Algorithmen und Komplexität
Vorlesung 10

Wolfgang Globke

universität wien

DHBW
Duale Hochschule
Baden-Württemberg

8. Mai 2019

Wiederholung: Sortieren

Eines der wichtigsten (wenn nicht sogar das wichtigste) Problem in der Informatik ist das Sortieren.

Wir betrachten Elemente eines Datentyps D. Dabei nehmen wir an, dass es für D eine Ordnung gibt, die uns Vergleiche der Form $x \leq y$ für Elemente $x, y \in D$ erlaubt (z.B. Ordnung der natürlichen Zahlen, lexikographische Ordnung von Zeichenketten, ...).

Sortieren bedeutet, aus einer endlichen Liste $L = [x_1, \ldots, x_n]$ von Elementen aus D eine neue Liste L' zu erstellen, so dass

$$L' = [x_{j_1}, \ldots, x_{j_n}], \quad x_{j_1} \leq x_{j_2} \leq \ldots \leq x_{j_{n-1}} \leq x_{j_n},$$

gilt, wobei $j_1, \ldots, j_n$ eine Umordnung der Indizes $1, \ldots, n$ ist.

Das Wort „Liste" kann hier auch für Arrays oder ähnliche Datencontainer stehen!

Wiederholung: Sortieren

Ist D eine höhere Datenstruktur („Struktur", „Klasse"), so wird meist eine bestimmte Komponente von D als Schlüssel zum Sortieren verwendet.

Generell bezeichnen wir ein Sortierverfahren

- als in-place, wenn zusätzlich zum Speicher für die Liste L nur konstant viel Speicher verwendet werden muss,
- als stabil, wenn für zwei identische Listenelemente $L[i] = x = L[k]$ mit $i < k$, die Reihenfolge beim Sortieren erhalten bleibt, $L'[j_i] = x = L'[j_k]$ mit $j_i < j_k$.

Wiederholung: SelectionSort und InsertionSort

Wir haben bereits zwei Sortierverfahren kennengelernt,

- SelectionSort, das Schritt für Schritt das Minimum aus L entfernt und an eine neue Liste L' anhängt.
- InsertionSort, das nach dem selben Prinzip funktioniert wie das Sortieren von Spielkarten auf der Hand.

Der Aufwand von Sortierverfahren wird üblicherweise durch die erforderliche Anzahl der Vergleiche zum Sortieren einer Liste der Länge n bestimmt.

- SelectionSort hat asymptotischen Aufwand $\Theta(n^2)$.
- InsertionSort hat asymptotischen Aufwand $\mathrm{O}(n^2)$, im günstigsten Fall aber $\mathrm{O}(n)$.

Quadratischer Aufwand ist nicht sehr gut für große Datenmengen,
aber für kleine Datenmengen können diese Algorithmen trotzdem sehr gut laufen.

Sortierverfahren: QuickSort

Sortieren durch Teile-und-Herrsche

Die Sortierverfahren SelectionSort und InsertionSort funktionieren durch mehrfaches Iterieren über die Eingabeliste L.

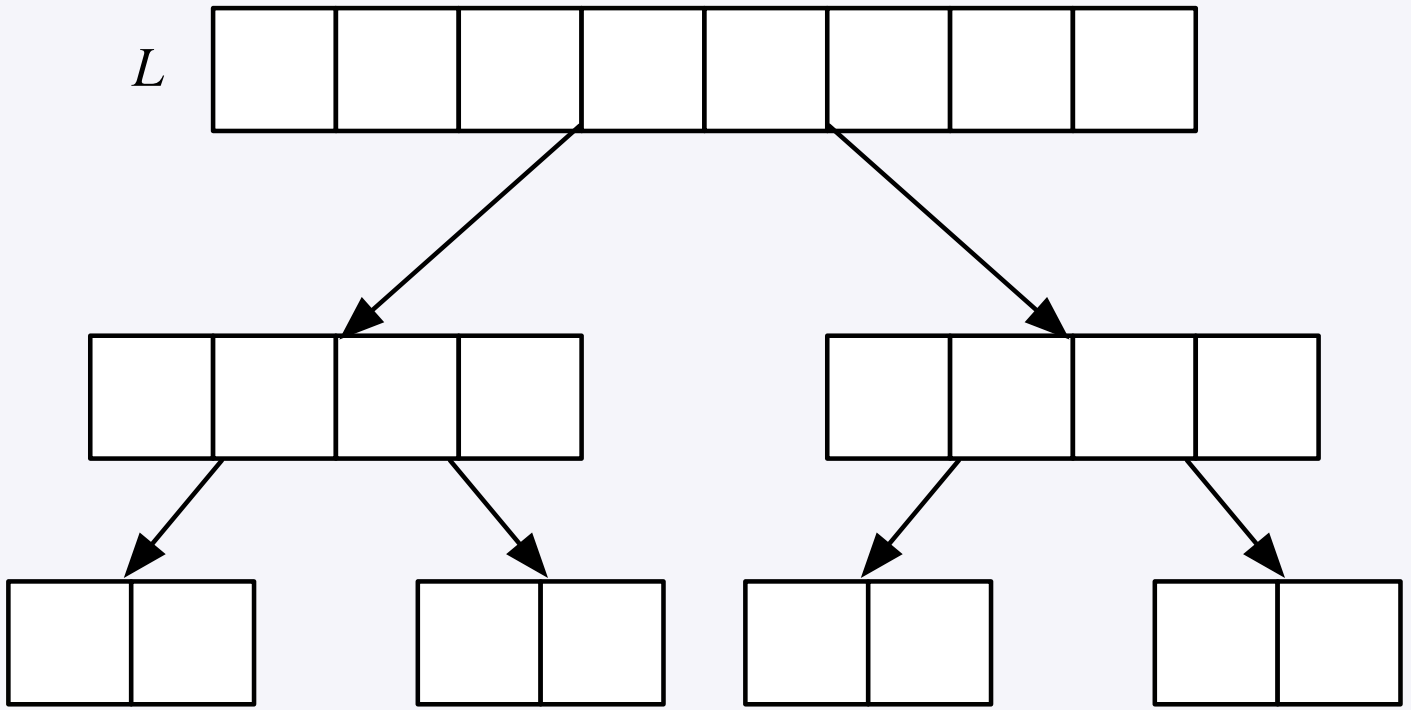
Eine andere Idee ist es, einen induktiven Teile-und-Herrsche-Ansatz zu probieren.

