§ Sortieren & Permutationen

```
L = A B C     # Hauptarray sortiert (0. Permutation)
I = 0 1 2     # Indexarray
```

```
L = A C B     # 1. Permutation
I = 0 2 1
```

$\pi : X \to X$ bijective

```
L = B A C     # 2. Permutation
I = 1 0 2
```

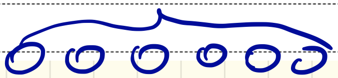
```
L = B C A     # 3. Permutation
I = 2 0 1
```

$|X| = n \Rightarrow \exists\ n!$

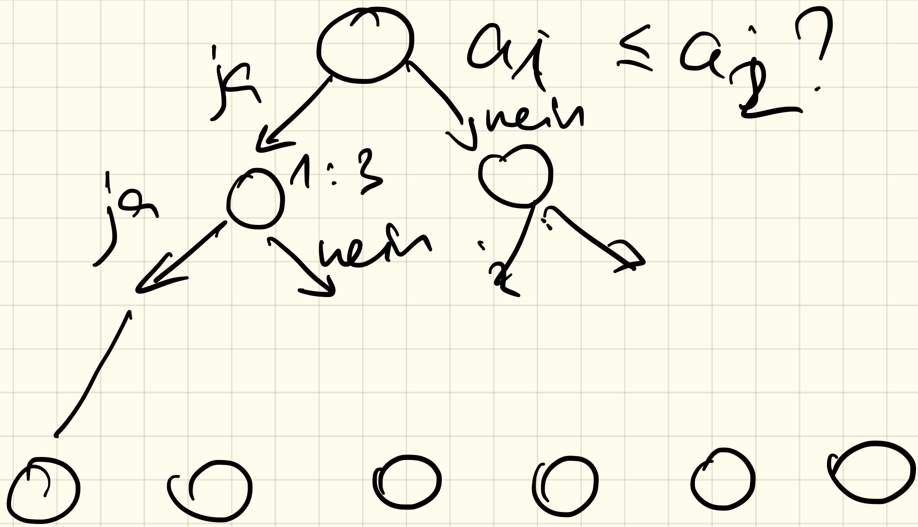
```
L = C A B     # 4. Permutation
I = 1 2 0
```

$|X| = n \quad n! \quad \leq$ Permutationen

```
L = C B A     # 5. Permutation
I = 2 1 0
```



```
def sortByIndexArray(L, I):
    R = [None]*len(L)        # zunächst leeres Ergebnisarray
    for k in  range(len(L)):
        R[k] = L[I[k]]       # Elemente sortiert in R einfügen
    return R
```



1. Wir möchten eine Permutation finden

$\to n!$ Permutationen.

2. Es ist erlaubt nur $a_i \le a_j$ Vergleiche benutzen. Jede Vergleich gibt 2 Antworten: ja / nein.

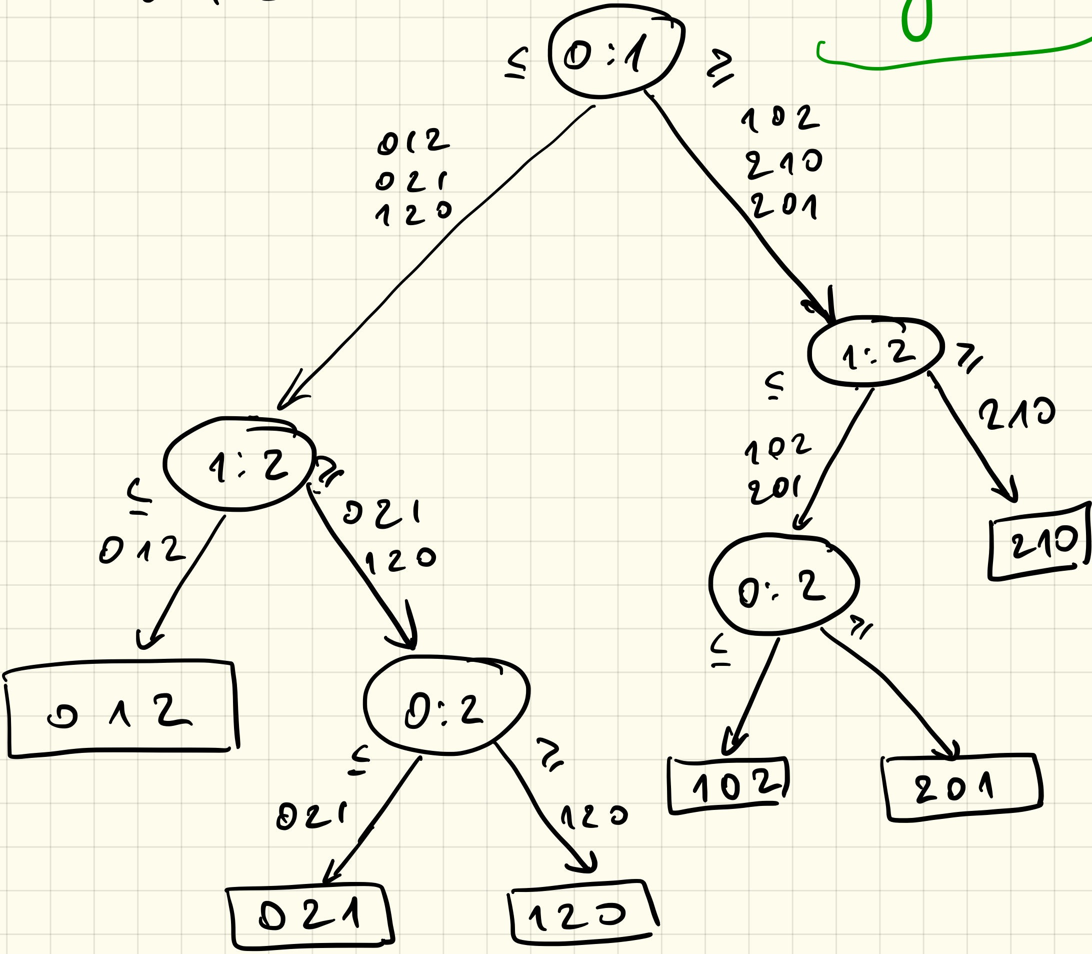
$\Rightarrow$ Entscheidungsprocess $\to$ ein binäres Baum mit $n!$ Blätter.

