Grundlagen der Programmierung

Vorlesung 12 vom 20.01.2005

Sebastian Iwanowski
FH Wedel

Entwurf von Algorithmen: Suchproblem

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

Lösungsskizze (Variante für sortierte Daten): *Binärsuche*

```
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
  (sortedData, indexConversion) := sortExt (data);
  resultIndex := binarySearch (sortedData, 1, length (data), k);
  if resultIndex = 0 then return 0;
  return indexConversion[resultIndex]
end {search}
```

Prozedur **`sortExt`** unterscheidet sich von Prozedur **`sort`** dadurch, dass zusätzlich zum sortierten Feld **`sortedData`** noch ein Feld **`indexConversion`** zurückgegeben wird, in dem zu Index j der Index vermerkt ist, an dem das Element, das jetzt in **`sortedData[j]`** steht, vorher in **`data`** gestanden hat.
(**`sortedData[j] = data [indexConversion[j]]`**).

Entwurf von Algorithmen: Sortierproblem

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

Rekursive Formulierung des *Mergesort*:

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

Entwurf von Algorithmen: Sortierproblem

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

Iterative Formulierung des *Mergesort*:

```
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: Sortierproblem

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

Ausformulierung der Hilfsprozedur *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}
```

Bei assign muss der Parameter fromPos als call by reference deklariert werden !

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 ?

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

Bewertung von Algorithmen

Was wird bewertet ?

- benötigte Rechenzeit
- benötigter Speicherplatz

Welche Eigenschaften sollte eine Bewertung haben ?

- unabhängig von der Implementierung
- unabhängig vom eingesetzten Computer

Wie ist das möglich ?

Bewertung von Algorithmen

Der Schlüssel zum Erfolg: Die Turingmaschine

- von Alan Turing 1937 konstruierter theoretischer „Urcomputer“

Für jeden bis heute konstruierten Computer gilt:

- Wenn ein Problem der Größe n
  auf einer Turingmaschine f(n) Rechenschritte braucht,
  dann benötigt es auf einem anderen Computer c·f(n) Rechenschritte.

  Hierbei ist c eine Konstante, die nur vom Computer abhängt und für alle Algorithmen gilt.

Daraus abgeleitetes Grundprinzip:

- **Vernachlässige Konstante,**
  die nicht von der Problemgröße abhängen !

Bewertung von Algorithmen

Definition Komplexitätsklasse:

Ein Algorithmus gehört bzgl. der Rechenzeit (des Speicherplatzes) zur Komplexitätsklasse **O(f(n))**, wenn es eine Konstante c gibt, sodass gilt:

Die Rechenzeit (Der Speicherplatz) für ein Problem der Größe n benötigt maximal c·f(n) Rechenschritte (Speicherplätze).

- c darf nicht von der Problemgröße n abhängen.
- c darf vom Algorithmus abhängen.
- c darf vom Computer und von der Implementierung abhängen.
- O wird das Landau-Symbol genannt (nach Edmund Landau, 1877-1938)

Typische Komplexitätsklassen von Algorithmen:

O(1), O(log n), O(n), O(n log n), O (n^2), O(2^n)

O(P(n)), wobei P ein Polynom ist

Bewertung von Algorithmen

Bewertung von Algorithmen

Gegeben ein Algorithmus:
Finde die günstigste Komplexitätsklasse bzgl. **Laufzeit** und Speicherplatz:

- **im ungünstigsten Fall (worst case)**
- im Durchschnittsfall (average case)

Beispiele:

	Laufzeit:	Speicherplatz:
Lineare Suche:	O(n)	O(n)
„DC-Suche“:	O(n)	O(n)
Binärsuche:	O(log n)	O(n)
Selectionsort:	$O(n^2)$	O(n)
Mergesort:	O(n log n)	O(n)
Quicksort:	$O(n^2)$ **w.c.** O(n log n) **a.c.**	O(n)

Bewertung von Algorithmen

Bewertung von Problemen

Gegeben ein Problem:
Finde die günstigste Komplexitätsklasse, zu der es einen Algorithmus gibt, der das Problem ***im allgemeinen Fall*** löst.

Ein Problem gehört bzgl. der Rechenzeit (des Speicherplatzes) zur Komplexitätsklasse **Ω(f(n))**, wenn es eine Konstante c gibt, so dass gilt:

Die Rechenzeit (Der Speicherplatz) ***jedes*** Algorithmus für das Problem der Größe n benötigt ***mindestens*** c·f(n) Rechenschritte (Speicherplätze).

Beispiele:

	Laufzeit:	Speicherplatz:
Allgemeine Suche:	Ω(n)	Ω(n)
Allgemeines Sortieren:	Ω(n log n)	Ω(n)

NP-Vollständigkeit

NP-vollständige Probleme sind folgendermaßen charakterisiert:

- Das Problem ist auf einer (hypothetischen) **nichtdeterministischen** Turingmaschine in polynomialer Zeit (O(P(n) für ein Polynom P) lösbar.

- Jedes NP-vollständige Problem ist **genau dann**
  auf einer normalen Turingmaschine in polynomialer Zeit lösbar,
  wenn *alle* NP-vollständigen Probleme
  auf einer normalen Turingmaschine in polynomialer Zeit lösbar sind.

Offene Frage der Informatik:

1) Sind NP-vollständige Probleme auf einer normalen Turingmaschine in polynomialer Zeit lösbar ?

2) Gehören sie zu einer Komplexitätsklasse Ω(f(n)),
   wobei f(n) stärker wächst als jedes Polynom P(n) ?

Was ist für die Klausur relevant ?

Vorlesung 1-10: Logik, Programmverifikation

siehe Ende von Vorlesung 10

Vorlesung 11-12: Entwurf und Analyse von Algorithmen

Grundideen der Algorithmen Lineare Suche, Binärsuche, Selectionsort und Mergesort, Kenntnis der Komplexitätseigenschaften dieser Algorithmen, Begründung, warum Komplexitätsklassen ein gutes Maß zur Bewertung von Algorithmen sind (zum Beispiel im Vergleich zu iterativ / rekursiv).

Das war die Vorlesung
Grundlagen der Programmierung

In folgenden Spezialvorlesungen könnten wir uns wieder begegnen:

Wissensbasierte Systeme (B_Inf, B_TInf, B_WInf 5, Master)

Künstliche Intelligenz (Master)

Verteilte Systeme (Master)

Service-orientierte SW-Architekturen (Master)

Objektorientierte Datenbanken (Master)