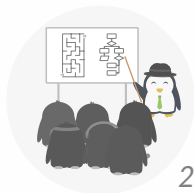


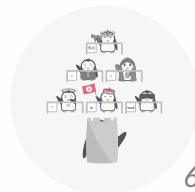
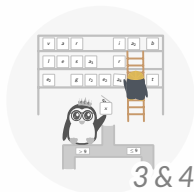
quaerere atque disponere

Tutorium Macht

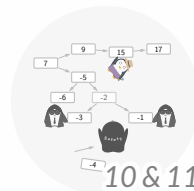
Theorie



Grundlagen



Vertiefungen



- Die Rekursionsfamilie (Methoden die sich selbst aufrufen) ist riesig:
 - Ruft sich eine Methode maximal einmal selbst auf, ist sie linear rekursiv.
 - Head-Rekursiv, wenn dieser Aufruf das erste Statement ist (alles passiert im Aufstieg)
 - Tail-Rekursiv, wenn dieser Aufruf das letzte Statement ist (alles passiert im Abstieg)
 - Ruft sie sich auch mehrfach pro Rekursionsfall auf, ist sie verzweigt rekursiv.
- Die „Big-O-Notation“ nutzen wir zum (z.B.) Vergleich der Laufzeitkomplexität:
 - $T(n) \in \mathcal{O}(f(n)) \iff \exists n_0 \in \mathbb{N} \, c \in \mathbb{R}^+ \, \forall n \geq n_0 : T(n) \leq c \cdot f(n)$
 - Ab einem gewissen Punkt (n_0) ist $c \cdot f(n)$ stets größer als $T(n)$

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(2^n)$	$\mathcal{O}(n!)$
Bez:	konst.	logarithm.	linear	linear log.	quadratisch	kubisch	exponentiell	faktoriell

Präsenzaufgabe

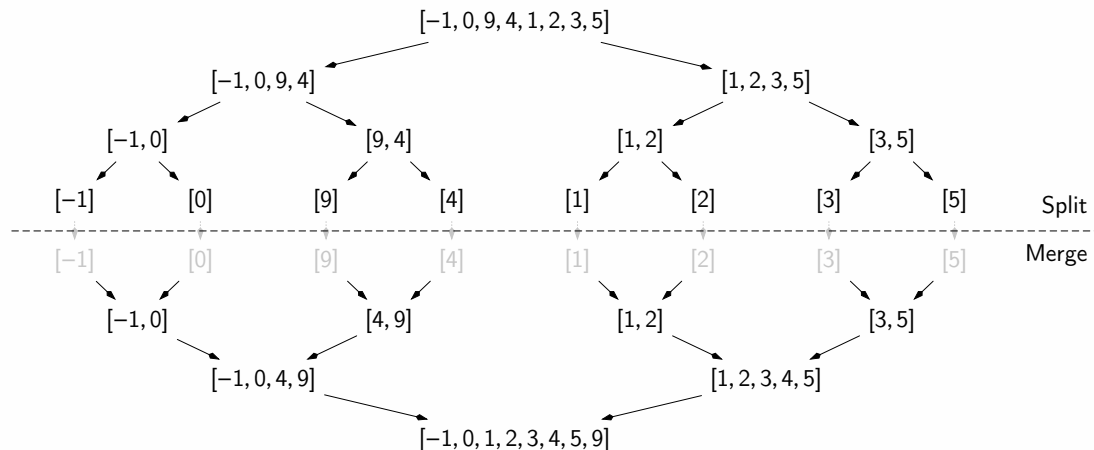
1

Mit vereinter Kraft!

Sortieren Sie das folgende Array händisch absteigend mit dem Merge Sort Algorithmus. Geben Sie die Split- und die Mergephase an und begründen Sie, anhand dieser informell die worst case Laufzeit von $\mathcal{O}(n \log n)$ für den Mergesort Algorithmus.

