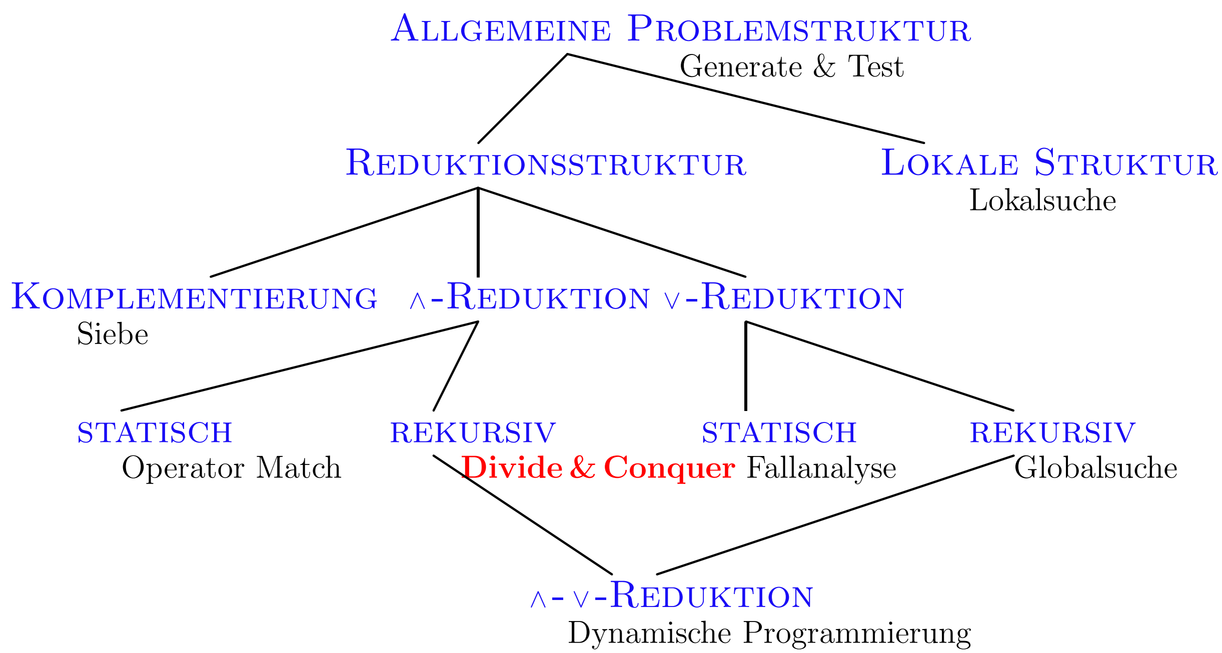
Divide & Conquer Algorithmen



- **Effiziente Verarbeitung strukturierter Daten**
  – Sehr gebräuchliche und einfache Programmiertechnik
  – **Aufteilen**: Zerlege Problem in kleinere Teilprobleme
  – **Erobern**: Löse Teilprobleme separat und setze Lösungen zusammen

- **$\wedge$-Reduktion des Problems**
  – Lösung benötigt alle Teillösungen gleichzeitig
  – Teillösungen bauen aufeinander auf

Ein typischer Divide & Conquer Algorithmus

- **Maximum einer nichtleeren Liste von Zahlen**

```
FUNCTION maxL(L:Seq(ℤ)):ℤ WHERE L≠[]
  RETURNS m SUCH THAT m∈L ∧ ∀x∈L. x≤m
≡ if |L|=1 then hd(L)
            else let a,L'=L
                 in let m'=maxL(L')
                   in if a<m' then m' else a
```

Einfache (primitive) Eingaben (`|L|=1`) erhalten direkte Lösung `hd(L)`

Andernfalls Dekomposition der Eingabe mit Funktion `HdTl: L ↦ a,L'`

Einfache Teillösung `a=id(a)` für `a` — rekursive Lösung `m'=maxL(L')` für `L'`

Komposition der Teillösungen `a` und `m'` mit der Funktion `max`:$\mathbb{Z}\to\mathbb{Z}$

- **Vereinheitlichung durch separierte Beschreibung**

```
FUNCTION maxL(L:Seq(ℤ)):ℤ WHERE L≠[]
  RETURNS m SUCH THAT m∈L ∧ ∀x∈L. x≤m
≡ if |L|=1 then hd(L)
            else (max ∘ (id×maxL) ∘ HdTl)(L)
```

Algorithmus zur Verarbeitung der Liste folgt festem Schema

Allgemeines Divide & Conquer Schema

Grundstruktur aller Divide & Conquer Algorithmen

FUNCTION $\boldsymbol{f}(x\mathtt{:}D)\mathtt{:}R$ WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y]$

