Universität Konstanz
Fachbereich Informatik & Informationswissenschaft
Stephan Heidinger

Algorithemn & Datenstrukturen
Wintersemester 2011/2012
30. Januar 2012

Find any errors? Please send them back, I want to keep them!

1 Einführung

1.1 Auswahlproblem

Ziel: „Bestimme das k. kleinste Element von n Elementen"

Spezialfälle: $k = \begin{cases} 1 & Minimumsuche \\ n & Maximumsuche \\ \lfloor \frac{n}{2} \rfloor & Median \end{cases}$

Verschiedene Ansätze:

1. Auswahl nach Sortieren
2. Wiederholte Minimumsuche
3. Aktualisierung einer vorläufigen Lösung
4. Nutzen von Standartbibliotheken

Bewertung ist schwer, bei 1 und 4 muss mehr über die Implementierung bekannt sein, bei allen Varianten muss zudem mehr über die Eingabe bekannt sein.

1.2 Maschinenmodell

Wie soll bewertet werden? Laufzeit in Sekunden? Hängt massgeblich ab von:

- Programmiersprache
- Rechner (Aufbau, Taktfrequenz, Speicher, ...)
- Eingabedaten

Müssen Bewertung unabhängig davon finden ⇒ **Zählen von *elementaren Schritten***

Random Access Machine

Maschinenmodell mit:

- endliche Zahl an Speicherzellen für Programm
- abzählbar endliche Zahl von Speicherzellen für Daten

- endliche Zahl von Registern
- arithmetisch-logische Einheit (ALU)

Anweisungen:

- Transportbefehle (Laden, Verschieben, Speichern)
- Sprungbefehle (bedingt, unbedingt; → Schleifen, Rekursionen)
- arithmetische und logische Vernküpfungen

1.3 Komplexität

Beschreiben Komplexität durch.

Laufzeit: Anahl Schritte (asymptotisch)

Speicherbedarf: Anzahl benutzter Speicherzellen

1.3.1 Definitionen

1. höchstens so schnell wachsen wie f, langsamer und gleich als f

$$\mathcal{O}(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \;\; \begin{array}{c} \text{es gibt Konstanten } c, n_0 > 0 \text{ mit} \\ |g(n)| \leq c \cdot |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

2. mindestens so schnell wachsen wie f, schneller und gleich als f

$$\Omega(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \;\; \begin{array}{c} \text{es gibt Konstanten } c, n_0 > 0 \text{ mit} \\ c \cdot |g(n)| \geq |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

3. genauso so schnell wachsen wie f

$$\Theta(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \;\; \begin{array}{c} \text{es gibt Konstanten } c_1, c_2, n_0 > 0 \text{ mit} \\ c_1 \leq \frac{|g(n)|}{|f(n)|} \leq c_2 \text{ für alle } n > n_0 \end{array} \right\}$$

4. die gegenüber f verschwinden, langsamer als f

$$o(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \;\; \begin{array}{c} \text{zu jedem } c > 0 \text{ex. } \textit{ein } n_0 > 0 \text{ mit} \\ c \cdot |g(n) \leq |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

5. denen gegenüber f verschwindet, schneller als f

$$\omega(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \begin{array}{c} \text{zu jedem } c > 0\text{ex. } \textit{ein } n_0 > 0 \text{ mit} \\ |g(n) \geq c \cdot |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

$$\log_a n \leq \sqrt{n} \leq n^2 \leq n^3 \leq 1,1^n \leq 2^n \leq n! \leq n^n$$

1.3.2 Satz

1. $g \in \mathcal{O}(f)$ genau dann, wenn $f \in \Omega(g)$
   $g \in \Theta(f)$ genau dann, wenn $f \in \Theta(g)$
2. $\log_b n \in \Theta(\log_2 n)$ für alle $b > 1$
   „Die Basis des Logarithmus spielt für das Wachstum keine Rolle“
3. $(\log_2 n)^d \in o(n^\varepsilon)$ für alle $d \in \mathbb{N}_0, \varepsilon > 0$
   „Logarithmen wachsen langsamer als alle Polynomialfunktionen “
4. $n^d \in o((1+\varepsilon)^n)$ für alle $d \in \mathbb{N}_0, \varepsilon > 0$
   „ Exponentielles Wachstum ist immer schneller als polynomielles “
5. $b^n \in o((b+\varepsilon)^n)$ für alle $b \geq 1, \varepsilon > 0$
   „ Jede Verringerung der Basis verlangt exponentielles Wachstum“
6. $\binom{a}{b} \in \Theta(n^k)$
7. $H_n := \sum_{k=1}^{n} \frac{1}{k} = \ln n + \mathcal{O}(1)$ (harmonische Zahlen)
8. $n! = \sqrt{2\pi n} \cdot \left(\frac{n}{c}^n\right) \cdot \left(1 + \Theta\left(\frac{1}{n}\right)\right)$ (Stirlingformel)

2 Sortieren

2.0.3 Gütekriterien

- Laufzeit
- Speicherbedarf
- Stabilität (kein Vertauschen der Reihenfolge schon sortierter Elemente)
- u.U. getrennte Betrachtung von Anzahl Vergleiche $C(n)$ und Anzahl Umspeicherungen $M(n)$

