Einführung in die Programmierung

Vorlesungsteil 8
Sortieren

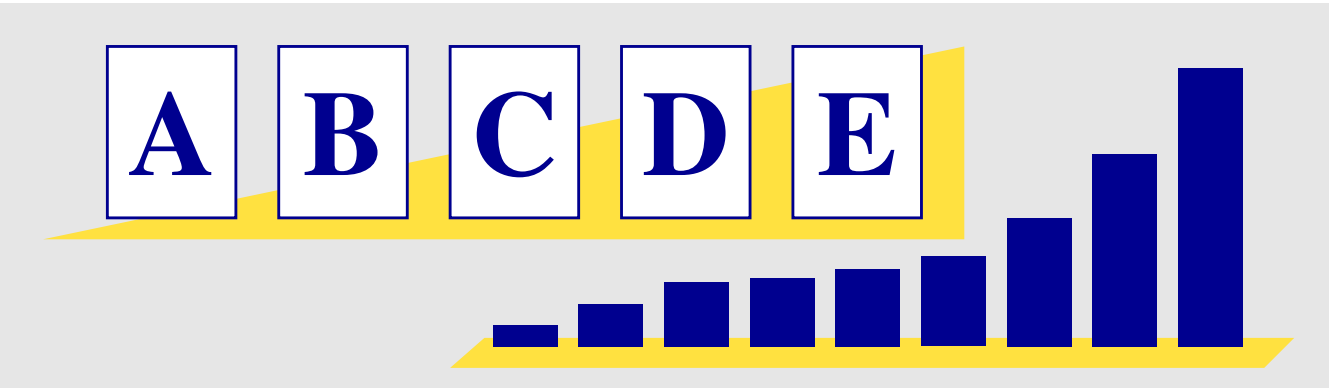
PD Dr. Thomas Hinze

Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik, Informations- und Medientechnik

Wintersemester 2015/2016

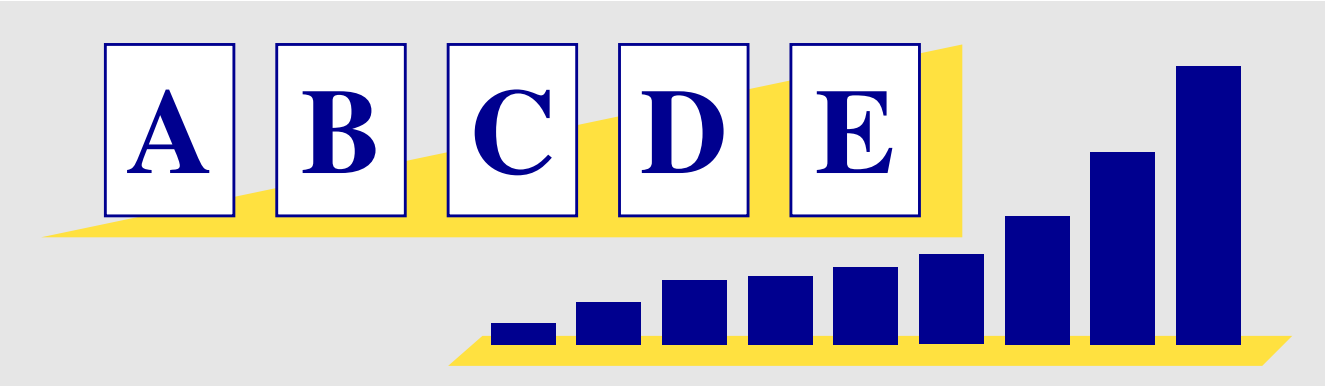
Brandenburgische
Technische Universität
Cottbus - Senftenberg

Sortieren als bedeutende Aufgabenklasse



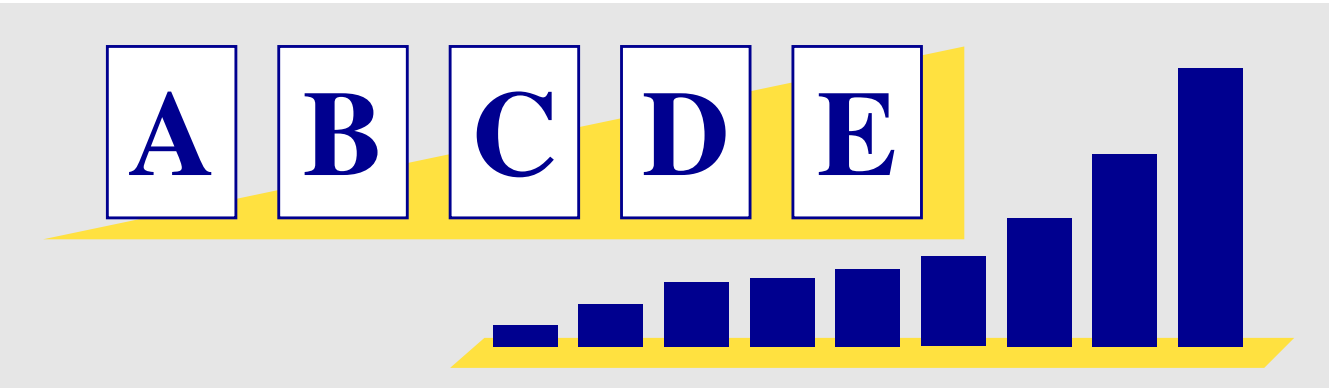
- 20 . . . 25% der Rechenzeit auf Großrechenanlagen entfallen auf Sortieraufgaben (Studie der Stanford University)
- Sortieren als eine der ersten Aufgabenklassen algorithmisch erschlossen

Sortieren als bedeutende Aufgabenklasse



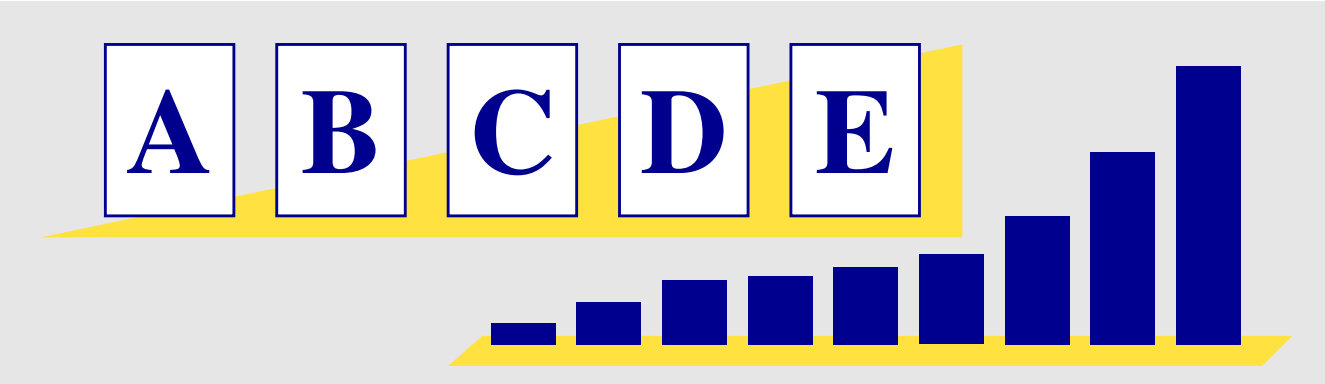
- 20 . . . 25% der Rechenzeit auf Großrechenanlagen entfallen auf Sortieraufgaben (Studie der Stanford University)
- Sortieren als eine der ersten Aufgabenklassen algorithmisch erschlossen
- Mehr als 40 verschiedene Sortierverfahren bekannt und genutzt

Sortieren als bedeutende Aufgabenklasse



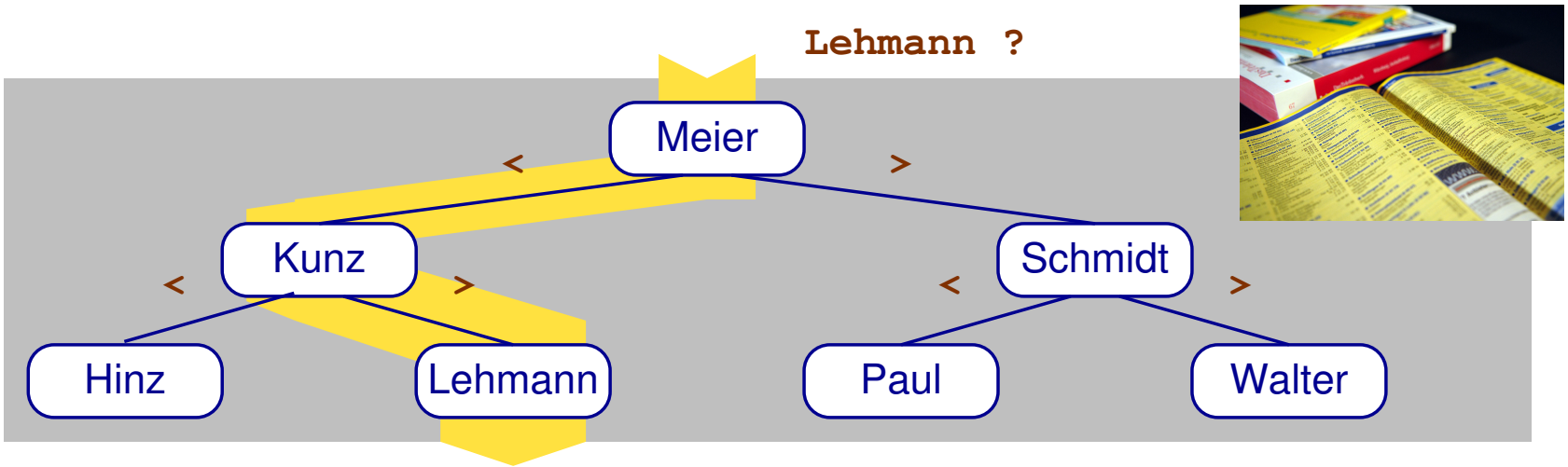
- 20 . . . 25% der Rechenzeit auf Großrechenanlagen entfallen auf Sortieraufgaben (Studie der Stanford University)
- Sortieren als eine der ersten Aufgabenklassen algorithmisch erschlossen
- Mehr als 40 verschiedene Sortierverfahren bekannt und genutzt
- Aufgabenklasse Sortieren in Theorie und Praxis *gut untersucht*

Sortieren als bedeutende Aufgabenklasse



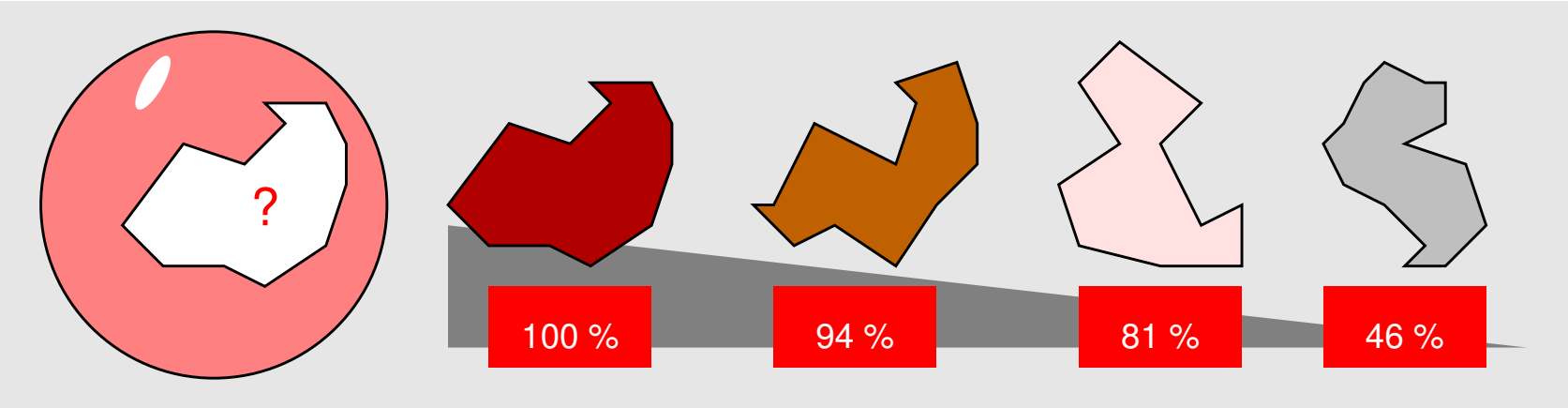
- 20 . . . 25% der Rechenzeit auf Großrechenanlagen entfallen auf Sortieraufgaben (Studie der Stanford University)
- Sortieren als eine der ersten Aufgabenklassen algorithmisch erschlossen
- Mehr als 40 verschiedene Sortierverfahren bekannt und genutzt
- Aufgabenklasse Sortieren in Theorie und Praxis *gut untersucht*
- Sortierverfahren breit gefächert nach konträren Optimierungszielen wie *Schnelligkeit*, wenig *Hilfsspeicherplatzverbrauch* und vorteilhafte Parallelverarbeitung

Sortieren dient effizienter Datenorganisation



- Sortierte Datenbestände lassen sich *deutlich schneller durchsuchen* als unsortierte (z.B. Telefonbuch)
- Datensätze dabei in Baumstrukturen sortiert (*binäre Suchbäume*), so dass mit jedem Vergleichsschritt der verbleibende Suchraum etwa halbiert wird
- Bei Änderungen im Datenbestand Neusortieren erforderlich, was aber häufig viel schneller als die Erstsortierung geschehen kann

Sortieren ist Bestandteil zahlreicher Algorithmen



- in *Greedy*-Algorithmen
- in *Heuristiken* (wie Selektion bei künstlicher Evolution)
- in Verfahren zum Prozess-Scheduling
- in Verfahren zur *statistischen Datenanalyse*
- in Verfahren zur perspektivischen Darstellung grafischer Objekte
- bei *Scoring-Verfahren* (z.B. Ähnlichkeit von Genomsequenzen)

Sortieren dient der Visualisierung von Daten

R	V	Verein	Sp	S	U	N	Tore	TD	P
1	(1)	**Bayern München** (M,P)	14	11	3	0	33:3	+30	**36**
2	(2)	VfL Wolfsburg	14	9	2	3	28:13	+15	**29**
3	(4)	FC Augsburg	14	8	0	6	20:14	+6	**24**
4	(3)	Bayer 04 Leverkusen	14	6	5	3	25:18	+7	**23**
4	(6)	FC Schalke 04	14	7	2	5	25:18	+7	**23**
6	(5)	Borussia Mönchengladbach	14	6	5	3	19:12	+7	**23**
7	(9)	Eintracht Frankfurt	14	6	3	5	27:26	+1	**21**
8	(7)	1899 Hoffenheim	14	5	5	4	21:22	-1	**20**
9	(8)	Hannover 96	14	6	1	7	14:21	-7	**19**
10	(11)	SC Paderborn 07 (N)	14	4	5	5	19:23	-4	**17**
11	(10)	1. FSV Mainz 05	14	3	7	4	17:20	-3	**16**
12	(12)	1. FC Köln (N)	14	4	3	7	14:20	-6	**15**
13	(17)	Hamburger SV	14	4	3	7	9:18	-9	**15**
14	(18)	Borussia Dortmund	14	4	2	8	15:21	-6	**14**
15	(13)	Hertha BSC	14	4	2	8	19:26	-7	**14**
16	(15)	SC Freiburg	14	2	7	5	15:21	-6	**13**
17	(14)	Werder Bremen	14	3	4	7	20:31	-11	**13**
18	(16)	VfB Stuttgart	14	3	3	8	18:31	-13	**12**

www.sportschau.de – Fußball-Bundesliga – Quizfrage: Von wann ist diese Tabelle?

- übersichtliche, leicht erfassbare Aufbereitung von Datenmengen
- Anordnung widerspiegelt Rangfolge und fokussiert Aufmerksamkeit des Betrachters

Sortierverfahren (Auswahl)

Bubblesort

Quicksort

Heapsort

Shellsort

Mergesort

Insertionsort

Bogosort

Radixsort

Selectionsort

Bucketsort

Odd–even sort

Flashsort

Sortiernetz

Bitonisches Mischen

Sortierverfahren (Auswahl)

Bubblesort

Shellsort

Insertionsort

Selectionsort

Odd–even sort

Quicksort

Heapsort

Mergesort

Bogosort

Radixsort

Bucketsort

Flashsort

Sortiernetz

Bitonisches Mischen

elementar: wenig Hilfsspeicherbedarf, leicht implementierbar

Sortierverfahren (Auswahl)

Bubblesort

Shellsort

Insertionsort

Selectionsort

Odd–even sort

Quicksort

Mergesort

Heapsort

Bogosort

Radixsort

Bucketsort

Flashsort

Sortiernetz

Bitonisches Mischen

rekursiv: schnell, elegant, gut skalierbar, Mergesort präferiert

Sortierverfahren (Auswahl)

Bubblesort

Quicksort

Heapsort

Shellsort

Mergesort

Insertionsort

Bogosort

Radixsort

Selectionsort

Bucketsort

Odd–even sort

Flashsort

Sortiernetz

Bitonisches Mischen

Fachverteilen: sehr schnell, kommt ohne Vergleiche aus

Sortierverfahren (Auswahl)

Bubblesort

Quicksort

Heapsort

Shellsort

Mergesort

Insertionsort

Bogosort

Radixsort

Selectionsort

Bucketsort

Odd–even sort

Flashsort

Sortiernetz

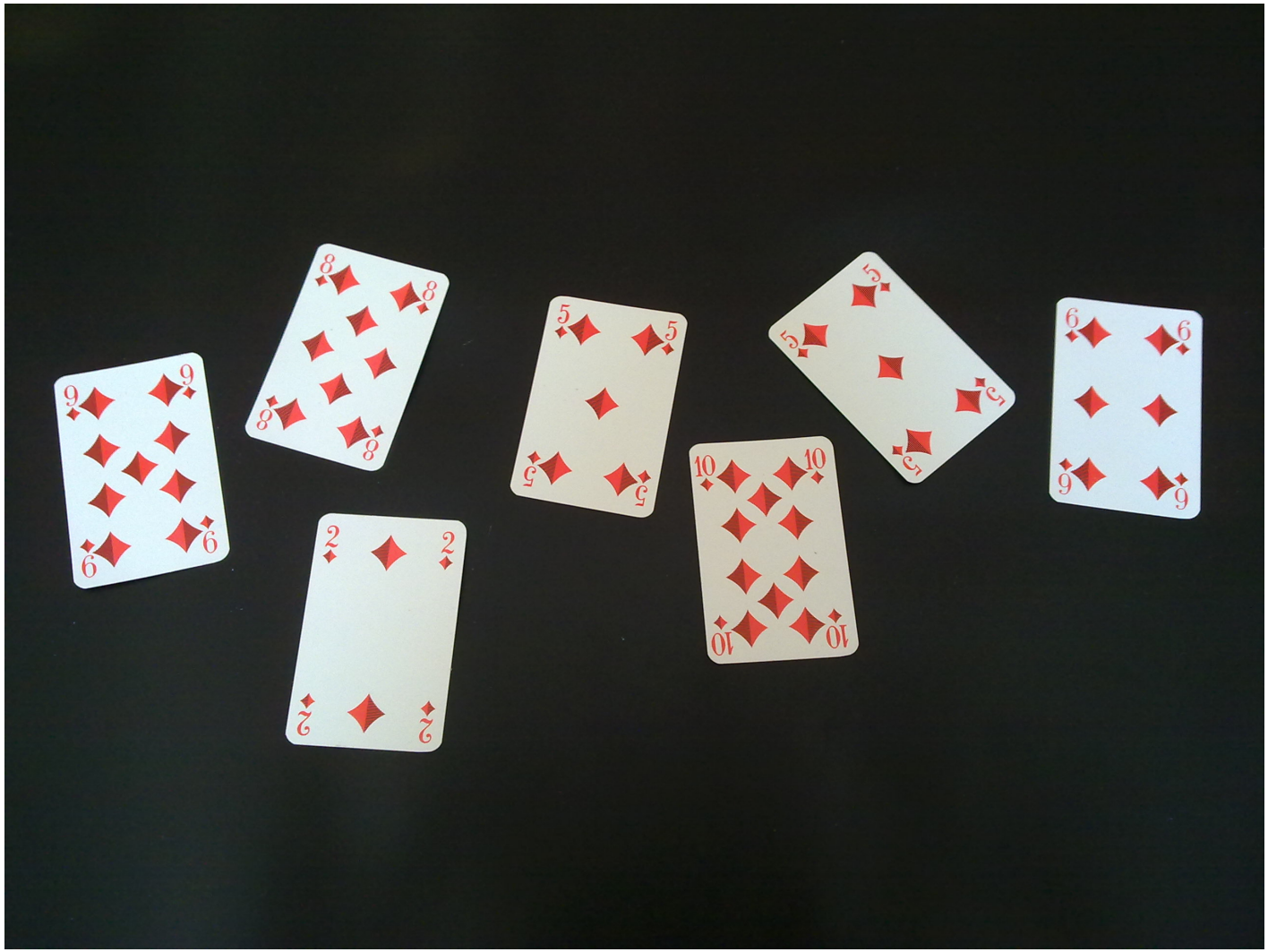
Bitonisches Mischen

parallel: sehr schnell, hardwarenah, datenflussorientiert

Vorlesung Einführung in die Programmierung mit C

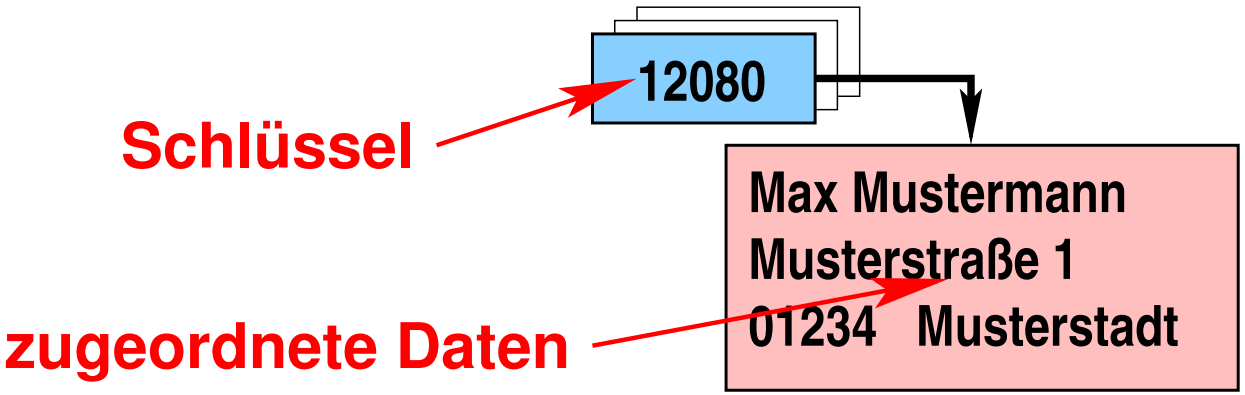
1. **Einführung und erste Schritte** . . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . C als Taschenrechner nutzen, Tastatureingabe → Formelberechnung → Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren
4. **Aussagenlogik in C** . . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren
5. **Funktionen selbst programmieren** . . . . . Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe
6. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
7. **Felder und Strukturierung von Daten** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Datensätzen
8. **Sortieren** . . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf
9. **Zeiger, Zeichenketten und Dateiarbeit** . . . . . Texte analysieren, ver- und entschlüsseln, Dateien lesen und schreiben
10. **Dynamische Datenstruktur „Lineare Liste“** . . . . . unsere selbstprogrammierte kleine Datenbank
11. **Ausblick und weiterführende Konzepte** . . . . .

Unsere Beispiele: Ganze Zahlen aufsteigend sortieren



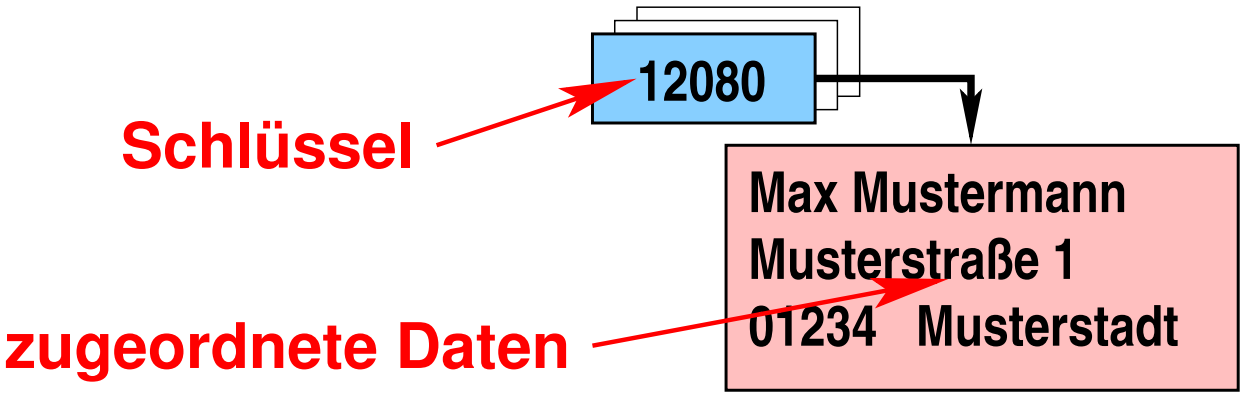
Algorithmen leicht modifizierbar auf Sortieren von Zeichenketten

Datensatz



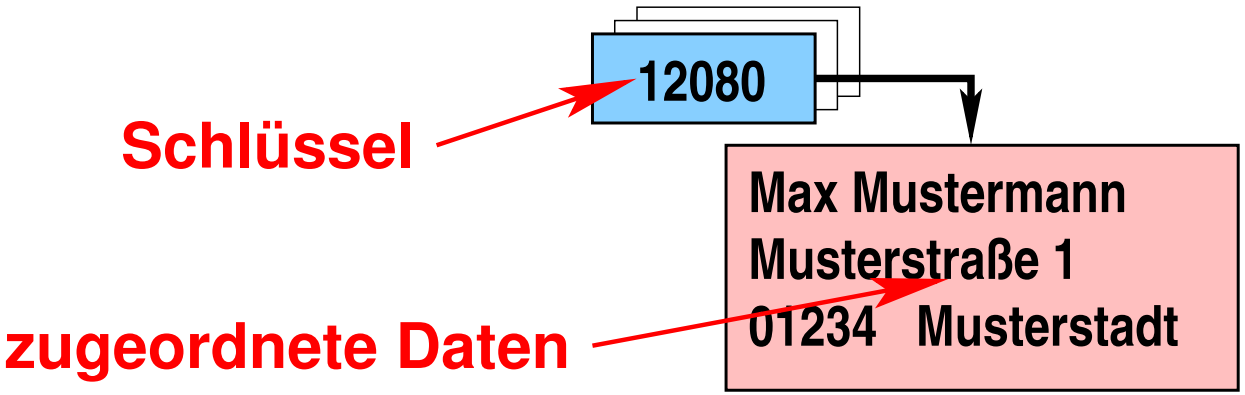
- ***Schlüssel:*** Ordnungsmerkmal für Datensätze, ***aufsteigend*** oder ***absteigend*** anordnen

Datensatz



- ***Schlüssel:*** Ordnungsmerkmal für Datensätze, ***aufsteigend*** oder ***absteigend*** anordnen
- ***Schlüsselduplikate*** beim Sortieren zulässig

Datensatz



- ***Schlüssel:*** Ordnungsmerkmal für Datensätze, ***aufsteigend*** oder ***absteigend*** anordnen
- ***Schlüsselduplikate*** beim Sortieren zulässig
- Ein Sortierverfahren, bei dem Schlüsselduplikate nach dem Sortieren in der gleichen Reihenfolge angeordnet bleiben wie vorher, heißt ***stabil***, anderenfalls instabil.

Annahmen

Datenhaltung

- n Schlüssel in ***Feld*** $a[0], a[1], \ldots, a[n-1]$ gespeichert.

Annahmen

Datenhaltung

- n Schlüssel in ***Feld*** $a[0], a[1], \ldots, a[n-1]$ gespeichert.
- Auf jeden Schlüssel (Feldelement) kann direkt (wahlfrei) zugegriffen werden (***internes Sortieren***).

Annahmen

Datenhaltung

- n Schlüssel in ***Feld*** $a[0], a[1], \ldots, a[n-1]$ gespeichert.
- Auf jeden Schlüssel (Feldelement) kann direkt (wahlfrei) zugegriffen werden (***internes Sortieren***).
- Feld $a[0], a[1], \ldots, a[n-1]$ enthält zu Beginn die Eingabe und nach Sortierende die Ausgabe (Sortieren ***„in-place“*** bzw. ***„in situ“***) – typisch für vergleichsbasierte Verfahren.

Annahmen

Datenhaltung

- n Schlüssel in ***Feld*** $a[0], a[1], \ldots, a[n-1]$ gespeichert.
- Auf jeden Schlüssel (Feldelement) kann direkt (wahlfrei) zugegriffen werden (***internes Sortieren***).
- Feld $a[0], a[1], \ldots, a[n-1]$ enthält zu Beginn die Eingabe und nach Sortierende die Ausgabe (Sortieren ***„in-place“*** bzw. ***„in situ“***) – typisch für vergleichsbasierte Verfahren.

Laufzeitrelevante elementare Arbeitsschritte

- ***Vergleich*** zweier Schlüssel
- ***Umspeichern*** von Schlüsseln
  (z.B. Vertauschen, Einfügen, Verschieben)
- Anzahl benötigte elementare Arbeitsschritte ist Maß für Schnelligkeit (Zeitkomplexität) des Sortierverfahrens

Begriff Sortieren

- Gegeben: n Datensätze mit Schlüsseln $a[0], \ldots, a[n-1]$.

Begriff Sortieren

- Gegeben: n Datensätze mit Schlüsseln $a[0], \ldots, a[n-1]$.
- Zwischen Schlüsseln ***totale Ordnung*** $\leq$ definiert.
- Totale Ordnung: Binäre Relation mit den Eigenschaften
  - ***reflexiv:*** $a[i] \leq a[i]$ (für alle $i = 0, \ldots, n-1$)
  - ***antisymmetrisch:*** Wenn $a[i] \neq a[k]$ und $a[i] \leq a[k]$ dann $a[k] \not\leq a[i]$ (für alle $i, k = 0, \ldots, n-1$)
  - ***transitiv:*** Wenn $a[i] \leq a[k]$ und $a[k] \leq a[p]$ dann auch $a[i] \leq a[p]$ (für alle $i, k, p = 0, \ldots, n-1$)

Begriff Sortieren

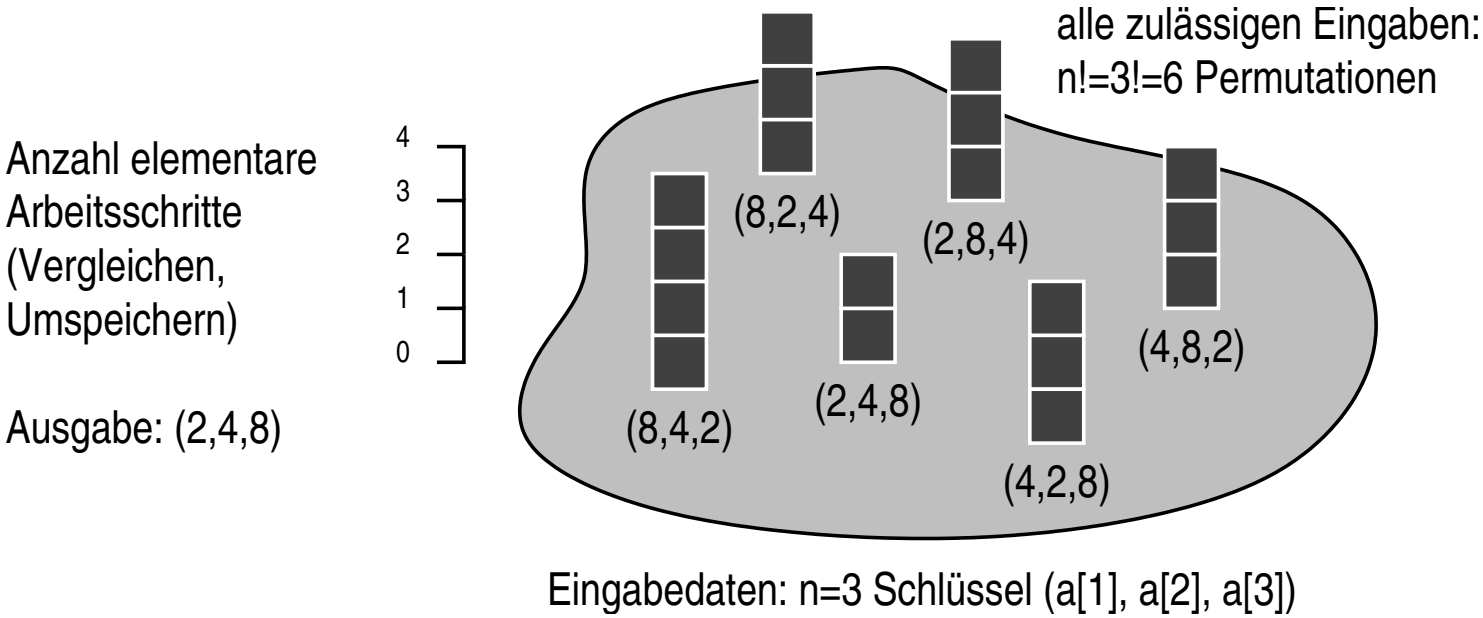
- Gegeben: n Datensätze mit Schlüsseln $a[0], \ldots, a[n-1]$.
- Zwischen Schlüsseln ***totale Ordnung*** $\leq$ definiert.
- Totale Ordnung: Binäre Relation mit den Eigenschaften
  - ***reflexiv:*** $a[i] \leq a[i]$ (für alle $i = 0, \ldots, n-1$)
  - ***antisymmetrisch:*** Wenn $a[i] \neq a[k]$ und $a[i] \leq a[k]$ dann $a[k] \not\leq a[i]$ (für alle $i, k = 0, \ldots, n-1$)
  - ***transitiv:*** Wenn $a[i] \leq a[k]$ und $a[k] \leq a[p]$ dann auch $a[i] \leq a[p]$ (für alle $i, k, p = 0, \ldots, n-1$)
- ***Sortieren:*** Eine ***Permutation*** durch Umindizieren der Schlüssel finden, so dass
  $a[0] \leq a[1] \leq a[2] \leq \ldots \leq a[n-1]$ ***(aufsteigend)*** bzw.
  $a[n-1] \leq \ldots \leq a[2] \leq a[1] \leq a[0]$ ***(absteigend sortiert)***

Begriff Sortieren

- Gegeben: n Datensätze mit Schlüsseln $a[0], \ldots, a[n-1]$.
- Zwischen Schlüsseln ***totale Ordnung*** $\leq$ definiert.
- Totale Ordnung: Binäre Relation mit den Eigenschaften
  - ***reflexiv:*** $a[i] \leq a[i]$ (für alle $i = 0, \ldots, n-1$)
  - ***antisymmetrisch:*** Wenn $a[i] \neq a[k]$ und $a[i] \leq a[k]$ dann $a[k] \not\leq a[i]$ (für alle $i, k = 0, \ldots, n-1$)
  - ***transitiv:*** Wenn $a[i] \leq a[k]$ und $a[k] \leq a[p]$ dann auch $a[i] \leq a[p]$ (für alle $i, k, p = 0, \ldots, n-1$)
- ***Sortieren:*** Eine ***Permutation*** durch Umindizieren der Schlüssel finden, so dass
  $a[0] \leq a[1] \leq a[2] \leq \ldots \leq a[n-1]$ ***(aufsteigend)*** bzw.
  $a[n-1] \leq \ldots \leq a[2] \leq a[1] \leq a[0]$ ***(absteigend sortiert)***
- n paarweise verschiedene Datensätze lassen sich in $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$ Permutationen anordnen (***Eingaberaumgröße***).

Idee der Laufzeitmessung

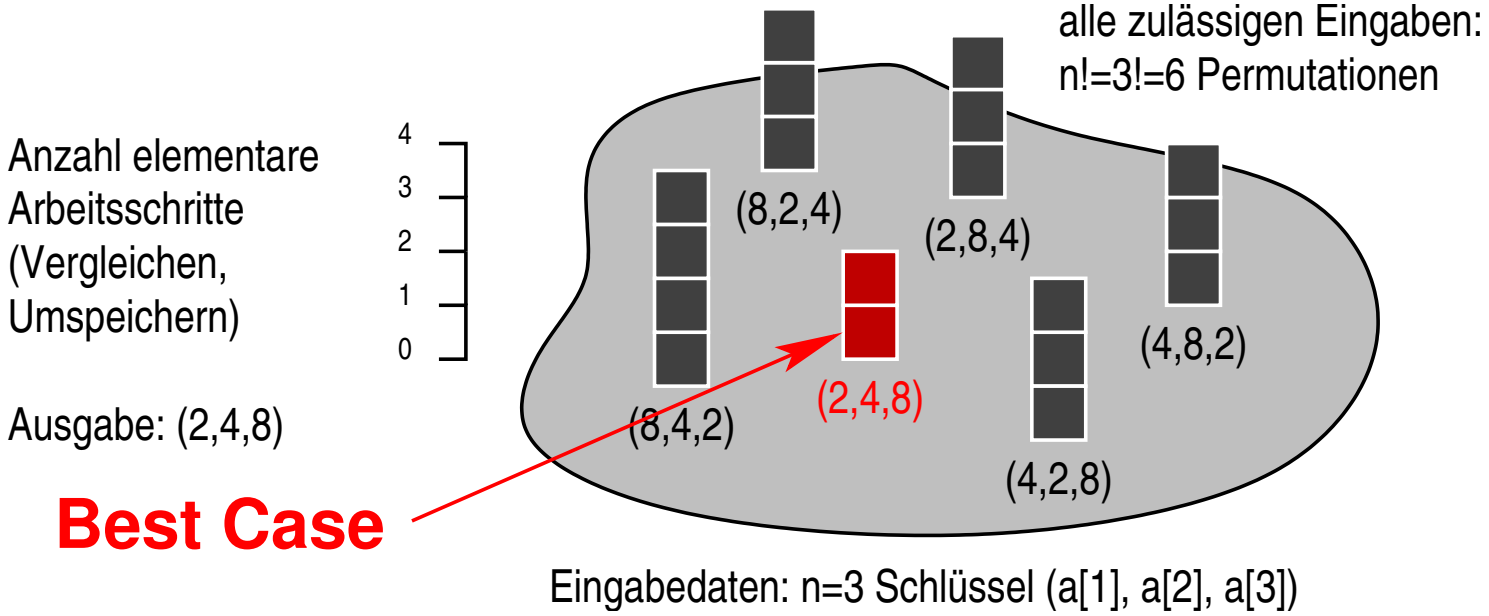
Gegeben sei ein beliebiger in-place Sortieralgorithmus.



- Für jede Problemgröße *n* (Anzahl Schlüssel) und
- alle zulässigen Eingaben (*n*! Permutationen) wird die
- Anzahl benötigter elementarer Arbeitsschritte (Vergleichen, Umspeichern) gezählt bzw. ermittelt.

Best Case

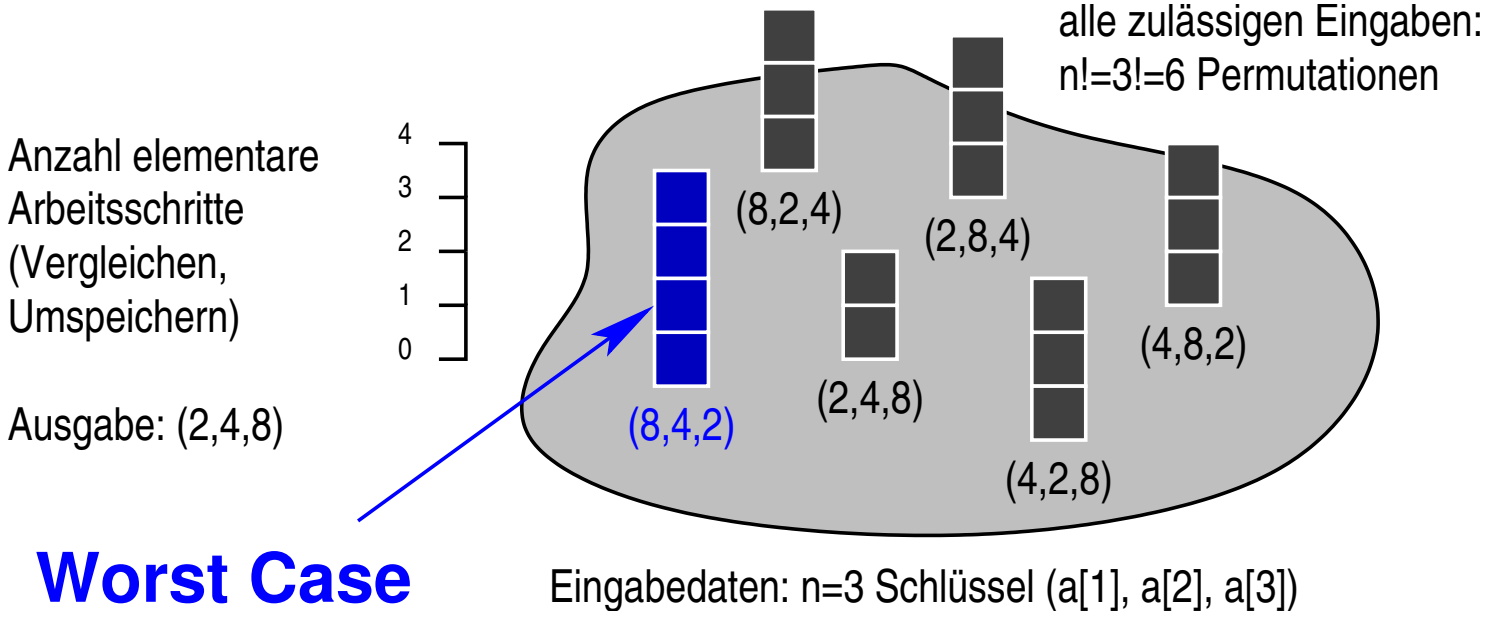
Gegeben sei ein fiktiver in-place Sortieralgorithmus.



- Diejenige Eingabe, bei der der Algorithmus die wenigsten elementaren Arbeitsschritte benötigt.
- Für jede Problemgröße n ergibt sich somit eine minimale Arbeitsschrittanzahl $\mathrm{f}_{\min}(n)$. (Im Beispiel: $\mathrm{f}_{\min}(3) = 2$)

Worst Case

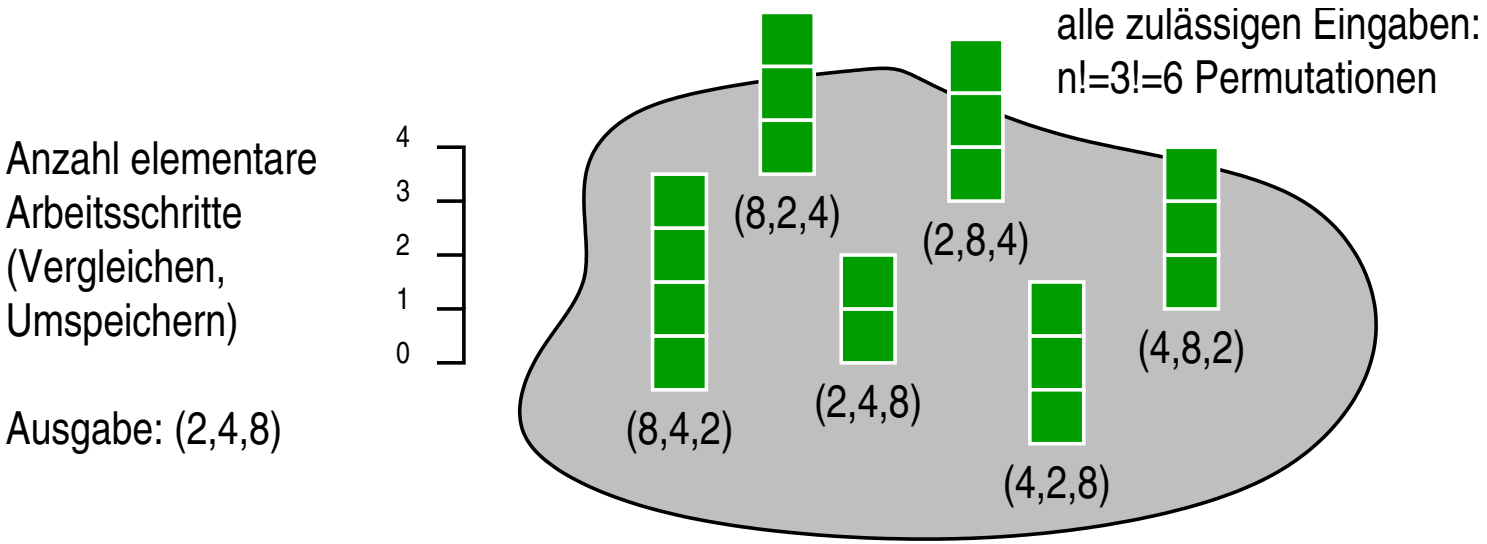
Gegeben sei ein fiktiver in-place Sortieralgorithmus.