$[-1, 0, 9, 4, 1, 2, 3, 5]$

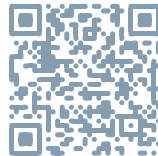
Ein wenig Mergesort



- › Mergesort ist „easy split, hard join“, bezeichne nun n die Eingabelänge
- › Das Aufteilen benötigt immer $\mathcal{O}(\log n)$ viele Schritte (wie bei der binären Suche)
Genau genommen kommt das auf die Implementation an, nutzen wir Arrays und müssen diese jedes mal kopieren landen wir schon eher bei $\mathcal{O}(n \log n)$,
- › Merge hat auch $\mathcal{O}(\log n)$ Schritte und muss pro Schritt $\mathcal{O}(n)$ Elemente vergleichen
Bei Mergesort vergleichen wir ja jeweils nur die vordersten Elemente beider Arrays, jeder Vergleich übernimmt ein Element in das Zielarray des Schrittes.
- › Alle anderen Operationen (Kopie der Zahl ins Ziel-Array, ...) sind konstant
Selbst, wenn sie nicht konstant wären, sind Array-Erzeugungen und Kopien alle auch maximal in $\mathcal{O}(n)$ machbar.
- › Wir erhalten: $\mathcal{O}(\log n) + \mathcal{O}(n) \cdot \mathcal{O}(\log n) = \mathcal{O}(\max\{\log n, n \cdot \log n\}) = \mathcal{O}(n \log n)$

Mehr Merge! Mehr Sort!

- › Es empfiehlt sich irgendwas zwischen die Zahlen zu setzen (wie ein Komma), sonst rutschen diese Erfahrungsgemäß (zu) eng zusammen
- › Die Verdopplung des Schrittes bei Split und Merge dient der Übersicht, sie ist nicht notwendig.
- › Merge-Sort gibt es in einigen Flavors. Beispielsweise in-place mergesort:



Übungsblatt 8

Aufgabe 1: Naive Sortialgorithmen

Teil von Übungsblatt 2 war der Entwurf eines naiven Sortialgorithmus, welcher durch iteratives Vertauschen eine gegebene Liste aufsteigend sortiert. Hier finden Sie die Algorithmen als Pseudocode nochmals aufgeführt:

Input : List $L = (l_1, \dots, l_N)$, Indizes i und j

- 1 make new N have value $\text{length}(L)$;
- 2 **if** $N < 2$ **or** $i < 1$ **or** $j < 1$ **or** $i > N$ **or** $j > N$ **then**
- 3 print out „Eingabe ungültig!“;
- 4 stop program;
- 5 make new t have value l_i ;
- 6 make l_i have value l_j ;
- 7 make l_j have value t ;

Input : List $L = (l_1, \dots, l_N)$

- 1 make new N have value $\text{length}(L)$;
- 2 **if** $N < 2$ **then** print out „Eingabe ungültig!“ and stop program ;
- 3 make new i have value 1;
- 4 **while** $i < N$ **do**
- 5 make new j have the value $i + 1$;
- 6 **while** $j \leq N$ **do**
- 7 **if** $l_i > l_j$ **then** call vertausche(L, i, j);
- 8 Increment j by 1;
- 9 Increment i by 1;

a) Implementieren Sie dieses Verfahren nun in Java. Halten Sie sich dabei genau an die Angaben aus den gegebenen Algorithmen und testen Sie Ihre Implementierung an einem Beispiel.

b) Für die Pseudocodevariante des Sortierverfahrens haben wir bereits eine Laufzeitabschätzung von:

$$T_{\max}^{\text{sort}}(n) = 3 + 3n + 12 \cdot \frac{n^2 - n}{3}$$
 Elementaroperationen (s. Übungsblatt 2) angegeben. Geben Sie nun hierfür die Laufzeit in der \mathcal{O} -Notation an und begründen Sie ihre Antwort kurz.

Von Zahlen war nicht die Rede. Von Arrays auch nicht!
Das sind (willkürlich sinnvolle) Annahmen.

```
public static void vertausche(int[] arr, int i, int j) {  
    int n = arr.length;  
    if(n < 2 || i < 0 || j < 0 || i >= n || j >= n) {  
        System.err.println("Eingabe ungültig!");  
        return;  
    }  
    int t = arr[i];  
    arr[i] = arr[j];  
    arr[j] = t;  
}
```

Mathematisch: $\{1, \dots, n\}$, Java: $\{0, \dots, n-1\}$

Was machen wir aus „stop program“?

- `System.exit(1)`?
- Werfen einer Exception?

