Das geht explizit in Linearzeit:

In Zelle Nr.	steht	gewünscht
1	$\pi(1)$	$\pi^{-1}(1)$
2	$\pi(2)$	$\pi^{-1}(2)$
3	$\pi(3)$	$\pi^{-1}(3)$
4	$\pi(4)$	$\pi^{-1}(4)$
5	$\pi(5)$	$\pi^{-1}(5)$

Also: Schreibe in die Speicherzelle

Nummer i den Wert $\pi^{-1}(i)$

Äquivalent schreibe in die Speicherzelle

Nummer $\pi(i)$ den Wert i.

Also in Pseudocode für neuen Array IP[.]

```
FOR ( i=1 TO n ) DO {
        IP[ P[i] ] := i
}
```

(dabei P[.]: Permutation, IP[.]: Inverse Permutation)

(2) Kennt man keine explizite Nummerierung, sondern kann die Objekte nur paarweise vergleichen (Balkenwaage o.ä.!), so wird es schwerer!

Beobachtung:

(ii) Vergleicht man zwei Objekte (x_i und x_j) in der Anordnung, so teilt man die Menge der ~~Permutationen~~ Permutationen in zwei Mengen:

(≥) Die Permutationen, bei denen x_i vor x_j steht;

(≤) die Permutationen, bei denen x_j vor x_j steht.

Wie schwerwiegend ist das?

Wenn man Pech hat, ist hinterher die größere Menge übrig, also kann man auch bei clevererer Auswahl bestenfalls eine Reduktion um einen

Faktor 2 erreichen!

Satz 5.1

(1) Die Objekte $1, \dots, n$ kann man in Linearzeit sortieren, wenn man Arrays verwenden und in $O(1)$ direkt auf diese zugreifen darf.

(2) Für n Objekte $x_1, \dots, x_n$ benötigt man zum Sortieren mindestens $\Omega(n \log n)$, wenn man die Objekte nur paarweise vergleichen kann.

Beweis:

(1) Siehe oben!

(2) (a) Am Anfang hat man $n! = n \cdot (n-1) \cdot \dots \cdot 3 \cdot 2 \cdot 1$ viele mögliche Permutationen.

(b) Jeder Vergleich teilt die Menge der verbliebenen Permutationen in zwei Teilmengen

(c) Im schlechtesten Falle bleibt die größere Teilmenge übrig, $\frac{n!}{2}$

(d) Man erreicht also bestenfalls eine Halbierung.

(e) Bis man eine eindeutige Permutation sicher identifiziert hat, braucht man also mindestens

$$\log_2(n!)$$

Vergleiche.

(f)
$$\log_2(n!) = \sum_{i=1}^{n} \log_2 n$$

$$\geq \sum_{i=\frac{n}{2}}^{n} \log_2 n$$

$$\geq \frac{n}{2}\left(\log_2 \frac{n}{2}\right)$$

$$= \frac{n}{2}\left(\log_2 n - 1\right)$$

$$\in \Omega(n \log n)$$

□

5.3 Mergesort

Wir wollen folgende Zahlen sortieren (aufsteigend nach Größe):

A: 8 3 9 6 3 11 7 12

Idee: ① Finde Minimum in A.
② Kopiere Minimum nach B (Array)
③ Lösche Minimum in A

Durchlauf

1 B: 3

2 B: 3 3

3 B: 3 3 6

⋮

8 B: 3 3 6 7 8 9 11 12

Laufzeit?

Speicherplatz?

Verwaltet man einen Pointer auf das Ende von B und hat man einen Pointer auf das aktuelle Minimum können ② und ③ in $O(1)$ ausgeführt werden.

Für ① braucht man im schlechtesten Fall im

1. Durchlauf $n-1$ Schritte / Vergleiche
2. " $n-2$ "
:
i. " $n-i$ "

$\Rightarrow$ Insgesamt $\sum_{i=1}^{n} n-i = n^2 - \frac{n}{2}\cdot(n+1) = n^2 - \frac{n^2}{2} - \frac{n}{2}$

$= \frac{n^2}{2} - \frac{n}{2} \in O(n^2)$

Vergleiche

Satz 5.1 gibt eine untere Schranke für die Anzahl der Vergleiche von $\Omega(n \log n)$.

Speicherplatz? 2. A

↳ Auch das geht besser: „In-Place-Sorting"

Zurück zur Laufzeit:

Der Algorithmus „Mergesort" kommt mit $O(n \log n)$ Vergleichen aus. Dabei wird das algorithmische Prinzip „Divide and Conquer" (Teile und herrsche) verwendet.