- Diejenige Eingabe, bei der der Algorithmus die meisten elementaren Arbeitsschritte benötigt.
- Für jede Problemgröße n ergibt sich somit eine maximale Arbeitsschrittanzahl $\mathrm{f}_{\max}(n)$. (Im Beispiel: $\mathrm{f}_{\max}(3) = 4$)

Average Case

Gegeben sei ein fiktiver in-place Sortieralgorithmus.



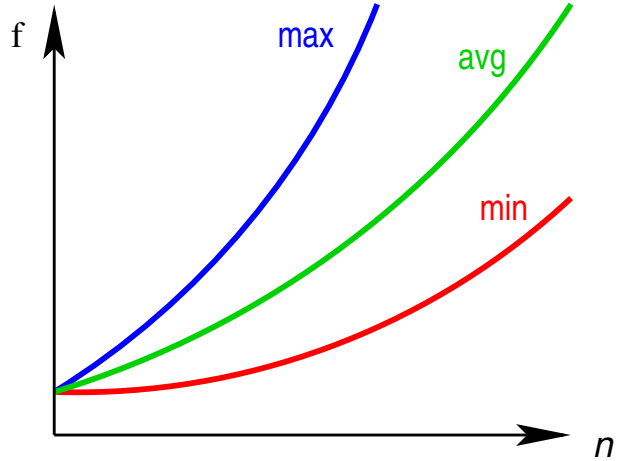
Average Case Eingabedaten: n=3 Schlüssel (a[1], a[2], a[3])

(4+3+2+3+3+3)/6 = 3 Arbeitsschritte im Mittel

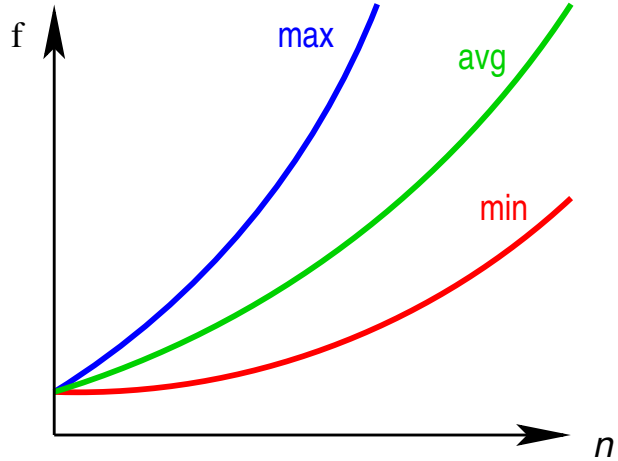
- Für jede Problemgröße *n* ergibt sich eine mittlere Arbeitsschrittanzahl $\mathrm{f}_{\mathrm{avg}}(n)$. (Im Beispiel: $\mathrm{f}_{\mathrm{avg}}(3) = 3$)

Wachstumsklassen (I)

- Für die Funktionen $f_{min}(n)$, $f_{max}(n)$ und $f_{avg}(n)$ zu einem Algorithmus können recht komplizierte mathematische Ausdrücke entstehen.

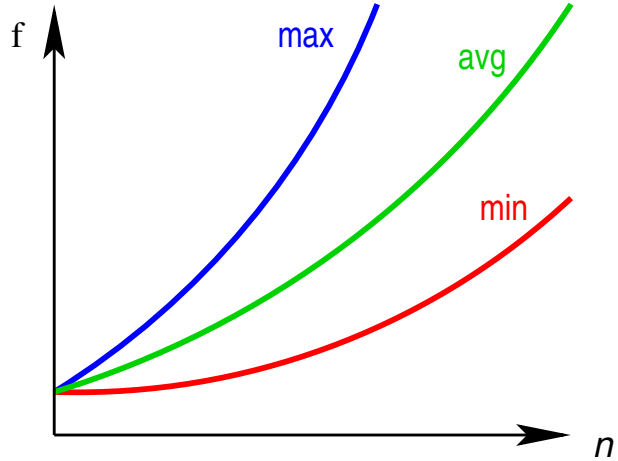


Wachstumsklassen (I)



- Für die Funktionen $f_{min}(n)$, $f_{max}(n)$ und $f_{avg}(n)$ zu einem Algorithmus können recht komplizierte mathematische Ausdrücke entstehen.
- Mit immer größer werdendem n sind jedoch nur Teile im Funktionsausdruck für den Wertezuwachs maßgeblich. (z.B. $f(n) = 2^n + 5 \cdot n^3 + 2 \cdot n$. Hier bestimmt der Term $g(n) = 2^n$ das Wachstum.)

Wachstumsklassen (I)



- Für die Funktionen $f_{min}(n)$, $f_{max}(n)$ und $f_{avg}(n)$ zu einem Algorithmus können recht komplizierte mathematische Ausdrücke entstehen.
- Mit immer größer werdendem n sind jedoch nur Teile im Funktionsausdruck für den Wertezuwachs maßgeblich. (z.B. $f(n) = 2^n + 5 \cdot n^3 + 2 \cdot n$. Hier bestimmt der Term $g(n) = 2^n$ das Wachstum.)
- Entsprechend fasst man Funktionen $f(n)$ mit qualitativ gleichem Wachstumsverhalten zu einer ***Wachstumsklasse*** $O(g(n))$ zusammen, die nach dem bestimmenden Term $g(n)$ benannt wird, Notation: $f(n) \in O(g(n))$

Wachstumsklassen (II)

- Man kann durch ein Regelwerk ausrechnen, zu welcher Wachstumsklasse eine gegebene Funktion $\mathrm{f}(n)$ gehört.

Wachstumsklassen (II)

- Man kann durch ein Regelwerk ausrechnen, zu welcher Wachstumsklasse eine gegebene Funktion $\mathrm{f}(n)$ gehört.
- Konstante Faktoren c werden hier (leider!) vernachlässigt: $O(\mathrm{c} \cdot \mathrm{f}(n)) = O(\mathrm{f}(n))$

Wachstumsklassen (II)

- Man kann durch ein Regelwerk ausrechnen, zu welcher Wachstumsklasse eine gegebene Funktion $\mathrm{f}(n)$ gehört.
- Konstante Faktoren c werden hier (leider!) vernachlässigt: $O(\mathrm{c} \cdot \mathrm{f}(n)) = O(\mathrm{f}(n))$
- Merke: $O(1) \subset O(\log(n)) \subset O(n) \subset O(n \cdot \log(n)) \subset O(n^2) \subset O(2^n) \subset O(n!)$
  (von links nach rechts: steigende Komplexität)

Wachstumsklassen (II)

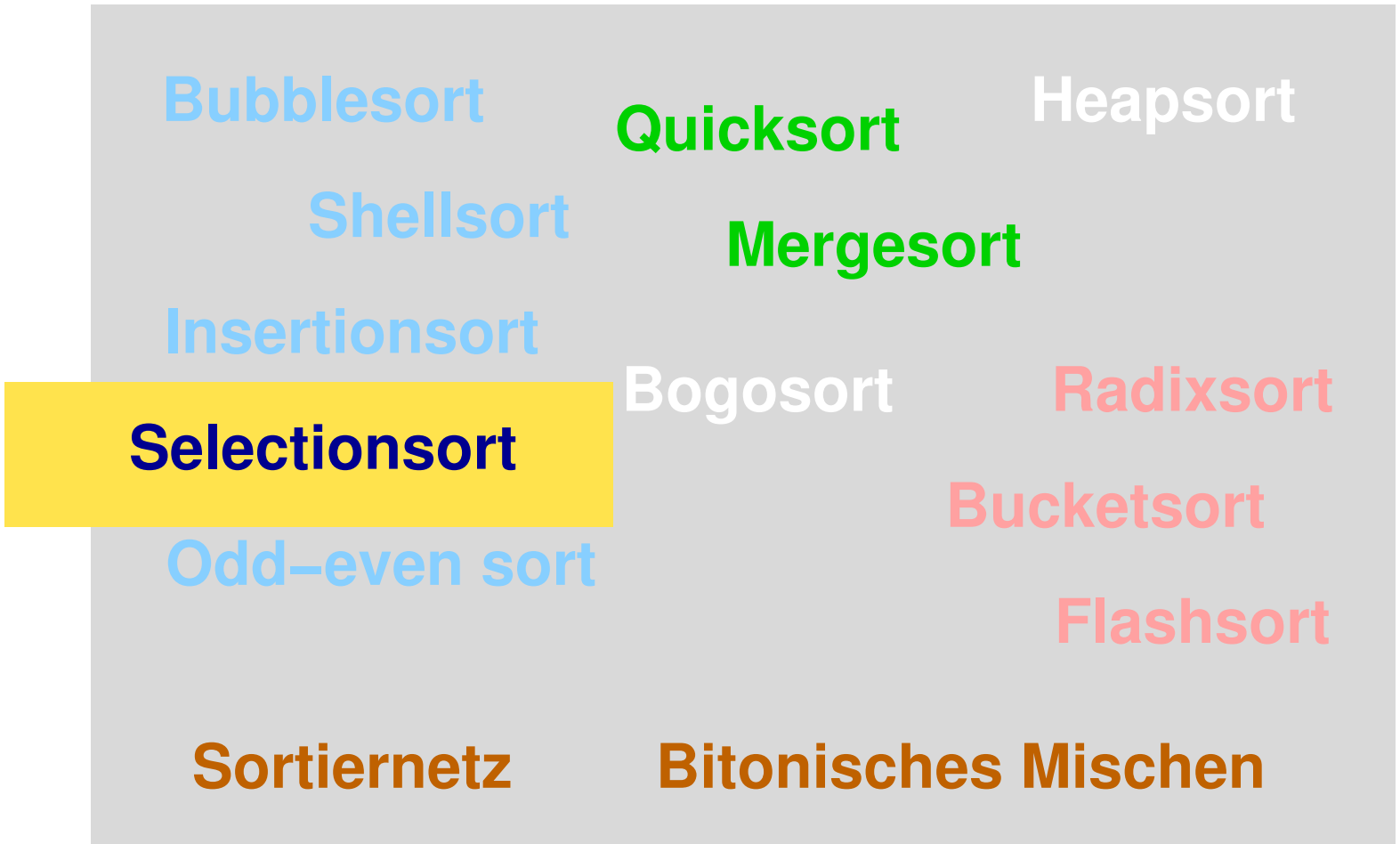
- Man kann durch ein Regelwerk ausrechnen, zu welcher Wachstumsklasse eine gegebene Funktion $\mathrm{f}(n)$ gehört.
- Konstante Faktoren c werden hier (leider!) vernachlässigt: $O(\mathrm{c} \cdot \mathrm{f}(n)) = O(\mathrm{f}(n))$
- Merke: $O(1) \subset O(\log(n)) \subset O(n) \subset O(n \cdot \log(n)) \subset O(n^2) \subset O(2^n) \subset O(n!)$
  (von links nach rechts: steigende Komplexität)
- $O(\mathrm{f}_{\max}(n))$ und $O(\mathrm{f}_{\mathrm{avg}}(n))$ kennzeichnen die ***Zeitkomplexität*** (Effizienz) eines Algorithmus.

Wachstumsklassen (II)

- Man kann durch ein Regelwerk ausrechnen, zu welcher Wachstumsklasse eine gegebene Funktion $\mathrm{f}(n)$ gehört.
- Konstante Faktoren c werden hier (leider!) vernachlässigt: $O(\mathrm{c} \cdot \mathrm{f}(n)) = O(\mathrm{f}(n))$
- Merke: $O(1) \subset O(\log(n)) \subset O(n) \subset O(n \cdot \log(n)) \subset O(n^2) \subset O(2^n) \subset O(n!)$
  (von links nach rechts: steigende Komplexität)
- $O(\mathrm{f}_{\max}(n))$ und $O(\mathrm{f}_{\mathrm{avg}}(n))$ kennzeichnen die ***Zeitkomplexität*** (Effizienz) eines Algorithmus.
- Algorithmen bis zur Wachstumsklasse $O(n^2)$ gelten als effizient.

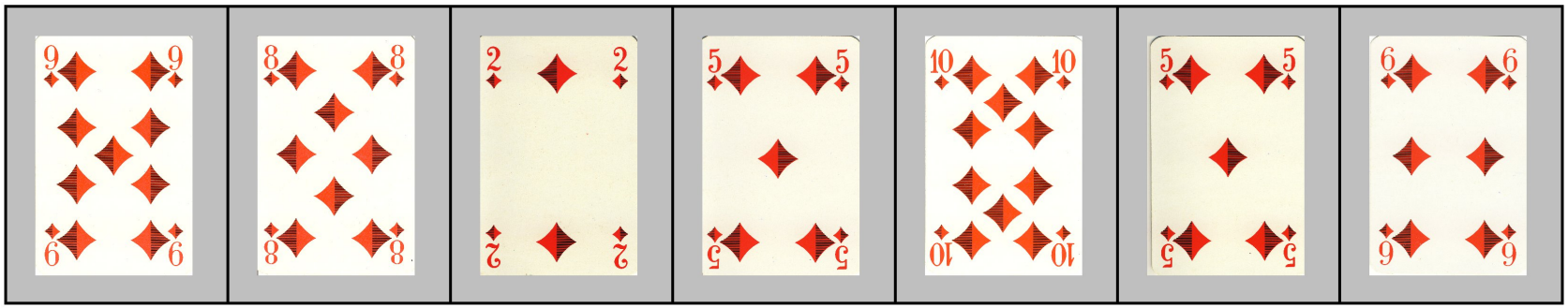
Selectionsort

Sortieren durch fortlaufendes Auswählen des kleinsten Elements



Selectionsort – Idee

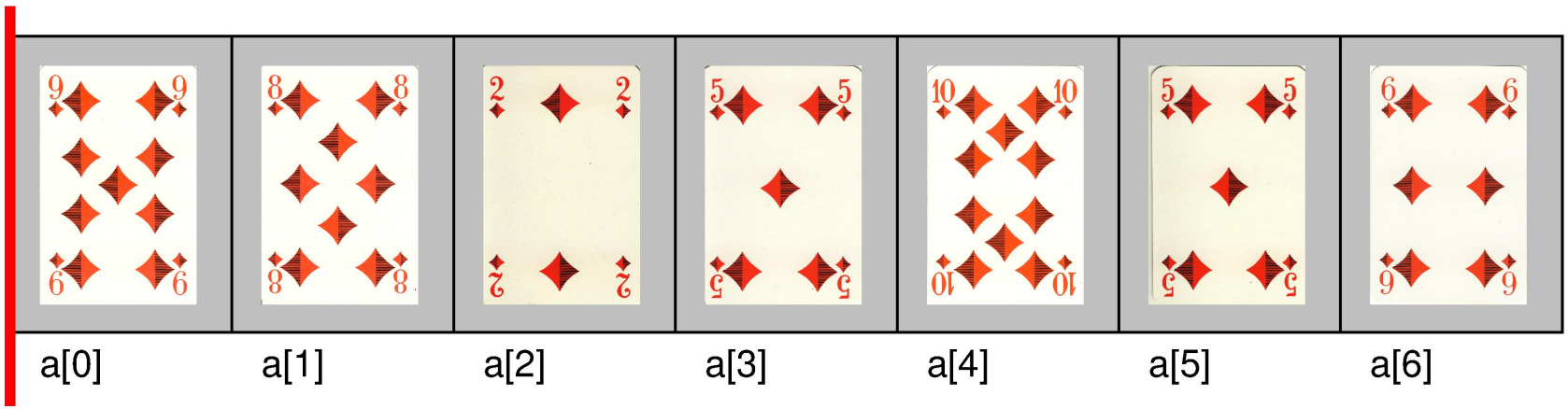
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[0]`

Selectionsort – Idee

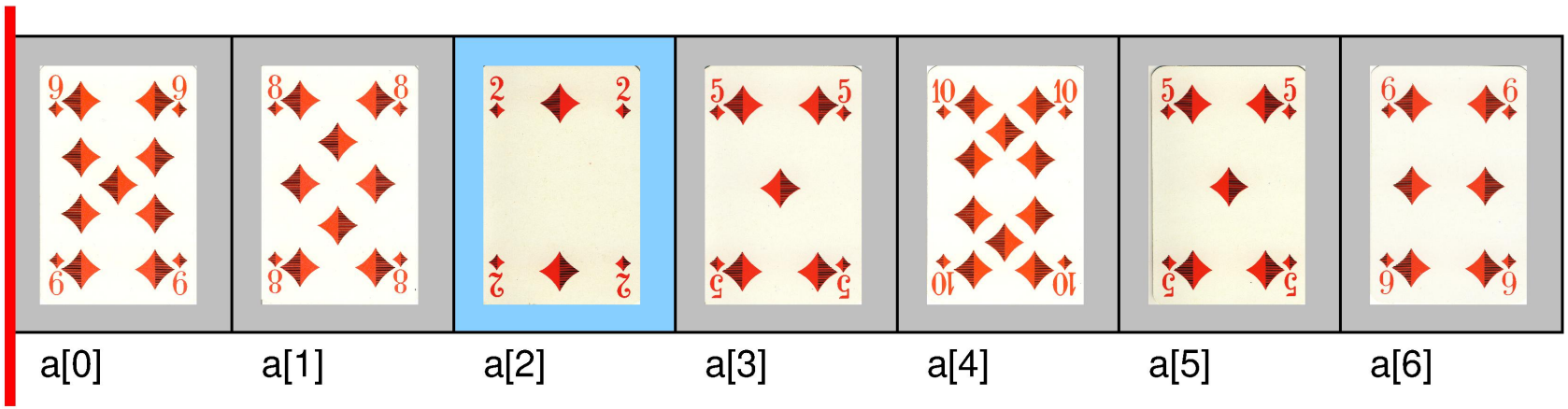
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[0]`

Selectionsort – Idee

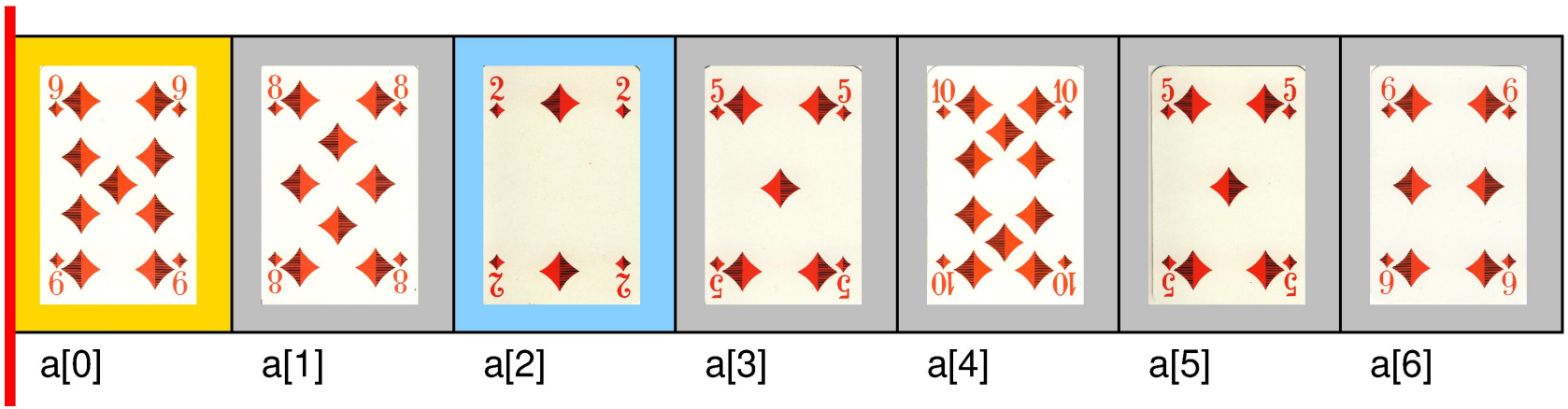
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[0]`

Selectionsort – Idee

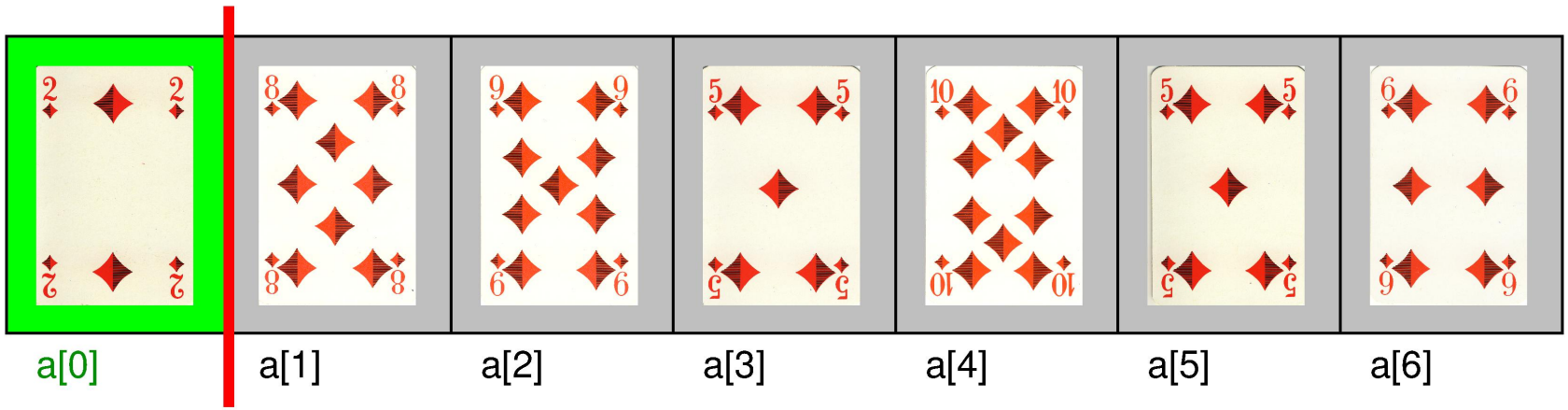
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[0]`

Selectionsort – Idee

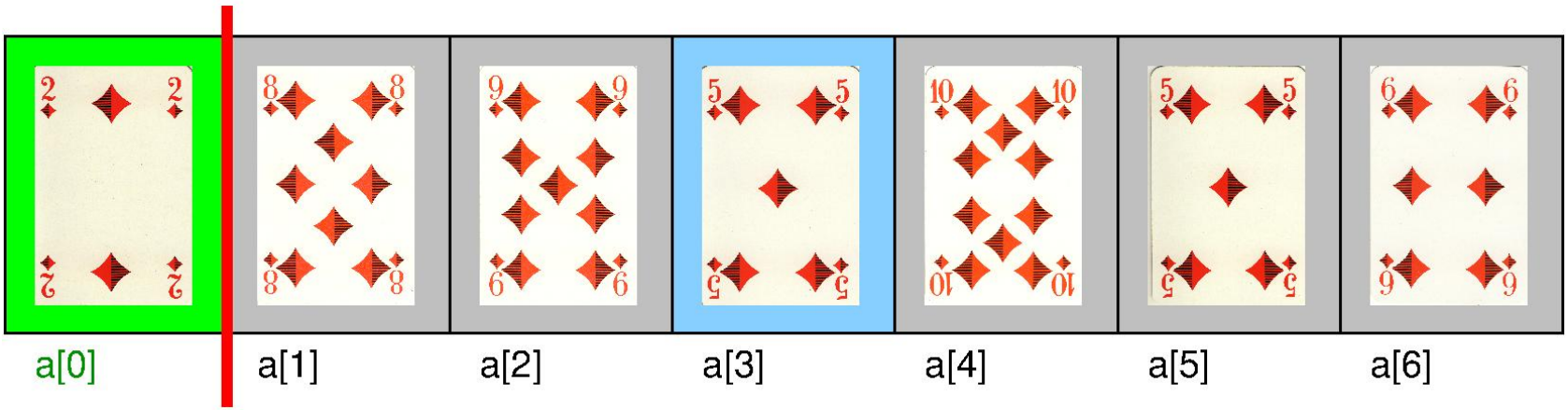
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Fertig sortierter Bereich enthält jetzt `a[0]`

Selectionsort – Idee

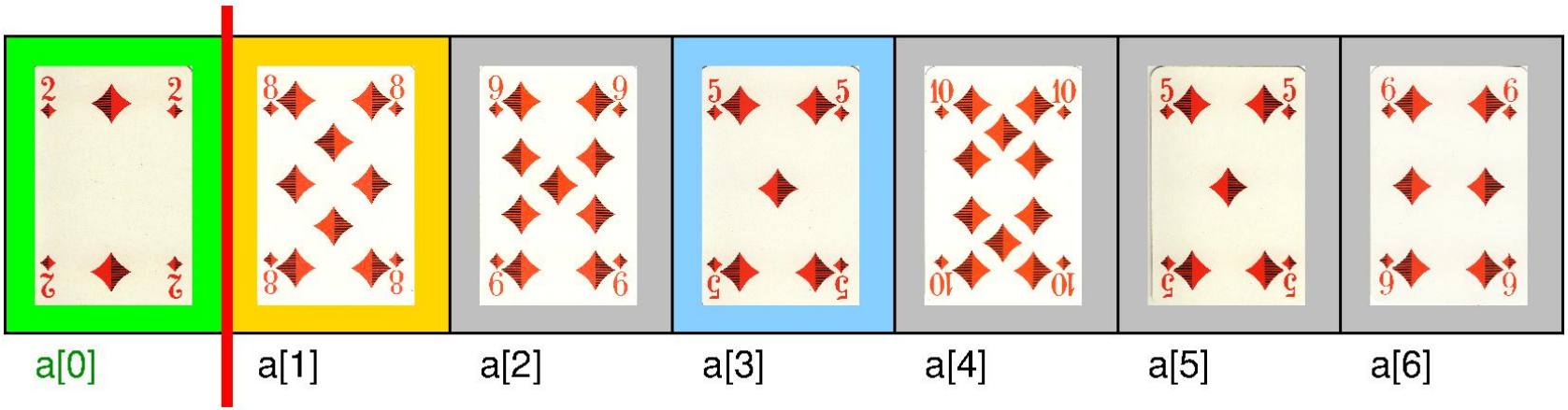
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[1]`

Selectionsort – Idee

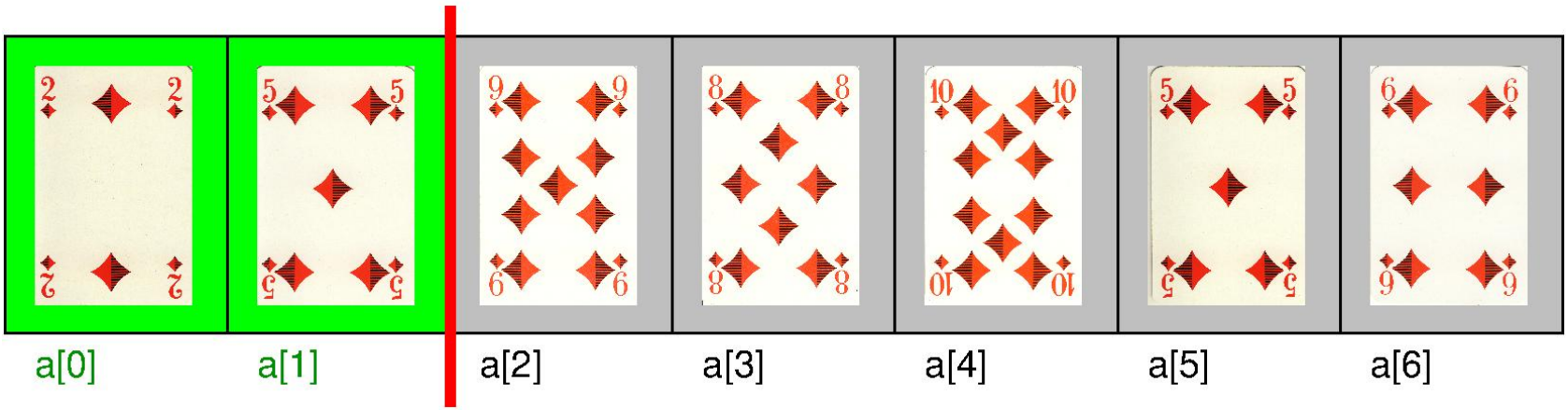
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[1]`

Selectionsort – Idee

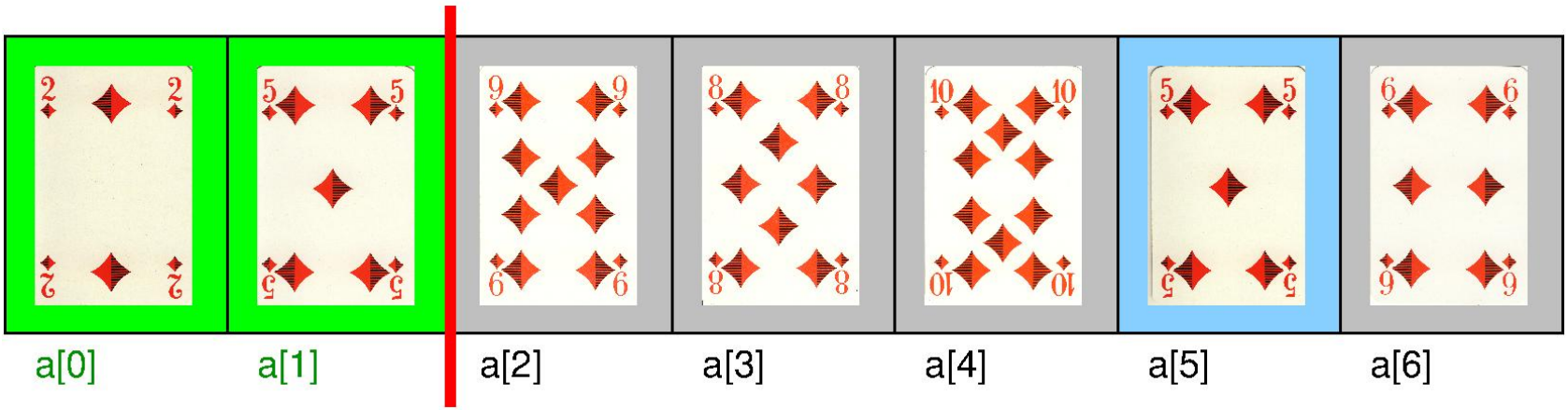
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Fertig sortierter Bereich enthält jetzt `a[0]...a[1]`

Selectionsort – Idee

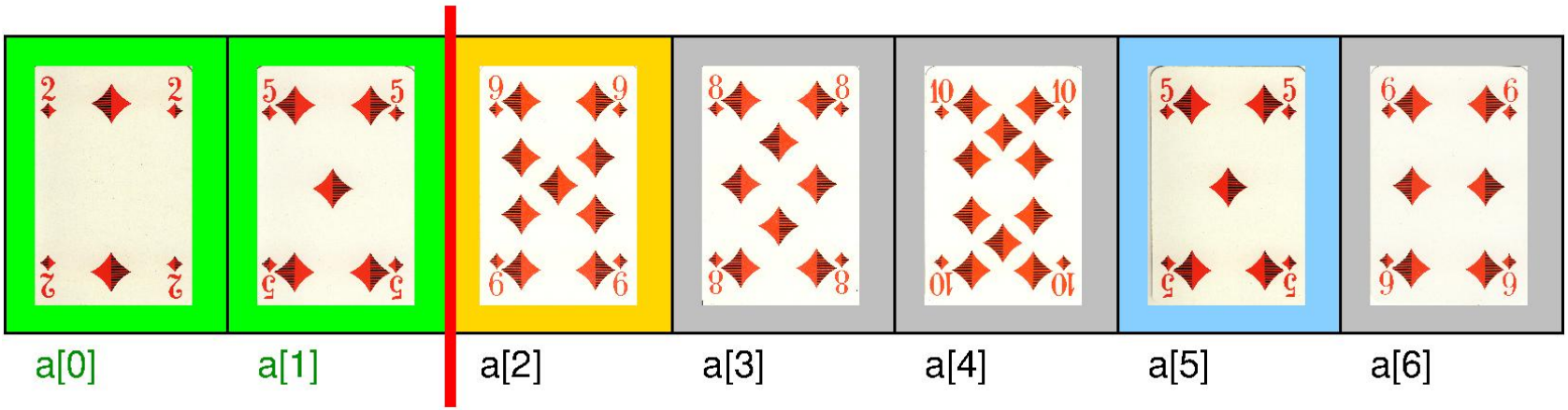
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[2]`

Selectionsort – Idee

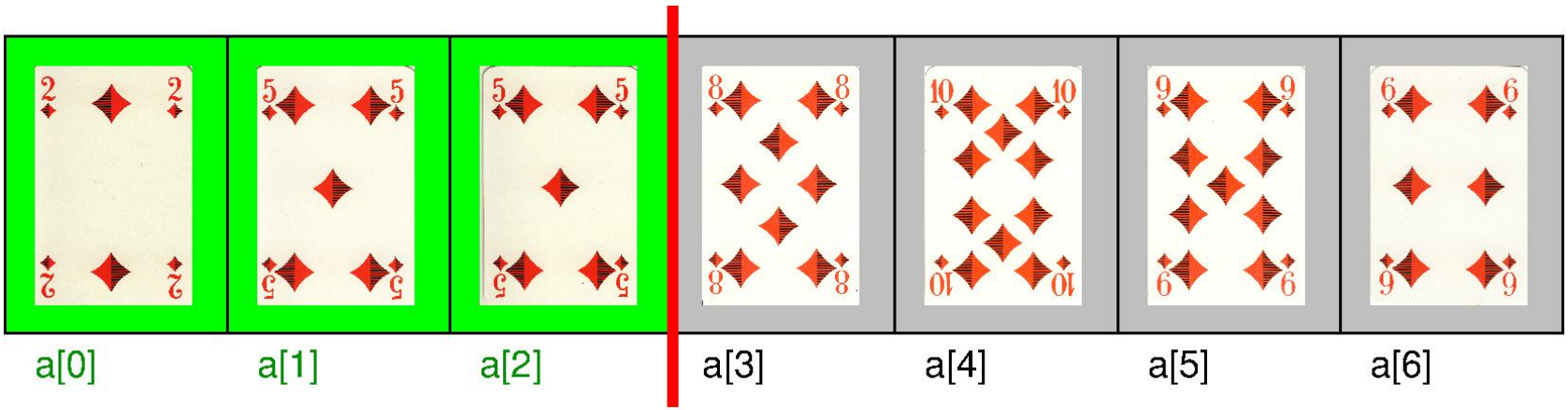
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[2]`

Selectionsort – Idee

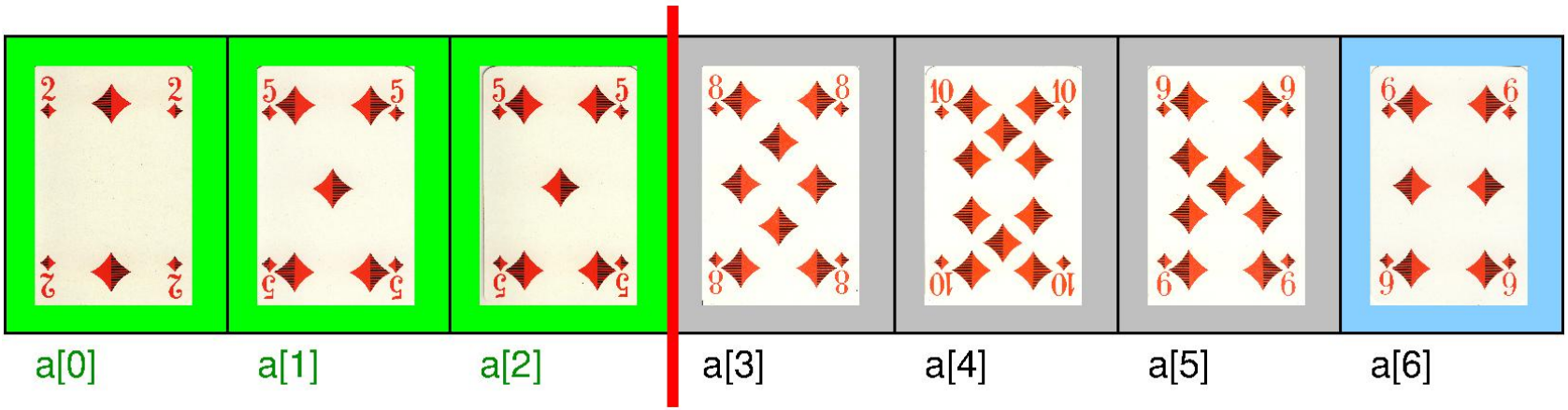
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Fertig sortierter Bereich enthält jetzt `a[0]...a[2]`

Selectionsort – Idee

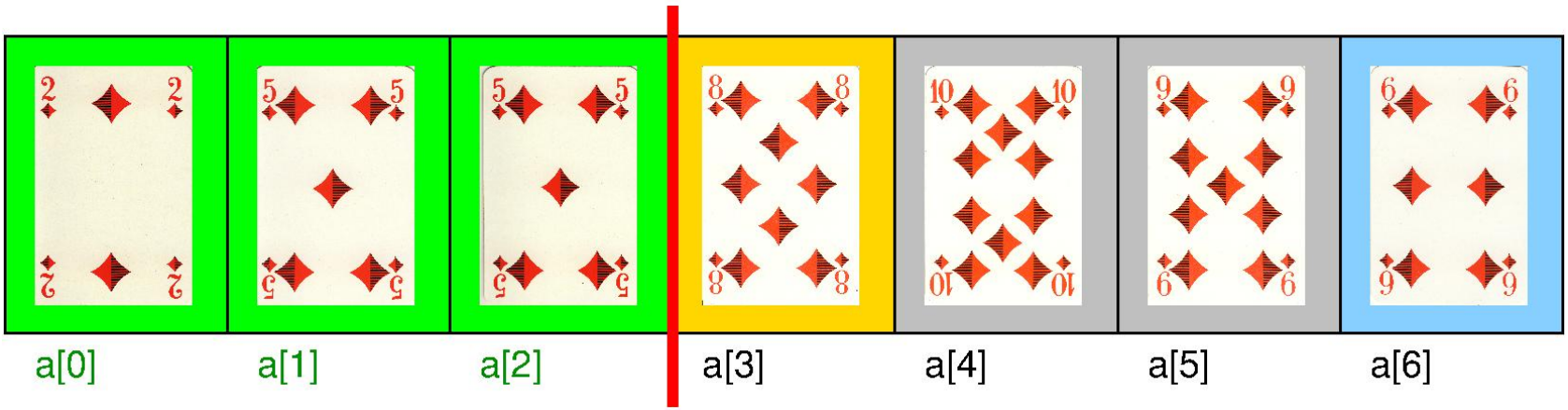
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[3]`

Selectionsort – Idee

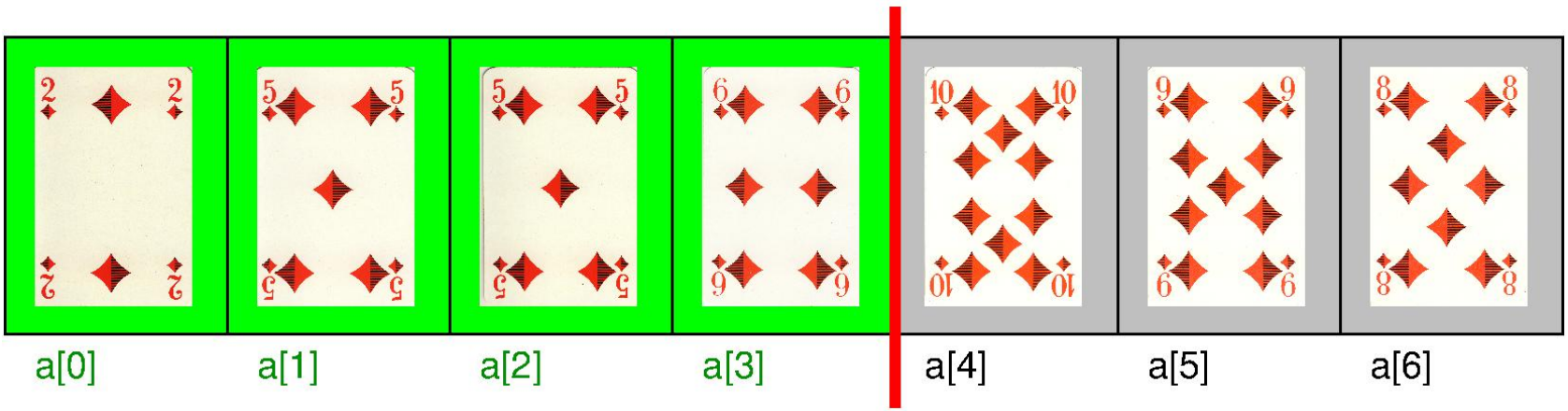
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[3]`

Selectionsort – Idee

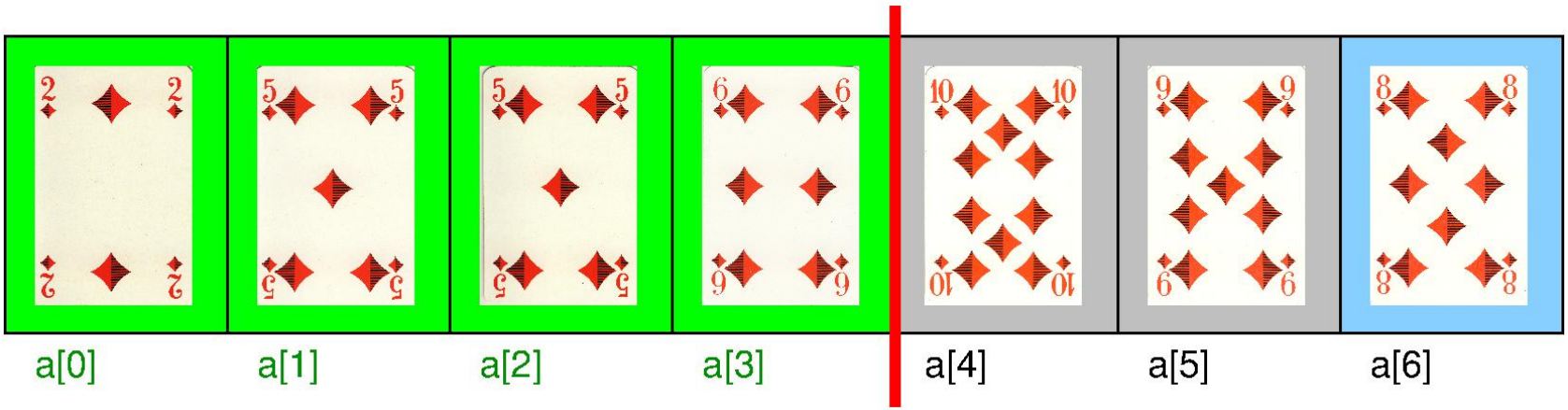
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Fertig sortierter Bereich enthält jetzt `a[0]...a[3]`

