Efficient Sorting Algorithms

Helmut Alt
Freie Universität Berlin
alt@mi.fu-berlin.de

September 22, 2014

Specification of the Problem

Specification of the Problem

input:

a sequence of numbers (or other objects which can be compared to each other and sorted)

Specification of the Problem

input:
a sequence of numbers (or other objects which can be compared to each other and sorted)

output:
The same numbers in ascending order (i.e., from smallest to largest)

Algorithm 1: a Simple Algorithm

Algorithm 1: a Simple Algorithm

-
  - go through the sequence from left to right, in each step:

Algorithm 1: a Simple Algorithm

-
  - go through the sequence from left to right, in each step:
    - if the current element is larger than its right neighbor: swap the two elements

.

Algorithm 1: a Simple Algorithm

-
  - go through the sequence from left to right, in each step:
    - if the current element is larger than its right neighbor: swap the two elements

.

Algorithm 1: a Simple Algorithm

- repeat until no more swaps occur:
  - go through the sequence from left to right, in each step:
    - if the current element is larger than its right neighbor: swap the two elements

.

Algorithm 1: a Simple Algorithm

- repeat until no more swaps occur:
  - go through the sequence from left to right, in each step:
    - if the current element is larger than its right neighbor: swap the two elements

This algorithm is called Bubblesort.

Runtime Analysis of Bubblesort

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Sorting 8 numbers:

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Sorting 8 numbers:

- One scan makes 7 comparisons.

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Sorting 8 numbers:

- One scan makes 7 comparisons.
- The first scan brings the largest element to the rightmost position.

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Sorting 8 numbers:

- One scan makes 7 comparisons.
- The first scan brings the largest element to the rightmost position.
- The second scan brings the second largest element to the second rightmost position.

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Sorting 8 numbers:

- One scan makes 7 comparisons.
- The first scan brings the largest element to the rightmost position.
- The second scan brings the second largest element to the second rightmost position.
- etc.

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Sorting 8 numbers:

- One scan makes 7 comparisons.
- The first scan brings the largest element to the rightmost position.
- The second scan brings the second largest element to the second rightmost position.
- etc.
- so: at most 7 scans

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Sorting 8 numbers:

- One scan makes 7 comparisons.
- The first scan brings the largest element to the rightmost position.
- The second scan brings the second largest element to the second rightmost position.
- etc.
- so: at most 7 scans
- altogether: at most 49 comparisons

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Sorting n numbers:

- One scan makes 7 comparisons.
- The first scan brings the largest element to the rightmost position.
- The second scan brings the second largest element to the second rightmost position.
- etc.
- so: at most 7 scans
- altogether: at most 49 comparisons

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Sorting n numbers:

- One scan makes $n - 1$ comparisons.
- The first scan brings the largest element to the rightmost position.
- The second scan brings the second largest element to the second rightmost position.
- etc.
- so: at most 7 scans
- altogether: at most 49 comparisons

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Sorting n numbers:

- One scan makes $n - 1$ comparisons.
- The first scan brings the largest element to the rightmost position.
- The second scan brings the second largest element to the second rightmost position.
- etc.
- so: at most $n - 1$ scans
- altogether: at most 49 comparisons

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Sorting n numbers:

- One scan makes $n-1$ comparisons.
- The first scan brings the largest element to the rightmost position.
- The second scan brings the second largest element to the second rightmost position.
- etc.
- so: at most $n-1$ scans
- altogether: at most $(n-1)^2$ comparisons

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Sorting n numbers:

- One scan makes $n-1$ comparisons.
- The first scan brings the largest element to the rightmost position.
- The second scan brings the second largest element to the second rightmost position.
- etc.
- so: at most $n-1$ scans
- altogether: at most $(n-1)^2$ comparisons
- improvement: make 1st scan for n elements, 2nd for $n-1$, etc.:

Runtime Analysis of Bubblesort

We count the number of comparisons the algorithm makes.

Sorting n numbers:

- One scan makes $n-1$ comparisons.
- The first scan brings the largest element to the rightmost position.
- The second scan brings the second largest element to the second rightmost position.
- etc.
- so: at most $n-1$ scans
- altogether: at most $(n-1)^2$ comparisons
- improvement: make 1st scan for n elements, 2nd for $n-1$, etc.:
  $n(n-1)/2$ comparisons.

Algorithm 2

Algorithm 2

1. If the sequence contains only one number, return it immediately, otherwise:

Algorithm 2

1. If the sequence contains only one number, return it immediately, otherwise:
2. Split the sequence into two parts of equal size. Give each part to a helper and ask him, to sort it recursively, i.e., also exactly by the method described here.

Algorithm 2

1. If the sequence contains only one number, return it immediately, otherwise:
2. Split the sequence into two parts of equal size. Give each part to a helper and ask him, to sort it recursively, i.e., also exactly by the method described here.
3. Wait until both helpers have given back the sorted parts. Then traverse both sequences from left to right and merge the cards by a kind of zipper-principle to a sorted full sequence.

