Kapitel ADS:III

III. Sortieren

- Sortieralgorithmen
- Insertion Sort
- Heapsort
- Merge Sort
- Quicksort
- Counting Sort
- Radix Sort
- Bucket Sort
- Minimales vergleichsbasiertes Sortieren

Counting Sort

Algorithmus

Problem: Sortieren

Instanz: A. Folge von n Zahlen $A = (a_1, a_2, \ldots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \ldots, a'_n)$ von A, so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz A löst.

Idee: Sortieren durch Abzählen von Elementen.

Counting Sort

Algorithmus

Problem: Sortieren

Instanz: A. Folge von n Zahlen $A = (a_1, a_2, \ldots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \ldots, a'_n)$ von A, so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz A löst.

Idee: Sortieren durch Abzählen von Elementen.

Voraussetzung:

- Alle zu sortierenden Elemente sind im Intervall $[0, k]$.

➜ Die Position des i-ten Elements von A ist damit durch Abzählen bestimmbar.

Counting Sort

Algorithmus

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

```
C = array(k)
FOR i = 1 TO n DO
   C[A[i]] = C[A[i]] + 1
ENDDO
FOR i = 1 TO k DO
   C[i] = C[i] + C[i - 1]
ENDDO
FOR i = n DOWNTO 1 DO
   B[C[A[i]]] = A[i]
   C[A[i]] = C[A[i]] - 1
ENDDO
```

Counting Sort

Algorithmus

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

```
C = array(k)
FOR i = 1 TO n DO
    C[A[i]] = C[A[i]] + 1
ENDDO
FOR i = 1 TO k DO
    C[i] = C[i] + C[i - 1]
ENDDO
FOR i = n DOWNTO 1 DO
    B[C[A[i]]] = A[i]
    C[A[i]] = C[A[i]] - 1
ENDDO
```

Beispiel:

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

	1	2	3	4	5	6	7	8
B								

	0	1	2	3	4	5
C						

Counting Sort

Algorithmus

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

```
C = array(k)
FOR i = 1 TO n DO
    C[A[i]] = C[A[i]] + 1
ENDDO
FOR i = 1 TO k DO
    C[i] = C[i] + C[i-1]
ENDDO
FOR i = n DOWNTO 1 DO
    B[C[A[i]]] = A[i]
    C[A[i]] = C[A[i]] - 1
ENDDO
```

Beispiel:

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

	1	2	3	4	5	6	7	8
B								

	0	1	2	3	4	5
C	2	0	2	3	0	1

Counting Sort

Algorithmus

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

```
C = array(k)
FOR i = 1 TO n DO
   C[A[i]] = C[A[i]] + 1
ENDDO
FOR i = 1 TO k DO
   C[i] = C[i] + C[i − 1]
ENDDO
FOR i = n DOWNTO 1 DO
   B[C[A[i]]] = A[i]
   C[A[i]] = C[A[i]] − 1
ENDDO
```

Beispiel:

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

	1	2	3	4	5	6	7	8
B								

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

Counting Sort

Algorithmus

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

```
C = array(k)
FOR i = 1 TO n DO
    C[A[i]] = C[A[i]] + 1
ENDDO
FOR i = 1 TO k DO
    C[i] = C[i] + C[i - 1]
ENDDO
FOR i = n DOWNTO 1 DO
    B[C[A[i]]] = A[i]
    C[A[i]] = C[A[i]] - 1
ENDDO
```

Beispiel:

	1	2	3	4	5	6	7	8 (*i*)
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B								

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

Counting Sort

Algorithmus

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

```
C = array(k)
FOR i = 1 TO n DO
    C[A[i]] = C[A[i]] + 1
ENDDO
FOR i = 1 TO k DO
    C[i] = C[i] + C[i - 1]
ENDDO
FOR i = n DOWNTO 1 DO
    B[C[A[i]]] = A[i]
    C[A[i]] = C[A[i]] - 1
ENDDO
```

Beispiel:

i = 7

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

	1	2	3	4	5	6	7	8
B							3	

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

Counting Sort

Algorithmus

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

$C = \mathit{array}(k)$
FOR $i = 1$ **TO** n **DO**
  $C[A[i]] = C[A[i]] + 1$
ENDDO
FOR $i = 1$ **TO** k **DO**
  $C[i] = C[i] + C[i-1]$
ENDDO
FOR $i = n$ **DOWNTO** 1 **DO**
  $B[C[A[i]]] = A[i]$
  $C[A[i]] = C[A[i]] - 1$
ENDDO

Beispiel:

	1	2	3	4	5	6 (i)	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B		0					3	

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

Counting Sort

Algorithmus

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

```
C = array(k)
FOR i = 1 TO n DO
    C[A[i]] = C[A[i]] + 1
ENDDO
FOR i = 1 TO k DO
    C[i] = C[i] + C[i − 1]
ENDDO
FOR i = n DOWNTO 1 DO
    B[C[A[i]]] = A[i]
    C[A[i]] = C[A[i]] − 1
ENDDO
```

Beispiel:

	1	2	3	4	5 (i)	6	7	8
A	2	5	3	0	2	3	0	3

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

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

Counting Sort
Algorithmus

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

$C = \mathit{array}(k)$
FOR $i = 1$ **TO** n **DO**
  $C[A[i]] = C[A[i]] + 1$
ENDDO
FOR $i = 1$ **TO** k **DO**
  $C[i] = C[i] + C[i-1]$
ENDDO
FOR $i = n$ **DOWNTO** 1 **DO**
  $B[C[A[i]]] = A[i]$
  $C[A[i]] = C[A[i]] - 1$
ENDDO

Beispiel:

	1	2	3	4 (i)	5	6	7	8
A	2	5	3	0	2	3	0	3

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

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

Counting Sort

Algorithmus

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

```
C = array(k)
FOR i = 1 TO n DO
   C[A[i]] = C[A[i]] + 1
ENDDO
FOR i = 1 TO k DO
   C[i] = C[i] + C[i − 1]
ENDDO
FOR i = n DOWNTO 1 DO
   B[C[A[i]]] = A[i]
   C[A[i]] = C[A[i]] − 1
ENDDO
```

Beispiel:

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

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

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

Counting Sort

Algorithmus

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

$C = \mathit{array}(k)$
FOR $i = 1$ **TO** n **DO**
  $C[A[i]] = C[A[i]] + 1$
ENDDO
FOR $i = 1$ **TO** k **DO**
  $C[i] = C[i] + C[i-1]$
ENDDO
FOR $i = n$ **DOWNTO** 1 **DO**
  $B[C[A[i]]] = A[i]$
  $C[A[i]] = C[A[i]] - 1$
ENDDO

Beispiel:

i

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

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

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

Counting Sort

Algorithmus

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

```
C = array(k)
FOR i = 1 TO n DO
    C[A[i]] = C[A[i]] + 1
ENDDO
FOR i = 1 TO k DO
    C[i] = C[i] + C[i − 1]
ENDDO
FOR i = n DOWNTO 1 DO
    B[C[A[i]]] = A[i]
    C[A[i]] = C[A[i]] − 1
ENDDO
```

Beispiel:

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

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

	0	1	2	3	4	5
C	0	2	3	4	7	7

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

```
C = array(k)
FOR i = 1 TO n DO
    C[A[i]] = C[A[i]] + 1
ENDDO
FOR i = 1 TO k DO
    C[i] = C[i] + C[i - 1]
ENDDO
FOR i = n DOWNTO 1 DO
    B[C[A[i]]] = A[i]
    C[A[i]] = C[A[i]] - 1
ENDDO
```

Beispiel:

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

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

	0	1	2	3	4	5
C	0	2	2	4	7	7

Counting Sort

Algorithmus

Algorithmus: Counting Sort.

Eingabe: A. Array von n ganzen Zahlen.
B. Array der Länge n als Ausgabe.
k. Wert des größten Elements in A.

Ausgabe: Eine aufsteigend sortierte Permutation von A in B.

CountingSort(A, B, k)

```
C = array(k)
FOR i = 1 TO n DO
   C[A[i]] = C[A[i]] + 1
ENDDO
FOR i = 1 TO k DO
   C[i] = C[i] + C[i - 1]
ENDDO
FOR i = n DOWNTO 1 DO
   B[C[A[i]]] = A[i]
   C[A[i]] = C[A[i]] - 1
ENDDO
```

Laufzeit:

- Zwei For-Schleifen in $\Theta(n)$, eine in $\Theta(k)$.

→ $T(n) = \Theta(n + k)$

- Wenn $k = O(n)$, dann $T(n) = \Theta(n)$.

Platz:

- $S(n) = \Theta(n + k)$

Eigenschaften:

- Stabilität: Die Reihenfolge gleicher Elemente in A bleibt erhalten.

Radix Sort

Algorithmus

Problem: Sortieren

Instanz: A. Folge von n Zahlen $A = (a_1, a_2, \ldots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \ldots, a'_n)$ von A, so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz A löst.

Idee: Sortieren durch wiederholtes Abzählen von Elementausschnitten.

Radix Sort
Algorithmus

Problem: Sortieren

Instanz: A. Folge von n Zahlen $A = (a_1, a_2, \ldots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \ldots, a'_n)$ von A, so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz A löst.

Idee: Sortieren durch wiederholtes Abzählen von Elementausschnitten.

Algorithmus: Radix Sort.

Eingabe: A. Array von n ganzen Zahlen.
d. Zahl der Stellen pro Zahl.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

RadixSort(A, d)

1. **FOR** $i = 1$ **TO** d **DO**
2. sort array A on digit i using a **stable sorting algorithm**
3. **ENDDO**

Radix Sort

Beispiel

A

```
329
457
657
839
436
720
355
```

- Sei jedes Element des Arrays A eine Zahl mit $d = 3$ Stellen.

Radix Sort

Beispiel

A		$i=1$
329		720
457		355
657		436
839	→	457
436		657
720		329
355		839

- Sei jedes Element des Arrays A eine Zahl mit $d = 3$ Stellen.
- Sortiere stellenweise, angefangen bei der niedrigstwertigen (rechten) Stelle.
- Voraussetzung für Korrektheit: stabiles Sortieren (z.B. mit Counting Sort).