Selectionsort – Idee

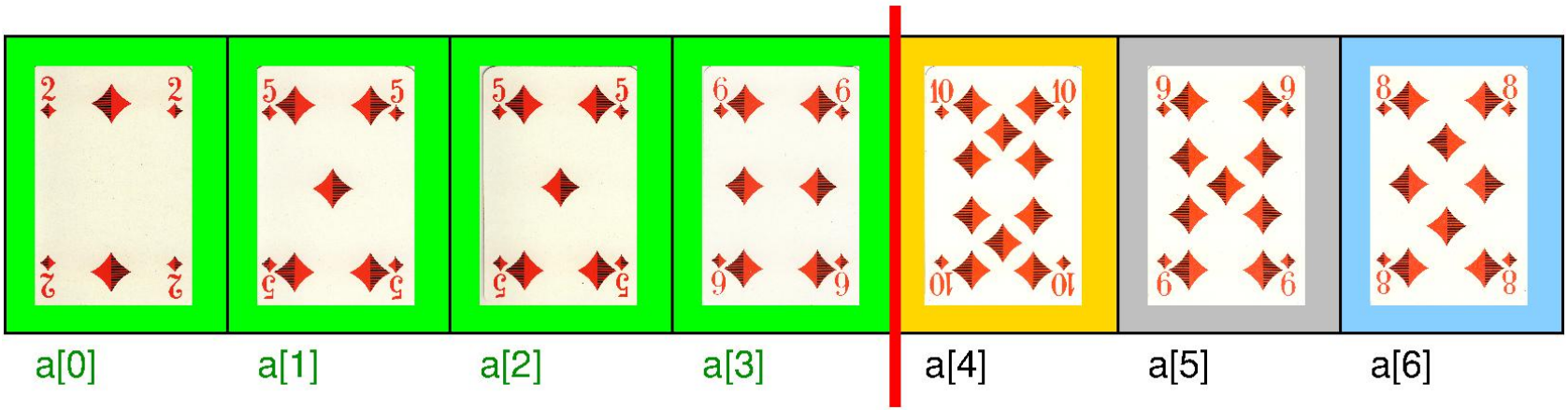
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[4]`

Selectionsort – Idee

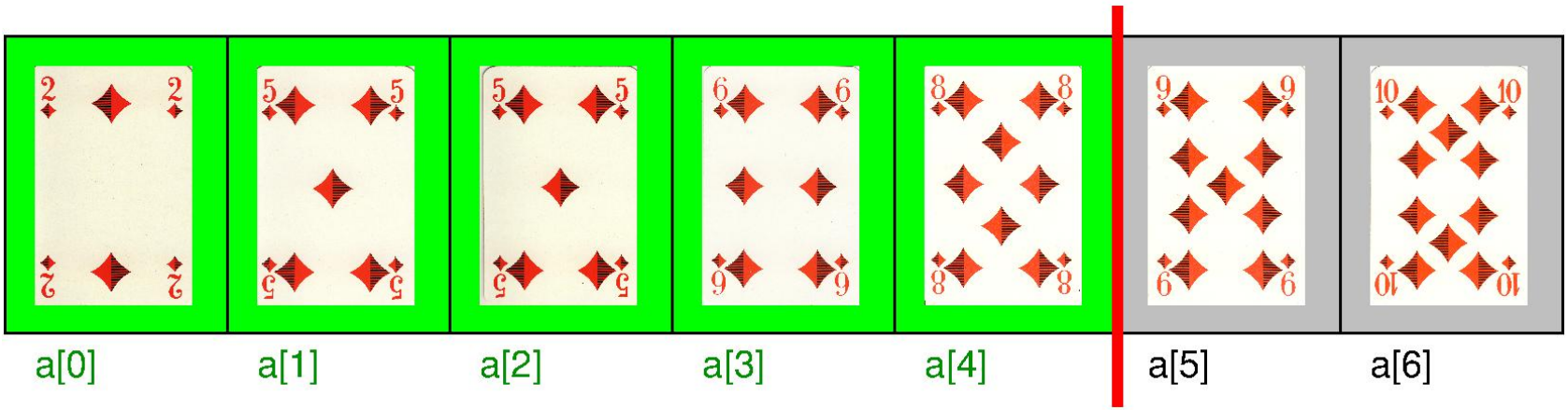
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[4]`

Selectionsort – Idee

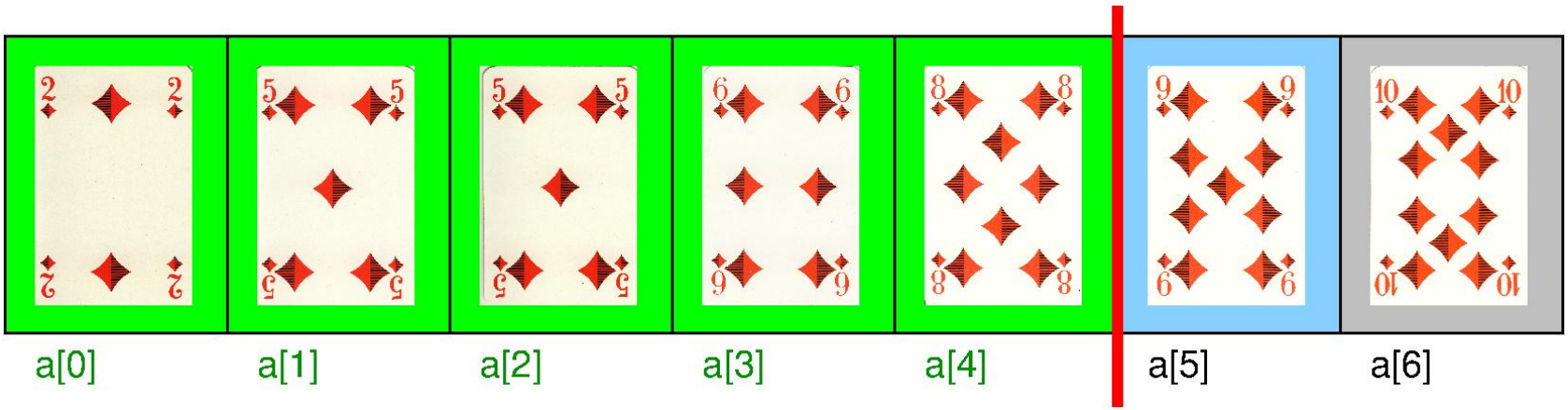
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Fertig sortierter Bereich enthält jetzt `a[0]...a[4]`

Selectionsort – Idee

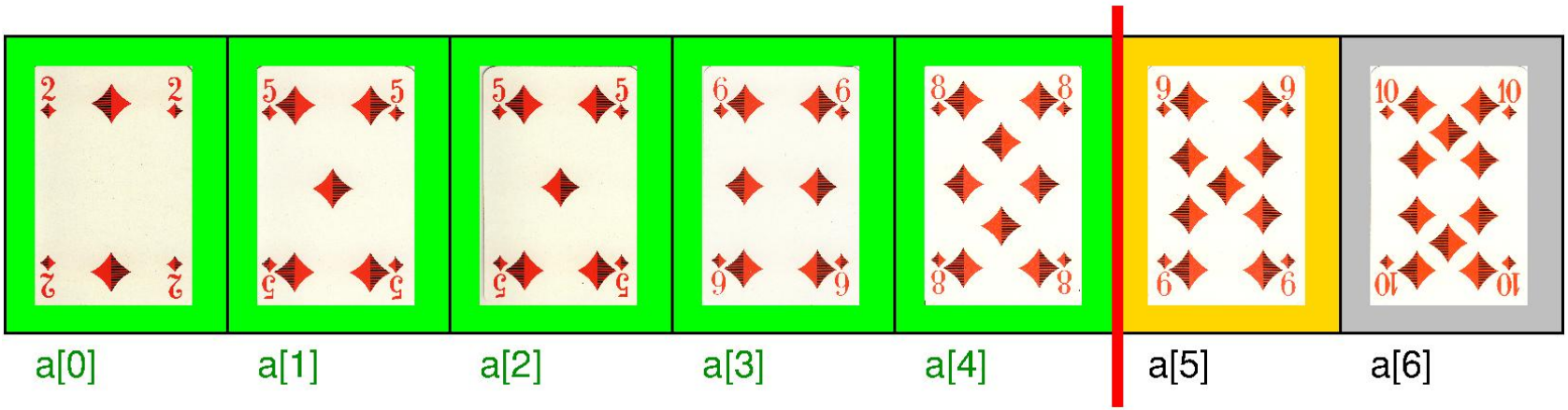
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[5]`

Selectionsort – Idee

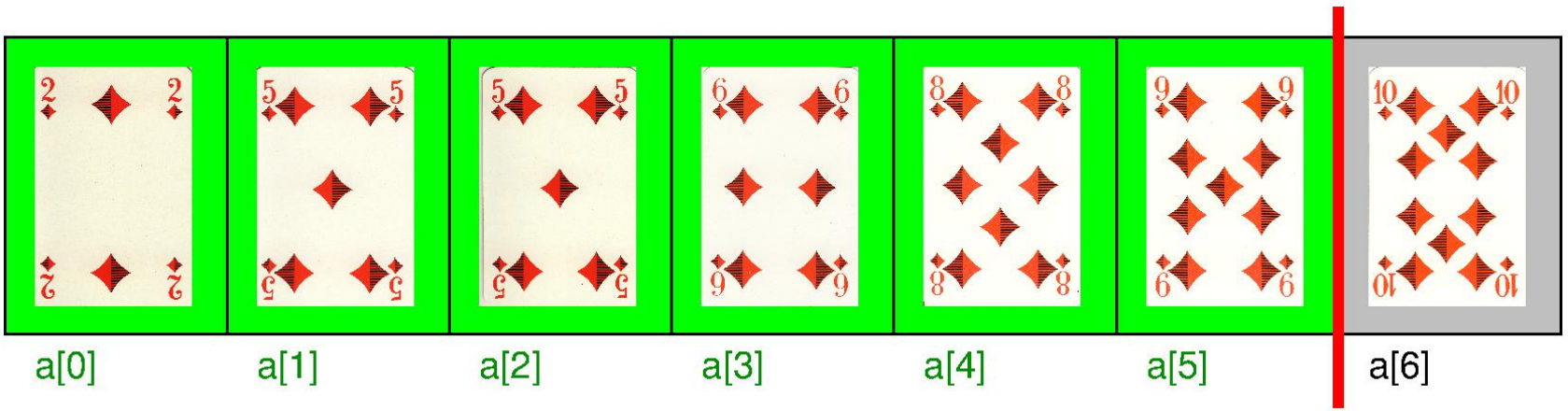
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[5]`

Selectionsort – Idee

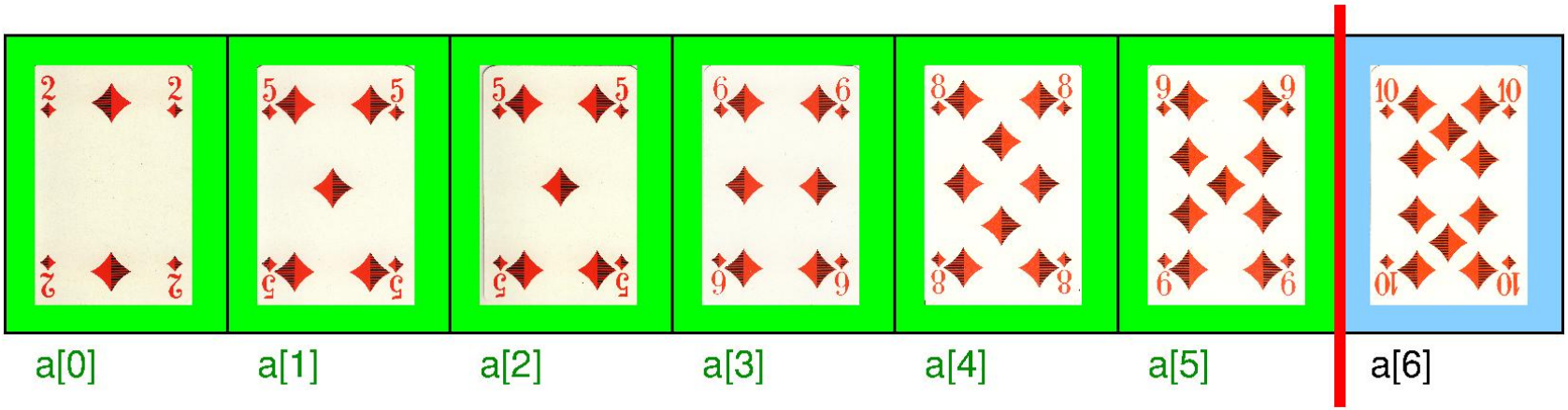
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Fertig sortierter Bereich enthält jetzt `a[0]...a[5]`

Selectionsort – Idee

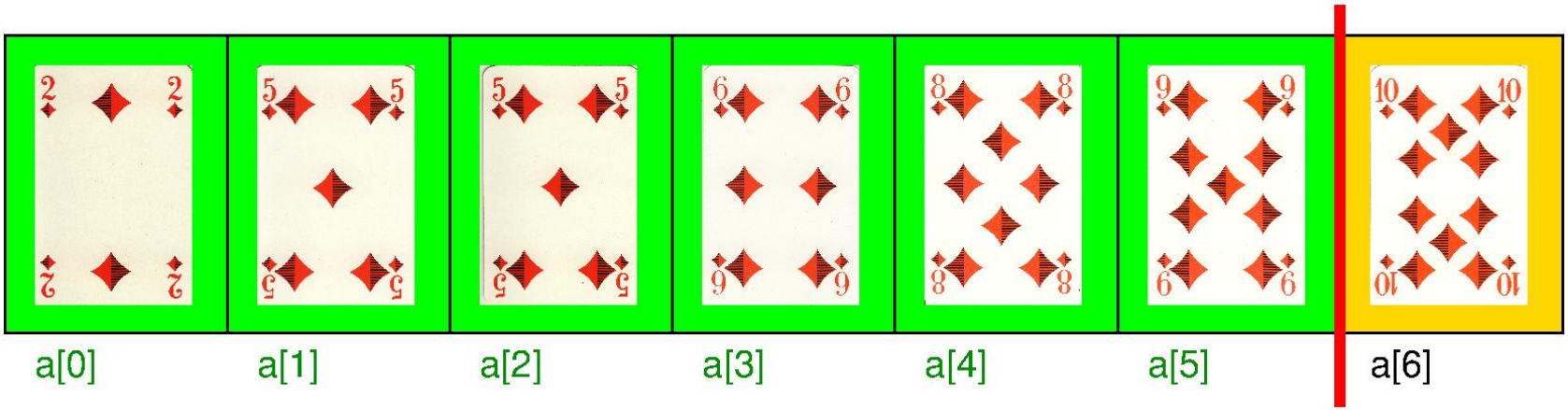
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[6]`

Selectionsort – Idee

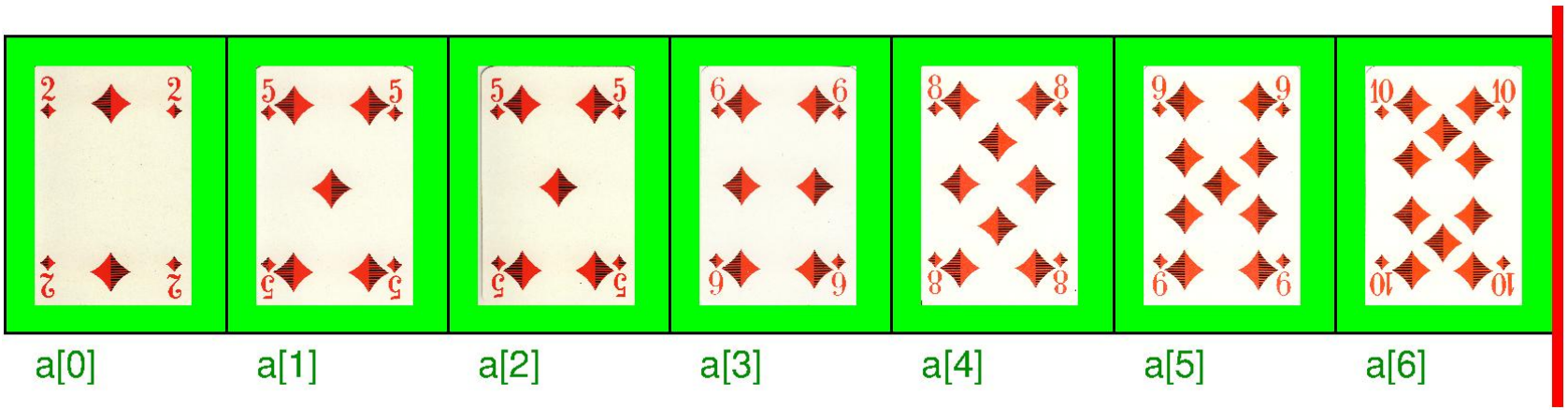
Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Suche im unsortierten Teil des Feldes (grau hinterlegt) das ***kleinste*** Element und vertausche es mit `a[6]`

Selectionsort – Idee

Aufsteigend sortieren durch fortlaufendes Auswählen des kleinsten Elements



Fertig sortiertes Feld `a[0]...a[6]`

Selectionsort ausprogrammiert

selectionsort.c – Funktion selectionsort zum aufsteigenden Sortieren eines Feldes

```c
#include <stdio.h>

#define N  7  //Anzahl zu sortierender Werte

void selectionsort(long a[])     //Datenfeld, dessen Elemente
                                 //aufsteigend sortiert werden
{
    int i, k, min;
    long t;

    for(i = 0; i < N-1; i++)
    {
        min = i;
        for(k = i+1; k < N; k++)  //Kleinstes Element im noch
        {                         //zu sortierenden Teilfeld finden
            if( a[k] < a[min])
                min = k;
        }
        if (i != min)
        {
          t = a[min];          //Vertausche kleinstes Element mit
          a[min] = a[i];       //Anfangselement im noch zu
          a[i] = t;            //sortierenden Teilfeld
        }
    }
    return;
}
```

Selectionsort ausprogrammiert

selectionsort.c – `main`-Funktion und Initialisierung des zu sortierenden Feldes

```c
int main(void)
{
  long datenfeld[N] = {9, 8, 2, 5, 10, 5, 6}; //zu sortierendes Datenfeld
  int i;

  selectionsort(datenfeld); //in-place sortieren
  for(i = 0; i < N; i++)
  {
    printf("%ld  ", datenfeld[i]); //Feldelemente ausgeben
  }
  printf("\n");
  return 0;
}
```

```
2  5  5  6  8  9  10
```

Steckbrief Selectionsort zum aufsteigenden Sortieren

Algorithmus . . . . . iterativ, vergleichsbasiert

Eigenschaften . . . . . in-place, stabil (gezeigte Variante)

Steckbrief Selectionsort zum aufsteigenden Sortieren

Algorithmus . . . . . iterativ, vergleichsbasiert

Eigenschaften . . . . . in-place, stabil (gezeigte Variante)

Worst Case . . . . . Folge anfänglich *absteigend* geordnet

Best Case . . . . . Folge anfänglich schon *aufsteigend* sortiert

---

Steckbrief Selectionsort zum aufsteigenden Sortieren

Algorithmus . . . . . iterativ, vergleichsbasiert

Eigenschaften . . . . . in-place, stabil (gezeigte Variante)

Worst Case . . . . . Folge anfänglich *absteigend* geordnet

Best Case . . . . . Folge anfänglich schon *aufsteigend* sortiert

---

Problemgröße n . . . . . Anzahl zu sortierender Feldelemente

Anzahl Vergleiche . . . . . $n \cdot (n-1)/2$

Anzahl Vertauschungen . . . . . $< n$

Steckbrief Selectionsort zum aufsteigenden Sortieren

Algorithmus . . . . . iterativ, vergleichsbasiert

Eigenschaften . . . . . in-place, stabil (gezeigte Variante)

Worst Case . . . . . Folge anfänglich *absteigend* geordnet

Best Case . . . . . Folge anfänglich schon *aufsteigend* sortiert

---

Problemgröße n . . . . . Anzahl zu sortierender Feldelemente

Anzahl Vergleiche . . . . . $n \cdot (n-1)/2$

Anzahl Vertauschungen . . . . . $< n$

Zeitkomplexität im Worst Case . . . . . $O(n^2)$

Selectionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `min`)
- wenig Umspeichervorgänge (Vertauschungen) im Feld
- jedes Feldelement höchstens einmal bewegt
- sortierbegleitend entsteht fertig sortierter Bereich, der schon ausgelesen werden kann, während an anderen Stellen im Feld noch sortiert wird
- Laufzeitverhalten im Best Case und Worst Case unterscheidet sich nur geringfügig
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld geeignet

Selectionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und **`min`**)
- wenig Umspeichervorgänge (Vertauschungen) im Feld
- jedes Feldelement höchstens einmal bewegt
- sortierbegleitend entsteht fertig sortierter Bereich, der schon ausgelesen werden kann, während an anderen Stellen im Feld noch sortiert wird
- Laufzeitverhalten im Best Case und Worst Case unterscheidet sich nur geringfügig
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld geeignet

Selectionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `min`)
- wenig Umspeichervorgänge (Vertauschungen) im Feld
- jedes Feldelement höchstens einmal bewegt
- sortierbegleitend entsteht fertig sortierter Bereich, der schon ausgelesen werden kann, während an anderen Stellen im Feld noch sortiert wird
- Laufzeitverhalten im Best Case und Worst Case unterscheidet sich nur geringfügig
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld geeignet

Selectionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `min`)
- wenig Umspeichervorgänge (Vertauschungen) im Feld
- jedes Feldelement höchstens einmal bewegt
- sortierbegleitend entsteht fertig sortierter Bereich, der schon ausgelesen werden kann, während an anderen Stellen im Feld noch sortiert wird
- Laufzeitverhalten im Best Case und Worst Case unterscheidet sich nur geringfügig
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld geeignet

Selectionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `min`)
- wenig Umspeichervorgänge (Vertauschungen) im Feld
- jedes Feldelement höchstens einmal bewegt
- sortierbegleitend entsteht fertig sortierter Bereich, der schon ausgelesen werden kann, während an anderen Stellen im Feld noch sortiert wird
- Laufzeitverhalten im Best Case und Worst Case unterscheidet sich nur geringfügig
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld geeignet

Selectionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `min`)
- wenig Umspeichervorgänge (Vertauschungen) im Feld
- jedes Feldelement höchstens einmal bewegt
- sortierbegleitend entsteht fertig sortierter Bereich, der schon ausgelesen werden kann, während an anderen Stellen im Feld noch sortiert wird
- Laufzeitverhalten im Best Case und Worst Case unterscheidet sich nur geringfügig
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld geeignet

Selectionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `min`)
- wenig Umspeichervorgänge (Vertauschungen) im Feld
- jedes Feldelement höchstens einmal bewegt
- sortierbegleitend entsteht fertig sortierter Bereich, der schon ausgelesen werden kann, während an anderen Stellen im Feld noch sortiert wird
- Laufzeitverhalten im Best Case und Worst Case unterscheidet sich nur geringfügig
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld geeignet

Selectionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `min`)
- wenig Umspeichervorgänge (Vertauschungen) im Feld
- jedes Feldelement höchstens einmal bewegt
- sortierbegleitend entsteht fertig sortierter Bereich, der schon ausgelesen werden kann, während an anderen Stellen im Feld noch sortiert wird
- Laufzeitverhalten im Best Case und Worst Case unterscheidet sich nur geringfügig
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld geeignet

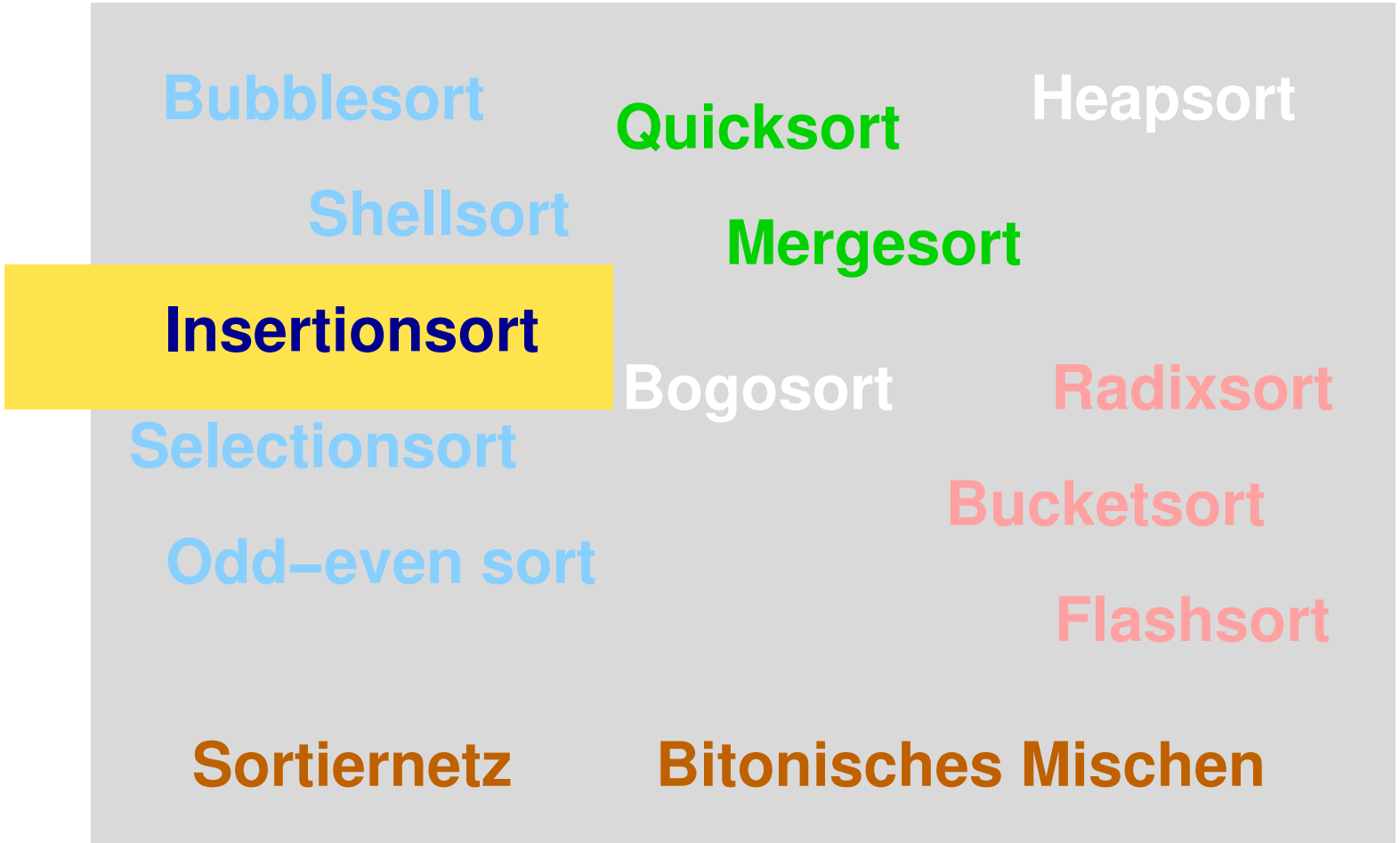
Selectionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und **`min`**)
- wenig Umspeichervorgänge (Vertauschungen) im Feld
- jedes Feldelement höchstens einmal bewegt
- sortierbegleitend entsteht fertig sortierter Bereich, der schon ausgelesen werden kann, während an anderen Stellen im Feld noch sortiert wird
- Laufzeitverhalten im Best Case und Worst Case unterscheidet sich nur geringfügig
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld geeignet

Nachteile: viele Vergleiche, daher langsam auch im Best Case

Insertionsort

Sortieren durch Einfügen nach Vorbild des Kartenspielers

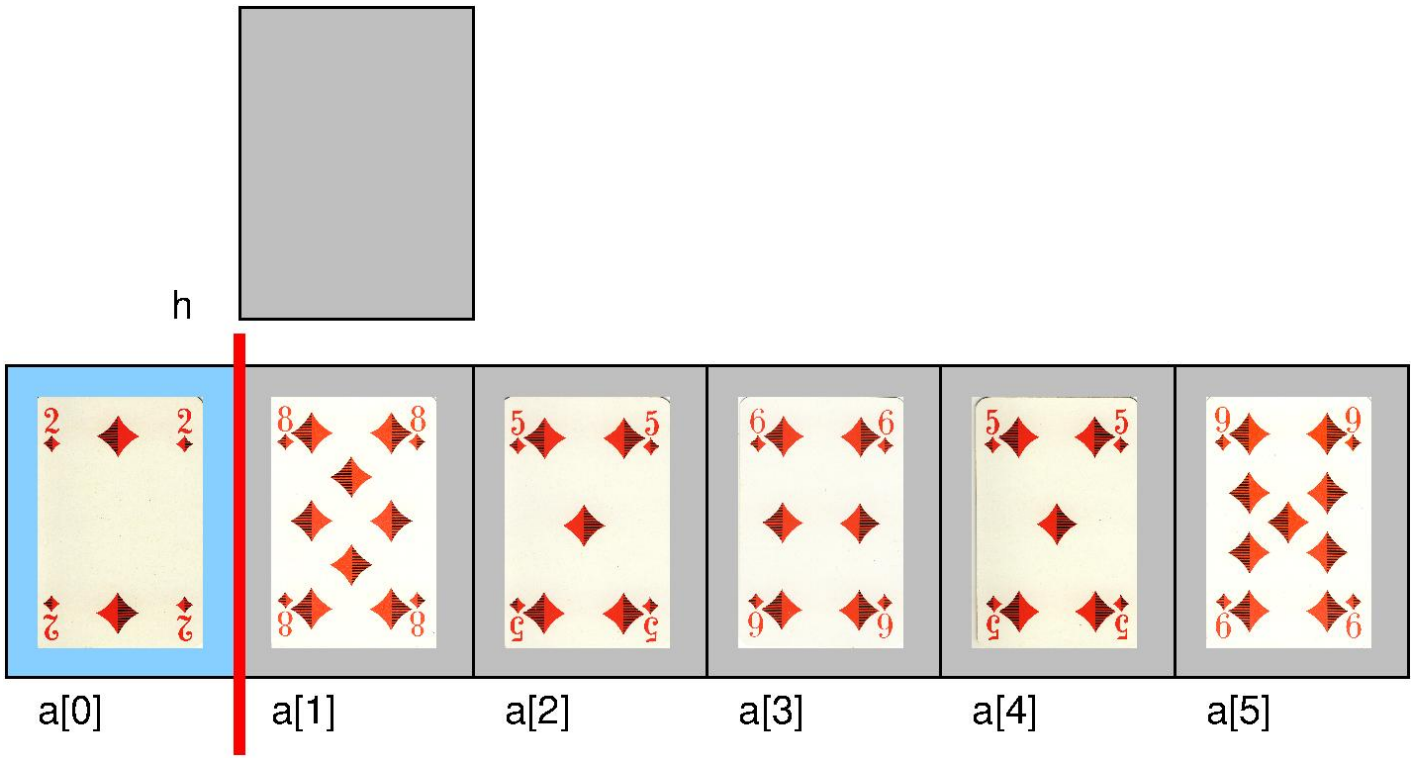


Insertionsort – Idee

Kartenspieler *fügt* Karte für Karte an der richtigen Stelle *ein*

Insertionsort – Idee

Sortieren durch Einfügen nach Vorbild des Kartenspielers

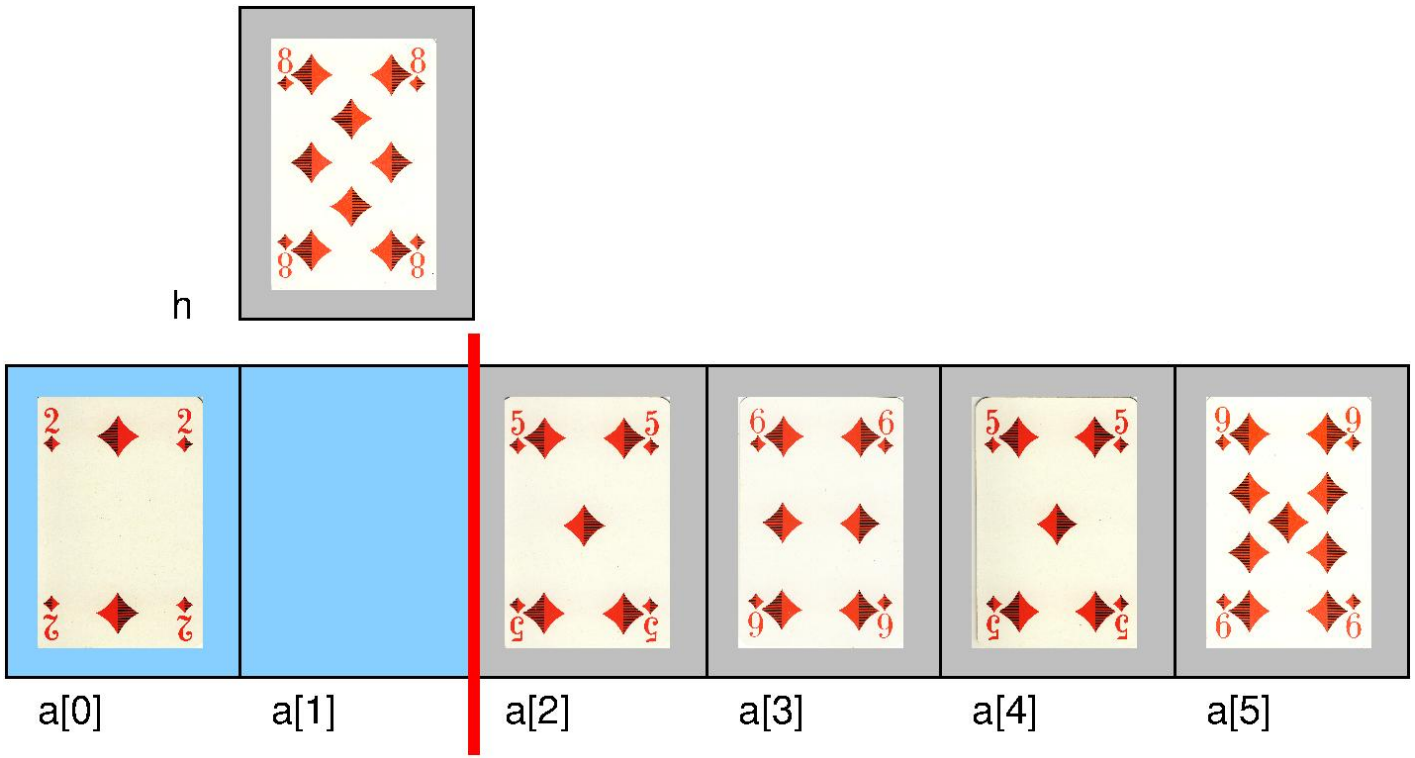


aufsteigend sortiertes Teilfeld, in das eingefügt wird: blau.
Teilfeld, dessen Elemente noch eingefügt werden müssen: grau.

Entnimm Element `a[1]`

Insertionsort – Idee

Sortieren durch Einfügen nach Vorbild des Kartenspielers

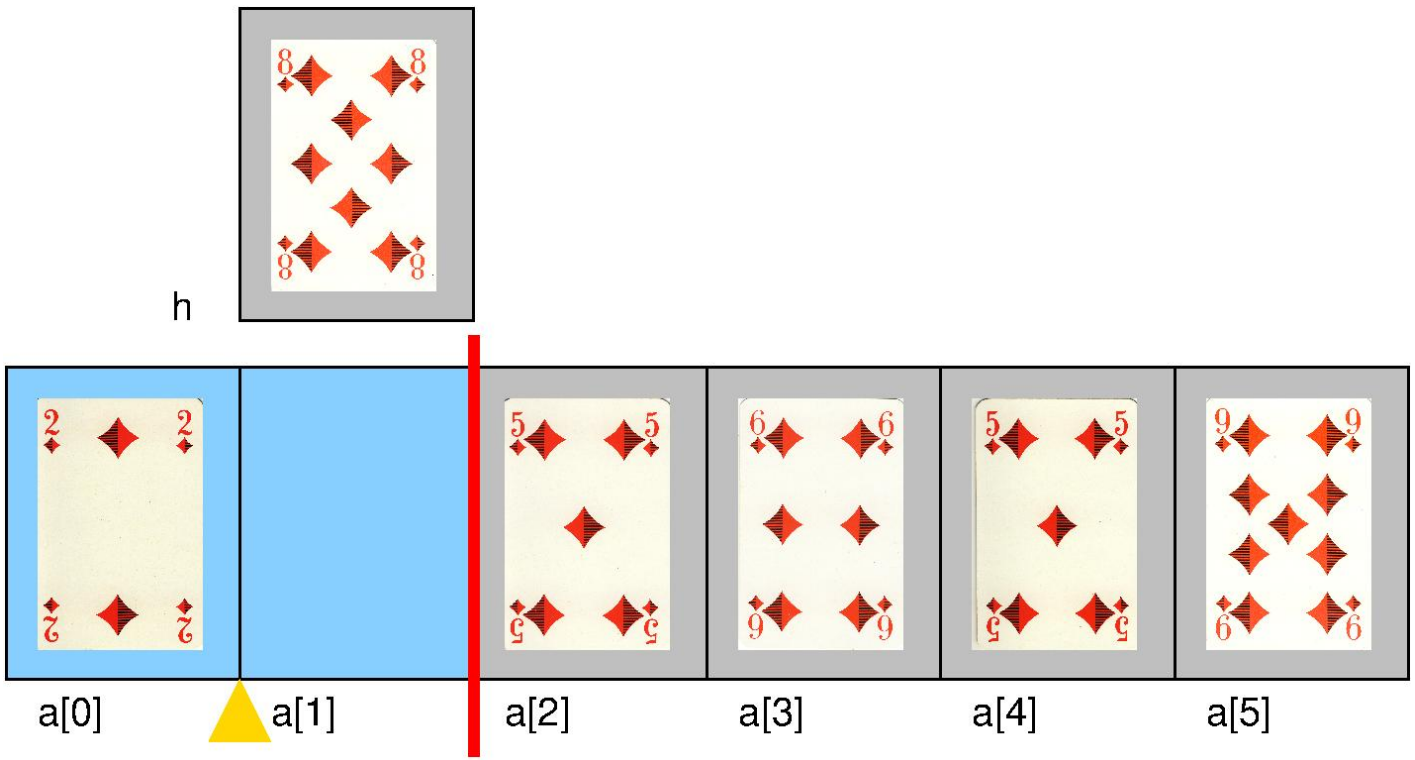


aufsteigend sortiertes Teilfeld, in das eingefügt wird: blau.
Teilfeld, dessen Elemente noch eingefügt werden müssen: grau.

Entnimm Element `a[1]`

Insertionsort – Idee

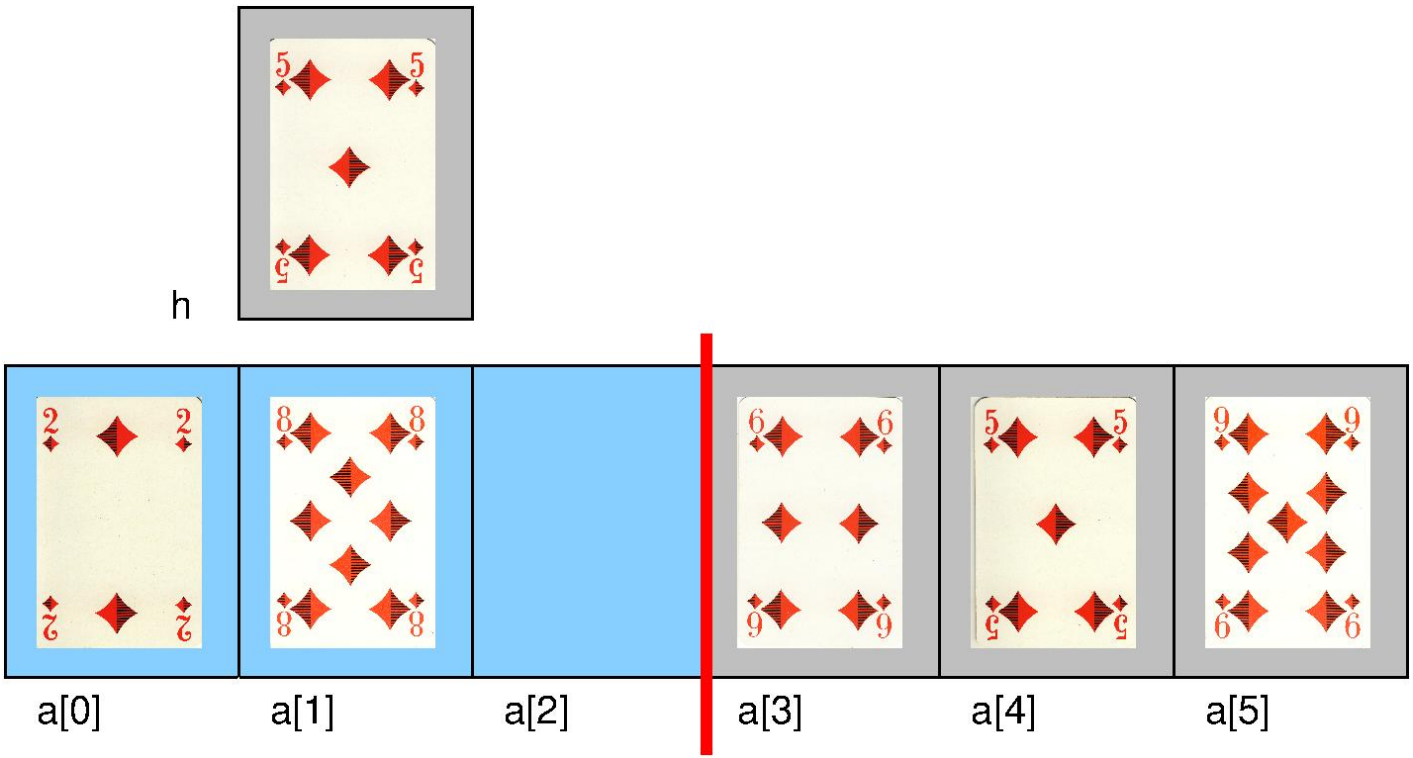
Sortieren durch Einfügen nach Vorbild des Kartenspielers



Durchlaufe sortiertes Teilfeld von rechts nach links bis zur Einfügeposition. Verschiebe dabei Elemente um eine Position nach rechts, bis Lücke an der passenden Stelle.

