Kapitel III Algorithmen und Datenstrukturen

1. Analyse von Algorithmen

Wir wollen die Ressourcen bestimmen, die, in Abhängigkeit von der Eingabe, ein Algorithmus benötigt, z.B.

1. Laufzeit
2. Speicherplatz
3. Anzahl Prozessoren
4. Programmlänge
5. Tinte
6. ...

Beispiel 159

Prozedur für Fakultätsfunktion

func $fak(n)$
  $m := 1$
  for $i := 2$ **to** n **do**
    $m := m * i$
  do
  return (m)

Diese Prozedur benötigt $O(n)$ Schritte bzw. arithmetische Operationen.

Jedoch: die Länge der Ausgabe ist etwa

$$\lceil \log_2 n! \rceil = \Omega(n \log n) \text{ Bits.}$$

Bemerkung:

Um die Zahl $n \in \mathbb{N}_0$ in Binärdarstellung hinzuschreiben, benötigt man

$$\ell(n) := \begin{cases} 1 & \text{für } n = 0 \\ 1 + \lfloor \log_2(n) \rfloor & \text{sonst} \end{cases}$$
$$= \begin{cases} 1 & \text{für } n = 0 \\ \lceil \log_2(n+1) \rceil & \text{sonst} \end{cases}$$

Um die Notation zu vereinfachen, vereinbaren wir im Zusammenhang mit Komplexitätsabschätzungen

$$\log(0) := 0\,.$$

1.1 Referenzmaschine

Wir wählen als Referenzmaschine die Registermaschine (engl. random access machine, RAM) oder auch WHILE-Maschine, also eine Maschine, die WHILE-Programme verarbeiten kann, erweitert durch

- IF ... THEN ... ELSE ... FI
- Multiplikation und Division
- indirekte Adressierung
- arithmetische Operationen wie $\sqrt{n}$, $\sin n$,...

1.2 Zeit- und Platzkomplexität

Beim Zeitbedarf zählt das uniforme Kostenmodell die Anzahl der von der Registermaschine durchgeführten Elementarschritte, beim Platzbedarf die Anzahl der benutzten Speicherzellen.

Das logarithmische Kostenmodell zählt für den Zeitbedarf eines jeden Elementarschrittes

$$\ell(\text{größter beteiligter Operand}),$$

beim Platzbedarf

$$\sum_x \ell(\text{größter in } x \text{ gespeicherter Wert}) \,,$$

also die maximale Anzahl der von allen Variablen x benötigten Speicherbits.

Beispiel 160

Wir betrachten die Prozedur

func $dbexp(n)$
  $m := 2$
  for $i := 1$ **to** n **do**
    $m := m^2$
  do
  return (m)

Die Komplexität von $dbexp$, die $n \mapsto 2^{2^n}$ berechnet, ergibt sich bei Eingabe n zu

	Zeit	Platz
uniform	$\Theta(n)$	$\Theta(1)$
logarithmisch	$\Theta(2^n)$	$\Theta(2^n)$

Bemerkung:
Das (einfachere) uniforme Kostenmodell sollte also nur verwendet werden, wenn alle vom Algorithmus berechneten Werte gegenüber den Werten in der Eingabe nicht zu sehr wachsen, also z.B. nur polynomiell.

1.3 Worst Case-Analyse

Sei A ein Algorithmus. Dann sei

$$T_A(x) := \text{Laufzeit von } A \text{ bei Eingabe } x\,.$$

Diese Funktion ist i.A. zu aufwändig und zu detailliert. Stattdessen:

$$T_A(n) := \max_{|x|=n} T_A(x) \quad (= \text{maximale Laufzeit bei Eingabelänge } n)$$

1.4 Average Case-Analyse

Oft erscheint die Worst Case-Analyse als zu pessimistisch. Dann:

$$T_A^{\text{ave}}(n) = \frac{\sum_{x;\ |x|=n} T_A(x)}{|\{x;\ |x| = n\}|}$$

oder allgemeiner

$$\begin{aligned} T_A^{\text{ave}}(n) &= \sum T_A(x) \cdot \Pr\{x \mid |x| = n\} \\ &= \mathbf{E}_{|x|=n}\left[T_A(x)\right] \ , \end{aligned}$$

wobei eine (im Allgemeinen beliebige) Wahrscheinlichkeitsverteilung zugrunde liegt.