Bsp: $\Rightarrow$ Projektor

A B C

0 1 2

$d \geq O(\lg(n!))$



Allgemein gilt

$$d \geq \log_2(n!)$$

Wir können die Tiefe am einfachsten durch die *Stirlingsche Näherungsformel für die Fakultät* abschätzen:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n,$$

die asymptitisch für große n gilt. Einsetzen liefert

$$d \geq \log_2(n!) \in \Omega\left(\log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)\right)$$

Der Logarithmus eines Produkts ist gleich der Summe der Logarithmen der einzelnen Faktoren:

$$\Omega\left(\log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)\right) = \cancel{\Omega(\log_2(\sqrt{2\pi}))} + \cancel{\Omega(\log_2(\sqrt{n}))} + \underbrace{\Omega(\log_2(n^n))}_{\Omega(n \log_2 n)} - \underbrace{\Omega(\log_2(e^n))}_{n \log e}$$

Wir vereinfachen die rechte Seite nach den Regeln der O-Notation: nur der am schnellsten wachsende Term bleibt übrig:

$$\Downarrow$$

$$\Omega(n \log_2 n)$$

3 $\Rightarrow$ Höhe des Baums $\log(n!) =$

$= \Omega(n \log n)$.

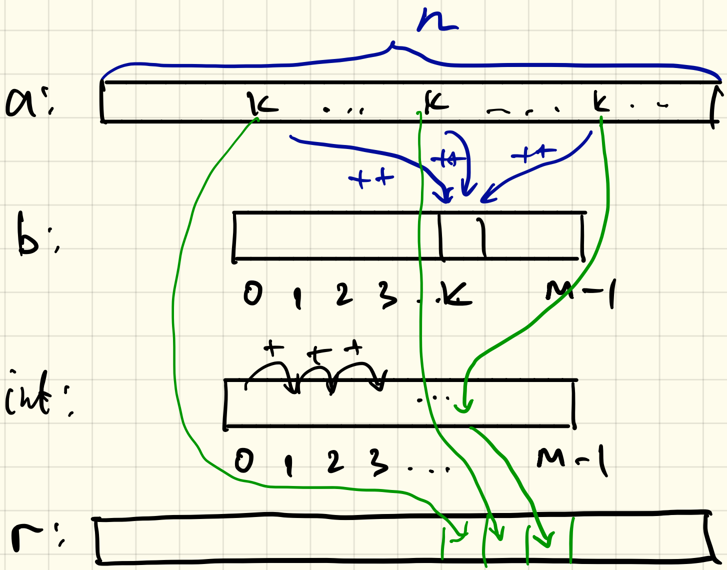
---

§ Digitale Sortierverfahren

§ Counting Sort.

Stable.