Input : List $L = (l_1, \dots, l_N)$, Indizes i und j

- 1 make new N have value $\text{length}(L)$;
- 2 if $N < 2$ or $i < 1$ or $j < 1$ or $i > N$ or $j > N$ then
 - 3 print out „Eingabe ungültig!“;
 - 4 stop program;
- 5 make new t have value l_i ;
- 6 make l_i have value l_j ;
- 7 make l_j have value t ;

*Ich habe mir hier die Freiheit genommen,
 L und N umzubenennen.*

Sort it!

```
public static void sortiere(int[] arr) {  
    int n = arr.length;  
    if (n < 2) {  
        System.err.println("Eingabe ungültig!");  
        return;  
    }  
    int i = 0;  
    while(i < arr.length - 1) {  
        int j = i + 1;  
        while(j < arr.length) {  
            if (arr[i] > arr[j]) {  
                vertausche(arr, i, j);  
            }  
            j++;  
        }  
        i++;  
    }  
}
```

Input : List $L = (l_1, \dots, l_N)$

- 1 make new N have value length(L);
- 2 if $N < 2$ then print out „Eingabe ungültig!“ and stop program ;
- 3 make new i have value 1;
- 4 while $i < N$ do
- 5 | make new j have the value $i + 1$;
- 6 | while $j \leq N$ do
- 7 | | if $l_i > l_j$ then call
- 8 | | | vertausche(L, i, j);
- 9 | | Increment j by 1;
- Increment i by 1;

Sort it, for the children loops!

```
public static void sortiere(int[] arr) {  
    int n = arr.length; // n = arr.length - 1?  
    if (n < 2) {  
        System.err.println("Eingabe ungültig!");  
    }  
    for (int i = 0; i < arr.length; i++) { // vs. arr.length - 1?  
        for (int j = i + 1; j < arr.length; j++) {  
            if (arr[i] > arr[j]) {  
                vertausche(arr, i, j);  
            }  
        }  
    }  
}
```

Hier sind auch **for**-Schleifen in Ordnung, da sie für den gezeigten Fall ja nur eine Kurzschreibweise sind.



Testen an einem Beispiel

 NaiveSort.java

```
public class NaiveSort {  
    public static void vertausche(int[] arr, int i, int j) { ... }  
    public static void sortiere(int[] arr) { ... }
```

[▷ display in browser](#)

```
public static void main(String[] args) {  
    int[] arr = { 5, 8, 1, -1 };  
    sortiere(arr);  
    for (int i : arr)  
        System.out.println(i);  
}  
}
```

- › Nun fehlt noch die Laufzeit in \mathcal{O} -Notation für:

$$T_{\max}^{\text{sort}}(n) = 3 + 3n + 12 \cdot \frac{n^2 - n}{3}$$

- › Vereinfachen wir zunächst:

$$T_{\max}^{\text{sort}}(n) = 3 + 3n + 4 \cdot n^2 - 4 \cdot n = 3 - 1 \cdot n + 4 \cdot n^2$$

- › Was Wachstumsverhalten des Ausdrucks wird von n^2 dominiert, der Leitkoeffizient entfällt wie der Rest nach den Rechenregeln.
- › Damit ist: $T_{\max}^{\text{sort}}(n) \in \mathcal{O}(n^2)$

Aufgabe 2: Nicht ganz so naive Sortieralgorithmen

Betrachten Sie folgende Implementierung von Insertion Sort zur aufsteigenden Sortierung:

```
public static void insertionSort(int[] arr) {  
    int pos, key;  
    for (int i = 1; i < arr.length; i++) {  
        pos = i - 1;  
        key = arr[pos + 1];  
        while (pos >= 0 && arr[pos] > key) {  
            arr[pos + 1] = arr[pos];  
            pos--;  
        }  
        arr[pos + 1] = key;  
    }  
}
```

- a) Unter welchen Bedingungen tritt im obigen Sortieralgorithmus der **best case** ein? Geben Sie die Laufzeit hierfür in O -Notation an.
- b) Unter welchen Bedingungen tritt im obigen Sortieralgorithmus der **worst case** ein? Geben Sie die Laufzeit hierfür in O -Notation an.

- a) Unter welchen Bedingungen tritt im obigen Sortieralgorithmus der **best case** ein? Geben Sie die Laufzeit hierfür in \mathcal{O} -Notation an.

Ist das Array bereits *aufsteigend* sortiert, läuft die äußere Schleife $n - 1$ -mal und die innere nie, da nie „`arr[pos] > arr[pos + 1]`“ eintritt.

Dies ist der best case mit $\mathcal{O}(n)$ (der Rest ist $\mathcal{O}(1)$).

