Algorithmics

Sebastian Iwanowski
FH Wedel

1. Introduction into formal algorithmics

Algorithmics 1

1.1 Comparing basic sorting techniques

- Decription and functionality of algorithms: Permutationsort, Selectionsort, Mergesort, Quicksort

  Description in words, graphic visualization using arrays

- Estimating the run time for the worst case

  Setting up recursive equations, computing an explicite solution

  Run time estimation using the Big-O notation

- Results:

Permutationsort:	O(exp(n))
Selectionsort:	$O(n^2)$
Mergesort:	O(n log n)
Quicksort:	$O(n^2)$

References

Alt S. 4 – 7 (in German), Cormen ch. 2, Levitin ch. 3.1, ch. 4

visual demonstration: https://www.youtube.com/watch?v=yn0EgXHb5jc

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of Selectionsort: „brute force“ strategy

- Pass all positions of data array in order.
- Search the minimum element upward from current position.
- Swap this element with element of current position.
- Output the new array after all position have been passed.

```
procedure selectionsort (data): array
begin
  pos := 1;
  while pos < length(data) do
  begin
    newPos := minPos (data, pos, length(data));
    aux := data[pos];
    data[pos] := data[newPos];
    data[newPos] := aux;
    pos := pos + 1;
  end; {while}
  return data;
end {selectionsort}
```

```
procedure sort (data): array
begin
  newData := copy (data);
  return selectionsort (newData);
end {sort}
```

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of auxiliary procedure *minPos*:

```
procedure minPos (data, first, last): integer
begin
  resultPos := first;
  resultValue := data[resultPos];
  pos := first;
  while pos < last do
  begin
    pos := pos + 1;
    if data[pos] < resultValue
      then
        begin
          resultPos := pos;
          resultValue := data[resultPos];
        end;
  end; {while}
  return resultPos;
end {minPos}
```

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of Mergesort: „divide and conquer“ strategy

- Divide data array into 2 halves.
- Sort the halves separately.
- Merge the sorted halves into a second array.

```
procedure mergesort
        (fromData, toData, left, right)
begin
  if left < right
  then
    begin
      mid := (left + right) div 2;
      mergesort (toData, fromData,
                              left, mid);
      mergesort (toData, fromData,
                           mid+1, right);
      merge (fromData, toData,
               left, mid, mid+1, right);
    end {if}
  end {mergesort}
```

Recursive version

```
procedure sort (data): array
begin
  data1 := copy (data);
  data2 := copy (data);
  mergesort (data1,
     data2, 1, length(data));
  return data2
end {sort}
```

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of Mergesort: „divide and conquer“ strategy

- Divide data array into 2 halves.
- Sort the halves separately.
- Merge the sorted halves into a second array.

Iterative version

```
procedure mergesortIter (data): array
begin
  data2 := copy (data); n := length(data);
  sortedLength := 1;
  while sortedLength < n do
  begin
    left1 := 1;
    while (left1+sortedLength) < n do
    begin
      right1 := left1+sortedLength; left2 := right1+1; right2 := left2+sortedLength;
      merge (data, data2, left1, right1, left2, right2);
      left1 := right2 + 1
    end;
    sortedLength := sortedLength + sortedLength;
    aux := data; data := data2; data2 := aux
  end;
  return data
end {sort2}
```

```
procedure sort (data): array
begin
  newData := copy (data);
  return mergesortIter(newData)
end {sort}
```

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of auxiliary procedure *merge*:

```
procedure merge (fromData, toData, left1,
                    right1, left2, right2)
begin
  pos1 := left1; pos2 := left2; pos := left1;
  while (pos ≤ right2) do
  begin
    if pos1 > right1
      then
        assign (fromData, toData, pos2, pos)
      else if pos2 > right2
        then
          assign (fromData, toData, pos1, pos)
      else if fromData[pos1] ≤ fromData[pos2]
        then
          assign (fromData, toData, pos1, pos)
        else
          assign (fromData, toData, pos2, pos);
    pos := pos + 1
  end {while}
end {merge}
```

```
procedure assign (fromData, toData,
                     fromPos, toPos)
begin
  toData[toPos] := fromData[fromPos];
  fromPos := fromPos + 1;
end {assign}
```

assign must declare the parameters toData and fromPos as call by reference !

