Lemma 2

$a \geq 1$, $b > 1$, $n = b^k$, $f \geq 0$, $g(n) = \sum_{\ell=0}^{k-1} a^\ell f\left(\frac{n}{b^\ell}\right)$. Dann gilt:

1. *Falls $f(n) = \mathcal{O}\left(n^{\log_b a - \epsilon}\right)$, dann gilt $g(n) = \mathcal{O}\left(n^{\log_b a}\right)$;*
2. *falls $f(n) = \Theta\left(n^{\log_b a}\right)$, dann gilt $g(n) = \Theta\left(n^{\log_b a} \log n\right)$;*
3. *falls $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ und es ein $c < 1$ gibt, sodass $af\left(\frac{n}{b}\right) \leq cf(n)$, dann gilt $g(n) = \Theta(f(n))$.*

2.Fall: $g(n) = \Theta(h(n))$ mit

$$h(n) = \sum_{\ell=0}^{k-1} a^\ell \left(\frac{n}{b^\ell}\right)^{\log_b a} = \sum_{\ell=0}^{k-1} a^\ell \left(\frac{n^{\log_b a}}{a^\ell}\right)$$
$$= \Theta\left(n^{\log_b a} \log n\right)$$

3.Fall: $g(n) = \Omega(f(n))$, $f\left(\frac{n}{b}\right) \leq \frac{c}{a} f(n)$
Folglich

$$
\begin{aligned}
g(n) &= \sum_{\ell=0}^{\log_b n - L - 1} a^\ell f\left(\frac{n}{b^\ell}\right) + \sum_{\ell=\log_b n - L}^{\log_b n - 1} a^\ell f\left(\frac{n}{b^\ell}\right) \\
&\leq \sum_{\ell=0}^{\log_b n - L - 1} c^\ell f(n) + \mathcal{O}\left(a^{\log_b n}\right) \\
&\leq \sum_{\ell \geq 0} c^\ell f(n) + \mathcal{O}\left(a^{\log_b n}\right) = \frac{f(n)}{1-c} + \mathcal{O}\left(a^{\log_b n}\right) \\
&= \mathcal{O}\left(f(n)\right),
\end{aligned}
$$

da $a^{\log_b n} = b^{\log_b n \log_b a} = n^{\log_b a}$. □

Lemma 3

$a \geq 1$, $b > 1$, $n = b^k$, $f \geq 0$,

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1, \\ aT\left(\frac{n}{b}\right) + f(n) & \text{falls } n = b^k. \end{cases}$$

Dann gilt:

1. *Falls* $f(n) = \mathcal{O}\left(n^{\log_b a - \epsilon}\right)$, *dann gilt* $T(n) = \mathcal{O}\left(n^{\log_b a}\right)$;
2. *falls* $f(n) = \Theta\left(n^{\log_b a}\right)$, *dann gilt* $T(n) = \Theta\left(n^{\log_b a} \log n\right)$;
3. *falls* $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ *und es ein* $c < 1$ *gibt, sodass* $af\left(\frac{n}{b}\right) \leq cf(n)$, *dann gilt* $T(n) = \Theta(f(n))$.

Beweis: Nach Lemma 1 gilt $T(n) = \Theta\left(n^{\log_b a}\right) + g(n)$. □

Die exakte Rekursion: Betrachten wir

(1) $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n)$,

(2) $T(n) = aT\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n)$.

Alle Abschätzungen im Beweis beruhten auf Abschätzungen der Form

$$f(n) \leq Cn^D \text{ oder } f(n) \geq cn^D.$$

Wegen $\left\lfloor \frac{n}{b} \right\rfloor \leq \frac{n}{b}$ gilt

$$f\left(\left\lfloor \frac{n}{b} \right\rfloor\right) \leq C\left\lfloor \frac{n}{b} \right\rfloor^D \leq C\left(\frac{n}{b}\right)^D,$$

was obere Schranken für (2) liefert.
Analog bekommt man untere Schranken für (1).

Obere Schranken für (1): Wir setzen

$$n_0 := n, \quad n_1 := \left\lceil \frac{n}{b} \right\rceil, \quad n_2 := \left\lceil \frac{\left\lceil \frac{n}{b} \right\rceil}{b} \right\rceil, \ldots$$

also

$$n_j = \begin{cases} n & \text{für } j = 0, \\ \left\lceil \frac{n_{j-1}}{b} \right\rceil & \text{für } j > 0. \end{cases}$$

Daraus folgt

$$n_1 \leq \frac{n}{b} + 1, \quad n_2 \leq \frac{\frac{n}{b} + 1}{b} + 1 = \frac{n}{b^2} + \frac{1}{b} + 1,$$

$$n_\ell \leq \frac{n}{b^\ell} + \sum_{i=0}^{\ell-1} \frac{1}{b^i} < \frac{n}{b^\ell} + \frac{b}{b-1}$$

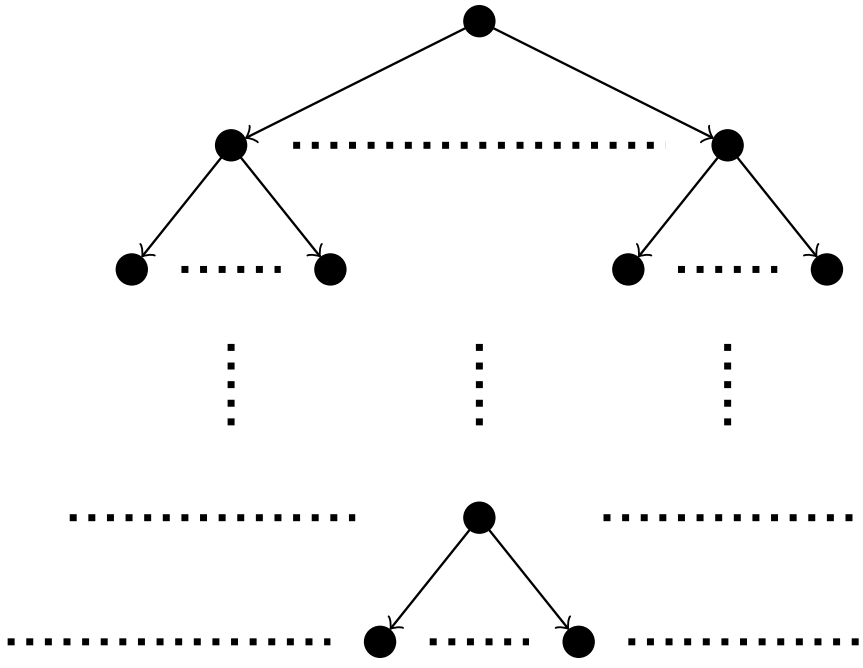
Für $\ell = k := \lfloor \log_b n \rfloor$ ergibt sich

$$n_k < \frac{n}{b^k} + \frac{b}{b-1} \leq \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} = b + \frac{b}{b-1} = \mathcal{O}(1).$$

