Freie Universität Berlin

# Paralleles Sortierung

Björn Rathjen Patrick Winterstein
Freie Universität Berlin

Proseminar Algorithmen, SS14

# Inhalt

Motivation

Grundlage des Sortierens
- Komparator

Sortiernetzwerk
- Aufbau
- Korrektheit
- Biton-Sortierer
- Odd-Even-Mergesort

Laufzeit
- Herleitung
- Vergleich mit Software sortieren

Zusammenfassung

Ausblick
- Anhang

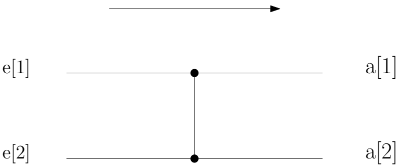
# Motivation : Allgemein

Sortieren
ist Grundlage für :

- Suche
- (Sortierung)
  - Listen
  - Wörterbücher
  - ...
- Ist dies auch effektiver / in Hardware möglich ?

# Aufbau

- 2 Eingänge
- vergleichender Baustein
- 2 Ausgänge



# Vergleichender Baustein (ii)

```
void comp(chan in1, in2, out1, out2 Comparer{}){
    a := <- in1;
    b := <- in2;

    if (a < b){
        out1 <- a;
        out2 <- b;
        return void;
    }
    out1 <- b;
    out2 <- a;
    return void;
}
```

# Erweiterung : Aufbau , Aufgabe

Aufbau :

- mehrere Eingabeleitungen (gleiche Anzahl an Ausgabeleitungen)
- mehrere vergleichende Schritte

# Erweiterung : Aufbau , Aufgabe

Aufbau :

- mehrere Eingabeleitungen (gleiche Anzahl an Ausgabeleitungen)
- mehrere vergleichende Schritte

Aufgabe :

- Resultat soll sortierte Ausgabe sein

# Erweiterung : Aufbau , Aufgabe

Aufbau :

- mehrere Eingabeleitungen (gleiche Anzahl an Ausgabeleitungen)
- mehrere vergleichende Schritte

Aufgabe :

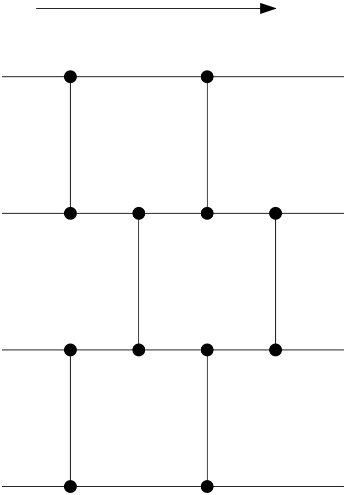
- Resultat soll sortierte Ausgabe sein

grundlegendes Prinzip :

