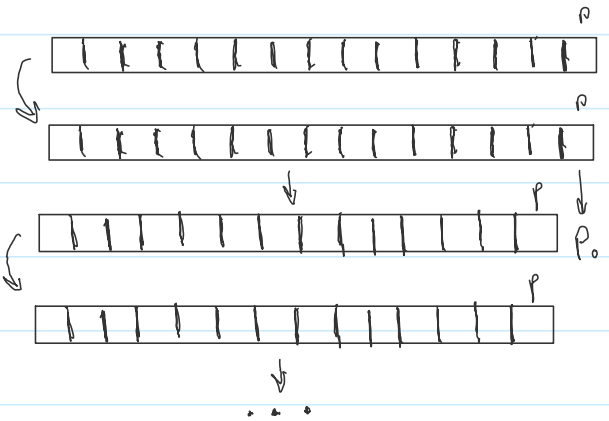
# AuD Hausübung 3

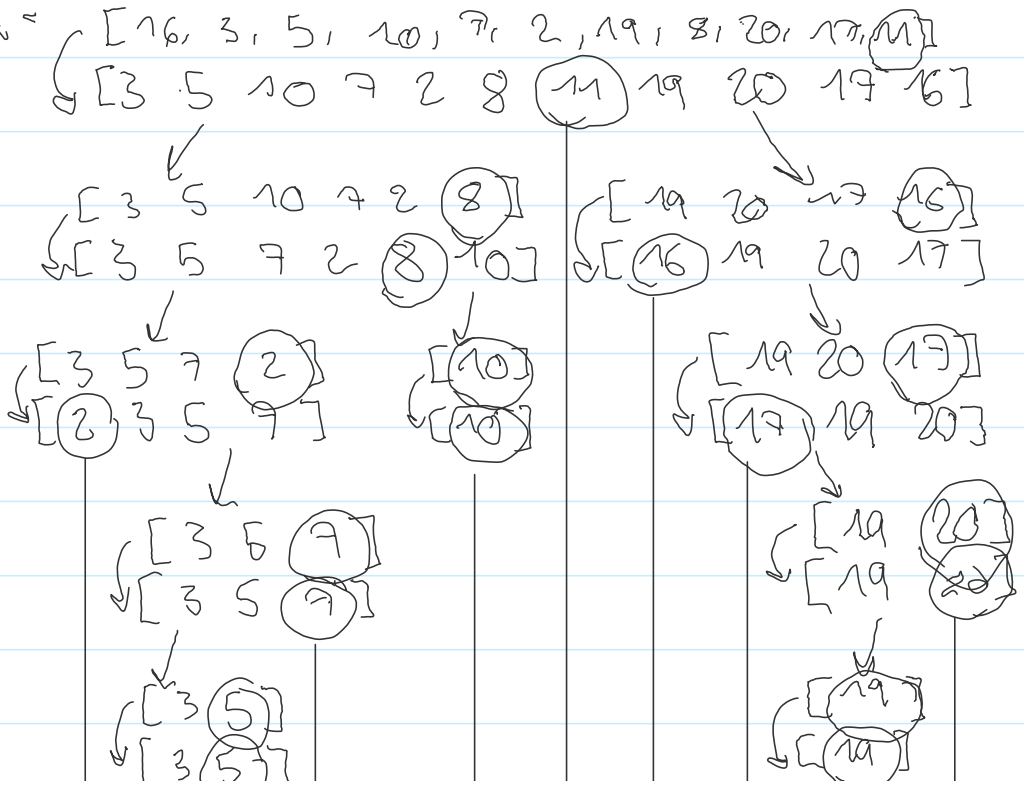
Nicolas Petermann, Julian Imhof, Jonas Franz

