Wiederholung

- Endliche Körper K
 - K^* ist zyklisch.
 - K^* mit q Elementen hat $\phi(q-1)$ Generatoren
- Von ganzen Zahlen zu Polynomen
 - EEA für Polynome
 - Primzahlen versus irreduzible Polynome
- Endliche Körper mit $q=p^n$ Elementen, p prim
 - Sei K endlicher Körper mit p Elementen.
 - Sei $\pi(x)$ irreduzibles Polynome vom Grad n .
 - $\mathbb{F}_q = \mathbb{F}_p^n = K[x]/\pi(x)$ ist endlicher Körper mit q Elementen.

Beispiel $\mathbb{F}_4$

- Körper $\mathbb{F}_{2^2} = \mathbb{F}_4$:
 - Körpererweiterung des Grundkörpers $\mathbb{F}_2$
 - $\pi(x) = (x^2+x+1)$ ist irreduzibel über $\mathbb{F}_2$, wegen $\pi(0) = \pi(1) = 1$.
- Repräsentanten von $\mathbb{F}_4$: $\{0, 1, x, x+1\}$
 - Inverses von x : EEA($x, \pi(x)$) liefert $1 \cdot (x^2+x+1) + x \cdot (x+1) = 1$
 $\Rightarrow x \cdot (x+1) = 1 \pmod{\pi(x)}$
 - Alternativ: Löse Gleichungssystem in a, b im $\mathbb{F}_2[x]/(x^2+x+1)$:
 $(ax+b) \cdot (x+1) = 1 \Leftrightarrow (ax^2+(a+b)x+b) = 1$
 $\Leftrightarrow (a(x+1)+(a+b)x+b) = 1$
 $\Leftrightarrow (bx+(a+b))=1$
 $\Rightarrow b=0$ und $a+b=1$, d.h. $a=1$. D.h. x ist das Inverse von $x+1$.
- $\mathbb{F}_4$ hat $\phi(3)=2$ Generatoren: x und $x+1$.
 - $x^2=x+1$, $x^3=1$ und $(x+1)^2=x$, $(x+1)^3=1$

Existenz der Galoiskörper $\mathbb{F}_{p^n}$

Satz: Für jedes prime p und jedes $n \in \mathbb{N}$ gibt es ein irreduzibles Polynom vom Grad n über $\mathbb{F}_p$.

Korollar: Für jedes prime p und jedes $n \in \mathbb{N}$ gibt es einen Körper $\mathbb{F}_{p^n}$ mit p^n Elementen.

- Je zwei endliche Körper mit gleicher Ordnung sind isomorph.
(ohne Beweis)

Algorithmendesign: Divide & Conquer

Bisher schon verwendet:

- Multiplikation von zwei n -Bit Zahlen in $\mathcal{O}(n^{\log_2 3})$.
- Schnelle Fouriertransformation (FFT):
Auswerten eines Polynoms vom Grad n an n Stellen in $\mathcal{O}(n \log n)$.

Divide & Conquer Paradigma:

- **Divide:** Teile das Probleme in Subprobleme gleicher Struktur auf.
- **Conquer:**
 - Löse die Subprobleme rekursiv. Sofern die Größe der Subprobleme hinreichend klein ist, verwende trivialen Algorithmus.
 - *Kombiniere* die Lösung der Subprobleme zur Lösung des Originalproblems

Idee der Binären Suche

Gegeben:

- Array $a[1..n]$ mit $a[1] < a[2] < \dots < a[n]$
- Element x

Gesucht:

- i mit $x = a[i]$ oder 0 falls $x \neq a[i]$ für alle i

Vereinfachende Annahme: $n = 2^k$.

- Divide: Teile Array $a[1..n]$ in zwei Teilarrays $a[1..n/2]$ und $a[n/2+1..n]$ gleicher Größe.
- Conquer:
 - Falls $x \leq a[n/2]$, dann kann x nur im linken Teilarray sein.
 - Falls $x > a[n/2]$, dann kann x nur im rechten Teilarray sein.
 - Es genügt, das Problem im jeweiligen Teilarray zu lösen.
 - Kombination: Korrekte Verwaltung der Indizes.