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of Quicksort „divide and conquer“ strategy

- Quicksort (A, i, j):

  A is an array of n elements (a[1], …, a[n]).
  i,j are indices between 1 and n.
  At the end, the elements between a[i] and a[j] are sorted in an increasing order.

- Partition (A,i,k,j) → order:

  At the end, A is rearranged between a[i] and a[j] such that
  first, only elements ≤ x := a[k] are chosen, then x, then only elements > x.
  The return value order is the new position of x.

- Implementation of Quicksort (A, i, j): Start with Quicksort (A,1,n)

```
if i < j
   then k := random number between i and j;
        dividingIndex := Partition (A,i,k,j);
        Quicksort (A, i, dividingIndex-1);
        Quicksort (A, dividingIndex+1,j);
```

References:
Cormen ch. 7.1 (algorithm there without random number)
Levitin ch. 4.2

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of Quicksort „divide and conquer“ strategy

- Partition (A,i,k,j) → order:

  At the end, A is rearranged between a[i] and a[j] such that
  first, only elements ≤ x := a[k] are stored, then x, then only elements > x.
  The return value `order` is the new position of x.

- Implementation of Partition:

```
x := a[k];
count := number of elements ≤ x between a[i] and a[j];
order := i+count-1;
Swap x with a[order]; // now x is placed on correct new position
right := j;
for left := 0 to count-2 do
   if a[i+left] > x
      then while a[right] > x do right := right - 1;
           Swap a[i+left] with a[j];
return order;
```

References:
Cormen ch. 7.1 (algorithm there without random number)
Levitin ch. 4.2

Algorithmics 1

1.1 Comparing basic sorting techniques

Exact run time estimate: $\Theta(n^2)$

- lower run time estimate $\Omega(n^2)$:

  For each n there is an input of size n with run time in $\Omega(n^2)$

- upper run time estimate $O(n^2)$:

  using the recursive equation of script and explicite solution of the following: $T(n) \leq c \bullet n^2$ (proof by complete induction by n)

  Remark to German script:
  The proposition that k=1 or k= n are the worst cases (which is true) is not proven in the script, but this is not necessary to show in order to show the above run time limits.