$\equiv$ if $primitive[x]$ then $Directly\text{-}solve[x]$

else $(Compose \circ g{\times}\boldsymbol{f} \circ Decompose)\,(x)$

- **5 zentrale Komponenten der Algorithmentheorie**
  - *Decompose*: $D \rightarrow D' \times D$ Aufspalten der Eingabe in Teilprobleme
  - Rekursiver Aufruf von $\boldsymbol{f}$ zusammen mit Hilfsfunktion $\boldsymbol{g}$: $D \rightarrow R'$
    - Funktionsprodukt $\boldsymbol{g}{\times}\boldsymbol{f}(x,y) = (g(x),\ f(y))$
  - *Compose*: $R' \times R \rightarrow R$ Zusammensetzen der Teillösungen
  - *Directly-solve*: $D \rightarrow R$: Direkte Lösung für einfache Teilprobleme
  - *primitive*: $D \rightarrow \mathbb{B}$: Test, ob Eingabe “einfach” ist

Korrektheit folgt aus wenigen Voraussetzungen

Korrektheit des Divide & Conquer Schemas

$\texttt{FUNCTION}\ \boldsymbol{f}(x\texttt{:}D)\texttt{:}R\ \texttt{WHERE}\ I[x]\ \texttt{RETURNS}\ y\ \texttt{SUCH THAT}\ O[x,y]$

$\equiv\ \texttt{if}\ \mathit{primitive}[x]\ \texttt{then}\ \mathit{Directly\text{-}solve}[x]\ \texttt{else}\ \texttt{(}\mathit{Compose} \circ g{\times}\boldsymbol{f} \circ \mathit{Decompose}\texttt{)}\,(x)$

ist korrekt, wenn 6 Axiome erfüllt sind

---

1. **Direkte Lösung korrekt für primitive Eingaben**

   $\texttt{FUNCTION}\ \boldsymbol{f_p}(x\texttt{:}D)\texttt{:}R\ \texttt{WHERE}\ I[x] \wedge \mathit{primitive}[x]\ \texttt{RETURNS}\ y\ \texttt{SUCH THAT}\ O[x,y]$

2. **Ausgabebedingung $\boldsymbol{O}$ rekursiv zerlegbar in $\boldsymbol{O_D}$, $\boldsymbol{O'}$ und $\boldsymbol{O_C}$**

   $\boldsymbol{O_D[x,y',y] \wedge O'[y',z'] \wedge O[y,z] \wedge O_C[z',z,t] \Rightarrow O[x,t]}$

   (Strong Problem Reduction Principle)

3. **Dekomposition erfüllt $\boldsymbol{O_D}$ und ‘verkleinert’ Problem**

   $\texttt{FUNCTION}\ \boldsymbol{f_d}(x\texttt{:}D)\texttt{:}D\texttt{'}{\times}D\ \texttt{WHERE}\ I[x] \wedge \neg\mathit{primitive}[x]$

   $\texttt{RETURNS}\ y',y\ \texttt{SUCH THAT}\ I'[y'] \wedge I[y] \wedge x \succ y \wedge O_D[x,y',y]$

4. **Hilfsfunktion $\boldsymbol{g}$ erfüllt $\boldsymbol{O'}$**

   $\texttt{FUNCTION}\ \boldsymbol{g}(y'\texttt{:}D\texttt{'})\texttt{:}R\texttt{'}\ \texttt{WHERE}\ I'[y']\ \texttt{RETURNS}\ z'\ \texttt{SUCH THAT}\ O'[y',z']$

5. **Komposition erfüllt $\boldsymbol{O_C}$**

   $\texttt{FUNCTION}\ \boldsymbol{f_c}(z',z\texttt{:}R\texttt{'}{\times}R)\texttt{:}R\ \texttt{WHERE}\ \texttt{true}\ \texttt{RETURNS}\ y\ \texttt{SUCH THAT}\ O_C[z',z,t]$

6. **Verkleinerungsrelation $\succ$ ist wohlfundierte Ordnung auf $\boldsymbol{D}$**

   Sechste Komponente in Algorithmentheorie, nötig für Terminierungsbeweis

Divide & Conquer Schema: Korrektheitsbeweis

FUNCTION $\boldsymbol{f}(x\!:\!D)\!:\!R$ WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y]$

$\equiv$ if $primitive[x]$ then $Directly\text{-}solve[x]$ else $(Compose \circ g\times\boldsymbol{f} \circ Decompose)\,(x)$

- **Partielle Korrektheit: strukturelle Induktion über ($\boldsymbol{D}$,$\succ$)**
  - **Primitive Eingaben:** $I[x] \wedge primitive[x]$
    - $f(x) = Directly\text{-}solve[x]$ Korrektheit folgt direkt aus Axiom 1
  - **Nichtprimitive Eingaben:** $I[x] \wedge \neg primitive[x]$
    - $f(x) = (Compose \circ g\times f \circ Decompose)\,(x)$
    - $Decompose[x]$ liefert y', y mit $O_D[x, y', y]$ und $x \succ y$ Axiom 3
    - $g(y')$ liefert z' mit $O'[y', z']$ Axiom 4
    - $f(y)$ liefert z mit $O[y, z]$ Induktionsannahme
    - $Decompose[z', z]$ liefert t mit $O_C[z', z, t]$ Axiom 5
    - t ist das Ergebnis $(f(x) = t)$ und es gilt $O[x, t]$ Axiom 2
- **Terminierung: Wohlfundiertheit von $\succ$** Axiom 6