- intuitiver Einsatz von Vergleichen
- Schrittweises sortieren

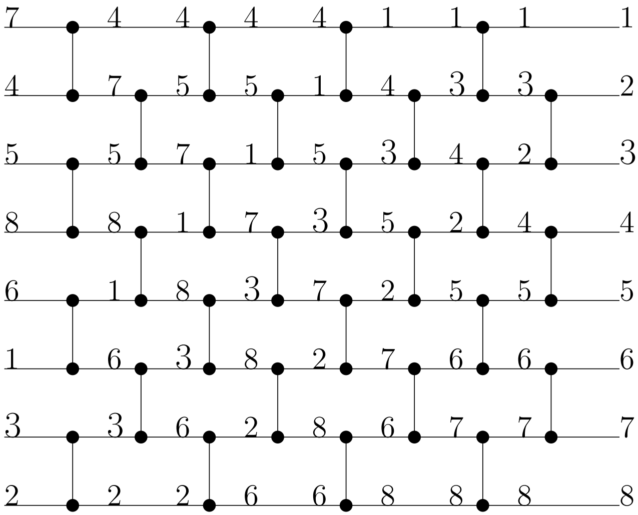
# naiv : grundlegendes Prinzip

Input

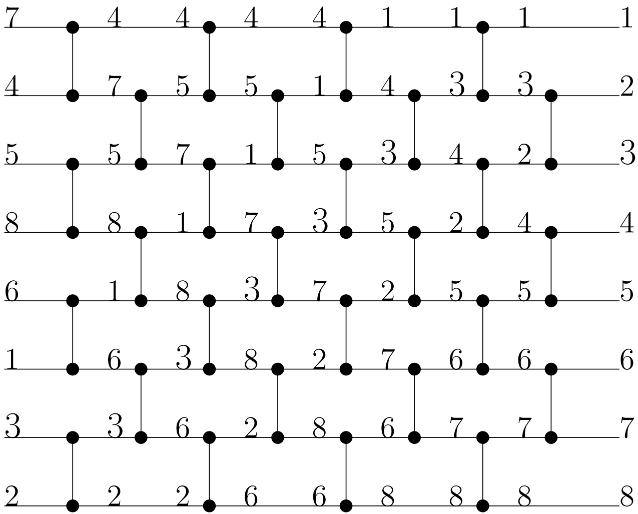


Output

# Demonstration



# Demonstration




Das Zahlenbeispiel funktioniert, aber funktioniert auch jedes andere ?

Lemma:
Wenn es eine Folge A gibt, die ein Sortiernetzwerk nicht sortiert, so existiert auch eine 0,1-Folge, die von diesem Netzwerk nicht sortiert wird.

# 0,1-Prinzip

Lemma:
Wenn es eine Folge A gibt, die ein Sortiernetzwerk nicht sortiert, so existiert auch eine 0,1-Folge, die von diesem Netzwerk nicht sortiert wird.

- Führen einen Beweis durch Widerspruch

# 0,1-Prinzip

Lemma:
Wenn es eine Folge A gibt, die ein Sortiernetzwerk nicht sortiert, so existiert auch eine 0,1-Folge, die von diesem Netzwerk nicht sortiert wird.

- Führen einen Beweis durch Widerspruch
- d.h : Wenn ein Algorithmus eine Eingabefolge nicht sortiert, so sortiert er alle 0,1-Folgen.

# 0,1-Prinzip

Lemma:
Wenn es eine Folge A gibt, die ein Sortiernetzwerk nicht sortiert, so existiert auch eine 0,1-Folge, die von diesem Netzwerk nicht sortiert wird.

- Führen einen Beweis durch Widerspruch
- d.h : Wenn ein Algorithmus eine Eingabefolge nicht sortiert, so sortiert er alle 0,1-Folgen.
- somit ist das Ziel eine 0,1-Folge zu finden, die nicht sortiert wird

# Beweis : 0,1-Prinzip

**Proof.**

- benötigen folgendes

□

# Beweis : 0,1-Prinzip

**Proof.**

- benötigen folgendes
  - Eingabefolge $E = e_0 \dots e_N$

$\square$

# Beweis : 0,1-Prinzip

**Proof.**

- benötigen folgendes
  - Eingabefolge $E = e_0 \dots e_N$
  - sortierte Folge $S = s_0 \dots s_N$

$\square$

# Beweis : 0,1-Prinzip

**Proof.**

- benötigen folgendes
  - Eingabefolge $E = e_0 \dots e_N$
  - sortierte Folge $S = s_0 \dots s_N$
  - unsortierte Ausgabefolge von E $U = u_0 \dots u_N$

$\square$

# Beweis : 0,1-Prinzip

**Proof.**

- benötigen folgendes
  - Eingabefolge $E = e_0 \ldots e_N$
  - sortierte Folge $S = s_0 \ldots s_N$
  - unsortierte Ausgabefolge von E $U = u_0 \ldots u_N$
  - k : (kleinster) Index an dem $u_k \neq s_k$

$\square$

# Beweis : 0,1-Prinzip

**Proof.**

- benötigen folgendes
  - Eingabefolge $E = e_0 \ldots e_N$
  - sortierte Folge $S = s_0 \ldots s_N$
  - unsortierte Ausgabefolge von E $U = u_0 \ldots u_N$
  - k : (kleinster) Index an dem $u_k \neq s_k$
- dann gilt :

  (1) $u_i = s_i \ \forall\, 0 \leq i < k$

□

## Beweis : 0,1-Prinzip

### Proof.

- benötigen folgendes
  - Eingabefolge $E = e_0 \ldots e_N$
  - sortierte Folge $S = s_0 \ldots s_N$
  - unsortierte Ausgabefolge von E $U = u_0 \ldots u_N$
  - k : (kleinster) Index an dem $u_k \neq s_k$
- dann gilt :
  (1) $u_i = s_i \; \forall \, 0 \leq i < k$
  (2) $u_r = s_k$ mit : $r > k$