Divide & Conquer

Sukzessives Einsetzen analog zu Lemma 1 ergibt

$$T(n) = \Theta\left(n^{\log_b a}\right) + \underbrace{\sum_{j=0}^{k-1} a^j f(n_j)}_{=:g(n)}.$$



$f(n)$

$af(n_1)$

$a^2 f(n_2)$

$a^{k-1} f(n_{k-1})$

$a^k = a^{\lfloor \log_b n \rfloor} = b^{\log_b a \lfloor \log_b n \rfloor}$

1.Fall: $f(n) = \mathcal{O}\left(n^{\log_b a-\varepsilon}\right)$, $g(n) = \sum_{j=0}^{k-1} a^j f(n_j)$, Wegen $j \leq \lfloor \log_b n \rfloor$ gilt $\frac{b^j}{n} \leq 1$ und daher

$$
\begin{aligned}
f(n_j) &\leq c n_j^{\log_b a-\varepsilon} \\
&\leq c \left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon} \left(1 + \frac{b}{b-1} \cdot \underbrace{\frac{b^j}{n}}_{\leq 1}\right)^{\log_b a-\varepsilon} \\
&= \mathcal{O}\left(\frac{n^{\log_b a-\varepsilon}\, b^{\varepsilon j}}{a^j}\right) \cdot \mathcal{O}(1) \\
\Longrightarrow\ & g(n) = \mathcal{O}\left(n^{\log_b a}\right)
\end{aligned}
$$

Divide & Conquer

2.Fall: $f(n) = \Theta(n^{\log_b a})$, $g(n) = \sum_{j=0}^{k-1} a^j f(n_j)$,
Wegen $j \leq \lfloor \log_b n \rfloor$ gilt $\frac{b^j}{n} \leq 1$ und daher

$$
\begin{aligned}
f(n_j) &\leq c n_j^{\log_b a} \leq c \left(\frac{n}{b^j} + \frac{b}{b-1} \right)^{\log_b a} \\
&= \left(\frac{n}{b^j} \right)^{\log_b a} \left(1 + \frac{b}{b-1} \frac{b^j}{n} \right)^{\log_b a} \\
&= \mathcal{O}\left(\frac{n^{\log_b a}}{a^j} \right) \cdot \mathcal{O}(1) \implies g(n) = \mathcal{O}\left(n^{\log_b a} \log n \right)
\end{aligned}
$$

Divide & Conquer

3.Fall: $f(n) = \Omega\left(n^{\log_b a} + \varepsilon\right)$, $g(n) = \sum_{j=0}^{k-1} a^j f(n_j)$,
$a\, f\left(\left\lfloor \frac{n}{b} \right\rfloor\right) \leq c\, f(n)$.
Daraus folgt

$$a^j f(n_j) \leq c^j f(n)$$

und daher $T(n) = \Theta(n^{\log_b a}) + \mathcal{O}\left(f(n)\right)$.

Untere Schranke: analog. □

5) Probabilistische Analyse und randomisierte Algorithmen

Das Bewerberproblem

Ein Headhunter schickt Personen $K_1, K_2, \ldots, K_n$ (in dieser Reihenfolge), die sich um eine Stelle bewerben.
Kosten: Interview, Headhunter, Abfertigungen, ...

Strategie: Hire and Fire, d.h., wenn K_i besser ist als $K_1, K_2, \ldots, K_{i-1}$, dann K_i einstellen!

Algorithm HIRE(n)

```
1: best := 0
2: for i = 1 to n do
3:     interviewe K_i
4:     if K_i > best then
5:         kündige K_best und stelle K_i ein
6:         best := i
7:     end if
8: end for
```

Probabilistische Analyse

Kosten:

- C_I ... Interviewkosten
- C_H ... Einstellungskosten

$m =$ Anzahl der eingestellten Personen,
Gesamtkosten: $C_I n + C_H m$.

Analyse:

- Worst-case: $m = n$ $\quad \rightsquigarrow$ Kosten: $(C_I + C_H)n$
- Average-case: m ist Zufallsvariable (Wahrscheinlichkeitsmodell!)

Average-case

Annahme: $\mathrm{rg}(K_1), \ldots, \mathrm{rg}(K_n)$ is zufällige Permutation $\pi \in S_n$.
$X_n =$ Anzahl der eingestellten Personen (Zufallsvariable),
wir suchen $\mathbb{E}X_n$.

Sei $X_{n,i} = \mathbb{1}_{[K_i \text{ wird eingestellt}]}$; dann

$$X_n = \sum_{i=1}^{n} X_{n,i}, \qquad \mathbb{E}X_{n,i} = \mathbb{P}\{X_{n,i} = 1\} = \frac{1}{i}$$

und daher

$$\mathbb{E}X_n = \sum_{i=1}^{n} \frac{1}{i} = \log n + \mathcal{O}(1).$$

Randomisierte Algorithmen

Problem: Sind die Eingabedaten zufällig?

Lösung: Randomisieren.

Bewerberproblem: Eingabe zuerst permutieren.

Notwendig: Weg zum Erzeugen einer zufälligen Permutation.

Permutieren durch Sortieren:

$A[i]$ wird zufällige Priorität zugewiesen, dann A nach Priorität sortieren.