Insertionsort – Idee

Sortieren durch Einfügen nach Vorbild des Kartenspielers

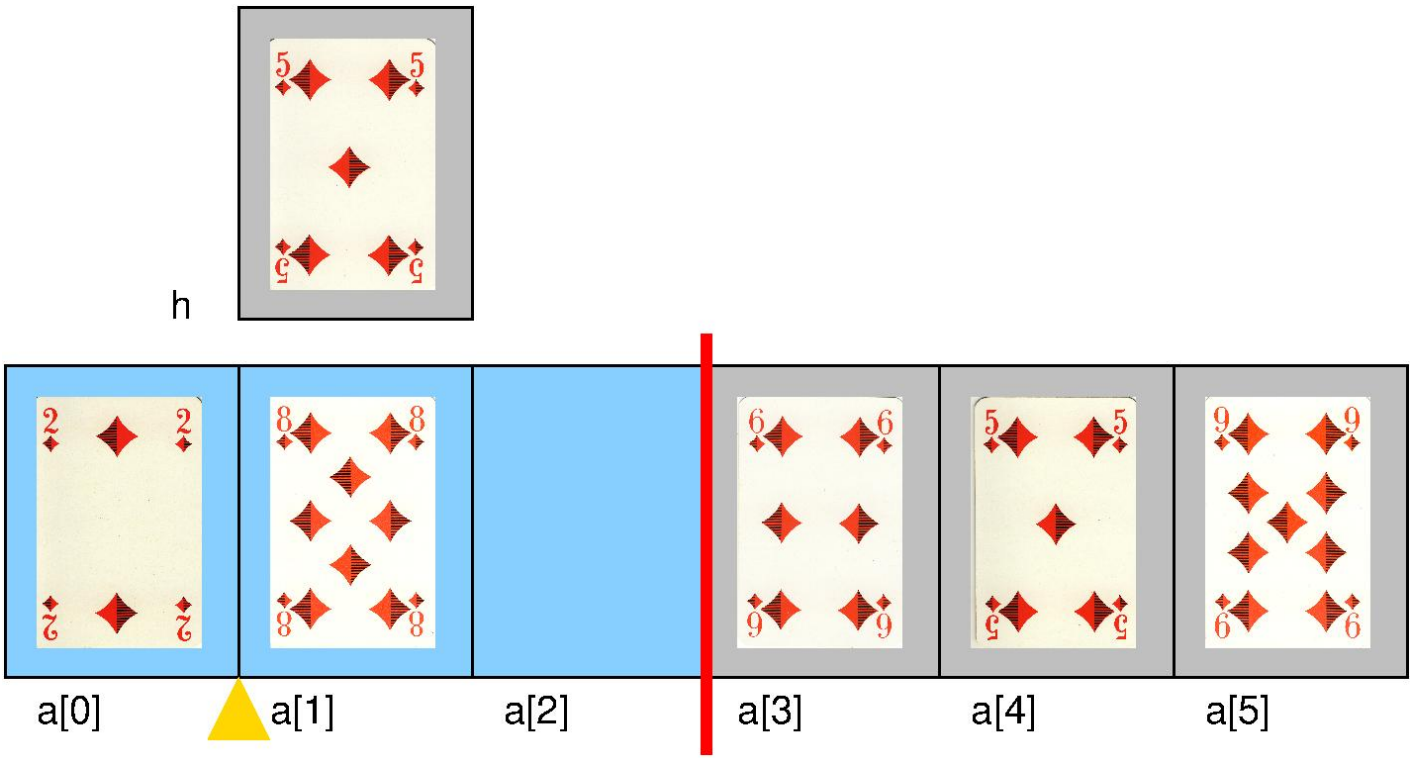


aufsteigend sortiertes Teilfeld, in das eingefügt wird: blau.
Teilfeld, dessen Elemente noch eingefügt werden müssen: grau.

Entnimm Element `a[2]`

Insertionsort – Idee

Sortieren durch Einfügen nach Vorbild des Kartenspielers

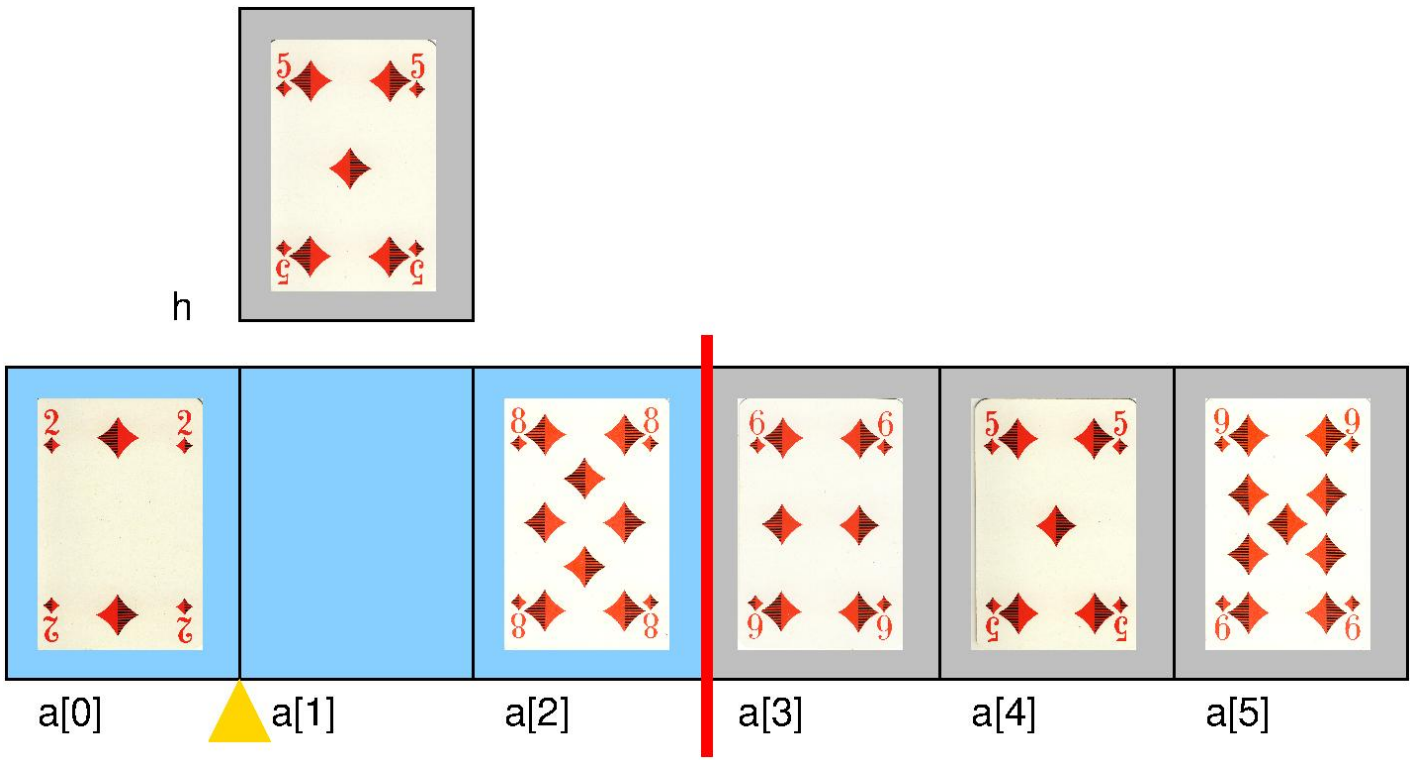


aufsteigend sortiertes Teilfeld, in das eingefügt wird: blau.
Teilfeld, dessen Elemente noch eingefügt werden müssen: grau.

Entnimm Element `a[2]`

Insertionsort – Idee

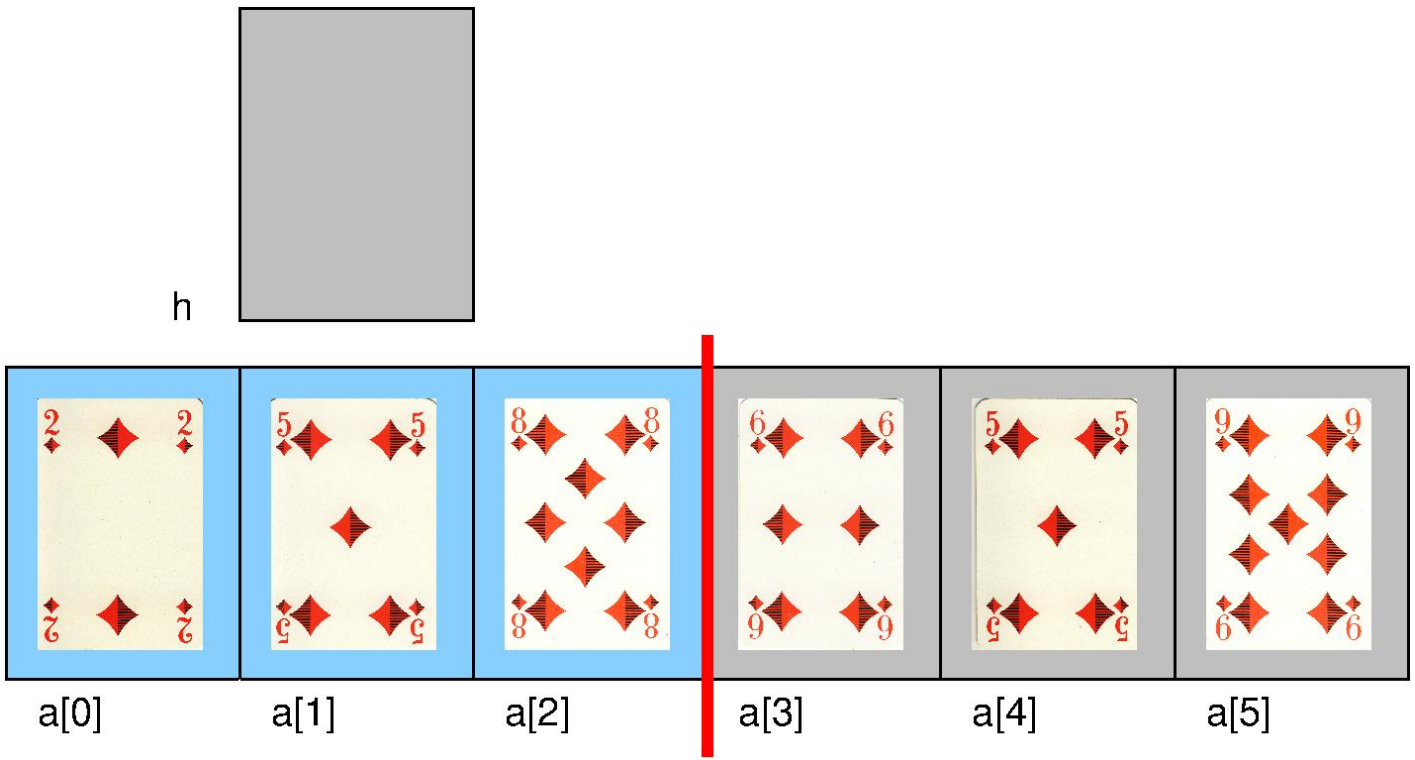
Sortieren durch Einfügen nach Vorbild des Kartenspielers



Durchlaufe sortiertes Teilfeld von rechts nach links bis zur Einfügeposition. Verschiebe dabei Elemente um eine Position nach rechts, bis Lücke an der passenden Stelle.

Insertionsort – Idee

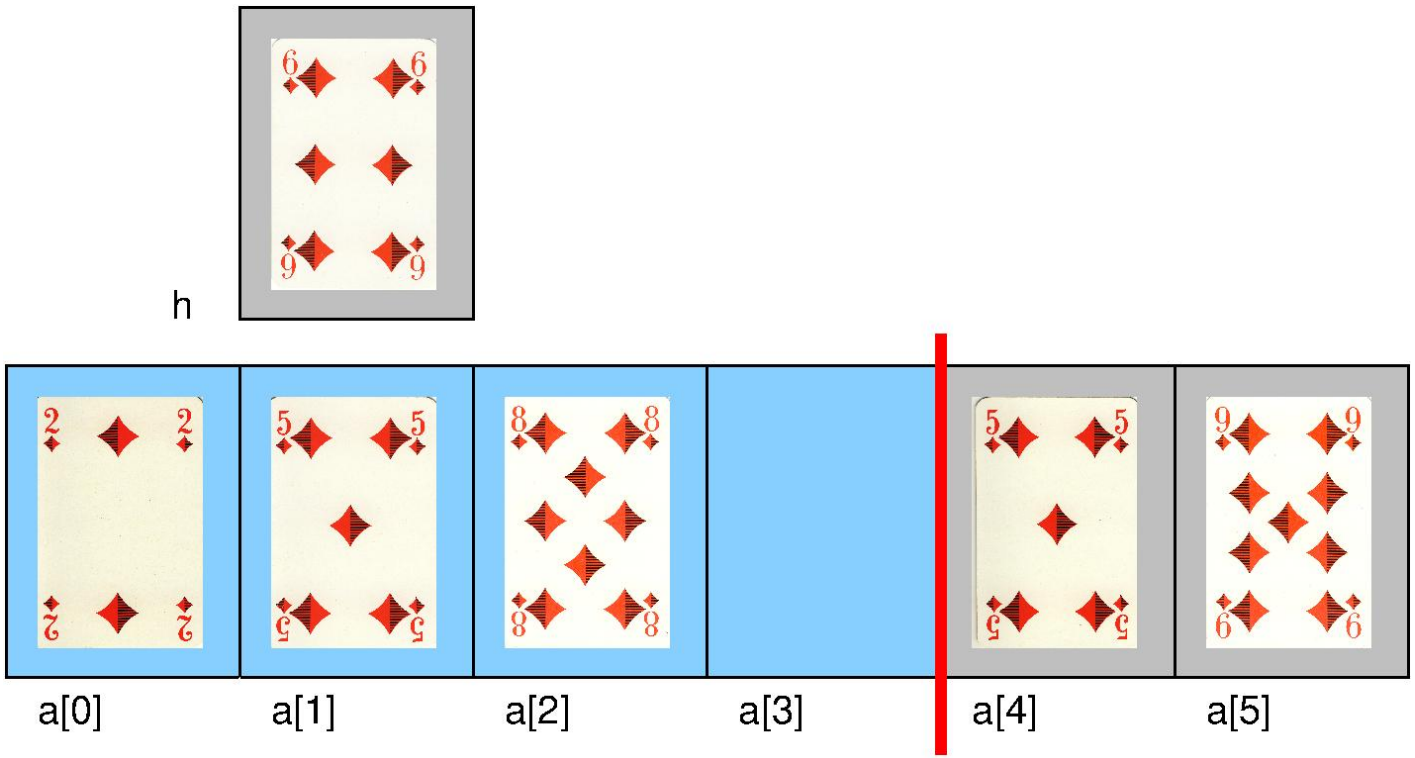
Sortieren durch Einfügen nach Vorbild des Kartenspielers



Durchlaufe sortiertes Teilfeld von rechts nach links bis zur Einfügeposition. Verschiebe dabei Elemente um eine Position nach rechts, bis Lücke an der passenden Stelle.

Insertionsort – Idee

Sortieren durch Einfügen nach Vorbild des Kartenspielers

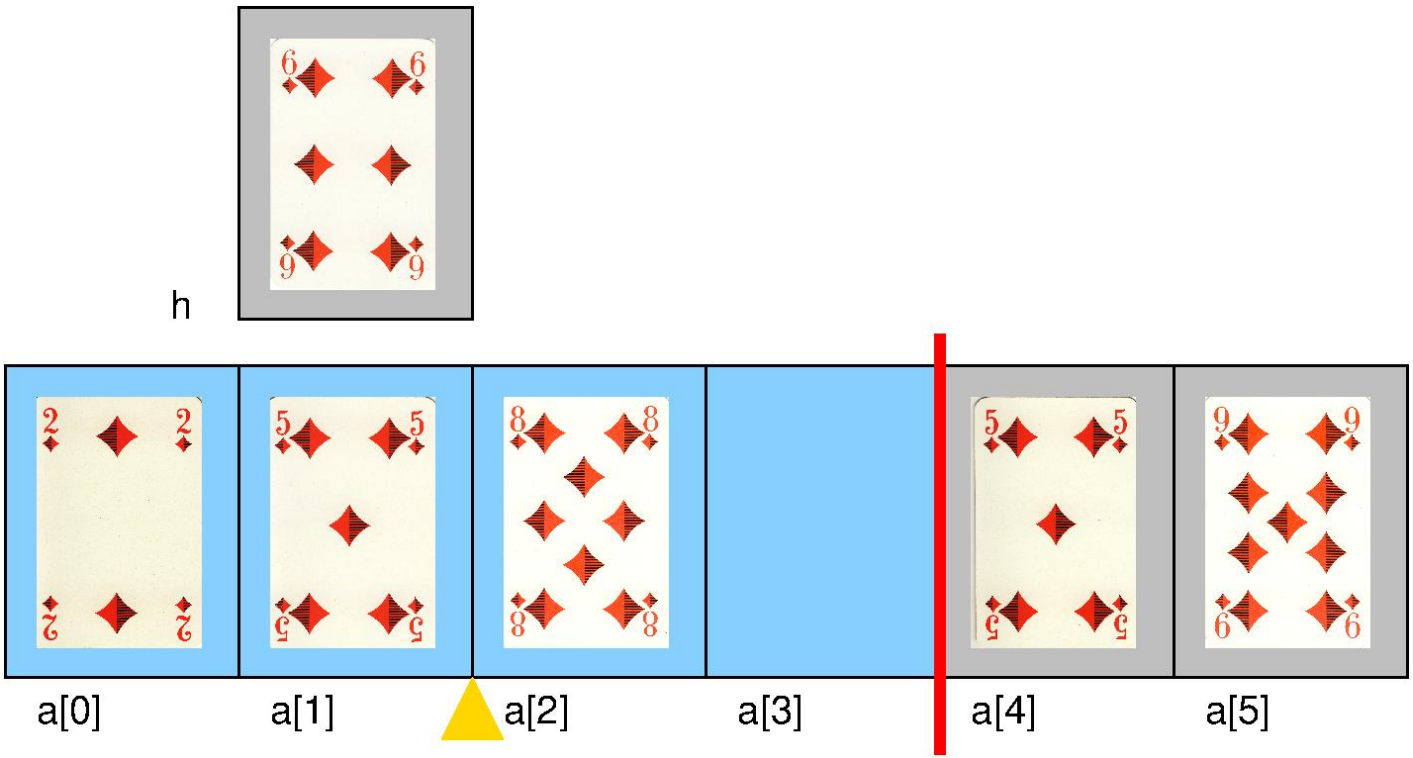


aufsteigend sortiertes Teilfeld, in das eingefügt wird: blau.
Teilfeld, dessen Elemente noch eingefügt werden müssen: grau.

Entnimm Element `a[3]`

Insertionsort – Idee

Sortieren durch Einfügen nach Vorbild des Kartenspielers

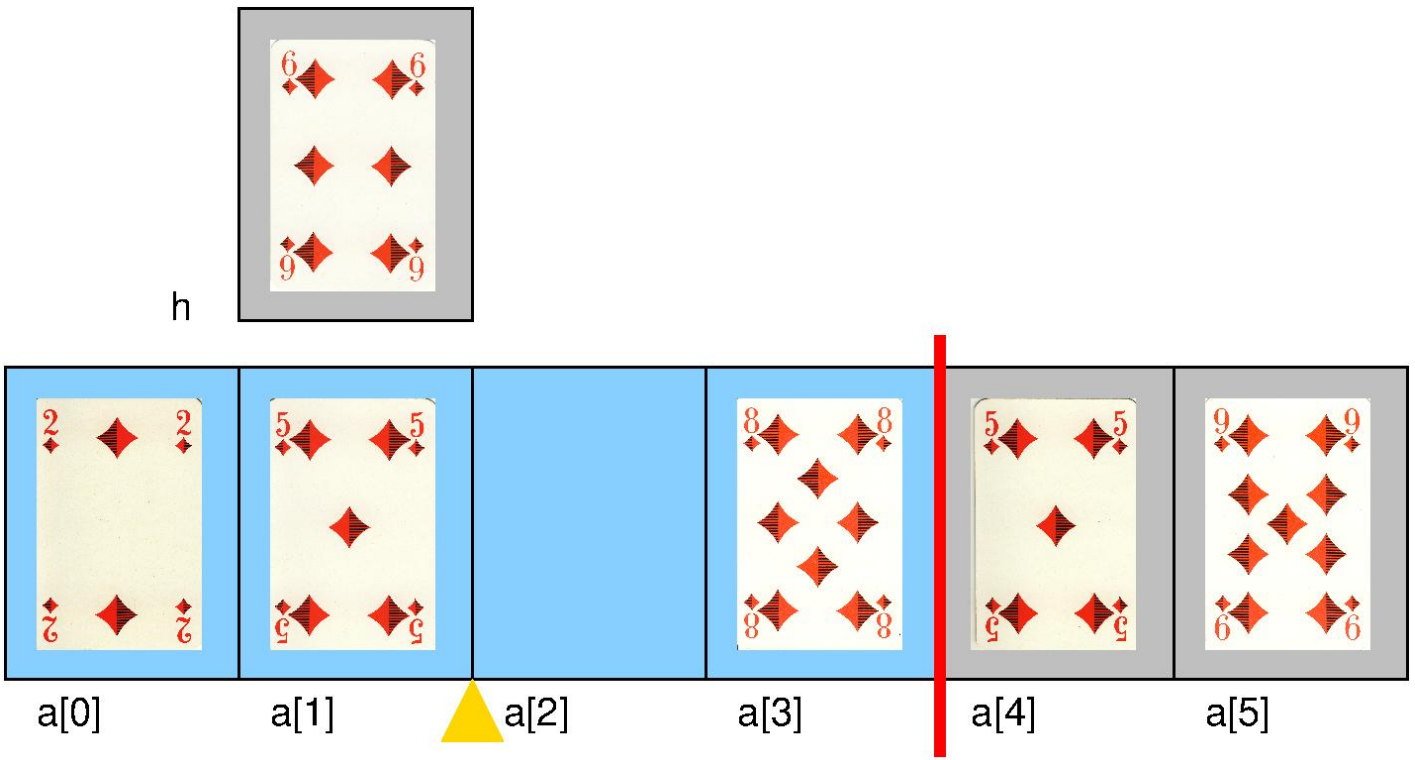


aufsteigend sortiertes Teilfeld, in das eingefügt wird: blau.
Teilfeld, dessen Elemente noch eingefügt werden müssen: grau.

Entnimm Element `a[3]`

Insertionsort – Idee

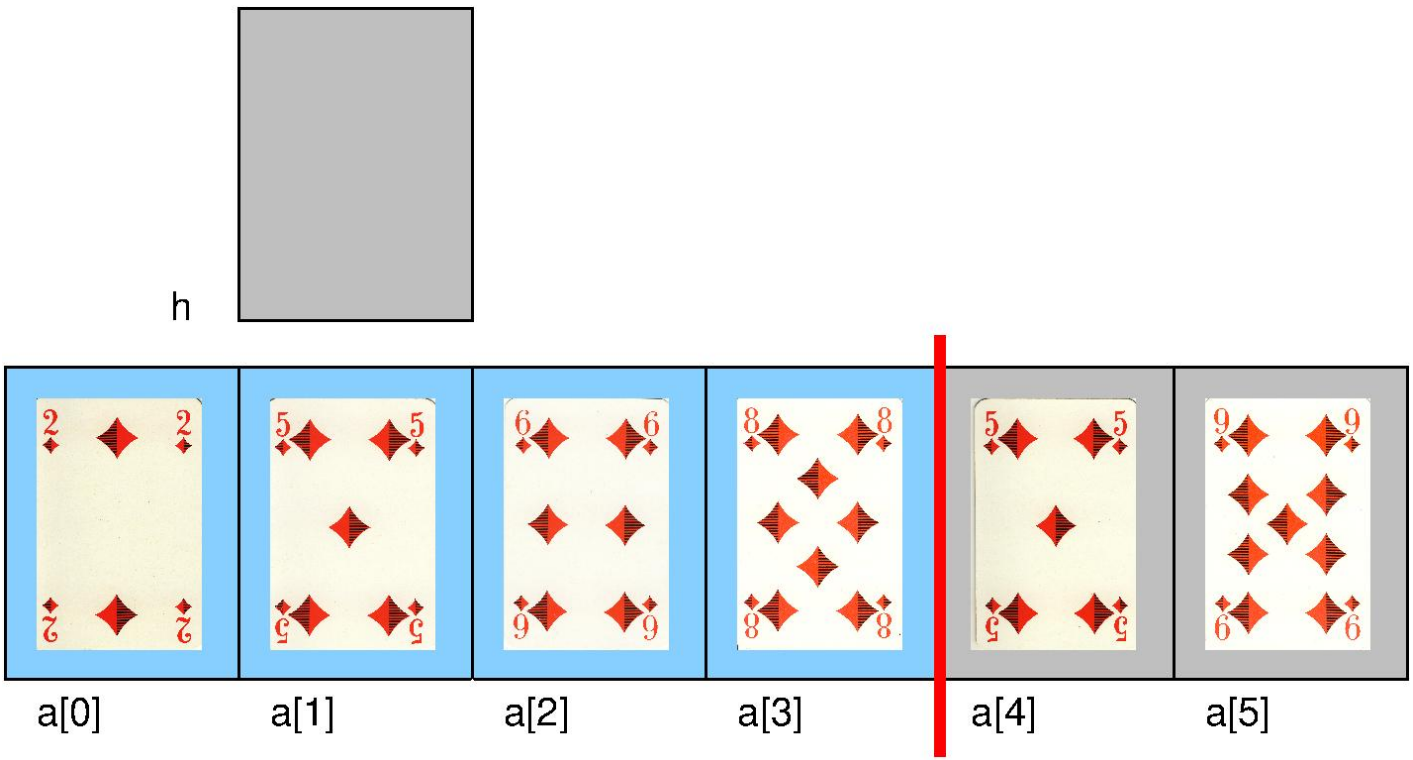
Sortieren durch Einfügen nach Vorbild des Kartenspielers



Durchlaufe sortiertes Teilfeld von rechts nach links bis zur Einfügeposition. Verschiebe dabei Elemente um eine Position nach rechts, bis Lücke an der passenden Stelle.

Insertionsort – Idee

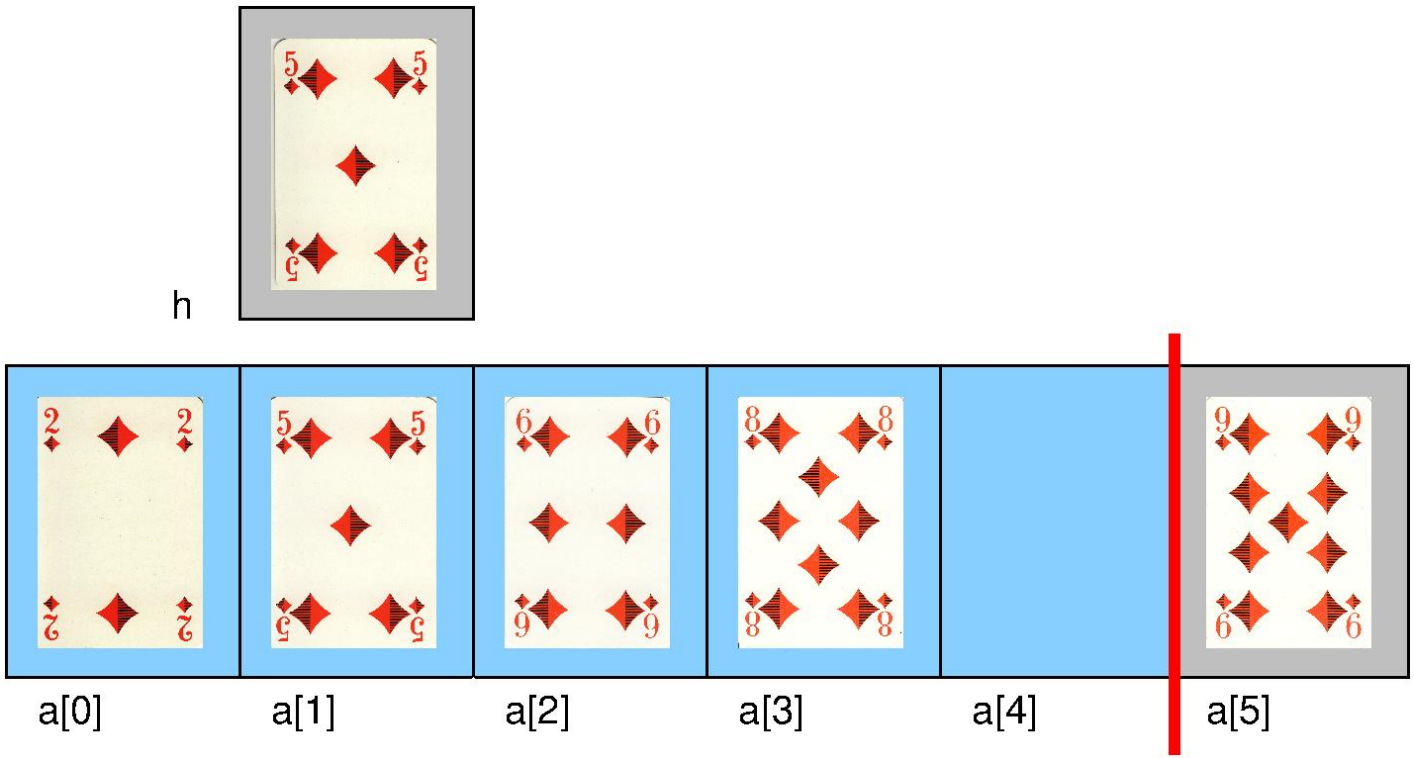
Sortieren durch Einfügen nach Vorbild des Kartenspielers



Durchlaufe sortiertes Teilfeld von rechts nach links bis zur Einfügeposition. Verschiebe dabei Elemente um eine Position nach rechts, bis Lücke an der passenden Stelle.

Insertionsort – Idee

Sortieren durch Einfügen nach Vorbild des Kartenspielers

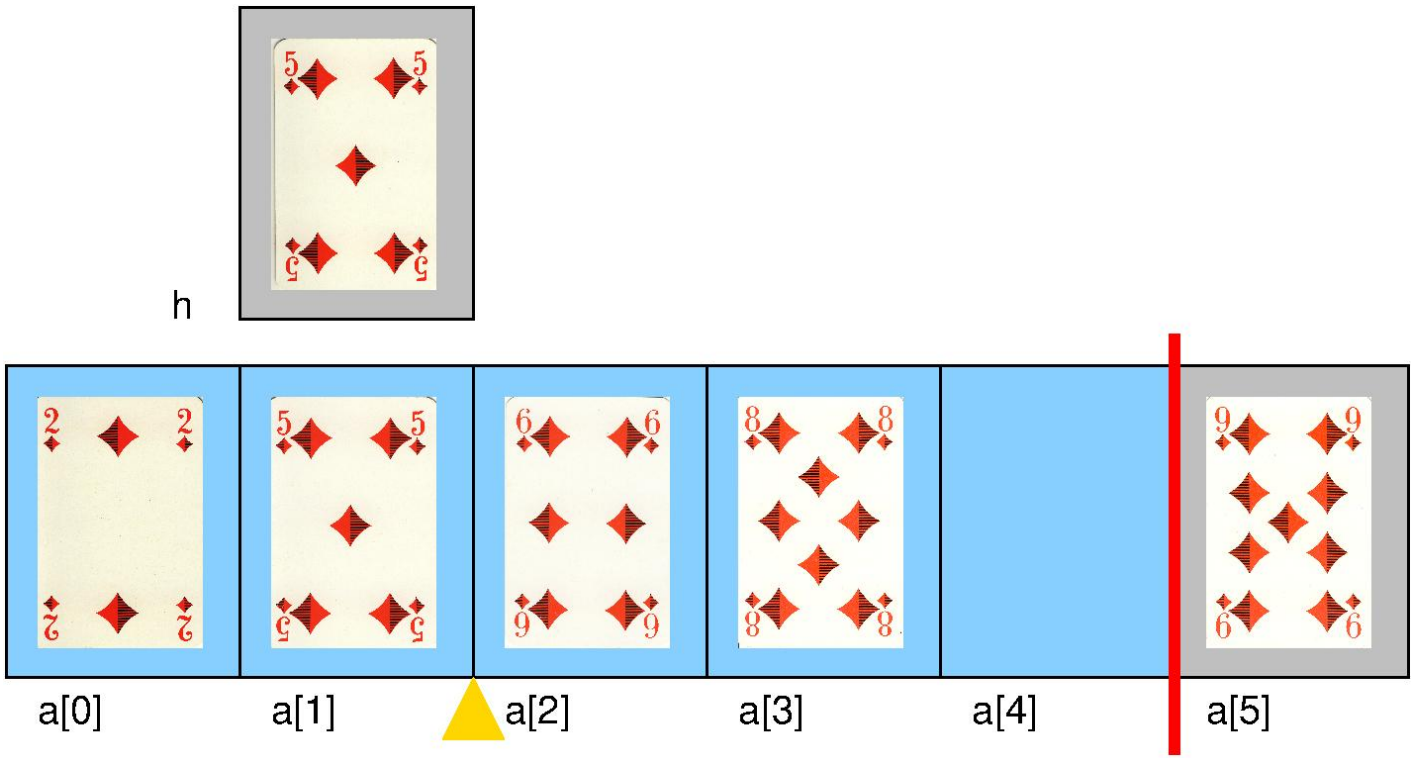


aufsteigend sortiertes Teilfeld, in das eingefügt wird: blau.
Teilfeld, dessen Elemente noch eingefügt werden müssen: grau.

Entnimm Element `a[4]`

Insertionsort – Idee

Sortieren durch Einfügen nach Vorbild des Kartenspielers

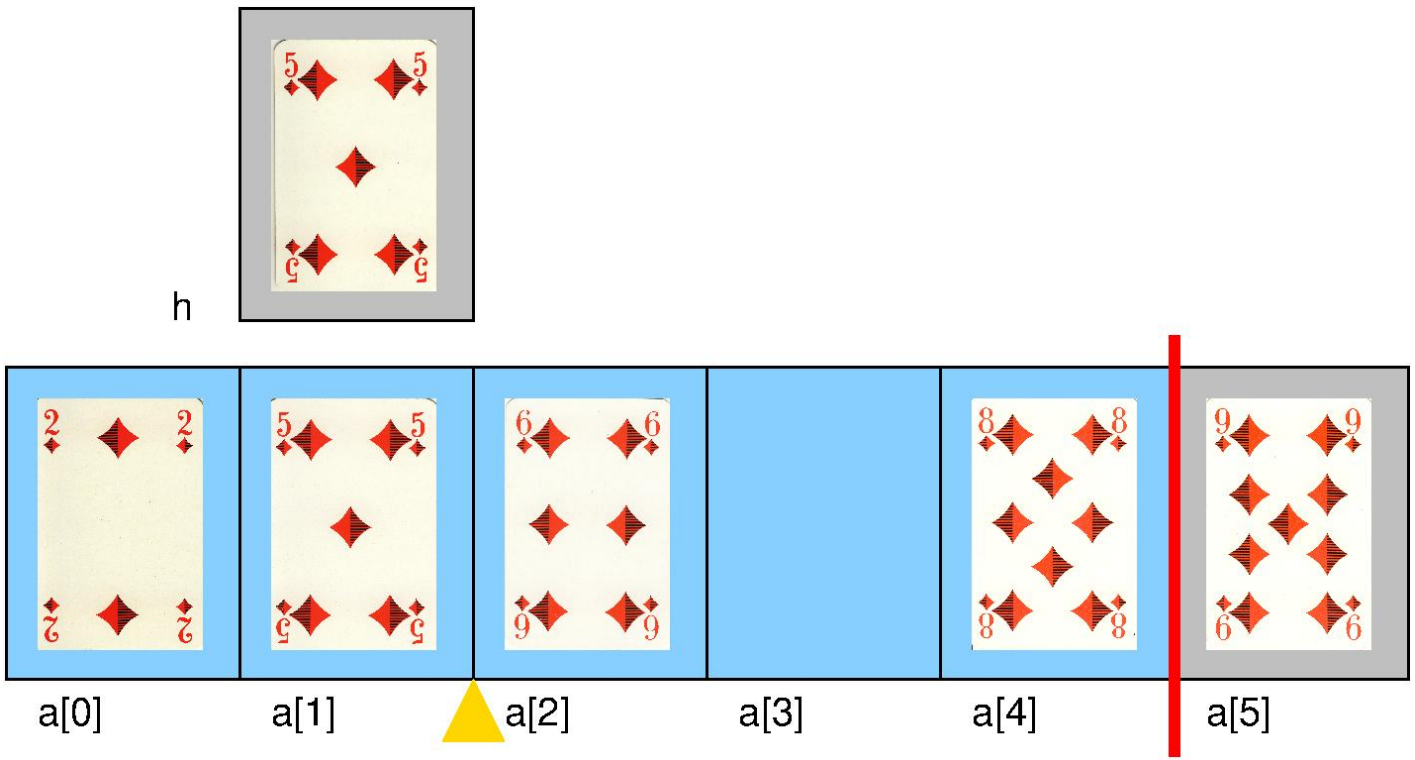


aufsteigend sortiertes Teilfeld, in das eingefügt wird: blau.
Teilfeld, dessen Elemente noch eingefügt werden müssen: grau.

Entnimm Element `a[4]`

Insertionsort – Idee

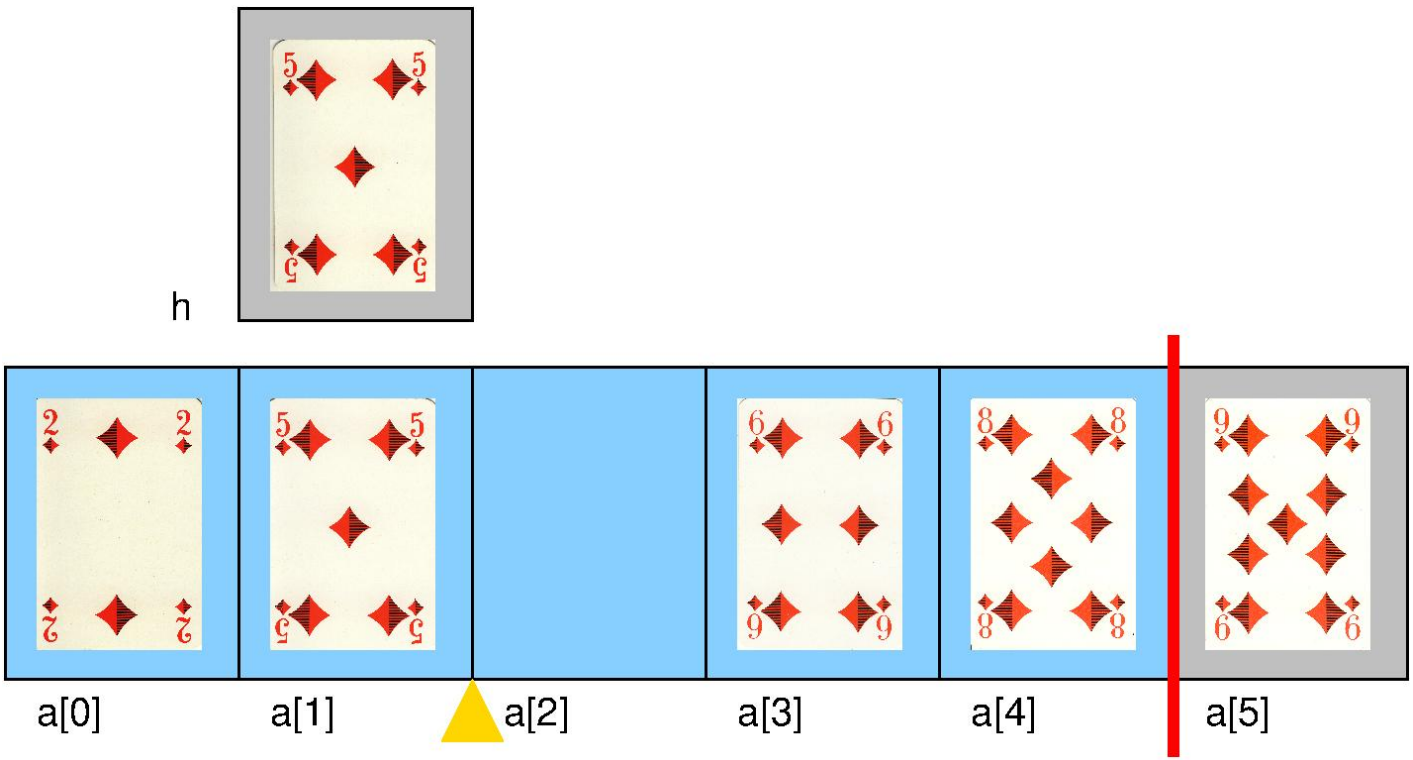
Sortieren durch Einfügen nach Vorbild des Kartenspielers



Durchlaufe sortiertes Teilfeld von rechts nach links bis zur Einfügeposition. Verschiebe dabei Elemente um eine Position nach rechts, bis Lücke an der passenden Stelle.

Insertionsort – Idee

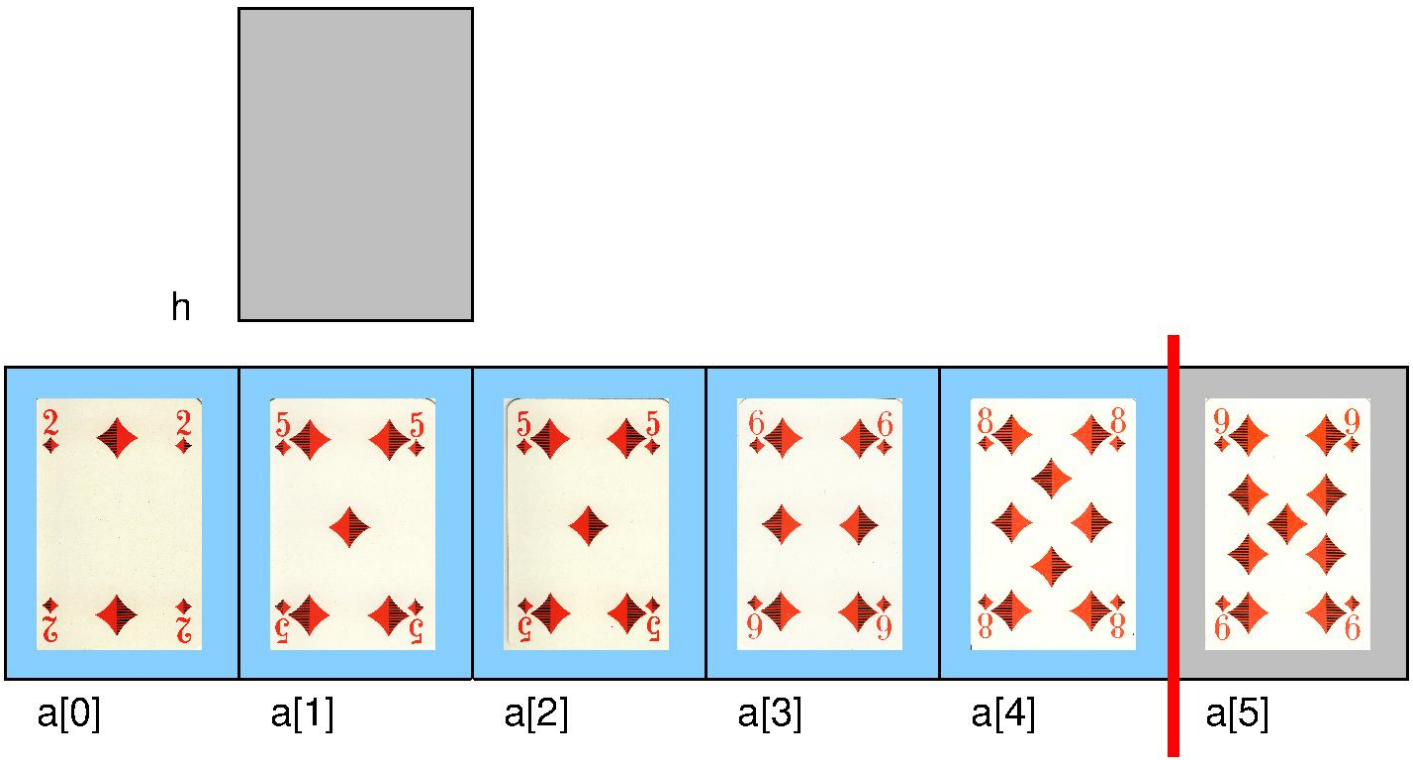
Sortieren durch Einfügen nach Vorbild des Kartenspielers



Durchlaufe sortiertes Teilfeld von rechts nach links bis zur Einfügeposition. Verschiebe dabei Elemente um eine Position nach rechts, bis Lücke an der passenden Stelle.