Radix Sort
Beispiel

A		$i{=}1$		$i{=}2$
329		720		720
457		355		329
657		436		436
839	→	457	→	839
436		657		355
720		329		457
355		839		657

- Sei jedes Element des Arrays A eine Zahl mit $d = 3$ Stellen.
- Sortiere stellenweise, angefangen bei der niedrigstwertigen (rechten) Stelle.
- Voraussetzung für Korrektheit: stabiles Sortieren (z.B. mit Counting Sort).

Radix Sort

Beispiel

A	$i=1$	$i=2$	$i=3$
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

- Sei jedes Element des Arrays A eine Zahl mit $d = 3$ Stellen.
- Sortiere stellenweise, angefangen bei der niedrigstwertigen (rechten) Stelle.
- Voraussetzung für Korrektheit: stabiles Sortieren (z.B. mit Counting Sort).
- Die Stelligkeit d einer Zahl hängt von der Basis der Zahldarstellung ab.

Radix Sort

Stelligkeit

Zahlendarstellung im Dualsystem:

$$2\,931\,468\,631_{10} \;=\; 10101110101110101010100101010111_2$$

Abhängig von b und dem gewählten r ist die Zahl der Stellen $d = \lceil b/r \rceil$

Radix Sort

Stelligkeit

Zahlendarstellung im Dualsystem:

$$2\,931\,468\,631_{10} \;=\; \overbrace{\underbrace{10101110}\underbrace{10111010}\underbrace{10101001}\underbrace{01010111}_{\substack{\text{Stelle mit}\\ r\,=\,8\text{ bit}}}}^{b\,=\,32\text{-bit Wortlänge}}{}_2$$

Abhängig von b und dem gewählten r ist die Zahl der Stellen $d = \lceil b/r \rceil = 4$.

Radix Sort

Stelligkeit

Zahlendarstellung im Dualsystem:

$$2\,931\,468\,631_{10} = \overbrace{10101110\,10111010\,10101001\,\underbrace{01010111}_{\text{Stelle mit } r = 8 \text{ bit}}}^{b = 32\text{-bit Wortlänge}}{}_2$$

Abhängig von b und dem gewählten r ist die Zahl der Stellen $d = \lceil b/r \rceil = 4$.

Laufzeit

- Laufzeit eines stabilen Sortieralgorithmus: $\Theta(n + k)$ für Zahlen in $[0, k]$.
- d Sortierschritte mit stabilem Sortieralgorithmus.

➜ $T(n) = \Theta(d(n + k))$; wenn $k = O(n)$, dann $T(n) = \Theta(dn)$.

Radix Sort

Stelligkeit

Zahlendarstellung im Dualsystem:

$$2\,931\,468\,631_{10} \;=\; \overbrace{\underbrace{10101110}\underbrace{10111010}\underbrace{10101001}\underbrace{01010111}_{\text{Stelle mit } r\,=\,8 \text{ bit}}}^{b\,=\,32\text{-bit Wortlänge}}{}_2$$

Abhängig von b und dem gewählten r ist die Zahl der Stellen $d = \lceil b/r \rceil = 4$.

Laufzeit

- Laufzeit eines stabilen Sortieralgorithmus: $\Theta(n + k)$ für Zahlen in $[0, k]$.
- d Sortierschritte mit stabilem Sortieralgorithmus.

→ $T(n) = \Theta(d(n + k))$; wenn $k = O(n)$, dann $T(n) = \Theta(dn)$.

- Im Dualsystem ist $k = 2^r - 1$, so dass $T(n) = \Theta(\frac{b}{r}(n + 2^r))$.
- Für $b < \lfloor \lg n \rfloor$ führt jedes $r \leq b$ zur Laufzeit $T(n) = \Theta(n)$.
- Für $b \geq \lfloor \lg n \rfloor$ minimiert $r = \lg n$ den Term $\frac{b}{r}(n + 2^r)$, so dass $T(n) = \Theta(bn/\lg n)$.

Bemerkungen:

- „Radix“ ist das englische Wort für die Zahl verschiedener Ziffern, die zur Darstellung von Zahlen verwendet wird. Im Dezimalsystem ist der Radix 10, da zehn verschiedene Ziffern $0-9$ verwendet werden, im Dualsystem ist der Radix 2, 0 und 1.

- Radix Sort ist eines der ältesten Sortierverfahren der Informatik. Seine Umsetzung in Form des von Herman Hollerith erfundenen elektromechanischen Systems zur Verarbeitung von Lochkarten führte zur Gründung des Vorläufers von IBM, die Holleriths Idee in Form der ersten Computer vermarktete.

- Im Beispiel wird die Array-Darstellung als Zeile von Feldern zugunsten einer Spalte von Elementen, bei der gleichwertige Stelle untereinander stehen, aufgegeben.

- Es können auch Zahlen unterschiedlicher Stelligkeit sortiert werden. Dazu werden zunächst alle Zahlen nach der Anzahl ihrer Stellen sortiert, da Zahlen mit mehr Stellen auf jeden Fall größer sind als Zahlen mit weniger Stellen (unter der Annahme, dass negative Zahlen und Zahlen mit führenden Nullen nicht vorkommen). Daraufhin werden Zahlen gleicher Stelligkeit wie oben beschrieben sortiert.

