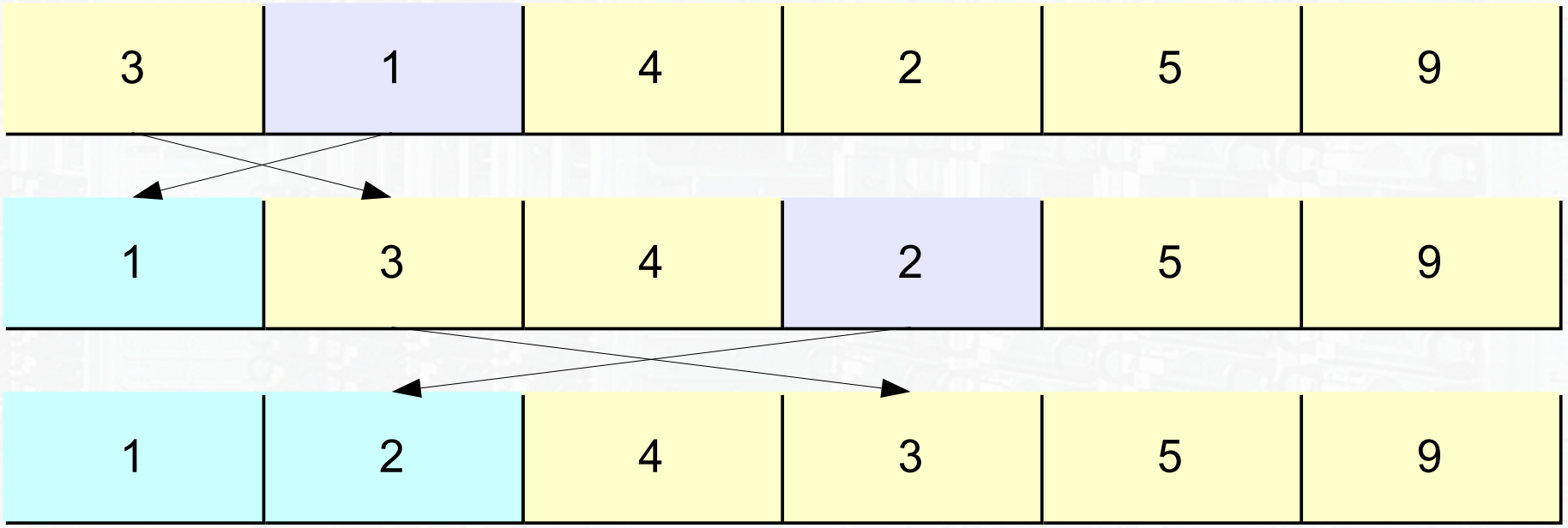
Sortieralgorithmen

Selection Sort

- intuitivster Suchalgorithmus
- In jedem Schritt wird das kleinste Element im noch unsortierten Array gesucht und ans Ende des bisher sortierten Teilarrays gehangen



Stephan Alt
stephan.alt@rwth-aachen.de

Sortieralgorithmen

Selection Sort

- Laufzeit: $O(n^2)$ in allen Fällen
- Benutzung zweier ineinander verschachtelten Schleifen, dabei gibt die äußere Schleife den aktuell linken Rand des noch unsortierten Arrays an, während die innere Schleife im noch unsortierten Array das Minimum sucht