Insertionsort – Idee

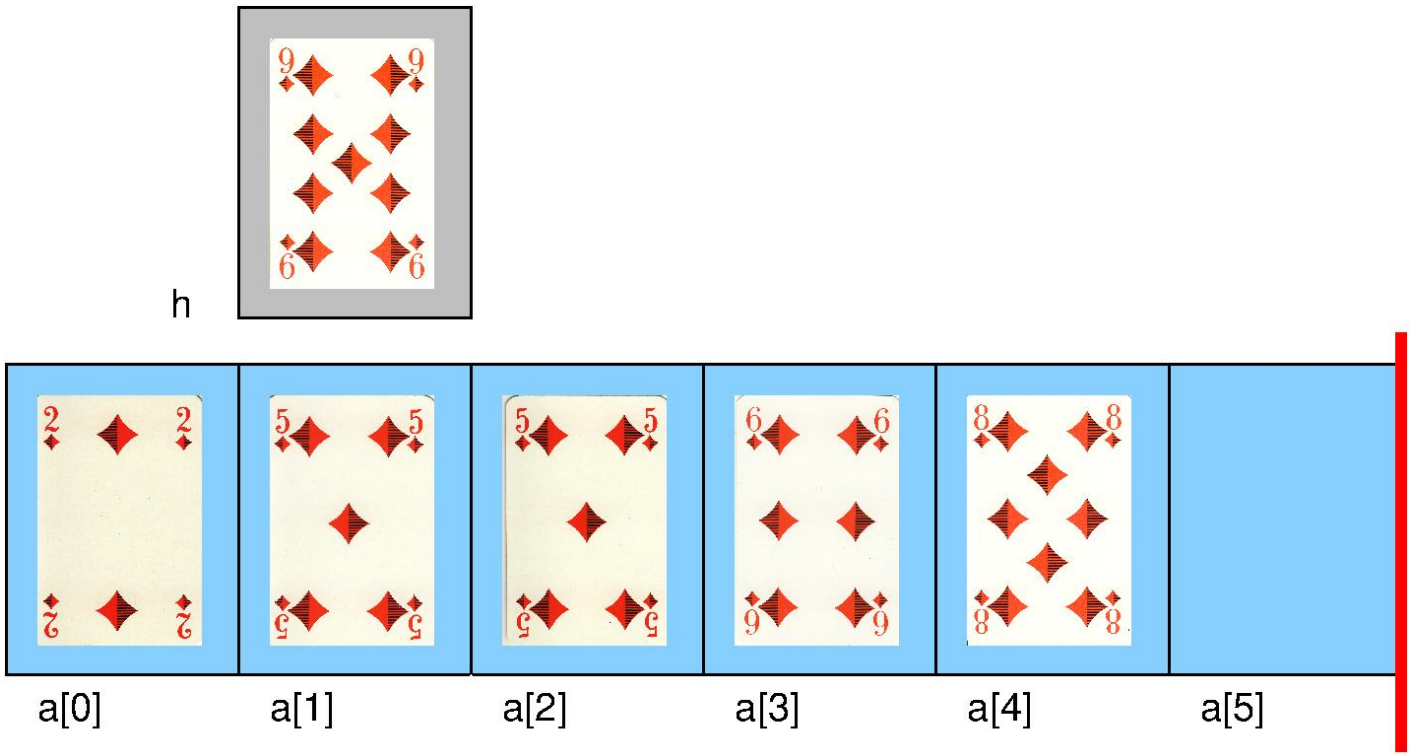
Sortieren durch Einfügen nach Vorbild des Kartenspielers



Durchlaufe sortiertes Teilfeld von rechts nach links bis zur Einfügeposition. Verschiebe dabei Elemente um eine Position nach rechts, bis Lücke an der passenden Stelle.

Insertionsort – Idee

Sortieren durch Einfügen nach Vorbild des Kartenspielers

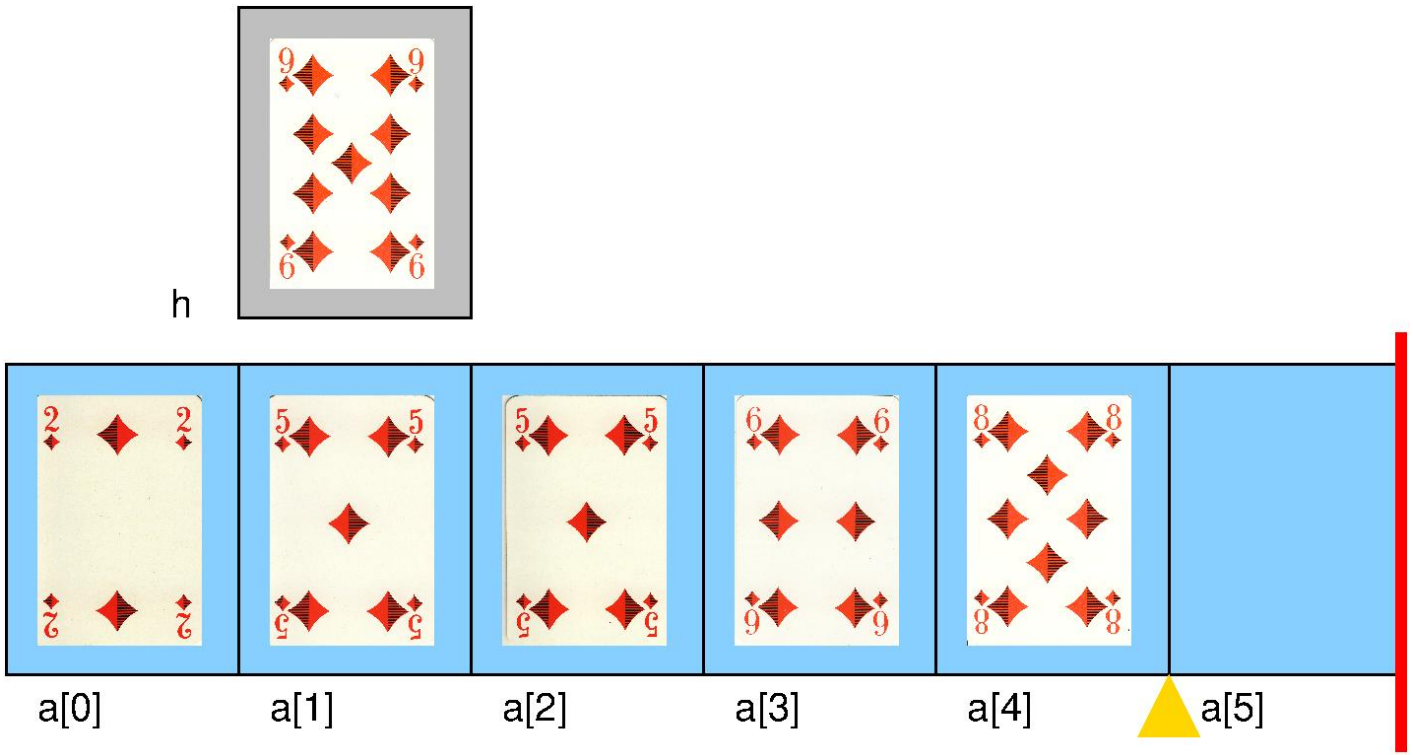


aufsteigend sortiertes Teilfeld, in das eingefügt wird: blau.
Teilfeld, dessen Elemente noch eingefügt werden müssen: grau.

Entnimm Element `a[5]`

Insertionsort – Idee

Sortieren durch Einfügen nach Vorbild des Kartenspielers

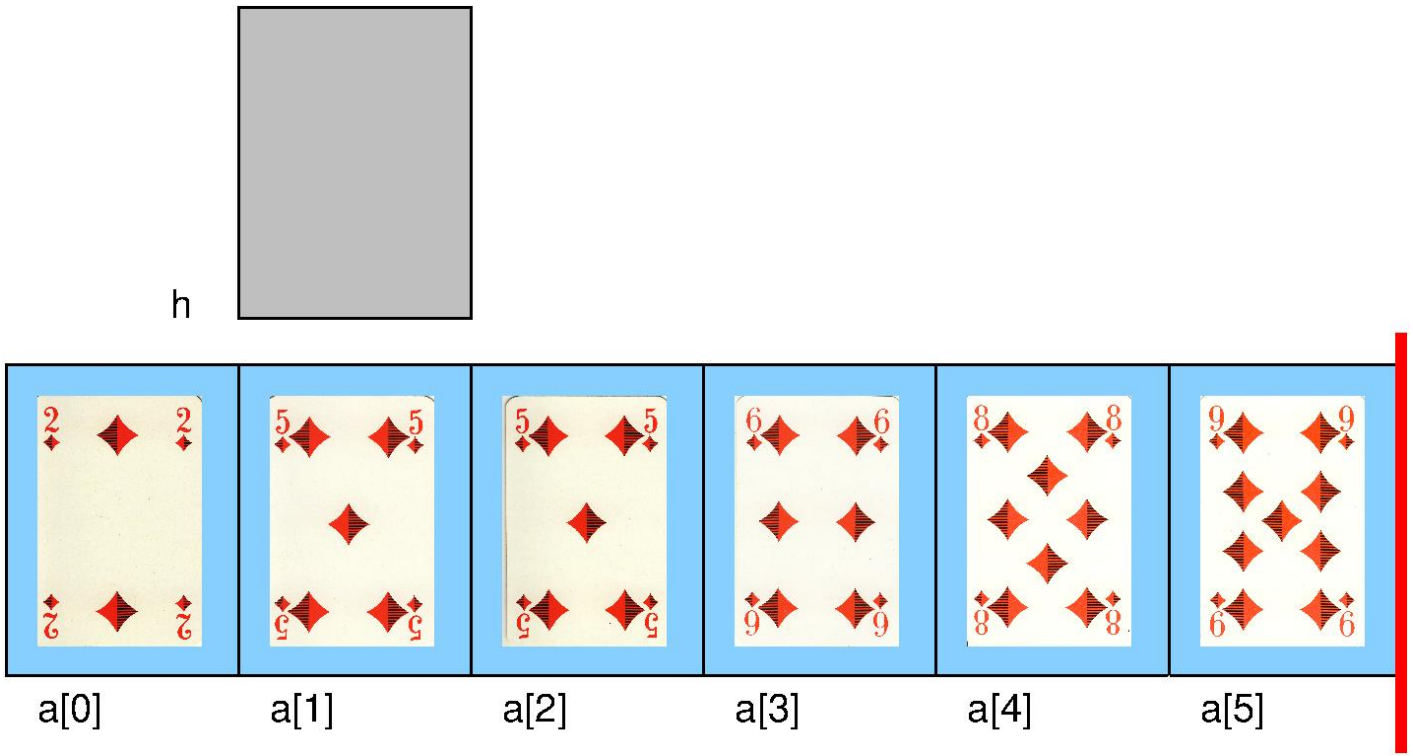


aufsteigend sortiertes Teilfeld, in das eingefügt wird: blau.
Teilfeld, dessen Elemente noch eingefügt werden müssen: grau.

Entnimm Element `a[5]`

Insertionsort – Idee

Sortieren durch Einfügen nach Vorbild des Kartenspielers



Durchlaufe sortiertes Teilfeld von rechts nach links bis zur Einfügeposition. Kein Verschieben notwendig, direkt in Lücke einfügen. Fertig.

Insertionsort ausprogrammiert

insertionsort.c – Funktion insertionsort zum aufsteigenden Sortieren eines Feldes

```c
#include<stdio.h>

#define N  6  //Anzahl zu sortierender Werte

void insertionsort(long a[])
{
  int i, k;
  long h;   //zum Zwischenspeichern des einzusortierenden Wertes

  for (i = 1; i < N; i++)
  {
    h = a[i]; //einzusortierenden Wert nehmen
    k = i;
    while ((k > 0) && (h < a[k-1]))
    {
      a[k] = a[k-1]; //groessere Feldwerte um eine Position nach rechts schieben
      k--;
    }
    a[k] = h; //einzusortierenden Wert in die passende Luecke im Feld a schreiben
  }
  return;
}
```

Insertionsort ausprogrammiert

insertionsort.c – `main`-Funktion und Initialisierung des zu sortierenden Feldes

```c
int main(void)
{
  long datenfeld[N] = {2, 8, 5, 6, 5, 9}; //zu sortierendes Datenfeld
  int i;

  insertionsort(datenfeld); //in-place sortieren
  for(i = 0; i < N; i++)
  {
    printf("%ld  ", datenfeld[i]); //Feldelemente ausgeben
  }
  printf("\n");
  return 0;
}
```

```
2  5  5  6  8  9
```

Steckbrief Insertionsort zum aufsteigenden Sortieren

Algorithmus . . . . . iterativ, vergleichsbasiert

Eigenschaften . . . . . in-place, stabil

Steckbrief Insertionsort zum aufsteigenden Sortieren

Algorithmus . . . . . iterativ, vergleichsbasiert

Eigenschaften . . . . . in-place, stabil

Worst Case . . . . . Folge anfänglich *absteigend* geordnet

Best Case . . . . . Folge anfänglich schon *aufsteigend* sortiert

---

Steckbrief Insertionsort zum aufsteigenden Sortieren

Algorithmus ..... iterativ, vergleichsbasiert

Eigenschaften ..... in-place, stabil

Worst Case ..... Folge anfänglich *absteigend* geordnet

Best Case ..... Folge anfänglich schon *aufsteigend* sortiert

---

Problemgröße n ..... Anzahl zu sortierender Feldelemente

Anzahl Vergleiche ..... $\leq n \cdot (n-1)/2$

Anzahl Elementverschiebungen ..... $< n^2$

Steckbrief Insertionsort zum aufsteigenden Sortieren

Algorithmus ..... iterativ, vergleichsbasiert

Eigenschaften ..... in-place, stabil

Worst Case ..... Folge anfänglich *absteigend* geordnet

Best Case ..... Folge anfänglich schon *aufsteigend* sortiert

---

Problemgröße n ..... Anzahl zu sortierender Feldelemente

Anzahl Vergleiche ..... $\leq n \cdot (n-1)/2$

Anzahl Elementverschiebungen ..... $< n^2$

Zeitkomplexität im Worst Case ..... $O(n^2)$

Insertionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `h`)
- im Best Case lineares Verhalten $O(n)$ (wenig Vergleiche, kein Verschieben)
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld sehr gut geeignet, arbeitet dort schneller, da kein Verschieben von Elementen erfolgen muss, um Platz an der Einfügeposition zu schaffen.
- vorteilhaft für vorsortierte Datenbestände, in die neue Schlüssel eingefügt werden

Insertionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `h`)
- im Best Case lineares Verhalten $O(n)$ (wenig Vergleiche, kein Verschieben)
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld sehr gut geeignet, arbeitet dort schneller, da kein Verschieben von Elementen erfolgen muss, um Platz an der Einfügeposition zu schaffen.
- vorteilhaft für vorsortierte Datenbestände, in die neue Schlüssel eingefügt werden

Insertionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `h`)
- im Best Case lineares Verhalten $O(n)$ (wenig Vergleiche, kein Verschieben)
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld sehr gut geeignet, arbeitet dort schneller, da kein Verschieben von Elementen erfolgen muss, um Platz an der Einfügeposition zu schaffen.
- vorteilhaft für vorsortierte Datenbestände, in die neue Schlüssel eingefügt werden

Insertionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `h`)
- im Best Case lineares Verhalten $O(n)$ (wenig Vergleiche, kein Verschieben)
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld sehr gut geeignet, arbeitet dort schneller, da kein Verschieben von Elementen erfolgen muss, um Platz an der Einfügeposition zu schaffen.
- vorteilhaft für vorsortierte Datenbestände, in die neue Schlüssel eingefügt werden

Insertionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `h`)
- im Best Case lineares Verhalten $O(n)$ (wenig Vergleiche, kein Verschieben)
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld sehr gut geeignet, arbeitet dort schneller, da kein Verschieben von Elementen erfolgen muss, um Platz an der Einfügeposition zu schaffen.
- vorteilhaft für vorsortierte Datenbestände, in die neue Schlüssel eingefügt werden

Insertionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `h`)
- im Best Case lineares Verhalten $O(n)$ (wenig Vergleiche, kein Verschieben)
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld sehr gut geeignet, arbeitet dort schneller, da kein Verschieben von Elementen erfolgen muss, um Platz an der Einfügeposition zu schaffen.
- vorteilhaft für vorsortierte Datenbestände, in die neue Schlüssel eingefügt werden

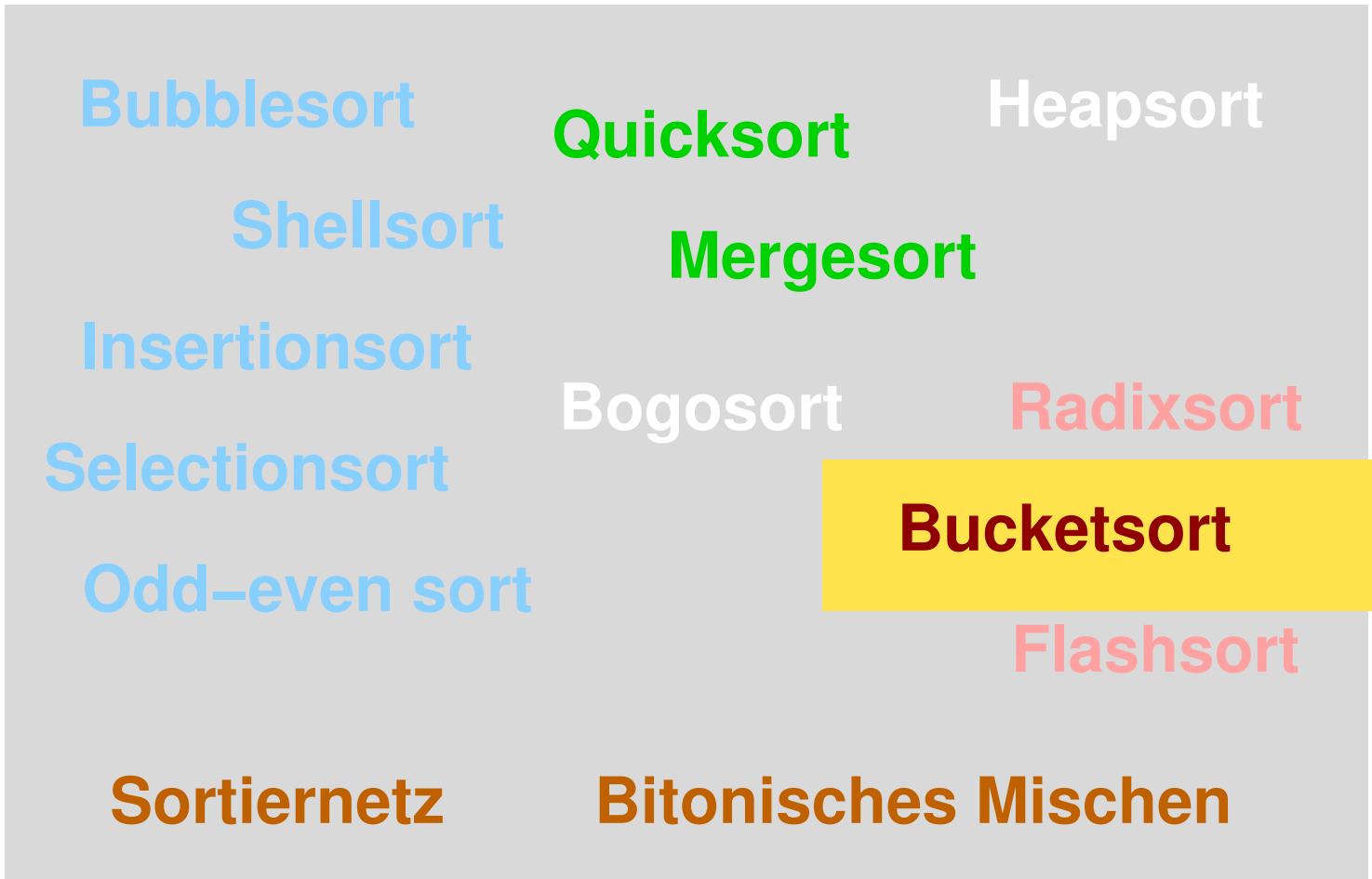
Insertionsort – Vorteile und Einsatzgebiete

- kaum Hilfsspeicher benötigt (nur Laufvariablen und `h`)
- im Best Case lineares Verhalten $O(n)$ (wenig Vergleiche, kein Verschieben)
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- Einsatz für kleinere Datenmengen (Richtwert: $n \leq 100$)
- zur Implementierung auf linearer Liste statt Feld sehr gut geeignet, arbeitet dort schneller, da kein Verschieben von Elementen erfolgen muss, um Platz an der Einfügeposition zu schaffen.
- vorteilhaft für vorsortierte Datenbestände, in die neue Schlüssel eingefügt werden

Nachteile: außerhalb des Best Case viele Umspeichervorgänge

Bucketsort

Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Bucketsort – Idee

www.deutschepost.de

Einsortieren von Briefen in *Fächer* nach Postleitzahlbereichen in einem Briefverteilungszentrum

Bucketsort – Idee

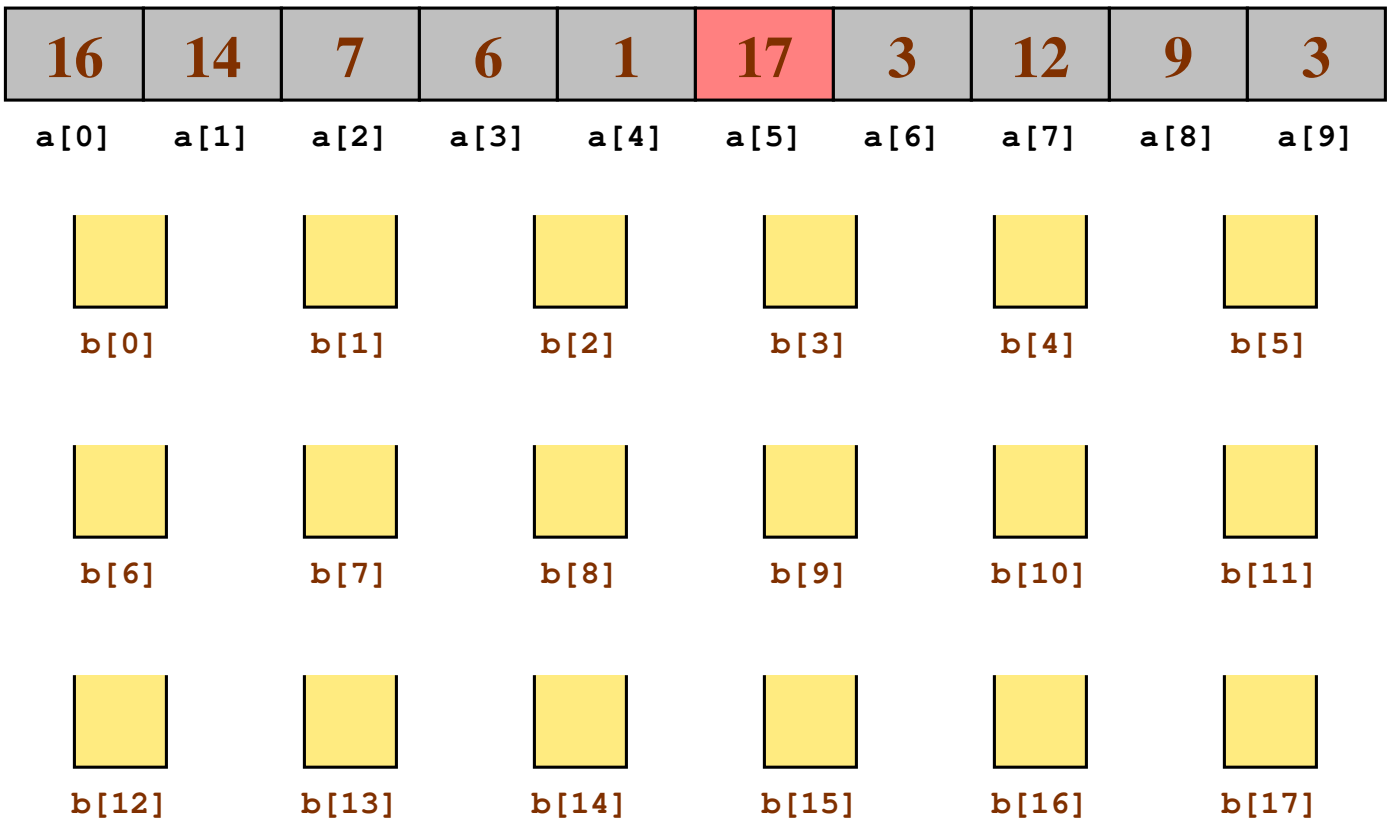
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost

16	14	7	6	1	17	3	12	9	3
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Phase 1: Größtes Element n im Feld `a` finden und $n + 1$ leere Fächer (buckets) im Feld `b` bereitstellen, Indizes $0 \ldots n$. Alle `b[j] = 0`

Bucketsort – Idee

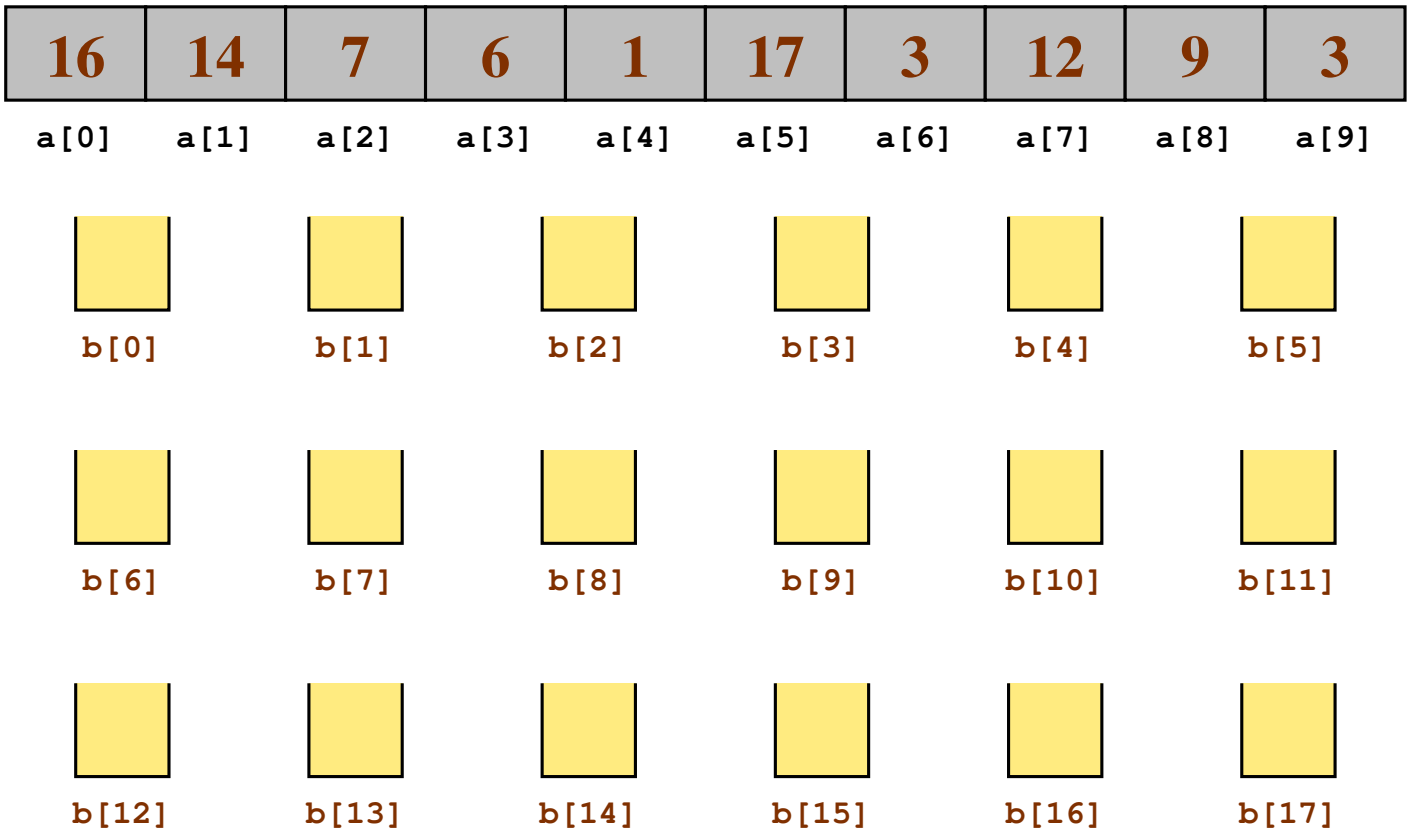
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 1: Größtes Element n im Feld `a` finden und $n + 1$ leere Fächer (buckets) im Feld `b` bereitstellen, Indizes $0 \ldots n$. Alle `b[j] = 0`

Bucketsort – Idee

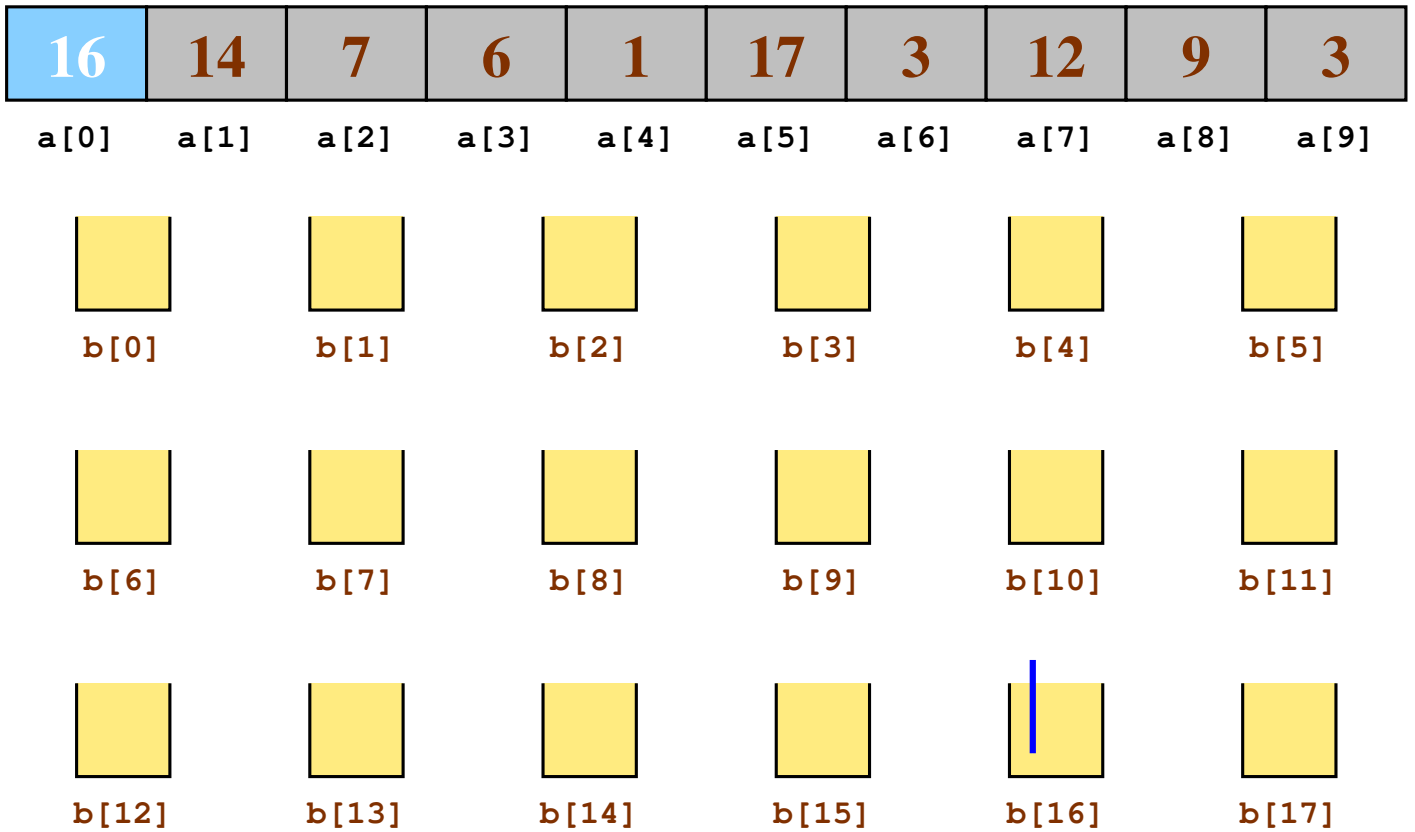
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 2: Befüllen der Fächer (buckets). Dazu wird Feld `a` durchlaufen und `b[a[i]]` um eins erhöht.

Bucketsort – Idee

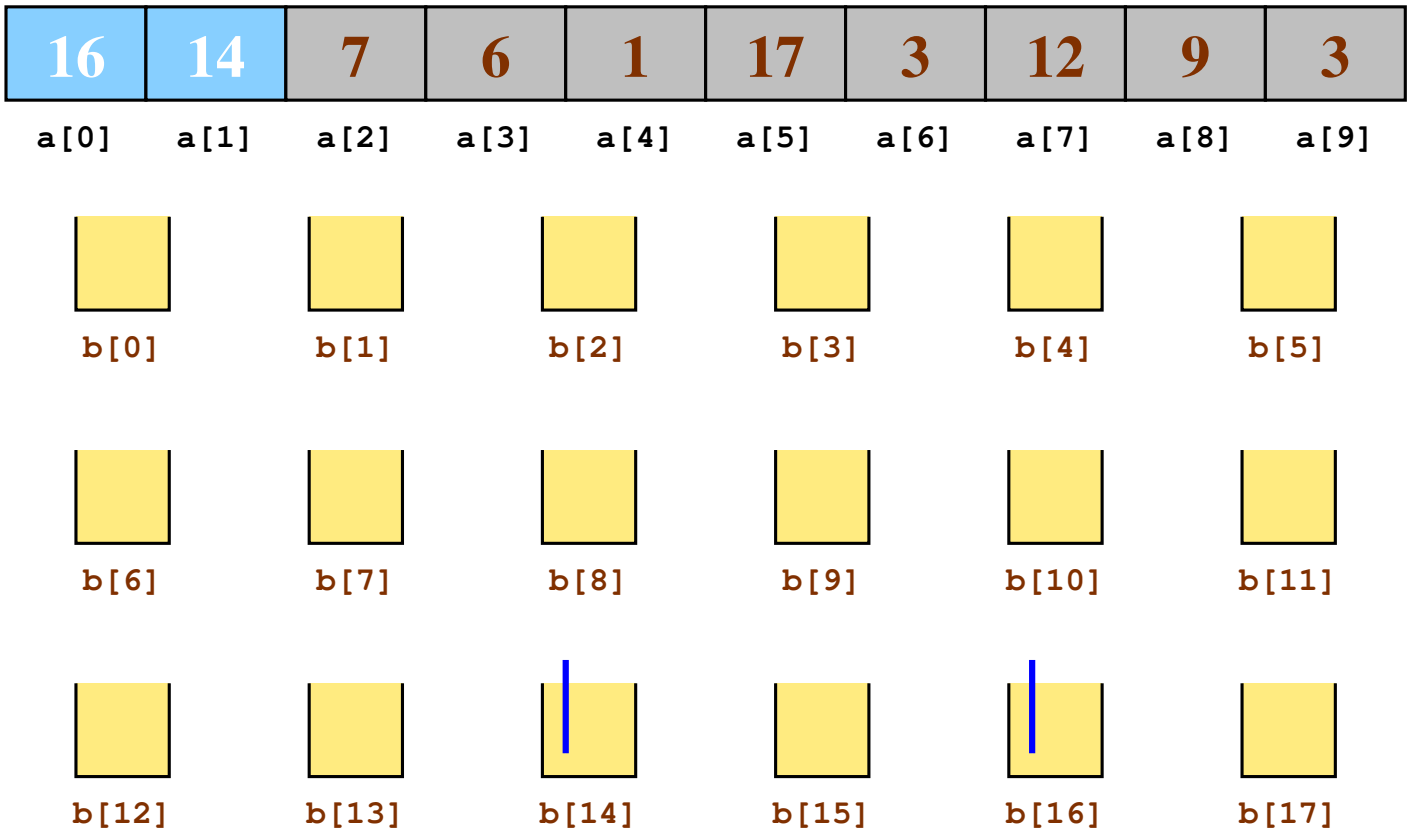
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 2: Befüllen der Fächer (buckets). Dazu wird Feld `a` durchlaufen und `b[a[i]]` um eins erhöht.

Bucketsort – Idee

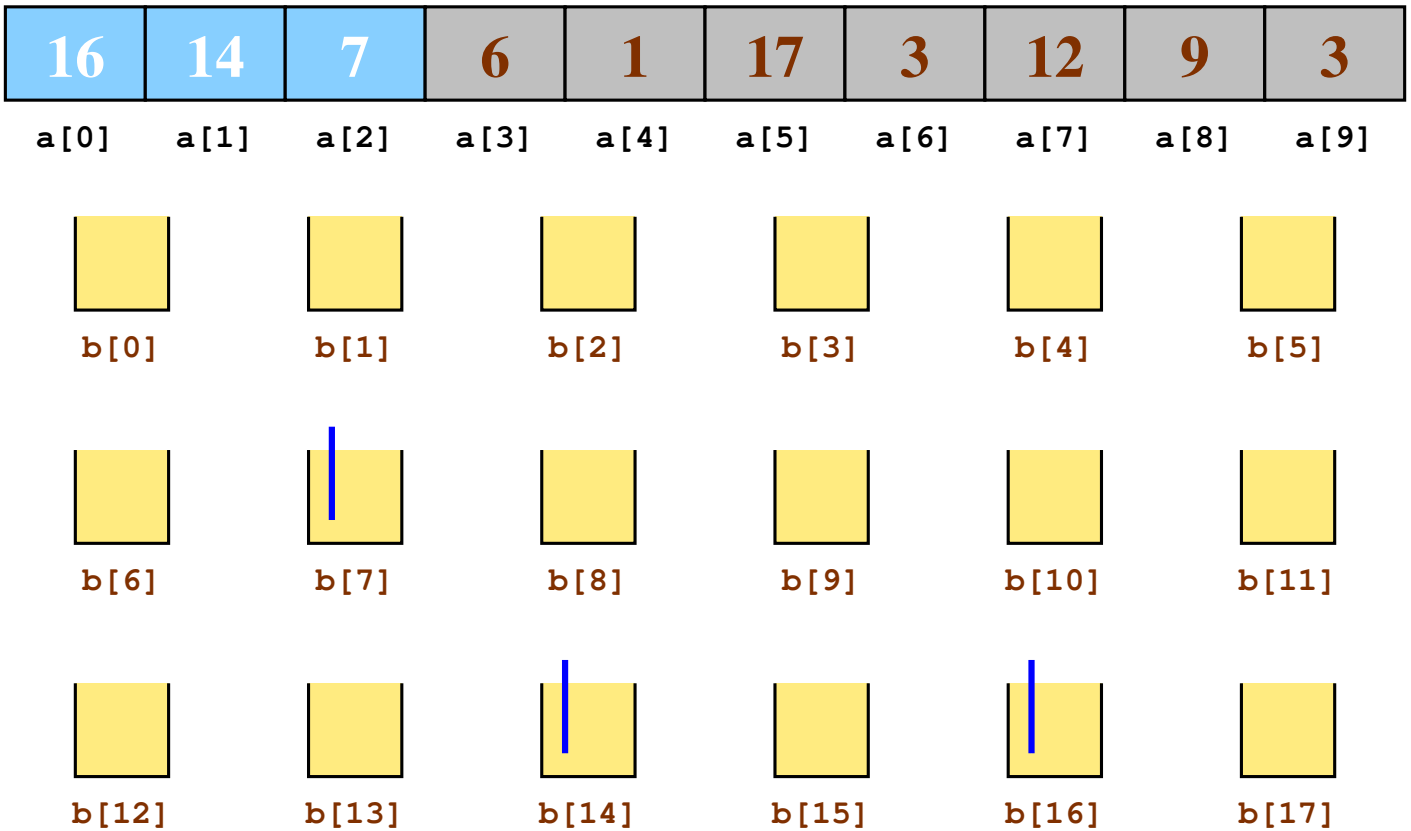
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 2: Befüllen der Fächer (buckets). Dazu wird Feld `a` durchlaufen und `b[a[i]]` um eins erhöht.

Bucketsort – Idee

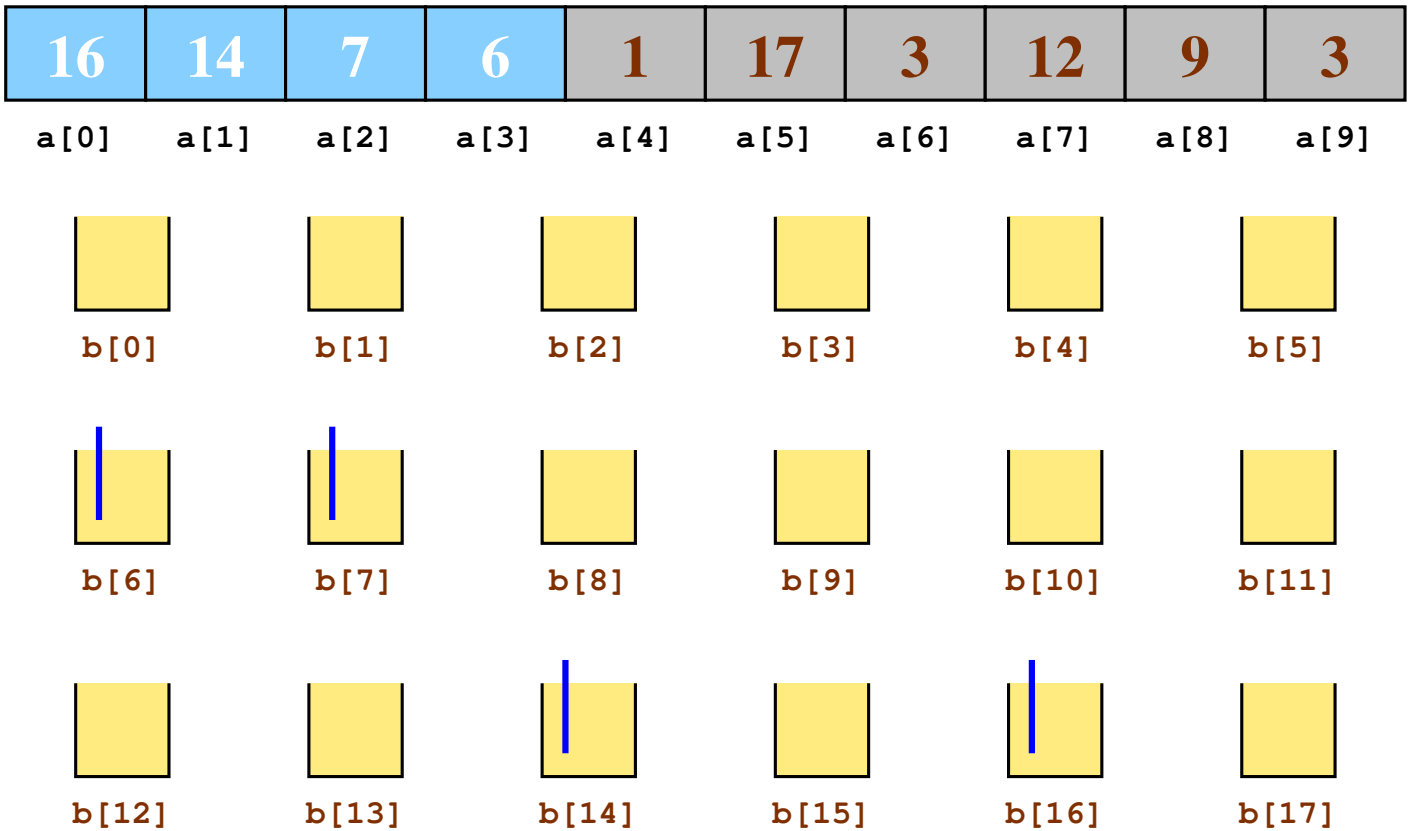
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 2: Befüllen der Fächer (buckets). Dazu wird Feld `a` durchlaufen und `b[a[i]]` um eins erhöht.

