Abschnitt: Algorithmendesign und Laufzeitanalyse

Definition Divide-and-Conquer Paradigma

Divide-and-Conquer Algorithmen verwenden die Strategien

1. **Divide:** Teile das Problem *rekursiv* in Subproblem gleicher Struktur auf. Sofern die Größe der Subprobleme hinreichend klein ist, verwende trivialen Algorithmus zum Lösen.
2. **Conquer:** Kombiniere Lösungen der Subprobleme zur Lösung des Ausgangsproblems.

Beispiele:

- Multiplikation von zwei n-Bit Zahlen mit Karatsuba-Methode.
  - Divide: Teile in Multiplikation von drei $\frac{n}{2}$-Bit Zahlen.
  - Conquer: Kombiniere die $\frac{n}{2}$-Bit Ergebnisse zur n-Bit Lösung.
- Multiplikation von zwei Polynomen vom Grad n mittels FFT.
  - Divide: Teile in zwei Polynome a_g, a_u vom Grad $\frac{n}{2}$.
  - Conquer: Kombiniere $a(x) = a_g(x^2) + x \cdot a_u(x^2)$.

Idee der Binären Suche

Problem Suche eines Elements

Gegeben: Sortiertes Array $a[1 \ldots n]$ mit $a[1] < \ldots < a[n]$, Element x

Gesucht: Index $i \in \mathbb{Z}_{n+1}$ mit $i = \begin{cases} j & \text{falls } x = a[j] \text{ für ein } j \in [n] \\ 0 & \text{sonst.} \end{cases}$

Lösung mittels Divide and Conquer

- Wir treffen die vereinfachende Annahme $n = 2^k$.
- **Divide:** Teile $a[1 \ldots n]$ in Teilarrays $a[1 \ldots \frac{n}{2}]$ und $a[\frac{n}{2} + 1 \ldots n]$.
- **Conquer:** Falls $x \leq a[\frac{n}{2}]$, ist x im linken Teilarray.
- Falls $x > a[\frac{n}{2}]$, ist x im rechten Teilarray.
- Das Problem wird rekursiv im korrekten Teilarray gelöst.
- Kombination erfordert die korrekte Verwaltung der Indizes.

Binäre Suche

Algorithmus BINÄRE-SUCHE

EINGABE: Array a aufsteigend sortiert, x, Intervallgrenzen ℓ, r

1. If $(\ell = r)$ then
   1. If $(x = a[r])$ return r;
   2. else return 0;
2. else
   1. $m \leftarrow \ell + \lfloor \frac{r-\ell}{2} \rfloor$;
   2. If $x \leq a[m]$ then BINÄRE-SUCHE(a, x, ℓ, m);
   3. else BINÄRE-SUCHE$(a, x, m+1, r)$;

AUSGABE: Index $i \in \mathbb{Z}_{n+1}$ mit $i = \begin{cases} j & \text{falls } x = a[j] \text{ für ein } j \in [n] \\ 0 & \text{sonst.} \end{cases}$

- Initialer Aufruf ist BINÄRE-SUCHE$(a, x, 1, n)$.
- Korrektheit folgt aus der Überlegung auf der vorigen Folie.

Laufzeit Binäre Suche

Satz Laufzeit BINÄRE-SUCHE

BINÄRE-SUCHE benötigt für Arrays der Länge $n = 2^k$ genau $k + 1 = \log_2 n + 1$ Elementvergleiche.

Beweis: per Induktion über k

- **IV** für $k = 0$, d.h. $n = 1$: Benötigen einen Vergleich in Schritt 1.1.
- **IS** $k - 1 \leftarrow k$: Wir benötigen einen Vergleich $x \leq a[m]$.
- Rekursion liefert Teilarray der Größe $\frac{n}{2} = 2^{k-1}$.
- Nach IA benötigen wir für das Teilarray $(k - 1) + 1$ Vergleiche.
- Damit erhalten wir insgesamt $k + 1$ Elementvergleiche.

Anmerkung für beliebiges n

- Wir erhalten $\lceil \log n \rceil$ durch Auffüllen von a auf die nächste 2er-Potenz.

Sortieren von n Elementen

Problem Sortieren von n Elementen

Gegeben: Array $a[1 \ldots n]$
Gesucht: $a[1 \ldots n]$ mit $a[1] \leq a[2] \leq \ldots \leq a[n]$

Idee: Sortiere $a[i]$ in $a[1 \ldots i-1]$ ein für $i = 2, \ldots, n$.