- Beobachtung:
  Für Listen der Länge $n = 1$ oder $n = 2$ ist das Sortieren trivial.
- Ansatz:
  Durch wiederholtes Zerteilen der Liste L lässt sich das Sortierproblem rekursiv auf mehrere Instanzen des trivialen Falls zurückführen.
- Schwierigkeit:
  Es ist sicherzustellen, dass sortierte Teillisten in korrekter Weise zur sortierten Gesamtliste L' zusammengefügt werden.

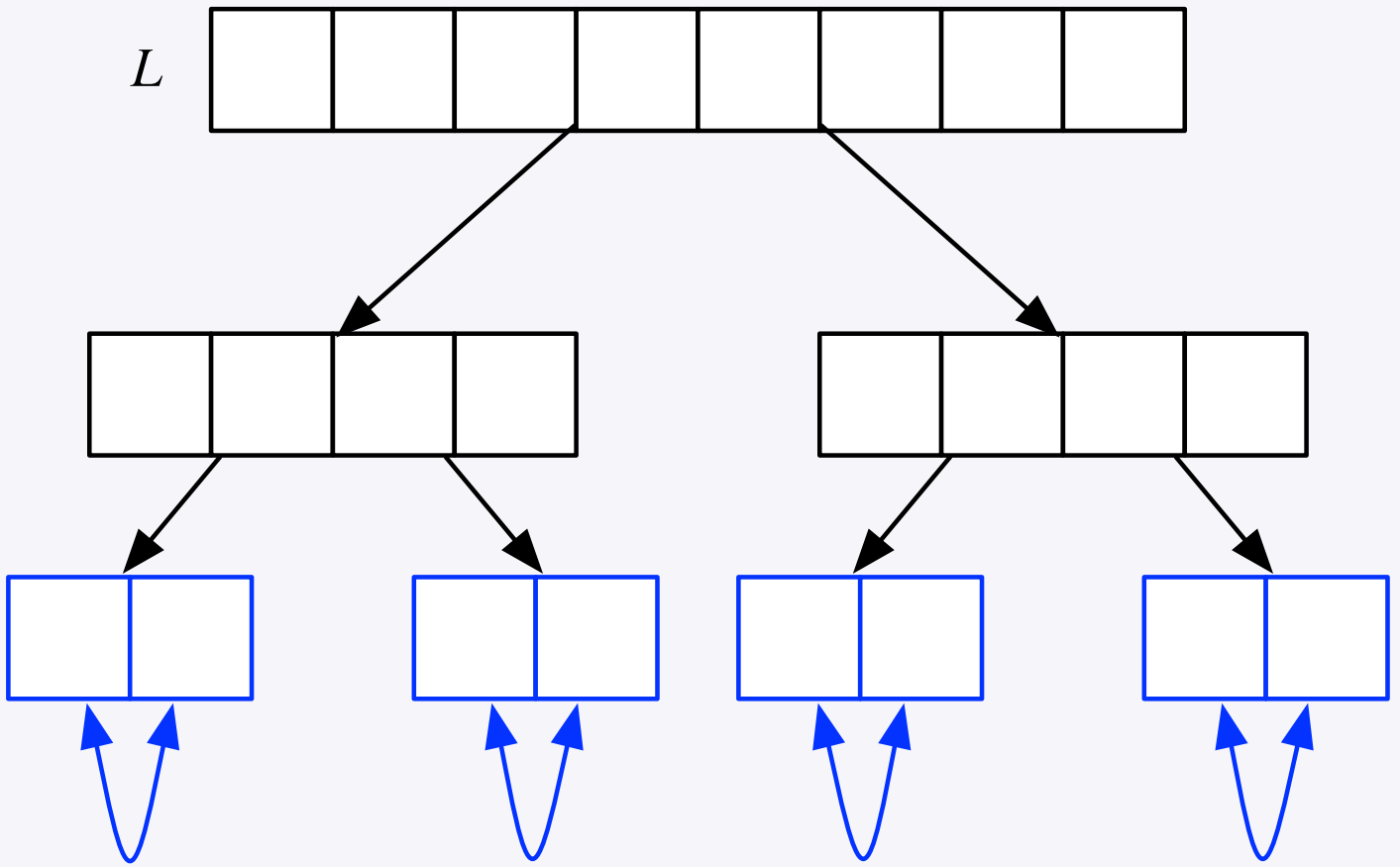
Sortieren durch Teile-und-Herrsche

Die grobe Idee:

Die grobe Idee:

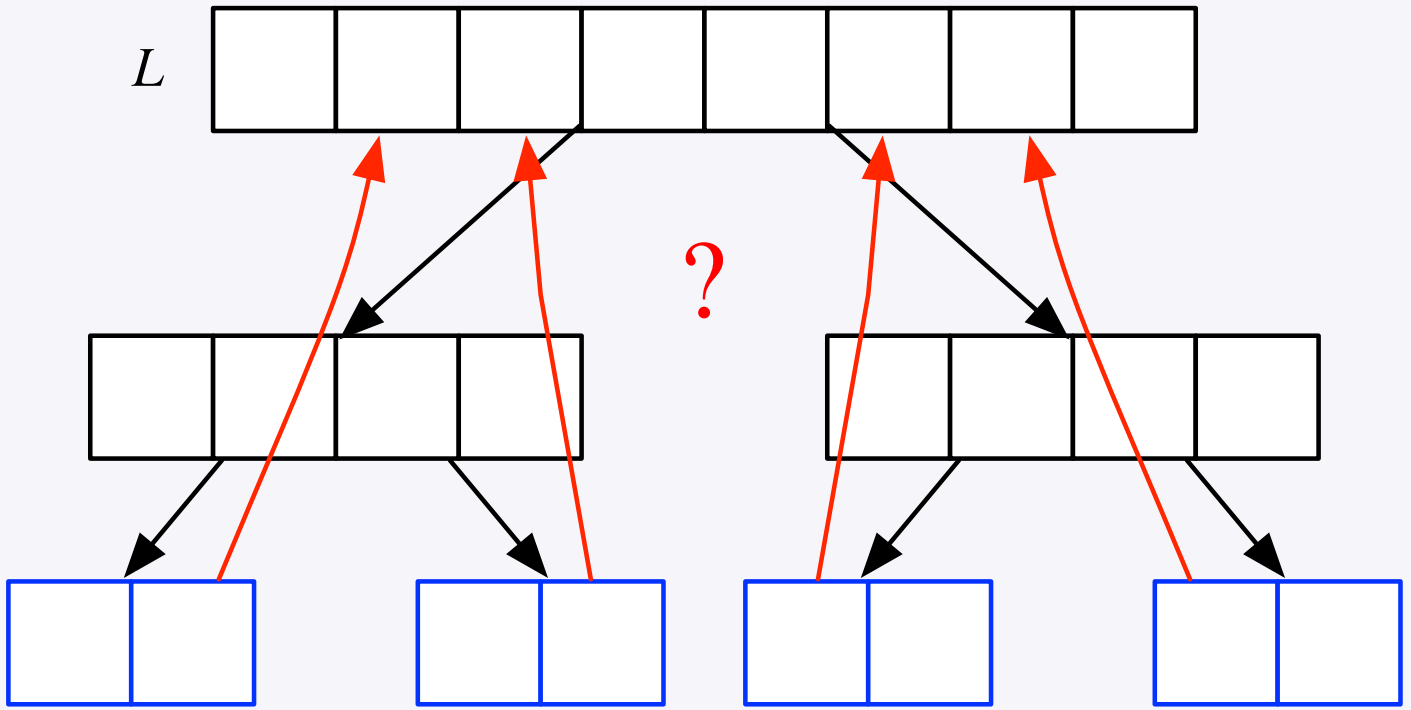


Die grobe Idee:

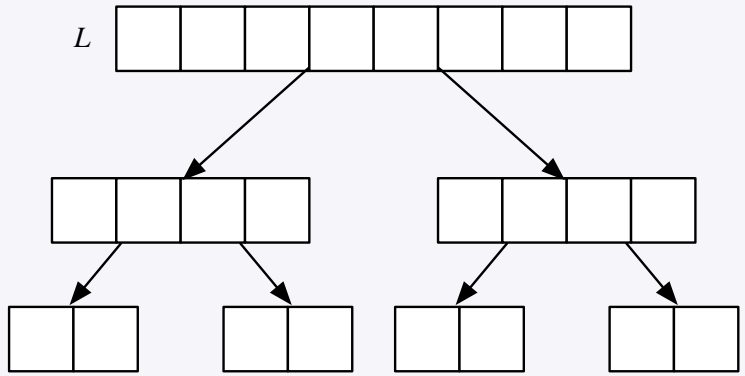


leicht zu sortieren

Die grobe Idee:



Sortieren durch Teile-und-Herrsche



Es gibt zwei Sortierverfahren mit unterschiedlichen Ansätzen für das Zusammenfügen der sortierten Teile:

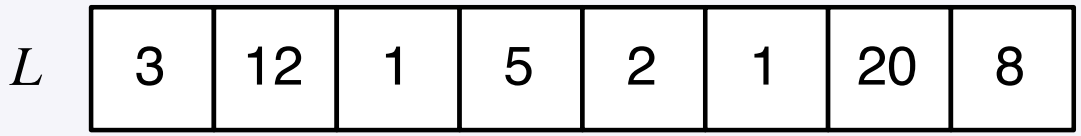
1. QuickSort (oder Sortieren durch Zerlegen) nimmt bei jeder Teilung eine grobe Sortierung vor, so dass bereits alle Elemente im linken Teilbaum kleiner sind als alle Elemente im rechten Teilbaum.
   Am Schluss müssen die einzelnen Teile nur noch andeinander gereiht werden.
2. MergeSort (oder Sortieren durch Mischen ?!) nimmt beim Zerteilen keine Sortierung vor, sondern sortiert beim Zusammenfügen die einzelnen Teile nach dem „Reißverschlussverfahren”.

QuickSort

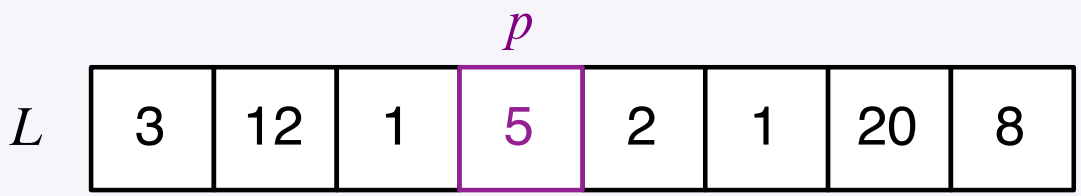
Wir betrachten zuerst QuickSort. Der Algorithmus sortiert eine Liste L der Länge n folgendermaßen:

1. Enthält L nur ein Element, so ist L bereits sortiert.
2. Andernfalls, wähle ein bestimmtes Element p in L aus, das Pivotelement (z.B. $p = L[\lfloor \frac{n}{2} \rfloor]$).
3. Schreibe alle Elemente $L[i] < p$ in eine neue Liste $L_<$ und alle Elemente $L[j] \geq p$ in eine neue Liste $L_>$ (außer p selbst).
4. Wende das Verfahren jeweils auf $L_<$ und $L_>$ an.
5. Nun sind $L_<$ und $L_>$ jeweils sortiert. Füge sie zusammen zum Ergebnis $L' = [L_< \mid p \mid L_>]$.