$\square$

## Proof.

- man kann jede Zahlenfolge durch eine 0,1 Folge repräsentieren
  Konstante c (hier : $c = s_k$) und Zahlenfolge E mit den Elementen $e_i$

$$f(e) = \begin{cases} 0, & if \;\; e_i \leq s_k \\ 1, & if \;\; e_i > s_k \end{cases}$$

$\square$

## Proof.

- man kann jede Zahlenfolge durch eine 0,1 Folge repräsentieren Konstante c (hier : $c = s_k$) und Zahlenfolge E mit den Elementen $e_i$

$$f(e) = \begin{cases} 0, & if \;\; e_i \leq s_k \\ 1, & if \;\; e_i > s_k \end{cases}$$

- Ausgabe muss also wie folgt aussehen

$$00 \ldots\ldots 01 \;\; \ldots 0 \;\; \ldots$$

$\square$

## Proof.

- man kann jede Zahlenfolge durch eine 0,1 Folge repräsentieren Konstante c (hier : $c = s_k$) und Zahlenfolge E mit den Elementen $e_i$

$$f(e) = \begin{cases} 0, & if \;\; e_i \leq s_k \\ 1, & if \;\; e_i > s_k \end{cases}$$

- Ausgabe muss also wie folgt aussehen

$$00\ldots\ldots01_k\ldots0_r\ldots$$

$\square$

## Proof.

- man kann jede Zahlenfolge durch eine 0,1 Folge repräsentieren Konstante c (hier : $c = s_k$) und Zahlenfolge E mit den Elementen $e_i$

$$f(e) = \begin{cases} 0, & if \;\; e_i \leq s_k \\ 1, & if \;\; e_i > s_k \end{cases}$$

- Ausgabe muss also wie folgt aussehen

$$00 \ldots\ldots 01_k \ldots 0_r \ldots$$

$\Rightarrow$ 0,1-Folge ist nicht sortiert

$\square$

## Proof.

- man kann jede Zahlenfolge durch eine 0,1 Folge repräsentieren Konstante c (hier : $c = s_k$) und Zahlenfolge E mit den Elementen $e_i$

$$f(e) = \begin{cases} 0, & if \;\; e_i \leq s_k \\ 1, & if \;\; e_i > s_k \end{cases}$$

- Ausgabe muss also wie folgt aussehen

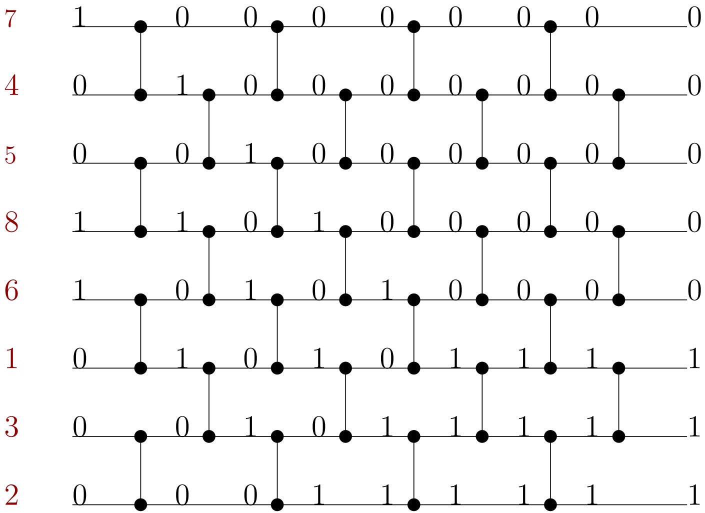
$$00 \ldots\ldots 01_k \ldots 0_r \ldots$$

⇒ 0,1-Folge ist nicht sortiert

⇒ Widerspruch zur Annahme

□

# 0,1- Beispiel


7 1 0 0 0 0 0 0 0 0
4 0 1 0 0 0 0 0 0 0
5 0 0 1 0 0 0 0 0 0
8 1 1 0 1 0 0 0 0 0
6 1 0 1 0 1 0 0 0 0
1 0 1 0 1 0 1 1 1 1
3 0 0 1 0 1 1 1 1 1
2 0 0 0 1 1 1 1 1 1