Binäre Suche

Algorithmus Binäre-Suche

Initialer Aufruf: Binäre-Suche(a,1,n,x)

Eingabe: Aufsteigend sortiertes Array a, l, r, x

1. If (l=r) then
 1. if $x = a[r]$ then return r;
 2. else return 0;
2. else
 1. $m \leftarrow l + \lfloor (r-l)/2 \rfloor$;
 2. If $x \leq a[m]$ then Binäre-Suche(a,l,m,x)
 3. else Binäre-Suche(a,m+1,r,x)

Ausgabe: i mit $x=a[i]$ oder 0 falls $x \neq a[i]$ für alle i

Korrektheit: klar

Laufzeit Binäre Suche

Satz: Die binäre Suche benötigt für Arrays der Länge $n=2^k$ genau $k+1=\log n+1$ Elementvergleiche.

Induktion über k :

- $k=0$, d.h. $n=1$: Ein Vergleich in Schritt 1.1.
- $k-1 \rightarrow k$:
 - Ein Vergleich: $x \leq a[m]$
 - Teilen Array der Größe $n=2^k$ in Array der Größe $n/2=2^{k-1}$.
 - IV: Benötigen $(k-1)+1$ Vergleiche für neues Array
 - Insgesamt: $k+1 = \log n+1$ Vergleiche.

Für beliebiges n : $\lceil \log n \rceil + 1$ durch „Auffüllen“ auf nächste 2er-Potenz.

Sortieren von n Elementen

Problem: Sortiere Array $a[1..n]$ aufsteigend.

Algorithmus Insertion-Sort:

Eingabe: $a[1..n]$

1. For $i = 2$ to n do
 1. /* Sortiere $a[i]$ in $a[1..i-1]$ ein. */
 2. $\text{merk} \leftarrow a[i]; j \leftarrow i-1;$
 3. while ($j > 0$ and $\text{merk} < a[j]$) do
 1. $a[j+1] \leftarrow a[j]; j--;$
 4. $a[j+1] \leftarrow \text{merk};$

Ausgabe: $a[1..n]$ mit $a[1] \leq a[2] \leq \dots \leq a[n]$

Korrektheit:

- Im i -ten Schritt gilt $a[1] \leq \dots \leq a[i]$, d.h. für $i=n$ ist das Array sortiert.

Laufzeit:

- In der i -ten Iteration: höchstens $i-1$ Vergleiche
- Gesamtzahl Vergleiche: höchstens $\frac{1}{2} \cdot (n-1)n = \mathcal{O}(n^2)$
- Beachte: best-case $n-1 = \mathcal{O}(n)$ Vergleiche

Divide & Conquer Version

Formulierung von Insertion-Sort mittels Divide & Conquer:

- Divide: Splitte $a[1..n]$ in $a[1..n-1]$ und $a[n]$
- Conquer:
 - Sortiere $a[1..n-1]$ rekursiv. Abbruch für $a[1]$.
 - Kombiniere: Sortiere $a[n]$ in sortiertem $a[1..n-1]$ ein.

Laufzeit: Anzahl der Vergleiche: $T(1) = 0$ Vergleiche,

$$\begin{aligned} T(n) &\leq T(n-1) + n-1 &&\leq T(n-2) + n-2 + n-1 \\ &&&\leq \dots \\ &&&\leq T(1) + 1 + 2 + \dots + n-1 = \frac{1}{2} * n(n-1) \end{aligned}$$

Merge-Sort

- Merge-Sort: Lösung mittels Divide & Conquer Strategie
- Vereinfachende Annahme: $n=2^k$
- **Divide:**
Teile das Array der Länge n in zwei Teilarrays der Länge $n/2$.
- **Conquer:**
 - Sortiere die Teilarrays rekursiv. Abbruch für Arraylänge 1.
 - Kombiniere: Verschmelze (=merge) zwei sortierte Teilarrays zu einem sortierten Array.