Algorithm PERM-BY-SORT(A)

1: $n := |A|$
2: Sei $P[1 \ldots n]$ ein Array
3: **for** $i = 1$ to n **do**
4: $\quad P[i] :=$RANDOM$(1, n^3)$ %Bem.: Pseudozufallszahl zwischen 1 und n^3
5: **end for**
6: SORT(A) (bzgl P)

Bem.: Es gilt $\mathbb{P}\{\forall i, j \text{ mit } i \neq j : P[i] \neq P[j]\} > 1 - \frac{1}{2n}$.

Satz

Falls $P[1..n]$ paarweise verschiedene Einträge enthält, dann ist die durch $\mathrm{PERM\text{-}BY\text{-}SORT}(A)$ *ausgegebene Permutation eine zufällige Permutation.*

Beweis: Sie $E_i = [\mathrm{rg}(P[i]) = i]$. Bekanntlich gilt

$$\mathbb{P}\{A \mid B\} = \frac{\mathbb{P}\{A \cap B\}}{\mathbb{P}\{B\}}, \quad \mathbb{P}\{A \mid B \cap C\} = \frac{\mathbb{P}\{A \cap B \cap C\}}{\mathbb{P}\{B \cap C\}}, \quad \textit{usw.}$$

und daher

$$\mathbb{P}\left\{\bigcap_{i=1}^{n} E_i\right\} = \mathbb{P}\{E_1\} \prod_{j=2}^{n} \mathbb{P}\left\{E_j \middle| \bigcap_{k=1}^{j-1} E_k\right\}$$
$$= \frac{1}{n} \cdot \frac{1}{n-1} \cdots \frac{1}{1} = \frac{1}{n!}.$$

In-place-Permutation

Knuth-Shuffle oder Fisher-Yates-Algorithmus:

Algorithm FISHER-YATES(A)

```
1: n := |A|
2: for i = 1 to n do
3:     A[i] ↔ A[RANDOM(i, n)]
4: end for
```

Satz

Die Ausgabe $[A[\pi[1]], \ldots, A[[\pi[n]]]$ *von* FISHER-YATES*(A) ist eine zufällige Permutation von* $A[1..n]$.

Beweis: Wir verwenden die Schleifeninvariante
„Vor der i-ten Iteration der Schleife gilt: Wenn $(b_1, \ldots, b_{i-1})$ eine Anordnung von $i-1$ Elementen aus $A[1..n]$ ist, dann haben wir

$$\mathbb{P}\{A[1..i-1] = (b_1, \ldots, b_{i-1})\} = \frac{(n-i+1)!}{n!}.\text{“}$$

Initialisierung: Wenn $i = 1$, dann $\mathbb{P}\{A[\,] = (\,)\} = 1$.

„Vor der i-ten Iteration der Schleife gilt: Wenn $(b_1, \ldots, b_{i-1})$ eine Anordnung von $i-1$ Elementen aus $A[1..n]$ ist, dann haben wir

$$\mathbb{P}\{A[1..i-1] = (b_1, \ldots, b_{i-1})\} = \frac{(n-i+1)!}{n!}.\text{“}$$

Aufrechterhaltung: Vor i-ter It. gelte Schleifeninvariante.
Nach i-ter It.: $(b_1, \ldots, b_i)$; in der i-ten It.: $A[i] := b_i$

$$E_1 = [\text{nach } i-1 \text{ Iterationen gilt } A[1..i-1] = (b_1, \ldots, b_{i-1})]$$
$$E_2 = [i\text{-te Iteration: } A[i] := b_i]$$
$$\mathbb{P}\{E_1\} = \frac{(n-i+1)!}{n!},$$
$$\mathbb{P}\{E_1 \cap E_2\} = \mathbb{P}\{A[1..i] = (b_1, \ldots b_i)\} = \mathbb{P}\{E_2 \mid E_1\}\,\mathbb{P}\{E_1\}$$
$$\mathbb{P}\{E_2 \mid E_1\} = \mathbb{P}\{A[\text{RANDOM}(i,n)] = b_i\} = \frac{1}{n-i+1}$$
$$\Longrightarrow \mathbb{P}\{E_1 \cap E_2\} = (n-i)!/n!$$

„Vor der i-ten Iteration der Schleife gilt: Wenn $(b_1, \ldots, b_{i-1})$ eine Anordnung von $i-1$ Elementen aus $A[1..n]$ ist, dann haben wir

$$\mathbb{P}\{A[1..i-1] = (b_1, \ldots, b_{i-1})\} = \frac{(n-i+1)!}{n!}.\text{“}$$

Terminierung: Aus $i = n+1$ folgt $\mathbb{P}\{A[1..n] = (b_1, \ldots, b_n)\} = \frac{1}{n!}$

Eine Variante des Bewerberproblems

- nicht alle interviewen, fast beste Person genügt;
- jede Person entweder sofort eingestellt oder abgelehnt; vergeben nach jedem Interview eine Bewertung, SCORE(i), keine zwei Bewertung seien gleich; Einstellung erfolgt, wenn SCORE größer als ein Schwellwert ist;
- Einstellung von ℓ Personen, $\ell > 1$.
- Strategie: Lehne k Personen ab, dann nimm die erste Person, die besser ist als die ersten k. Optimiere bzgl. k.

Algorithm ON-LINE-MAX(k, n)

```
 1: bestscore:=−∞
 2: for i = 1 to k do
 3:     if score(i) > bestscore then
 4:         bestscore:=score(i)
 5:     end if
 6: end for
 7: for i = k + 1 to n do
 8:     if score(i) > bestscore then
 9:         return i
10:     end if
11: end for
12: return n
```

Wir definieren $M_j = \max_{1 \le i \le j} \text{score}(i)$ und folgende Ereignisse

- $S =$ [bester Bewerber wird genommen]
- $S_i =$ [Bester wird genommen und ist in Position i]
- $B_i =$ [Bester in Position i]
- $C_i =$ [für $k+1 \le j \le i$ gilt $\text{score}(j) \le M_k$]

Dann gilt

$$\mathbb{P}\{S\} = \sum_{i=k+1}^{n} \mathbb{P}\{S_i\}$$

$$\mathbb{P}\{S_i\} = \mathbb{P}\{B_i \cap C_{i-1}\} = \mathbb{P}\{B_i\}\,\mathbb{P}\{C_{i-1}\} = \frac{1}{n} \cdot \frac{k}{i-1}$$

$$\Longrightarrow \mathbb{P}\{S\} = \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i} \sim \frac{k}{n}(\log n - \log k)$$

Sei

$$f(x) = \frac{x}{n}(\log n - \log x),$$

dann gilt

$$f'(x) = \frac{1}{n}(\log n - \log x - 1) = 0 \iff x = \frac{n}{e}.$$

D.h. für $k = n/e$ ist $\mathbb{P}\{S\}$ maximal und

$$\mathbb{P}\{S\} = \frac{1}{e} \approx 0,36788.$$

Randomisierte Algorithmen

Vorgestellte Algorithmen fallen in die Klasse der Las Vegas Algorithmen:

- mit Sicherheit korrekt,
- mit hoher (oder gewisser) Wahrscheinlichkeit effizient.

