insertion sort: correctness and time complexity

```
Algorithm insertionSort(A, n):
    for j := 2 to n do
        key := A[j]
        i := j – 1
        while i ≥ 1 and A[i] > key do
            A[i + 1] := A[i]
            i := i – 1
        A[i + 1] := key
```

our model

- computer: Random Access Machine (RAM)
- algorithm: description in pseudo-code
- data structure: specification as Abstract Data Type (ADT)
- (worst-case) time complexity: (upper bound on) running time as function of the input size
- theoretical analysis: independent of hardware, software, and for all inputs

Random Access Machine (RAM)

- Central Processing Unit (CPU) with memory
- unlimited number of memory cells (registers)
- primitive operations take constant (little) time
we write algorithms in pseudo-code

pseudo-code resembles a programming language
but is independent of specific syntax

pseudo-code: globally

• input
• output using **return**
• block structure via indentation
• declare and call procedures
• declare and use data structures
• recursive calls
• objects with attributes, for example **A.length**

pseudo-code: calculating

• **booleans**: true, false
• calculating with booleans: **and**, **or** (short-circuiting)

• **integers**
• calculating with integers: addition, subtraction, multiplication, modulo
• elementary tests on integers: greater than, less than

pseudo-code: control

• declare and use variables
• assignment
• declare and update arrays and array elements
• if **then**, **while** **do**, for **do**, **repeat**,
time complexity

function $T(n)$ computes the amount of steps
we are mostly interested in the growth of $T(n)$ if $n$ increases
we often give the asymptotic approximation $\Theta$, $O$

overview

- recap
- selection sort
- bubble sort
- merge sort
- divide and conquer
- material

overview

- recap
- selection sort
- bubble sort
- merge sort
- divide and conquer
- material

selection sort: idea and example

the list consists of a sorted part followed by an unsorted part

initially: the sorted part is empty

loop: while the unsorted part is non-empty:
look for its smallest element
swap it with the first element
that element now belongs to the sorted part

5, 4, 3, 7, 1
1, 4, 3, 7, 5
1, 3, 4, 7, 5
1, 3, 4, 5, 7
selection sort: pseudo-code

**Algorithm** selectionSort($A, n$):
   for $i := 1$ to $n - 1$ do
      $m := i$
      for $j = i + 1$ to $n$ do
         if $A[j] < A[m]$ then
            $m := j$
      $x := A[m]$
      $A[m] := A[i]$
      $A[i] := x$
   we do a swap

selection sort: correctness

**loop-invariant $I$:**
   at the start of the for-loop:
   $A[1\ldots (i − 1)]$ is sorted,
   and every element in $A[1\ldots (i − 1)]$ is smaller than
   every element in $A[i\ldots n]$

   **init:** $I$ is true initially (for $i = 1$, so for $A[1\ldots i] = []$)

   **loop:** $I$ remains true under the loop (!)

   **end:** $I$ holds for $i = n$ implies that the array is sorted

selection sort: worst-case time complexity

worst-case: if $A[j] < A[m]$ we do an assignment for $m$
we do the swap anyway
for a fixed $i$, we do $n − i$ times the test $A[j] < A[m]$
the important part:

\[
\sum_{i=1}^{n-1} (n - i) = (n - 1) + \ldots + 1 + 0
\]
\[
= \sum_{j=1}^{n-1} j
\]
\[
= \frac{1}{2}(n - 1)n
\]

selection sort: asymptotic bound

worst-case time complexity of selection sort is in $\Theta(n^2)$
**bubble sort: idea**

Repeatedly traverse the list from left to right, compare each pair of adjacent numbers, and swap them if they are in the wrong order.

After a traversal, the rightmost element is always largest in the traversed list, so can be left out from the next traversal.

**bubble-sort: pseudocode**

**Algorithm** `bubbleSort(A, n):
  for i := n – 1 downto 1 do
    for j := 1 to i do
        swap(A[j], A[j + 1])

\textbf{input}: array } A \text{ of length } n
bubble sort: analysis

correctness?
worst-case time complexity?
best-case time complexity?

merge sort: idea

how do we sort?

loop: while the sequence is non-empty

split the sequence into two parts of (almost) equal length

sort the two subsequences using mergesort (recursion)

merge the two sorted sublists

how do we merge?

loop: as long as both sequences are non-empty

compare the first elements of both sequences

remove the smallest and move it to the end of the output-sequence

last: if one of the sequences is empty, add the other to the tail of the output-sequence

mergesort: some pseudo-code

Algorithm mergeSort(A, p, r):
  if p < r then
    q := ⌊(p + r)/2⌋
    mergeSort(A, p, q)
    mergeSort(A, q + 1, r)
    Merge(A, p, q, r)

where is the work done?

for pseudo-code Merge see the book
input:

$A = [1, 3, 5, 7, 2, 4, 6, 8]$

$p = 1$

$q = 4$

$r = 8$

$A[1 \ldots 4]$ is sorted and $A[5 \ldots 8]$ is sorted

question

give the recursion tree of applying mergesort to the sequence

merge sort: example

for input of size $n = 2^i$, the recursion tree has $\log n + 1 = i + 1$ layers

(for general $n$: $\lceil \log n \rceil + 1$)

the work per layer is in $\Theta(n)$

so the worst-case time complexity is in $\Theta(n \log n)$
**mergesort: time complexity via recurrence equations**

*divide* is easy, in $\Theta(1)$

*conquer* uses recursion, two subproblems of size half

*combine* is difficult, in $\Theta(n)$

First assume $n = 2^i$

$$T(n) = \begin{cases} 
1 & \text{als } n = 1 \\
2T\left(\frac{n}{2}\right) + n & \text{als } n > 1
\end{cases}$$

**mergesort: drawback**

**space complexity**

Merging two sublists takes place outside the memory for the input-sequence

So worst-case time complexity of insertion sort is in $\Theta(n^2)$

Worst-case time complexity of merge sort is in $\Theta(n \cdot \log n)$

But insertion sort is in-place and merge sort is not in-place

**obvious question:** is there a $n \cdot \log n$ in-place sorting algorithm?

**mergesort: inventor**

Mergesort was invented by John von Neumann (1903–1957) in 1945

---

**overview**

- recap
- selection sort
- bubble sort
- merge sort
- divide and conquer
- material
divide and conquer

Mergesort is a divide-and-conquer algorithm

- **divide**: divide the problem into smaller subproblems
- **conquer**: solve the subproblems using recursion
- **combine**: combine the solutions to the subproblems to a solution of the original problem

Logarithms: what

Addition is symmetric, has inverse subtraction:
\[ (m + n) - m = (n + m) - n = m \]

Multiplication is symmetric, has inverse division:
\[ \frac{m \cdot n}{m} = \frac{n \cdot m}{n} = m \]

Exponentiation is not symmetric, has two inverses root and log:
\[ n^{\log(m^n)} = m \]

For mergesort, a divide-and-conquer algorithm dividing an input of size \(2^k\) into subproblems of size 1 takes \(k + 1 = \log 2^k + 1\) steps.

Correctness: use induction

Idea algorithm:

Divide and conquer: tile a \(2 \times 2\) square with \(L\)-tiles

Leave open exactly one tile which is selected beforehand

Typically logarithms occur in the complexity analysis of divide and conquer algorithms
overview

• recap
• selection sort
• bubble sort
• merge sort
• divide and conquer

material

sorteren: puzzle

• books on algorithms by David Harel
• art of computer programmingDonald Knuth
• art of computer programming
• how can we sort 5 elements using at most 7 comparisons?

experiment: sort 5 elements on 5 memory locations

from 'Sorting and Searching' by Knuth