- Es gilt $0 < r \leq b$. Wenn $r > \lg n$, wird 2^r groß, so dass $(n + 2^r)$ nicht mehr in $\Theta(n)$ ist. Beispiel: $r = 2 \lg n$ führt zu $2^{2 \lg n} = (2^{\lg n})^2 = n^2$. Wenn also $b \geq \lg n$, ist $r = \lg n$ optimal. Andernfalls genügt es $r = b$ zu wählen und linear Laufzeit zu erhalten.

Bucket Sort
Algorithmus

Problem: Sortieren

Instanz: A. Folge von n Zahlen $A = (a_1, a_2, \ldots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \ldots, a'_n)$ von A, so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz A löst.

Idee: Sortieren durch Ausnutzen der Werteverteilung.

Bucket Sort

Algorithmus

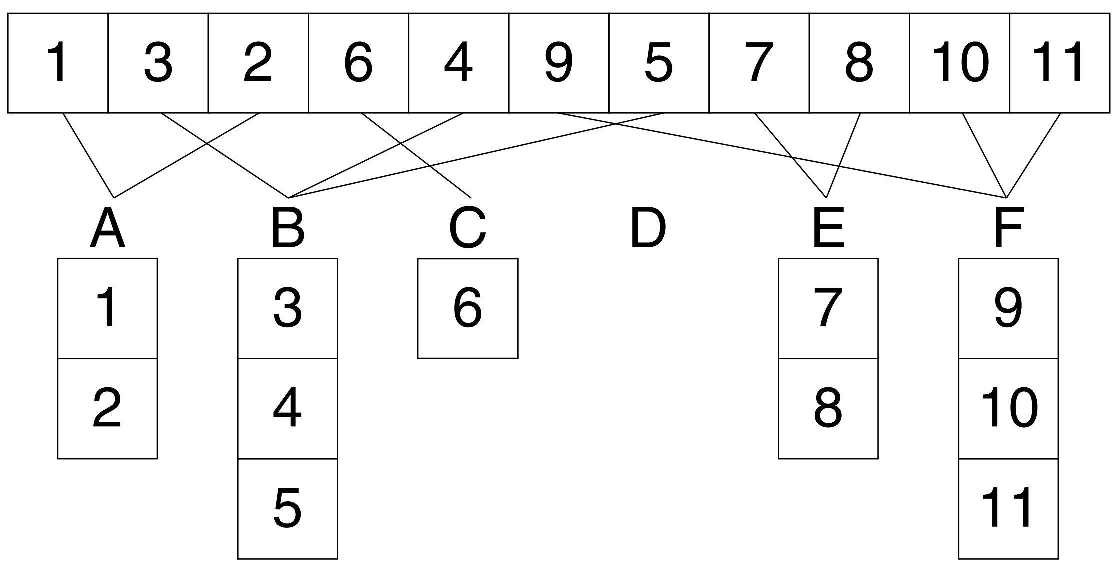
Problem: Sortieren

Instanz: A. Folge von n Zahlen $A = (a_1, a_2, \ldots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \ldots, a'_n)$ von A, so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz A löst.

Idee: Sortieren durch Ausnutzen der Werteverteilung.



Bucket Sort
Algorithmus

Problem: Sortieren

Instanz: A. Folge von n Zahlen $A = (a_1, a_2, \ldots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \ldots, a'_n)$ von A, so dass $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz A löst.

Idee: Sortieren durch Ausnutzen der Werteverteilung.

Voraussetzung:

- Die Werte sind zufällig gleichverteilte reelle Zahlen aus dem Intervall $[0, 1)$.

Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

Eingabe: A. Array von n reellen Zahlen aus $[0, 1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

$B =$ *array*(n)
FOR $i = 0$ **TO** $n - 1$ **DO**
  initialize empty list in $B[i]$
ENDDO
FOR $i = 1$ **TO** n **DO**
  insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$
ENDDO
FOR $i = 0$ **TO** $n - 1$ **DO**
  InsertionSort$(B[i])$
ENDDO
copy $B[0], \ldots, B[n-1]$ in order to A

Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

Eingabe: A. Array von n reellen Zahlen aus $[0, 1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

```
B = array(n)
FOR i = 0 TO n − 1 DO
  initialize empty list in B[i]
ENDDO
FOR i = 1 TO n DO
  insert A[i] into list B[⌊n · A[i]⌋]
ENDDO
FOR i = 0 TO n − 1 DO
  InsertionSort(B[i])
ENDDO
copy B[0], ..., B[n − 1] in order to A
```

Beispiel:

	A
1	.78
2	.17
3	.39
4	.26
5	.72
6	.94
7	.21
8	.12
9	.23
10	.68

Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

Eingabe: A. Array von n reellen Zahlen aus $[0, 1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

```
B = array(n)
FOR i = 0 TO n-1 DO
  initialize empty list in B[i]
ENDDO
FOR i = 1 TO n DO
  insert A[i] into list B[⌊n·A[i]⌋]
ENDDO
FOR i = 0 TO n-1 DO
  InsertionSort(B[i])
ENDDO
copy B[0],...,B[n-1] in order to A
```