Algorithm 2

1. If the sequence contains only one number, return it immediately, otherwise:
2. Split the sequence into two parts of equal size. Give each part to a helper and ask him, to sort it recursively, i.e., also exactly by the method described here.
3. Wait until both helpers have given back the sorted parts. Then traverse both sequences from left to right and merge the cards by a kind of zipper-principle to a sorted full sequence.
4. Return this sequence to your master.

Algorithm 2

1. If the sequence contains only one number, return it immediately, otherwise:
2. Split the sequence into two parts of equal size. Give each part to a helper and ask him, to sort it recursively, i.e., also exactly by the method described here.
3. Wait until both helpers have given back the sorted parts. Then traverse both sequences from left to right and merge the cards by a kind of zipper-principle to a sorted full sequence.
4. Return this sequence to your master.

This algorithm is called Mergesort.

Divide-and-Conquer

The strategy used in Mergesort:

Divide-and-Conquer

The strategy used in Mergesort:

1. If the problem size is small enough: solve directly, otherwise:

Divide-and-Conquer

The strategy used in Mergesort:

1. If the problem size is small enough: solve directly, otherwise:
2. Split the problem into several subproblems of smaller size and solve those recursively.

Divide-and-Conquer

The strategy used in Mergesort:

1. If the problem size is small enough: solve directly, otherwise:
2. Split the problem into several subproblems of smaller size and solve those recursively.
3. Combine the solutions of the subproblems to one of the complete problem.

Algorithm 3

Algorithm 3

1. If the sequence consists of one element only then return it immediately, otherwise:

Algorithm 3

1. If the sequence consists of one element only then return it immediately, otherwise:
2. Take the first card from the sequence. Go through the remaining cards and split them into the ones with a value notgreater than the one of the first card (sequence 1) and the ones with a value greater than the one of the first card (sequence 2).

Algorithm 3

1. If the sequence consists of one element only then return it immediately, otherwise:
2. Take the first card from the sequence. Go through the remaining cards and split them into the ones with a value notgreater than the one of the first card (sequence 1) and the ones with a value greater than the one of the first card (sequence 2).
3. Give each sequence obtained this way, if it contains cards at all, to a helper asking him to sort it recursively, i.e., also exactly by the method described here.

Algorithm 3

1. If the sequence consists of one element only then return it immediately, otherwise:
2. Take the first card from the sequence. Go through the remaining cards and split them into the ones with a value notgreater than the one of the first card (sequence 1) and the ones with a value greater than the one of the first card (sequence 2).
3. Give each sequence obtained this way, if it contains cards at all, to a helper asking him to sort it recursively, i.e., also exactly by the method described here.
4. Wait until both helpers have returned the sorted parts then put at the left the sorted sequence 1, then the card drawn in the beginning, then the sorted sequence 2, and return the whole as a sorted sequence.