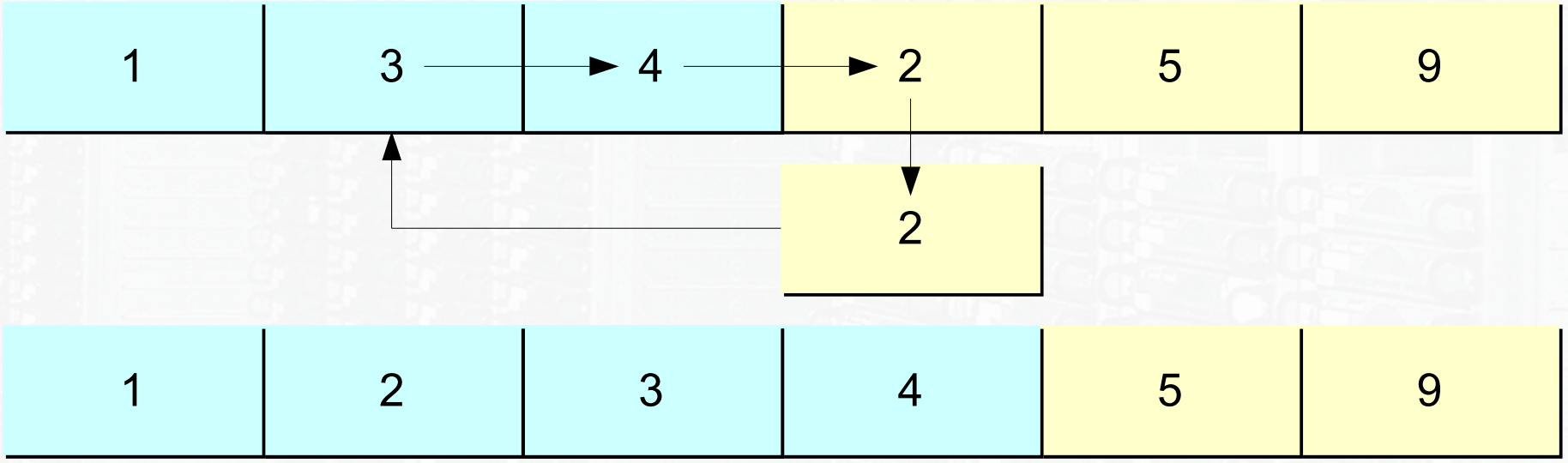
```
void selection_sort(int a[], int l, int r)   // a wird in den Grenzen l und r sortiert
{
      int i, j, min;                  // min speichert Index des akt. Minimums
      for (i = l; i < r; i++) {       // i ist akt. linke Grenze des unsortierten Arrays
            min = i;                  // min initialisieren
            for (j = i+1; j <= r; j++) {
                  if (a[j] < a[min]) min = j;  // min im unsortierten Array finden
            }
            exchange (&a[i], &a[min]);   // call by reference nötig
      }
}
```

Stephan Alt
stephan.alt@rwth-aachen.de

Sortieralgorithmen

Insertion Sort

- Sortierung durch Einrücken und Einfügen
- Beginnend mit dem zweiten Arrayelement wird das Element zwischengespeichert und danach jedes Element kleiner als das aktuell zwischengespeicherte um eine Stelle nach rechts geschoben. Anschließend wird das zwischengespeicherte Element wieder eingefügt.



Stephan Alt
stephan.alt@rwth-aachen.de

Sortieralgorithmen

Insertion Sort

- Laufzeit: $O(n^2)$ in allen Fällen
- Im Gegensatz zu Selection Sort fügt Insertion Sort nicht das kleinste Element des noch unsortierten Arrays ein, sondern das nächste, dafür muss aber die Position dieses Elements im bereits sortierten Array gesucht werden!

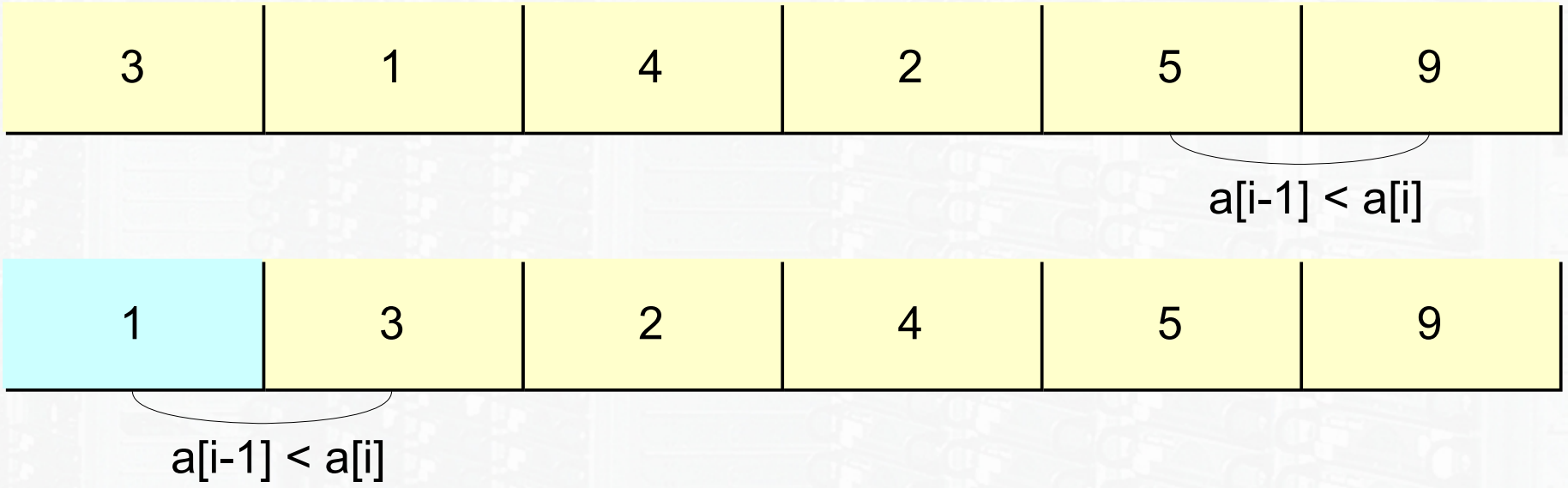
```
void insertion_sort(int a[], int l, int r) // a wird in den Grenzen l und r sortiert
{
    int i, j, v;                           // v speichert das einzufügende Element
    for (i = l+1; i <= r; i++) {           // i ist Index des einzufügenden Elements
        v = a[i]                           // v initialisieren
        for (j = i-1; j >= l; j--) {       // Durchsuchen des sortierten Teilarrays
            if (v < a[j]) a[j+1] = a[j];   // Elemente > v nach rechts schieben
            else break;
        }
        a[j+1] = v;    // Einfügen des zwischengespeicherten Elements
    }
}
```