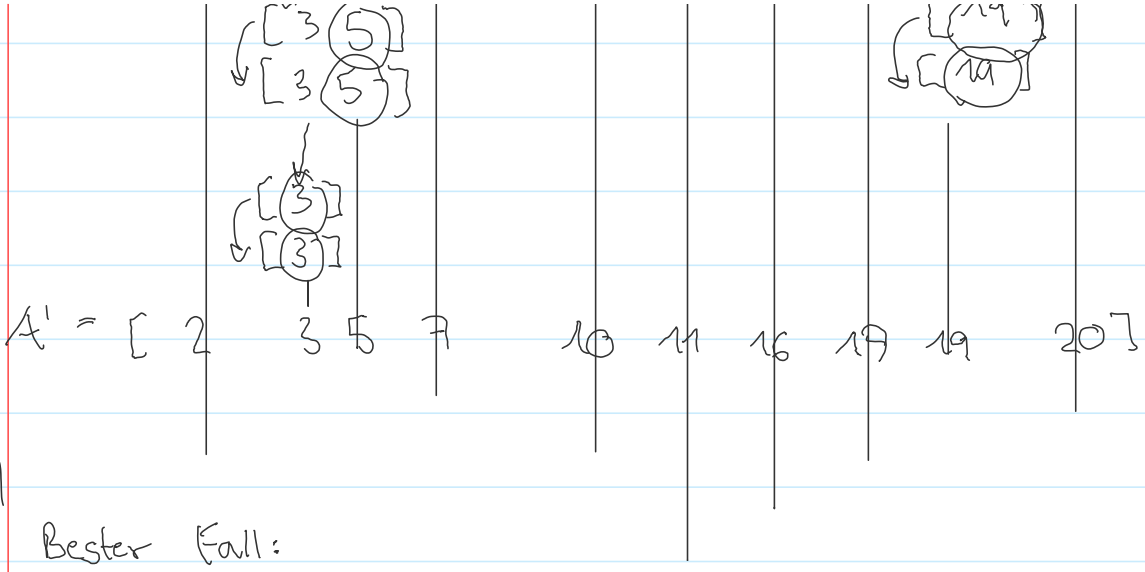
Donnerstag, 16. Mai 2019

H1) Die Laufzeit würde $O(n^2)$ betragen, da in jeder Iteration und jedem Teilarray unabhängig vom gewählten Pivot-Element keine Elemente existieren, welche größer als das Pivot-Element sind. Somit bleibt das Pivot-Element in jeder Iteration an seiner ursprünglichen Stelle, in diesem Fall rechts, was bedeutet, dass sich die Array-Länge pro Iteration sich nur um 1 reduziert. Somit haben wir einen listenähnlichen Rekursionsbaum:




b) A = [16, 3, 5, 10, 7, 2, 19, 8, 20, 17, 11]






c)

Bester Fall:

A = [2, 3, 7, 10, 5, 16, 17, 20, 19, 11]

Anzahl Vergleiche: 36

Schlechtester Fall:

B = [20, 19, 17, 16, 11, 10, 7, 5, 3, 2]

Anzahl Vergleiche: 45

(12)

a)

```
findIndex(A)
    if (A.length == 1)return 0
    if (A.length > 1)
        if (A [0] > A [1])return 0
        if (A [A.length - 1] > A [A.length - 2])return A.length - 1

    int l = 0
    int r = A.length
    while (r - l > 1) do
        int mid = floor((l + r) / 2)
        if (A [mid] > A [mid + 1])r = mid
        else l = mid
    return r
```

Korrektheit:

Schleifeninvariante: zu jedem Zeitpunkt befindet sich der gesuchte Index i innerhalb von l, r, sodass

sich der gesuchte Index i innerhalb von l, r, sodass $l \leq i \leq r$.

Schleifeninvariante gilt vor der Schleife: Da A[l,r] gesamtes Array abdeckt muss k innerhalb von l, r liegen.

Invariante gilt in der Schleife:
In jeder Iteration wird entweder l oder r verschoben. Zunächst wird mid mit $\lfloor (l+r)/2 \rfloor$ berechnet. mid repräsentiert den index in der Mitte zwischen l und r. Per abfrage wird entschieden, ob l oder r verschoben werden soll. Angenommen, der Wert in a an index mid ist größer als der Wert an 'dex mid+1, so muss der gesamte Teilarray A[mid,r] kleiner gleich A[mid] sein. Somit muss der gesuchte index im Bereich A[l, mid] liegen. Sollte dieser Fall eintreten, verschieben wir r zu mid. Somit muss laut oben der gesuchte index im neuen Teilarray liegen.

Invariante gilt nach der Schleife:
Die Schleife terminiert gdw. das Teilarray nur noch aus einem Wert besteht. Oben haben wir gezeigt, dass der gesuchte index das Teilarray nicht verlässt. Also muss es sich bei dem

nicht verlässt. Also muss es sich bei dem letzten Element um den index handeln.