- b) Unter welchen Bedingungen tritt im obigen Sortieralgorithmus der **worst case** ein? Geben Sie die Laufzeit hierfür in \mathcal{O} -Notation an.

Ist das Array *absteigend* sortiert, läuft die äußere Schleife $n - 1$ mal, und die innere über den kleinen Gauß insgesamt $\mathcal{O}(n^2)$ mal, da stets „`arr[pos] > arr[pos + 1]`“ gilt.

Somit haben wir einen worst case mit einer Laufzeitkomplexität von $\mathcal{O}(n^2)$.

Aufgabe 3: Rekursives Suchen

In dieser Aufgabe sollen Sie den iterativen Binary Search aus der Vorlesung in eine rekursive Variante umprogrammieren. Beantworten Sie die Fragen als Kommentar in der entsprechenden Java Datei.

- a) Implementieren Sie eine rekursive Version des Binary Search Algorithmus. Für den Fall, dass der gesuchte Wert nicht gefunden werden konnte, geben Sie eine Meldung auf die Konsole aus sowie den Index -1 zurück. Testen Sie Ihre Implementierung an einem Beispiel. Nehmen Sie an, dass die Eingaben immer sortiert sind.*
- b) Ist Ihre Implementierung aus Teilaufgabe a) Kopf- oder End-rekursiv? Begründen Sie Ihre Antwort kurz.*
- c) Geben Sie an, unter welchen Bedingungen für Ihre Implementierung der worst-case bzgl. der Laufzeit eintritt. Begründen Sie Ihre Antwort kurz.*
- d) Geben Sie die bestmögliche Laufzeitabschätzung für den worst-case in \mathcal{O} -Notation an. Begründen Sie Ihre Antwort kurz.*

Save me, Sonic!

```
public static int binSearchRec(int[] arr, int key, int left, int right) {  
    if (left > right) {  
        System.out.println("Wert konnte nicht gefunden werden!");  
        return -1;  
    }  
    int mid = (left + right) / 2;  
    if (arr[mid] == key) {  
        return mid;  
    } else if (arr[mid] > key) {  
        return binSearchRec(arr, key, left, mid - 1);  
    } else {  
        return binSearchRec(arr, key, mid + 1, right);  
    }  
}
```

Eine wunderschöne Tail(s)-Rekursion!

Kunckles! Echidnas want to test that!

 `BinarySearchRecursive.java`

```
public class BinarySearchRecursive {  
    public static int binSearchRec(int[] arr, int k, int l, int r) { ... }  
  
    public static void main(String[] ars) {  
        int[] arr = { 1, 2, 4, 8, 16, 32, 64, 128, 256 };  
        System.out.println("Pos: " + binSearchRec(arr, 7, 0, arr.length - 1));  
    }  
}
```

- b) *Ist Ihre Implementierung aus Teilaufgabe a) Kopf- oder End-rekursiv? Begründen Sie Ihre Antwort kurz.*

Wie bereits angemerkt liegt eine Endrekursion vor: alle Berechnungen passieren im Abstieg, die rekursiven Aufrufe sind für alle rekursiven Ausführungspfade das letzte Statement.

- c) *Geben Sie an, unter welchen Bedingungen für Ihre Implementierung der worst-case bzgl. der Laufzeit eintritt. Begründen Sie Ihre Antwort kurz.*

Wenn der gesuchte Wert nicht im Array enthalten ist, müssen wir die meisten Vergleiche durchführen.

Nachdem wir das letzte Element betrachtet haben, gehen wir noch einmal in die Rekursion um dann festzustellen, dass es nichts mehr zu durchsuchen gibt — das Suchfenster ist leer.

- d) *Geben Sie die bestmögliche Laufzeitabschätzung für den worst-case in \mathcal{O} -Notation an. Begründen Sie Ihre Antwort kurz.*