Stephan Alt
stephan.alt@rwth-aachen.de

Sortieralgorithmen

Bubble Sort

- vergleicht lediglich benachbarte Arrayelemente
- beginnend mit dem rechtesten Paar wird nach jedem Vergleich und ggf. Tausch der Elemente das Paar eine Stelle weiter links verglichen. So durchläuft man das Array n-1 Mal, wobei man mit jedem Durchlauf eine Stelle früher aufhören kann, da sich am linken Rand das sortierte Array langsam aufbaut



Sortieralgorithmen

Bubble Sort

- Laufzeit: $O(n^2)$ im Durchschnitt (nach dieser Implementierung immer)
- Benutzung zweier ineinander verschachtelten Schleifen, dabei gibt die äußere Schleife den aktuell linken Rand des noch unsortierten Arrays an, während die innere Schleife die benachbarten Elemente – falls nötig – tauscht.

```
void bubble_sort(int a[], int l, int r)    // a wird in den Grenzen l und r sortiert
{
      int i, j;
      for (i = l; i < r; i++) {              // i = linker Rand des unsortierten Arrays
            for (j =r; j > i; j--) {         // Durchlauf des unsortierten Teilarrays
                  if (a[j-1] > a[j])         // falls die Nachbarn zueinander falsch stehen
                        exchange(&a[j-1], &a[j]);      // call by reference nötig
            }
      }
}
```

Stephan Alt
stephan.alt@rwth-aachen.de

Sortieralgorithmen

Quick Sort

- arbeitet nach dem Divide and Conquer Prinzip
- rekursive Implementierung, wobei immer kleinere Arrays sortiert werden sollen (Arrays der Länge 1 sind immer sortiert → Abbruch)
- Problematik: die Teilarrays müssen die richtigen Elemente enthalten, damit später das Gesamtarray auch sortiert ist (gewünschte Vorsortierung: Teilarray bestehend aus Elementen kleiner Trennelement → das Trennelement → Teilarray bestehend aus Elementen größer Trennelement
- Lösung: Partitionieren unseres Arrays, bevor die Teilarrays rekursiv aufgerufen werden → partition-Funktion

Sortieralgorithmen

Die Partition Funktion

- Aufruf der Funktion int partition(int linkesEnde, int rechtesEnde)
- i = linkesEnde, k = rechtesEnde, j = rechtesEnde - 1
- i läuft nach rechts, bis a[i] > a[k]
- j läuft nach links, bis a[j] < a[k]
- dann Tausch von i und j
- Abbruch falls i >= j: Tausche in diesem Fall i und k, wobei danach k das Array in zwei Hälften trennt mit