$a[j] \in \mathbb{N}, \quad 0 \le a[j] \le M-1, \quad M \in O(n) \; (M < n)$




```
b = [0]*M
for i in range(len(a)):
    b[a[i]] += 1

for i in range(1, M):
    b[i] += b[i-1]

for i in range(len(a)-1,-1,-1)
    b[a[i]] -= 1
    r[b[a[i]]] = a[i]
```

Complexity: $O(n+M)$;

$\Rightarrow$ if $M \in O(n) \Rightarrow O(n)$

§ Radix Sort

$M \leq 999$, $n = 20$

$i = 2\ 1\ 0$, $d = 2$

for i in range(d)
  Stable Sort w.r.t. i

Bsp:

	24	76	12	41
$d=0$:	41	12	24	76
$d=1$:	12	24	41	76

Einfachste Implementierung:

$$\overbrace{3 \quad 2 \quad 1 \quad 0}^{k=4} = i$$

$$8 \quad 8 \quad 8 \quad 8 \text{ bits}$$

$8 = m$

M

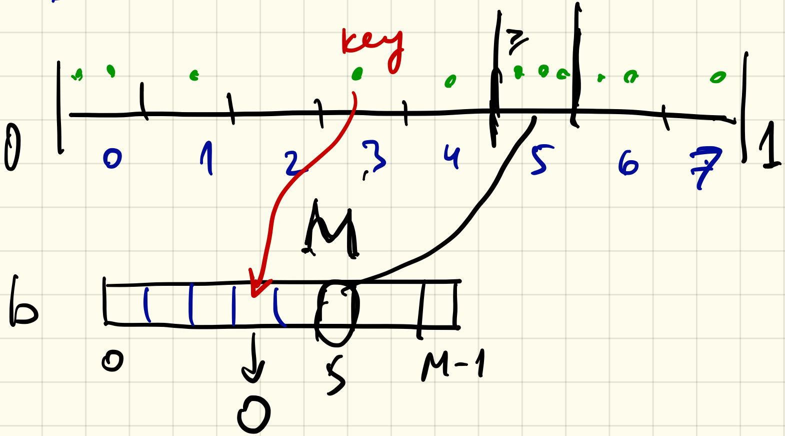
Komplexität: $O((m+n)\cdot k) = O((m+n)\cdot \log_m M)$

$n \cdot \log_2 M$

$\log n$

§ Bucket Sort

Annahme: Stätige Gleichverteilung auf $[0, 1]$



1. Wähle $M \approx n/d$,

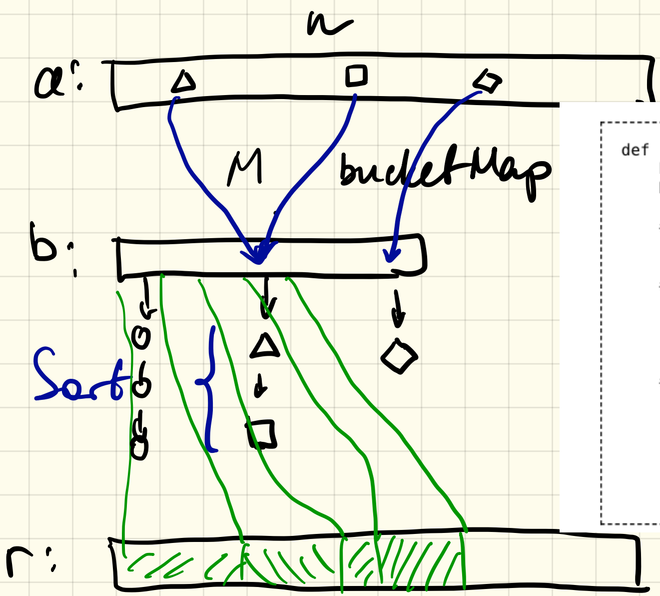
   $1 \leq d \leq 10$

2. Teile $[0,1]$ auf M Teile

3. Definiere Funktion:

```
def bucketMap(key, M)
    return int(key*M)
```

§ Bucket Sort




```
def bucketMap(key, M)
    return int(key*M)
```

```
def bucketSort(a, bucketMap, d):
    N = len(a)
    M = int(N / float(d))  # Anzahl der Buckets festlegen

    # M leere Buckets erzeugen
    buckets = [[] for k in range(M)]

    # Daten auf die Buckets verteilen
    for k in range(len(a)):
        index = bucketMap(a[k].key, M) # Bucket-Index berechnen
        buckets[index].append(a[k])    # a[k] im passenden Bucket einfügen

    # Daten sortiert wieder in a einfügen
    start = 0                          # Anfangsindex des ersten Buckets
    for k in range(M):
        insertionSort(buckets[k])      # Daten innerhalb des aktuellen Buckets sortiere
        end = start + len(buckets[k])  # Endindex des aktuellen Buckets
        a[start:end] = buckets[k]      # Daten an der richtigen Position in a einfügen
        start += len(buckets[k])       # Anfangsindex für nächsten Bucket aktualisieren
```

Complexity?

$n + M \cdot \mathcal{O}(\mathrm{sort}(d))$

$\sim \mathcal{O}(n + M \cdot d^2) \Rightarrow \mathcal{O}(n)$

if $Md^2 \in \mathcal{O}(n)$