Algorithmus INSERTION-SORT

EINGABE: $a[1 \ldots n]$

1. For $i \leftarrow 2$ to n do
   1. merk $\leftarrow a[i]$; $j \leftarrow i-1$;
   2. While ($j > 0$ and merk$< a[j]$) do $a[j+1] \leftarrow a[j]$; $j \leftarrow j-1$;
   3. $a[j+1] \leftarrow$ merk;

AUSGABE: $a[1 \ldots n]$ aufsteigend sortiert

Korrektheit und Laufzeit INSERTION-SORT

- **Korrektheit:** Nach der i-ten Iteration ist $a[1 \ldots i]$ sortiert.
- D.h. für $i = n$ ist das Array $a[1 \ldots n]$ aufsteigend sortiert.

- **Laufzeit:** In der i-ten Iteration werden $\leq i - 1$ Vergleiche benötigt.
- Damit ist die Gesamtzahl der Vergleiche höchstens
$$\sum_{i=2}^{n} i - 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \mathcal{O}(n^2).$$
- Man beachte: Im best-case, d.h. für aufsteigend sortierte Arrays, werden nur $n - 1 = \mathcal{O}(n)$ Vergleiche benötigt.

Divide and Conquer Version von Insertion-Sort

- Wir formulieren Insertion-Sort mittels Divide and Conquer.
- **Divide:** Splitte $a[1 \ldots n]$ in $a[1 \ldots n-1]$ und $a[n]$.
- **Conquer:** Sortiere $a[1 \ldots n-1]$ rekursiv. Abbruch für $a[1]$.
- Kombination: Sortiere $a[i]$ in sortiertem $a[1 \ldots i-1]$ ein.
- Sei $T(i)$ die Anzahl der Vergleiche, um $a[i]$ einzusortieren.
- Dann gilt $T(i) \leq i-1$ mit naivem Verfahren.
- Damit erhalten wir wie zuvor

$$T(n) \leq T(n-1) + n - 1 \leq \ldots \leq T(1) + \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}.$$

- **Frage:** Was passiert, wenn man statt des naiven Verfahrens BINÄRE-SUCHE verwendet?
- Dann kann mit nur $\mathcal{O}(\log i)$ Vergleichen der korrekte Index j zum Einsortieren für $a[i]$ gefunden werden.
- Allerdings benötigt das Verschieben des Arrays Zeit $\mathcal{O}(i-j)$.

Das Mergesort Verfahren

Idee des Mergesort Verfahrens

- Mergesort verwendet eine Divide and Conquer Strategie.
- Wir treffen die vereinfachende Annahme $n = 2^k$.
- **Divide:** Teile a in zwei Teilarrays gleicher Größe.
- **Conquer:** Sortiere die Teilarrays rekursiv.
- Ein Abbruch der Rekursion erfolgt für Arraylänge 1.
- Kombination: Verschmelze, d.h. merge, zwei sortierte Teilarrays der Länge $\frac{n}{2}$ zu einem sortierten Array der Länge n.

Verschmelzen von zwei Teilarrays

Algorithmus MERGE

EINGABE: $a[\ell \ldots m]$, $a[m+1 \ldots r]$ jeweils aufsteigend sortiert

1. $p_1 \leftarrow \ell$; $p_2 \leftarrow m+1$; $i \leftarrow 1$;
2. While ($p_1 \leq m$ and $p_2 \leq r$)
   1. If ($a[p_1] < a[p_2]$) then $b[i] \leftarrow a[p_1]$; $p_1 \leftarrow p_1 + 1$;
   2. else $b[i] \leftarrow a[p_2]$; $p_2 \leftarrow p_2 + 1$;
   3. $i \leftarrow i + 1$;
3. If ($p_1 \leq m$) then kopiere $a[p_1 \ldots m]$ nach $b[i \ldots r - \ell + 1]$.
4. else kopiere $a[p_2 \ldots r]$ nach $b[i \ldots r - \ell + 1]$.
5. Kopiere $b[1 \ldots r - \ell + 1]$ nach $a[\ell \ldots r]$.

AUSGABE: $a[l \ldots r]$ aufsteigend sortiert

Korrektheit und Laufzeit von MERGE

- **Korrektheit:** Array b enthalte i Elemente.
- Dann gilt $b[i] \leq a[j]$ für $j = p_1, \ldots, m$ und $j = p_2, \ldots, r$.
- D.h. $b[1 \ldots i]$ ist aufsteigend sortiert und enthält die i kleinsten Elemente beider Teilarrays.
- Damit ist $b[1 \ldots r - \ell + 1] = a[\ell \ldots r]$ aufsteigend sortiert.

