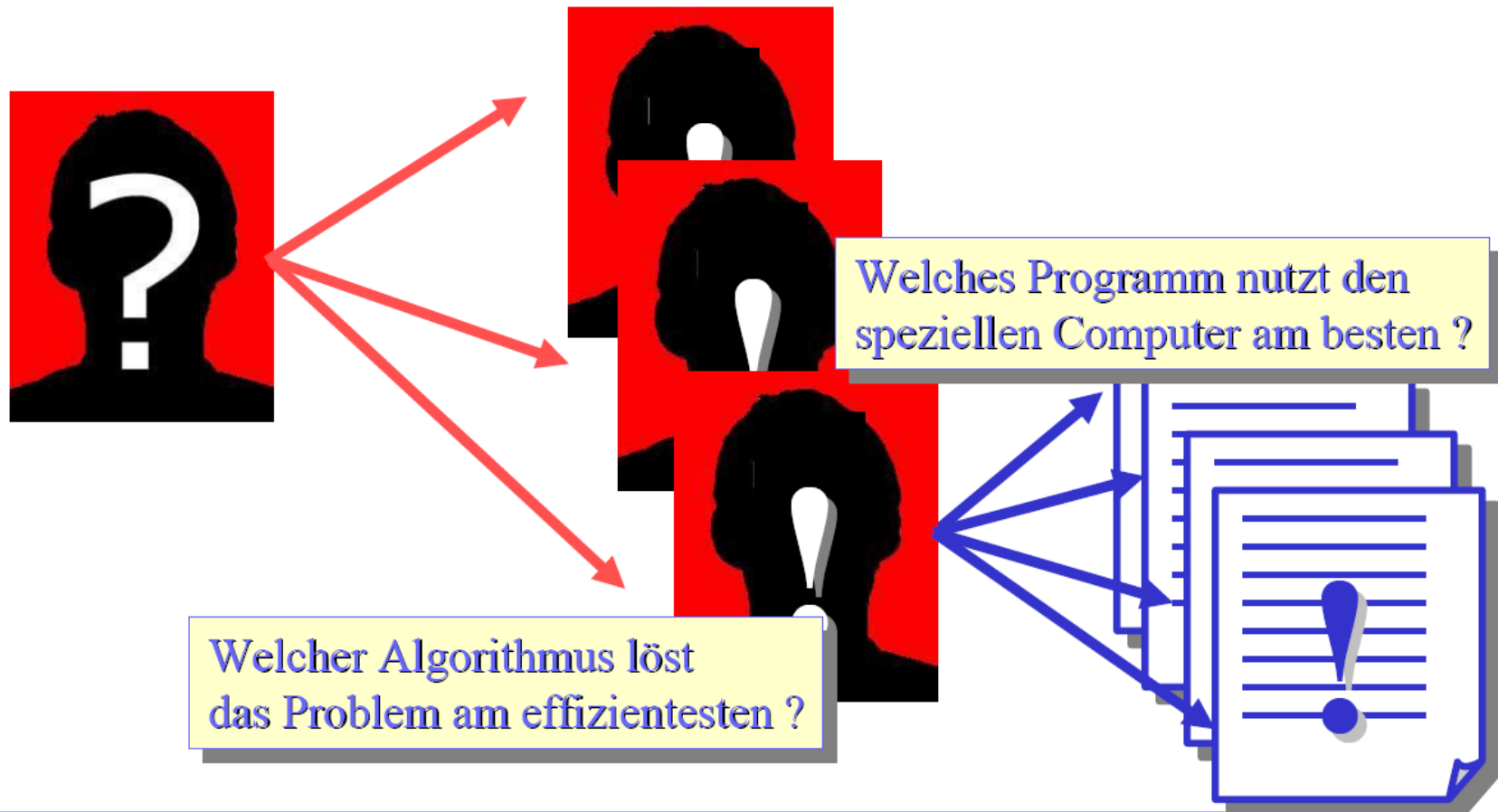


Konstruktion und Laufzeitanalyse von Algorithmen

Einige erste Beispiele

Problem – Algorithmus – Programm: Lösungsalternativen

- Jedes Problem lässt sich durch **verschiedene Algorithmen** lösen.
- Jeder Algorithmus lässt sich durch **verschiedene Programme** darstellen.



Wie gut ist ein Algorithmus?

- Mögliche Kriterien
 - ◆ Wie schnell wird das Ergebnis berechnet
 - ◊ Zeitkomplexität
 - ◆ Wieviel Speicherplatz wird benötigt
 - ◊ Platzkomplexität
- Wir werden und auf Zeitkomplexität konzentrieren
- Wie können Zeitkomplexitäten verglichen werden?
 - ◆ Erste idee: Vergleiche Laufzeiten von Implementierungen

Komplexität eines Algorithmus

- Es gibt wesentliche **Schwierigkeiten**, wenn Laufzeiten von Programmen statt von Algorithmen verglichen werden
 - ◆ Welche Realisierung des Algorithmus als Programm ist benutzt?
 - ◆ Wie ist das Programm kompiliert?
 - ◆ Welcher Computer wird zur Messung benutzt
- Analyse eines Algorithmus sollte davon unabhängig sein!

Aufwandsbestimmung

- **Zählung** der elementaren Operationen eines Algorithmus (in Abhängigkeit von der Größe der Eingabe) ist eine Möglichkeit der Aufwandsbestimmung (Komplexitätsbestimmung) des Algorithmus

Aufwandsbestimmung

● Beispiel

◆ Exakte Bestimmung der Komplexität

```
int f1 (int n) {  
    int res = 1; // Init  
    for(int j=1; j<n; j++)  
        for(int i=1; i<j; i++)  
            res = res * i;  
  
    return res;  
}
```

n	$Z(n)$	$V(n)$	$M(n)$	$I(n)$
1	2	0	0	0
2	3	1	0	1
3	5	3	1	3
4	8	6	3	6
5	12	10	6	10
6	17	15	10	15
7	23	21	15	21
8	30	28	21	28
9	38	36	28	36
10	47	45	36	45

Aufwandsbestimmung

● Weiteres Beispiel

```
int power(int m, int n)
{
    int res=1; // Initialize
    while(n > 0){
        if(n % 2 == 1){
            res = res * m;
            n = n-1;
        }
        else {
            m = m * m;
            n = n/2;
        }
    }
    return res;
}
```

n	$M(n)$
1	1
2	2
3	3
4	3
5	4
6	4
7	5
8	4
9	5
10	5

n	$M(n)$
11	6
12	5
13	6
14	6
15	7
16	5
17	6
18	6
19	7
20	6

Aufwandsfunktionen

- Die Laufzeit ergibt sich als Funktion der Eingabegröße
 - ◆ Wenn ein Eingabeparameter als ganze Zahl vorliegt, kann der Wert des Eingabeparameters direkt als Eingabegröße genommen werden
 - ◊ Manchmal wird aber die Stellenzahl, d.h. der 2er-Logarithmus, als Eingabegröße genommen
 - ◆ Bei mehreren Eingabeparametern (und nicht-numerischen Werten) muss erst eine Abbildung auf geeignete numerische Funktionen zur Messung der Eingabegröße durchgeführt werden