  $a[x] < a[k] \; \forall \; x < k$ und $a[x] > a[k] \; \forall \; x > k$

Sortieralgorithmen

Die Partition Funktion

```
int partition(int a[], int l, int r)      // a wird in den Grenzen l und r partitioniert
{
      int i = l, j = r-1, k = r;
      while (1) {                                    // Durchlauf bis break
            while (a[i] <= a[k] && i < r) i++;   // i darf nicht aus dem Array laufen
            while (a[j] >= a[k] && j>=l) j--;     // j darf bis -1 laufen
            if (i >= j) break;                          // aneinander vorbeigelaufen
            else exchange (&a[i], &a[j]);       // call by reference
      }
      exchange(&a[i], &a[k]);  // a[i] und a[k] werden in jedem Fall getauscht
      return i;                          // gebe Position des Trennelements zurück
}
```

Sortieralgorithmen

Quick Sort

- Laufzeit: O(n*log(n)) im average case ($O(n^2)$ bei vorsortiertem Array)
- Quick Sort ruft partition auf und danach rekursiv Quick Sort auf die beiden Teilarrays, die durch das Trennelement getrennt wurden. Falls das zu sortierende Array die Länge 1 oder kleiner besitzt, können wir abbrechen

```
void quick_sort(int a[], int l, int r)      // a wird in den Grenzen l und r sortiert
{
    if (r <= l) return;             // Abbruchbedingung bei Länge kleiner gleich 1
    int k = partition(a, l, r);     // Partitionieren des Arrays, Position des
                                    // Trennelements wird in k gespeichert
    quick_sort(a, l, k-1);          // linkes Teilarray
    quick_sort(a, k+1, r);          // rechtes Teilarray
}
```

Stephan Alt
stephan.alt@rwth-aachen.de

Sortieralgorithmen

Quick Sort

- Laufzeit: O(n*log(n)) im average case ($O(n^2)$ bei vorsortiertem Array)
- Quick Sort ruft partition auf und danach rekursiv Quick Sort auf die beiden Teilarrays, die durch das Trennelement getrennt wurden. Falls das zu sortierende Array die Länge 1 oder kleiner besitzt, können wir abbrechen

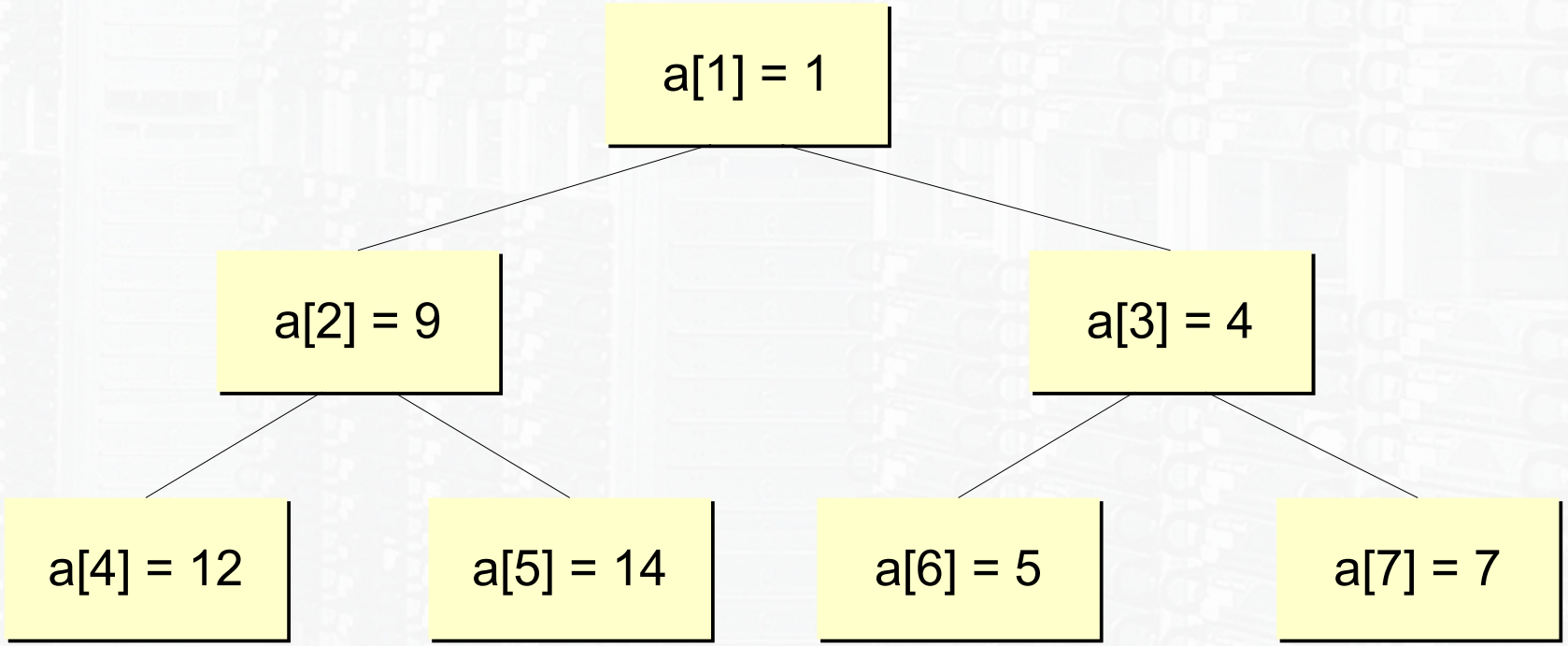
```
void quick_sort(int a[], int l, int r)      // a wird in den Grenzen l und r sortiert
{
    if (r <= l) return;             // Abbruchbedingung bei Länge kleiner gleich 1
    int k = partition(a, l, r);     // Partitionieren des Arrays, Position des
                                    // Trennelements wird in k gespeichert
    quick_sort(a, l, k-1);          // linkes Teilarray
    quick_sort(a, k+1, r);          // rechtes Teilarray
}
```