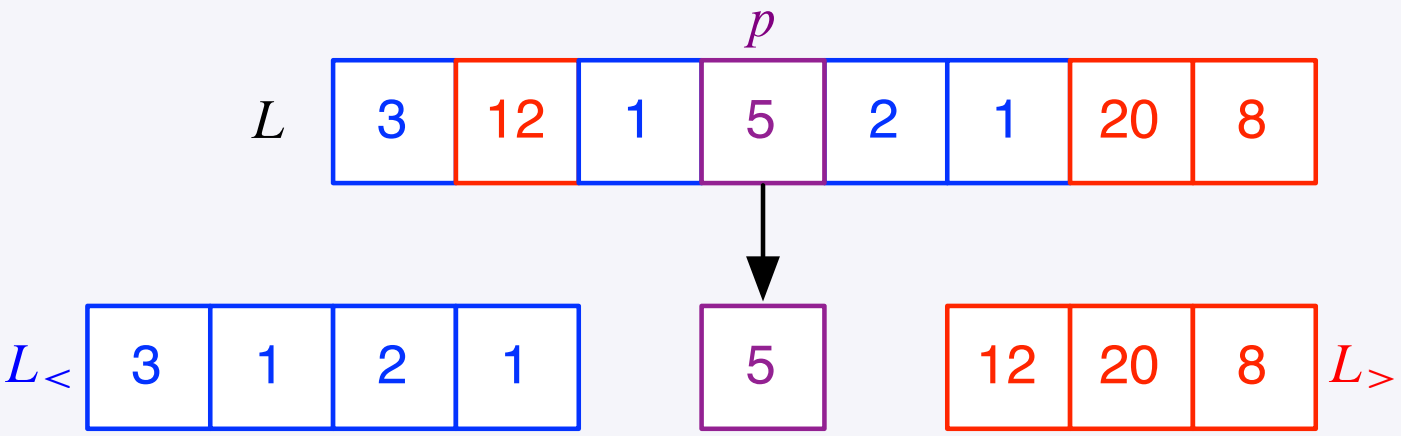
QuickSort – Beispiel



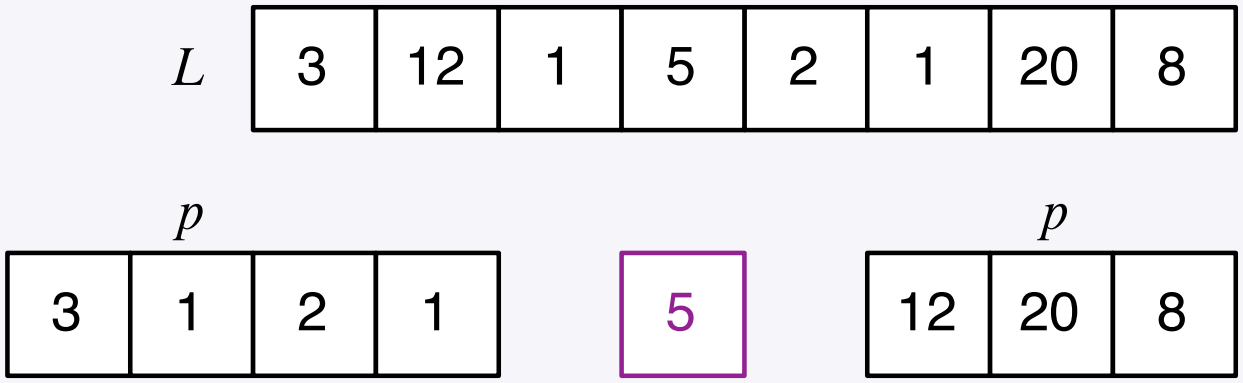
QuickSort – Beispiel



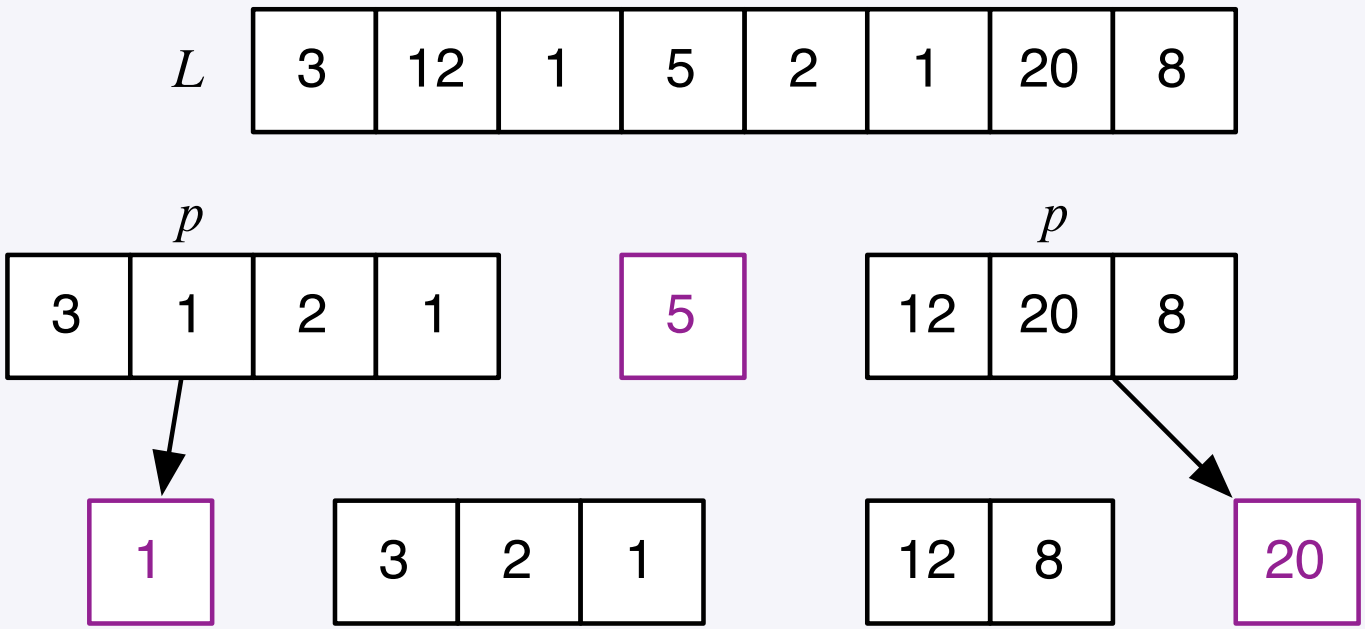
QuickSort – Beispiel



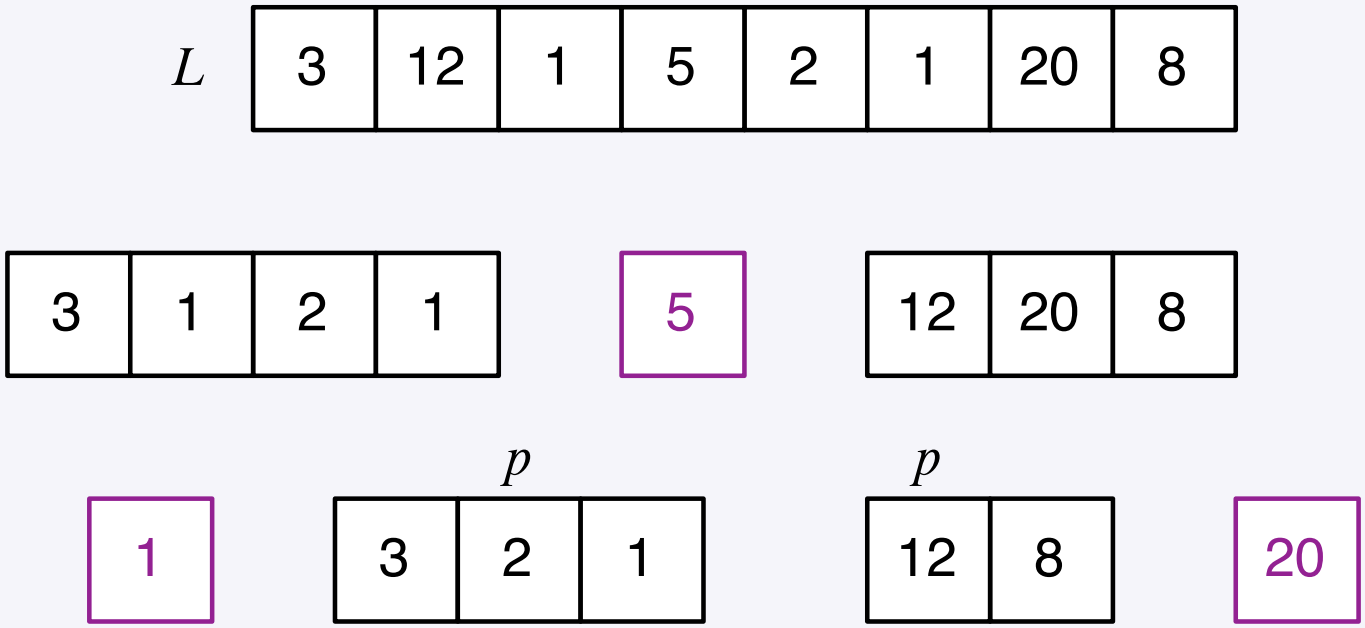
QuickSort – Beispiel



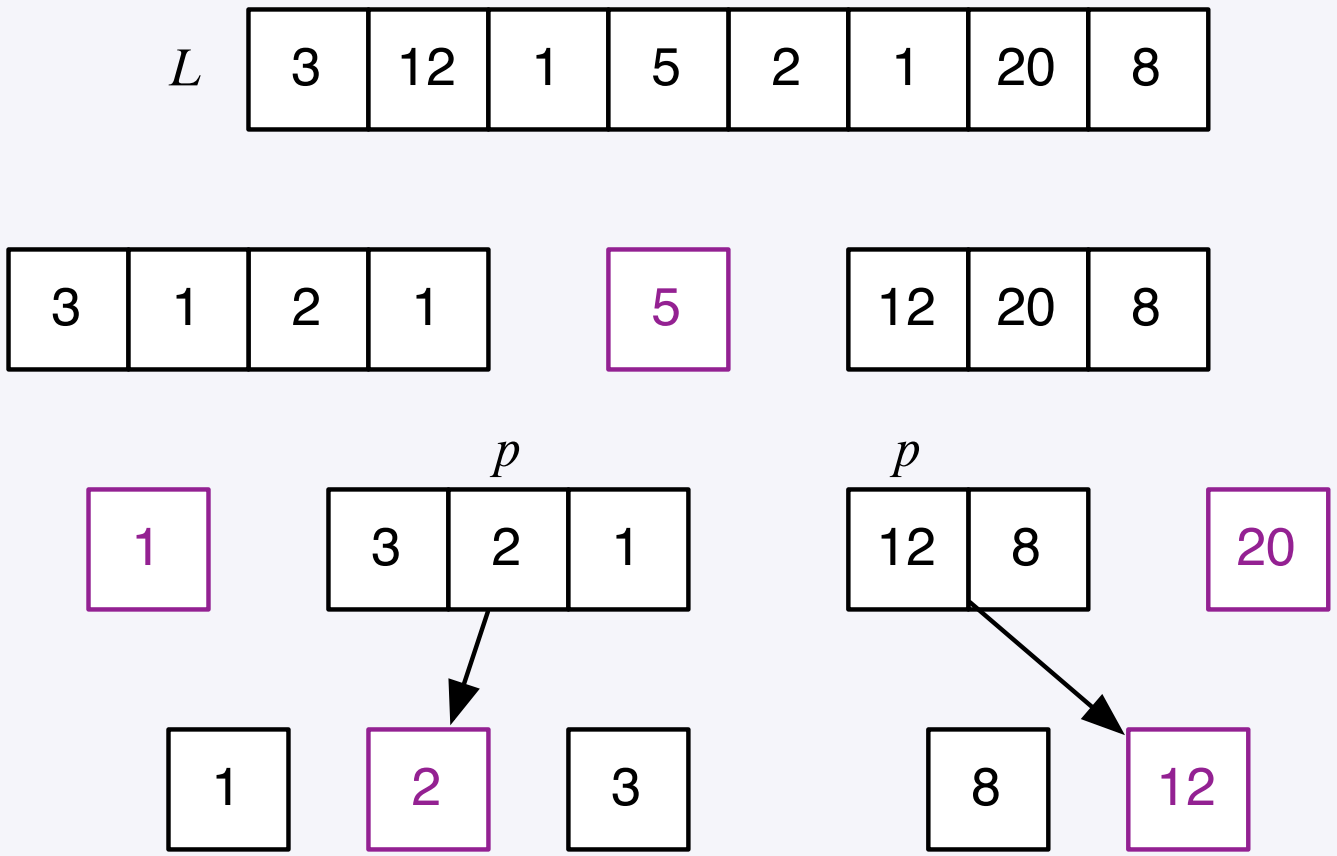
QuickSort – Beispiel



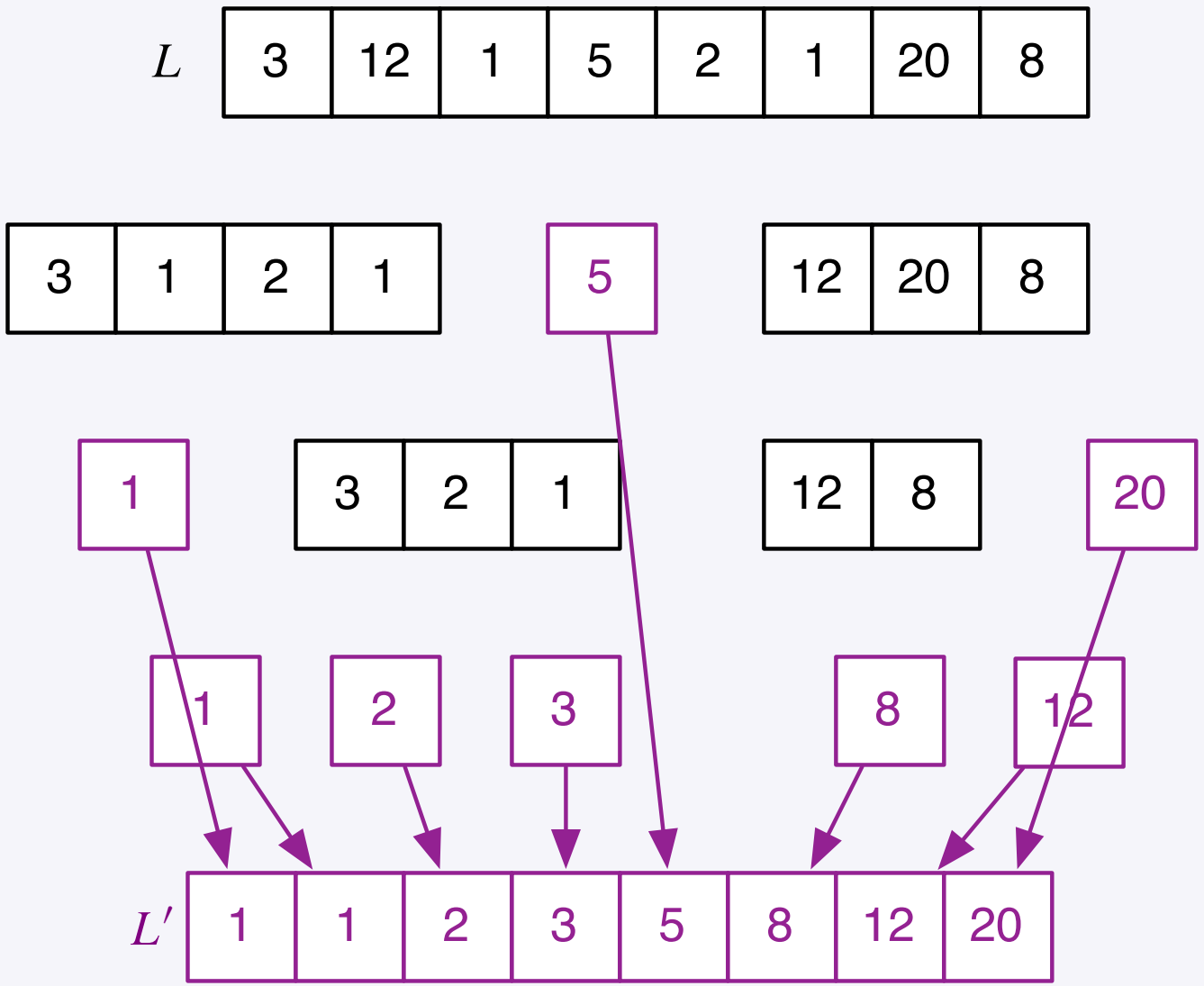
QuickSort – Beispiel



QuickSort – Beispiel



QuickSort – Beispiel



QUICKSORT(L, L')
    $n :=$ LENGTH(L)
    if $n = 1$ **then** $L' := L$
    else
        $p := L[\lfloor \frac{n}{2} \rfloor]$
        $(L_<, L_>) :=$ SPLIT(L, p)
        QUICKSORT$(L_<, L'_<)$
        QUICKSORT$(L_>, L'_>)$
        $L' :=$CONCAT$(L'_<, [p], L'_>)$
    end_if