- **Laufzeit:** Wir benötigen höchstens $r - \ell$ Vergleiche.
- D.h. die Anzahl der Vergleiche ist linear in der Intervallänge von a.

Mergesort

Algorithmus MERGESORT

EINGABE: $a[1 \ldots n], \ell, r$

1. If $(\ell < r)$
   1. $m \leftarrow \ell + \lfloor \frac{r-\ell}{2} \rfloor$;
   2. MERGESORT(a, ℓ, m);
   3. MERGESORT$(a, m+1, r)$;
   4. MERGE$(a[\ell \ldots m], a[m+1 \ldots r])$;

AUSGABE: $a[1 \ldots n]$ aufsteigend sortiert

- **Korrektheit:** Folgt aus Korrektheit von MERGE.
- **Laufzeit:** Sei $T(n)$ die Anzahl der Vergleiche.
- Dann gilt $T(n) = 2 \cdot T(\frac{n}{2}) + \mathcal{O}(n)$ mit $T(1) = \mathcal{O}(1)$.
- Lösung der Rekursion liefert $T(n) = \mathcal{O}(n \log n)$.
- Man beachte, dass MERGESORT im best-case ebenfalls $\mathcal{O}(n \log n)$ Vergleiche benötigt.

Das Quicksort-Verfahren

Idee des Quicksort-Verfahrens

- **Divide:** Wähle sogenanntes Pivotelement $p = a[i]$ für ein $i \in [n]$.
- Partitioniere das Array $a[n]$ in zwei nicht-leere Teilarrays $a[1 \ldots j], a[j+1 \ldots n]$ mit Elementen kleiner bzw. größer als p.
- D.h. $a[k] \leq p$ für $k = 1, \ldots, j$ und $a[k] \geq p$ für $k = j+1, \ldots, n$.
- **Conquer:** Sortiere beide Teilarrays rekursiv.
- Kombination: $a[n] = a[1 \ldots j]a[j+1 \ldots n]$ ist bereits sortiert.

Algorithmus QUICKSORT

EINGABE: $a[1 \ldots n], \ell, r$

1. If $(\ell < r)$
   1. $j \leftarrow$ PARTITION(a, ℓ, r);
   2. QUICKSORT(a, ℓ, j);
   3. QUICKSORT$(a, j+1, r)$;

AUSGABE: $a[1 \ldots n]$ aufsteigend sortiert

Die Partitionierung eines Arrays

Algorithmus PARTITION

EINGABE: $a[\ell \ldots r]$

1. $p \leftarrow a[\ell]$; $i \leftarrow \ell - 1$; $j \leftarrow r + 1$;
2. While (TRUE)
   1. Repeat ($i \leftarrow i + 1$) until $a[i] \geq p$;
   2. Repeat ($j \leftarrow j - 1$) until $a[j] \leq p$;
   3. If ($i < j$) vertausche $a[i] \leftrightarrow a[j]$;
   4. else return j;

AUSGABE: j mit $a[\ell], \ldots, a[j] \leq p$ und $p \leq a[j+1], \ldots, a[r]$.

- **Korrektheit:** $a[\ell \ldots i-1]$ enthält stets nur Elemente $\leq p$.
- Teilarray $a[j+1 \ldots r]$ enthält stets nur Elemente $\geq p$.
- Ein Abbruch der While-Schleife erfolgt für $i \geq j$.
- **Laufzeit:** Wir benötigen maximal $r - \ell + 1$ Vergleiche.
- D.h. die Anzahl der Vergleiche ist linear in der Intervallänge von a.

Eigenschaften von Quicksort

Laufzeit:

- Im worst-case: $T(n) = T(1) + T(n-1) + \mathcal{O}(n) = \mathcal{O}(n^2)$.
- Im best-case: $T(n) = 2 \cdot T(\frac{n}{2}) + \mathcal{O}(n) = \mathcal{O}(n \log n)$.
- Im average-case: Man kann zeigen, dass $T(n) = \mathcal{O}(n \log n)$ bei zufälliger Wahl des Pivotelements gilt.

Vorteile:

- Die Konstanten in der $\mathcal{O}$-Notation sind klein.
- QUICKSORT sortiert *in place*, d.h. QUICKSORT benötigt keinen zusätzlichen Speicherplatz.
- MERGESORT: Verwendung von zusätzlichem Array zum Mergen.
- QUICKSORT ist in der Praxis oft der schnellste Algorithmus.