Stephan Alt
stephan.alt@rwth-aachen.de

Sortieralgorithmen

Heap Sort

- Wir bauen uns einen Baum, in dem wir eine quasi-Sortierung einführen (jedes Element ist kleiner als alle Elemente unter diesem Element)
- In einem solchen Baum ist die Wurzel immer das kleinste Element



Sortieralgorithmen

Die Sink Funktion

- benötigt, um die Heap-Eigenschaft herzustellen und wiederherzustellen
- lässt schwere Elemente einsinken

```
sink(int a[ ], int l, int r) {
        sink = l;
        while(1) {
                suche min von (a[sink], a[sink*2+1] & a[sink*2]);
                wenn min = sink, Abbruch;
                sonst, vertausche a[sink] mit a[min];
                setze sink = min, dort liegt nämlich jetzt das
                        einzusinkende Element
        }
}
```

Sortieralgorithmen

Die Sink Funktion

```
void sink(int a[], int l, int r)          // in a soll l bis maximal zu r einsinken
{
      int son;        // speichert Position des kleinsten Sohns
      while(1) {
            if (2*l > r) break;       // kein Sohn innerhalb der Grenzen vorhanden
            son = 2*l;                // son initialisieren als linken Sohn
            if (2*l+1 <= r) {         // falls noch ein rechter Sohn exisitert
                  if (a[2*l] > a[2*l+1]) son = 2*l+1;
            }                         // ab hier zeigt son auf den kleineren Sohn
            if (a[l] > a[son]) {      // ist Tausch nötig?
                  exchange(&a[l], &a[son];      // call by reference
                  l = son;            // l muss evtl weiter einsinken!
            }
            else break;               // l ist weit genug eingesunken
}
```

Stephan Alt
stephan.alt@rwth-aachen.de

Sortieralgorithmen

Sortierung

- oberstes Element mit dem Ende tauschen, neues oberstes Element einsinken lassen um Heapeigenschaft wiederherzustellen
- Dabei werden alle bereits nach unten geschobenen kleinen Elemente „gesperrt“, sodass diese nicht mehr ihren Platz ändern können (ändern des rechten Randes bei sink)

Herstellung des Heaps

- sink-Funktion auf die ersten n-halbe Elemente des Arrays anwenden, beginnend mit dem n-halbsten bis zum ersten

Sortieralgorithmen

Heap Sort

- Laufzeit: O(n*log(n)) im average case
- wir behandeln mit Heap Sort nur Arrays in den Grenzen 1 bis r, da dadurch die Position der Kinder leichter zu berechnen ist! Deshalb ist es nur nötig die rechte Grenze als Parameter zu übergeben

```
void heap_sort(int a[], int r)          // a wird in den Grenzen 1 und r sortiert
{
      int i;
      for (i = r/2; i > 0; i--) {
            sink(a, i, r);                    // Heap aufbauen
      }
      for (i = r; i > 1; i--) {               // Heap wird immer kleiner
            exchange (&a[1], &a[ i ]);        // Minimum ans Heapende setzen
            sink(a, 1, i-1);                  // Heap reparieren
      }
}
```

Stephan Alt
stephan.alt@rwth-aachen.de