Mergen von zwei Arrays

Algorithmus Merge

Eingabe: $a[l..m]$, $a[m+1..r]$ aufsteigend sortiert

1. $p1 \leftarrow l$; $p2 \leftarrow m+1$; $i \leftarrow 1$;
2. while ($p1 \leq m$ and $p2 \leq r$) do
 1. If ($a[p1] < a[p2]$) then $b[i] \leftarrow a[p1]$, $p1 \leftarrow p1+1$;
 2. else $b[i] \leftarrow a[p2]$; $p2 \leftarrow p2+1$;
 3. $i \leftarrow i+1$;
3. if ($p1 \leq m$) Kopiere $a[p1..m]$ nach $b[i..r-l+1]$.
4. else Kopiere $a[p2..r]$ nach $b[i..r-l+1]$.
5. Kopiere b nach $a[l..r]$.

Ausgabe: $a[l..r]$ aufsteigend sortiert.

Korrektheit:

- Array b enthalte i Elemente.
Dann gilt $b[i] \leq a[j]$ für $j=p1, \dots, m$ und $j=p2, \dots, r$

Laufzeit:

- Höchstens $r-l$ Vergleiche, d.h. linear in der Intervalllänge.

Mergesort

Algorithmus Mergesort

Eingabe: $a[1..n]$, l , r

1. If $l < r$ then
 1. $m \leftarrow l + \lfloor (r-l)/2 \rfloor$
 2. Mergesort(a , l , m);
 3. Mergesort(a , $m+1$, r);
 4. Merge($a[l..m]$, $a[m+1..r]$)

Ausgabe: $a[1..n]$ aufsteigend sortiert

Korrektheit: klar

Laufzeit: $T(n) = 2 \cdot T(n/2) + \mathcal{O}(n)$ mit $T(1) = \mathcal{O}(1)$

$\Rightarrow T(n) = \mathcal{O}(n \log n)$

Best-case ebenfalls $\Theta(n \log n)$.

Quicksort

- Divide:
 - Wähle Pivotelement $p=a[i]$ für ein $i \in [n]$.
 - Partitioniere das Array $a[1..n]$ in zwei nicht-leere Teilarrays $a[1..j]$, $a[j+1..n]$ mit
 - $a[k] \leq p$ für $k=1..j$
 - $a[k] \geq p$ für $k=j+1..n$
- Conquer:
 - Sortiere Teilarrays rekursiv.
 - Kombination der Teilarrays: nichts zu tun.

Algorithmus Quicksort

Initialer Aufruf: Quicksort($a, 1, n$)

Eingabe: $a[1..n]$, l , r

1. If ($l < r$) then
 1. $j \leftarrow \text{Partition}(a, l, r)$
 1. Quicksort(a, l, j)
 2. Quicksort($a, j+1, r$)

Ausgabe: $a[1..n]$ sortiert.

Partitionierung

Algorithmus Partition

Eingabe: $a[l..r]$

1. $x \leftarrow a[l]$; /*Pivotelement ist Element an linker Arraygrenze*/
2. $i \leftarrow l-1$; $j \leftarrow r+1$;
3. while TRUE do
 1. repeat ($i \leftarrow i+1$) until $a[i] \geq x$;
 2. repeat ($j \leftarrow j-1$) until $a[j] \leq x$;
 3. if ($i < j$) vertausche $a[i] \leftrightarrow a[j]$;
 4. else return j ;

Ausgabe: j mit $a[l], \dots, a[j] \leq x$ und $x \leq a[j+1], \dots, a[r]$

Korrektheit:

- $a[l..i-1]$ enthält stets nur Elemente $\leq x$.
- $a[j+1..r]$ enthält stets nur Elemente $\geq x$.
- Abbruch für $i \geq j$.

Laufzeit:

- Maximal $r-l+1$ Vergleiche, d.h. linear in der Intervalllänge

Eigenschaften von Quicksort

Laufzeit:

- worst-case: $T(n) = T(1) + T(n-1) + \mathcal{O}(n) = \mathcal{O}(n^2)$
- best-case: $T(n) = 2T(n/2) + \mathcal{O}(n) = \mathcal{O}(n \log n)$
- average-case:
Ebenfalls $\mathcal{O}(n \log n)$ bei zufälliger Wahl des Pivotelement.

Verhalten in der Praxis:

- Kleine Konstante in $\mathcal{O}$ -Notation, oft der schnellste Algorithmus.
- Benötigt keinen zusätzlichen Speicherplatz.