Die Analyse bleibt zur iterativen Variante weitgehend unverändert. Wir können das Array $\log_2 n$ -mal teilen und haben somit $\mathcal{O}(\log_2 n)$ rekursive Aufrufe. Da alle anderen Operationen konstant sind, landen wir somit bei:

$$T_{\max}^{\text{rec. bin-search}} \in \mathcal{O}(\log n)$$

- › Im Folgenden ein leicht modifizierter Ansatz an die binäre Suche

Aus dem letzten Semester

- › Wenn ein Element mehrfach enthalten ist, möchten wir nun den kleinsten Index
- › Sonst folgen ein paar ganz brave Pingus...

```
> int search(int[] arr, int l, int r, int e) {  
    if(r < l) return -1;  
    int mid = l + (r - l) / 2;  
    if (arr[mid] == e) return mid;  
    else if (arr[mid] > e)  
        return search(arr, l, mid - 1, e);  
    else  
        return search(arr, mid + 1, r, e);  
}  
  
int find(int[] arr, int e) {  
> int idx = search(arr, 0, arr.length - 1, e);  
    if(idx == -1) return -1;  
    for(int i = 1; i <= idx; i++) {  
        if(arr[idx - i] != e)  
            return idx - i + 1;  
    }  
    return 0;  
}  
  
    > find(new int[]{-14, -14, 24, 24, 42, 109, 113}, -14);
```




```
int search(int[] arr, int l, int r, int e) {
    if(r < l) return -1; r=6, l=0
    int mid = l + (r - l) / 2; mid=3
    if (arr[mid] == e) return mid; e=-14
    else if (arr[mid] > e) 24 > -14
        > return search(arr, l, mid - 1, e);
    else
        return search(arr, mid + 1, r, e);
}

int find(int[] arr, int e) {
    > int idx = search(arr, 0, arr.length - 1, e);
    if(idx == -1) return -1;
    for(int i = 1; i <= idx; i++) {
        if(arr[idx - i] != e)
            return idx - i + 1;
    }
    return 0;
}

> find(new int[]{-14, -14, 24, 24, 42, 109, 113}, -14);
```



```
int search(int[] arr, int l, int r, int e) {  
    if(r < l) return -1; r=2, l=0  
    int mid = l + (r - l) / 2; mid=1  
> if (arr[mid] == e) return mid; e=-14, mid=1  
    else if (arr[mid] > e)  
        > return search(arr, l, mid - 1, e);  
    else  
        return search(arr, mid + 1, r, e);  
}
```



```
int find(int[] arr, int e) {  
> int idx = search(arr, 0, arr.length - 1, e);  
    if(idx == -1) return -1;  
    for(int i = 1; i <= idx; i++) {  
        if(arr[idx - i] != e)  
            return idx - i + 1;  
    }  
    return 0;  
}
```



```
> find(new int[]{-14, -14, 24, 24, 42, 109, 113}, -14);
```

```
int search(int[] arr, int l, int r, int e) {  
    if(r < l) return -1;  
    int mid = l + (r - l) / 2;  
    if (arr[mid] == e) return mid;  
    else if (arr[mid] > e)  
        return search(arr, l, mid - 1, e);  
    else  
        return search(arr, mid + 1, r, e);  
}  
  
int find(int[] arr, int e) {  
    int idx = search(arr, 0, arr.length - 1, e);  
> if(idx == -1) return -1; idx=1  
    for(int i = 1; i <= idx; i++) {  
        if(arr[idx - i] != e)  
            return idx - i + 1;  
    }  
    return 0;  
}
```



find(new int[]{-14, -14, 24, 24, 42, 109, 113}, -14);

```
int search(int[] arr, int l, int r, int e) {  
    if(r < l) return -1;  
    int mid = l + (r - l) / 2;  
    if (arr[mid] == e) return mid;  
    else if (arr[mid] > e)  
        return search(arr, l, mid - 1, e);  
    else  
        return search(arr, mid + 1, r, e);  
}  
  
int find(int[] arr, int e) {  
    int idx = search(arr, 0, arr.length - 1, e);  
    if(idx == -1) return -1;  
    for(int i = 1; i <= idx; i++) {  
        if(arr[idx - i] != e)  
            return idx - i + 1;  
    }  
    return 0; //Wir sind ganz links  
}
```