References:
Alt S. 7 (in German)
Cormen ch. 7.2

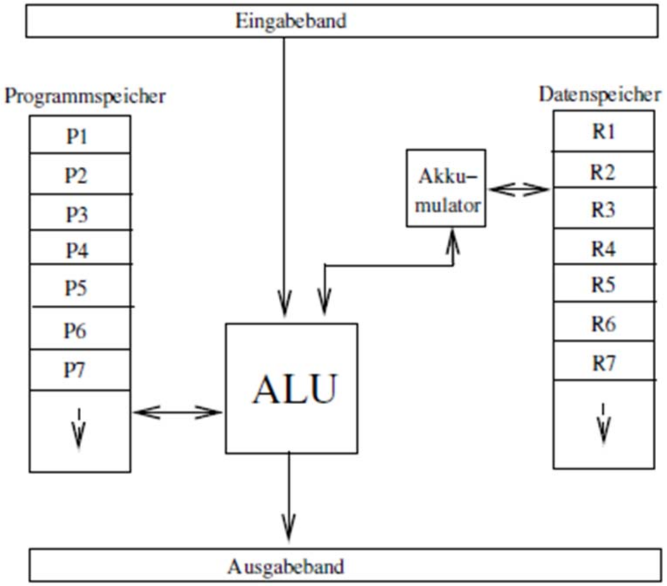
Algorithmics 1

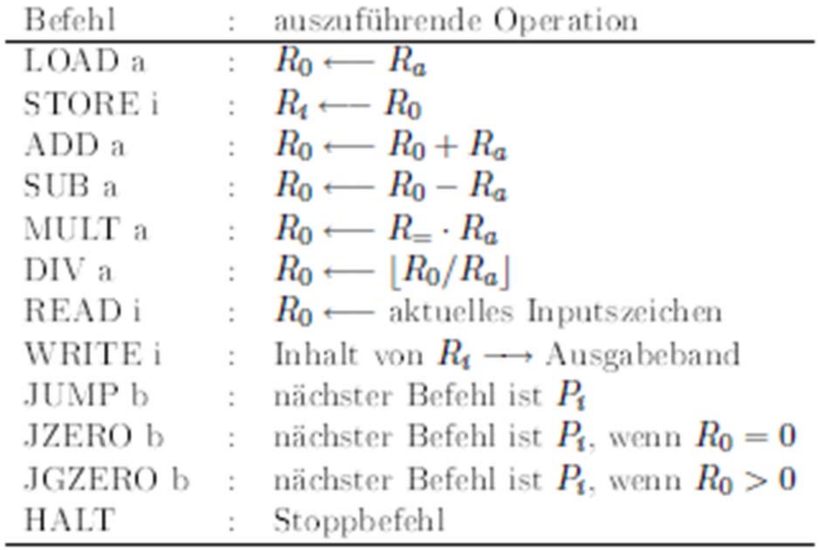
1.2 Complexity measures for the analysis of algorithms

Computational model: RAM (Random Access Machine)

- Definition of a RAM

  small assembler-like command pool,
  control unit with constant time access to program storage and data storage





Befehl	:	auszuführende Operation
LOAD a	:	$R_0 \longleftarrow R_a$
STORE i	:	$R_i \longleftarrow R_0$
ADD a	:	$R_0 \longleftarrow R_0 + R_a$
SUB a	:	$R_0 \longleftarrow R_0 - R_a$
MULT a	:	$R_0 \longleftarrow R_= \cdot R_a$
DIV a	:	$R_0 \longleftarrow \lfloor R_0 / R_a \rfloor$
READ i	:	$R_0 \longleftarrow$ aktuelles Inputszeichen
WRITE i	:	Inhalt von $R_i \longrightarrow$ Ausgabeband
JUMP b	:	nächster Befehl ist P_i
JZERO b	:	nächster Befehl ist P_i, wenn $R_0 = 0$
JGZERO b	:	nächster Befehl ist P_i, wenn $R_0 > 0$
HALT	:	Stoppbefehl

from Lang, ch. 4.5

References:
Alt S. 11-13 (in German)
Skript Lang, Kap. 4.5 (in German)
Mehlhorn ch. 2.2, 2.3 (outline, with a different perspective)

Algorithmics 1

1.2 Complexity measures for the analysis of algorithms

Computational model: RAM (Random Access Machine)

- Cost measures

UCM: All operations cost the same independent of operands' size.

LCM: The cost of an operation depends on size of operand.

- Run time equivalence

Algorithm requires on a RAM time in Θ(f(n)) (UCM oder LCM)
⇔ Algorithm requires the same time class Θ(f(n)) on a „normal" computer.

- Polynomial relation

Algorithm requires on a RAM time in Θ(f(n)) using LCM
⇔ Algorithm requires on a Turing machine time in Θ(P(f(n))) for a polynomial P.