Aufwandsfunktionen

- Exakte Bestimmung dieser Funktion des zeitlichen Aufwands eines Algorithmus als Funktion der Eingabegröße im Allgemeinen zu komplex
 - ◆ Ergebnis auch nicht mehr sinnvoll interpretierbar
- Beschränkung auf Wachstumsraten häufig eingesetztes Hilfsmittel („groß“-O-Notation)
 - ◆ Siehe später

Komplexität im schlechtesten, mittleren (und besten) Fall

- Analyse wird oftmals aufgeteilt in folgende Fälle
 - ◆ Komplexität der Wachstumsfunktion im schlechtesten Fall (worst case complexity)
 - ǒ Wachstumsrate im schlechtesten Fall
 - ◆ Komplexität der Wachstumsfunktion im mittleren Fall (average case complexity)
 - ǒ Wachstumsrate im “durchschnittlichen” Fall
 - ǒ Kann sehr viel besser als der „schlechteste Fall“ sein
 - Erfordert aber Wissen über statistische Verteilung von Eingabegrößen
 - ◆ Bester Fall oftmals weniger relevant
 - ǒ Durch Speicherung des Ergebnisses für bestimmte Fälle kann immer eine sehr geringe Komplexität im besten Fall erreicht werden

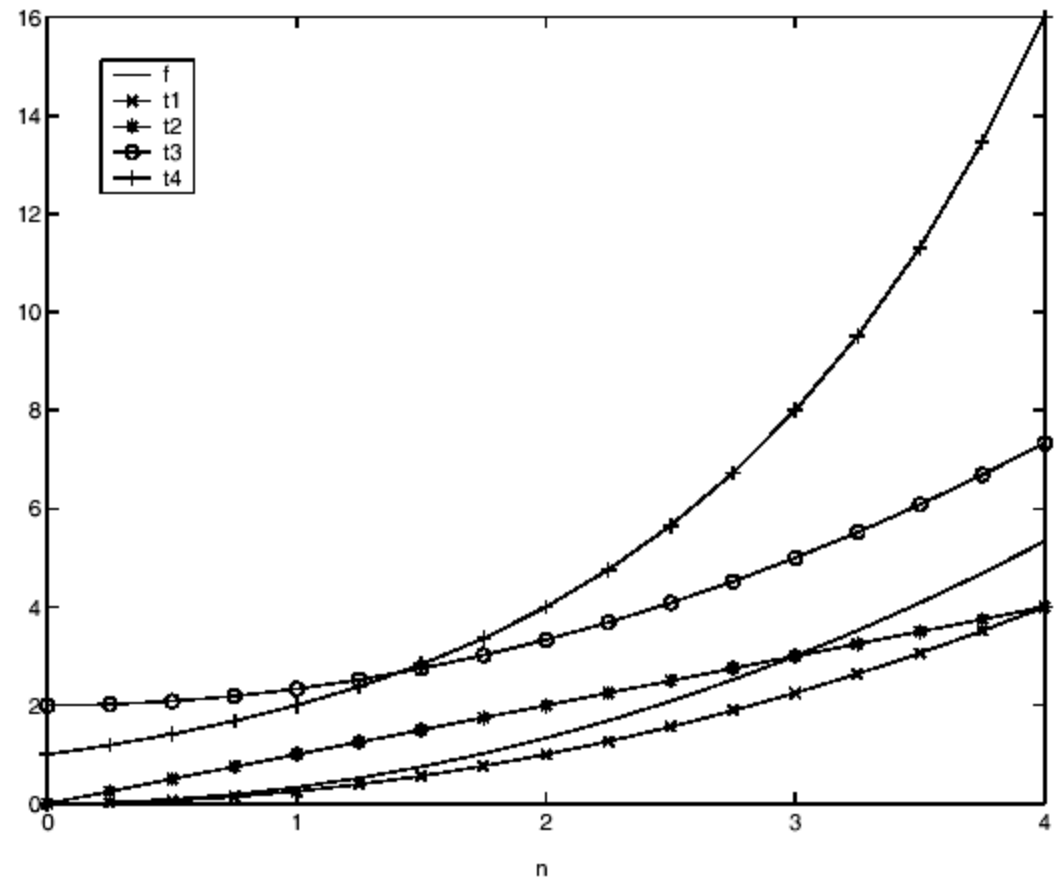
Aufwand und asymptotische Komplexität

Aufwand und asymptotische Komplexität

● Beispiel für Wachstum von Funktionen

Wir wählen $f(n) = \frac{1}{3}n^2$. Es sei

$$\begin{aligned}t_1(n) &= \frac{1}{4}n^2, \\t_2(n) &= n, \\t_3(n) &= \frac{1}{3}n^2 + 2, \\t_4(n) &= 2^n.\end{aligned}$$



Definition 10.4.4. Sei $f : \mathbb{N} \rightarrow \mathbb{R}^*$. Die **Ordnung** von f (the order of f) ist die Menge

$$O(f(n)) = \{t : \mathbb{N} \rightarrow \mathbb{R}^* \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : t(n) \leq c \cdot f(n)\}$$

Definition 10.4.11 („Omega“). Für eine Funktion $f : \mathbb{N} \rightarrow \mathbb{R}^*$ ist die Menge Ω wie folgt definiert:

$$\Omega(f(n)) = \{t : \mathbb{N} \rightarrow \mathbb{R}^* \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : t(n) \geq c \cdot f(n)\}$$

Definition 10.4.12 („Theta“). Die exakte Ordnung Θ von $f(n)$ ist definiert als:

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

O-Notation: Effekt von Konstanten

● Beispiele:

◆ $T_1(n) = 2^n + n^2 + n + 1 + \log(n)$

∅ $T_1(n) \hat{=} O(2^n)$

◆ $T_2(n) = n^{20000} + 2^n$

∅ $T_2(n) \hat{=} O(2^n)$

◆ $T_3(n) = n + \log_2(n)$

∅ $T_3(n) \hat{=} O(n)$

◆ $T_4(n) = 10$

∅ $T_4(n) \hat{=} O(1)$

O-Notation: Effekt von Konstanten

● Beispiele:

◆ $T_6(n) = 1000000000 + \log_2(n) + \log_{20}(n)$

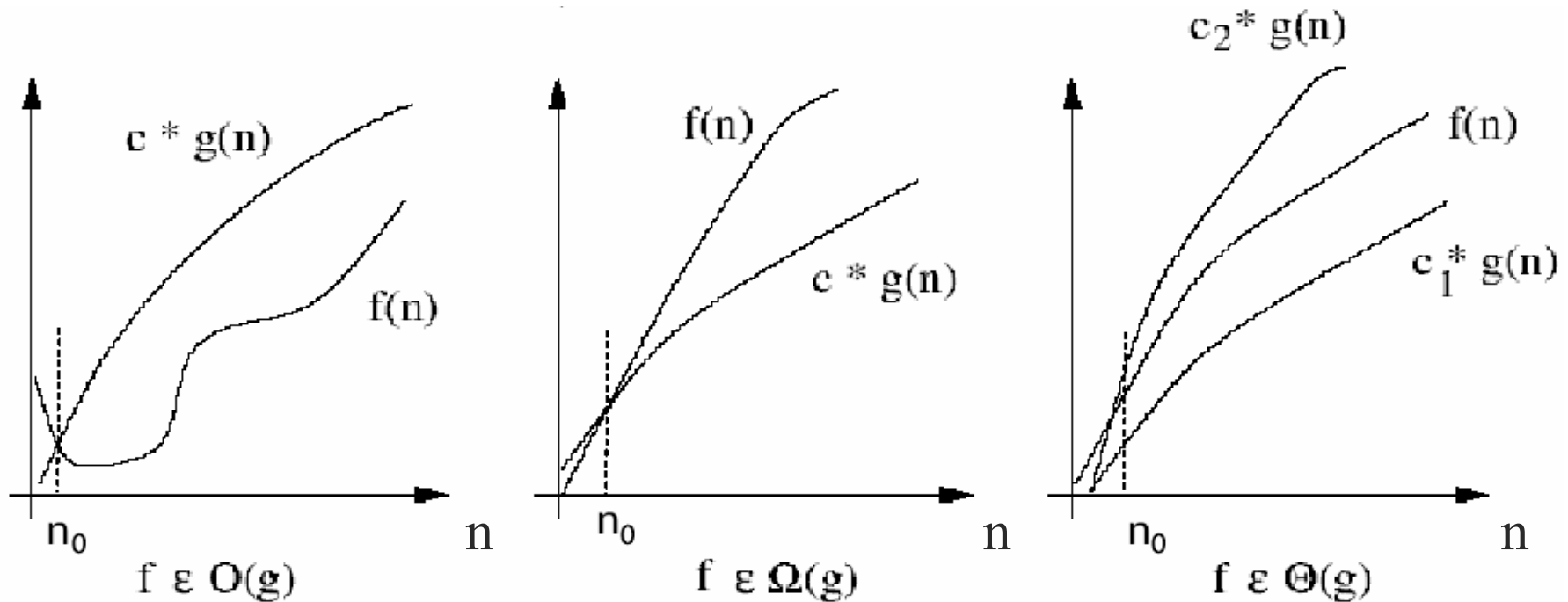
⚡ Bemerkung:

- $\log_2(n) = \log_2(e) \log(n)$
- $\log_{20}(n) = \log_{20}(e) \log(n)$

→ $T_6(n) \hat{=} O(\log(n))$

O-Notation

Beispiele



O-Notation

● Beispiele:

◆ Sei:

$$g(n) = n^2 + 4$$

◆ Dann gilt:

$$\{f(n) = 2n^2 + \log_2(n+1)\} \in O(g(n))$$

◆ Beweis:

$c=4$

$$4g(n) > f(n) \quad \text{if} \quad n \geq 1$$

$n_0=1$

O-Notation

● Beispiele:

◆ Sei:

$$g(n) = n^2 + 4$$

◆ Dann gilt:

$$\{f(n) = 2n^2 + \log_2(n+1)\} \in \Omega(g(n))$$


◆ Beweis:

$$f(n) > \frac{1}{4} g(n) \quad \text{if} \quad n \geq 1$$

O-Notation

● Beispiele:

$$\left\{ f(n) = 2n^2 + \log_2(n+1) \right\} \in \Omega(g(n)) \quad \left\{ f(n) = 2n^2 + \log_2(n+1) \right\} \in O(g(n))$$


$$\left\{ f(n) = 2n^2 + \log_2(n+1) \right\} \in \Theta(g(n))$$

Einige nützliche mathematische Lemmata

- Im folgenden sollen einige nützliche mathematische Lemmata zusammengestellt werden
 - ◆ Diese implizieren, dass die Verwendung der Mengen von Funktionen, die durch die O-Notation beschrieben werden, sehr viel einfacher ist, als Wachstumsfunktion direkt zu beschreiben

Bemerkung:

- ◆ Wir verwenden O-Notation hauptsächlich im Zusammenhang mit Zeitkomplexitäten
- ◆ Kann auch für Platz-Komplexitäten verwendet werden
- ◆ Oder auch in ganz anderen Zusammenhängen

Einige nützliche mathematische Lemmata

● Lemma 1:

$$O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

◆ Beispiel:

$$O(n^2 + 2^n) = O(2^n)$$

◆ Beweis:

$$\begin{aligned} f(n) + g(n) &\leq 2 \cdot \max(f(n), g(n)) \\ \Rightarrow O(f(n) + g(n)) &= O(\max\{f(n), g(n)\}) \end{aligned}$$

Einige nützliche mathematische Lemmata

● Lemma 2:

$$a.) \quad O(f(n)) \subseteq O(g(n)) \Leftrightarrow f(n) \in O(g(n))$$

$$b.) \quad O(f(n)) = O(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge g(n) \in O(f(n))$$

$$c.) \quad O(f(n)) \subset O(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge g(n) \notin O(f(n))$$

◆ Beispiel:

$$O(n^2) \subset O(2^n)$$

Einige nützliche mathematische Lemmata

- Lemma 3:

$$O(n^m) \subset O(n^{m+1})$$

- ◆ Beweis: Durch vollständige Induktion über m

Einige nützliche mathematische Lemmata

● Lemma 4:

◆ Sei

$$A(n) := a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0 \quad a_i \in \mathbb{R}$$