Bemerkung:
Wir werden Laufzeiten $T_A(n)$ meist nur bis auf einen multiplikativen Faktor genau berechnen, d.h. das genaue Referenzmodell, Fragen der Implementierung, usw. spielen dabei eine eher untergeordnete Rolle.

2. Sortierverfahren

Unter einem Sortierverfahren versteht man ein algorithmisches Verfahren, das als Eingabe eine Folge $a_1, \ldots, a_n$ von n Schlüsseln $\in \Sigma^*$ erhält und als Ausgabe eine auf- oder absteigend sortierte Folge dieser Elemente liefert. Im Folgenden werden wir im Normalfall davon ausgehen, dass die Elemente aufsteigend sortiert werden sollen. Zur Vereinfachung nehmen wir im Normalfall auch an, dass alle Schlüssel paarweise verschieden sind.

Für die betrachteten Sortierverfahren ist natürlich die Anzahl der Schlüsselvergleiche eine untere Schranke für die Laufzeit, und oft ist letztere von der gleichen Größenordnung, d.h.

$$\text{Laufzeit} = O(\text{Anzahl der Schlüsselvergleiche}) .$$

2.1 Selection-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus SelectionSort

for $i := n$ **downto** 2 **do**
  $m :=$ Index des maximalen Schlüssels in $A[1..i]$
  vertausche $A[i]$ und $A[m]$
od

Nach Beendigung von SelectionSort ist das Feld A aufsteigend sortiert.

Satz 161

SelectionSort *benötigt zum Sortieren von* n *Elementen genau* $\binom{n}{2}$ *Vergleiche.*

Beweis:

Die Anzahl der Vergleiche (zwischen Schlüsseln bzw. Elementen des Feldes A) zur Bestimmung des maximalen Schlüssels in $A[1..i]$ ist $i-1$.

Damit ergibt sich die Laufzeit von SelectionSort zu

$$T_{\text{SelectionSort}} = \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}.$$

□

2.2 Insertion-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus InsertionSort

for $i := 2$ **to** n **do**
  $m :=$ Rang ($\in [1..i]$) von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$
  $a := A[i]$
  schiebe $A[m..i-1]$ um eine Position nach rechts
  $A[m] := a$
od

Nach Beendigung von InsertionSort ist das Feld A aufsteigend sortiert.

Der Rang von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$ kann trivial mit $i-1$ Vergleichen bestimmt werden. Damit ergibt sich

Satz 162

InsertionSort *benötigt zum Sortieren von* n *Elementen maximal* $\binom{n}{2}$ *Vergleiche.*

Beweis:

Übungsaufgabe! □

Die Rangbestimmung kann durch binäre Suche verbessert werden. Dabei benötigen wir, um den Rang eines Elementes in einer k-elementigen Menge zu bestimmen, höchstens

$$\lceil \log_2(k+1) \rceil$$

Vergleiche, wie man durch Induktion leicht sieht.

Satz 163

InsertionSort *mit binärer Suche für das Einsortieren benötigt zum Sortieren von n Elementen maximal*

$$n \lceil \mathrm{ld}\, n \rceil$$

Vergleiche.

Beweis:

Die Abschätzung ergibt sich durch einfaches Einsetzen. □

Achtung: Die Laufzeit von InsertionSort ist dennoch auch bei Verwendung von binärer Suche beim Einsortieren im schlechtesten Fall $\Omega(n^2)$, wegen der notwendigen Verschiebung der Feldelemente.

Verwendet man statt des Feldes eine doppelt verkettete Liste, so wird zwar das Einsortieren vereinfacht, es kann jedoch die binäre Suche nicht mehr effizient implementiert werden.

2.3 Merge-Sort

Sei wiederum eine Folge $a_1, \ldots, a_n$ von Schlüsseln im Feld $A[1..n]$ abgespeichert.

Algorithmus MergeSort

proc merge (r, s) **co** sortiere $A[r..s]$**oc**
  if $s \leq r$ **return fi**
  merge$(r, \lfloor \frac{r+s}{2} \rfloor)$; merge$(\lfloor \frac{r+s}{2} \rfloor + 1, s)$
  verschmelze die beiden sortierten Teilfolgen
end

merge$(1, n)$

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Satz 164

MergeSort *sortiert ein Feld der Länge* n *mit maximal* $n \cdot \lceil \mathrm{ld}(n) \rceil$ *Vergleichen.*

Beweis:

In jeder Rekursionstiefe werde der Vergleich dem kleineren Element zugeschlagen. Dann erhält jedes Element pro Rekursionstiefe höchstens einen Vergleich zugeschlagen. □