# Hallo Welt für Fortgeschrittene

## Sortier- und Suchalgorithmen

Stefan Hengelein

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Informatik 2 • Programmiersysteme • Martensstraße 3 • 91058 Erlangen

# Motivation

*“Sorting is the most fundamental algorithmic problem in computer science“*

Skiena/Revilla („Programming Challanges“)

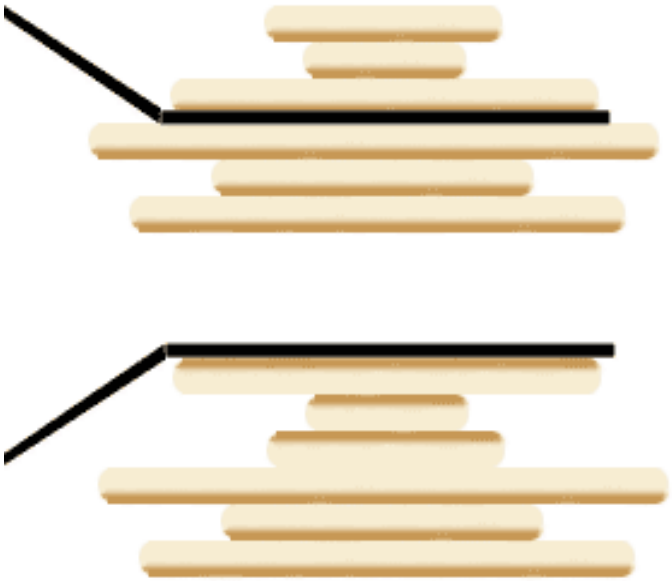
# Motivation

*“Computer manufacturers of the 1960s estimated that more than 25% of the running time on their computers was spend on sorting. [...] In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time.“*

Donald E. Knuth (Knuth 1998)

# Motivation

- **Idee:** Den Stapel Pancakes so oft wenden bis sie der Größe nach sortiert sind



Implementierung siehe ICPC Wintercontest 2011: Bazinga!

# Motivation: Sortieren - Suchen

- **Sortieren, wozu?**
  - Optimierung der Daten für anderen Algorithmus (zb. Suchen)
  - Test auf Eindeutigkeit
  - Entfernung von Duplikaten
  - Priorisierung von Ereignissen
  - Mengenschnitt / Vereinigung
  - Zählen von Häufigkeiten
  - …
- **Was wird gesucht?**
  - Beliebige Suchschlüssel
  - Minima / Maxima / Medians (Mittleres Element)
  - Allgemein: order statistics
  - ...

Allgemein:

lower: $\left\lfloor \frac{(n+1)}{2} \right\rfloor$

upper: $\left\lceil \frac{(n+1)}{2} \right\rceil$

Falls Array-Index gesucht noch -1!

$k^{th}$ order statistic = k-kleinstes Element

# Übersicht

- **Motivation**
- **Einleitung**
- **Sortieralgorithmen**
  - Vergleichsbasierte Algorithmen
  - Nicht vergleichsbasierte Algorithmen
- **Suchalgorithmen**
  - Selektierung
- **RangeMinimumQuery (RMQ) – Problem**
  - lowest common ancestor (LCA) – Problem
  - Segment Tree
- **Binary Indexed Tree (BIT) / Fenwick Tree**
- **Bibliotheksfunktionen**
- **Quellen**

# Eigenschaften von Algorithmen

- **Berechnungskomplexität**
  - Best case
  - Average case
  - Worst case
- **Speicherbedarf**
  - In-place (in-situ): zusätzlicher Speicher konstant & unabhängig von n
  - Out-of-place (ex-situ): zusätzlicher Speicher abhängig von n
- **Reihenfolge während Sortierung**
  - Stabil: Elemente mit gleichen Schlüsseln (oder Werten) behalten ihre relative Ordnung bei
  - Instabil: Stabilität nicht gewährleistet
- **Zusätzlich für Sortieralgorithmen**
  - natürlich: schneller auf vorsortierten Daten

Nützlich wenn z.B. Objekte mit mehreren Attributen sortiert werden sollen

# Erinnerung: O – Kalkül & Landau-Symbole

- **Berechnungskomplexität**
  - f(n) ∈ O(g(n)) : „g(n) ist asymptotische obere Schranke für f(n)“
    - heißt soviel wie: $g(n) \leq c \cdot f(n)$
  - f(n) ∈ Ω(g(n)) „g(n) ist asymptotische untere Schranke für f(n)“
    - heißt soviel wie: $g(n) \geq c \cdot f(n)$
  - f(n) ∈ Θ(g(n)) „g(n) hat gleiche Wachstumsordnung wie f(n)“
    - heißt soviel wie: f ∈ O(g) ^ f ∈ Ω(g)
  - f(n) ∈ o(g(n)) „g(n) hat kleinere Wachstumsordnung als f(n)“
    - heißt soviel wie: $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$
  - f(n) ∈ ω(g(n)) „g(n) hat größere Wachstumsordnung als f(n)“
    - heißt soviel wie: $$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$
  - f(n) ~ g(n) „g(n) und f(n) sind asymptotisch äquivalent“
    - heißt soviel wie: $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$$

# Sortieralgorithmen: Eigenschaften

- **Man unterscheidet:**
  - vergleichsbasierte Verfahren (comparison sorts)
  - lineare Verfahren, nicht vergleichsbasiert (linear sorts)

- **Beste Komplexität im *average* case:**
  - $\Theta(n \cdot \log(n))$

    vs.
  - $\Theta(n)$

- **Die Theoretische Informatik hat für <u>vergleichsbasierte</u> Verfahren gezeigt:** $\Omega(n \cdot \log(n))$

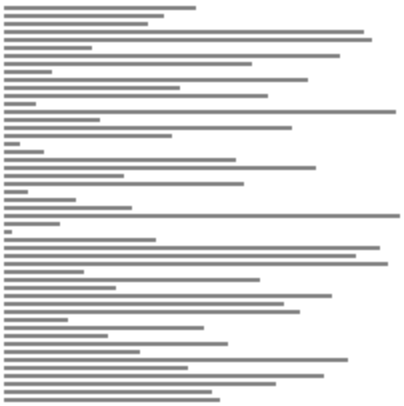
Besser geht's nicht!

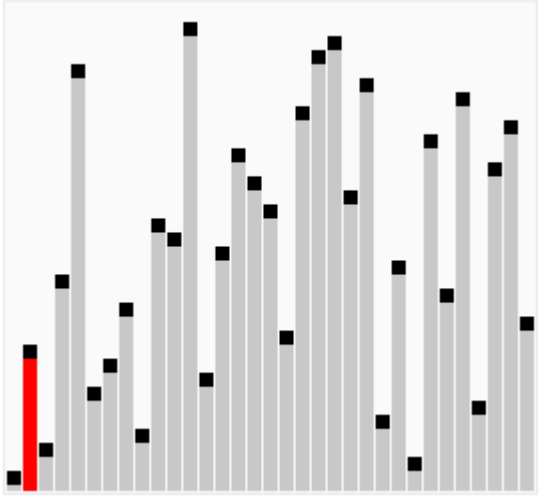
# Sortieren: Einige Algorithmen

- **Naive & asymptotisch <u>nicht</u> optimale Verfahren**
  - Bubblesort / Cocktailsort
  - Insertionsort
  - Shellsort
- **Asymptotisch optimale Algorithmen**
  - Mergesort
  - Quicksort
  - Heapsort
- **lineare Verfahren**
  - Bucketsort / Radixsort
- **Übersicht**