◆ Dann gilt

$$A(n) \in O(n^m)$$

◆ Beweis:

$$A(n) \leq \max(a_0, \dots, a_m) \cdot m \cdot n^m \in O(n^m)$$

Greedy- und Divide-and-Conquer-Verfahren

Einige erste Analyse-Beispiele von „Greedy“- und „Divde-and-Conquer“-Verfahren

Design Paradigmem

- Es gibt einige wichtige **allgemeine Design Paradigmen** für Algorithmen
 - ◆ Greedy Method
 - “gierige” Methoden
 - ◆ Divide and conquer
 - ◊ Teile-und-Herrsche-Methoden
 - *Divide et impera*
 - ◆ Dynamic programming
 - ◆ Search and enumeration
- An dieser Stelle sollen nur erste Beispiele gegeben werden von
 - ◆ Greedy-Verfahren
 - ◆ Divide-and-conquer-Verfahren

Greedy- und Divide-and-Conquer-Verfahren

- Schritte in einem greedy-Verfahren
 - ◆ **Behandle** einfache und triviale Fälle
 - ◆ **Reduziere** Problem (in eine Richtung)
 - ◆ **Löse** durch Iteration oder Rekursion
- Schritte in Divide-and-Conquer-Verfahren
 - ◆ **Behandle** einfache und triviale Fälle
 - ◆ **Divide**: Reduziere das problem in zwei (oder mehr) Unterprobleme
 - Die in etwa “gleich groß” sind
 - ◆ **Conquer**: Löse die Subprobleme
 - Im allgemeinen durch Rekursion
 - ◆ **Kombiniere**: Erhalte Lösung des Ausgangsproblems durch Kombination der zuvor erhaltenen Teillösungen

Greedy- und Divide-and-Conquer-Verfahren

● Beispiel: Exponentiation x^y

ǒ Greedy

$$x^y = x \cdot x^{y-1}$$

```
int power(int x, int y)
{
    if (y == 0) then return 1;
    else return x*power(x,y-1);
}
```

ǒ Divide –and-Conquer

$$x^y = x^{\frac{y}{2}} \cdot x^{\frac{y}{2}}$$

```
int power(int x, int y)
{
    if (y == 0) then return 1;
    else
    {
        if (y % 2 == 0) {int c=power(x,y/2); return c*c}
        else return x*power(x,y-1);
    }
}
```

Check if y is even

Bestimmung des Aufwandes

- Beispiel:

- Exponentiation x^y

- ◆ Greedy:

- ⌘ y Multiplikationen sind notwendig

```
int power(int x, int y)
{
    if (y == 0) then return 1;
    else return x*power(x,y-1);
}
```

- ◆ Divide-Conquer-Fall:

- ⌘ Bester Fall:

- “In etwa” $\log_2(y)$ Multiplikationen

- ⌘ Schlechter Fall

- “In etwa” $2*\log_2(y)$ Multiplikationen

```
int power(int x, int y)
{
    if (y == 0) then return 1;
    else
    {
        if (y % 2 == 0) {int c=power(x,y/2); return c*c}
        else return x*power(x,y-1);
    }
}
```

- ◆ Siehe auch folgende Folien

Bestimmung des Aufwandes

- Analyse des “divide-and-conquer”-Exponentiations-Verfahrens
 - ◆ Anzahl der benötigten Multiplikationen M ist $(S-1)+E$
 - ⌘ In binärer Repräsentation der Eingabegröße n
 - S Stellenanzahl
 - E Anzahl der Stellen mit 1
 - ⌘ Beispiel: $n = 9 = 1 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 1001_2$
 - $E = 2$
 - $S = 4$
 - $M = (4-1)+2 = 5$

Bestimmung des Aufwandes

● Beste Fälle

- ◆ $n=2^k$

- ✗ Binärdarstellung hat eine 1 und k Nullen

- ◆ $M_{best}(n) = M(2^k) = k + 1 = \log_2(n) + 1$

● Schlechterster Fall

- ◆ $n=2^k-1$

- ✗ Binärdarstellung von n enthält keine Ziffern 0

- ◆ $M_{worst}(n) = M(2^k-1) = 1 + M(2^k-2) = 2 + M(2^{k-1}-1) = 3 + M(2^{k-1}-2) = 4 + M(2^{k-2}-1)$
 $= \dots$

- $\dots = 2k + M(1) = 2k + 1 = 2 \lfloor \log_2(n) \rfloor + 1$

Sortieralgorithmen

Weitere Beispiele von „Greedy“- und „Divide-and-Conquer“-Verfahren

Einleitung und Problemstellung

- Aufgabe: Sortiere Folge F aufsteigend (bzw. absteigend)
- Klassifikation elementarer Sortierverfahren
 - ◆ greedy
 - ◆ divide-and-conquer
- Implementierungen basieren auf Folgen als
 - ◆ Reihungen (arrays)
 - ◆ Listen (linked lists)
 - ◊ Siehe Vorlesung im Sommersemester
- Wichtige Gesichtspunkte:
 - ◆ Laufzeitkomplexität (asymptotisch)
 - ◆ Speicherplatz:
 - ◊ Bei Reihungen Operationen „in place“ bevorzugt
 - ◊ Bei jeder Rekursion Platz für Parameter und Variablen

Grundprinzipien elementarer Sortiervverfahren: Greedy

- Sortieren durch Auswahl (selection sort)
 - ◆ Finde in F kleinstes Element und füge es an das Ende von S an.
- Sortieren durch Einfügen (insertion sort)
 - ◆ Nehme erstes Element aus F und füge es an richtiger Stelle in S ein.
- Sortieren durch Austauschen (bubble sort)
 - ◆ Gehe von links nach rechts durch F und vertausche benachbarte Elemente, falls falsch geordnet. Wiederhole den Schritt, bis Folge fertig sortiert.

Grundprinzipien elementarer Sortiervverfahren: divide-and-conquer

● Quicksort

- ◆ Teile F in Teilfolgen F_1, F_2 , wobei Elemente in F_1 kleiner als Elemente in F_2 sind.
- ◆ Sortiere F_1, F_2 rekursiv
- ◆ Füge sortierte Teilfolgen zusammen.

● Sortieren durch Mischen (merge sort)

- ◆ Teile F in Teilfolgen F_1, F_2
- ◆ Sortiere F_1, F_2 rekursiv
- ◆ Bilde S durch iteratives Anhängen des kleineren der jeweils ersten Elemente aus F_1, F_2

Selection sort: Sortieren durch Auswahl

- Prinzip: Solange es Elemente in der Folge F gibt, nehme das kleinste Element und hänge es an die Resultatfolge S an.

SelectSort (F)

// Die unsortierte Folge F wird in die sortierte Folge S überführt.

1. **Initialisiere:** $S = [\text{ }]$;
2. **Trivialfall:** if ($F == [\text{ }]$) return (S);
3. **Reduktion:** Minimum aus S nach F überführen.
 $a = \text{select}(F)$;
 $S = \text{appendElem}(S, a)$;
 $F = \text{deleteElem}(F, a)$;
4. **Iteriere:** Weiter bei 2.