Hierbei ist SPLIT eine Prozedur, die L in die Teillisten $L_<$ und $L_>$ der Elemente $< p$ bzw. $\geq p$ (ohne p selbst) aufteilt, und CONCAT ein Prozedur, die mehrere Listen aneinanderhängt.

QuickSort: Korrektheit

Wir nehmen an, dass SPLIT und CONCAT korrekt sind.

Satz
QUICKSORT *ist korrekt und terminiert nach endlich vielen Schritten.*

- Terminierung: Die Listen $L_<$ und $L_>$ enthalten weniger Elemente als L, da das Element p keiner Liste zugeteilt wird. Somit endet die Rekursion nach $\leq n$ Schritten.
- Korrektheit: Beweis durch Induktion über n.
  Induktionsanfang $n = 1$: Eine Liste mit einem Element ist sortiert.
- Induktionsvoraussetzung: QUICKSORT sortiert Listen der Länge $\leq n$ korrekt.
- Induktionsschluss $n \rightsquigarrow n + 1$:
  - Da SPLIT korrekt ist, sind alle Elemente in $L_<$ kleiner als p und alle Elemente in $L_>$ größer-gleich p.
  - Da $L_<$ und $L_>$ echt kleiner als L sind (p fehlt), sind nach Induktionsvoraussetzung $L'_<$ und $L'_>$ sortiert.
  - Zusammen: $L' = [L'_< \mid p \mid L'_>]$ ist sortiert. □

QuickSort: Aufwand

Es bezeichne $T_{\text{qs}}(n)$ den Aufwand von QUICKSORT für eine Liste L der Länge n. Außerdem bezeichnen $n_<$ und $n_>$ die jeweiligen Längen von $L_<$ und $L_>$.

Es gilt

$$T_{\text{qs}}(1) = c = \text{const.}$$
$$T_{\text{qs}}(n) = T_{\text{SPLIT}} + T_{\text{qs}}(n_<) + T_{\text{qs}}(n_>) + T_{\text{CONCAT}}.$$

Bevor wir genaueres zum Aufwand von QUICKSORT sagen können, müssen wir uns die Prozedur SPLIT genauer ansehen.

Zerlegen der Liste

Wir betrachten eine Version von SPLIT, die besonders gut geeignet ist, wenn L ein Array ist (oder eine andere Datenstruktur mit Zugriff auf Elemente in O(1)).