# Bubblesort – of rabbits and turtles

- Laufe Sequenz von [0,n] ab und vertausche jeweils benachbarte Elemente die in falscher Reihenfolge sind
  - Falls Element n erreicht, beginne von vorn
- Große Elemente am Anfang (rabbits) wandern schnell nach hinten (blubbern nach oben), kleinere hinten (turtles) aber nur langsam nach vorne
- Besser: Cocktailsort





| | |
|---|---|
| Avg | $O(n^2)$ |
| Stabil | ja |
| Speicher | in-place |

# Cocktailsort - Implementierung

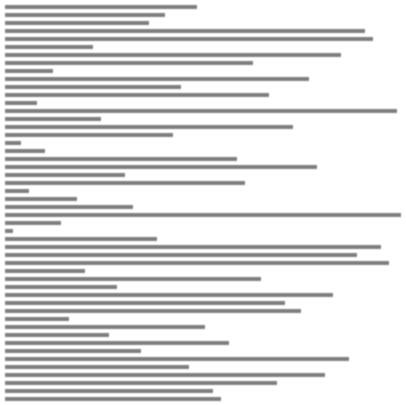
- Funktioniert wie Bubblesort, nur in beide Richtungen!

```
public static void cocktailSort(int a[]) {
  int lower = 1, higher = a.length-1, lastswap = a.length-1;
  do {
    for (int i = higher; i >= lower; i--) {              // bubble down
      if (v[i-1] > v[i]) {
        int swp = v[i-1];
        v[i-1] = v[i];  v[i] = swp;  lastswap = i;
    }} lower = lastswap + 1;
    for (int i = lower; i <= higher; i++) {              // bubble up
      if (v[i-1] > v[i]) {
        int swp = v[i-1];
        v[i-1] = v[i];
        v[i] = swp;
        lastswap = i;
    }} higher = lastswap - 1;
  } while (lower <= higher);
}
```

# Insertionsort

```
public static void insertionSort(int a[], int size) {
    for (int i = 1; i < size; i++) {
        int j, tmp = a[i];
        for (j = i - 1; j >= 0; j--) {
            if (a[j] > tmp) a[j+1] = a[j];
            else break;
        } a[j+1] = tmp; } }
```

| | |
|---|---|
| Avg | $O(n^2)$ |
| Stabil | ja |
| Speicher | in-place |



```
public static void binaryInsertionSort(int a[], int size) {
    for (int i = 1; i < size; i++) {
        int ins = BinarySearch (a, 0, i, a[i]);  // (a[], lowerB, upperB, key)
        if (ins < i) {
            int tmp = a[i];
            for (int j = i - 1; j >= ins; j--)
                a[j + 1] = a[j];
            a[ins] = tmp; }
} }
```

$\log_2(n)$ vergleiche im worst case → $\Theta(n \cdot \log(n))$ aber:

schneller als normaler Insertionsort aber wegen der benötigten swaps immer noch $O(n^2)$

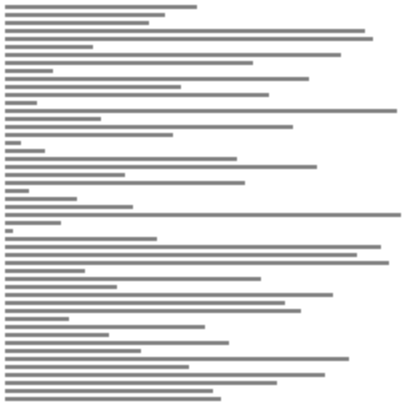
# Shellsort – ein Insertionsort-Derivat

- **Idee:**
  - Aufteilen in Gruppen und Spalten sortieren bis columns[n-1]
  - Zusammenfügen und sortieren

Empirisch ermittelte gap sequence

```
public static void shellSort(int a[]) {
    int[ ] columns = {9841, 3280, 1093, 364, 121, 40, 13, 4, 1};
    for (int c : columns) {
        for (int i = c; i < a.length; i++) {
            int t = a[i];
            int j = i;
            while(j >= c && a[j-c] > t) {
                a[j] = a[j-c];
                j -= c;
            }
            a[j] = t;
        }
    }
}
```

| | |
|---|---|
| Avg | $O(n\log^2(n))$ |
| Stabil | ja |
| Speicher | in-place |



# Naive Verfahren....

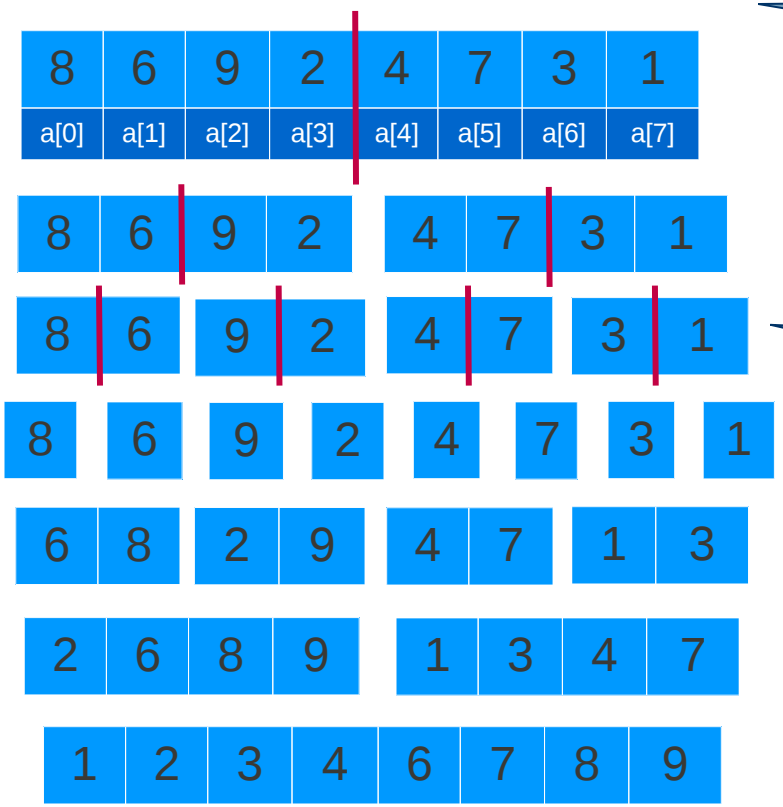
- **Selectionsort** Maximum suchen und an den Anfang der Ergebnisliste anfügen
- **Shellsort** verbesserter Insertionsort, der immer kleiner werdende „Spalten“ sortiert
- ...und viele mehr

- **Problem:**
  - Besitzen (fast) alle $O(n^2)$ – Komplexität
- **aber sind zum Teil für:**
  - <u>kleine</u> und/oder
  - <u>vorsortierte</u> Sequenzen

  Ziemlich effizient und meistens leicht zu implementieren (insbesondere Insertionsort)

# Mergesort – divide and conquer

- **Idee:** teile Sequenz in 2 Hälften auf und sortiere rekursiv die entstandenen Teilsequenzen wieder mit Mergesort
  - Wenn nur noch 1 Element in Teilliste, füge diese nach dem Reißverschlussprinzip zusammen




Merge

Vergleicht nur die ersten Elemente der Teilsequenz → kein wahlfreier Zugriff nötig

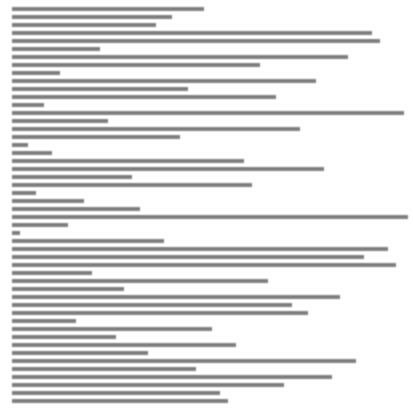
verbesserte Implementierungen verwenden bei kleinen Teilmengen naive Sortieralgorithmen

| | |
|---|---|
| Avg | O(nlog(n)) |
| Stabil | ja |
| Speicher | O(n) |

# Mergesort – Implementierung

```
public static void mergeSort(int arr[], int tmp[], int start, int end) {
    if (start < end) {
        int mid = (start+end)/2;
        mergeSort(arr, tmp, start, mid);
        mergeSort(arr, tmp, mid+1, end);
        merge(arr, tmp, start, mid, end);
    }  }
```

```
public static void merge(int arr[], int tmp[], int start, int m, int end) {
    int i = 0, j = start, k;
    while(j <= m) tmp[i++] = arr[j++];
    i = 0, k = start;
    while ( k < j && j <= end) {
        if(tmp[i] <= arr[j])
            arr[k++] = tmp[i++];
        else
            arr[k++] = arr[j++];
    }
    while (k < j) arr[k++] = tmp[i++];
}
```



# Quicksort

- **Idee:**
  - Wähle ein beliebiges Pivotelement
  - Sortiere alle kleineren Elemente links und alle größeren rechts vom Pivotelement ein → Pivotelement ist an finaler Position
  - Wiederhole rekursiv für die entstandenen Teillisten (solange |Teilliste| > 1) links & rechts des Pivots

| 8 | 6 | 9 | 2 | 4 | 7 | 3 | 1 |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |

Pivot: 6

| 2 | 4 | 3 | 1 | 6 | 8 | 7 | 9 |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |

sortieren

| 2 | 4 | 3 | 1 | 6 | 8 | 7 | 9 |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |

Pivot: 3, 7

| 2 | 1 | 3 | 4 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |

sortieren

Pivot: 2, 8

| 2 | 1 | 3 | 4 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |

sortieren

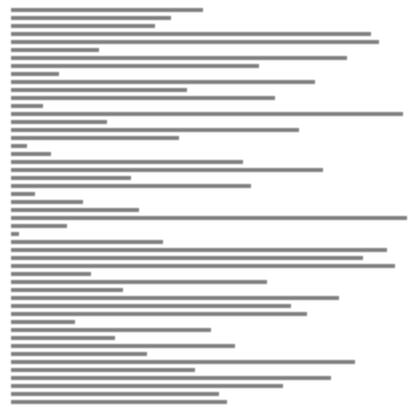
| 1 | 2 | 3 | 4 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |

| | |
|---|---|
| Avg | O(nlog(n)) |
| Stabil | nein |
| Speicher | in-place |

# Quicksort - Implementierung

```
public static void swap(int feld[], int a, int b) {
    int t = feld[a];
    feld[a] = feld[b];     feld[b] = t;     }
```

```
public static int partition(int feld[], int a, int b) {
    int p = (int)(Math.random()*(b-a+1) + a);  // random Pivot
    swap(feld, a, p);        int l = a, r = b;
    p = feld[a];
    while (l < r) {
        while (feld[r] > p) r--;
        while (feld[l] <= p && l < r) l++;
        if (l < r) swap(feld, l, r);
    }  swap(feld, a, l);
    return l;     }
```



```
public static void quickSort(int feld[], int a, int b) {
    if(end <= start) return;
    int i = partition(feld, a, b);
    quickSort(feld, a, i-1);
    quickSort(feld, i+1, b)
}
```

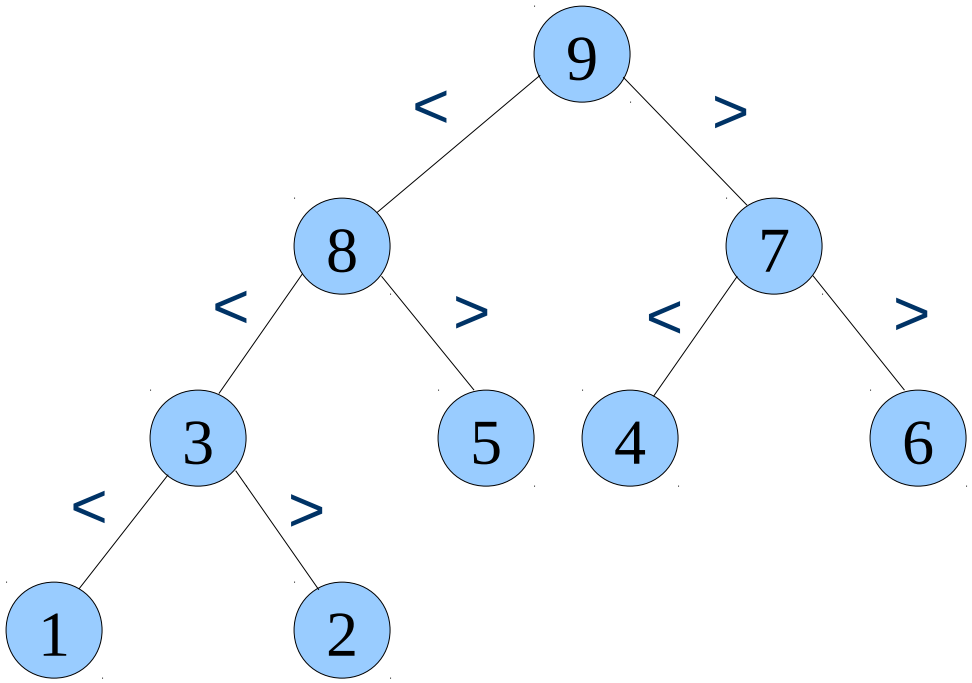
# Quicksort – zur Pivotwahl...

- **Quicksort steht und fällt mit der Wahl des Pivotelements**
  - schlechte Pivotwahl(z.B. immer kleinstes / größtes Element) → $O(n^2)$ Aufwand
  - passiert bei Pivotwahl am Anfang oder Ende mit vorsortierter Menge → Schrittweite 1 → $O(n^2)$
- **Verschiedene Ansätze um dies zu vermeiden:**
  - mittleres Element verwenden (günstig bei vorheriger partieller Sortierung)
  - Zufälliges Element: Chance auf worst-case Fall verringert
  - Median-of-three: Median von 3 zufälligen Elementen verwenden

# Heapsort

- **Max-Heap:**

| 5 | 8 | 6 | 2 | 9 | 4 | 7 | 1 | 3 |
|---|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] |




| 9 | 8 | 7 | 3 | 5 | 4 | 6 | 1 | 2 |
|---|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] |

**Zugriff auf Kinder:**
links: Index 2i+1
rechts: Index 2i+2

| | |
|---|---|
| Avg | O(nlog(n)) |
| Stabil | nein |
| Speicher | in-place |

# Heapsort

- **Idee:** Organisiere ein Array als Heap
  - stelle Heap-Eigenschaft her
  - tausche das unterste rechte Element (das letzte im unsortierten Teilarray) mit dem größten Element (Wurzel)
  - entferne das Maximum aus dem Heap (size--)
  - stelle Heap-Eigenschaft wieder her
  - wiederhole bis size == 1

In der Praxis meist etwas langsamer als Q-sort aber besser im worst-case

Vorallem für Priority Queues nützlich

- **Beispiel:**
  - Mit Heap-Eigenschaft:

    → Tafel

| 5 | 8 | 6 | 2 | 9 |
|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] |

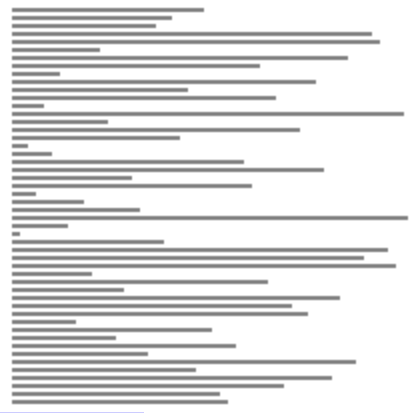
| 9 | 8 | 6 | 2 | 5 |
|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] |

**Zugriff auf Kinder:**
links: Index 2i+1
rechts: Index 2i+2

# Heapsort - Implementierung

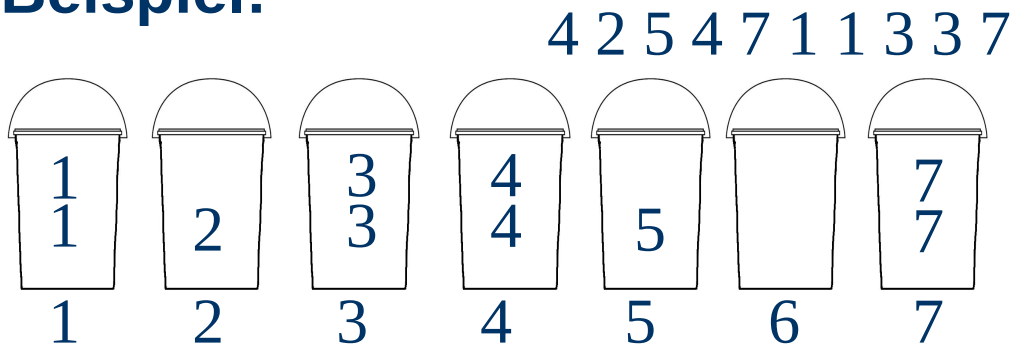
```
public static void heapSort(int[] a) {
    int n = a.length;
    for (int i = n/2; i >= 0; i--) heapify(a, i, n); // build heap
    while (--n > 0) {                                // sort
        swap(a, 0, n);
        heapify(a, 0, n);
}  }
```

```
public static void heapify(int[] a, int start, int end) {
    int j = 2*start+1,                    // pos. left child
    x = a[start];                         // save node
    while ((j = start*2 + 1) < end) {
        if (( j+1 < end) && (a[j+1] > a[j])) j++;
        if (a[j] <= x) break;
        a[start] = a[j];
        start = j;
    }
    a[start] = x;
}
```