Selection sort: Sortieren durch Auswahl

Anfang

Position	0	1	2	3	4	5
F	5	2	4	6	1	3
S	-	-	-	-	-	-

Schritt 1

Position	0	1	2	3	4	5
F	5	2	4	6	3	-
S	1	-	-	-	-	-

Schritt 2

Position	0	1	2	3	4	5
F	5	4	6	3	-	-
S	1	2	-	-	-	-

Selection sort: Sortieren durch Auswahl

Schritt 3

Position	0	1	2	3	4	5
F	5	4	6	-	-	-
S	1	2	3	-	-	-

Schritt 4

Position	0	1	2	3	4	5
F	5	6	-	-	-	-
S	1	2	3	4	-	-

Schritt 5

Position	0	1	2	3	4	5
F	6	-	-	-	-	-
S	1	2	3	4	5	-

Schritt 6

Position	0	1	2	3	4	5
F	-	-	-	-	-	-
S	1	2	3	4	5	6

Selection sort: Array Implementierung I

- Reihung $\text{data}[0], \dots, \text{data}[n-1]$ partitioniert in
 - ◆sortierter Anfang S : $\text{data}[0], \dots, \text{data}[\text{next}-1]$
 - ◆unsortierter Rest F : $\text{data}[\text{next}], \dots, \text{data}[n-1]$
- Finde Index min des minimalen Elementes in F
- Vertausche $\text{data}[\text{next}]$ mit $\text{data}[\text{min}]$
- Inkrementiere next und iteriere bis $\text{next}=n$

Selection sort: Array Implementierung II

```
void swap(int* data, int a, int b) {
    int temp;
    temp = data[a];
    data[a] = data[b];
    data[b] = temp;
}

int findMin(int* data, int size, int pos) {
    int min = pos;
    int i;
    for(i = pos; i < size; i++) {
        if(data[i] < data[min])
            min = i;
    }
    return min;
}

void selectSort(int* data, int size) {
    int min, next;
    for(next = 0; next < size - 1; next++) {
        min = findMin(data, size, next);
        swap(data, next, min);
    }
}
```


Insertion sort: Sortieren durch Einfügen

- Prinzip: Nehme jeweils das erste Element aus F und füge es in S an der richtigen Stelle ein.

InsertSort(F)

// Die unsortierte Folge F wird in die sortierte Folge S überführt.

1. **Initialisiere:** $S = \boxed{\mathbb{W}}$;
2. **Trivialfall:** if ($F == \boxed{\mathbb{W}}$) return(S);
3. **Reduziere** F : ein Element aus F sortiert nach S überführen.
 $a = \text{takeFirst}(F)$;
 $S = \text{insertSorted}(S, a)$;
4. **Iteriere:** Weiter bei 2.

Insertion sort: Sortieren durch Einfügen

Anfang

Position	0	1	2	3	4	5
F	5	2	4	6	1	3
S	-	-	-	-	-	-

Schritt 1

Position	0	1	2	3	4	5
F	2	4	6	1	3	-
S	5	-	-	-	-	-

Schritt 2

Position	0	1	2	3	4	5
F	4	6	1	3	-	-
S	2	5	-	-	-	-

Insertion sort: Sortieren durch Einfügen

Schritt 3

Position	0	1	2	3	4	5
F	6	1	3	-	-	-
S	2	4	5	-	-	-

Schritt 4

Position	0	1	2	3	4	5
F	1	3	-	-	-	-
S	2	4	5	6	-	-

Schritt 5

Position	0	1	2	3	4	5
F	3	-	-	-	-	-
S	1	2	4	5	6	-

Schritt 6

Position	0	1	2	3	4	5
F	-	-	-	-	-	-
S	1	2	3	4	5	6

Insertion sort: arraybasiert

- Reihung $\text{data}[0], \dots, \text{data}[n-1]$ partitioniert in
 - ◆sortierter Anfang S : $\text{data}[0], \dots, \text{data}[\text{next}-1]$
 - ◆unsortierter Rest F : $\text{data}[\text{next}], \dots, \text{data}[n-1]$
- Wähle Element $\text{data}[\text{next}]$ zum Einfügen
- Suche Einfügestelle absteigend von $\text{next}-1$ bis 0
- Schiebe dabei jedes zu große Element eine Position nach hinten bis Einfügestelle gefunden

Bubble sort: Sortieren durch Austauschen

- Gehe von links nach rechts durch Folge F , vertausche falsch geordnete Nachbarn. So wird größtes Element an das Ende bewegt.
- Wiederholen bis die ganze Folge geordnet ist. (So oft wie die Folge Elemente hat.)
- S bildet sich also rechts von F und dehnt sich nach links aus.
- Optimierung: Größte Elemente nach Sortierung nicht mehr beachten.
- Zeiteffizient für vorsortierte Folgen.
- Platzeffizient, da nur eine Folge benutzt wird.

Bubble sort: Sortieren durch Austauschen

Anfang

Position	0	1	2	3	4	5
F	5	2	4	6	1	3

Schritt 1

Position	0	1	2	3	4	5
F	2	5	4	6	1	3

Schritt 2

Position	0	1	2	3	4	5
F	2	4	5	6	1	3

Schritt 3

Position	0	1	2	3	4	5
F	2	4	5	6	1	3

Schritt 4

Position	0	1	2	3	4	5
F	2	4	5	1	6	3

Bubble sort: Sortieren durch Austauschen

Schritt 5

Position	0	1	2	3	4	5
F	2	4	5	1	3	6

Schritt 6

Position	0	1	2	3	4	5
F	2	4	5	1	3	6

Schritt 7

Position	0	1	2	3	4	5
F	2	4	5	1	3	6

Schritt 8

Position	0	1	2	3	4	5
F	2	4	1	5	3	6

Schritt 9

Position	0	1	2	3	4	5
F	2	4	1	3	5	6

Bubble sort: Sortieren durch Austauschen

Schritt 10

Position	0	1	2	3	4	5
F	2	4	1	3	5	6

Schritt 11

Position	0	1	2	3	4	5
F	2	4	1	3	5	6

Schritt 12

Position	0	1	2	3	4	5
F	2	1	4	3	5	6

Schritt 13

Position	0	1	2	3	4	5
F	2	1	3	4	5	6

Schritt 14

Position	0	1	2	3	4	5
F	2	1	3	4	5	6

Bubble sort: Sortieren durch Austauschen

Schritt 15

Position	0	1	2	3	4	5
F	2	1	3	4	5	6

Schritt 16

Position	0	1	2	3	4	5
F	1	2	3	4	5	6

...