Die Aufteilung von L in Listen $L_<$ und $L_>$ kann flexibler erfolgen als zuvor:
Für gegebenes Pivotelement p soll die Prozedur SPLIT L in zwei Listen $L_<$ und $L_>$ zerlegen, so dass

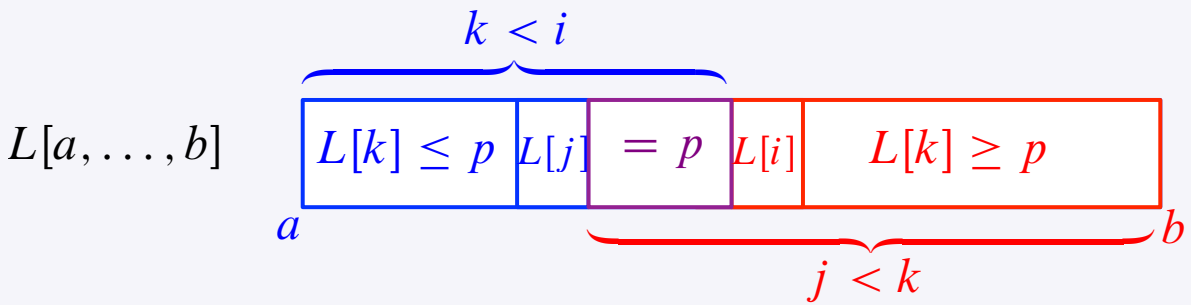
- $L_<$ alle Elemente $L[i]$ aus L enthält, die $L[i] < p$ erfüllen,
- $L_>$ alle Elemente $L[j]$ aus L enthält, die $L[j] > p$ erfüllen,
- die Elemente mit Wert $L[i] = p$ können sowohl in $L_<$ als auch in $L_>$ liegen, wobei mindestens eines dieser Elemente keiner der beiden Listen zugeteilt wird.

Zerlegen der Liste

Wir wollen SPLIT geschickt entwerfen, so dass die Zerlegung innerhalb der Liste L realisiert werden kann (kein zusätzlicher Speicher erforderlich).

- SPLIT muss in der Lage sein, auf Teillisten $L[a,\ldots,b]$ von L zu arbeiten (mit $0 \le a \le b \le n-1$).
  Es sollte also als Argumente L und die Indexgrenzen a, b einer Teilliste von L erhalten.
- Am Ende von SPLIT sollte $L[a,\ldots,b]$ in zwei (überlappende) Segmente $L[a,\ldots,i]$ und $L[j,\ldots,b]$ eingeteilt sein, so dass $j < i$ und

$$\begin{array}{ll} L[k] \le p & \text{für } a \le k < i \\ L[k] \ge p & \text{für } j < k \le b \\ L[k] = p & \text{für } j < k < i \end{array} .$$

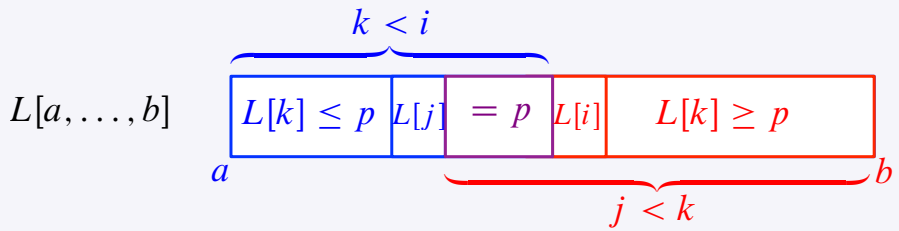


Zerlegen der Liste

Wir konstruieren SPLIT(L, a, b) so, dass durchgehend gilt:

$$\begin{array}{ll} L[k] \leq p & \text{für } a \leq k < i \\ L[k] \geq p & \text{für } j < k \leq b \\ L[k] = p & \text{für } j < k < i \end{array} \quad . \qquad \text{(I)}$$

1. Beobachte, dass Bedingung (I) gilt, wenn wir mit $a = i$ und $j = b$ anfangen.
2. Bedingung (I) bleibt erhalten, wenn wir i erhöhen und j verkleinern, solange $L[i] < p < L[j]$ gilt:
   while $L[i] < p$ **do** $i := i + 1$ **end_while**
   while $L[j] > p$ **do** $j := j - 1$ **end_while**
3. Danach gilt $L[i] \geq p \geq L[j]$. Falls die Endbedingung $j < i$ noch nicht gilt, vertausche $L[i]$ und $L[j]$. Dann gilt (I) sogar mit „$k \leq i$" bzw. „$j \leq k$".
4. Also setze $i := i + 1$ und $j := j - 1$, und (I) gilt immer noch.
5. Wiederhole Schritte 2 bis 4 solange, bis $j < i$ gilt.
   Dann ist der gesuchte Zustand erreicht:



Zerlegen der Liste

```
SPLIT(L, a, b)
    n := LENGTH(L)
    i := a
    j := b
    p := L(⌊n/2⌋)
    while i ≤ j do
        while L[i] < p do i := i + 1 end_while
        while L[j] > p do j := j − 1 end_while
        if i ≤ j do
            tmp := L[i]
            L[i] := L[j]
            L[j] := tmp
            i := i + 1
            j := j − 1
        end_if
    end_while
```

Zerlegen: Korrektheit und Aufwand

Die Korrektheit von SPLIT ergibt sich direkt aus der Konstruktion, da die Invariante (I) durchweg erhalten blieb.

In jedem Schleifendurchlauf ändert sich mindestens einer der beiden Indizes. Da während des Ablaufs $a \leq i \leq j \leq b$ gilt, gibt es somit maximal $2(b-a)$ Schleifendurchläufe. Also ist der Aufwand von SPLIT

$$T(\text{SPLIT}) \in \mathrm{O}(b-a).$$

Wir können nun eine modifizerte Version von QUICKSORT angeben, die ohne eine zweite Liste L' und ohne den CONCAT-Befehl auskommt.

```
QUICKSORT(L, a, b)
    if b > a then
        i := a
        j := b
        p := L[⌊n/2⌋]
        while i ≤ j do
            while L[i] < p do i := i + 1 end_while
            while L[j] > p do j := j − 1 end_while
            if i ≤ j do
                tmp := L[i]
                L[i] := L[j]
                L[j] := tmp
                i := i + 1
                j := j − 1
            end_if
        end_while
        QUICKSORT(L, a, j)
        QUICKSORT(L, i, b)
    end_if
```

QuickSort: Aufwand

Für die modifizierte Version von QUICKSORT gilt

$$T_{\text{qs}}(1) = c = \text{const.}$$
$$T_{\text{qs}}(n) = T_{\text{SPLIT}} + T_{\text{qs}}(n_<) + T_{\text{qs}}(n_>)$$

mit $T(\text{SPLIT}) \in \mathrm{O}(n-1)$.