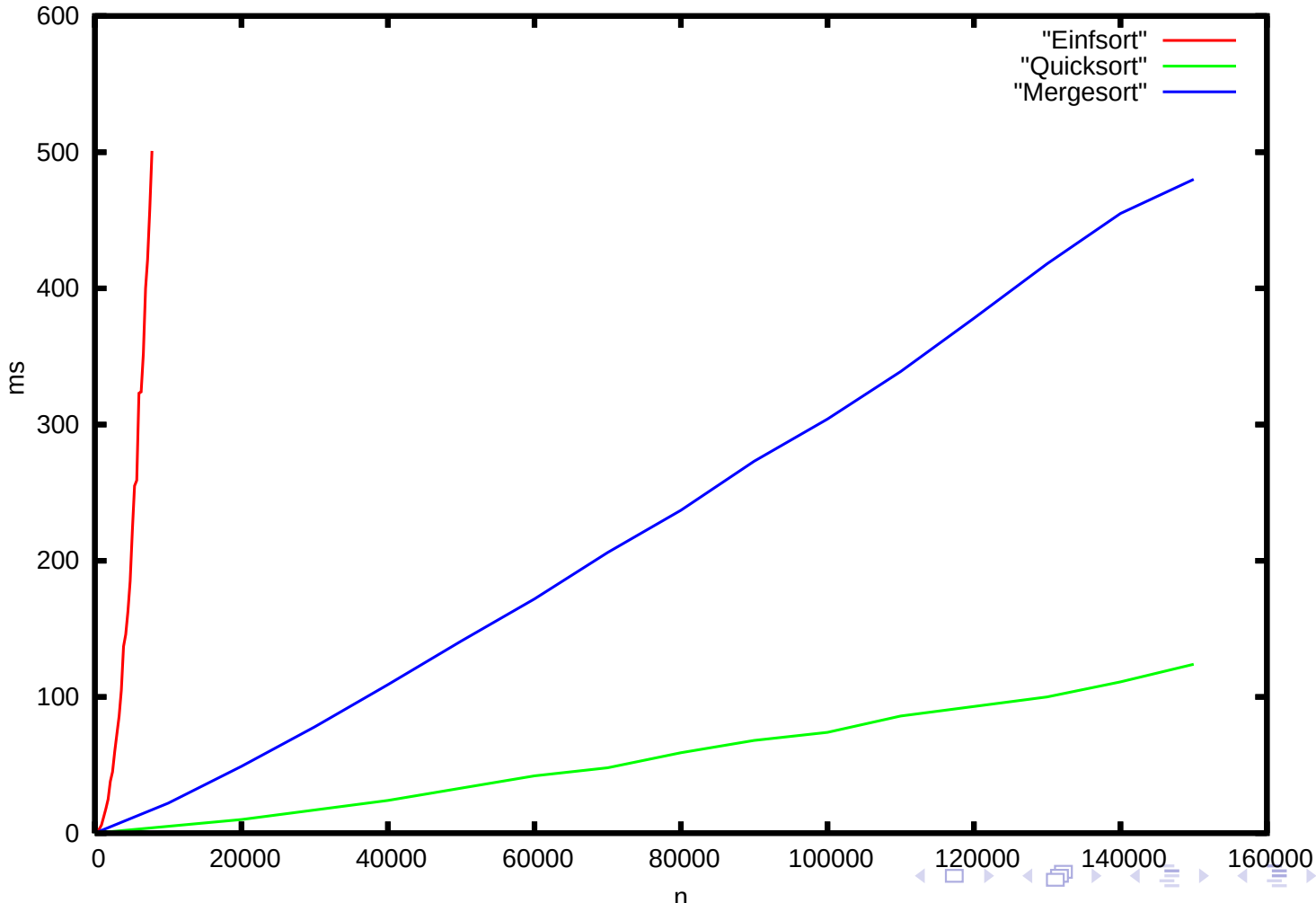
Algorithm 3

1. If the sequence consists of one element only then return it immediately, otherwise:
2. Take the first card from the sequence. Go through the remaining cards and split them into the ones with a value notgreater than the one of the first card (sequence 1) and the ones with a value greater than the one of the first card (sequence 2).
3. Give each sequence obtained this way, if it contains cards at all, to a helper asking him to sort it recursively, i.e., also exactly by the method described here.
4. Wait until both helpers have returned the sorted parts then put at the left the sorted sequence 1, then the card drawn in the beginning, then the sorted sequence 2, and return the whole as a sorted sequence.

This algorithm is called Quicksort.

Experimental comparison of the sorting algorithms

Runtime of an implementation in milliseconds for different sizes of the input sequence.



Remarks

Remarks

- The sorting algorithms presented cannot only sort numbers but any kind of objects where one can be compared to another, e.g., names by alphabetic order, packages by weight using a balance scale,...

Remarks

- The sorting algorithms presented cannot only sort numbers but any kind of objects where one can be compared to another, e.g., names by alphabetic order, packages by weight using a balance scale,...
- Recursion is an important tool in algorithm design and available in most high level programming languages. (Not in Scratch, though.)

Remarks

- The sorting algorithms presented cannot only sort numbers but any kind of objects where one can be compared to another, e.g., names by alphabetic order, packages by weight using a balance scale,...
- Recursion is an important tool in algorithm design and available in most high level programming languages. (Not in Scratch, though.)
- An algorithm can be formulated in a natural language and even (roughly) analyzed if described in this manner. An explicit program is not necessary: difference between algorithm design and programming.

Remarks

- The sorting algorithms presented cannot only sort numbers but any kind of objects where one can be compared to another, e.g., names by alphabetic order, packages by weight using a balance scale,...
- Recursion is an important tool in algorithm design and available in most high level programming languages. (Not in Scratch, though.)
- An algorithm can be formulated in a natural language and even (roughly) analyzed if described in this manner. An explicit program is not necessary: difference between algorithm design and programming.
- There are large differences in the runtimes of different sorting algorithms. A careful runtime analysis already when designing an algorithm makes sense.

Links

- **Visual comparison of sorting algorithms**
  `http://www.sorting-algorithms.com/`
- **Acoustic demonstration of sorting algorithms**
  `http://www.youtube.com/watch?v=kPRA0W1kECg`
- **Sorting algorithms demonstrated by folk dances**
  - **Bubblesort**
    `http://www.youtube.com/watch?v=lyZQPjUT5B4`
  - **Mergesort**
    `http://www.youtube.com/watch?v=XaqR3G_NVoo`
  - **Quicksort**
    `http://www.youtube.com/watch?v=ywWBy6J5gz8`
- **Demo in Scratch**
  `http://scratch.mit.edu/projects/682862/`

Homework

1. Sort the following sequence of 3-letter words *alphabetically* by Bubblesort, Mergesort, and Quicksort:
   JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC.
2. Give a recursive algorithm for the *selection* problem specified as follows:
   Given an unsorted sequence of numbers (or other comparable elements) and a number k, *find* the k-th smallest element in the sequence, i.e., the one that occurs at the k-th position from the left, if the input sequence is sorted.
   For example, the 4th smallest element in the sequence of exercise 1 would be FEB.
   Don't use sorting, but find a more efficient algorithm.
   Demonstrate your algorithm with an example.
   Hint: The idea of splitting the input sequence as in Quicksort may be used, then one recursive call to a smaller subsequence gives the result.