Schritt 20

Position	0	1	2	3	4	5
F	1	2	3	4	5	6

Tausch am Anfang eines Durchlaufs, danach nicht mehr: Fertig!

Bubble sort: Arrayimplementierung

```
void bubbleSort(int* data, int size) {  
    int pos;  
    int hasSwapped;  
    do {  
        hasSwapped = 0;  
        for(pos = 0; pos < size - 1; pos++) {  
            if(data[pos] > data[pos + 1]) {  
                swap(data, pos, pos + 1);  
                hasSwapped = 1;  
            }  
        }  
    } while(hasSwapped == 1);  
}
```

Anmerkung: Keine Suche mehr nach dem Minimum!
=> bei langen Folgen schneller

divide-and-conquer: Prinzip

Beim Sortieren gibt es zwei Ausprägungen:

Hard split / easy join: Dabei wird die gesamte Arbeit beim Teilen des Problems verrichtet und die Kombination ist trivial, das heißt, F wird so in F_1 und F_2 partitioniert, daß $S = S_1 \ S_2$. Dieses Prinzip führt zum *Quicksort*-Algorithmus.

Easy split / hard join: Dabei ist die Aufteilung $F = F_1 \ F_2$ trivial und die ganze Arbeit liegt beim Zusammensetzen von S_1 und S_2 zu S . Dieses Prinzip führt zum *Mergesort*-Algorithmus.

quicksort: Allgemeines

- Einer der besten und meist genutzten Sortieralgorithmen
- Hauptvorteile:
 - ◆ sortiert Array in place
 - ◆ $O(n \log n)$ Komplexität im Mittel
 - ◆ $O(\log n)$ zusätzlicher Speicherplatz auf Stack
 - ◆ in Praxis schneller als andere Verfahren mit derselben asymptotischen Komplexität $O(n \log n)$

quick sort: Prinzip

- Wähle und entferne Pivotelement p aus F
- Zerlege F in Teilfolgen F_1 , F_2 so daß
 - ◆ $e_1 < p \leq e_2$ für alle e_1 in F_1 , e_2 in F_2
 - ◆ Sortiere (rekursiv) F_1 zu S_1 und F_2 zu S_2
- Gesamtlösung ist $S_1 p S_2$

quicksort: Zerlegung in Teilfolgen

1. Für Zerlegung in F_1, F_2 ist p mit jedem Element von $F \setminus \{p\}$ zu vergleichen
2. Zerlegung soll durch Vertauschen von Elementen geschehen (kein Zusatzspeicher, in place!)
3. Wähle p als rechtes Randelement von F
4. Suche $i = \arg \min_i F[i] \geq p$,
 $j = \arg \max_j F[j] < p$
5. Paar $F[i], F[j]$ „steht falsch“, also vertausche $F[i]$ mit $F[j]$
6. Terminierung (ohne Vertauschen!), wenn erstmals $i > j$, sonst gehe zum Suchschritt 4
7. Vertausche p mit linkem Rand von F_2 (also mit $F[i]$)

quicksort: Wahl des Pivotelements

- Algorithmus ist schnell wenn die Teilfolgen F_1 , F_2 etwa gleich groß
- Im schlimmsten Fall ist eine Teilfolge immer leer (kann vorkommen, wenn Folge vorsortiert ist)
- Daher wählt man häufig p als Median von drei Stichproben, z. B. erstes, mittleres und letztes Element von F
- (b ist Median von a, b, c , wenn $a \leq b \leq c$)

quicksort: Beispiel

(9, 7, 1, 6, 2, 3, 8, 4)
(9, 7, 1, 6, 2, 3, 8), (4)
(3, 7, 1, 6, 2, 9, 8), (4)
(3, 2, 1, 6, 7, 9, 8), (4)
(3, 2, 1, 4, 7, 9, 8, 6)

Nun ist die 4 am richtigen Platz angelangt. Danach geht es weiter mit dem rekursiven Aufruf auf den beiden Teilfolgen:

$$F_1 = (3, 2, 1) \text{ und } F_2 = (7, 9, 8, 6)$$

quick sort: Implementierung I/II

/**

* Sorts data[l], data[l+1], ... , data[r]

* in ascending order using the quicksort algorithm

* Precondition: $0 \leq l \leq r < \text{size}$.

*/

```
public void quickSort(int * data, int size, int l, int r)
{
```

```
    // 0. Initialisiere
```

```
    int i = l, j = r-1;
```

```
    // 1. Check auf Trivialfall
```

```
    if (l >= r) return;
```

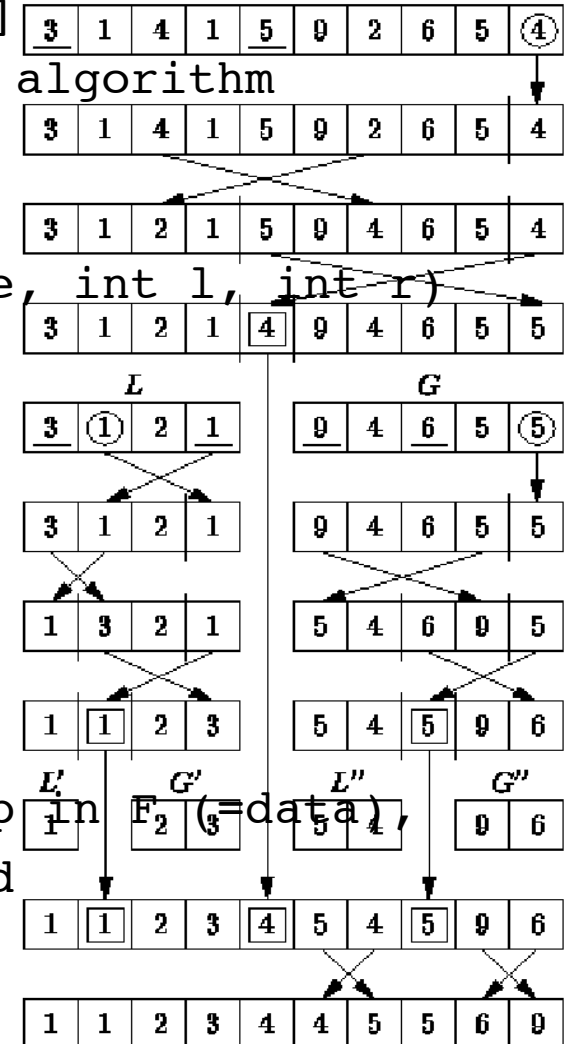
```
    // Initialisiere pivot
```

```
    int pivot = data[r];
```

```
    // 2. Teile: Füge pivot an einen Platz p in F2 (=data),
```

```
    // ein, so dass F[i] < F[p] für l ≤ i < p und
```