Beispiel:

	A		*B*
1	.78	0	/
2	.17	1	/
3	.39	2	/
4	.26	3	/
5	.72	4	/
6	.94	5	/
7	.21	6	/
8	.12	7	/
9	.23	8	/
10	.68	9	/

Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

Eingabe: A. Array von n reellen Zahlen aus $[0, 1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

```
B = array(n)
FOR i = 0 TO n − 1 DO
  initialize empty list in B[i]
ENDDO
FOR i = 1 TO n DO
  insert A[i] into list B[⌊n · A[i]⌋]
ENDDO
FOR i = 0 TO n − 1 DO
  InsertionSort(B[i])
ENDDO
copy B[0], . . . , B[n − 1] in order to A
```

Beispiel:

	A		*B*
1	.78	0	/
2	.17	1	/
3	.39	2	/
4	.26	3	/
5	.72	4	/
6	.94	5	/
7	.21	6	/
8	.12	7	/
9	.23	8	/
10	.68	9	/

Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

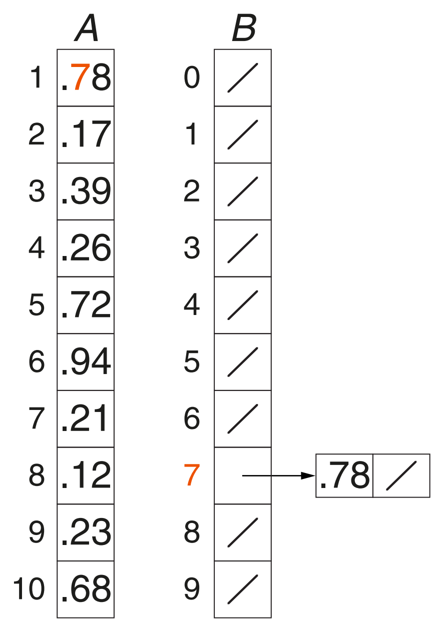
Eingabe: A. Array von n reellen Zahlen aus $[0,1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

```
B = array(n)
FOR i = 0 TO n-1 DO
  initialize empty list in B[i]
ENDDO
FOR i = 1 TO n DO
  insert A[i] into list B[⌊n · A[i]⌋]
ENDDO
FOR i = 0 TO n-1 DO
  InsertionSort(B[i])
ENDDO
copy B[0],...,B[n-1] in order to A
```

Beispiel:



Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

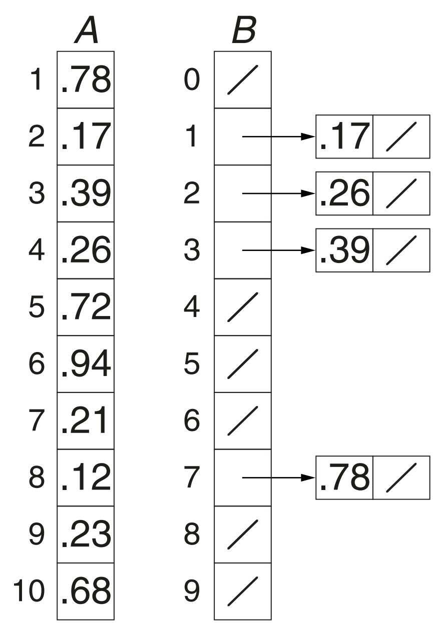
Eingabe: A. Array von n reellen Zahlen aus $[0, 1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

```
B = array(n)
FOR i = 0 TO n-1 DO
  initialize empty list in B[i]
ENDDO
FOR i = 1 TO n DO
  insert A[i] into list B[⌊n · A[i]⌋]
ENDDO
FOR i = 0 TO n-1 DO
  InsertionSort(B[i])
ENDDO
copy B[0],...,B[n-1] in order to A
```

Beispiel:



Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

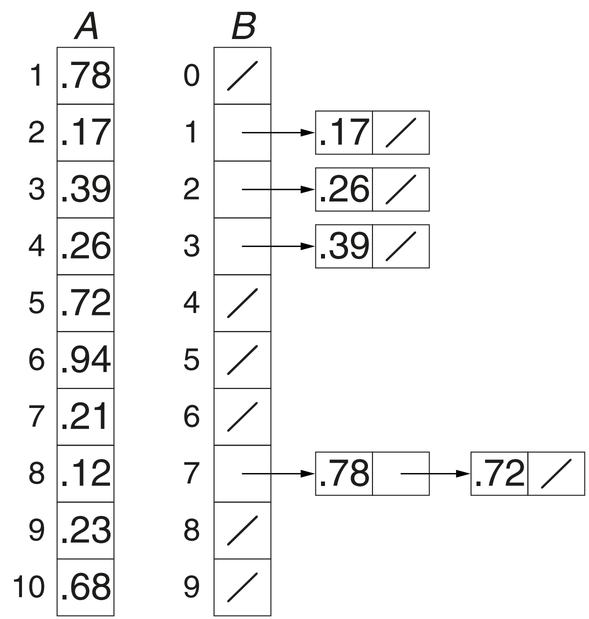
Eingabe: A. Array von n reellen Zahlen aus $[0, 1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

