rightarrow$ with $c' = (0.5a+b)$ it holds for $n \geq 1$ that $T(n) \leq c' \cdot n^2$

Selection Sort: Analysis III

Too generous bound?

We show for $n \geq 2$: $T(n) \geq c \cdot n^2$ for some constant c

$$\begin{aligned} T(n) &= \cdots = 0.5a(n-1)n + b(n-1) \\ &\geq 0.5a(n-1)n \\ &\geq 0.25an^2 \qquad (n-1 \geq 0.5n \text{ for } n \geq 2) \end{aligned}$$

$\Rightarrow$ with $c = 0.25a$ it holds for $n \geq 2$ that $T(n) \geq c \cdot n^2$

Theorem

Selection sort has quadratic running time, i.e., there are constants $c > 0, c' > 0, n_0 > 0$ such that for $n \geq n_0$: $cn^2 \leq T(n) \leq c'n^2$.

Selection Sort: Analysis IV

Quadratic running time: twice as large input, fourfold running time

What does this mean in practice?

- Assumption: $c = 1$, one operation takes on average 10^{-8} sec.
- With 1000 elements, we wait $10^{-8} \cdot (10^3)^2 = 10^{-8} \cdot 10^6 = 10^{-2} = 0.02$ seconds.
- With 10 thousand elements, we wait $10^{-8} \cdot (10^4)^2 = 1$ second.
- With 100 thousand elements $10^{-8} \cdot (10^5)^2 = 100$ seconds.
- With 1 million elements $10^{-8} \cdot (10^6)^2$ seconds $= 2.77$ hours.
- With 1 billion elements $10^{-8} \cdot (10^9)^2$ seconds $= 317$ years.
  1 billion numbers with 4 bytes/number are „only“ 4 GB.

Quadratic running time problematic for large inputs

A5.3 Summary

Summary

- Runtime analysis considers bounds on the number of executed operations.
  - We don't count exactly.
  - We ignore how long each operation actually takes.
  - Running time should be roughly proportional to the number of operations.
- Selection sort has quadratic running time and performs a linear number of swaps and a quadratic number of key comparisons.