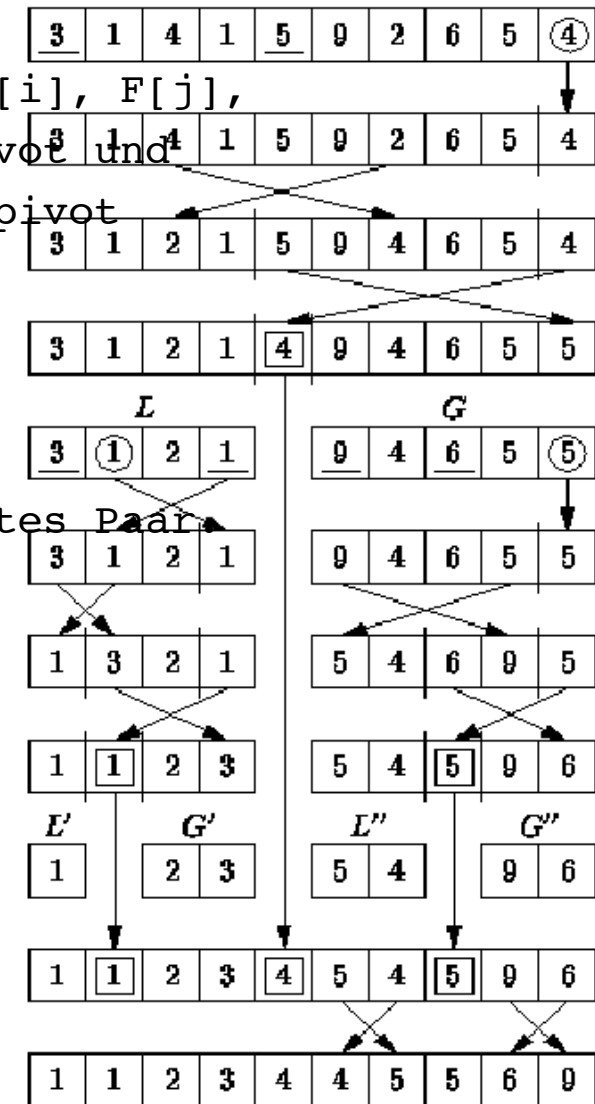
```
    // F[j] ≥ F[p] für p ≤ j ≤ r.
```



quicksort: Implementierung II

```
// 2.1 Finde äusserstes ungeordnetes Paar F[i], F[j],
//      i<j, mit F[i] >= pivot und F[j] < pivot und
//      F[s] < pivot für l<=s<i und F[s] >= pivot
//      für j<s<=r.
while( data[i]< pivot)  ) i++;
while( j>=l && data[j] >= pivot) j--;

// 2.2 Ordne Paar; finde nächstes ungeordnetes Paar
while( i < j ) {
    // i<j impliziert j>=l,
    // daher F[j]<pivot und F[i]>=pivot.
    swap(data, i, j);
    while( data[i] < pivot) i++;
    while( data[j]>= pivot) j--;
}
```



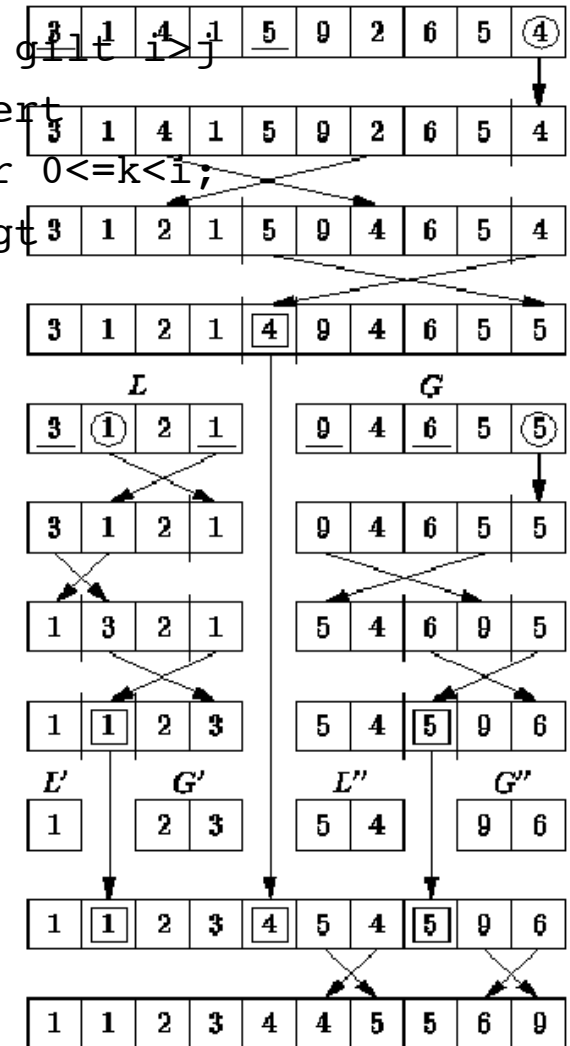
quicksort: Implementierung III

```
// 2.3 Endgültiger Platz für pivot ist i: es gilt  $i > j$ 
//      (und nicht nur  $i \geq j$ , denn  $i = j$  impliziert
//       $\text{pivot} \leq F[i] < \text{pivot}$ ) und  $F[k] < \text{pivot}$  für  $0 \leq k < i$ ;
//       $F[k] \geq \text{pivot}$  für  $j < k \leq r$ ; wegen  $i > j$  folgt
//       $F[k] \geq \text{pivot}$  für  $i \leq k \leq r$ .
swap(data, i, r);
```

```
// 3. Herrsche: Sortiere links und rechts
//      vom Ausgangspunkt.
quickSort(data, size, l, i-1);
quickSort(data, size, i+1, r);
// Das Combine war trivial!
```

```
}
```

```
}
```



quicksort: Komplexität I

- Anzahl der Vergleiche $T(n)$, n = Länge der Eingabe
- Auf jeder Rekursionsebene sind $O(n)$ Vergleiche von Datenelementen zu machen (genaue Zahl hängt ab von der Anzahl der swap-Operationen)
- Maximale Anzahl von Rekursionsebenen ist n (alle Folgelemente sind kleiner als Pivotelement)
- Im schlimmsten Fall also $O(n^2)$ Vergleiche

quicksort: Komplexität II

- Im besten Fall sind Teilfolgen gleich groß und ungefähre Rechnung für $n = 2^k$ ergibt

$$\begin{aligned}T_{\text{best}}(n) &= T_{\text{best}}(2^k) = n + 2 \cdot T_{\text{best}}(2^{k-1}) \\&= n + 2 \cdot n/2 + 4 \cdot T_{\text{best}}(2^{k-2}) = n + n + 4 \cdot T_{\text{best}}(2^{k-2}) \\&= \dots = n + n + \dots + n + 2^k \cdot T_{\text{best}}(1) \\&= k \cdot n + 2^k \cdot T_{\text{best}}(1) = k \cdot n + n \cdot c \\&= n \log_2 n + cn \quad \boxed{\approx} \quad O(n \log n)\end{aligned}$$

- Komplexität im Mittel ebenfalls $O(n \log n)$.
(Beweis siehe z.B. in Ottmann/Widmayer)

Mergesort: Allgemeines

- Prinzip: Teile Folge rekursiv in immer kleinere Teilfolgen und sortiere diese. Füge Teilfolgen dann sortiert zusammen.