Im Gegensatz dazu sind Monte Carlo Algorithmen:

- mit Sicherheit effizient,
- mit hoher (oder gewisser) Wahrscheinlichkeit korrekt.

Beispiel: Miller-Rabin Primzahltest

Satz

Sei p eine Primzahl und a eine zu p teilerfremde Zahl. Dann gilt $a^{p-1} \equiv 1 \mod p$

Beweis: Wir betrachten i, j mit $1 \leq i \leq p-1$ und $1 \leq j \leq p-1$. Dann folgt aus $ia \equiv ja \mod p$, dass $p \mid (i-j)a$ und folglich $p \mid i-j$ ↯

Daher sind $a, 2a, \ldots, (p-1)a$ (jeweils modulo p) paarweise verschieden.

Bilden wir das Produkt dieser Zahlen, so erhalten wir

$$a^{p-1}(p-1)! \equiv (p-1)! \mod p,$$

woraus $a^{p-1} \equiv 1 \mod p$ folgt. □

Idee zum Primzahltest für n: Wähle zufälliges $a \in \{1, \ldots, n-1\}$ und prüfe, ob $a^{n-1} \equiv 1 \mod n$.

- Falls nein, so ist n nicht prim.
  a heißt dann *Zeuge* der Nichtprimalität von n.
- Andernfalls ist n möglicherweise prim.

Falls n eine *Carmichaelzahl* ist, dann gilt für alle a mit $\mathrm{ggT}(a, n) = 1$, dass $a^{n-1} \equiv 1 \mod n$, obwohl n nicht prim ist.
$\Longrightarrow$ wenige Zeugen!

Carmichaelzahlen: 561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, ...

Satz

Sei p eine ungerade Primzahl. Dann hat die Gleichung $a^2 \equiv 1 \mod p$ genau zwei Lösungen in $\{1, \ldots, p-1\}$, nämlich 1 und $p-1$.

a mit $a^2 \equiv 1 \mod n$ und $2 \leq a \leq n-2$ heißt *nichttriviale Quadratwurzel* modulo n

Idee:
Testen auf nichttriviale Quadratwurzeln in Fermat-Test einbauen;
mit zufälligem a, $1 < a < n-1$.

Potenzieren ist effizent machbar: Sei $Q : a \mapsto a^2$ und $M = aQ$.

Beispiel: $52 = (110100)_2$.
Übersetzen mit $1 \to M$, $0 \to Q$, ohne führenden 1er: $QQMQM$
Offensichtlich gilt $a^{52} = QQMQMa$

$\mathcal{O}(\log n)$ Anwendungen der Operatoren Q und M,
jeweils $\mathcal{O}\left((\log n)^2\right)$ zum Ausführen von Q bzw. M modulo n.
$\longrightarrow$ Laufzeit $\mathcal{O}\left((\log n)^3\right)$, polynomiell in der Anzahl der Ziffern.

Idee:
Um $a^{n-1} \mod n$ wird höchstens $\lfloor \log_2 n \rfloor$-mal das Quadrat einer Zahl x berechnet.
Prüfe, ob x nichttriviale Quadratwurzel ist!
$\rightsquigarrow$ Anpassung des Begriffs des *Zeugen*!

Algorithm POT(a, e, n)

```
 1: is_prime:=TRUE
 2: if e = 0 then
 3:     is_prime:=FALSE
 4:     return (1,is_prime)
 5: else
 6:     (x,is_prime) :=POT(a, ⌊e/2⌋, n)
 7:     r := x * x  mod n
 8:     if r = 1 and x ≠ 1 and x ≠ n − 1 then
 9:         is_prime:=FALSE
10:     end if
11: end if
12: if e = 1  mod 2 then
13:     r := r * a
14: end if
15: return (r,is_prime)
```

Algorithm MILLER-RABIN(a, e, n)

1: a :=RANDOM$(2, n-1)$
2: (r,is_prime) :=POT$(a, n-1, n)$
3: **if** is_prime=FALSE **then**
4: **print** "n ist nicht prim."
5: **else**
6: **print** "n ist wahrscheinlich prim."
7: **end if**

Satz

Falls n eine ungerade zusammengesetzte Zahl ist, dann gibt es in der Menge $\{2, \ldots, n-1\}$ *mindestens* $\frac{n-1}{2}$ *Zeugen.*

Folgerung

Die Wahrscheinlichkeit, dass der Miller-Rabin-Test bei einer zusammengesetzten Zahl n versagt, ist kleiner als $\frac{1}{2}$.

Beweis des Satzes:

1. Fall: n ist keine Carmichaelzahl.

Dann gibt es ein $x \in \mathbb{Z}_n^*$ mit $x^{n-1} \not\equiv 1 \mod n$

Sei

$$G = \{a \in \mathbb{Z}_n^* \mid a^{n-1} \equiv 1 \mod n\}$$

- $1 \in G$, also $G \neq \emptyset$.
- Für $x, y \in G$ ist auch $xy \in G$.

$\Longrightarrow$ G ist (echte!) Untergruppe von $\mathbb{Z}_n^*$.

Daraus folgt

$$|G| \leq \frac{|\mathbb{Z}_n^*|}{2} \leq \frac{n-1}{2}$$

2. Fall: n ist Carmichaelzahl

nicht in dieser VO, da kompliziert.

