Theoretische Informatik II

Einheit 8.2

Abschätzung der Komplexität von Algorithmen

1. Suchverfahren

2. Sortieralgorithmen

Obere Schranken für die Laufzeit

- **Analyse auf Ebene der Algorithmen**
  - Algorithmische Elementaroperationen gelten als ein Schritt
  - Meist konstanter Expansionsfaktor bei Übersetzung in Maschinensprache
  - `+`, `-`, `*`,`/`,... Einzelschritte, wenn Zahlengröße beschränkt (z.B. 64-bit)
  - Höherer Aufwand bei beliebig großen Zahlen
- **Analyse abstrakter sequentieller Algorithmen**
  - Asymptotische Komplexität ist unabhängig von Programmiersprache
  - Parallele/nichtdeterministische Maschinen haben evtl. bessere Laufzeit

Sequentielle Suche

Teste, ob eine Zahl x in einer Liste L vorkommt

- **Durchsuche Liste L von links nach rechts**

```
function search_seq(x,L) ≡
   found := false;
   for i = 1 to length(L) do
      if L[i]=x then found:=true
   od;
   return found;
```

Verfahren ist anwendbar auf beliebige Listen

- **Laufzeitanalyse**
  - Eine Operation für Initialisierung `found:=false`
  - Je 2 Operationen pro Element von L in der `for`-Schleife
  - Eine Operation für Ausgabe des Ergebnisses
  - Insgesamt $2n+2$ Schritte, wenn n die Größe der Liste L ist

$\mapsto$ **Sequentielle Suche ist in $\mathcal{O}(n)$**

Binäre Suche

Nur anwendbar, wenn Liste L geordnet ist

- **Teste mittleres Element; suche dann rechts oder links**

```
function search_bin(x,L) ≡
   function search_bound(x,L,left,right) ≡
      if left>right then return false
         else
            mid := (left+right) div 2;
            if x<L[mid] then search_bound(x,L,left,mid-1)
               elseif x>L[mid] then search_bound(x,L,mid+1,right)
               else return true
            fi;
   return search_bound(x,L,1,length(L))
```

- **Grobe Laufzeitanalyse**
  - Konstante Anzahl von Operationen pro Aufruf von $\mathtt{search}_{bound}$
  - Wie oft wird $\mathtt{search}_{bound}$ aufgerufen?

Binäre Suche – Analyse

```
function search_bin(x,L) ≡
   function search_bound(x,L,left,right) ≡
      if left>right then return false
         else
            mid := (left+right) div 2;
            if x<L[mid] then search_bound(x,L,left,mid-1)
               elseif x>L[mid] then search_bound(x,L,mid+1,right)
               else return true
            fi;
   return search_bound(x,L,1,length(L))
```

Abstand von `left` und `right` halbiert sich pro Aufruf (mit Abrundung)

Anzahl von Operationen pro Aufruf von $\texttt{search}_{bound}$ ist eine Konstante k

Abstand zu Beginn ist $n{-}1$ (n ist die Größe der Liste L)

$\texttt{search}_{bound}$ terminiert bei Erfolg oder wenn Abstand Null ist

Lösung der Gleichung $time(n) = k{+}time(\lfloor n/2 \rfloor)$ ist $time(n) = k * \log_2 n$

$\Downarrow$

Binäre Suche ist in $\mathcal{O}(\log_2 n)$

Sortierverfahren

- **Ordne Elemente in aufsteigender Reihenfolge**
  - Geordnete Listen unterstützen effizienten Zugriff auf Elemente
  - Eine der häufigsten Operationen in der Programmierung
- **Viele Verfahren bekannt**
  - **Insertion Sort**: Einfügen des Listenanfangs in geordnete Teilliste
  - **Selection Sort**: Auswahl des jeweils kleinsten Elements als Listenanfang
  - **Bubblesort**: Austauschen benachbarter Elemente
  - **Quicksort**: Aufteilung nach Größe, Sortieren der entstehenden Teillisten
  - **Mergesort**: Aufteilen in Teillisten, Sortieren und Mischen der Teillisten
  - **Mergesort (II)**: Identifizieren und Mischen geordneter Teillisten

  Auswahl des 'besten' Verfahrens hängt von Größe des Problems ab

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Beispiel einer Sortierung mit Bubblesort**

1	2	5	6	7	8	9	$\surd$

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort - Laufzeitanalyse

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Feste Anzahl von Operationen im Schleifenrumpf**
  - Vergleich benachbarter Elemente
  - ggf. Austauch unter Verwendung einer Hilfsvariablen
- **Anzahl Schleifen abhängig von Listengröße n**
  - Innere Schleife wird jeweils genau `upper`-mal durchlaufen
  - Insgesamt n-1 + n-2 + ... + 2 + 1 = $n*(n\text{-}1)/2$ Durchläufe

$$\Downarrow$$

Bubblesort ist in $\mathcal{O}(n^2)$

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

  – Länge der Läufe wächst – Anzahl halbiert sich

- **Wiederhole bis Folge geordnet**

7	8	9	1	2	5	6
1	2	5	6	7	8	9

  – Liste ist eine einzige (komplett) geordnete Teilfolge $\surd$

Analyse des Verfahrens

- **Abstrakte Skizze reicht für Laufzeitanalyse**
- **Verschmelzen ist in $\mathcal{O}(n)$**
  - Folge wird jeweils komplett durchlaufen
- **Verschmelzen halbiert Anzahl der Läufe**
  - Je zwei Läufe werden zu einem gemischt
  - Nach maximal $\log_2 n$ Verschmelzungen bleibt ein einziger Lauf übrig
    d.h. die Liste ist sortiert

$$\Downarrow$$

Sortieren durch Verschmelzen ist in $\mathcal{O}(n * \log_2 n)$