Extremfall 1:
$p = \min L$ (analog: $p = \max L$) in jedem Schritt:

- Dann ist $L_< = [\,]$ und $L_> = L \backslash \{p\}$, also $n_> = n-1$.
- Somit ist $T_{\text{qs}}(n) = c(n-1) + T_{\text{qs}}(n-1)$ für eine geeignete Konstante c.
- Induktiv:

$$\begin{aligned} T_{\text{qs}}(n) &= c(n-1) + T_{\text{qs}}(n-1) \\ &= c(n-1) + c(n-2) + T_{\text{qs}}(n-2) \\ &\vdots \\ &= c\big((n-1) + (n-2) + \ldots + 2 + 1\big) = c\,\frac{(n-1)n}{2}. \end{aligned}$$

Also gilt $T_{\text{qs}}(n) \in \Theta(n^2)$.

QuickSort: Aufwand

Für die modifizierte Version von QUICKSORT gilt

$$T_{\text{qs}}(1) = c = \text{const.}$$
$$T_{\text{qs}}(n) = T_{\text{SPLIT}} + T_{\text{qs}}(n_<) + T_{\text{qs}}(n_>)$$

mit $T(\text{SPLIT}) \in \text{O}(n-1)$.

Extremfall 2:
$n_< \approx \frac{n}{2} \approx n_>$ in jedem Schritt:

- Dann ist $T_{\text{qs}}(n) = c(n-1) + 2T_{\text{qs}}(\frac{n}{2})$.
- Nach dem Master-Theorem (na, wer erinnert sich?) gilt

$$T_{\text{qs}}(n) \in \Theta(n \log(n)).$$

QuickSort: Erwarteter Aufwand

Im ungünstigsten Fall hat QUICKSORT den Aufwand $\Theta(n^2)$.
Im günstigsten Fall hat QUICKSORT den Aufwand $\Theta(n \log(n))$.

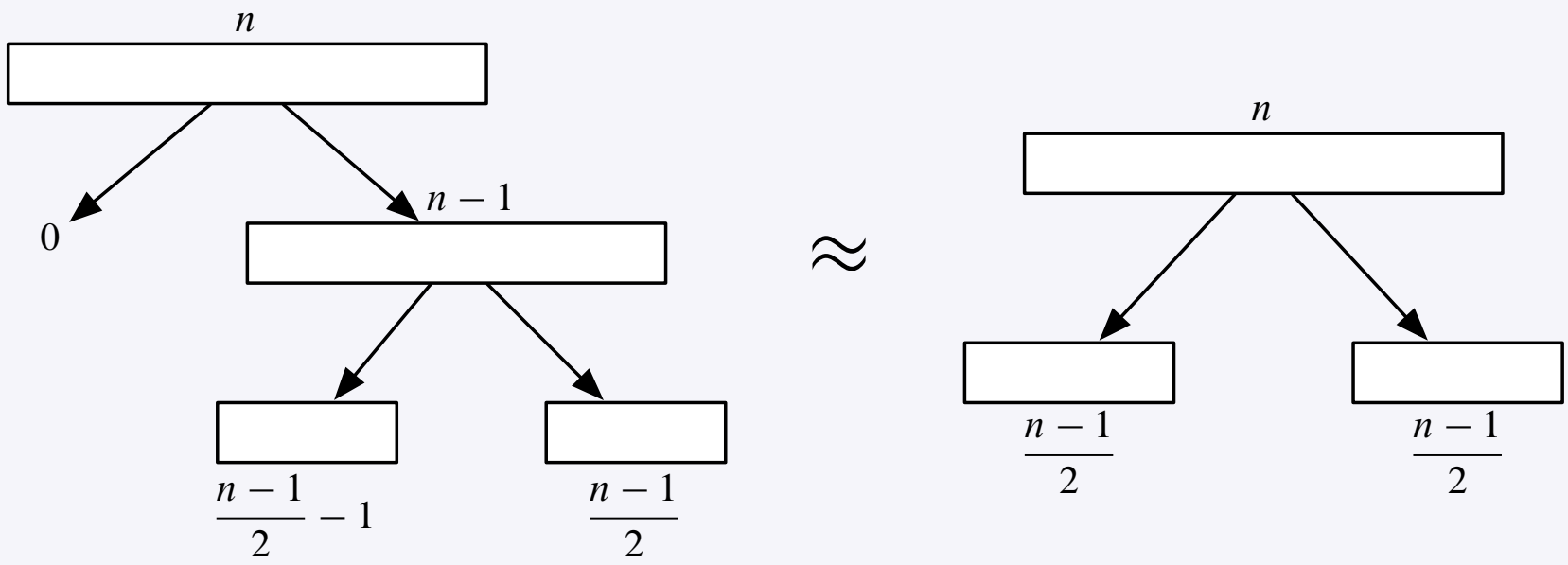
Aber was ist der Aufwand, den wir erwarten sollten?

Heuristisch:

- Der Aufwand von QUICKSORT wird durch die relative Reihenfolge der Elemente in L bestimmt (nicht durch die konkreten Werte).
- Wir nehmen an, alle Permutationen der Werte seien gleich wahrscheinlich als Eingabe.
- Es ist sehr unwahrscheinlich, dass in jedem Rekursionsschritt einer der Extremfälle eintritt.
- Nehmen wir an, dass „gute" Aufteilungen ($n_< \approx \frac{n}{2} \approx n_>$) und „schlechte" Aufteilungen ($p = \min L$) sich in der Rekursion abwechseln.
- Dies führt zu einer Aufteilung in drei Teilfelder der Größen 0, $\frac{n-1}{2} - 1$, $\frac{n-1}{2}$ mit Zerlegungskosten $\mathrm{O}(n) + \mathrm{O}(n-1) = \mathrm{O}(n)$.
- Asymptotisch ist das nicht schlechter als eine einzelne Aufteilung im günstigsten Fall. Somit ist der „erwartete Aufwand" asymptotisch gleich dem günstigsten Aufwand, $\mathrm{E}(T_{\mathrm{qs}}(n)) \in \mathrm{O}(n \log(n))$.

QuickSort: Erwarteter Aufwand

Eine „schlechte” Zerlegung gefolgt von einer „guten” Zerlegung (links) ist asymptotisch in $O(n)$, genauso wie eine einzelne „gute” Zerlegung (rechts).



Natürlich ist die Konstante, die in $O(n)$ versteckt ist, auf der linken Seite größer.

Fragen

- Ist QUICKSORT in-place?
  Jein!
  (Hängt davon ab, ob „in-place” Speicherverbrauch auf dem Call-Stack einschließt oder nicht.)
- Ist QUICKSORT ein stabiler Sortieralgorithmus?
  $\rightsquigarrow$ Übungsblatt IV.