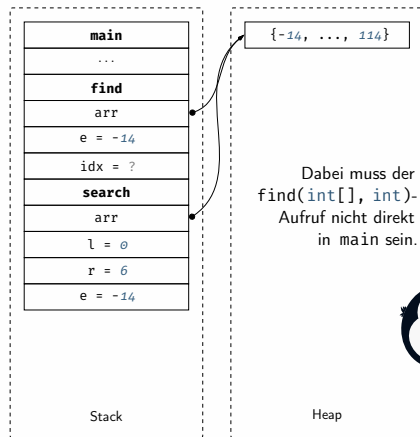


find(new int[]{-14, -14, 24, 24, 42, 109, 113}, -14); < // → 0

```
> int search(int[] arr, int l, int r, int e) {
    if(r < l) return -1;
    int mid = l + (r - l) / 2;
    if (arr[mid] == e) return mid;
    else if (arr[mid] > e)
        return search(arr, l, mid - 1, e);
    else
        return search(arr, mid + 1, r, e);
}

int find(int[] arr, int e) {
> int idx = search(arr, 0, arr.length - 1, e);
    if(idx == -1) return -1;
    for(int i = 1; i <= idx; i++) {
        if(arr[idx - i] != e)
            return idx - i + 1;
    }
    return 0;
}
```

```
> find(new int[]{-14, -14, 24, 24, 42, 109, 113}, -14);
```



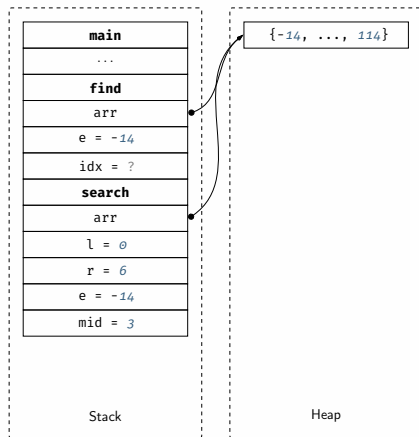
```

int search(int[] arr, int l, int r, int e) {
    if(r < l) return -1; r=6, l=0
    int mid = l + (r - l) / 2; mid=3
    if (arr[mid] == e) return mid; e=-14
    else if (arr[mid] > e) 24 > -14
        > return search(arr, l, mid - 1, e);
    else
        return search(arr, mid + 1, r, e);
}

int find(int[] arr, int e) {
> int idx = search(arr, 0, arr.length - 1, e);
    if(idx == -1) return -1;
    for(int i = 1; i <= idx; i++) {
        if(arr[idx - i] != e)
            return idx - i + 1;
    }
    return 0;
}

```

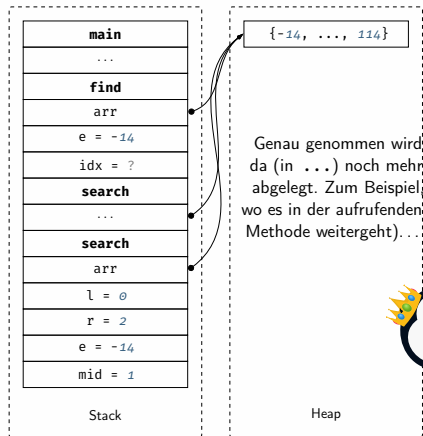
> find(new int[]{-14, -14, 24, 24, 42, 109, 113}, -14);



```
int search(int[] arr, int l, int r, int e) {
    if(r < l) return -1; r=2, l=0
    int mid = l + (r - l) / 2; mid=1
> if (arr[mid] == e) return mid; e=-14, mid=1
    else if (arr[mid] > e)
        > return search(arr, l, mid - 1, e);
    else
        return search(arr, mid + 1, r, e);
}

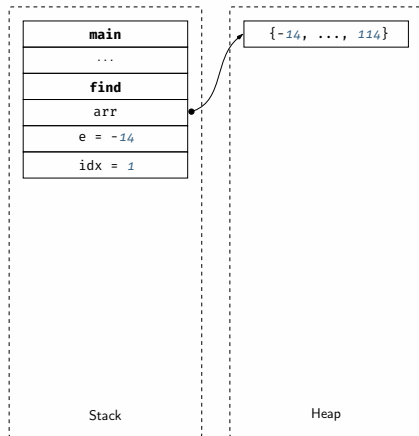
int find(int[] arr, int e) {
> int idx = search(arr, 0, arr.length - 1, e);
    if(idx == -1) return -1;
    for(int i = 1; i <= idx; i++) {
        if(arr[idx - i] != e)
            return idx - i + 1;
    }
    return 0;
}
```

> find(new int[]{-14, -14, 24, 24, 42, 109, 113}, -14);