# Bucketsort

- **Idee:**
  - fasse die Elemente zu kleineren Untermengen zusammen (kategorisiert)
  - Hauptunterschied zu den vorherigen Verfahren – es wird nicht verglichen
  - Üblicherweise wird, wenn die Problemgröße ausreichend verkleinert wurde, auf einen vergleichsbasierten Algorithmus gewechselt
  - Elemente einordnen: O(n) - Buckets auslesen: O(m) → O(n+m)
- **Beispiel:**




→ 1 1 2 3 3 4 4 5 7 7

# Bucketsort

```
public static void bucketSort(int[] a, int m) {
    int[] buckets = new int[m];
    for (int i = 0; i < a.length; i++) {
        buckets[ a[i] ]++;
    }
    for (int i = 0, j = 0; i < m; i++) {
        while (buckets[i]-- > 0)
            a[j++] = i;
    }
}
```

# Radixsort

- **Idee:**
  - Zum Sortieren von Integern
  - Basiert auf Bucketsort, nur dass hier die „Keys“ in ihre Segmente aufgeteilt werden
  - Beginne beim „unwichtigsten“ Segment → nach letzter Stelle sortieren
  - Nacheinander in die Liste zurückschreiben(relative Reihenfolge bleibt erhalten → stabil), und vorletzte Ziffern sortieren

# Übersicht: Sortieralgorithmen

| Algorithmus | Best | Average | Worst | Stabil | Speicher | Wann? |
|---|---|---|---|---|---|---|
| Bubble/ Cocktail | $O(n)$ | $O(n^2)$ | $O(n^2)$ | ja | in-place | kleine Probl. |
| Insertion | $O(n)$ | $O(n^2)$ | $O(n^2)$ | mögl. | in-place | kleine Probl. |
| Selection | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | mögl. | in-place | |
| Shell* | $O(n)$ | $O(n \log^2(n))$ | $O(n^{1,5})$ | nein | in-place | |
| Merge | $O(n \cdot \log(n))$ | $O(n \cdot \log(n))$ | $O(n \cdot \log(n))$ | ja | $O(n)$ | Listen / Arrays |
| Quick | $O(n \cdot \log(n))$ | $O(n \cdot \log(n))$ | $O(n^2)$ | nein | in-place | Arrays |
| Heap | $O(n \cdot \log(n))$ | $O(n \cdot \log(n))$ | $O(n \cdot \log(n))$ | nein | in-place | |
| Bucket / Counting | $O(n+m)$<br>m = Anzahl der Buckets | | | ja | $O(m)$ | Beschränkter Wertebereich |
| Radix | $O(k \cdot n)$<br>k = Anzahl der Segmente | | | ja | $O(n)$ | Segmentierbare Schlüssel |

* Stark implementierungabhängig

# Suchen - Übersicht

- **Suchalgorithmen**
  - Sequenzielle Suche (min / max)
  - Binäre Suche
  - Interpolierte Suche
  - Ternäre Suche

- **Selektion**
  - Quickselekt

# Sequenzielle Suche (min / max)

- Die intuitivste Lösung um in einem Array ein bestimmtes Element / das Minimum oder Maximum zu suchen ist das Array von Anfang bis Ende durch zulaufen und bei jedem Schritt zu vergleichen
  - Abwägen ob sich sortieren lohnt
- Maximum oder Minimum suchen in O(n)

# Sequenzielle Suche (min / max) - Implementierung

```
public static int linearSearch(int gesucht, int[] daten) {
    for (int i = 0; i < daten.length; i++) {
        if (daten[i] == gesucht)
            return i;
    }
    return -1;
}
```

```
public static int minSearch(int[] daten) {
    int min = 0;
    for (int i = 1; i < daten.length; i++) {
        if (daten[i] < daten[min])          // > für maxima
            min = i;
    }
    return a[min];
}
```

Falls Index gesucht wird:
**return** min;

# Binäre Suche – Divide and Conquer

- **Voraussetzung:**
  - sortierte Sequenz oder
  - Monoton steigende / fallende Funktion → Epsilon(nächste Seite)

Abwägen ob sich sortieren lohnt

- **Idee:** Betrachte das mittlere Element
  - Wenn der Schlüssel des mittleren Elements
    - Gleich des gesuchten Schlüssels ist → gefunden!
    - Kleiner ist, betrachte linke Seite
    - Größer ist, betrachte rechte Seite
    - Teilsequenz ist leer → Element nicht enthalten
      - (nicht bei Funktionen)
  - Wiederhole solange bis gefunden oder Genauigkeit ausreichend groß ist

# Binäre Suche – Implementierung

- **Falls mit Gleitkommazahlen gerechnet werden soll (z.B. bei Funktionen):**
  - Epsilon definieren (Genauigkeit)
  - | r – l | > Epsilon als Bedingung (begrenzte Genauigkeit von Gleitkommazahlen → Gefahr einer Endlosschleife)

```
public static int binarySearch(int[] a, int value) {
    int l = 0, r = n - 1;
    while (l <= r) {
        int m = (l + r) / 2;
        if (val == a[m]) return m;
        if (val < a[m]) {
            r = m - 1;
        } else {
            l = m + 1;
        }
    }
    return -1;  // wenn nicht gefunden
}
```

# Binäre Suche – Beispiel

monoton steigend

- **Age of the Universe** $expansion(t) = \sqrt[a]{b \cdot t^c \cdot \log_{1+d}(1+t^e) \cdot \exp(t^f)}$
  - Man bekommt Werte vorgegeben: expansion(t), a, b, c, d, e, f
  - $0 \leq t \leq 42$
  - Naiver Ansatz → durchprobieren, zu ungenau / zu langsam
  - Binäre Suche, werte Funktion mit mittlerem t aus, vergleiche mit expansion(t),
    - falls größer suche links weiter
    - sonst rechts
  - Beispiel mit konkreten Werten:

| expansion(t) | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| 9999 | 5 | 1 | 1 | 1 | 1 | 1 |

| t | f(t) |
|---|---|
| 21.0 | 165,3242 |
| 32.0 | 1663.7056 |
| 37.5 | 5203.8409 |
| 40.25 | 9182.4612 |
| 41.625 | 12191.8635 |
| ... | ... |
| 40.66306375403656 | 9999.000000264296 |

# Binäre Suche – Age of the Universe

```
final double EPSILON = 0.000001;
int exp, a, b, c, d, e, f;
//Einlesen der Werte
double l = 0.0, r = 42.0;
while (Math.abs(r - l) >= EPSILON) {
    double m = (l + r) / 2;
    double tmp = Math.pow(b*Math.pow(t, c)
                        *(Math.log(1+Math.pow(t, e))/Math.log(1+d))
                        * Math.exp(Math.pow(t, f)), (1.0/a);
    if (Math.abs(exp-tmp) < EPSILON) {
        System.out.println("t: " + m);
        break;
    } if (exp < tmp) {
        r = m - 1;
    } else {
        l = m + 1;
    }
}
```

# Binäre Suche – Eigenschaften

- **Problemgröße halbiert sich bei jedem Schritt**
  - konvergiert schnell
  - Komplexität O(log(n))
- **Auch bekannt als <u>Bisektion</u>**

  Beispielsweise zur Nullstellenbestimmung von Funktionen

- **Etwas andere Variante:** <u>Interpolationssuche</u>
  - Funktioniert wie Binäre suche nur wird anstatt des mittleren Elements ein Teilungspunkt t berechnet
    - Average Case: O(*log(log(n))*) aber worst case: O(n)
    - Intervallgrenzen werden abgeschätzt:

$$t = \frac{Anz.Elemente}{Anzahl\ verschiedener\ Elemente} \cdot gesuchter\ Wert \qquad t = l + (r - l) \cdot \frac{key - arr[l]}{arr[r] - arr[l]}$$