Mergesort (F)

1. **Trivialfall:** $\text{if}(|F| == 1) \text{ return } (F) ;$
2. **Divideschritt:** *teile F in zwei Hälften A und B*
 $X = \text{Mergesort}(A),$
 $Y = \text{Mergesort}(B)$
3. **Mergeschritt:** $\text{return } (\text{merge}(X, Y));$

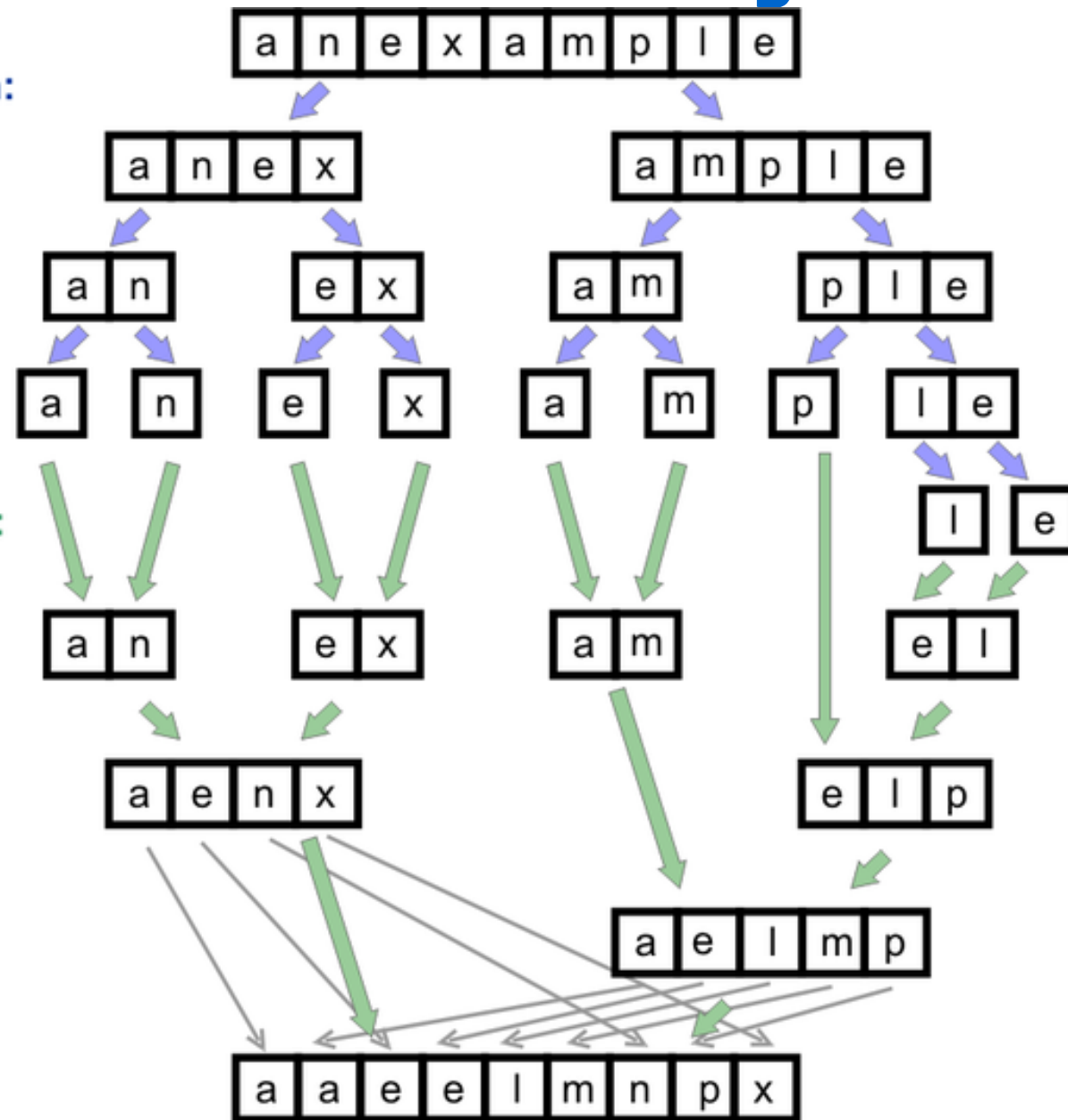
$\text{merge}(A, B)$

1. **Initialisiere:** $S = \boxed{\mathbb{W}};$
2. **Trivialfall:** $\text{if } (A == \boxed{\mathbb{W}} \text{ und } B = \boxed{\mathbb{W}}) \text{ return}(S);$
3. **Verknüpfe:** *Durchlaufe A und B parallel,*
 $a = \min(A, B);$ // Ist immer das erste Element von A oder B
 $S = \text{appendElem}(S, a);$
 $F = \text{deleteElem}(A \text{ oder } B, a);$
4. **Iteriere:** *Weiter bei 2.*

Mergesort: Beispiel

Aufteilen:

Mischen:



Mergesort: Arrayimplementierung

```
void merge(int* data, int start, int haelfte, int ende) {
    int pos1 = start, pos2 = haelfte + 1, i = 0;
    int temp[ende - start + 1];
    while(pos1 <= haelfte || pos2 <= ende) {
        if(pos2 > ende) {
            temp[i++] = data[pos1++];
        } else if (pos1 > haelfte) {
            temp[i++] = data[pos2++];
        } else if (data[pos1] < data[pos2]) {
            temp[i++] = data[pos1++];
        } else {
            temp[i++] = data[pos2++];
        }
    }
    for(i=0; i< ende - start + 1; i++) {
        data[start + i] = temp[i];
    }
}

void mergeSort(int* data, int start, int ende) {
    int haelfte;
    if(start != ende) {
        haelfte = (start + ende) / 2;
        mergeSort(data, start, haelfte);
        mergeSort(data, haelfte + 1, ende);
        merge(data, start, haelfte, ende);
    }
}
```