2.1 SelectionSort

2.1.1 Algorithmus

for $i = 1, \ldots, n-1$ **do**
  $m \leftarrow i$
  for $j = i+1, \ldots, n$ **do**
    if $M[j] < M[m]$ **then**
      $m \leftarrow j$
    end if
  end forvertausche $M[i]$ und $M[m]$
end for

2.1.2 Laufzeit

$$\text{Anzahl Vergleiche:} \quad C(n) = n-1+n-2+\cdots+1 = \sum_{n-1}^{i=i} i = \frac{n-1)\cdot n}{2}$$

$$\text{Anzahl Umspeicherungen:} \quad M(n) = 3\cdot(n-1)$$

Laufzeit liegt damit in $\Theta(n^2)$

2.1.3 Stabilität

Da weiter vorne stehende Elemente hinter gleiche andere vertauscht werden können: **nicht stabil**

2.2 Divide & Conquer: QuickSort

Divide & Conquer:
Zerteile Aufgabe in kleinere Aufgaben & Löse diese rekursiv.

Idee: Wähle ein Element p („Pivot“) und teile die anderen Elemente der Eingabe auf in :

M_1**:** die höchstens kleineren Elemente

M_2**:** die größeren Elemente

Sortierung von M erhält man nun durch Hintereinanderschalten von M_1, p, M_2

2.2.1 Algorithmus

```
quicksort(M,l,r)
if l < r then
   i ← l + 1
   j ← r
   p ← M[l]
   while i ≤ j do

      while i ≤ j && M[i] ≤ p do
         i ← i + 1
      end while
      while i ≤ j && M[j] ≥ p do
         j ← j − 1
      end while
      if i < j then
         vertausche M[i] und M[j]
      end if
   end while
   if l < j then
      vertausche M[l] und M[j]
      quicksort(M, l, j − 1)
   end if
   if j < r then
      quicksort(M,j+1,r)
   end if
end if
```

2.2.2 Laufzeit

im besten Fall: $\Theta(n \log n)$

im schlechtesten Fall: $\Theta(n^2)$ (bereits sortierte Eingabe)

mittlere Laufzeit: $\Theta(n \log n)$

Eine zufällige Auswahl des Pivot führt zu einem Algorithmus, der im Mittel auch auf vorsortierten Eingaben schnell ist.

2.3 Divide & Conquer: MergeSort

MergeSort quasi umgekehrt zu QuickSort: triviale Aufteilung, linearer Aufwand bei Rekombination

2.3.1 Algorithmus

```
mergesort(M,l,r)
if l < r then
   m ← ⌊(l+r−1)/2⌋
   mergesort(M,l,m)
   mergesort(M,m+1,r)
   i ← l; j ← m + 1; k ← l
   while i ≤ m && j ≤ r do

      if M[i] ≤ M[j] then
         M'[k] ← M[i]
         i ← i + 1
      else
         M'[k] ← M[j]
         j ← j + 1
      end if
      k ← k + 1
   end while
   for h = i, . . . , m do
      M[k + (h − 1)] ← M[h]
   end for
   for h = l, . . . , k − 1 do
      M[h] ← M'[h]
   end for
end if
```

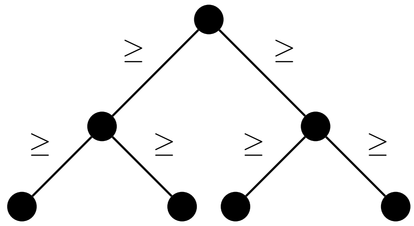
2.3.2 Laufzeit

Laufzeit ist in $\Theta(n \log n)$ (unabhängig von der Eingabe, dafür aber höherer Speicherbedarf)

2.4 HeapSort

2.4.1 Heap

Heap-Bedingung:
Für jeden Knoten gilt, dass der darin gespeicherte Wert nicht kleienr ist, als die beiden Werte in seinen Kindern.



Einfügen: Das neue Objekt wird hinten ins Array eingefügt, dann wird es solange mit seinem Vorgänger vertauscht, bis die *Heap-Bedingung* wiederhergestellt ist. **Laufzeit:** $\mathcal{O}(\log n)$

`insert`
$i \leftarrow n+1$
while $i > 1$ && $M\left[\lfloor \frac{i}{2} \rfloor\right] < a$ **do**
  $M[i] \leftarrow M\left[\lfloor \frac{i}{2} \rfloor\right]$
  $i \leftarrow \lfloor \frac{i}{2} \rfloor$
end while
$M[i] \leftarrow a$

ExtractMax: Zum Entfernen wird das erste Element entfernt. Das letzte Element des Arrays wird an die Spitze gefüllt und „versickert“ nun nach unten. Dies wird mittels „heapify“ erreicht.

$a \leftarrow M[i]$
$j \leftarrow 2 \cdot i$
while $j \leq r$ **do**

  if $j < r$ && $M[j+1] > M[j]$ **then**

  end if
  if $a < M[j]$ **then**
    $M[i] \leftarrow M[j]$
    $i \leftarrow j$
    $j \leftarrow 2 \cdot i$
  else
    $j \leftarrow r+1$
  end if
end while
$M[i] \leftarrow a$

2.4.2 Algorithmus

2.4.3 Laufzeit