# Ternäre Suche

- Suche ein x auf dem Intervall [a,b], so dass die Aus - wertung einer Funktion f(x) maximal (oder minimal) wird
  - Intervall [a,b] von reellen Zahlen
- **Voraussetzung:**
  - Die Funktion hat nur einen „Gipfel“
  - Für Maximum:
    - Monoton steigend auf dem Intervall [a,x]
    - Monoton fallend auf dem Intervall [x,b]
  - Für Minimum: fallend [a,x], steigend [x,b]
- **Eigenschaften:**
  - Reduziert [a,b] pro Iteration auf 2/3 seiner vorherigen Größe
  - Nach 200 Iterationen ist die Abweichung auf $(2/3)^{200} = 6{,}05 \cdot 10^{-36}$ des vorherigen Intervalls geschrumpft

# Ternäre Suche - Implementierung

```
public static double ternarySearch(double a, double b) {
    double l = a, r = b;
    for(int i=0; i<200; i++) {
        double l1 = (l*2+r)/3;
        double l2 = (l+2*r)/3;
        if(f(l1) > f(l2))
            r = l2;
        else l = l1;
    }
    return l;
}
```

# Quickselect – sortieren, nur soweit nötig

- **Selektion:** Element finden welches in sortierter Sequenz den Index k hätte (k+1$^{th}$ order statistic)
  - Nur 1 Element aus einer unsortierten Liste gesucht, wozu dann komplett sortieren und suchen?
- **Wie funktioniert's ? „partieller Quicksort“**
  - partitioniere um ein Pivotelement
  - Vergleiche neuen Index mit k
    - wenn gleich, gefunden!
    - wenn k größer, suche in der rechten Teilhälfte weiter
    - wenn k kleiner, suche in der linken Teilhälfte weiter

vgl. Quicksort
gute Pivotwahl nötig

# Quickselect – Implementierung

```
public static int quickSelect(int feld[], int a, int b, int i) {
    while (a < b) {
        int p = partition(feld, a, b);
        if (p - a == i) return feld[p];
        if (p - a > i) {
            b = p - 1;
        } else {
            i -= p - a + 1;
            a = p + 1;
        }
    }
    return feld[a];
}
```

Arbeitet in-place! Wenn partielle Sortierung nicht erwünscht, vorher das Array kopieren

# Übersicht: Such-/ Selektionsalgorithmen

| Algorithmus | Best | Average | Worst |
|---|---|---|---|
| sequenziell | O(1) | O(n) | O(n) |
| sequenziell (min/max) | O(n) | O(n) | O(n) |
| binär | O(1) | O(log(n)) | O(log(n)) |
| ternär | O(log(n)) | O(log(n)) | O(log(n)) |
| interpoliert | O(1) | O(log(log(n))) | O(n) |
| Quickselect | O(1) | O(n) | $O(n^2)$ |

# Das RMQ - Problem

- **Problem:** Man hat ein Array der Länge n und möchte die Position eines Minimums im Teilarray [i, j] wobei $0 \leq i \leq j \leq n$

| 8 | 6 | 9 | 2 | 4 | 7 | 3 | 1 |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |

$RMQ_a(2,6) = 3$

- **Naive Lösung:** sequenzielle Suche des Minimums im Teilarray
  - Bei vielen Anfragen (queries) ziemlich aufwändig O(j-i) pro query
- **Mit Vorarbeit:** durchlaufe im Array alle Permutationen von i, j und speichere die Minima in 2-dim. Array an Position (i,j) → $<O(n^3), O(1)>$
  - mit dynamischer Programmierung $<O(n^2), O(1)>$

<O(preprocess), O(query)>

# RMQ – Sparse Table

- **Besser:** Sparse Table (ST) Algorithm <O(n·log(n)), O(1)>
- **Idee:** Vorberechnung für alle Teilarrays der Länge $2^k$ (DP)
  - Speichere Ergebnisse in Array $M[0...n-1][0...\log_2(n)]$
    - M[i][j] heißt: Teilarray startet bei i und hat die Länge $2^j$

| 8 | 6 | 9 | 2 | 4 | 7 | 3 | 1 |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |

M[1][0]

M[1][1]

M[1][2]

$M[1][0] = RMQ_a(1,1) = 1$
$M[1][1] = RMQ_a(1,2) = 1$
$M[1][2] = RMQ_a(1,4) = 3$

$$M[i][0]=i$$

$$M[i][j]=\begin{cases} M[i][j-1] & A[M[i][j-1]] \le A[M[i+2^{j-1}][j-1]] \\ M[i+2^{j-1}][j-1] & otherwise \end{cases}$$

M[0...7][0...3]

| $M^{4x8}$ | i=0 | i=1 | i=2 | i=3 | i=4 | i=5 | i=6 | i=7 |
|---|---|---|---|---|---|---|---|---|
| j=0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| j=1 | 1 | 1 | 3 | 3 | 4 | 6 | 7 | - |
| j=2 | 3 | 3 | 3 | 3 | 7 | - | - | - |
| j=3 | 7 | - | - | - | - | - | - | - |

# RMQ – Sparse Table - Query

- **Query $RMQ_A(i,j)$:**
  - Betrachte 2 Blöcke die das Intervall i,j überdecken, deren Minimum ist das Ergebnis
  - Überdeckung existiert immer, denn $|2 \cdot 2^k|$ > Intervall zw. i und j
- **Auswahl über:**

$$k = \lfloor \log_2(j-i+1) \rfloor$$

$$RMQ_A[i][j] = \begin{cases} M[i][k] & A[M[i][k]] \leq A[M[j-2^k+1][k]] \\ M[j-2^k+1][k] & otherwise \end{cases}$$

- **Beispiel:** $RMQ_a(1,6) = 3$ k=2

$$RMQ_A[1][6] = \begin{cases} M[1][2] & A[M[1][2]] \leq A[M[6-2^2+1][2]] \\ M[6-2^2+1][2] & otherwise \end{cases}$$

$$RMQ_A[1][6] = \begin{cases} M[1][2] & A[M[1][2]] \leq A[M[3][2]] \\ M[3][2] & otherwise \end{cases}$$

$$RMQ_A[1][6] = \begin{cases} M[1][2] & A[3] \leq A[3] \\ M[3][2] & otherwise \end{cases}$$

# RMQ – ST Implementierung

```
public static void preprocess(int M[][], int A[]) {
    //initialize M for the intervals with length 1
    for (int i = 0; i < A.length; i++)  M[i][0] = i;
    //compute values from smaller to bigger intervals
    for (int j = 1; (1 << j) <= A.length; j++)
        for (int i = 0; i + (1 << j) - 1 < A.length; i++)
            if (A[M[i][j - 1]] < A[M[i + (1 << (j - 1))][j - 1]])
                M[i][j] = M[i][j - 1];
            else
                M[i][j] = M[i + (1 << (j - 1))][j – 1];
}
```

```
public static int query(int M[][], int A[], int a, int b) {
    int k =  (int) (Math.log(b - a + 1) / Math.log(2));
    b -= (1 << k); b++;
    return A[M[a][k]] <= A[M[b][k]] ? M[a][k] : M[b][k];
}
```

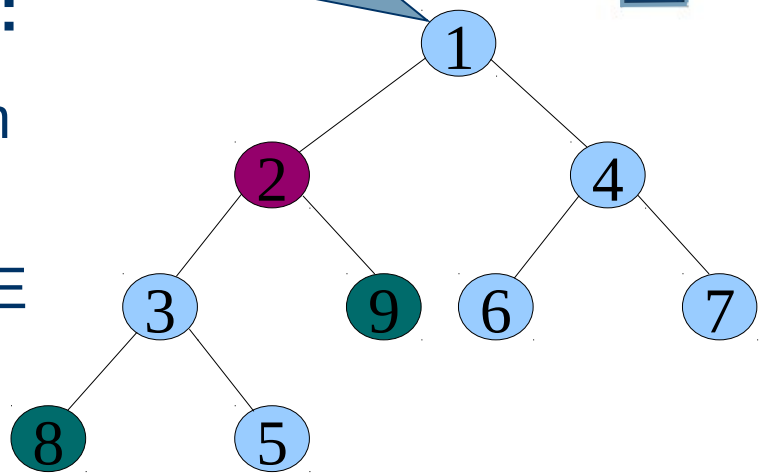
# RMQ und „Lowest Common Ancestor“ (LCA) - Problem