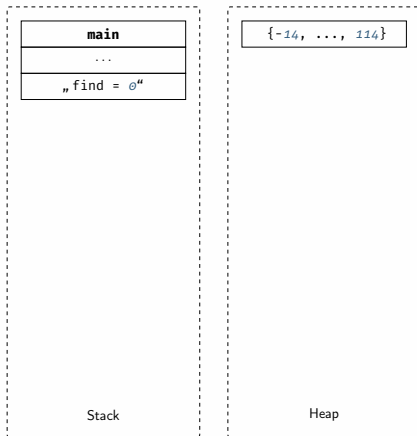
```
int search(int[] arr, int l, int r, int e) {  
    if(r < l) return -1;  
    int mid = l + (r - l) / 2;  
    if (arr[mid] == e) return mid;  
    else if (arr[mid] > e)  
        return search(arr, l, mid - 1, e);  
    else  
        return search(arr, mid + 1, r, e);  
}  
  
int find(int[] arr, int e) {  
    int idx = search(arr, 0, arr.length - 1, e);  
> if(idx == -1) return -1; idx=1  
    for(int i = 1; i <= idx; i++) {  
        if(arr[idx - i] != e)  
            return idx - i + 1;  
    }  
    return 0;  
}
```

```
find(new int[]{-14, -14, 24, 24, 42, 109, 113}, -14);
```




```
int search(int[] arr, int l, int r, int e) {  
    if(r < l) return -1;  
    int mid = l + (r - l) / 2;  
    if (arr[mid] == e) return mid;  
    else if (arr[mid] > e)  
        return search(arr, l, mid - 1, e);  
    else  
        return search(arr, mid + 1, r, e);  
}  
  
int find(int[] arr, int e) {  
    int idx = search(arr, 0, arr.length - 1, e);  
    if(idx == -1) return -1;  
    for(int i = 1; i <= idx; i++) {  
        if(arr[idx - i] != e)  
            return idx - i + 1;  
    }  
    return 0; //Wir sind ganz links  
}
```

```
find(new int[]{-14, -14, 24, 24, 42, 109, 113}, -14); < // → 0
```



Aussicht: Übungsblatt 9

Aufgabe 1: Heap Sort

- › Bei Heapsort verwenden wir die Datenstruktur, nicht den Heap-Speicher
- › Wir unterscheiden zwei Heap-Arten:
 - Max-Heap:** Eltern sind mindestens so groß wie die Kinder ($\text{parent} \geq \text{children}$)
 - Min-Heap:** Eltern sind maximal so groß wie die Kinder ($\text{parent} \leq \text{children}$)

FANTASTISCHE HEAPS

... und wo sie zu finden sind.
Datastructure Love!

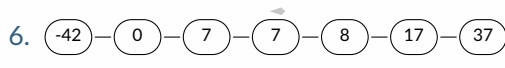
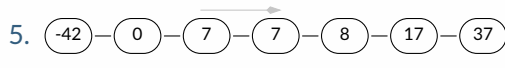
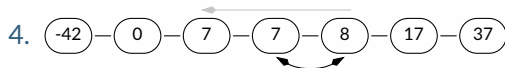
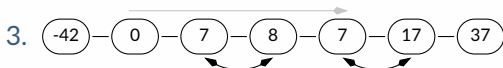
Florian Sihler

11. November 2022
SP, Universität Ulm



Aufgaben 2 und 3

- › Hier geschieht nicht wirklich etwas spannendes
- › Shaker-Sort sei hier einmal visualisiert für



- › Der letzte Schritt muss, je nach Abbruchbedingung nicht stattfinden.
- › Bei Aufgabe 3 ist Code-Verständnis gefragt

*Thus all sorts of sophisticated order-systems become possible, which
keep successively modifying themselves and hence also the
computational processes that are likewise under their control.
— John v. Neumann [Yon58]*

Abschließendes

[Yon58] John Yon Neumann. „The computer and the brain“. 1958

- Such- und Sortiervverfahren sind mit die „Musterkinder“ in der Informatik
 - Suchverfahren können durch eine vorhergehende Sortierung beschleunigt werden
 - Verschiedene Verfahren haben verschiedene Eigenschaften
 - Selectionsort kann beispielsweise jederzeit abgebrochen werden und liefert immer noch eine Teil-Sortierung (Mergesort beispielsweise nicht!)
- Es ist wichtig, ein Grundverständnis für alle Such- und Sortiervverfahren zu entwickeln
- Rekursion wird uns nicht nur beim Suchen und Sortieren begegnen
Uuuuh, dynamische-Datenstruktur mich! Weitere Verfahren folgen nächste Woche.



You expected... a recursion gag? Read this line again!
Still funny, after all those semesters...