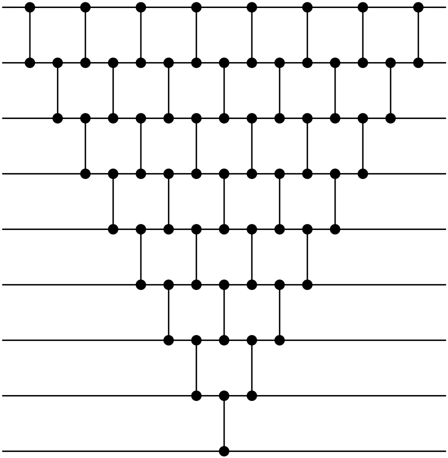
# 0,1-Prinzip

Vorteile des 0,1-Prinzips :

- einfach
  - sauberer Testcode
  - Sicherheit
- Anzahl der Testfälle sinkt

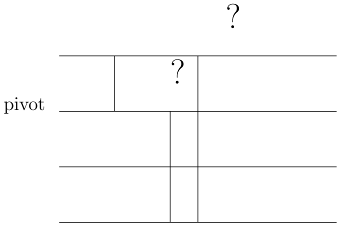
$$W^n \rightarrow 2^n$$

# Bubblesort im Komparatornetz

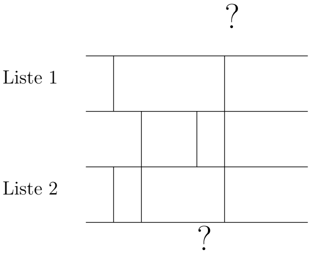


# Softwaresortieren im Hardwarenetz

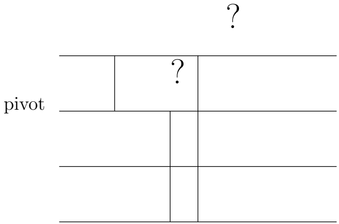
## Quicksort



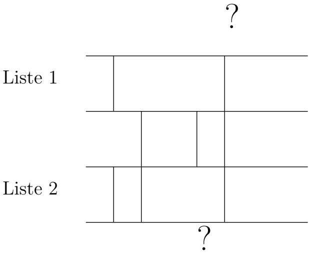

## Mergesort



Quicksort

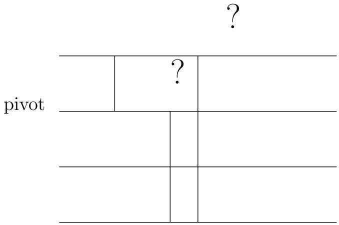



Mergesort

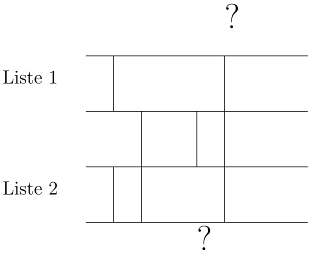



Quicksort : wo ist das Pivot Element ?
Mit welchem Element müssen wir nun vergleichen?

Quicksort




Mergesort




Quicksort : wo ist das Pivot Element ?
Mit welchem Element müssen wir nun vergleichen?
Mergesort : Wo ist nun das größte Element ?
welcher Vergleich kommt nun?)

# effektiveres Netzwerk

Aufgabe :

- Resultat soll sortierte Ausgabe sein
- soll effizient sein

## effektiveres Netzwerk

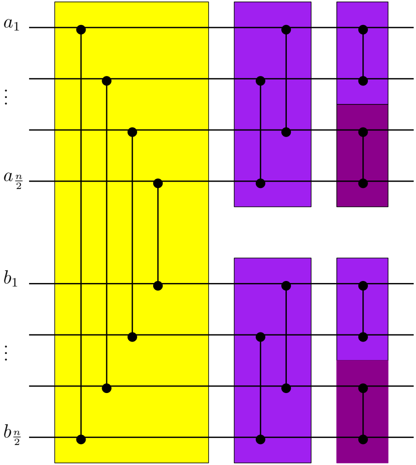
Aufgabe :

