

Grundlagen der Programmierung

Vorlesung 11 vom 13.01.2005
Sebastian Iwanowski
FH Wedel

Grundlagen der Programmierung

1. Einführung

Grundlegende Eigenschaften von Algorithmen und Programmen

2. Logik

Aussagenlogik

Prädikatenlogik

3. Programmentwicklung und –verifikation

Grundlagen der Programmverifikation

Zuweisungen und Verbundanweisungen

Verzweigungen

Schleifen

Modularisierung

Rekursion

→ 4. Entwurf und Analyse von Algorithmen

Klassifikation von Algorithmen

Programmierung von Algorithmen

Bewertung von Algorithmen

Entwurf von Algorithmen

Gegeben eine Spezifikation:

Wie konstruiert man einen Algorithmus dafür ?

**Da die allgemeine Aufgabe nicht automatisierbar ist,
kann es nur Empfehlungen für Lösungsstrategien geben**

Suchproblem:

```
procedure search (data, k): integer
```

Eingabe: Datenfeld aus n Zahlen, gesuchte Zahl k

Ausgabe: Position von k im Datenfeld (wenn vorhanden, sonst 0)

Sortierproblem:

```
procedure sort (data): array
```

Eingabe: Datenfeld aus n Zahlen

Ausgabe: Ein neues Datenfeld mit denselben Zahlen,
aber aufsteigend sortiert

Entwurf von Algorithmen: Suchproblem

1. Strategie: Tue das Nächstliegende (greedy-Strategie)

Lösungsskizze: *Lineare Suche*

- Durchlaufe die Elemente vom Datenfeld der Reihe nach und vergleiche sie mit k. Erhöhe in jedem Durchlauf den Positionszähler.
- Wenn im Durchlauf das Element gefunden wird, dann gib den Positionszähler aus.
- Wenn im Durchlauf das Element nicht gefunden wird, dann gib 0 aus.

```
procedure searchLin (data, k): integer
begin
  pos := 1;
  while pos ≤ length(data) do
  begin
    if data[pos] = k then return pos;
    pos := pos + 1;
  end {while}
  return 0;
end {searchLin}
```

```
procedure search (data, k): integer
begin
  return searchLin (data, k)
end {search}
```

Entwurf von Algorithmen: Suchproblem

1. Strategie: Tue das Nächstliegende (greedy-Strategie)

Lösungsskizze: *Lineare Suche*

- Durchlaufe die Elemente vom Datenfeld der Reihe nach und vergleiche sie mit k. Erhöhe in jedem Durchlauf den Positionszähler.
- Wenn im Durchlauf das Element gefunden wird, dann gib den Positionszähler aus.
- Wenn im Durchlauf das Element nicht gefunden wird, dann gib 0 aus.

```
procedure searchLinRec (data, pos, k): integer
begin
  if pos > length(data) return 0;
  if data[pos] = k then return pos;
  return searchLinrec (data, pos+1, k)
end {searchLinRec}
```

```
procedure search (data, k): integer
begin
  return searchLinRec (data, 1, k)
end {search}
```

Entwurf von Algorithmen: Suchproblem

2. Strategie: Teile und herrsche (divide and conquer)

Lösungsskizze (Variante für allgemeine Daten):

- Vergleiche k mit dem Element in der Mitte:
- Wenn k gleich ist, dann gib die Mittelposition aus.
- Wenn k ungleich ist, dann suche in der linken Hälfte.
- Wenn k dort gefunden wurde, dann gib das Ergebnis als Gesamtergebnis aus.
- Wenn k dort nicht gefunden wurde, dann suche in der rechten Hälfte und gib das Ergebnis als Gesamtergebnis aus.

```
procedure searchDC (data, left, right, k): integer
```

```
begin
```

```
  if left > right then return 0;
```

```
  mid := (left + right) div 2;
```

```
  if k = data[mid] then return mid;
```

```
  pos := searchDC (data, left, mid-1, k);
```

```
  if pos ≠ 0 then return pos;
```

```
  return searchDC (data, mid+1, right, k);
```

```
end {searchDC}
```

```
procedure search (data, k): integer
```

```
begin
```

```
  return searchDC (data, 1, length (data), k)
```

```
end {search}
```

Entwurf von Algorithmen: Suchproblem

2. Strategie: Teile und herrsche (divide and conquer)

Lösungsskizze (Variante für sortierte Daten): *Binärsuche*

- Vergleiche k mit dem Element in der Mitte:
- Wenn k gleich ist, dann gib die Mittelposition aus.
- Wenn k kleiner ist, dann suche in der linken Hälfte und gib das Ergebnis aus.
- Wenn k größer ist, dann suche in der rechten Hälfte und gib das Ergebnis aus.

```
procedure binarySearch (data, left, right, k): integer
begin
  if left > right then return 0;
  mid := (left + right) div 2;
  if k = data[mid] then return mid;
  if k < data[mid] then return binarySearch (data, left, mid-1, k);
  return binarySearch (data, mid+1, right, k)
end {binarySearch}

procedure search (data, k): integer
begin
  newData = sort (data);
  return binarySearch (newData, 1, length (data), k)
end {search}
```

Entwurf von Algorithmen: Sortierproblem

1. Strategie: Tue das Nächstliegende (greedy-Strategie)

Lösungsskizze: *Selectionsort*

- Durchlaufe die Positionen des neuen Datenfelds der Reihe nach.
- Suche das kleinste Element ab der laufenden Position im neuen Datenfeld.
- Vertausche dieses Element mit dem Element der laufenden Position.
- Gib am Ende des Durchlaufs das neue Datenfeld aus.

```
procedure selectionsort (data): array
begin
  pos := 1;
  while pos ≤ length(newData) 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}
```

Entwurf von Algorithmen: Suchproblem

2. Strategie: Teile und herrsche (divide and conquer)

Lösungsskizze: *Mergesort*

- Teile das Datenfeld in 2 Hälften auf.
- Sortiere beide Hälften getrennt.
- Mische die beiden sortierten Hälften in ein zweites Datenfeld.

```
procedure mergesort (fromData, toData, left, right)
```

```
begin
```

```
  if left < right-1
```

```
  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}
```

```
procedure sort (data): array
```

```
begin
```

```
  data2 := copy (data);
```

```
  mergesort (data, data2, 1, length(data));
```

```
  return data2
```

```
end {sort}
```

Entwurf von Algorithmen: Suchproblem

2. Strategie: Teile und herrsche (divide and conquer)

Lösungsskizze: *Mergesort*

- Teile das Datenfeld in 2 Hälften auf.
- Sortiere beide Hälften getrennt.
- Mische die beiden sortierten Hälften in ein zweites Datenfeld.

```
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}
```

Entwurf von Algorithmen

Wie klassifiziert man Algorithmen ?

- **offensichtlich nicht durch die Unterscheidung rekursiv / iterativ !**
- **Unterscheidung nach Lösungsstrategie (greedy, divide and conquer, ...) schon besser !**

Welche Implementierungsvariante (rekursiv / iterativ) ist zu einem gegebenen Algorithmus zu bevorzugen ?

Entwurf von Algorithmen

Welche Lösungsstrategie ist für ein gegebenes Problem besser ?

Unterscheiden sich die Algorithmen, die zu einer Lösungsstrategie gehören ?

Wie unterscheidet man zwischen Implementierungsvarianten desselben Algorithmus und zwei verschiedenen Algorithmen ?

Beim nächsten Mal:
Bewertung von Algorithmen