References:

Alt S. 11-13 (in German)

Mehlhorn ch. 2.2, 2.3 (outline, with a different perspective)

Skript Lang, Kap. 4.5

Algorithmics 1

1.2 Complexity measures for the analysis of algorithms

Calculating with Landau symbols ("asymptotic size")

- Definition of O, Ω and Θ

  $T(n) \in O\,(f(n)) \Leftrightarrow \exists c \in \mathbb{R}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0\colon T(n) \leq c \bullet f(n)$
  $T(n) \in \Omega\,(f(n)) \Leftrightarrow \exists c \in \mathbb{R}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0\colon T(n) \geq c \bullet f(n)$
  $T(n) \in \Theta\,(f(n)) \Leftrightarrow \exists c_1, c_2 \in \mathbb{R}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0\colon c_1 \bullet f(n) \leq T(n) \leq c_2 \bullet f(n)$

- Computational rules for Landau symbols

  1) $x < y \Rightarrow O(n^x) \subsetneq O(n^y)$

  2) $x > 0 \Rightarrow O(\log n) \subsetneq O(n^x)$

  3) $O\,(f(n)+g(n)) \in O(f(n)) \cup O(g(n))$ ("maximum")

References:
Cormen ch. 3

1.2 Complexity measures for the analysis of algorithms

Master-Theorem for the asymptotic run time estimation of divide & conquer algorithms

Let $T(n)$ be the recursive equation for a divide & conquer algorithm given by:

$$T(n) = a\,T(n/b) + f(n)$$

Then for $f(n) \in \Theta(n^k)$ holds:

1) $a < b^k \Rightarrow T(n) \in \Theta(n^k)$

2) $a = b^k \Rightarrow T(n) \in \Theta(n^k \log n)$

3) $a > b^k \Rightarrow T(n) \in \Theta(n^{\log_b a})$

The same results hold for O and Ω

References:
Cormen ch. 4

Algorithmics 1

1.2 Complexity measures for the analysis of algorithms

Denoting the complexity of algorithms by Landau symbols

Let I(A) be an admissible input for algorithm A and size(I(A)) be the input size.
Let $T_A(I(A))$ be the run time of A (counting the number of operations), when I(A) is the input.

- Upper run time limit in worst case:

  A is an O(f(n)) algorithm $\Leftrightarrow \forall n \in \mathbb{N}\ \forall\ I(A), size(I(A))=n: T_A(I(A)) \in O\ (f(n))$
  "All inputs are bounded by this asymptotic run time."

- Lower run time limit in worst case:

  A is an Ω(f(n)) algorithm $\Leftrightarrow \forall n \in \mathbb{N}\ \exists\ I(A), size(I(A))=n: T_A(I(A)) \in \Omega\ (f(n))$
  "For each n there is an input with this asymptotic run time bound."

- Exact asymptotic run time in worst case:

  A is a Θ(f(n)) algorithm in a weak sense $\Leftrightarrow$
  A is an O(f(n)) algorithm and A is an Ω(f(n)) algorithm

  A is a Θ(f(n)) algorithm in a strong sense $\Leftrightarrow \forall n \in \mathbb{N}\ \forall\ I(A), size(I(A))=n: T_A(I(A)) \in \Theta\ (f(n))$
  "All inputs have this asymptotic run time."

References: ? (thanks for giving me hints)

Algorithmics 1

1.3 Lower bounds for algorithms using comparisons only

- Lower bound for the search of a maximum element

  Given n elements (input size).
  Compare graph must be connected ➔ at least n-1 comparisons (Ω(n))
  There is an O(n) algorithm for this problem ➔ This algorithm is optimal.

- Lower bound for the search of the k-th element of a given set

  Given n elements (input size).
  Compare graph must be connected ➔ at least n-1 comparisons (Ω(n))
  Optimal algorithm for this problem? ➔ Chapter 2

- Lower bound for sorting

  Correlate depth of a compare tree with the number of comparisons
  Correlate depth of a binary search tree with the number of leaves
  Estimate n! and make a conclusion for log (n!) ➔ at least Ω(n log n) comparisons
  Mergesort needs only O(n log n) comparisons ➔ Mergesort is optimal.

References:

Alt S. 17 – 21 (in German) Cormen ch. 8.1 Levitin ch. 11.1 (outline)