- Resultat soll sortierte Ausgabe sein
- soll effizient sein

grundlegendes Prinzip :

- intuitiver Einsatz von Vergleichen
  + Einbezug von Teile und Herrsche

# Übertragung auf Komparatornetzwerk

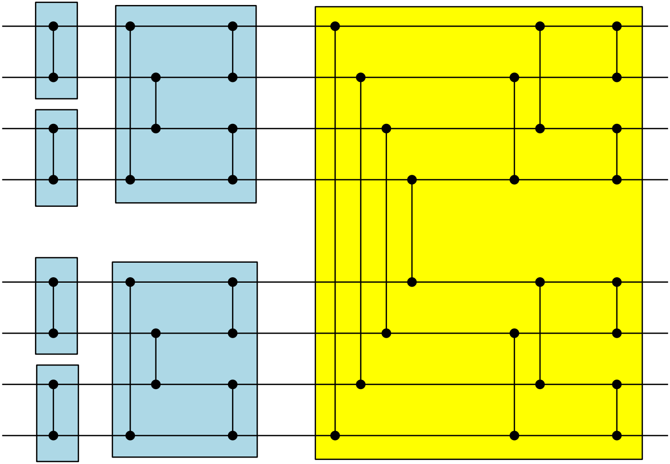


# Bitonmischer

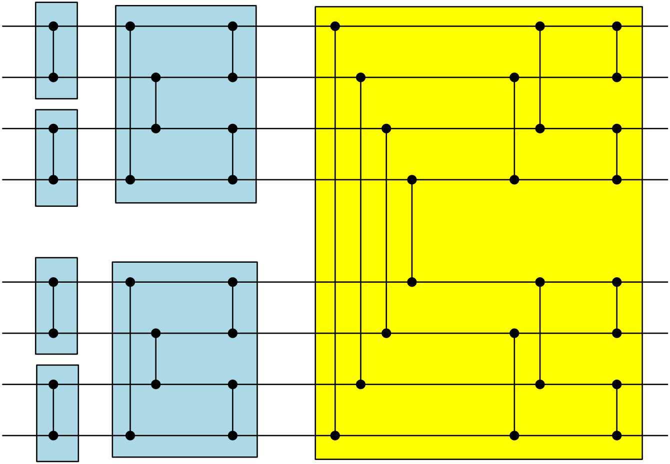
Ablauf :

- sortiert / mischt Eingabelisten
  - untere Hälfte alle größer als in oberer
- rekursiv die kleineren Listen
- Resultat eine Sortierte Liste

# Biton -Sortierer : Aufbau



# Biton -Sortierer : Aufbau



Beispiel an der Tafel

# odd even sort

Ablauf :

Ablauf :

- Voraussetzung : bekommen zwei sortierte Listen

Ablauf :

- Voraussetzung : bekommen zwei sortierte Listen
- trennen in geraden und ungeraden Index

# odd even sort

Ablauf :

- Voraussetzung : bekommen zwei sortierte Listen
- trennen in geraden und ungeraden Index
- fassen a(even) b(odd) = c und a(odd) b(even) = d zusammen (Resultat wird rekursiv sortiert)

Ablauf :

- Voraussetzung : bekommen zwei sortierte Listen
- trennen in geraden und ungeraden Index
- fassen a(even) b(odd) = c und a(odd) b(even) = d zusammen (Resultat wird rekursiv sortiert)
- sortierte c und d werden indexweise verschachtelt

# odd even sort

Ablauf :

- Voraussetzung : bekommen zwei sortierte Listen
- trennen in geraden und ungeraden Index
- fassen a(even) b(odd) = c und a(odd) b(even) = d zusammen (Resultat wird rekursiv sortiert)
- sortierte c und d werden indexweise verschachtelt
- aufeinander folgende Paare werden verglichen und in richtige Reihenfolge gebracht