- **Eine Anwendung für RMQ ist LCA:**
  - Man hat einen Baum und möchte für 2 Knoten i, j wissen welcher ihr kleinster gemeinsamer Vorfahre ist
    - dh. es ist ein Knoten gesucht, der beide Knoten i, j als Nachfahre besitzt und gleichzeitig soweit wie möglich von der Wurzel entfernt ist
- Es ist möglich das LCA – Problem auf das RMQ – Problem zu reduzieren → benutze RMQ um LCA zu lösen

# RMQ – LCA - Beispiel

- **$LCA_T(u, v) = E[RMQ_L(H[u], H[v])]$:**
  - E beschreibt Eulerzyklus um Baum
  - L beschreibt Level der Knoten
  - H erster Besuch eines Knotens in E
  - E[1...2·N-1] / L[1...2·N-1] / H[1...N]

Baum darf auch höherer Ordnung sein (z.B. Ternär)




LCA(8, 9) = E[8] = 2

Zwischen erstem / letzten erscheinen eines Knotens nur Kinder / er selbst

E:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 8 | 3 | 5 | 3 | 2 | 9 | 2 | 1 | 4 | 6 | 4 | 7 | 4 | 1 |

If H[u] > H[v]
$RMQ_L(H[v],H[u])$

u v

H:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 12 | 6 | 13 | 15 | 4 | 9 |

L:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 2 | 3 | 2 | 1 | 2 | 1 | 0 | 1 | 2 | 1 | 2 | 1 | 0 |

$RMQ_L(4,9) = 8$

Unabhängig vom Inhalt des Knotens!

# Segment Tree – Allgemein

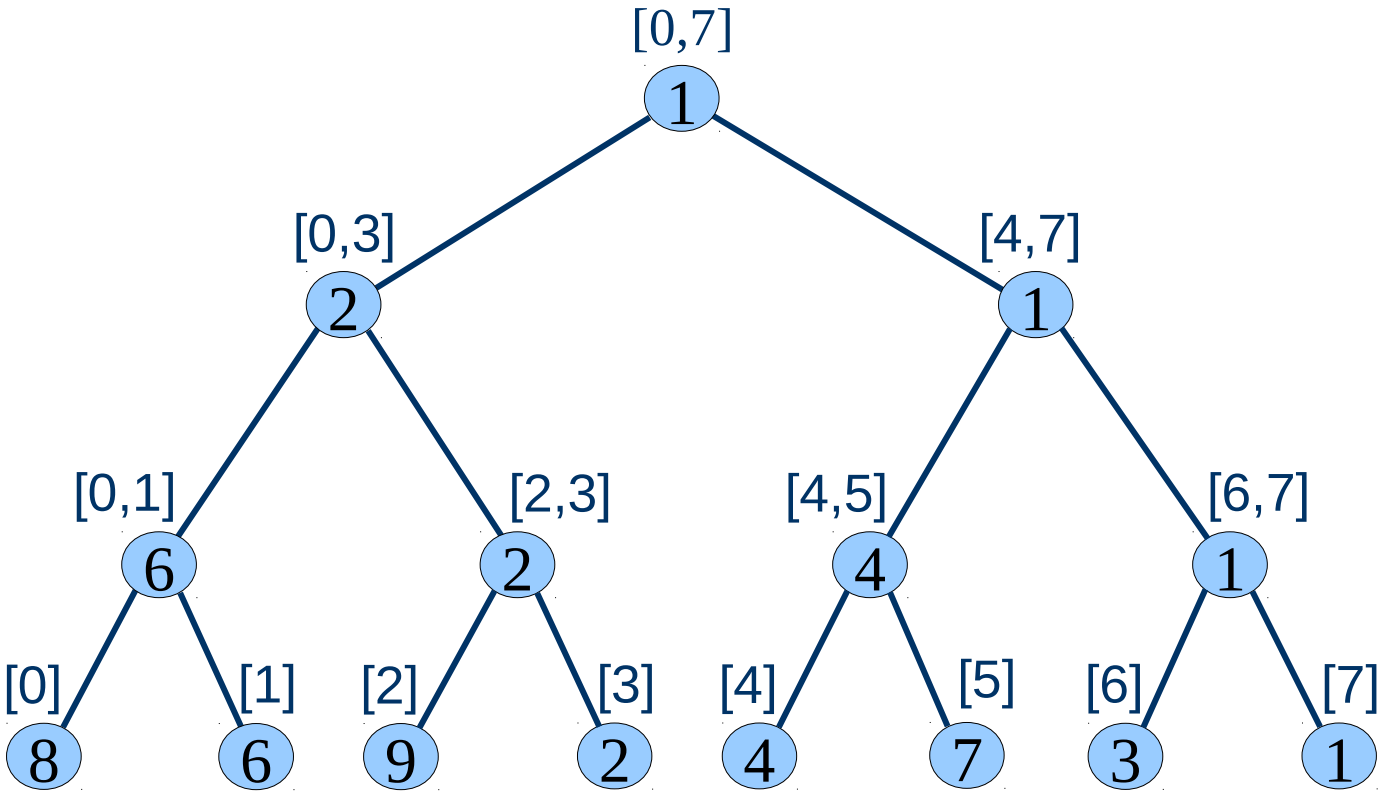
- Eine statische Baum-Datenstruktur um Intervalle / Segmente zu speichern
- Erlaubt die Abfrage welcher der gespeicherten Intervalle einen bestimmen Schlüssel enthält
- Speicher und Konstruktionsaufwand $O(n \cdot \log(n))$
- Suchaufwand $O(\log(n) + k)$ ($k$ = Anzahl der gefundenen Intervalle)
- **Definition:**
  - Ist ein Binärbaum
  - Baum speichert Intervalle, rekursiv definiert:
    - Erster Knoten $[i,j]$, linkes Kind $[i, (i+j)/2]$, rechtes Kind $[(i+j)/2+1, j]$
  - Jedes Blatt hat nur 1 Element (=Elementarintervall, sortiert)
  - Jeder Knoten N hat 2 Nachfolger $(i,j)$, der Intervall von $N = i \cup j$
    - Aber nicht den Intervall seines Parent(N)
  - Höhe des Segmentbaums mit n Elementen ist $\log(n)+1$

# Segment Tree – RMQ/LCA

- Um das RMQ – Problem zu lösen, kann man einen Segmentbaum nutzen, Aufwand <O(n), O(log(n))>
- Die Segmentbaum– Datenstruktur ist dem Heap sehr ähnlich
  - Update / query Operationen in O(log(n))
- Speichern in Array $M[0...2 \cdot 2^{\log(n)+1}]$ wobei M[i] die Position des minimal Wertes des Intervalls ist, der zum Knoten i gehört
  - Wurzel i, Kind links 2i+1, Kind rechts 2i+2

# Segment Tree – RMQ/LCA

| 8 | 6 | 9 | 2 | 4 | 7 | 3 | 1 |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |



# RMQ/LCA – Segment Tree - Implementierung

```
//Initialisierung; Aufwand O(N)
int init(int p, int i, int j, int M[], int A[]) {
    if (i == j) return M[p] = i;
    int p1 = init(2*p+1, i, (i+j)/2);
    int p2 = init(2*p+2, (i+j)/2+1, j);
    return M[p] = (A[p1] <= A[p2]) ? p1 : p2;
}
void init(int M[], int A[]) { init(0, 0, A.length-1, M, A); }
//Liefert minimalen Index eines Minimums im Intervall [a,b],
//0 <= a <= b <= N-1; Aufwand O(log(N))
int query(int p, int i, int j, int M[], int A[], int a, int b) {
    if (j < a || i > b) return -1;
    if (i >= a && j <= b) return M[p];
    int p1 = query(2*p+1, i, (i+j)/2, a, b);
    int p2 = query(2*p+2, (i+j)/2+1, j, a, b);
    if (p1 == -1) return p2;
    return (p2 == -1 || A[p1] <= A[p2]) ? p1 : p2;
}
int query(int M[], int A[], int a, int b) { return query(0, 0, A.length-1, M, A, a, b); }
```

# Übersicht: RMQ

| Algorithmus | Best | Average | Worst |
|---|---|---|---|
| linear | $O(1)$ | $O(j-i)$ | $O(n)$ |
| preprocess | $<O(n^3), O(1)>$ | $<O(n^3), O(1)>$ | $<O(n^3), O(1)>$ |
| preprocess DP | $<O(n^2), O(1)>$ | $<O(n^2), O(1)>$ | $<O(n^2), O(1)>$ |
| Sparse Table (ST) | $<O(n \cdot \log(n)), O(1)>$ | $<O(n \cdot \log(n)), O(1)>$ | $<O(n \cdot \log(n)), O(1)>$ |
| Segment Tree | $<O(n), O(\log(n))>$ | $<O(n), O(\log(n))>$ | $<O(n), O(\log(n))>$ |