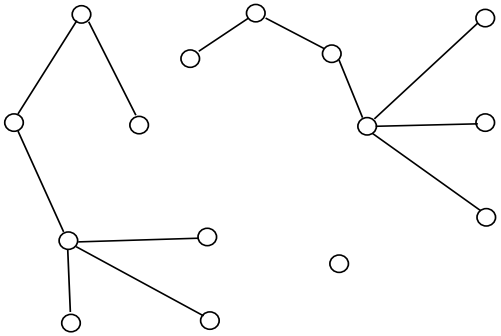
Für die Praxis weniger relevant, weil selten.

Auch hier gilt, dass alle Nichtzeugen in einer echten Untergruppe von $\mathbb{Z}_n^*$ enthalten sind. □

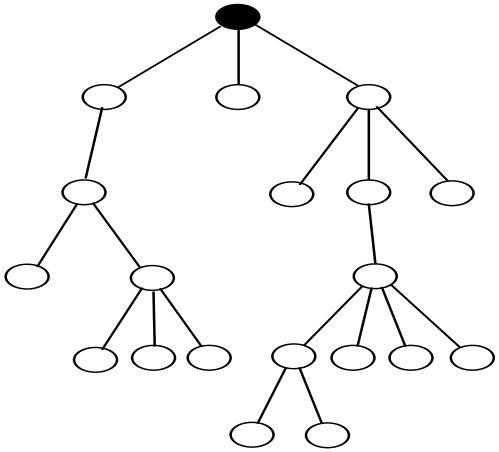
6) Heapsort

Heaps

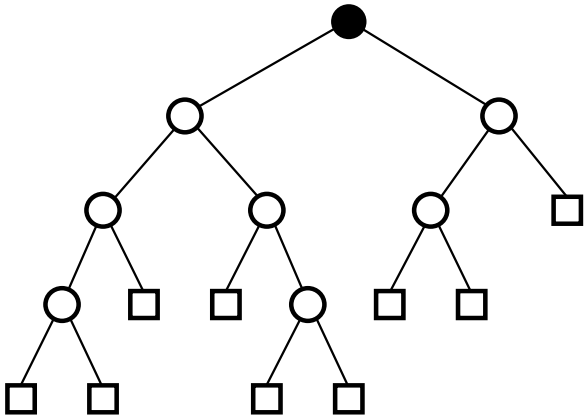
Bäume:



Wurzelbäume

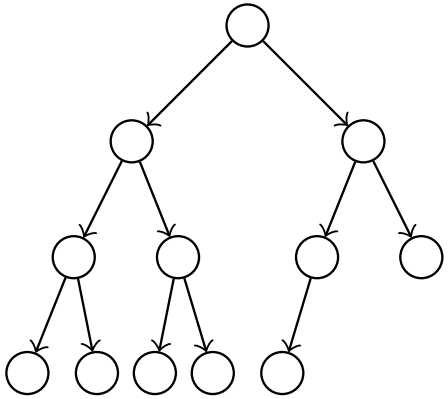


Binärbäume

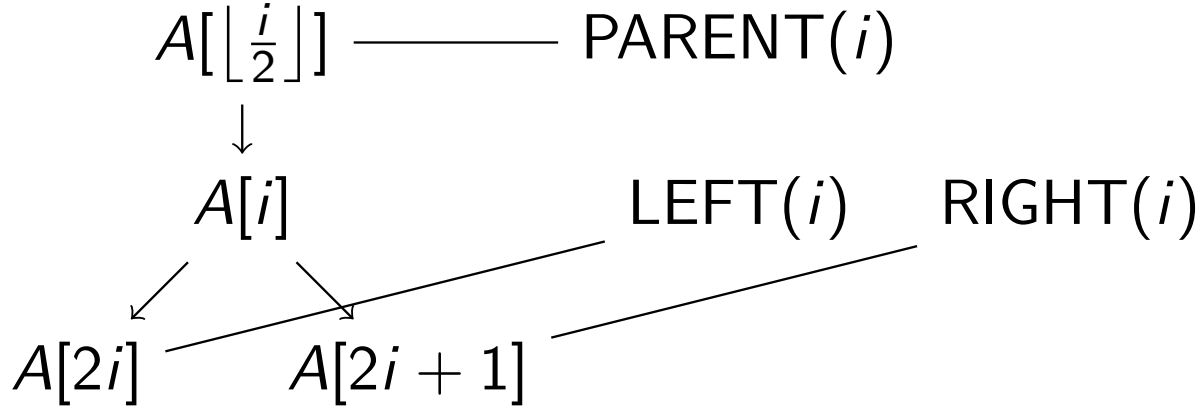


Heapsort

Heap: Datenstruktur, die fast vollständigem Binärbaum entspricht,



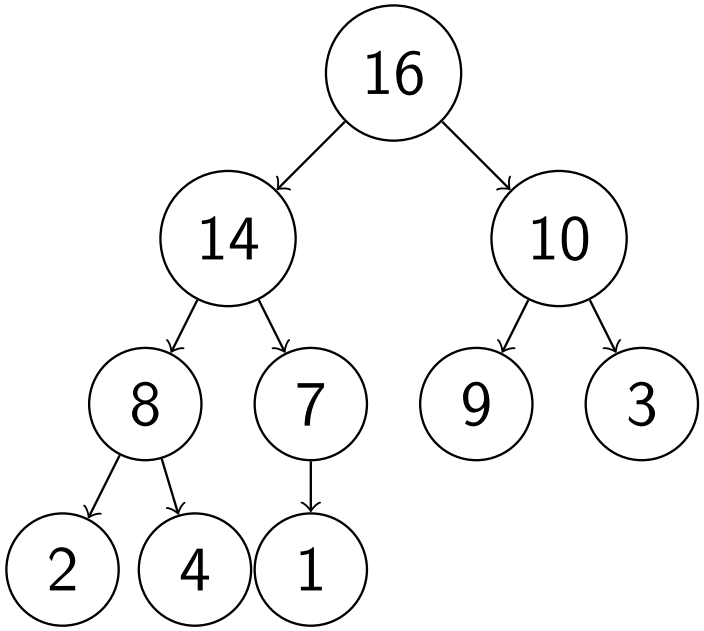
genauer: ein Datenfeld $A[1..n]$, bei dem die Position von $A[i]$ wie folgt eingebettet ist:



Weiters muss immer $A[\lfloor \frac{i}{2} \rfloor] \geq A[i]$ gelten (Max-Heap) bzw. $A[\lfloor \frac{i}{2} \rfloor] \leq A[i]$ (Min-Heap).