# Beispiel : Odd-Even-Mergesort

Beispiel :

$$A = 2, 3, 4, 8$$

$$B = 1, 5, 6, 7$$

# Beispiel : Odd-Even-Mergesort

Beispiel :

$$A = 2, 3, 4, 8$$

$$B = 1, 5, 6, 7$$

$$A_e = 2, 4 \quad B_o = 5, 7 \quad \Rightarrow \quad C = 2, 4, 5, 7$$

# Beispiel : Odd-Even-Mergesort

Beispiel :

$$A = 2, 3, 4, 8$$

$$B = 1, 5, 6, 7$$

$$A_e = 2, 4 \quad B_o = 5, 7 \quad \Rightarrow \quad C = 2, 4, 5, 7$$

$$A_o = 3, 8 \quad B_e = 1, 6 \quad \Rightarrow \quad D = 1, 3, 6, 8$$

Beispiel :

$$A = 2, 3, 4, 8$$

$$B = 1, 5, 6, 7$$

$$A_e = 2, 4 \quad B_o = 5, 7 \quad \Rightarrow \quad C = 2, 4, 5, 7$$

$$A_o = 3, 8 \quad B_e = 1, 6 \quad \Rightarrow \quad D = 1, 3, 6, 8$$

nach dem Verschachteln

$$2, 1, 4, 3, 5, 6, 7, 8$$

# Beispiel : Odd-Even-Mergesort

Beispiel :

$$A = 2, 3, 4, 8$$

$$B = 1, 5, 6, 7$$

$$A_e = 2, 4 \quad B_o = 5, 7 \quad \Rightarrow \quad C = 2, 4, 5, 7$$

$$A_o = 3, 8 \quad B_e = 1, 6 \quad \Rightarrow \quad D = 1, 3, 6, 8$$

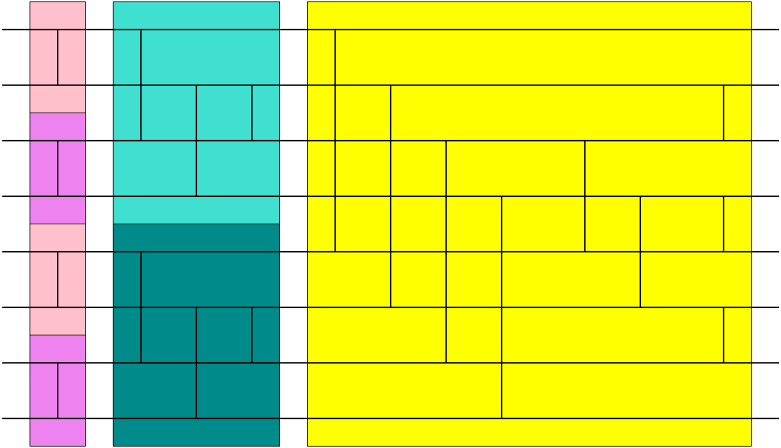
nach dem Verschachteln

$$2, 1, 4, 3, 5, 6, 7, 8$$

Resultat :

$$1, 2, 3, 4, 5, 6, 7, 8$$

# Sortiernetzwerk



# Wieso funktioniert das?

- Veranschaulichung mit dem 0,1-Pinzip
  - gleichmäßiges Aufteilen von Nullen und Einsen
  - somit alle nah beieinander

Motivation

Grundlage des Sortierens

Sortiernetzwerk

Laufzeit

- Herleitung
- Vergleich mit Software sortieren

Zusammenfassung

Ausblick

# Herleitung der Laufzeit

| Länge der Eingabe | Anzahl der Schritte |
| --- | --- |
| $2^1$ | |

# Herleitung der Laufzeit

| Länge der Eingabe | Anzahl der Schritte |
|---|---|
| $2^1$ | 1 |

# Herleitung der Laufzeit

| Länge der Eingabe | Anzahl der Schritte |
|---|---|
| $2^1$ | 1 |
| $2^2$ | |

| Länge der Eingabe | Anzahl der Schritte |
|---|---|
| $2^1$ | $1$ |
| $2^2$ | $1 + 2$ |

# Herleitung der Laufzeit

| Länge der Eingabe | Anzahl der Schritte |
| --- | --- |
| $2^1$ | $1$ |
| $2^2$ | $1+2$ |
| $2^k$ | |

# Herleitung der Laufzeit

| Länge der Eingabe | Anzahl der Schritte |
|---|---|
| $2^1$ | $1$ |
| $2^2$ | $1+2$ |
| $2^k$ | $1+2+3+...+k-1+k=\sum_{i=1}^{k} i$ |