$<O(n^2), O(1)>$ = <preprocess time, query time>

# Binary Indexed Tree (BIT) / Fenwick Tree

- Baum-Datenstruktur die zuerst für Datenkompression zuerst genutzt wurde
- wird heute hauptsächlich zum speichern von Frequenzen bzw. kumulativer Frequenz Tabellen genutzt
- kann Probleme wie RMQ lösen, ist aber leichter zu impl. und braucht weniger Speicher
- die Indizes des Arrays in dem der Baum abgelegt ist, sind die Bits die den Inhalt (die Intervalle der Summen) des Baums identifizieren
  - Voraussetzung: Schlüssel sind Integers mit festem Wertebereich
- **Idee:**
  - jede Integer kann als Summe von Zahlen der Form $2^k$ dargestellt werden (Binär Zahlen)
  - ebenso können kumulative Frequenzen als Summe ihrer Subfrequenzen dargestellt werden
  - die Länge der Intervalle dieser Summen wird bestimmt durch das letzte Bit des Indexes

# Binary Indexed Tree (BIT) / Fenwick Tree

bei 011 r=0
bei 010 r=1

- **Notation:**
  - idx ist Index von BIT / r ist Position der letzten 1 in idx (Binary)
  - tree[idx] = Summe von Frequenzen von ($idx-2^r+1$) bis idx
  - idx ist <u>verantwortlich</u> für Indizes von ($idx-2^r+1$) bis idx
  - f[i] – Frequenz mit dem Index i, i = [1...N]
  - c[i] – Kumulative Frequenz für Index i, [f(1) + f(2) + … + f(i) ]
  - tree[i] – Summe der Frequenzen die im Baum mit Index i gespeichert sind

$$\text{tree}[4] = \sum_{i=4-2^2+1}^{4} f(i) = 4$$

$$\text{tree}[6] = \sum_{i=6-2^1+1}^{6} f(i) = 4$$

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| f | 1 | 0 | 2 | 1 | 1 | 3 | 0 | 4 | 2 | 5 | 2 | 2 |
| c | 1 | 1 | 3 | 4 | 5 | 8 | 8 | 12 | 14 | 19 | 21 | 23 |
| tree | 1 | 1 | 2 | 4 | 1 | 4 | 0 | 12 | 2 | 7 | 2 | 11 |

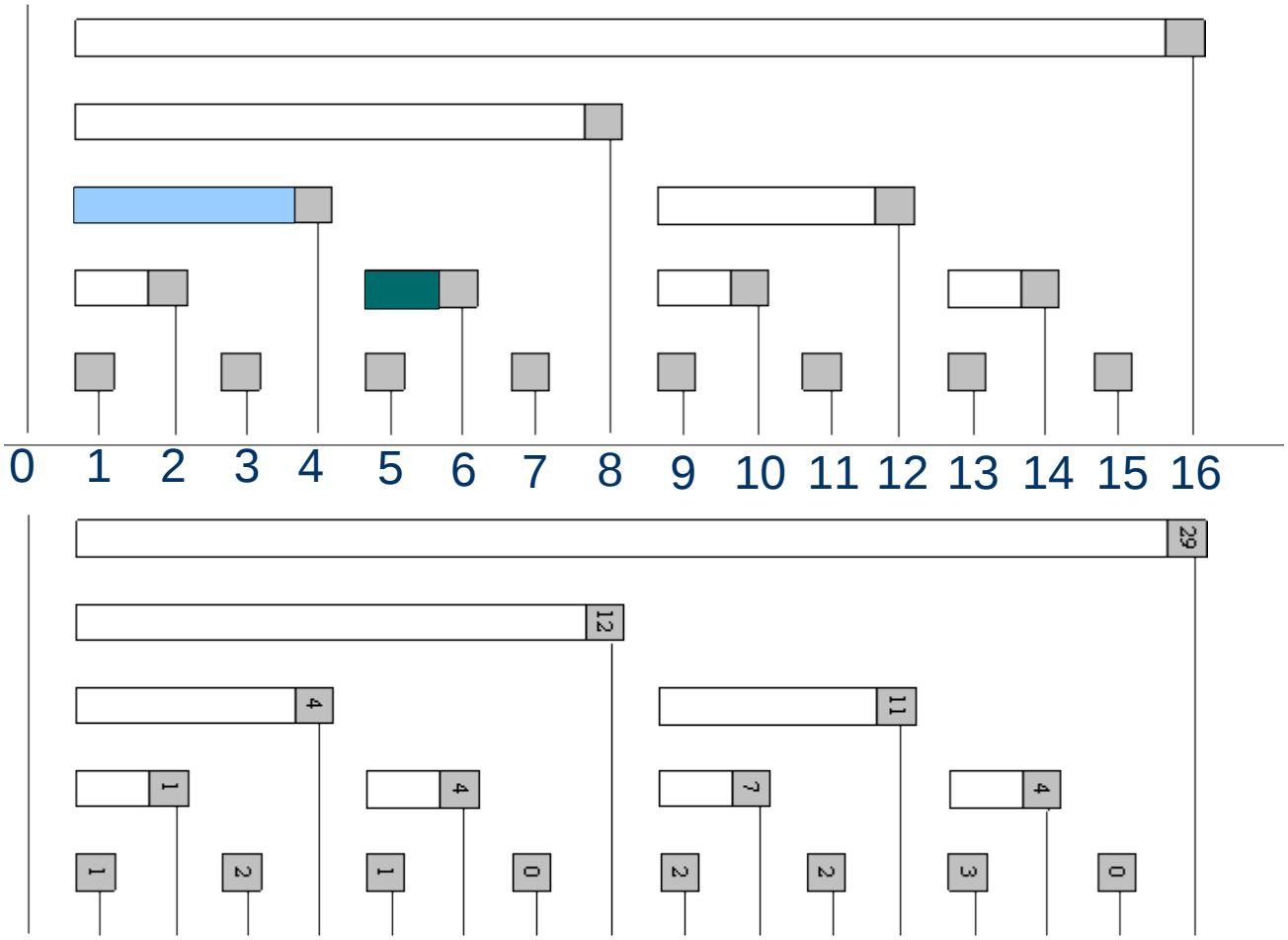
…

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| tree | 1 | 1..2 | 3 | 1..4 | 5 | 5..6 | 7 | 1..8 | 9 | 9..10 | 11 | 9..12 |

responsibility table

# Binary Indexed Tree (BIT) / Fenwick Tree

- **Responsibility:**




| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frequenzen: | (0) | (1) | (0) | (2) | (1) | (1) | (3) | (0) | (4) | (2) | (5) | (2) | (2) | (3) | (1) | (0) | (2) |

# Binary Indexed Tree (BIT) / Fenwick Tree - Implementierung

## Alle Operationen auf einem BIT in O(log(n))

```
int read(int idx){
        int sum = 0;
        while (idx > 0){
                sum += tree[idx];
                idx -= (idx & -idx);
        }
        return sum;
}
```

```
void update(int idx ,int val){
        while (idx <= MaxVal){
                tree[idx] += val;
                idx += (idx & -idx);
        }
}
```