Zunächst am obigen Beispiel:

8 3 9 6 | 3 11 7 12

Teile: 8 3 9 6 ; 3 11 7 12

Teile: 8 3 | 9 6 ; 3 11 | 7 12

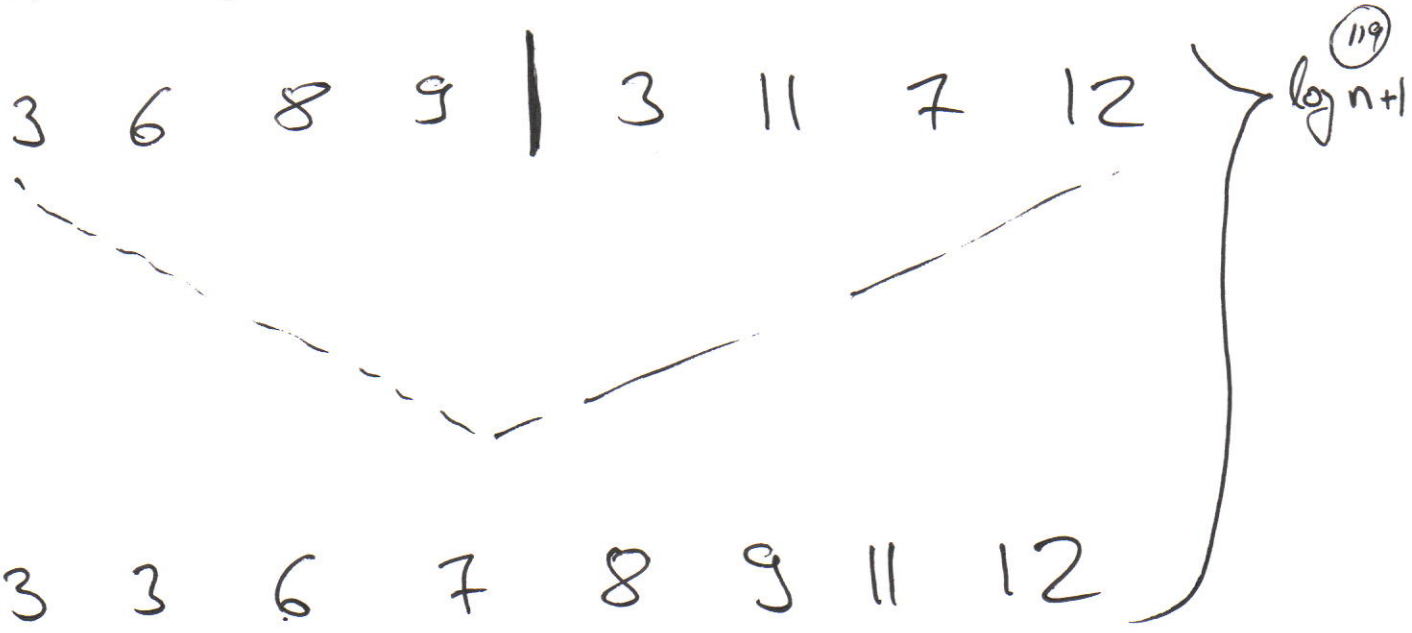
Teile: 8 | 3 | 9 | 6 | 3 | 11 | 7 | 12

Nun kann jedes (dieser 8) Teilproblem in $O(1)$ gelöst werden.

Die Teillösungen müssen nur noch zusammengefügt werden.

z.B. Teillsg. 1. Prob. 8
" 2. " 3
} → 3 | 8

3 8 | 6 9 | 3 11 | 7 12



Divide-and-Conquer-Algorithmen bestehen aus 3 wesentlichen Schritten:

Divide: Problem in Teilprobleme aufteilen

Conquer: Rekursiv die Teilprobleme lösen, wenn sie „klein genug" (z.B. in $O(1)$ lösbar) sind.

Combine: Teillösungen zur Gesamtlösung vereinigen

Algorithmus 5.2: Mergesort(A, p, r)

Input: Subarray von $A = [1, \dots n]$ der bei p beginnt und bei r endet ($A[p \dots r]$)

Output: Den sortierten Subarray

```
1  IF  p < r  THEN
2      q ← ⌊(p+r)/2⌋        ← Divide
3      Mergesort(A, p, q)
4      Mergesort(A, p+1, r)  } ← Conquer
5      Merge(A, p, q, r)     ← Combine
```

- Sortieren von $A = (A[1], \ldots, A[n])$: Mergesort$(A, 1, n)$
- Solange $p < r$ wird geteilt/halbiert.
- Falls $p \geq r$ besteht der Subarray nur noch aus einem Element und ist damit sortiert.

Noch zu klären:

- $\lfloor x \rfloor$ Gauß-Klammer

  $\lfloor x \rfloor = \max \{ k \in \mathbb{Z} : k \leq x \}$

  z.B. $\lfloor 4{,}3 \rfloor = 4$, $\lfloor -2{,}3 \rfloor = -3$, $\lfloor 5 \rfloor = 5$

- Merge(A, p, q, r)

  ↳ Folie

  ↳ ausführlich am Freitag in der 90. Übg