$B =$ *array*(n)
FOR $i = 0$ **TO** $n - 1$ **DO**
  initialize empty list in $B[i]$
ENDDO
FOR $i = 1$ **TO** n **DO**
  insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$
ENDDO
FOR $i = 0$ **TO** $n - 1$ **DO**
  InsertionSort$(B[i])$
ENDDO
copy $B[0], \ldots, B[n-1]$ in order to A

Beispiel:



Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

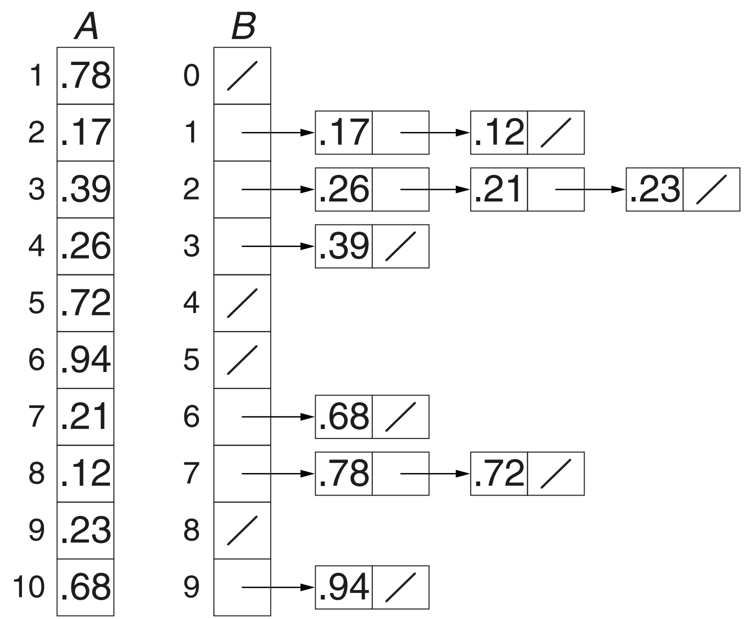
Eingabe: A. Array von n reellen Zahlen aus $[0, 1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

```
B = array(n)
FOR i = 0 TO n − 1 DO
  initialize empty list in B[i]
ENDDO
FOR i = 1 TO n DO
  insert A[i] into list B[⌊n · A[i]⌋]
ENDDO
FOR i = 0 TO n − 1 DO
  InsertionSort(B[i])
ENDDO
copy B[0], . . . , B[n − 1] in order to A
```

Beispiel:



Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

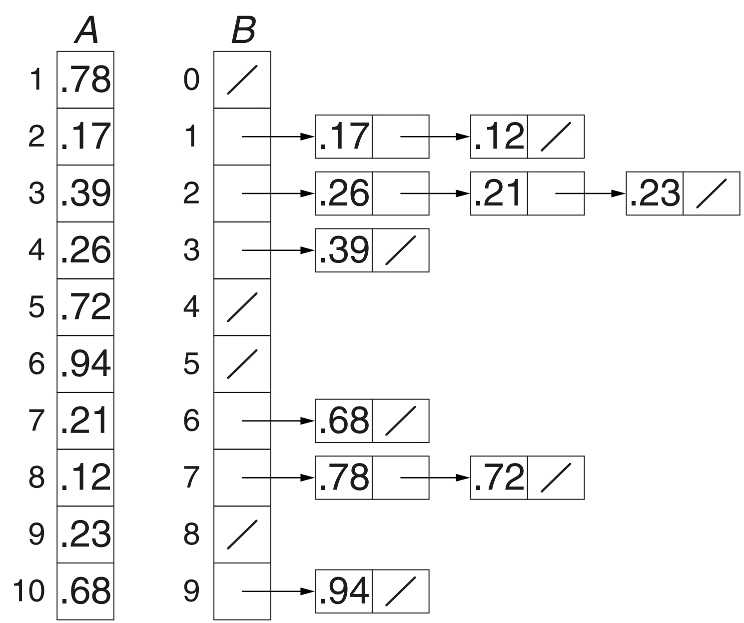
Eingabe: A. Array von n reellen Zahlen aus $[0, 1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

```
B = array(n)
FOR i = 0 TO n − 1 DO
  initialize empty list in B[i]
ENDDO
FOR i = 1 TO n DO
  insert A[i] into list B[⌊n · A[i]⌋]
ENDDO
FOR i = 0 TO n − 1 DO
  InsertionSort(B[i])
ENDDO
copy B[0], . . . , B[n − 1] in order to A
```

Beispiel:



Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

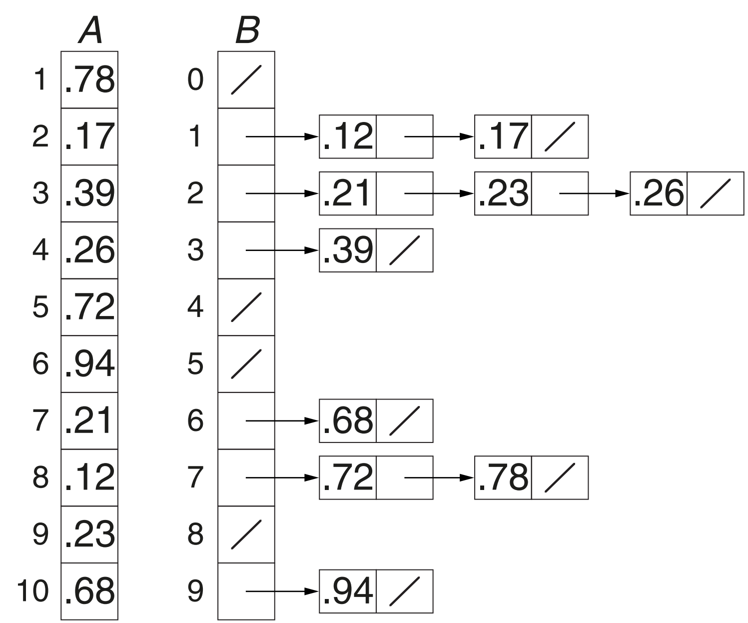
Eingabe: A. Array von n reellen Zahlen aus $[0, 1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

```
B = array(n)
FOR i = 0 TO n − 1 DO
  initialize empty list in B[i]
ENDDO
FOR i = 1 TO n DO
  insert A[i] into list B[⌊n · A[i]⌋]
ENDDO
FOR i = 0 TO n − 1 DO
  InsertionSort(B[i])
ENDDO
copy B[0], ..., B[n − 1] in order to A
```

Beispiel:



Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

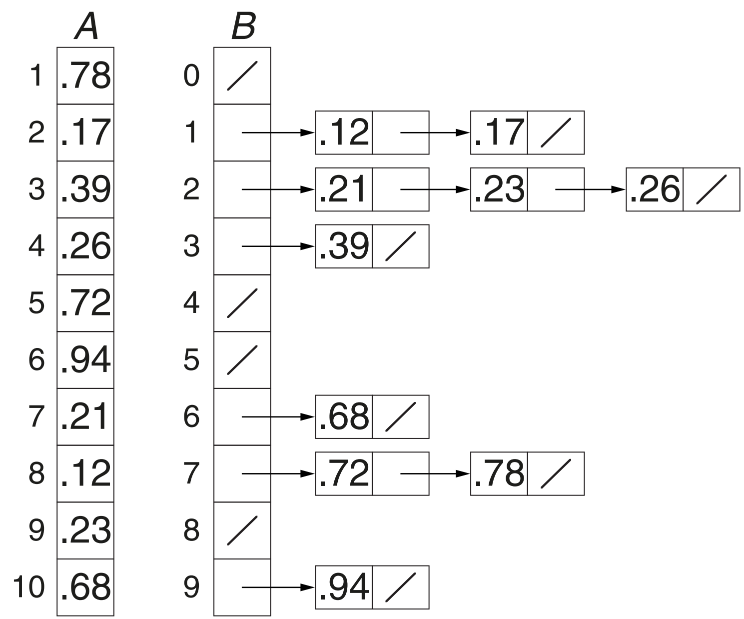
Eingabe: A. Array von n reellen Zahlen aus $[0, 1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

```
B = array(n)
FOR i = 0 TO n − 1 DO
  initialize empty list in B[i]
ENDDO
FOR i = 1 TO n DO
  insert A[i] into list B[⌊n · A[i]⌋]
ENDDO
FOR i = 0 TO n − 1 DO
  InsertionSort(B[i])
ENDDO
copy B[0], ..., B[n − 1] in order to A
```

Beispiel:



Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

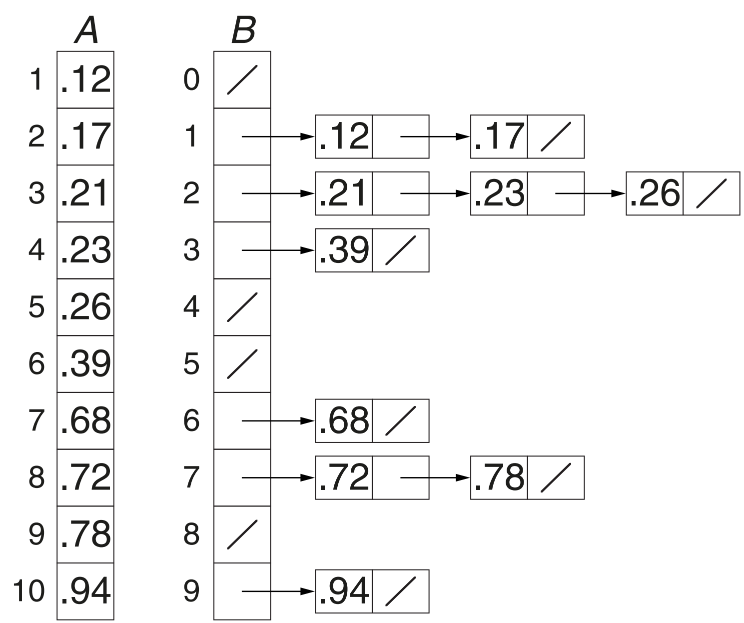
Eingabe: A. Array von n reellen Zahlen aus $[0, 1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

```
B = array(n)
FOR i = 0 TO n-1 DO
  initialize empty list in B[i]
ENDDO
FOR i = 1 TO n DO
  insert A[i] into list B[⌊n·A[i]⌋]
ENDDO
FOR i = 0 TO n-1 DO
  InsertionSort(B[i])
ENDDO
copy B[0],...,B[n-1] in order to A
```