Heapsort

Beispiel eines Max-Heaps: $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$



Wir definieren $h(x) =$ Höhe von x, $h(\text{Wurzel}) =:$ Höhe des Heaps.

Heapsort

Prozeduren für Heaps:

- PARENT(i) := $\lfloor \frac{i}{2} \rfloor$, LEFT($i$) := $2i$, RIGHT(i) := $2i + 1$;
- MAX-HEAPIFY: aufrechterhalten der Heap-Eigenschaft;
- BUILD-MAX-HEAP: erzeugen eines Max-Heap aus ungeordnetem Feld;
- HEAPSORT: in-place-Sortieren eines Feldes;
- Anwendung von Heaps: Prozeduren für die Implementierung und Wartung einer Prioritätswarteschlange.

Aufrechterhaltung der Heap-Eigenschaft

Voraussetzung: Binärbäume mit Wurzeln in den Positionen LEFT(i) und RIGHT(i) sind Max-Heaps,
möglicherweise $A[i] <$ Kinder von $A[i]$

Eingabe: A, i;
Ausgabe: Binärbaum mit Wurzel $A[i]$, der Max-Heap ist.

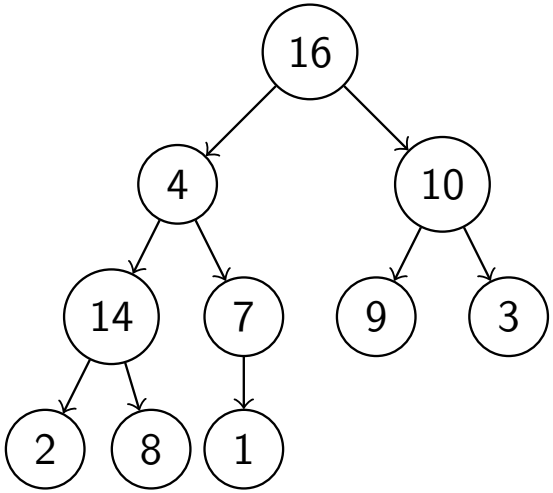
Idee: $A[i]$ nach unten befördern.

Heapsort

Algorithm MAX-HEAPIFY(A, i)

```
 1: l := LEFT(i)
 2: r := RIGHT(i)
 3: if l ≤ heap_size and A[l] > A[i] then
 4:     max := l
 5: else
 6:     max := i
 7: end if
 8: if r ≤ heap_size and A[r] > A[max] then
 9:     max := r
10: end if
11: if max ≠ i then
12:     A[i] ⟷ A[max]
13:     MAX-HEAPIFY(A, max)
14: end if
```

Beispiel:

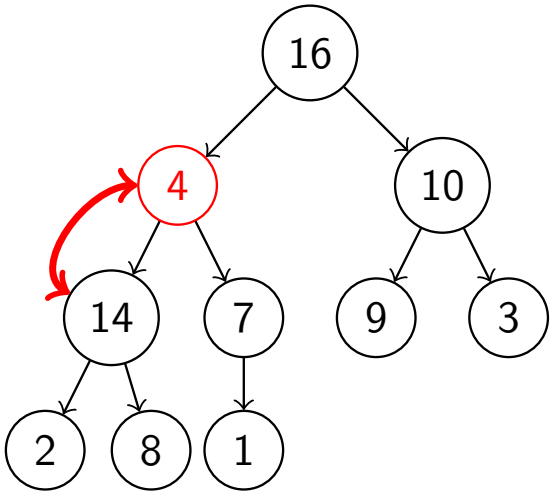


Heapsort

Algorithm MAX-HEAPIFY(A, i)

```
 1: l := LEFT(i)
 2: r := RIGHT(i)
 3: if l ≤ heap_size and A[l] > A[i] then
 4:     max := l
 5: else
 6:     max := i
 7: end if
 8: if r ≤ heap_size and A[r] > A[max] then
 9:     max := r
10: end if
11: if max ≠ i then
12:     A[i] ⟷ A[max]
13:     MAX-HEAPIFY(A, max)
14: end if
```

Beispiel:

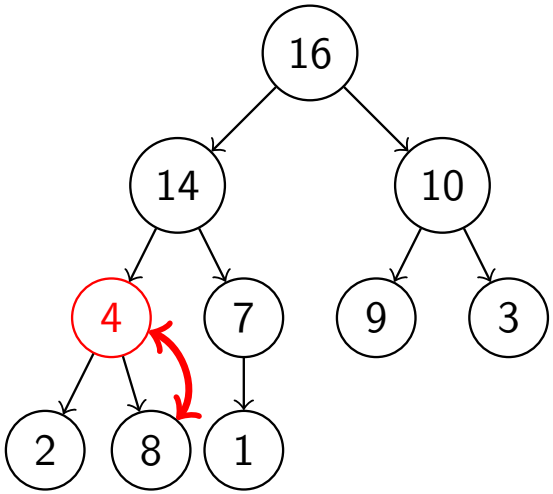


Heapsort

Algorithm MAX-HEAPIFY(A, i)

```
 1: l := LEFT(i)
 2: r := RIGHT(i)
 3: if l ≤ heap_size and A[l] > A[i] then
 4:     max := l
 5: else
 6:     max := i
 7: end if
 8: if r ≤ heap_size and A[r] > A[max] then
 9:     max := r
10: end if
11: if max ≠ i then
12:     A[i] ⟷ A[max]
13:     MAX-HEAPIFY(A, max)
14: end if
```

Beispiel:

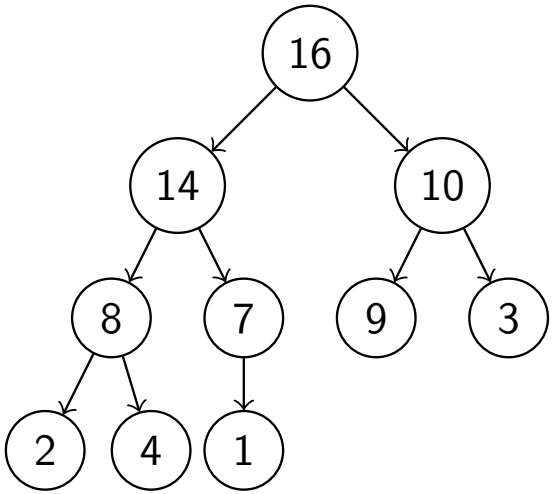


Heapsort

Algorithm MAX-HEAPIFY(A, i)

```
 1: l := LEFT(i)
 2: r := RIGHT(i)
 3: if l ≤ heap_size and A[l] > A[i] then
 4:     max := l
 5: else
 6:     max := i
 7: end if
 8: if r ≤ heap_size and A[r] > A[max] then
 9:     max := r
10: end if
11: if max ≠ i then
12:     A[i] ⟷ A[max]
13:     MAX-HEAPIFY(A, max)
14: end if
```

Beispiel:



Kosten:

$\mathcal{O}(h(i)) = \mathcal{O}(\log n)$

Bauen eines Heaps

MAX-HEAPIFY bottom-up, um $A[1..n]$ in Heap zu verwandeln.

Algorithm BUILD-MAX-HEAP(A)

1: heap_size:=$|A|$
2: **for** $i = \left\lfloor \frac{|A|}{2} \right\rfloor$ **downto** 1 **do**
3: MAX-HEAPIFY(A, i)
4: **end for**

Korrektheit: Verwende die Schleifeninvariante
„Vor jeder Iteration ist jeder Knoten der Liste $(i+1, i+2, \ldots, n)$ Wurzel eines Max-Heaps.“

Initialisierung: $\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2, \ldots, n$ sind die Blätter des Baums, also triviale Max-Heaps.

Heapsort

„Vor jeder Iteration ist jeder Knoten der Liste $(i+1, i+2, \ldots, n)$ Wurzel eines Max-Heaps.“

Aufrechterhaltung: Die Kinder von i sind größer als i und daher Wurzeln von Max-Heaps (wegen der Schleifeninvariante). Daher macht MAX-HEAPIFY(A, i) aus i die Wurzel eines Max-Heaps.

Terminierung: $i = 0$ und somit sind $1, 2, \ldots, n$ Wurzeln eines Max-Heaps, insbes. 1.

Kosten von BUILD-MAX-HEAP

$\frac{n}{2}$ Aufrufe von MAX-HEAPIFY, Aufwand jeweils $\mathcal{O}(\log n)$.

Besser:

- Kosten von MAX-HEAPIFY: $\mathcal{O}(h)$ für Knoten der Höhe h; .
- Anzahl solcher Knoten ist höchstens $\left\lceil \frac{n}{2^{h+1}} \right\rceil$;
- n-elementiger Heap hat Höhe $\lfloor \log_2 n \rfloor$

$\Longrightarrow$ Kosten beschränkt durch

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot \mathcal{O}(h) = \mathcal{O}\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \right) = \mathcal{O}(n)$$

Heapsort

Eingabe: $A[1 \ldots n]$

Idee:

1. BUILD-MAX-HEAP(A) $\leadsto$ $A[1]$ größtes Element;
2. $A[1] \longleftrightarrow A[n]$;
3. $A[n]$ entfernen, dann sind LEFT(1) und RIGHT(1) Wurzeln von Max-Heaps, $A[1]$ eventuell nicht;
4. MAX-HEAPIFY($A, 1$)

Algorithm HEAP-SORT(A)

```
1: BUILD-MAX-HEAP(A)
2: for i = |A| downto 2 do
3:     A[1] ⟷ A[i]
4:     heap_size:=heap_size-1
5:     MAX-HEAPIFY(A, 1)
6: end for
```