Bucketsort – Idee

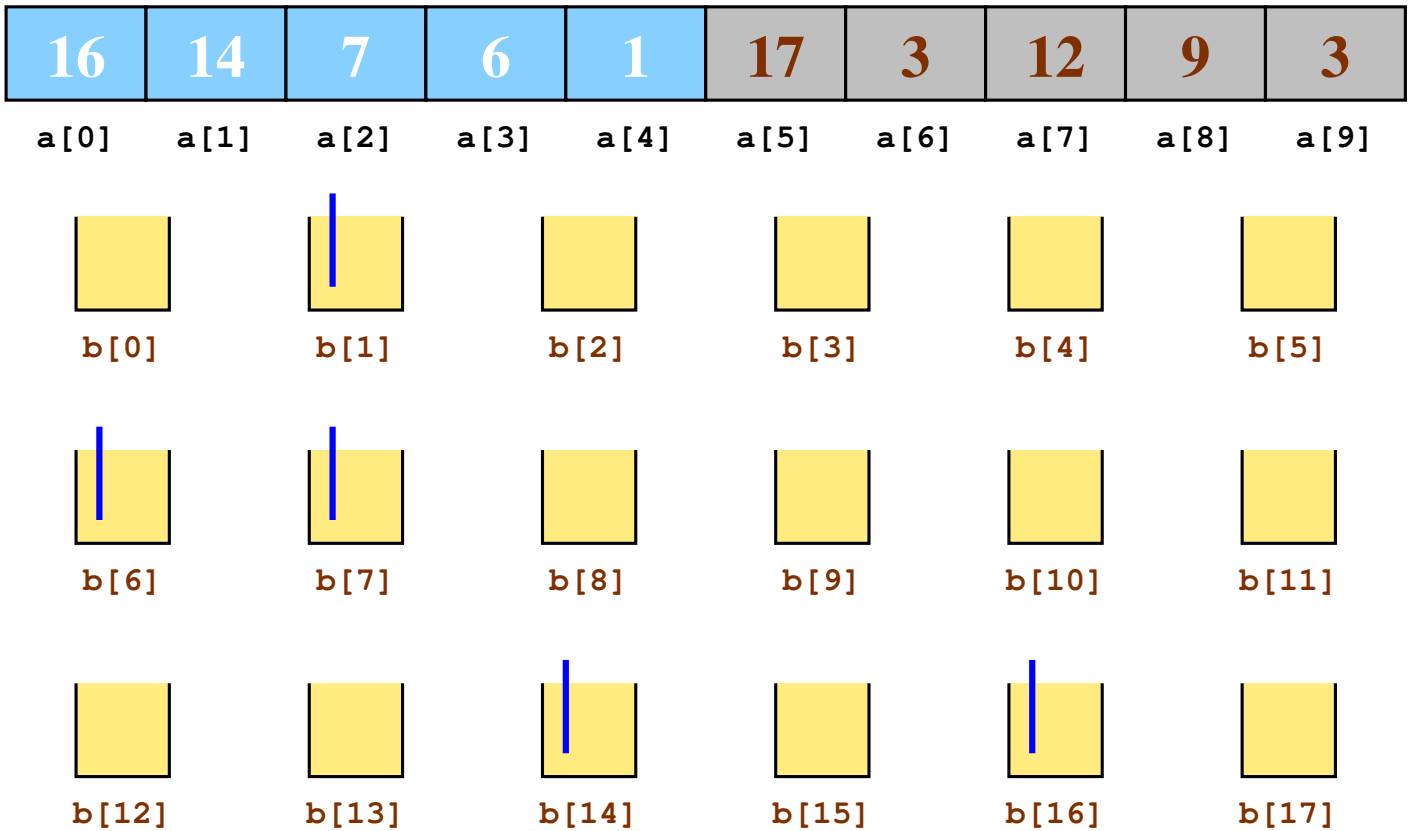
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 2: Befüllen der Fächer (buckets). Dazu wird Feld `a` durchlaufen und `b[a[i]]` um eins erhöht.

Bucketsort – Idee

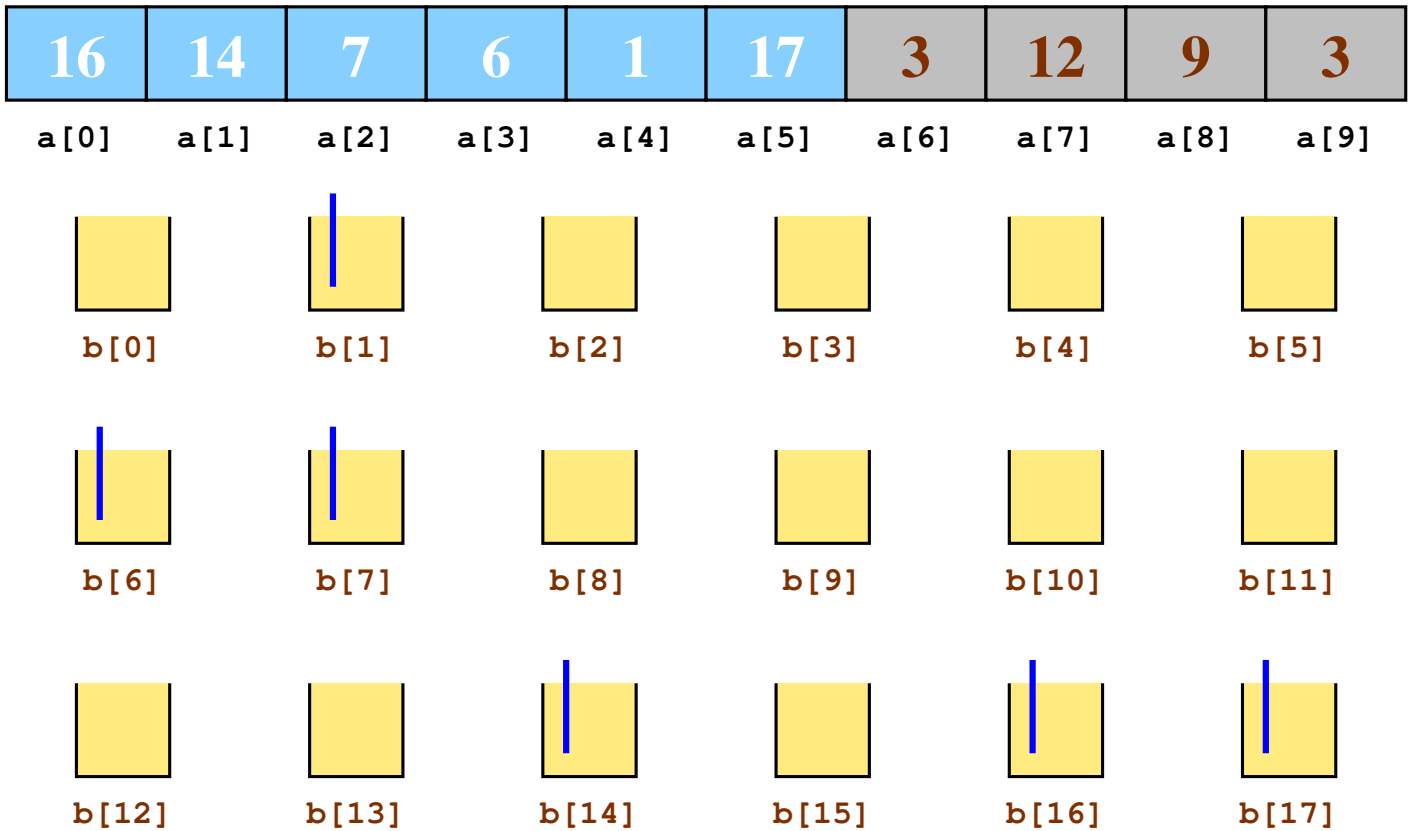
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 2: Befüllen der Fächer (buckets). Dazu wird Feld `a` durchlaufen und `b[a[i]]` um eins erhöht.

Bucketsort – Idee

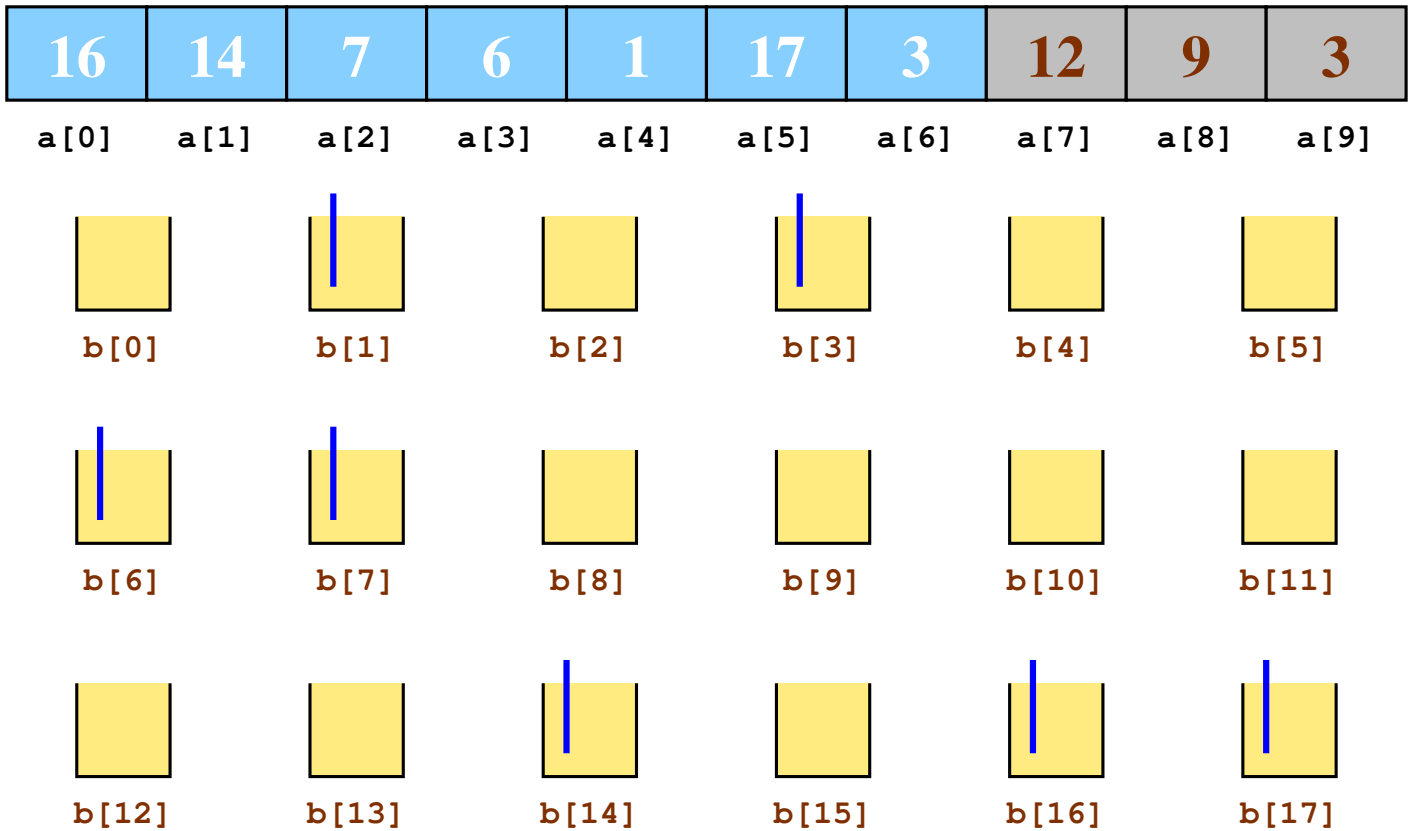
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 2: Befüllen der Fächer (buckets). Dazu wird Feld `a` durchlaufen und `b[a[i]]` um eins erhöht.

Bucketsort – Idee

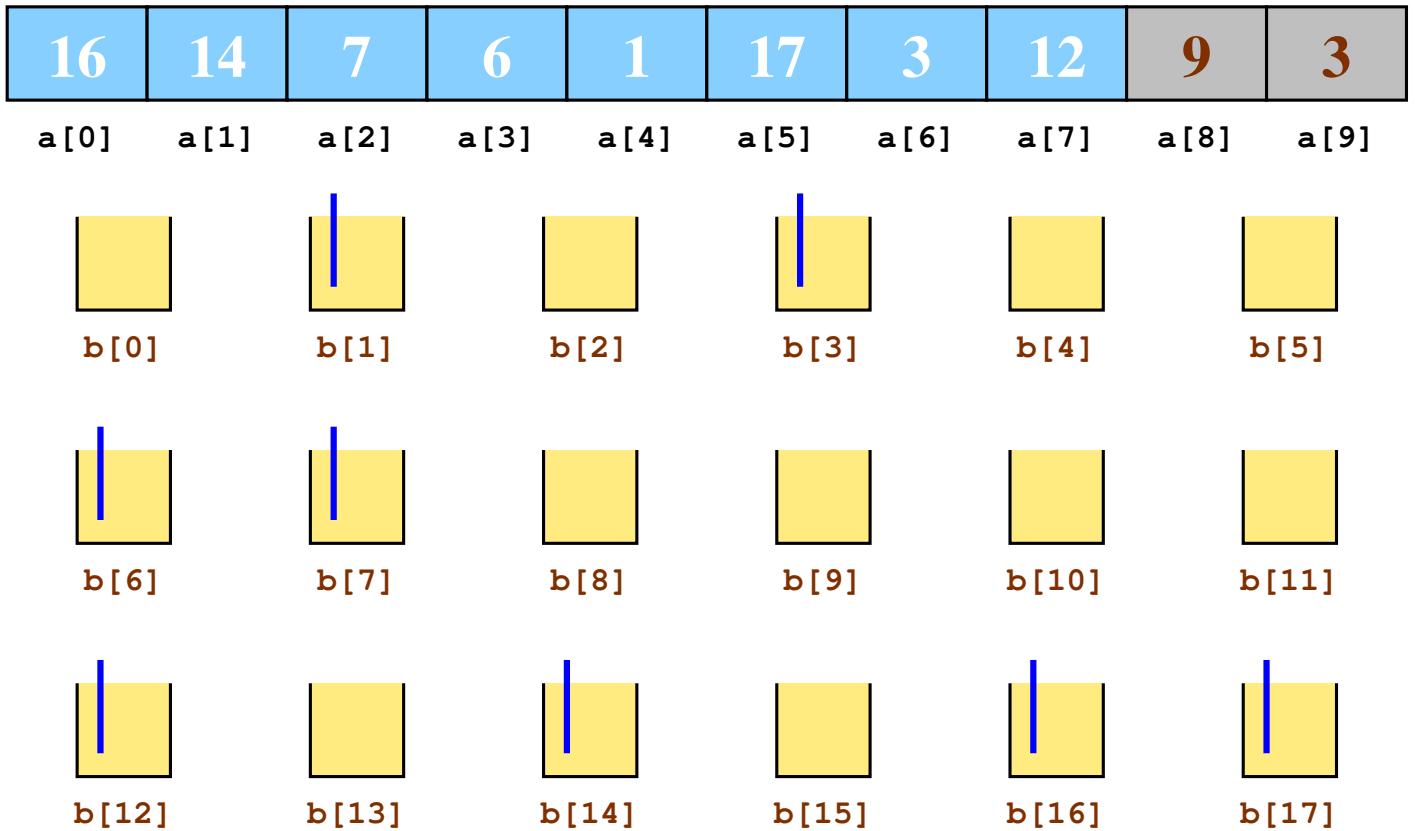
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 2: Befüllen der Fächer (buckets). Dazu wird Feld `a` durchlaufen und `b[a[i]]` um eins erhöht.

Bucketsort – Idee

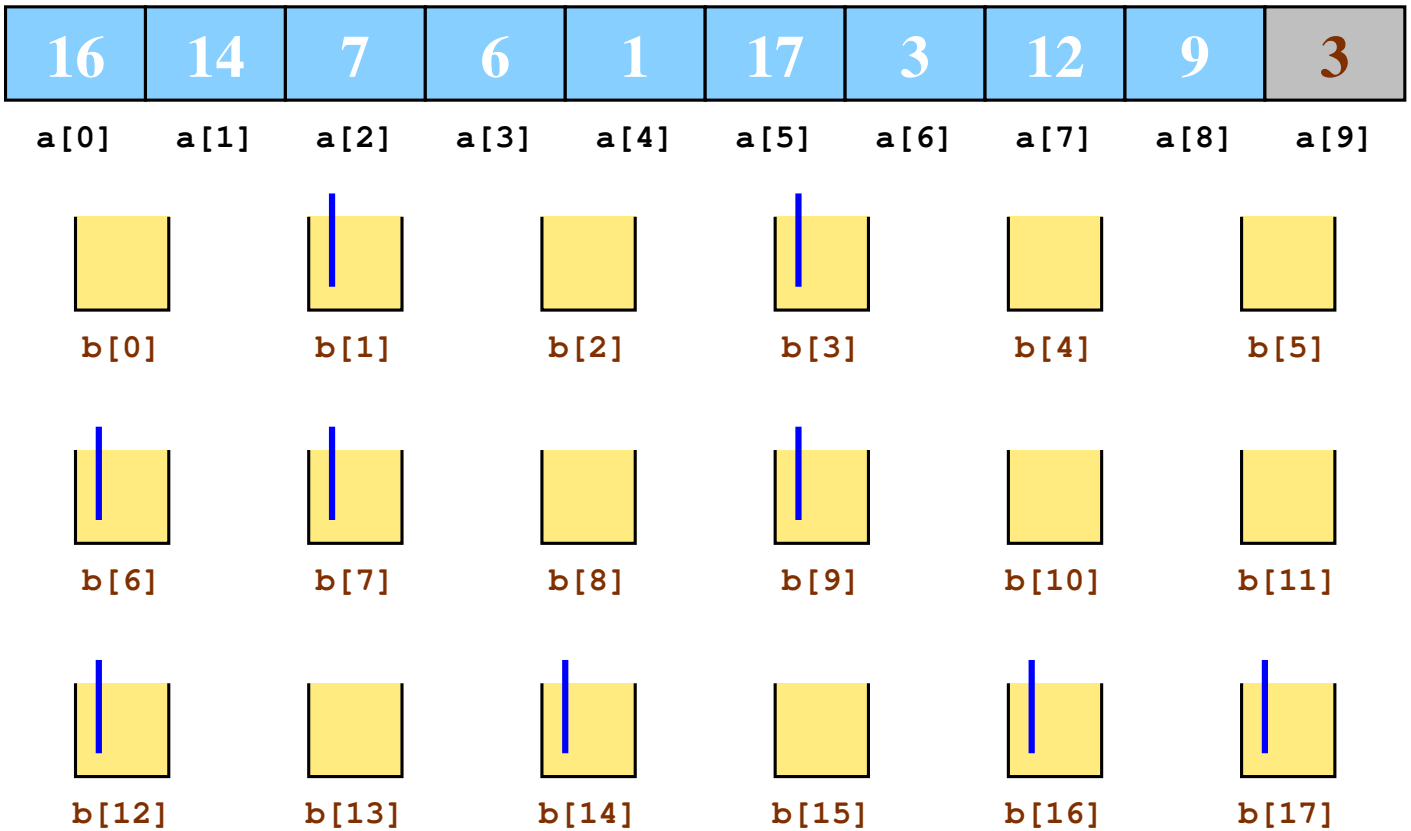
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 2: Befüllen der Fächer (buckets). Dazu wird Feld `a` durchlaufen und `b[a[i]]` um eins erhöht.

Bucketsort – Idee

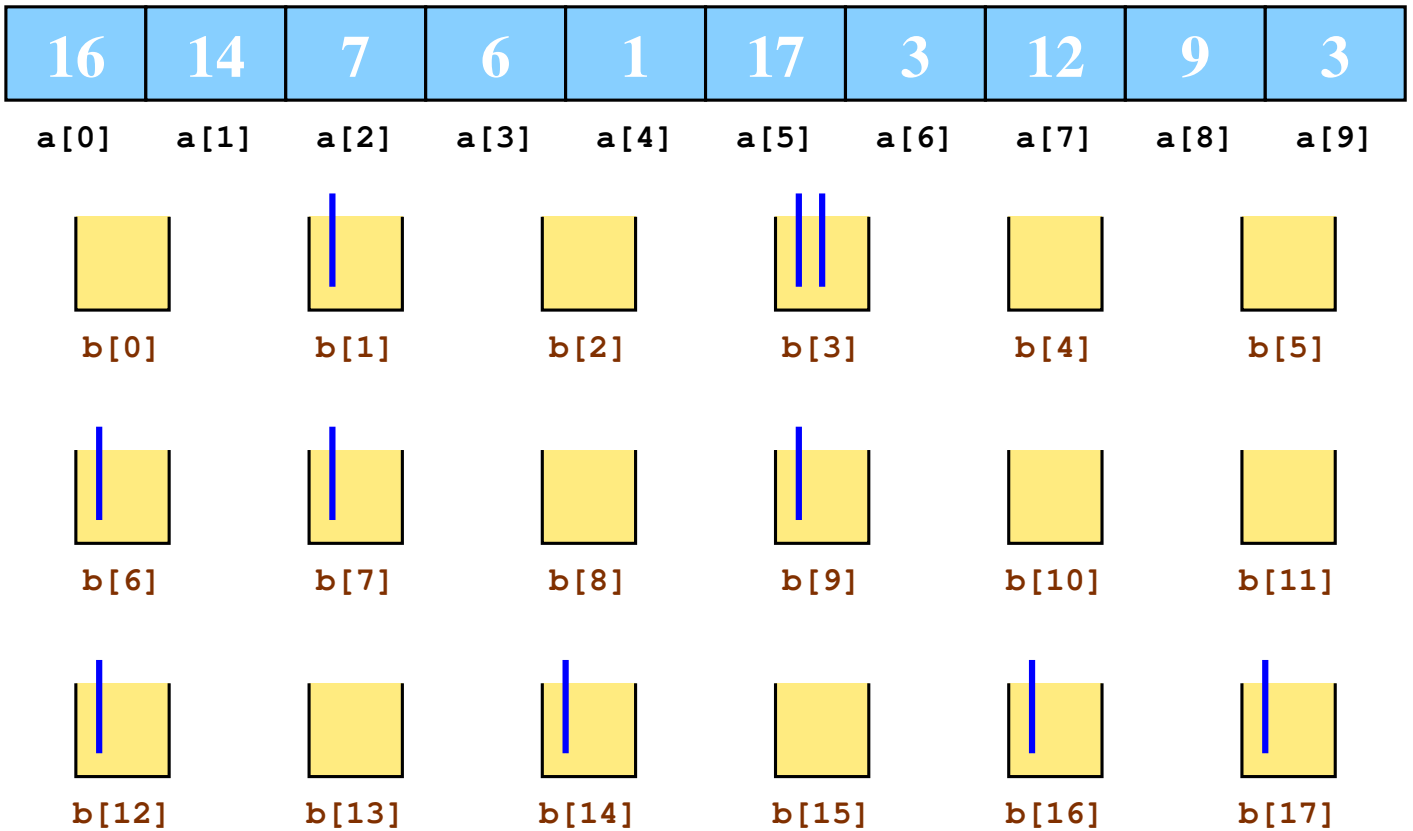
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 2: Befüllen der Fächer (buckets). Dazu wird Feld `a` durchlaufen und `b[a[i]]` um eins erhöht.

Bucketsort – Idee

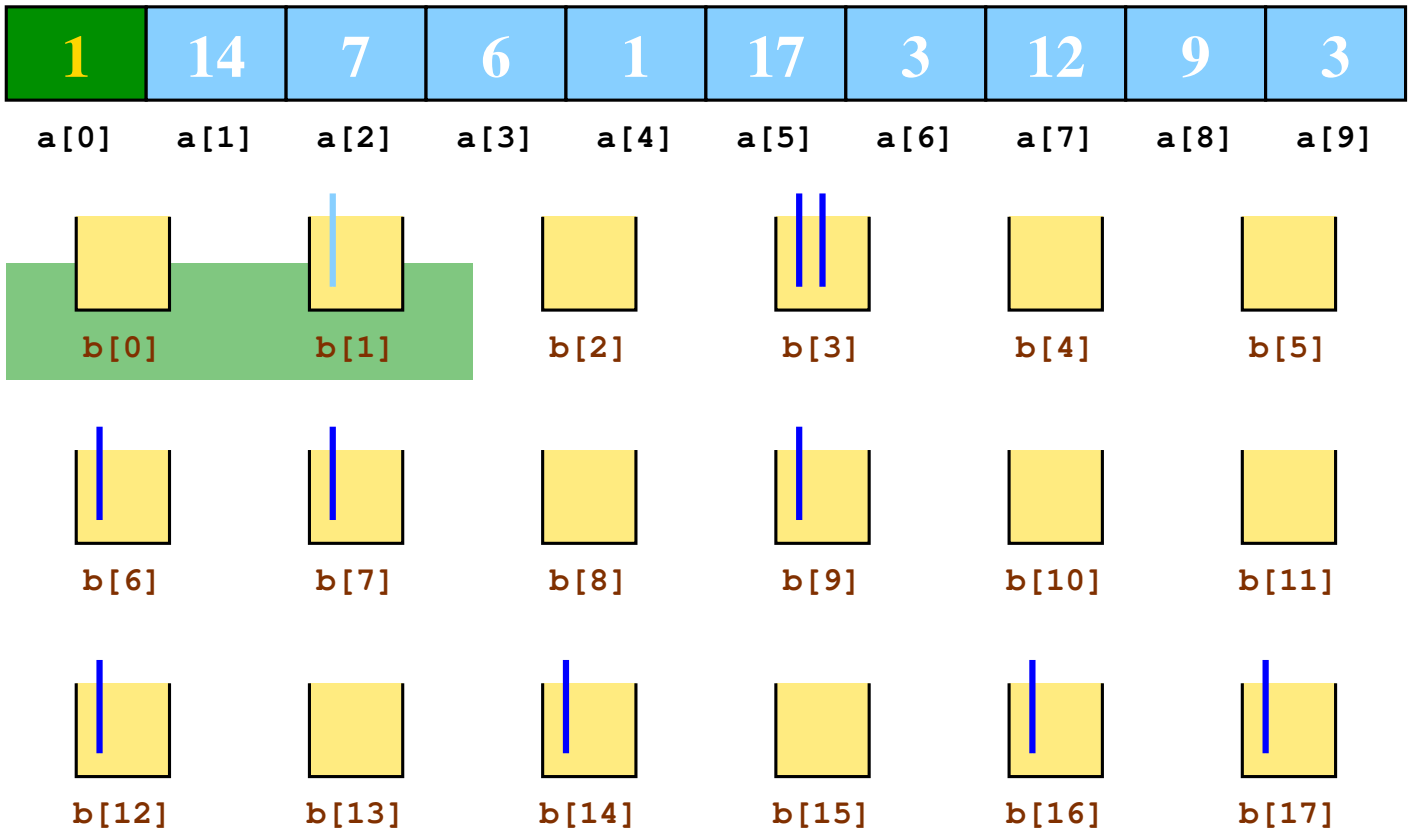
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 2: Befüllen der Fächer (buckets). Dazu wird Feld `a` durchlaufen und `b[a[i]]` um eins erhöht.

Bucketsort – Idee

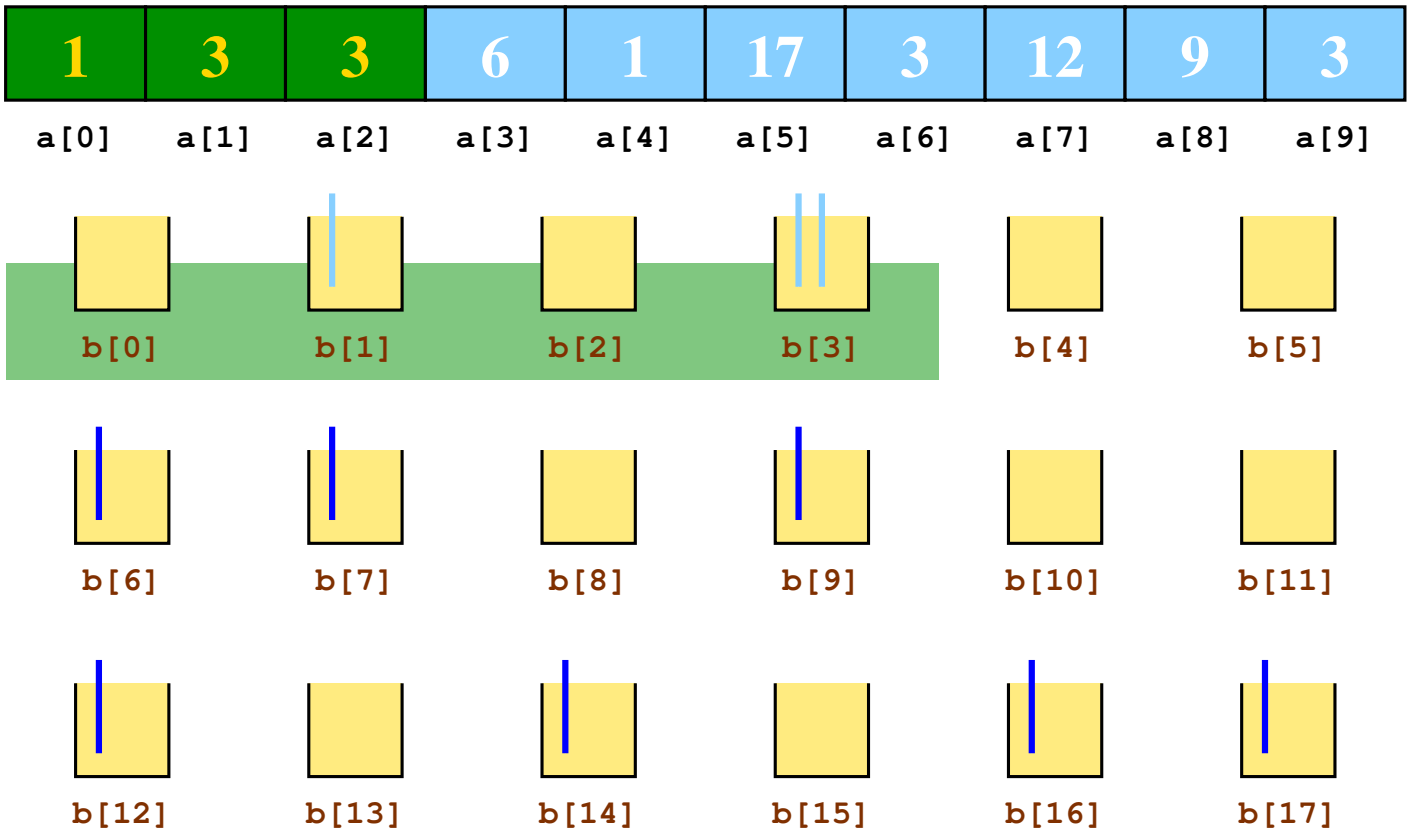
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 3: Bucket-Feld `b` durchlaufen. Immer dann, wenn `b[k] > 0`, den Wert `k` ins Feld `a` `b[k]`-mal hintereinander zurückschreiben.

Bucketsort – Idee

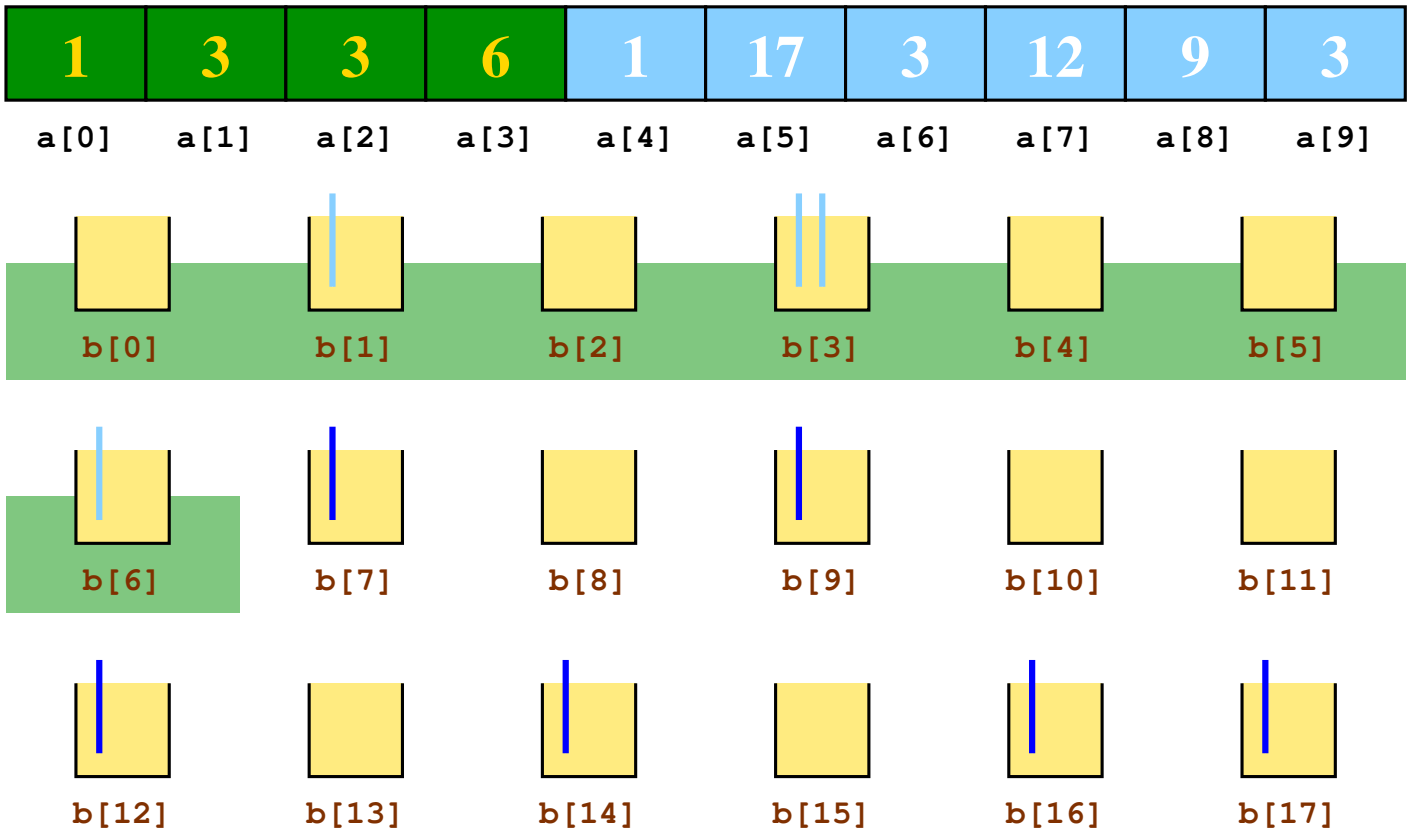
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 3: Bucket-Feld `b` durchlaufen. Immer dann, wenn `b[k] > 0`, den Wert `k` ins Feld `a` `b[k]`-mal hintereinander zurückschreiben.

Bucketsort – Idee

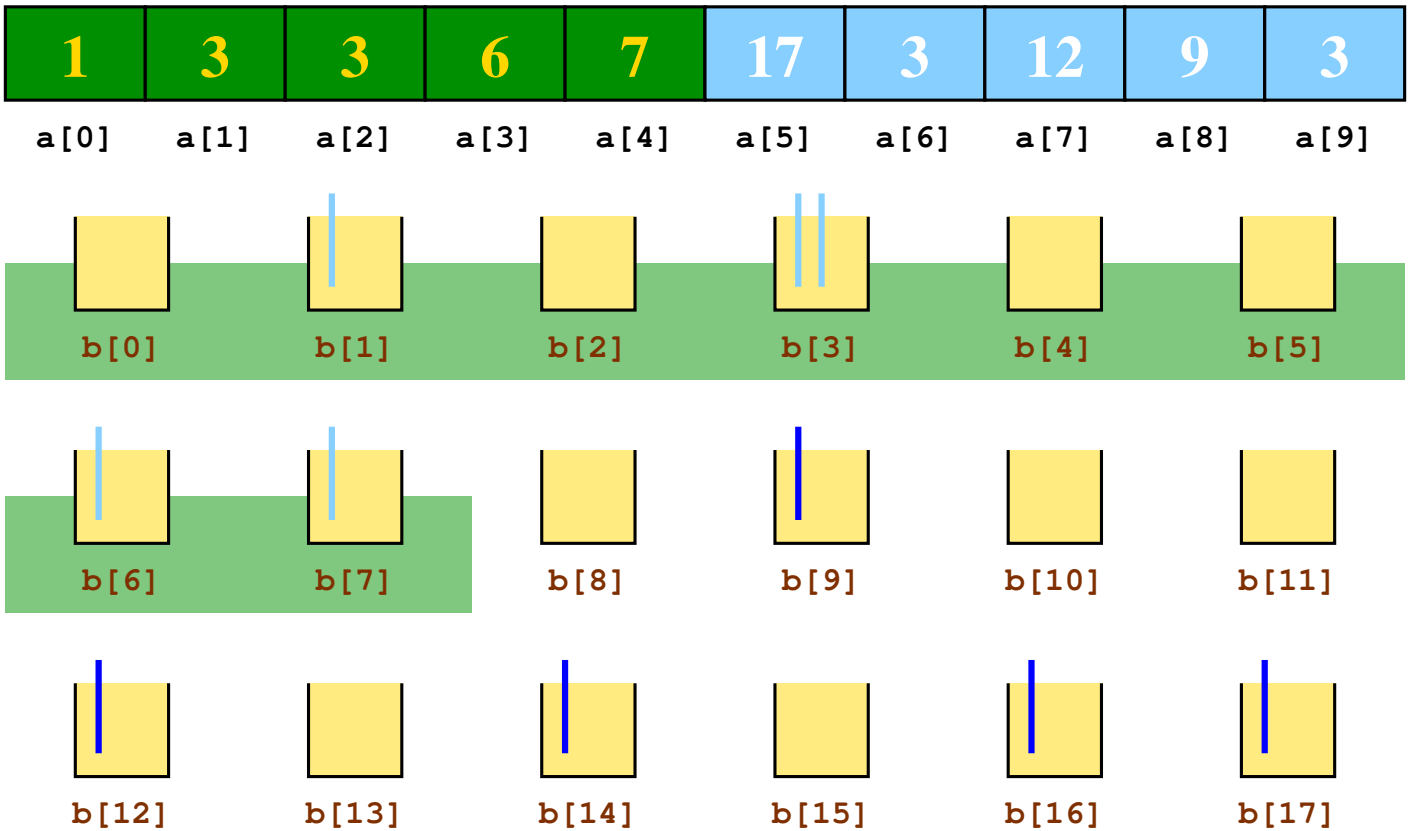
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 3: Bucket-Feld `b` durchlaufen. Immer dann, wenn `b[k] > 0`, den Wert `k` ins Feld `a` `b[k]`-mal hintereinander zurückschreiben.

Bucketsort – Idee

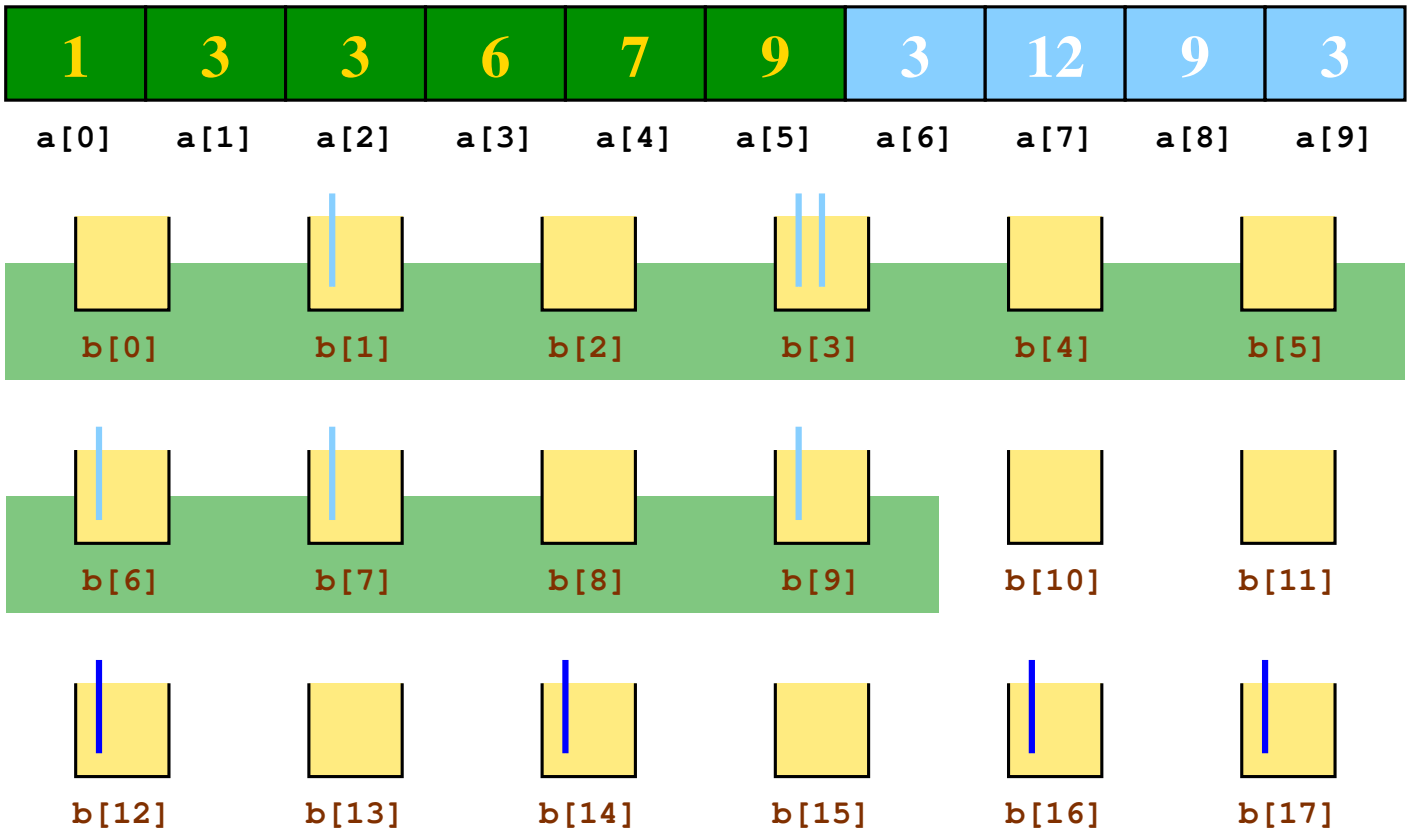
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 3: Bucket-Feld `b` durchlaufen. Immer dann, wenn `b[k] > 0`, den Wert `k` ins Feld `a` `b[k]`-mal hintereinander zurückschreiben.