Beispiel:



Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

Eingabe: A. Array von n reellen Zahlen aus $[0,1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

```
 1.  B = array(n)
 2.  FOR i = 0 TO n-1 DO
 3.    initialize empty list in B[i]
 4.  ENDDO
 5.  FOR i = 1 TO n DO
 6.    insert A[i] into list B[⌊n · A[i]⌋]
 7.  ENDDO
 8.  FOR i = 0 TO n-1 DO
 9.    InsertionSort(B[i])
10.  ENDDO
11.  copy B[0],...,B[n-1] in order to A
```

Laufzeit:

- Alle Zeilen außer Zeile 9: $\Theta(n)$.
- Zeile 9: $O(n_i^2)$ für $n_i = |B[i]|$.

➜ $$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Bucket Sort

Algorithmus

Algorithmus: Bucket Sort.

Eingabe: A. Array von n reellen Zahlen aus $[0, 1)$.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

BucketSort(A)

1. $B =$ *array*(n)
2. **FOR** $i = 0$ **TO** $n - 1$ **DO**
3.   initialize empty list in $B[i]$
4. **ENDDO**
5. **FOR** $i = 1$ **TO** n **DO**
6.   insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$
7. **ENDDO**
8. **FOR** $i = 0$ **TO** $n - 1$ **DO**
9.   *InsertionSort*$(B[i])$
10. **ENDDO**
11. copy $B[0], \ldots, B[n-1]$ in order to A

Laufzeit:

- Alle Zeilen außer Zeile 9: $\Theta(n)$.
- Zeile 9: $O(n_i^2)$ für $n_i = |B[i]|$.

➜ $$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

- Worst Case: ein $n_i = n \leadsto T(n) = O(n^2)$
- Best Case: alle $n_i = 1 \leadsto T(n) = \Theta(n)$
- Average Case: $T(n) = \Theta(n)$
  Erwartete Laufzeit auf Grundlage einer probabilistischen Analyse unter Voraussetzung der Gleichverteilung der Werte in A.

Bemerkungen:

- Bucket Sort verteilt die Werte in A auf sogenannte Buckets (*Eimer*), die nacheinander sortiert und deren sortierte Werte dann zu einem sortierten Array zusammengefügt werden.

- Für $A[i] \leq A[j]$ gilt, dass $\lfloor n \cdot A[i] \rfloor \leq \lfloor n \cdot A[j] \rfloor$, so dass $A[i]$ entweder ins gleiche Bucket eingefügt wird wie $A[j]$ oder in ein Bucket mit kleinerem Index. Auf diese Weise genügt es, die Inhalte der Buckets in aufsteigender Reihenfolge nacheinander zusammenzufügen, anstatt sie wie zum Beispiel bei Merge Sort zu vereinen.

- Die Buckets werden unter Verwendung der Datenstruktur Lineare Liste implementiert, die es erlaubt, Elemente nach Bedarf anzufügen. Auf diese Weise wird für jedes Bucket nur so viel zusätzlicher Speicher verwendet, wie nötig ist, um jedes Array-Element aus A zu verteilen.

- Die Funktion $\lfloor n \cdot A[i] \rfloor$ zum Verteilen des i-ten Elements aus A ist eine einfache Variante einer Hashfunktion.

- Die Voraussetzung reeller Zahlen aus dem Intervall $[0, 1)$ kann mit Hilfe von Normalisierung der Werte eines Arrays A in Linearzeit hergestellt werden.

- Die Durchschnittslaufzeit hängt vor allem davon ab, dass die Werte in A einem Prozess entstammen, der sie zufällig gleichverteilt hat entstehen lassen. Andernfalls kann das Laufzeitverhalten von Bucket Sort im Average Case nicht garantiert werden.

Bemerkungen: (Fortsetzung)

- Die Average-Case-Laufzeit von Bucket Sort kann mittels probabilistischer Analyse wie folgt hergeleitet werden.

  Erwartete Laufzeit:

$$E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] = \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$

  Wir betrachten n_i als Zufallsvariable, die angibt, wie viele Elemente im i-ten Bucket landen. Vorausgesetzt, dass die Werte in A unabhängig gleichverteilt gezogen wurden, folgt n_i der Binomialverteilung $B(n,p)$ für die Wahrscheinlichkeit $p = P(A[j]$ fällt in Bucket $i) = 1/n$.

  Aus der Wahrscheinlichkeitstheorie ist bekannt, dass

$$\begin{array}{rrcl} & E[n_i^2] & = & E[n_i]^2 \ + \ \mathrm{Var}(n_i) \\ \Leftrightarrow & E[n_i^2] & = & (n \cdot \frac{1}{n})^2 \ + \ n \cdot \frac{1}{n} \cdot (1 - \frac{1}{n}) \\ \Leftrightarrow & E[n_i^2] & = & 2 - \frac{1}{n} \end{array}$$

  Daraus folgt:

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n) = \Theta(n) + n \cdot O(1) = \Theta(n) \quad \square$$