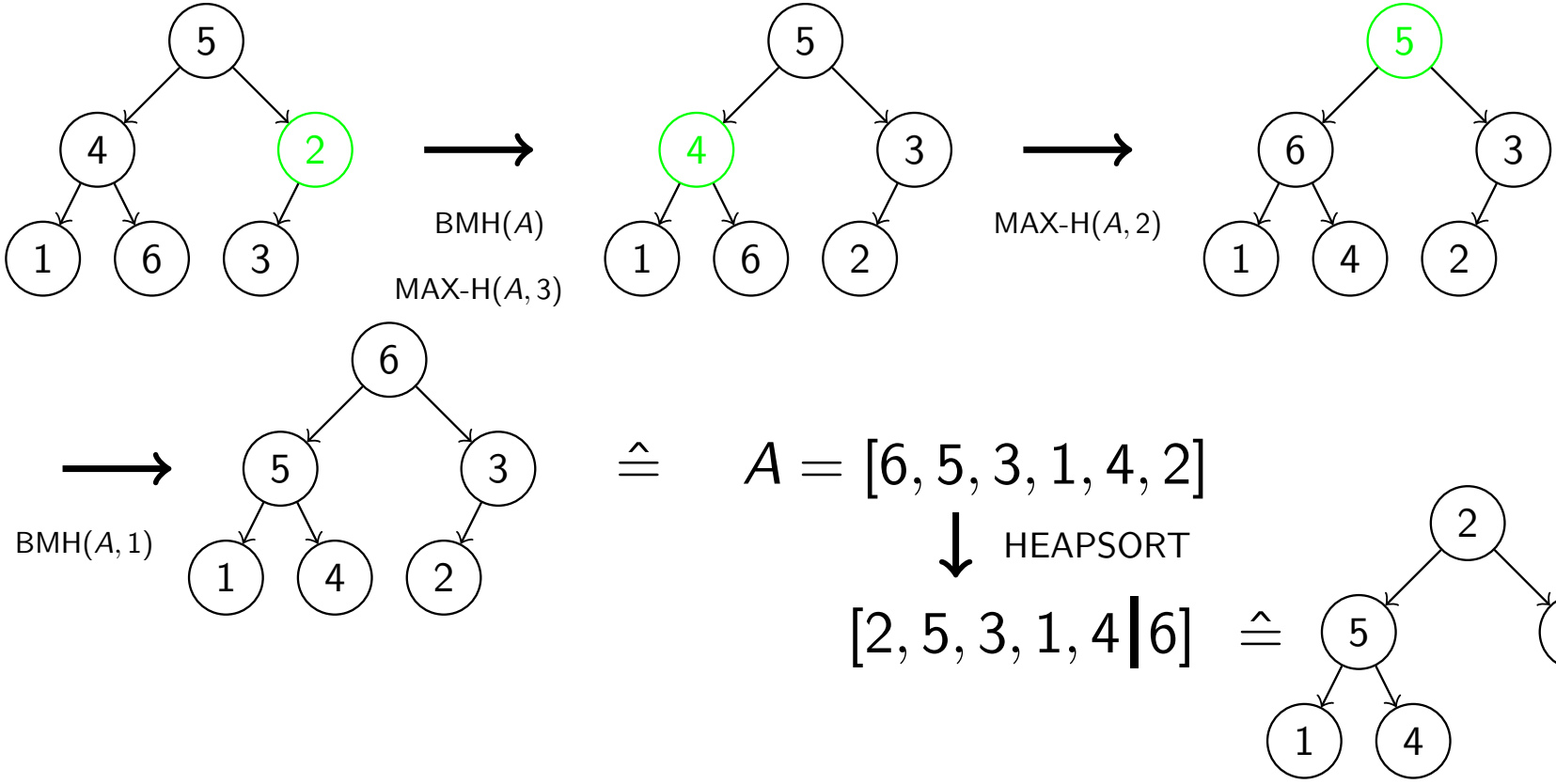
Heapsort

Kosten:

- BUILD-MAX-HEAP: $\mathcal{O}(n)$;
- MAX-HEAPIFY: $n - 1$ Aufrufe, jeweils $\mathcal{O}(\log n)$
- Gesamt: $\mathcal{O}(n \log n)$

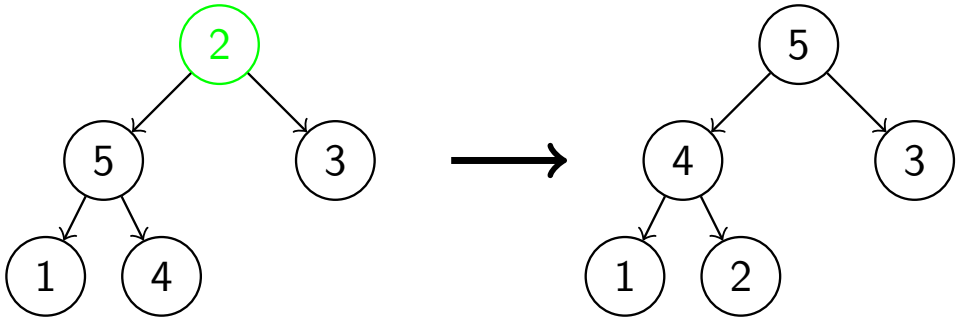
Heapsort

Beispiel $A = [5, 4, 2, 1, 6, 3]$,



Heapsort

$A = [2, 5, 3, 1, 4 \,|\, 6]$

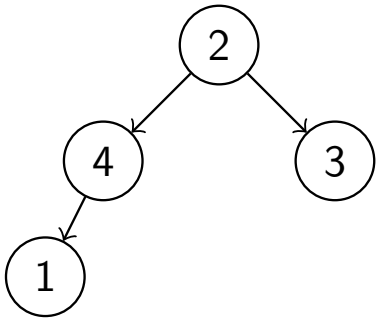


$\widehat{=}$ $A = [5, 4, 3, 1, 2 \,|\, 6]$

$\downarrow$ HEAPSORT

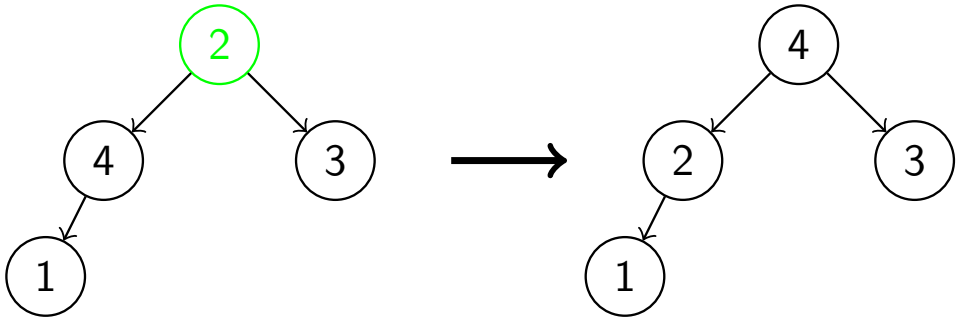
$[2, 4, 3, 1 \,|\, 5, 6]$

$\widehat{=}$



Heapsort

$A = [2, 4, 3, 1 \,|\, 5, 6]$

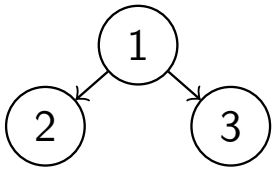


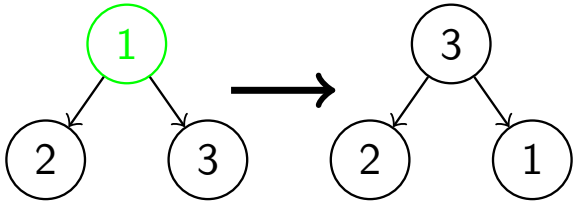
$\hat{=}$ $A = [4, 2, 3, 1 \,|\, 5, 6]$

$\downarrow$ HEAPSORT

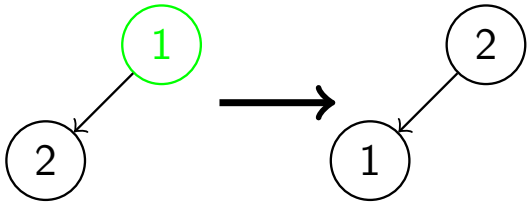
$[1, 2, 3 \,|\, 4, 5, 6]$

$\hat{=}$





$\hat{=} A = [3, 2, 1 \,|\, 4, 5, 6] \rightarrow [1, 2 \,|\, 3, 4, 5, 6]$



$\hat{=} A = [2, 1 \,|\, 3, 4, 5, 6] \rightarrow [1 \,|\, 2, 3, 4, 5, 6]$