# Herleitung der Laufzeit

| Länge der Eingabe | Anzahl der Schritte |
|---|---|
| $2^1$ | $1$ |
| $2^2$ | $1+2$ |
| $2^k$ | $1+2+3+...+k-1+k=\sum_{i=1}^{k} i$ |
| (kleiner Gauss) | |

# Herleitung der Laufzeit

| Länge der Eingabe | Anzahl der Schritte |
|---|---|
| $2^1$ | $1$ |
| $2^2$ | $1+2$ |
| $2^k$ | $1+2+3+...+k-1+k=\sum_{i=1}^{k} i$ |
| (kleiner Gauss) | $=\frac{1}{2}\cdot k\cdot(k+1)$ |

# Herleitung der Laufzeit

| Länge der Eingabe | Anzahl der Schritte |
|---|---|
| $2^1$ | $1$ |
| $2^2$ | $1+2$ |
| $2^k$ | $1+2+3+...+k-1+k=\sum_{i=1}^{k} i$ |
| (kleiner Gauss) | $=\frac{1}{2}\cdot k\cdot(k+1)$ |
| $(k=log_2 n)$ | |

# Herleitung der Laufzeit

| Länge der Eingabe | Anzahl der Schritte |
|---|---|
| $2^1$ | $1$ |
| $2^2$ | $1+2$ |
| $2^k$ | $1+2+3+...+k-1+k=\sum_{i=1}^{k} i$ |
| (kleiner Gauss) | $=\frac{1}{2}\cdot k\cdot(k+1)$ |
| $(k=log_2 n)$ | $\Rightarrow \frac{1}{2}\cdot \log_2 n\ (\log_2 n + 1)$ |

# Vergleich zu bekannten Softwareansätzen

- Schritte gegen Vergleiche
  - in jedem Schritt $\frac{n}{2}$ Komparatoren benötigt
    $\frac{n}{2} \cdot \left(\frac{1}{2} \cdot \log_2 n \ (\log_2 n + 1)\right)$
  - 1 Vergleicher bei Software benötigt
  - Laufzeit konstant
- Abhängigkeit von der Eingabe
- Bezug zum vorherigen Vergleich
- die andere Laufzeiten

| Algorithmus | Laufzeit | | |
|---|---|---|---|
| | best | worst | |
| Bubblesort | $O(n)$ | $O(n^2)$ | |
| Mergosort | $O(n \log n)$ | | |
| Quicksort | $O(n \log n)$ | $O(n^2)$ | |
| Netzwerk | $O(\log^2 n)$ | | |

# Vergleich zu bekannten Softwareansätzen

- Schritte gegen Vergleiche
  - in jedem Schritt $\frac{n}{2}$ Komparatoren benötigt
    $\frac{n}{2} \cdot \left(\frac{1}{2} \cdot \log_2 n \ (\log_2 n + 1)\right)$
  - 1 Vergleicher bei Software benötigt
  - Laufzeit konstant
- Abhängigkeit von der Eingabe
- Bezug zum vorherigen Vergleich
- die andere Laufzeiten

| Algorithmus | Laufzeit | | |
|---|---|---|---|
| | best | worst | avarage / normiert |
| Bubblesort | $O(n)$ | $O(n^2)$ | |
| Mergosort | $O(n \log n)$ | | $O(n \log n)$ |
| Quicksort | $O(n \log n)$ | $O(n^2)$ | $O(n \log n)$ |
| Netzwerk | $O(\log^2 n)$ | | $O(n \log^2 n)$ |

# Zusammenfassung

paralleles Sortieren ist

- schnell
- aber erfordert mehr Ressourcen
- problemabhängige Lösung (Eingabegröße)

# Ausblick

- aks-Netzwerke ($O(c \cdot n \cdot \log n)$
- $\rightarrow$ Hypercubes
- Paketrouting
- u.v.m.

# For Further Reading I


*Taschenbuch der Algorithmen*.
Springer Verlag , 2008.

Tom Leighton.
*Einführung in Parallele Algorithmen und Architekturen*
Gitter, Bäume und Hypercubes.
Thomsom Publisching , 1997.
3-8266-0248-X

Laufzeiten Sortieralgorithmen
www.wikipedia.de

Alternativer Ansatz zum Beweis des 0,1-Prinzip
http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/nulleins.htm

# Ende

Fragen, Anregungen?