Suche kumulative Freq. mit Index 11 = $1011_{(bin)}$

c[1101] = tree[1011]+tree[1010]+tree[1000]
= 2 + 7 + 12 = 21

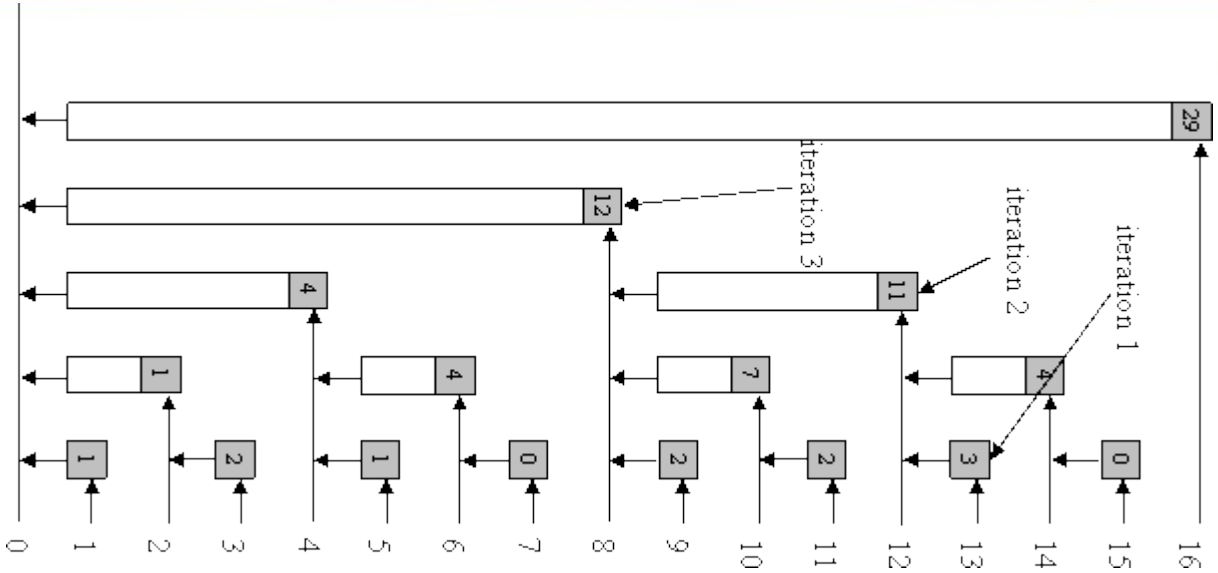
letzte Stelle isolieren: idx & -idx =
1101 & 0011 = 0001
1100 & 0100 = 0100

**<u>Es gibt noch:</u>**

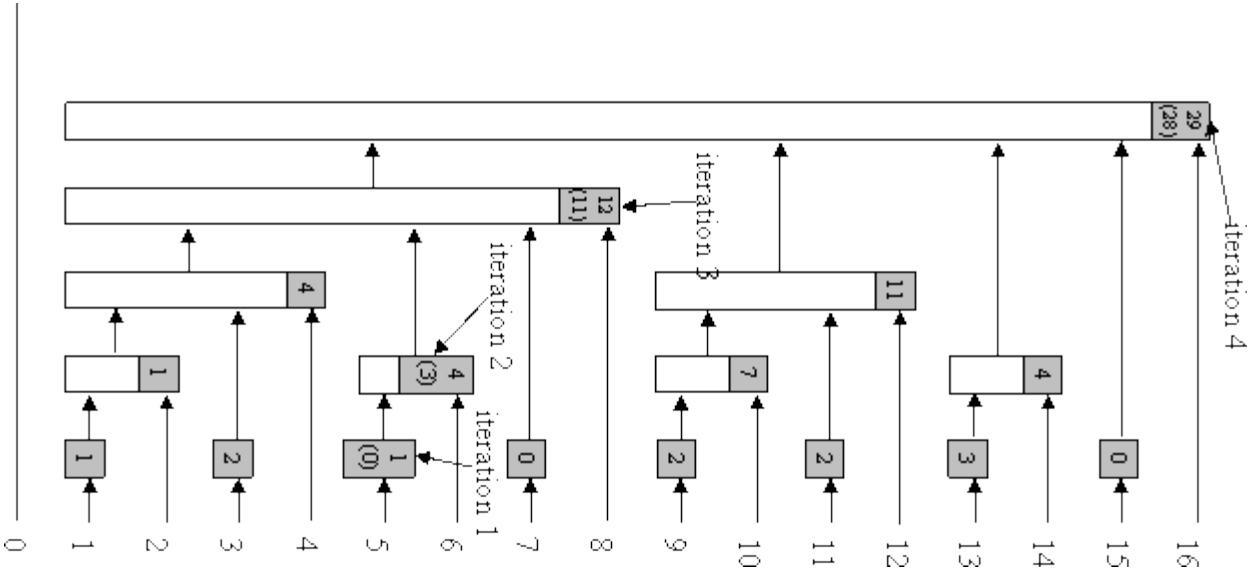
get_cumulative
get_interval
readSingle
Scale (c·f(i))
2D - BIT

# Binary Indexed Tree (BIT) / Fenwick Tree

## Read:




## Update:



# Binary Indexed Tree (BIT) / Fenwick Tree - Implementierung

```
int bit[MAXN+1]
void add(int i, int v) {
        for (; i <= MAXN; i += i & -i) bit[i] += v;        }
int get_cumulative(int i) {
        int res = 0;
        for (; i; i-= i & -i) res += bit[i];
        return res;        }
int get_interval(int a, int b) {
        return get_cumulative(b) – get_cumulative(a-1);        }
int find(int sum) {
        int n = 0, m=1, res = MAXN+1;
        while(2*m <= MAXN) m*= 2;
        while(m) {
                if(bit(n+m) >= sum)  { res = n+m;
                } else {
                n += m; sum -= bit[n]; }
                for (m /= 2; n+m > MAXN; m /= 2);
```

# Bibliotheksfunktionen - C

```
#include <stdlib.h>

// Sortierung
void qsort(void* base,                    // Anfangsadresse
           size_t n,                      // Anzahl Elemente
           size_t size,                   // Größe pro Element
           int (*cmp)(const void*, const void*));

// Binäre Suche
void *bsearch(const void *key,            // zu suchender Schlüssel
              const void *base,           // Anfangsadresse
              size_t n,                   // Anzahl Elemente
              size_t size,                // Größe pro Element
              int (*cmp)(const void *, const void *));
```

Liefert Pointer auf gefundenen Eintrag oder **NULL**, falls nicht enthalten

# Bibliotheksfunktionen - C++

```
#include <algorithm>
// Binäre Suche
bool binary_search(ForwardIterator first, ForwardIterator last,
                                        const LessThanCompareable& val);

// lineare Suche von Min/Max
InputIterator find( InputIterator first,
                    InputIterator last,
                    const EqualityCompareable& val);
// Selektion (benutzt Quickselect)
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                    RandomAcessIterator last);
```

last wenn nicht gefunden

Dort steht danach das gesuchte Element

- **Sortieren mit der STL**
  - Einige Librarys verwenden seit 2000 Introsort, schneller Hybridsortieralgorithmus
    - Startet mit Quicksort, falls Rekursionstiefe > log(n) Heapsort am Ende Insertionsort
    - Wie Q-Select gibt's auch Introspection-selection (Introselect)
    - Vergleichsbasiert / instabil / O(n*log(n))

```
#include <algorithm>
using namespace std;
template <class RandomAccessIterator>
void sort(RandomAccessIterator first,
          RandomAcessIterator last);

template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first,
          RandomAcessIterator last,
          Compare comp);
```

Benutzt den operator< , Operator-Überladung bei Objekten nötig

# Bibliotheksfunktionen - Java

- **Java SE 7 verwendet Timsort für Arrays (Python ab 2.3)**
  - Timsort, schneller Hybridsortieralgorithmus
    - Startet mit Mergesort, später Insertionsort
    - stabil
    - Natürlich (deswegen häufig  weniger Vergleiche als Ω(n*log(n))

```
import java.util.Arrays;
// Binary Search
public static <T> int binarySearch(T[] a, T key,
                                   Comparator<? super T> c);

// sorting
public static void sort(byte[] a);

// linear search for Min/Max
import java.util.Collections;
public static <T> T min(Collection<? extends T> coll,
                        Comperator<? super T> comp);
```

optimierter Quicksort (seltener $O(n^2)$ )

# Ende

Fragen?

# Quellen

- Allgemein
  - Wikipedia (EN)
    - http://en.wikipedia.org/wiki/Sorting_algorithm
    - http://en.wikipedia.org/wiki/Segment_tree
  - Segment Tree
    - http://scvalex.github.com/articles/SegmentTree.html
  - Animationen
    - http://www.sorting-algorithms.com/
- Topcoder.com
  - TernarySearch:

    http://apps.topcoder.com/forums/www.eclipse.org?module=Thread&threadID=670169&start=0&mc=2
  - BinSearch:

    http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=binarySearch
  - RMQ & LCA:

    http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Range_Minimum_Query_%28RMQ%29
  - BIT:

    http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees#prob
- Folien der Vorgänger
  - SS2008 / SS2009 / SS2010