Bucketsort – Idee

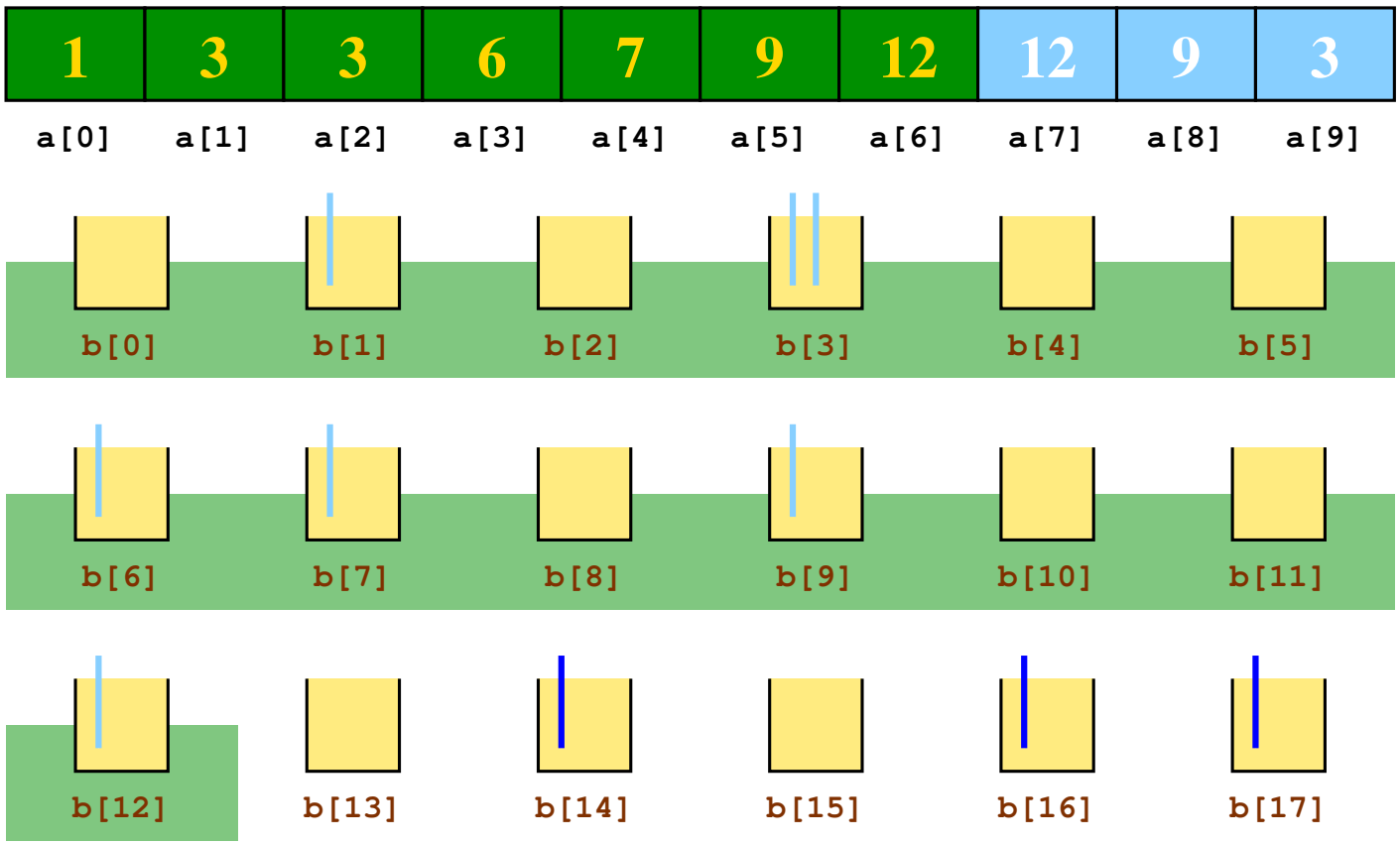
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 3: Bucket-Feld `b` durchlaufen. Immer dann, wenn `b[k] > 0`, den Wert `k` ins Feld `a` `b[k]`-mal hintereinander zurückschreiben.

Bucketsort – Idee

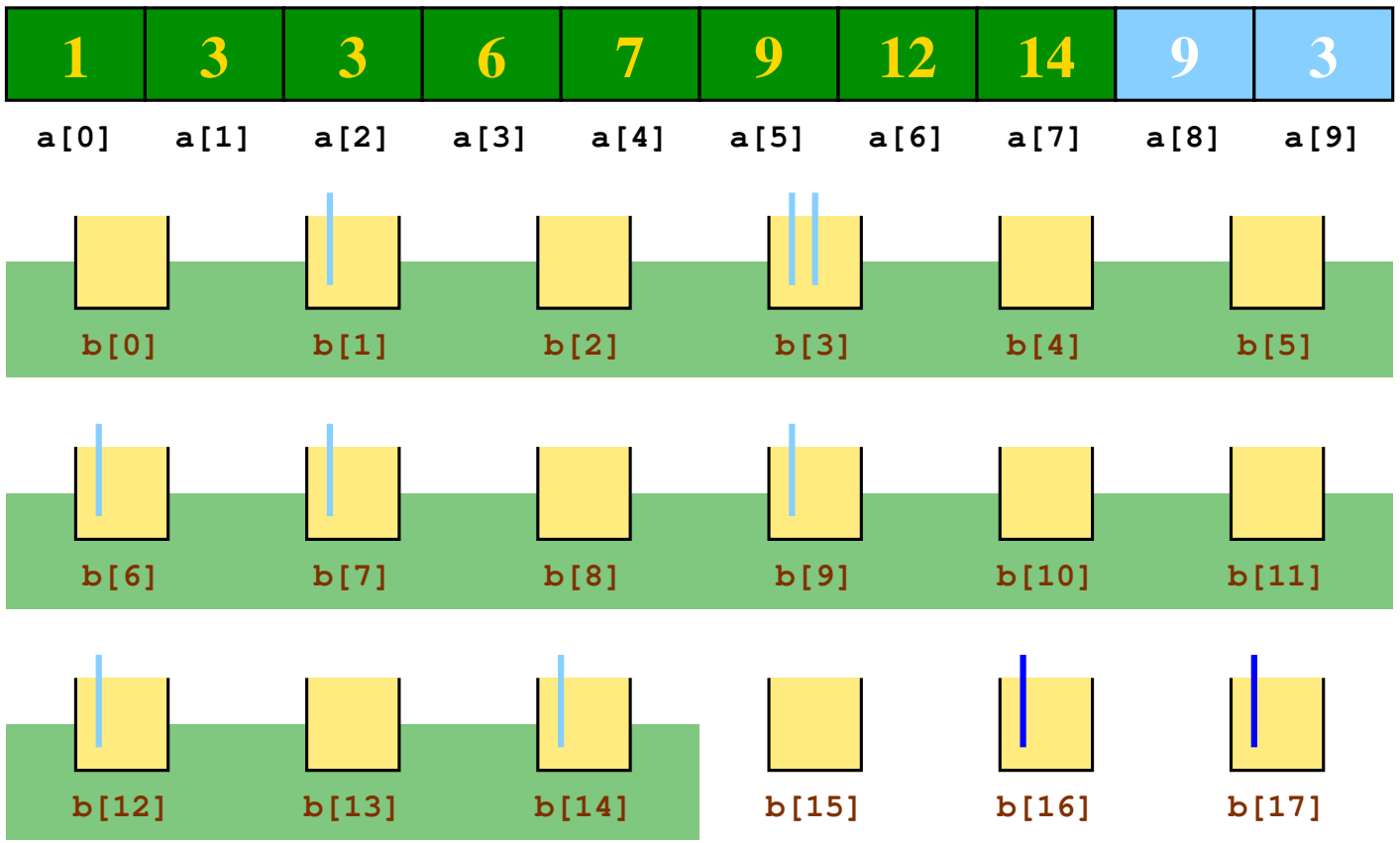
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 3: Bucket-Feld `b` durchlaufen. Immer dann, wenn `b[k] > 0`, den Wert `k` ins Feld `a` `b[k]`-mal hintereinander zurückschreiben.

Bucketsort – Idee

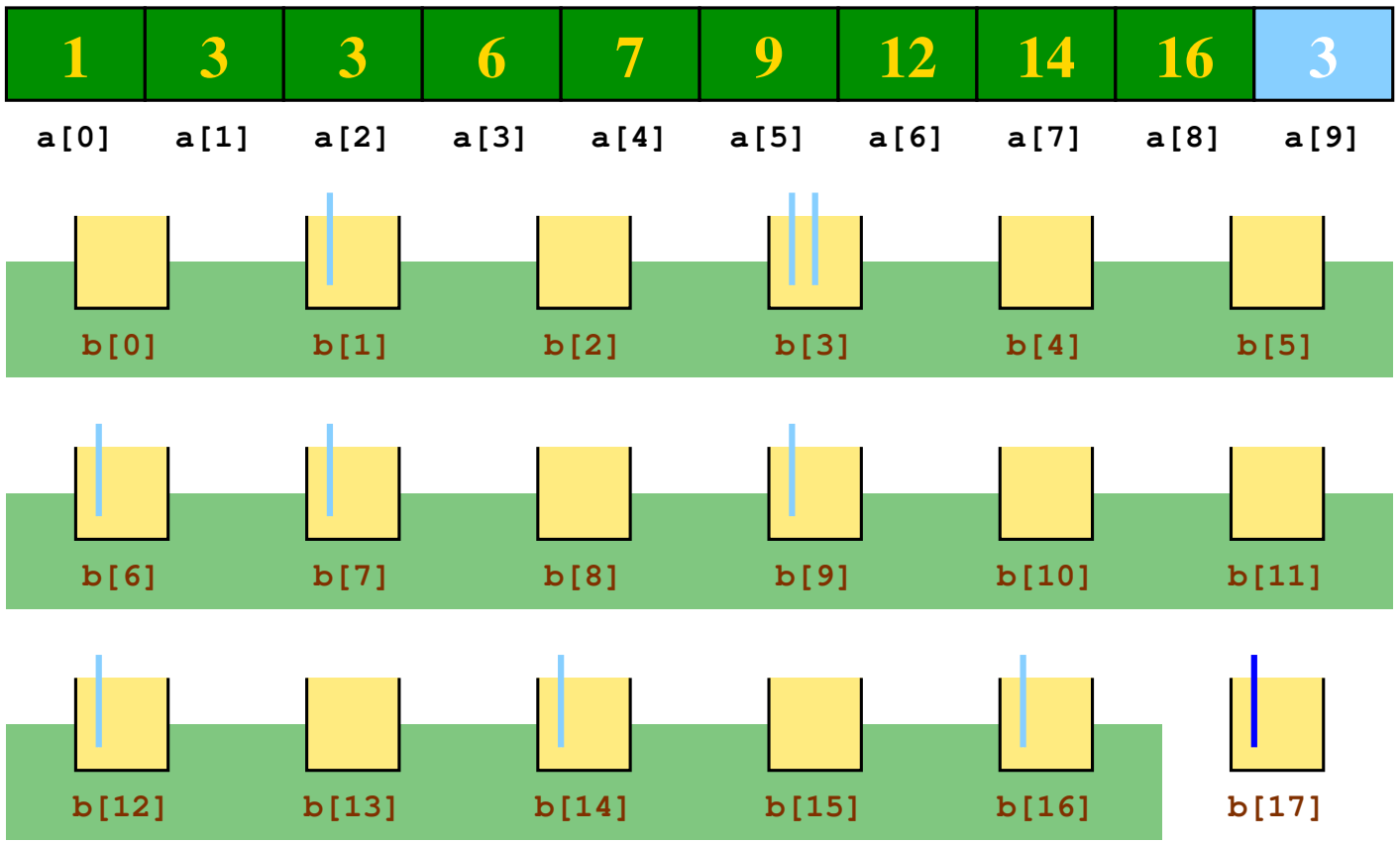
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 3: Bucket-Feld `b` durchlaufen. Immer dann, wenn `b[k] > 0`, den Wert `k` ins Feld `a` `b[k]`-mal hintereinander zurückschreiben.

Bucketsort – Idee

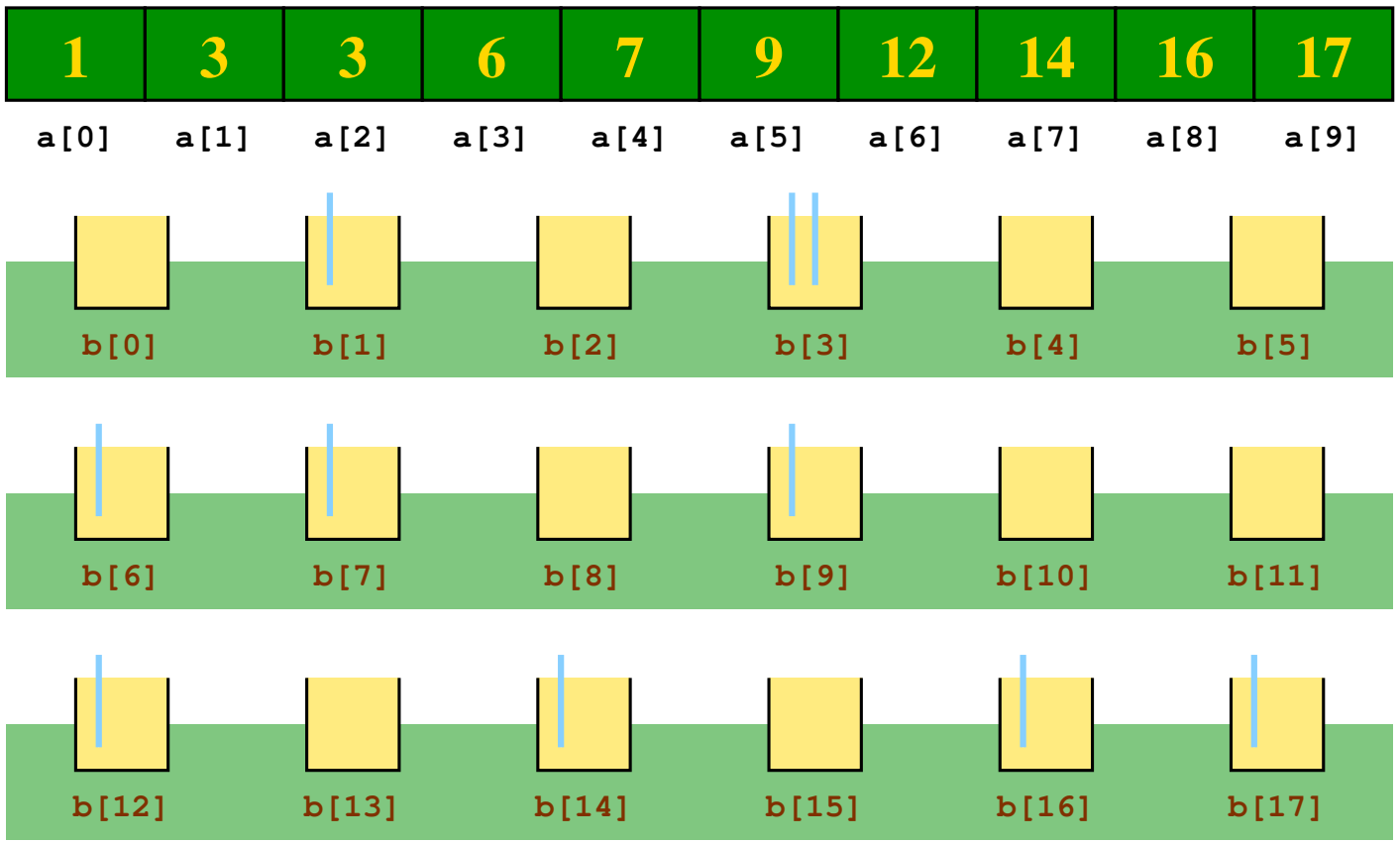
Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 3: Bucket-Feld `b` durchlaufen. Immer dann, wenn `b[k] > 0`, den Wert `k` ins Feld `a` `b[k]`-mal hintereinander zurückschreiben.

Bucketsort – Idee

Sortieren durch Fachverteilen nach dem Vorbild der Briefpost



Phase 3: Bucket-Feld `b` durchlaufen. Immer dann, wenn `b[k] > 0`, den Wert `k` ins Feld `a` `b[k]`-mal hintereinander zurückschreiben.

Bucketsort ausprogrammiert

bucketsort.c – Funktion sort zum aufsteigenden Sortieren eines Feldes

```
#include <stdio.h>

#define L 10    //Anzahl Elemente im zu sortierenden Feld
#define M 65535 //pow(2,16)-1 groesster darstellbarer Wert in unsigned short


void sort(int n, unsigned short a[])          // sortiere Feld a
{
    unsigned short b[M];           // M Buckets, davon bis Index n benoetigt
    int i, j, k;                                // Laufvariablen

    for (i=0; i < n; i++)
    {
      b[i] = 0;                                 // setze alle Buckets auf 0
    }

    for (i=0; i < L; i++)                       // fuer jedes Feldelement
    {
      b[a[i]]++;                                // zustaendiges Bucket erhoehen
    }

    k = 0;
    for (i=0; i < n; i++) {                     // fuer jedes Bucket
      for (j=0; j < b[i]; j++) {                // gemaess Zaehlerstand
        a[k++] = i;                             // seinen Index uebernehmen
      }
    }
}
```

Bucketsort ausprogrammiert

bucketsort.c – `main`-Funktion und Initialisierung des zu sortierenden Feldes

```
b[0]: 0
b[1]: 1
b[2]: 0
b[3]: 2
b[4]: 0
b[5]: 0
b[6]: 1
b[7]: 1
b[8]: 0
b[9]: 1
b[10]: 0
b[11]: 0
b[12]: 1
b[13]: 0
b[14]: 1
b[15]: 0
b[16]: 1
b[17]: 1
```

```c
int main(void)
{
    unsigned short arr[L] = {16, 14, 7, 6, 1, 17, 3, 12, 9, 3};
    int m, n;

    n = arr[0];                    // groesstes Elem im Feld finden
    for (m = 0; m < L; m++)
    {
      if (n < arr[m])
      {
        n = arr[m];
      }
    }
    n++;
    sort(n, arr); // BucketSort aufrufen. n: Anzahl benoetigter Buckets
    for (m = 0; m < L; m++)
    {   // sortiertes Feld ausgeben
      printf("%u  ", arr[m]);
    }
    printf("\n");
    return 0;
}
```

```
1  3  3  6  7  9  12  14  16  17
```

Steckbrief Bucketsort zum aufsteigenden Sortieren

Algorithmus . . . . . iterativ, ohne Vergleiche

Eigenschaften . . . . . ex-situ (nicht in-place), stabil

Steckbrief Bucketsort zum aufsteigenden Sortieren

Algorithmus ..... iterativ, ohne Vergleiche

Eigenschaften ..... ex-situ (nicht in-place), stabil

Worst Case .. Worst Case und Best Case nicht unterscheidbar

Best Case ..... beliebige Permutation der Feldelemente

Steckbrief Bucketsort zum aufsteigenden Sortieren

Algorithmus ..... iterativ, ohne Vergleiche

Eigenschaften ..... ex-situ (nicht in-place), stabil

Worst Case .. Worst Case und Best Case nicht unterscheidbar

Best Case ..... beliebige Permutation der Feldelemente

Problemgröße n ..... Anzahl zu sortierender Feldelemente

Anzahl Buckets ..... m

Anzahl Arbeitsschritte ..... ungefähr $n + m$

Steckbrief Bucketsort zum aufsteigenden Sortieren

Algorithmus ..... iterativ, ohne Vergleiche

Eigenschaften ..... ex-situ (nicht in-place), stabil

Worst Case .. Worst Case und Best Case nicht unterscheidbar

Best Case ..... beliebige Permutation der Feldelemente

Problemgröße n ..... Anzahl zu sortierender Feldelemente

Anzahl Buckets ..... m

Anzahl Arbeitsschritte ..... ungefähr $n + m$

Zeitkomplexität ..... $O(n + m)$

Bucketsort – Vorteile und Einsatzgebiete

- äußerst schnelles Sortierverfahren, das zudem keine Elementvergleiche benötigt
- geeignet für große Datenmengen, wenn die Schlüssel durch natürliche Zahlen dargestellt sind und die Schlüsselwerte eng beieinander liegen
- stets lineares Verhalten $O(n + m)$
- Laufzeitverhalten im Best Case und Worst Case identisch
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- vorteilhaft für große Anzahlen von 0 an durchnummerierter Objekte

Bucketsort – Vorteile und Einsatzgebiete

- äußerst schnelles Sortierverfahren, das zudem keine Elementvergleiche benötigt
- geeignet für große Datenmengen, wenn die Schlüssel durch natürliche Zahlen dargestellt sind und die Schlüsselwerte eng beieinander liegen
- stets lineares Verhalten $O(n + m)$
- Laufzeitverhalten im Best Case und Worst Case identisch
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- vorteilhaft für große Anzahlen von 0 an durchnummerierter Objekte

Bucketsort – Vorteile und Einsatzgebiete

- äußerst schnelles Sortierverfahren, das zudem keine Elementvergleiche benötigt
- geeignet für große Datenmengen, wenn die Schlüssel durch natürliche Zahlen dargestellt sind und die Schlüsselwerte eng beieinander liegen
- stets lineares Verhalten $O(n + m)$
- Laufzeitverhalten im Best Case und Worst Case identisch
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- vorteilhaft für große Anzahlen von 0 an durchnummerierter Objekte

Bucketsort – Vorteile und Einsatzgebiete

- äußerst schnelles Sortierverfahren, das zudem keine Elementvergleiche benötigt
- geeignet für große Datenmengen, wenn die Schlüssel durch natürliche Zahlen dargestellt sind und die Schlüsselwerte eng beieinander liegen
- stets lineares Verhalten $O(n + m)$
- Laufzeitverhalten im Best Case und Worst Case identisch
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- vorteilhaft für große Anzahlen von 0 an durchnummerierter Objekte

Bucketsort – Vorteile und Einsatzgebiete

- äußerst schnelles Sortierverfahren, das zudem keine Elementvergleiche benötigt
- geeignet für große Datenmengen, wenn die Schlüssel durch natürliche Zahlen dargestellt sind und die Schlüsselwerte eng beieinander liegen
- stets lineares Verhalten $O(n + m)$
- Laufzeitverhalten im Best Case und Worst Case identisch
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- vorteilhaft für große Anzahlen von 0 an durchnummerierter Objekte

Bucketsort – Vorteile und Einsatzgebiete

- äußerst schnelles Sortierverfahren, das zudem keine Elementvergleiche benötigt
- geeignet für große Datenmengen, wenn die Schlüssel durch natürliche Zahlen dargestellt sind und die Schlüsselwerte eng beieinander liegen
- stets lineares Verhalten $O(n + m)$
- Laufzeitverhalten im Best Case und Worst Case identisch
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- vorteilhaft für große Anzahlen von 0 an durchnummerierter Objekte

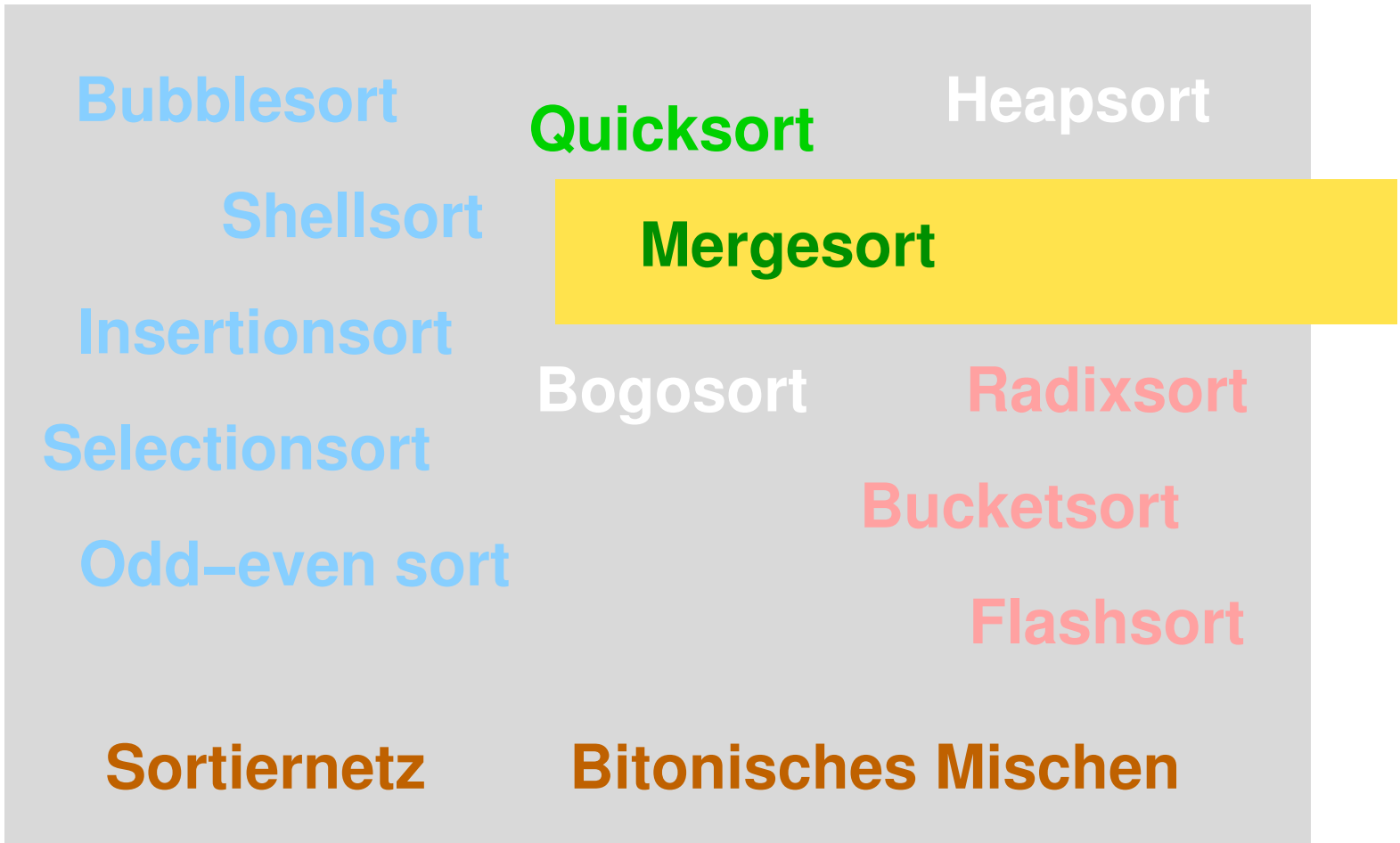
Bucketsort – Vorteile und Einsatzgebiete

- äußerst schnelles Sortierverfahren, das zudem keine Elementvergleiche benötigt
- geeignet für große Datenmengen, wenn die Schlüssel durch natürliche Zahlen dargestellt sind und die Schlüsselwerte eng beieinander liegen
- stets lineares Verhalten $O(n + m)$
- Laufzeitverhalten im Best Case und Worst Case identisch
- intuitiver, kurzer, gut überschaubarer und compilerfreundlicher Quelltext
- vorteilhaft für große Anzahlen von 0 an durchnummerierter Objekte

Nachteile: hoher Hilfsspeicherverbrauch, da oft viele Buckets ungenutzt. Eingeschränkt auf Sortierung natürlicher Zahlen

Mergesort

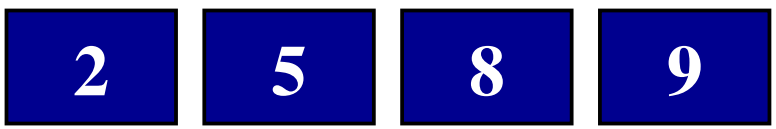
Sortieren durch ordnendes Mischen wachsender Teilfolgen



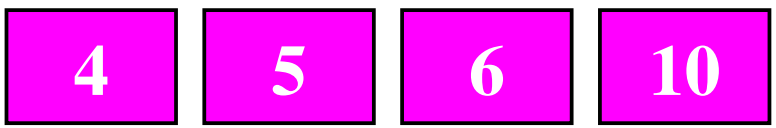
Mischen als Sortieridee von Mergesort

Zwei vorsortierte Teilfelder werden durch *Mischen* zu einem insgesamt sortierten Feld im „Reißverschlussverfahren“

Beispiel



vorsortiertes Teilfeld

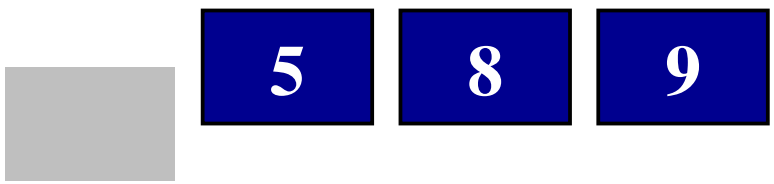


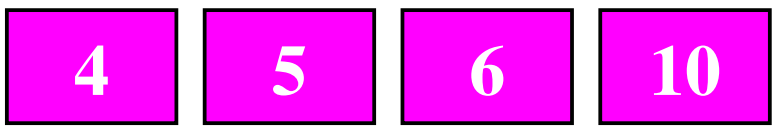
vorsortiertes Teilfeld

Mischen als Sortieridee von Mergesort

Zwei vorsortierte Teilfelder werden durch *Mischen* zu einem insgesamt sortierten Feld im „Reißverschlussverfahren“

Beispiel





Mischen als Sortieridee von Mergesort

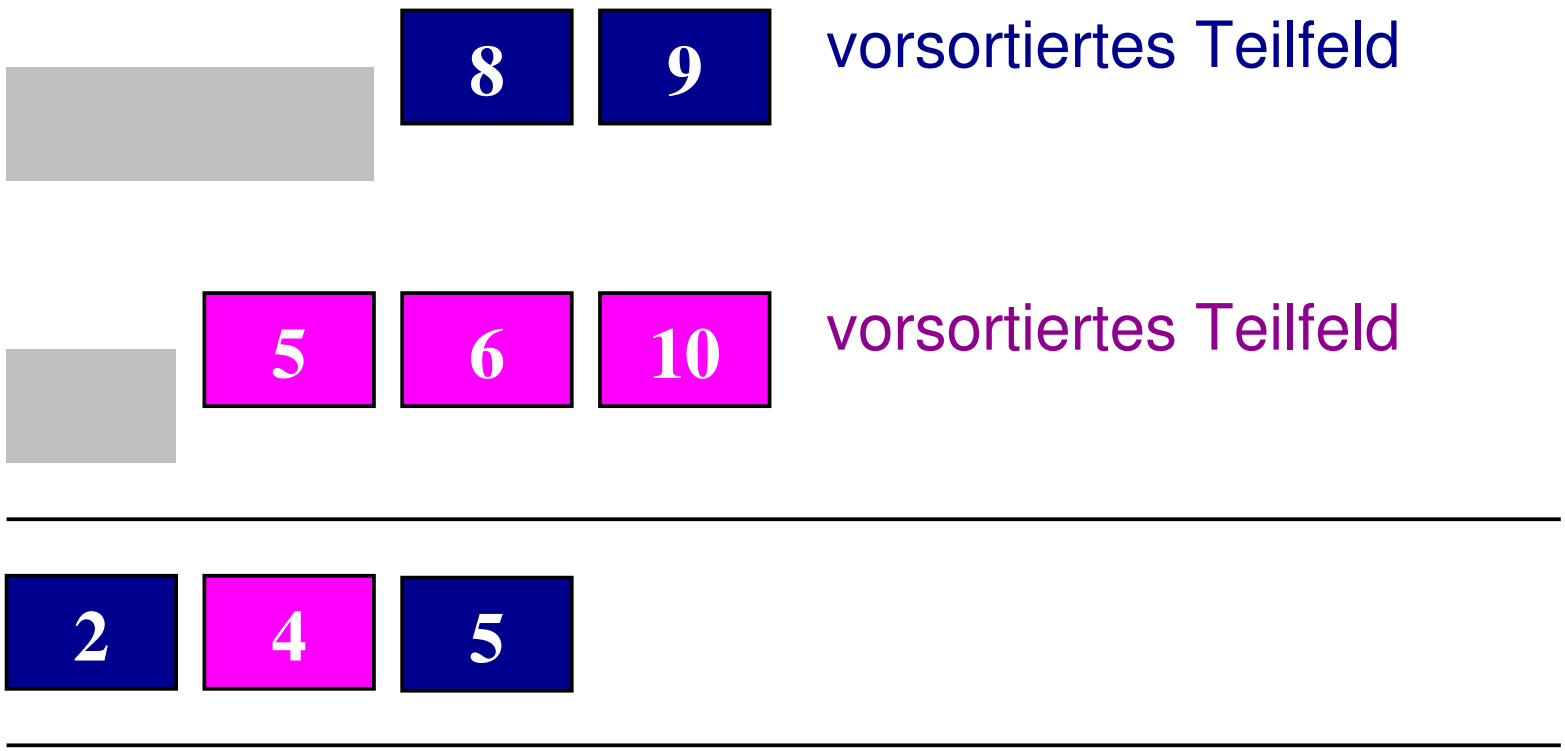
Zwei vorsortierte Teilfelder werden durch *Mischen* zu einem insgesamt sortierten Feld im „Reißverschlussverfahren“

Beispiel

Mischen als Sortieridee von Mergesort

Zwei vorsortierte Teilfelder werden durch *Mischen* zu einem insgesamt sortierten Feld im „Reißverschlussverfahren“

Beispiel



Mischen als Sortieridee von Mergesort

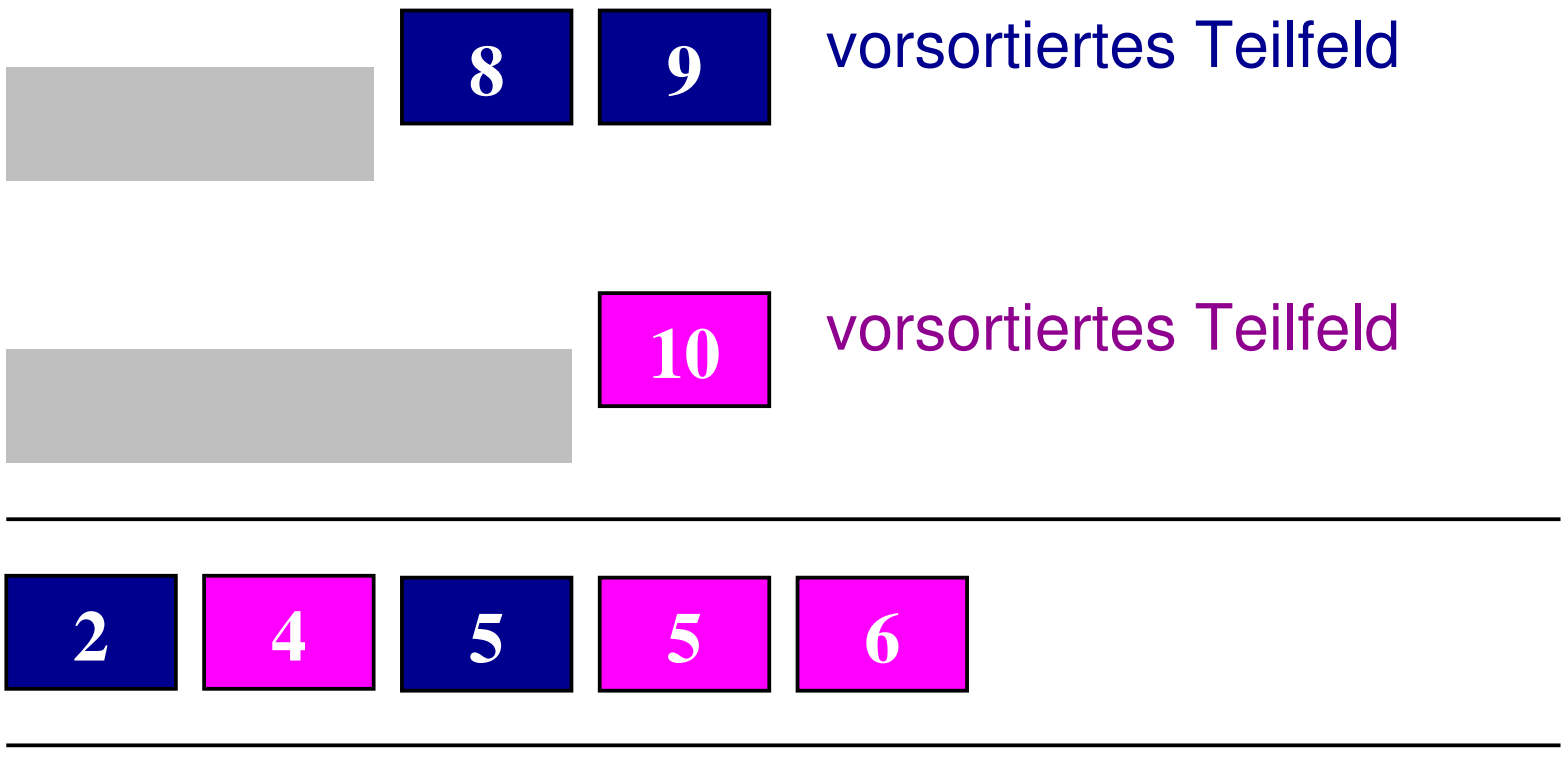
Zwei vorsortierte Teilfelder werden durch *Mischen* zu einem insgesamt sortierten Feld im „Reißverschlussverfahren“

Beispiel

Mischen als Sortieridee von Mergesort

Zwei vorsortierte Teilfelder werden durch *Mischen* zu einem insgesamt sortierten Feld im „Reißverschlussverfahren“

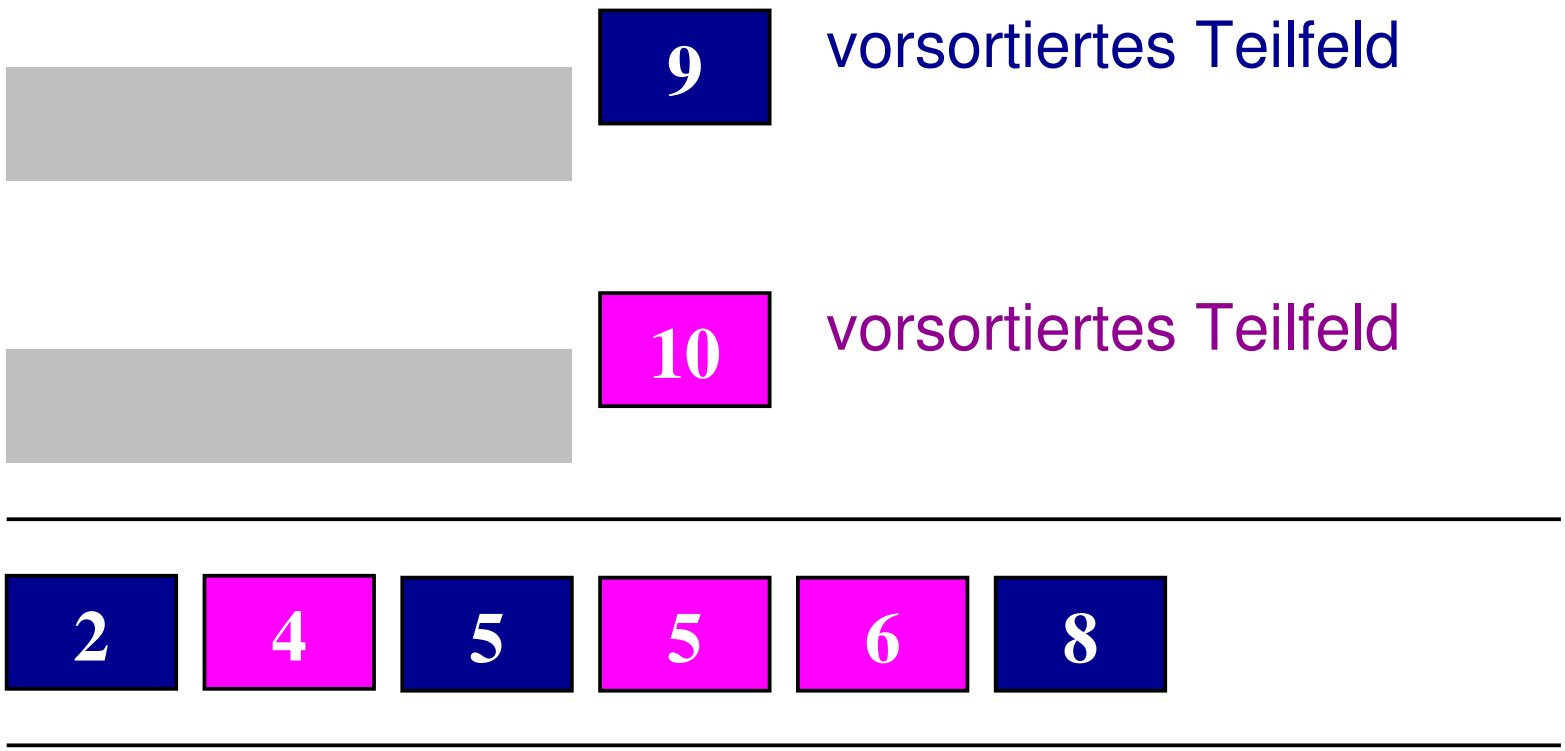
Beispiel



Mischen als Sortieridee von Mergesort

Zwei vorsortierte Teilfelder werden durch *Mischen* zu einem insgesamt sortierten Feld im „Reißverschlussverfahren“

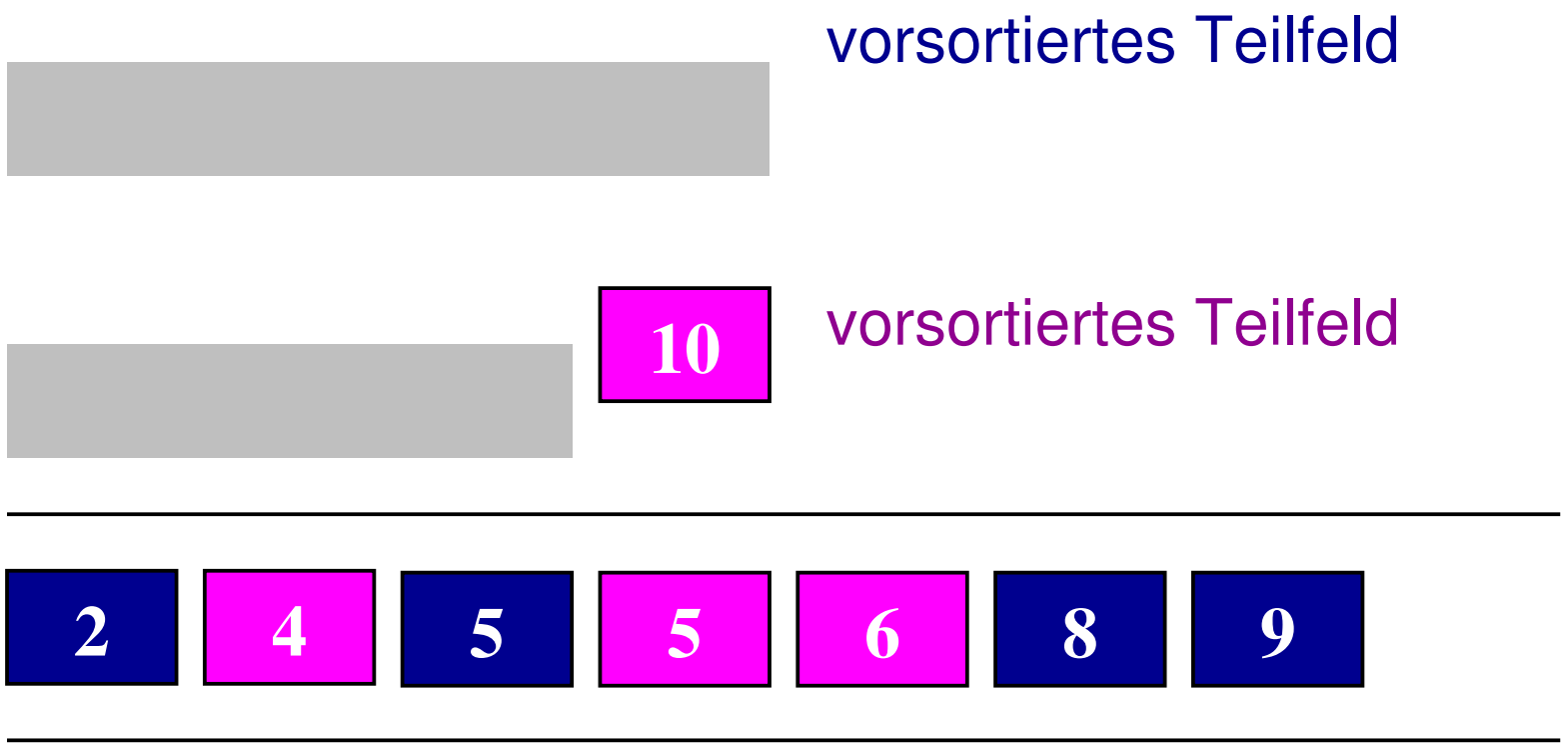
Beispiel



Mischen als Sortieridee von Mergesort

Zwei vorsortierte Teilfelder werden durch *Mischen* zu einem insgesamt sortierten Feld im „Reißverschlussverfahren“