Laufzeit:
Da sich die Länge des Arrays in jeder Iteration halbiert, erhalten wir so eine Laufzeit von $\Theta(\log n)$.

b)

```
findIndex(V)
    if (V.length == 1)return 0
    if (V.length > 1)
        if (V [0] > V [1])return 0
        if (V [V.length - 1] > V [V.length - 2])return
V.length - 1

    int l = 0
    int r = V.length
    while (r - l > 1) do
        int mid = floor((l + r) / 2)
        if (V [mid] > V [mid + 1])r = mid
        else l = mid
    return V [r]
```

c)

```
findIndex(W)
    if (W.length == 1)return 0
    if (W.length > 1)
        if (W [0] > W [1])return 0
        if (W [W.length - 1] > W [W.length - 2])return
W.length - 1

    int l = 0
    int r = W.length
    while (r - l > 1) do
        int mid = floor((l + r) / 2)
        if (W [mid] > W [mid + 1])r = mid
        else l = mid
    return W [r]
```

H3)

a)

a)

## H3

a) reflexiv: $(x_1 \leq x_1) \wedge (y_1 \leq y_1)$

folgt aus Eigenschaften von $\leq$

transitiv: $(x_1 \leq x_2) \wedge (y_1 \leq y_2)$

und $(x_2 \leq x_3) \wedge (y_2 \leq y_3)$

folgt $(x_1 \leq x_3) \wedge (y_1 \leq y_3)$

folgt auch aus Eigenschaften von $\leq$

antisymmetrie: $(x_1 \leq x_2) \wedge (y_1 \leq y_2)$

und $(x_2 \leq x_1) \wedge (y_2 \leq y_1)$

folgt $(x_1 = x_2) \wedge (y_1 = y_2)$

damit auch $P = Q$

Da die Relation reflexiv, transitiv und antisymmetrisch ist folgt, dass sie eine Ordnungsrelation ist.

Da weder $(2 \leq 1) \wedge (1 \leq 2)$

oder $(1 \leq 2) \wedge (2 \leq 1)$

für $P = (2, 1)$ und $Q = (1, 2)$

ist die Relation keine totale Ordnung

b)

```
List<Point> findMaxRec(List<Point> points) {
    if (points.size() == 1)return points;
    else {
        //Divide
        List<Point> p0 = new ArrayList(), p1 = new ArrayList();
        for (int i = 0; i < points.size(); i++) {
            if (i < points.size() / 2)p0.add(points.get(i));
            else p1.add(points.get(i));
        }

        List<Point> m0 = findMaxRec(p0), m1 = findMaxRec(p1);

        //Conquer
        List<Point> max = new ArrayList();
        m0.stream().forEach(x -> max.add(x));
        m1.stream().forEach(x -> max.add(x));

        List<Point> max0 = new ArrayList(max);
        max.stream().forEach(x -> {
            max.stream().filter(y -> !y.equals(x)).forEach(y -> {
                if (x.x <= y.x && x.y <= y.y)max0.remove(x);
            });
        });

        return max0;
    }
}

//Alternative nicht-rekursive Lösung
List<Point> findMax(List<Point> p) {
    List<List<Point>> c = new List(p);

    List<Point> r = new List();
    r.addAll(c.get(0));

    c.get(0).fe(x -> {
        c.get(0).filter(y -> !y.equals(x)).fe(y -> {
            if (x.x <= y.x && x.y <= y.y)r.vpop(x);
        });
    });

    return r;
}
```

c) Schleifeninvariante:

Die Zurückgegebene Liste enthält nur maximale Punkte.

Vor der Schleife:

Bis die Liste nur noch ein Element hat wird

vor der Schleife:
Bis die Liste nur noch ein Element hat wird sie rekursiv geteilt. Bei Rekursionsabbruch ist die Liste einelementig, der punkt ist also maximal.
Die beiden zurückgegebenen Listen enthalten aus dem Vorigen Schritt ebenfalls keine maximalen Punkte.

In der Schleife:
Während jedem Rekursionsschritt werden die zwei zurückgegebenen Listen zusammengeführt, Die einzelnen Listen enthalten nur maximale Punkte, die Vereinte Liste kann aber nun nicht-max. enthalten. Diese werden vor der Rückgabe entfernt. Die Rückgabe besteht also nur aus maximalen Punkten aus den beiden Teillisten.

Nach der Schleife:
Die letzte Rückgabe vereint die erste und zweite Listenhälfte und entfernt wie oben dargestellt nicht-max. Die letzte Rückgabe enthält also ebenfalls keine maximalen Punkte.

Laufzeit: - Liste wird in jeder Iteration halbiert
- Zusammenfügen erzeugt durch doppelte for Schleife
↳ T(n) = { c n=1

$$\hookrightarrow T(n) = \begin{cases} c & n = 1 \\ 2T(n-1) + cn^2 & n > 1 \end{cases}$$

Abschätzung: $\Theta(\log_n \cdot n^2)$