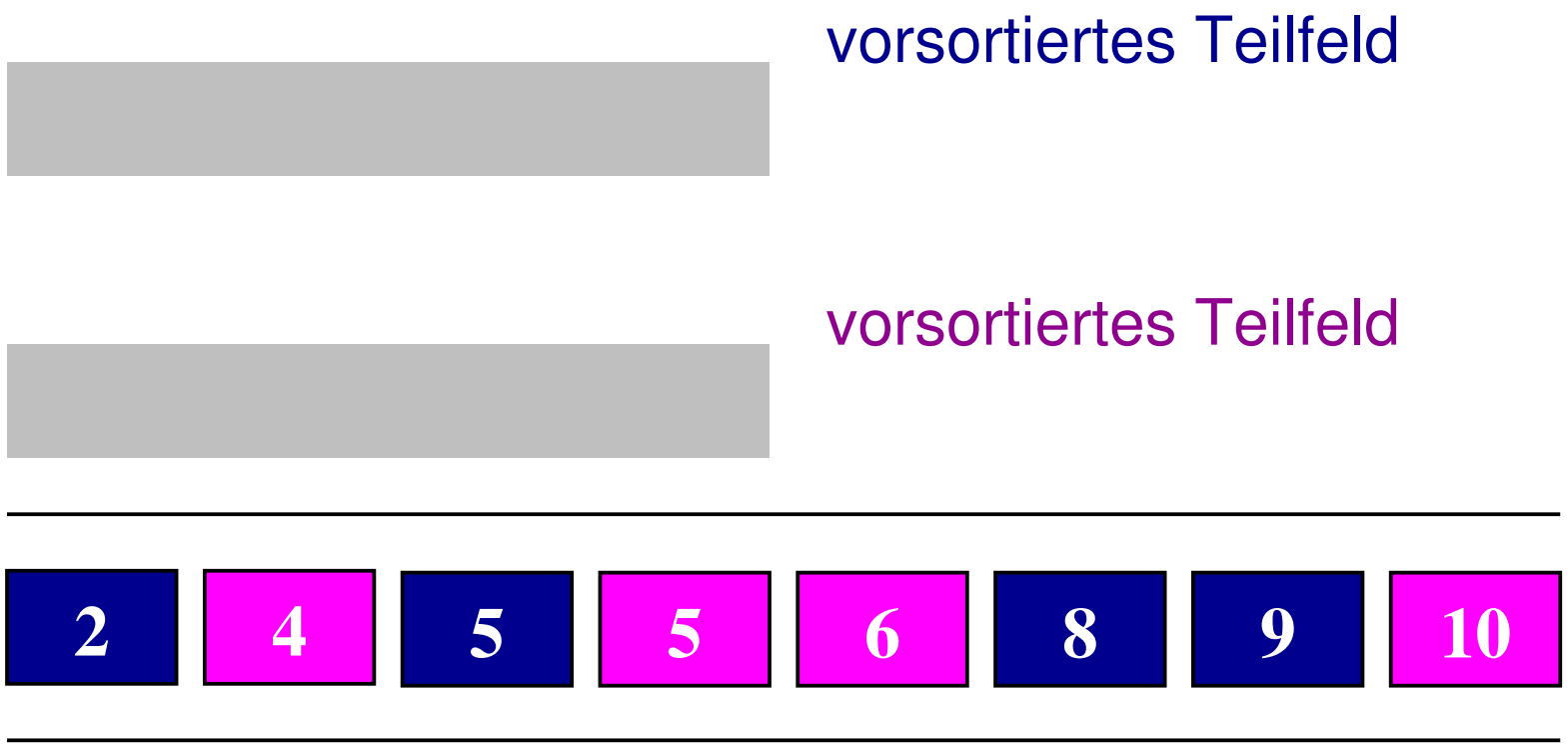
Beispiel



Mischen als Sortieridee von Mergesort

Zwei vorsortierte Teilfelder werden durch *Mischen* zu einem insgesamt sortierten Feld im „Reißverschlussverfahren“

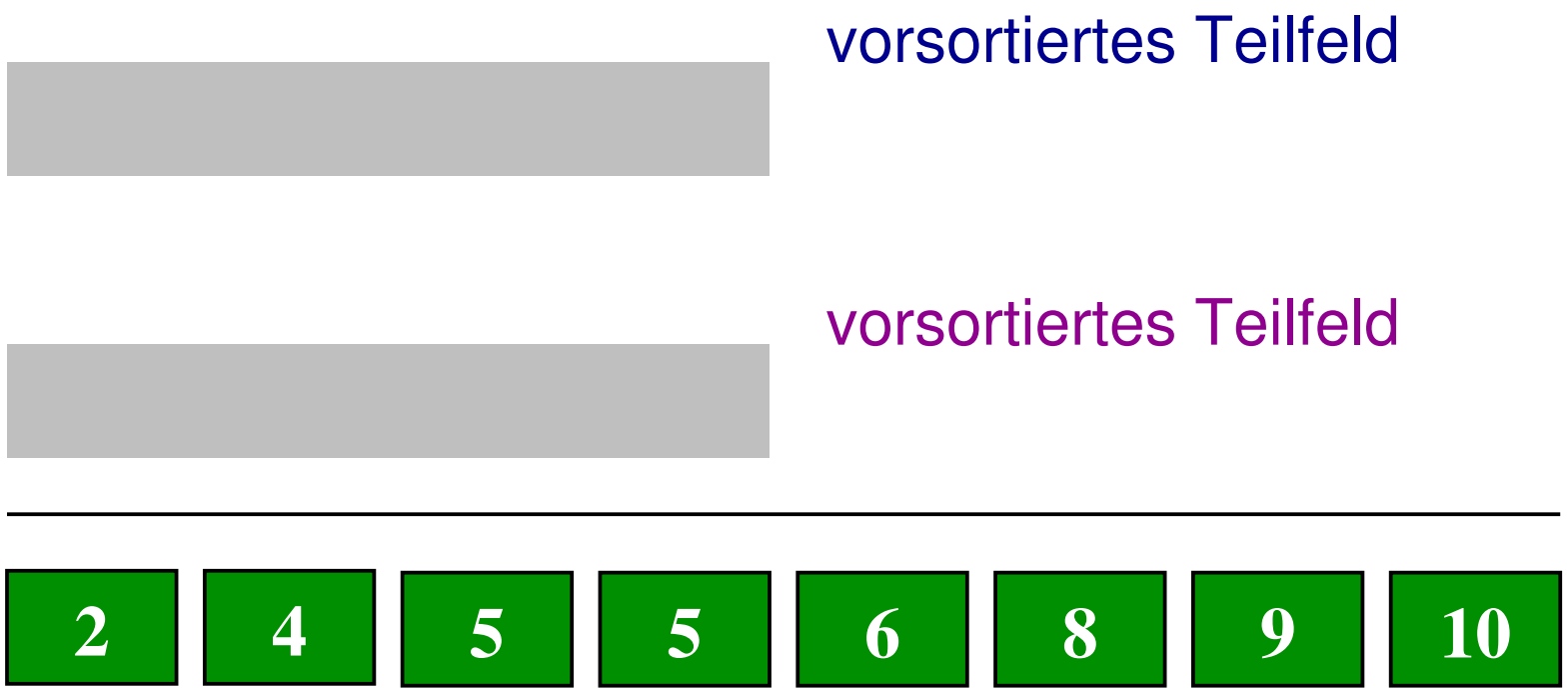
Beispiel



Mischen als Sortieridee von Mergesort

Zwei vorsortierte Teilfelder werden durch *Mischen* zu einem insgesamt sortierten Feld im „Reißverschlussverfahren“

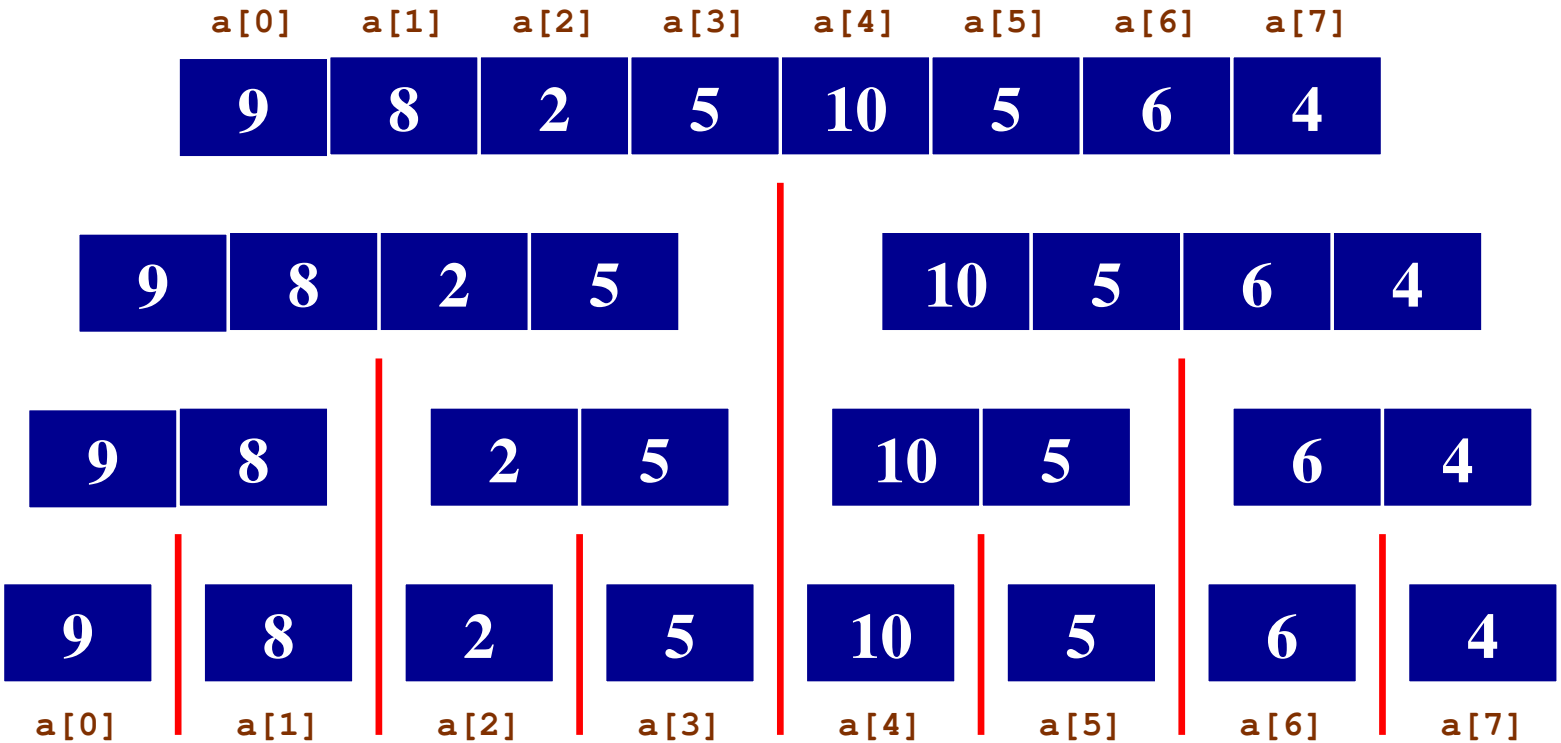
Beispiel



Mischergebnis als sortiertes Feld, das beide Teilfelder vereinigt

Mergesort – rekursives Sortieren

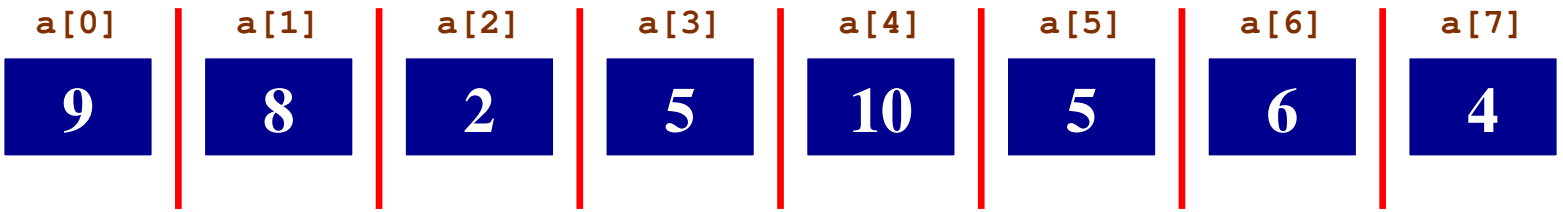
Zu sortierendes Feldes zuerst schrittweise immer weiter teilen, bis ausschließlich einelementige Teilfelder entstanden sind.



Anschließend im rekursiven Aufstieg Mischen von je zwei benachbarten vorsortierten Teilfeldern, bis Gesamtfeld sortiert

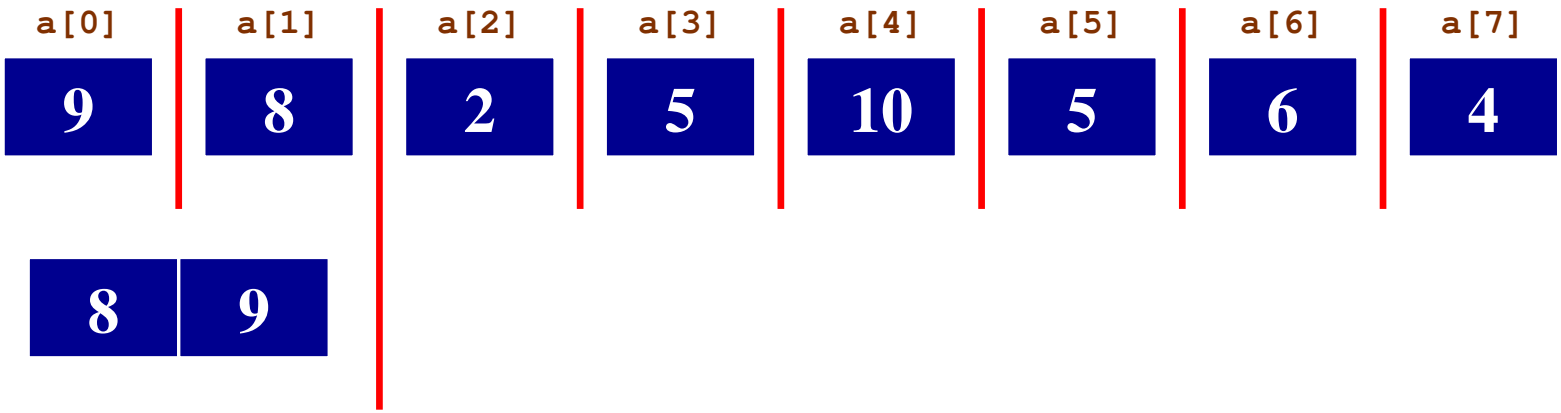
Mergesort – rekursives Sortieren

Mischen von je zwei benachbarten vorsortierten Teilfeldern, bis Gesamtfeld sortiert



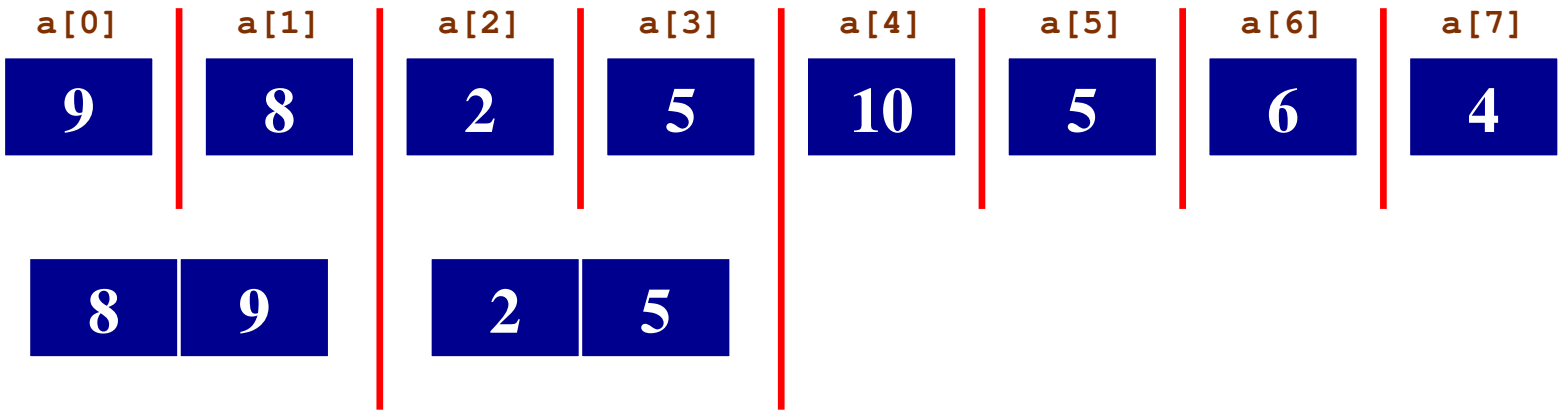
Mergesort – rekursives Sortieren

Mischen von je zwei benachbarten vorsortierten Teilfeldern, bis Gesamtfeld sortiert



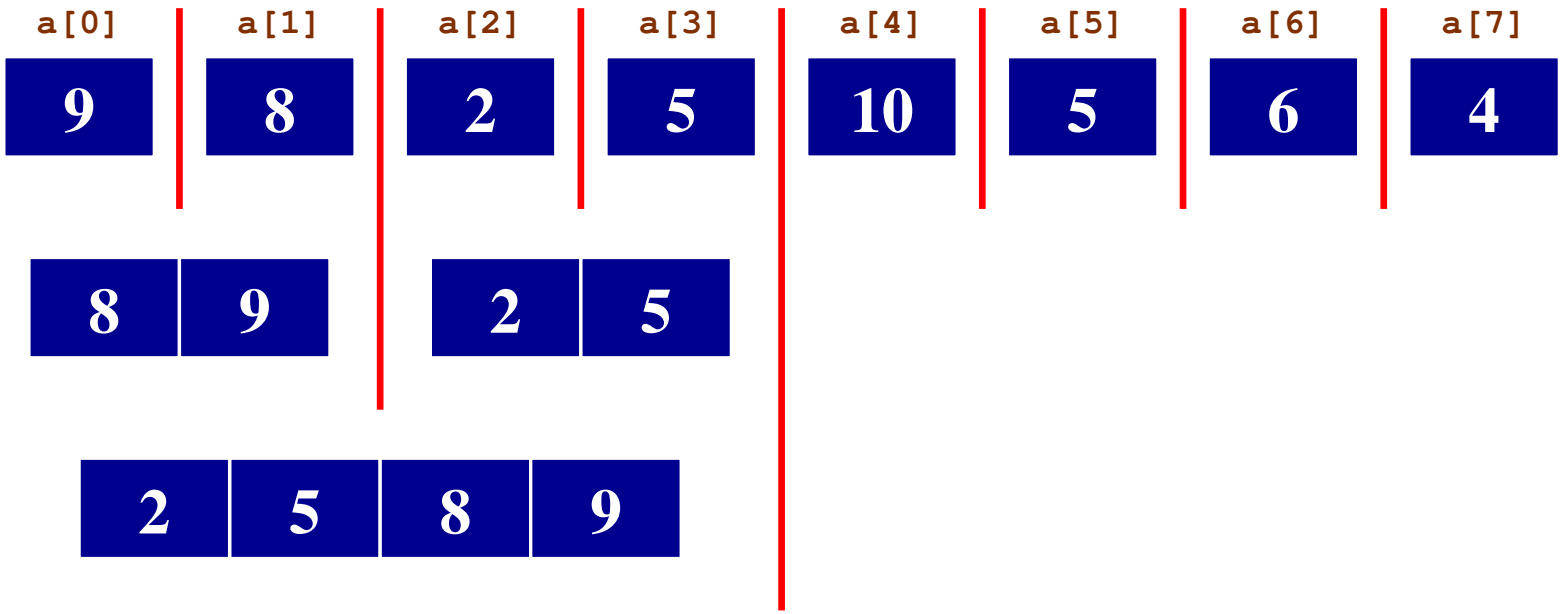
Mergesort – rekursives Sortieren

Mischen von je zwei benachbarten vorsortierten Teilfeldern, bis Gesamtfeld sortiert



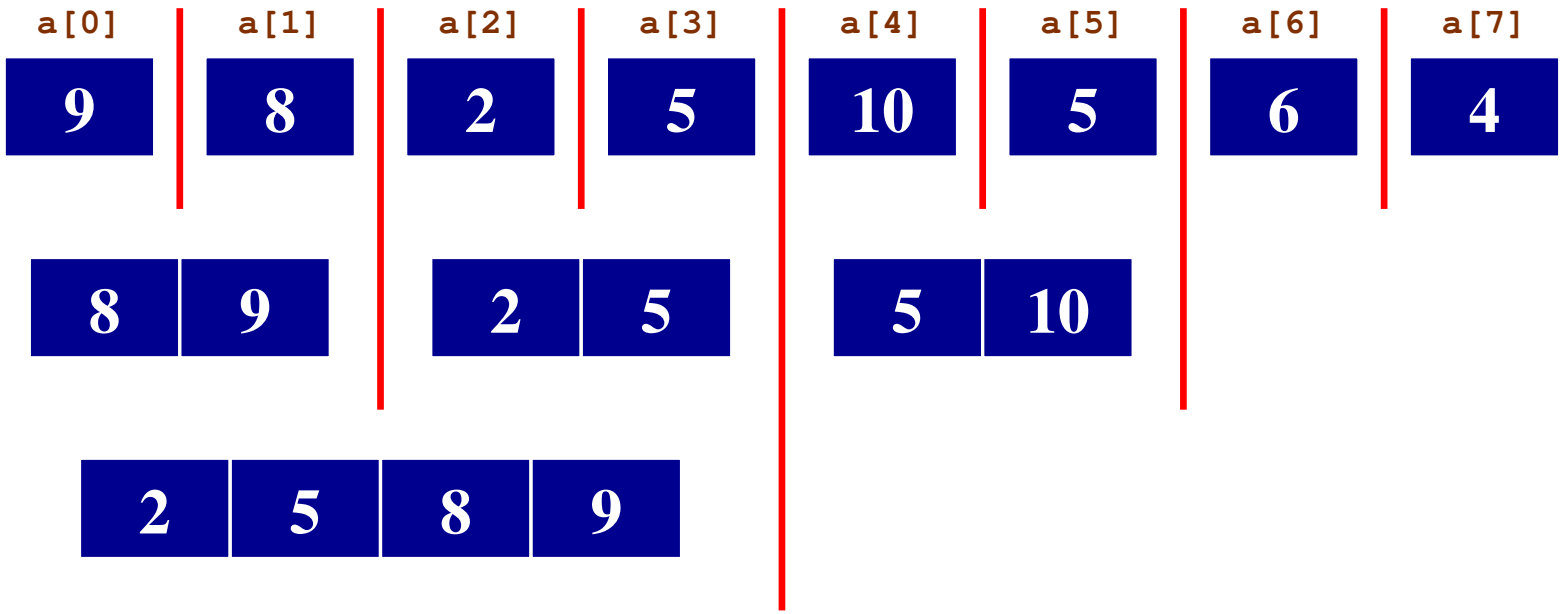
Mergesort – rekursives Sortieren

Mischen von je zwei benachbarten vorsortierten Teilfeldern, bis Gesamtfeld sortiert



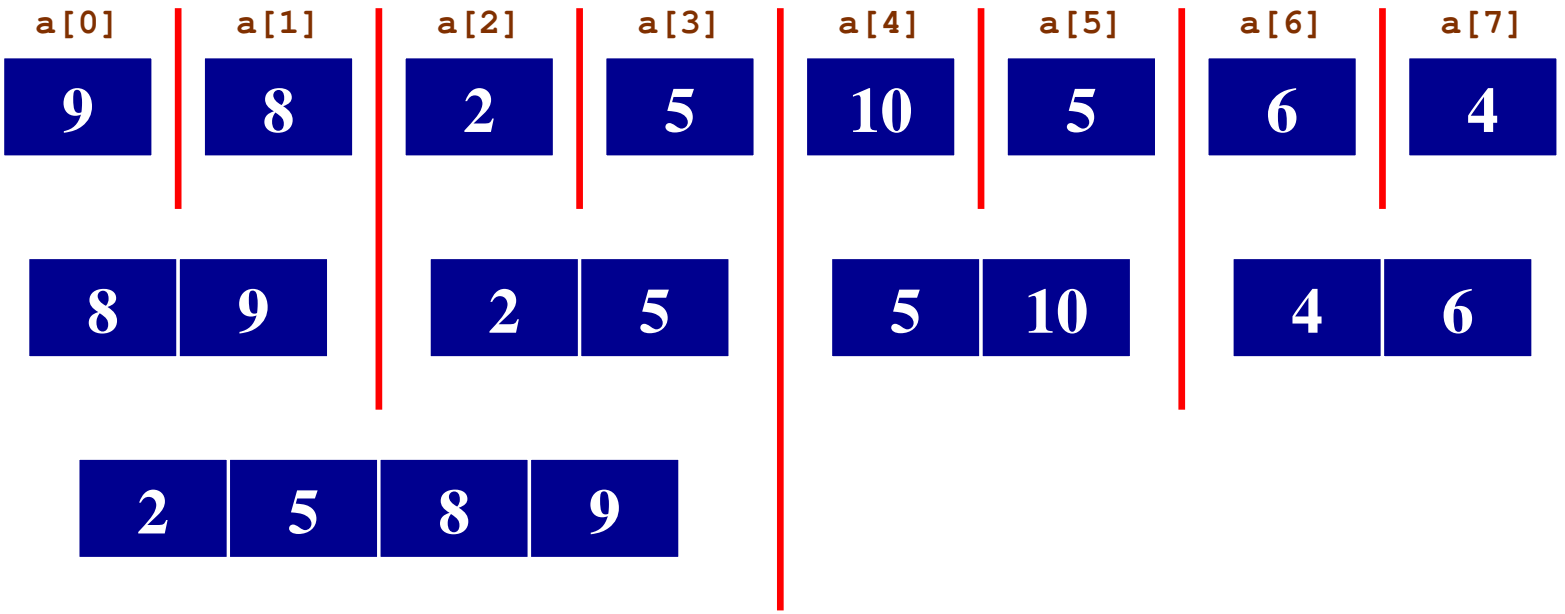
Mergesort – rekursives Sortieren

Mischen von je zwei benachbarten vorsortierten Teilfeldern, bis Gesamtfeld sortiert



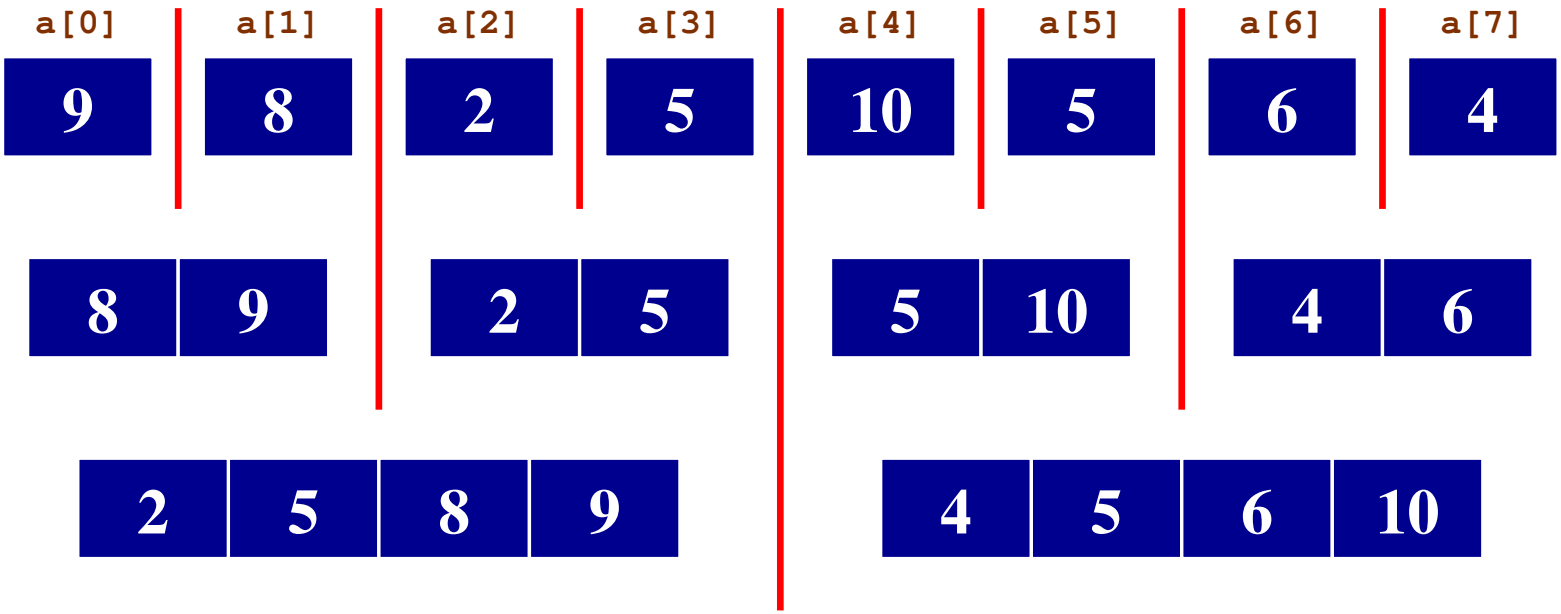
Mergesort – rekursives Sortieren

Mischen von je zwei benachbarten vorsortierten Teilfeldern, bis Gesamtfeld sortiert



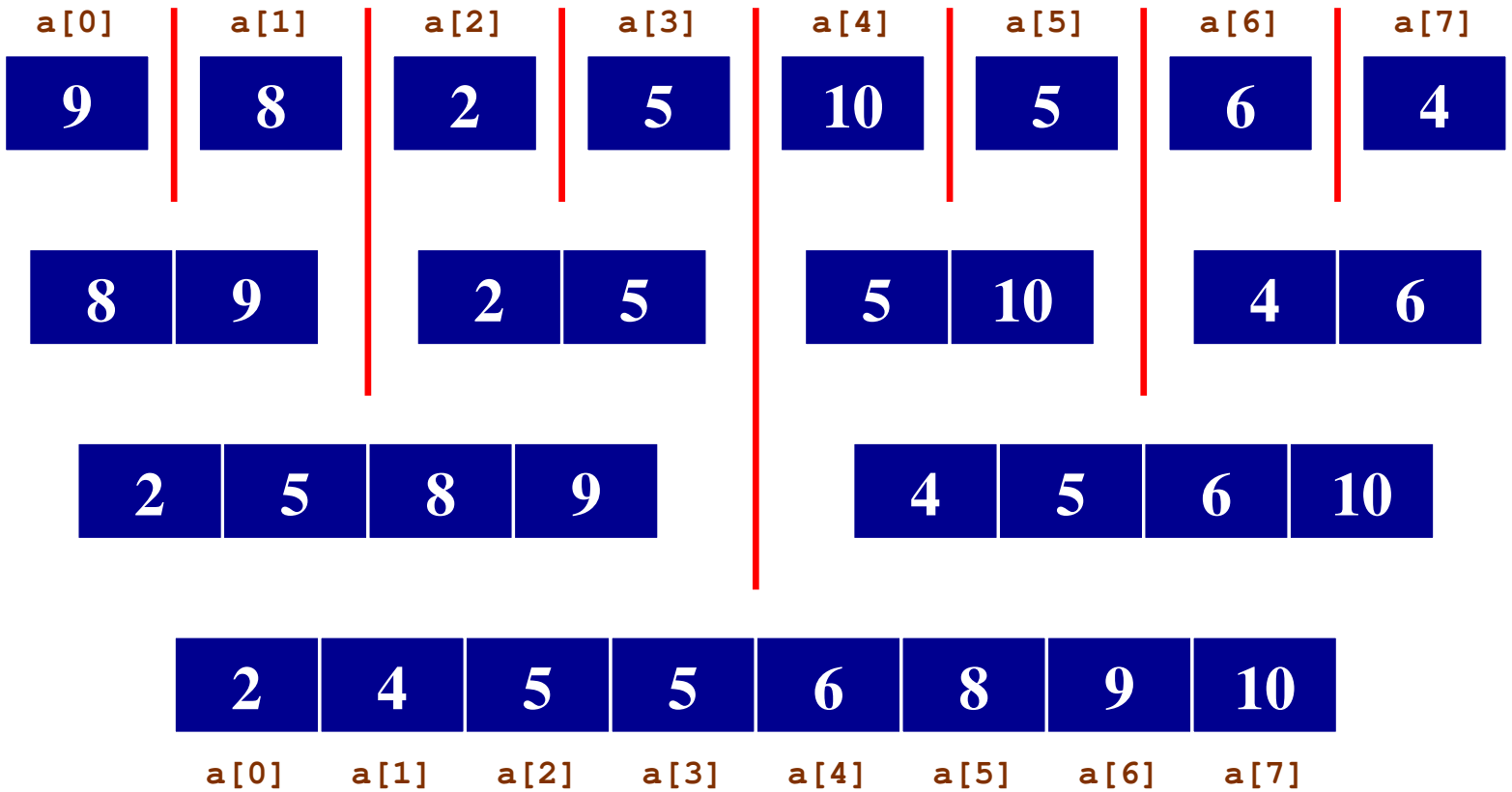
Mergesort – rekursives Sortieren

Mischen von je zwei benachbarten vorsortierten Teilfeldern, bis Gesamtfeld sortiert



Mergesort – rekursives Sortieren

Mischen von je zwei benachbarten vorsortierten Teilfeldern, bis Gesamtfeld sortiert



Mergesort ausprogrammiert

mergesort.c – Funktion merge zum Mischen zweier Teilfelder

```c
#include <stdio.h>

#define  N  8  //Anzahl zu sortierender Elemente


void merge(int l, int q, int r, long a[])
//Mische Teilfeld a[l]...a[q] mit Teilfeld a[q+1]...a[r]
{
  int h[N]; //Hilfsfeld zum Zwischenspeichern beim Mischen
  int i, j, k;

  for (i = l; i <= q; i++) {
    h[i] = a[i];              //Teilfeld a[l]...a[q] ins Hilfsfeld kopieren
  }
  for (j = q + 1; j <= r; j++) {
    h[r + q + 1 - j] = a[j];  //Teilfeld a[q+1]...a[r] revers ins Hilfsfeld kopieren
  }
  i = l;
  j = r;
  for (k = l; k <= r; k++) {
    if (h[i] <= h[j]) {       //Fuehrende Elemente in den zu mischenden Teilfeldern
      a[k] = h[i];            //vergleichen und kleineres davon in a zurueckschreiben
      i++;
    } else {
      a[k] = h[j];
      j--;
    }
  }
  return;
}
```

Mergesort ausprogrammiert

mergesort.c – Funktion sort zur rekursiven Ablaufsteuerung und `main`-Funktion

```
void sort(int l, int r, long a[])
//rekursive Steuerung des gesamten Sortierablaufs aus Teilen und Mischen
{
  int q;

  //Betrachte Teilfeld a[l]...a[r]
  if (l < r) {
    q = (l + r) / 2; //Teile das Teilfeld erneut, wenn es mehr als ein Element besitzt
    sort(l, q, a);      //sortiere linke Haelfte des geteilten Feldes
    sort(q + 1, r, a);  //sortiere rechte Haelfte des geteilten Feldes
    merge(l, q, r, a);  //mische die beiden sortierten Haelften und fusioniere sie
  }
  return;
}


int main(void)
{
  long datenfeld[N] = {9, 8, 2, 5, 10, 5, 6, 4};
  int i;

  sort(0, N-1, datenfeld); //gesamtes Datenfeld zum Sortieren uebergeben
  for (i = 0; i < N; i++)
  {
    printf("%ld  ", datenfeld[i]);
  }
  printf("\n");
  return 0;
}
```

```
2  4  5  5  6  8  9  10
```

Steckbrief Mergesort zum aufsteigenden Sortieren

Algorithmus . . . . . rekursiv, vergleichsbasiert

Eigenschaften . . . . . ex-situ (nicht in-place), stabil

Steckbrief Mergesort zum aufsteigenden Sortieren

Algorithmus . . . . . rekursiv, vergleichsbasiert

Eigenschaften . . . . . ex-situ (nicht in-place), stabil

Worst Case . Worst Case und Best Case kaum unterscheidbar

Best Case . . . . jeweils beliebige Permutation der Feldelemente

---

Steckbrief Mergesort zum aufsteigenden Sortieren

Algorithmus . . . . . rekursiv, vergleichsbasiert

Eigenschaften . . . . . ex-situ (nicht in-place), stabil

Worst Case . Worst Case und Best Case kaum unterscheidbar

Best Case . . . . jeweils beliebige Permutation der Feldelemente

---

Problemgröße n . . . . . Anzahl zu sortierender Feldelemente

Anzahl Vergleiche . . . . . $\approx n \cdot \mathrm{ld}(n)$

Anzahl Elementverschiebungen . . . . . $\approx n \cdot \mathrm{ld}(n)$

Steckbrief Mergesort zum aufsteigenden Sortieren

Algorithmus . . . . . rekursiv, vergleichsbasiert

Eigenschaften . . . . . ex-situ (nicht in-place), stabil

Worst Case . Worst Case und Best Case kaum unterscheidbar

Best Case . . . . jeweils beliebige Permutation der Feldelemente

---

Problemgröße n . . . . . Anzahl zu sortierender Feldelemente

Anzahl Vergleiche . . . . . $\approx n \cdot \mathrm{ld}(n)$

Anzahl Elementverschiebungen . . . . . $\approx n \cdot \mathrm{ld}(n)$

Zeitkomplexität im Worst Case . . . . . $O(n \cdot \log(n))$

Mergesort – Vorteile und Einsatzgebiete

- schnelles vergleichsbasiertes Sortierverfahren
- geeignet für große Datenmengen
- Laufzeitverhalten im Best Case und Worst Case nahezu identisch und mit $O(n \cdot \log(n))$ nahezu am Optimum $O(\log(n!))$ für sequentielle vergleichsbasierte Verfahren
- gut auf linearen Listen anstelle Feld implementierbar
- lässt sich gut parallelisieren
- durch rekursiven Ansatz auch zur Implementierung in funktionalen Programmiersprachen geeignet

Mergesort – Vorteile und Einsatzgebiete

- schnelles vergleichsbasiertes Sortierverfahren
- geeignet für große Datenmengen
- Laufzeitverhalten im Best Case und Worst Case nahezu identisch und mit $O(n \cdot \log(n))$ nahezu am Optimum $O(\log(n!))$ für sequentielle vergleichsbasierte Verfahren
- gut auf linearen Listen anstelle Feld implementierbar
- lässt sich gut parallelisieren
- durch rekursiven Ansatz auch zur Implementierung in funktionalen Programmiersprachen geeignet

Mergesort – Vorteile und Einsatzgebiete

- schnelles vergleichsbasiertes Sortierverfahren
- geeignet für große Datenmengen
- Laufzeitverhalten im Best Case und Worst Case nahezu identisch und mit $O(n \cdot \log(n))$ nahezu am Optimum $O(\log(n!))$ für sequentielle vergleichsbasierte Verfahren
- gut auf linearen Listen anstelle Feld implementierbar
- lässt sich gut parallelisieren
- durch rekursiven Ansatz auch zur Implementierung in funktionalen Programmiersprachen geeignet

Mergesort – Vorteile und Einsatzgebiete

- schnelles vergleichsbasiertes Sortierverfahren
- geeignet für große Datenmengen
- Laufzeitverhalten im Best Case und Worst Case nahezu identisch und mit $O(n \cdot \log(n))$ nahezu am Optimum $O(\log(n!))$ für sequentielle vergleichsbasierte Verfahren
- gut auf linearen Listen anstelle Feld implementierbar
- lässt sich gut parallelisieren
- durch rekursiven Ansatz auch zur Implementierung in funktionalen Programmiersprachen geeignet

Mergesort – Vorteile und Einsatzgebiete

- schnelles vergleichsbasiertes Sortierverfahren
- geeignet für große Datenmengen
- Laufzeitverhalten im Best Case und Worst Case nahezu identisch und mit $O(n \cdot \log(n))$ nahezu am Optimum $O(\log(n!))$ für sequentielle vergleichsbasierte Verfahren
- gut auf linearen Listen anstelle Feld implementierbar
- lässt sich gut parallelisieren
- durch rekursiven Ansatz auch zur Implementierung in funktionalen Programmiersprachen geeignet

Mergesort – Vorteile und Einsatzgebiete

- schnelles vergleichsbasiertes Sortierverfahren
- geeignet für große Datenmengen
- Laufzeitverhalten im Best Case und Worst Case nahezu identisch und mit $O(n \cdot \log(n))$ nahezu am Optimum $O(\log(n!))$ für sequentielle vergleichsbasierte Verfahren
- gut auf linearen Listen anstelle Feld implementierbar
- lässt sich gut parallelisieren
- durch rekursiven Ansatz auch zur Implementierung in funktionalen Programmiersprachen geeignet

Mergesort – Vorteile und Einsatzgebiete

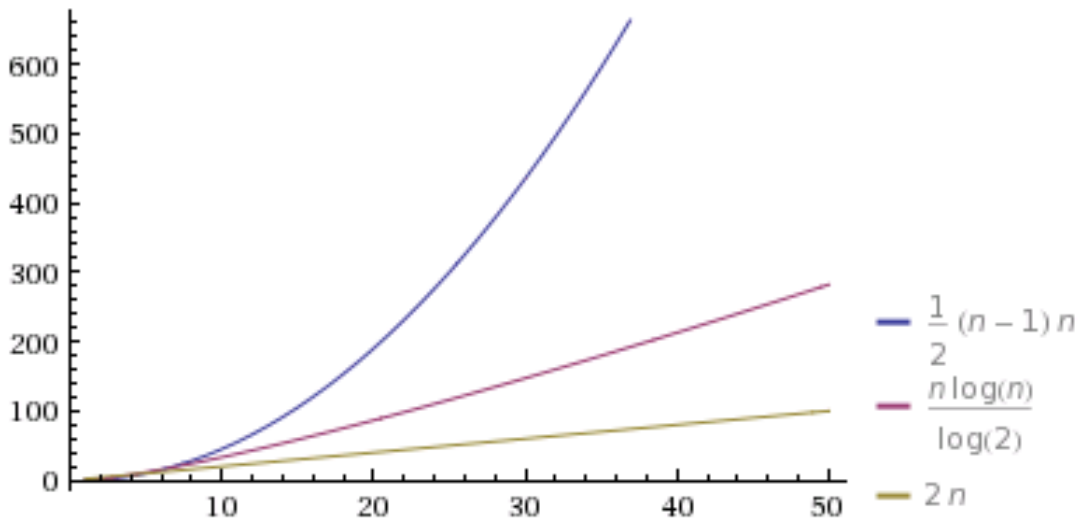
- schnelles vergleichsbasiertes Sortierverfahren
- geeignet für große Datenmengen
- Laufzeitverhalten im Best Case und Worst Case nahezu identisch und mit $O(n \cdot \log(n))$ nahezu am Optimum $O(\log(n!))$ für sequentielle vergleichsbasierte Verfahren
- gut auf linearen Listen anstelle Feld implementierbar
- lässt sich gut parallelisieren
- durch rekursiven Ansatz auch zur Implementierung in funktionalen Programmiersprachen geeignet

Nachteile: hoher Hilfsspeicherverbrauch bedingt durch Rekursion

Vergleich der Sortierverfahren

	Selectionsort	Insertionsort	Bucketsort	Mergesort
	iterativ	iterativ	iterativ	rekursiv
	stabil*	stabil	stabil	stabil
	in-place	in-place	ex-situ	ex-situ
	vergleichsbasiert	vergleichsbasiert	fachverteilend	vergleichsbasiert
Zeitbedarf	$O(n^2)$	$O(n^2)$	$O(n+m)$	$O(n \cdot \log(n))$
Speicherbed.	niedrig	niedrig	sehr hoch	hoch

* hier gezeigte Version, allgemein nicht stabil



Zeitbedarf in Abhängigkeit von Problemgröße n im Vergleich