Synthesestrategie für Divide & Conquer Algorithmen

Zerlege Problem in Spezifikationen für Teilprobleme

Start: `FUNCTION` $\boldsymbol{f}$`(`x`:`D`):`R `WHERE` $I[x]$ `RETURNS` y `SUCH THAT` $O[x,y]$

1. **Wähle $\succ$ und *Decompose* aus Wissensbank** $\mapsto$ $\boldsymbol{O_D}$,$\boldsymbol{D'}$, **Axiom 6**
2. **Konstruiere Hilfsfunktion $\boldsymbol{g}$:** $\mapsto$ $\boldsymbol{O', I'}$, **Axiom 4**
   – Heuristik: $\boldsymbol{g}{:=}\boldsymbol{f}$, falls $D'{=}D$, sonst $\boldsymbol{g}{:=}\boldsymbol{id}$
3. **Verifiziere *Decompose*, generiere Vorbedingung** $\mapsto$ *primitive*, **Axiom 3**
   – Heuristik: abgeleitete zusätzliche Vorbedingung für O_D ist $\neg primitive[x]$
4. **Konstruiere *Compose*** $\mapsto$ $\boldsymbol{O_C}$, **Axiome 5 & 2**
   – Heuristik: Erzeuge O_C mit Axiom 2;
   Synthetisiere *Compose* gemäß Axiom 5
5. **Konstruiere *Directly-solve*** $\mapsto$ **Axiome 1**
   – Heuristik: Suche nach vorgefertigten Lösungen, sonst neue Synthese
   – Falls dies nicht möglich ist, konstruiere eingeschränkte Vorbedingung $\hat{I}$
6. **Instantiiere das Divide & Conquer Schema**

Alternative Reihenfolgen der Synthesestrategie

FUNCTION $f(x{:}D){:}R$ WHERE $I[x]$ RETURNS y SUCH THAT $O[x,y]$

$\equiv$ if $\mathit{primitive}[x]$ then $\mathit{Directly\text{-}solve}[x]$ else $(\mathit{Compose} \circ g{\times}f \circ \mathit{Decompose})\,(x)$

- **Grundstrategie**
  - Wähle $\succ$ und *Decompose* aus Wissensbank
  - Konstruiere $\boldsymbol{g}$ heuristisch
  - Bestimme *primitive* durch Verifikation von *Decompose*
  - Konstruiere Spezifikation und Lösung für *Compose*
  - Konstruiere *Directly-solve*
- **Variante**
  - Wähle $\succ$ und *Decompose* aus Wissensbank
  - Konstruiere *Compose* heuristisch
  - Konstruiere Spezifikation und Lösung für $\boldsymbol{g}$
  - Bestimme *primitive*und konstruiere *Directly-solve*
- **Umgekehrte Strategie**
  - Wähle *Compose* aus Wissensbank
  - Konstruiere $\boldsymbol{g}$ heuristisch
  - Konstruiere Spezifikation und Lösung für *Decompose* und bestimme $\succ$
  - Bestimme *primitive* und konstruiere *Directly-solve*

Programmierwissen für Divide & Conquer

- **Standard-Zerlegungen für Typen (D)** $\mapsto$ *Decompose*, $\boldsymbol{O_D}$, $\boldsymbol{D'}$
  - Endliche Listen / Folgen (`Seq`(α)): `HdTl`, `ListSplit`
    - `ListSplit`(L) $\equiv$ `([`L_i`|i`$\in$`[1..|L|÷2]], [`L_i`|i`$\in$`[1+|L|÷2..|L|]])`

      mit $O_D[L, L_1, L_2] \equiv L{=}L_1{\circ}L_2$, $D' \equiv$ Seq(α)
  - Endliche Mengen (`Set`(α)): `ArbRest`
  - Produkträume (+ Mengen, Folgen, ...): Zerlegung in Einzelkomponenten
  - Natürlichen Zahlen ($\mathbb{N}$): Vorgängerfunktion
- **Standard-Wohlordnungen auf Typen (D)** $\mapsto$ $\succ$
  - Folgen und Mengen: Längen/Größenordnung $L \succ L' \equiv |L|>|L'|$
  - Produkträume (+ Mengen, Folgen, ...): Lexikographische Ordnung
    - $(a_1, b_1) \succ (a_2, b_2) \equiv a_1 > a_2 \lor (a_1 = a_2 \land b_1 > b_2)$
  - Zahlen ($\mathbb{N}/\mathbb{Z}$): Zahlenordnung bzw. Absolutordnung
- **Standard-Kompositionen für Typen (R)** $\mapsto$ *Compose*, $\boldsymbol{O_C}$, $\boldsymbol{R'}$
  - Endliche Folgen: `cons` ($a.l$), `append` ($L_1{\circ}L_2$)
  - Endliche Mengen: `insert` ($S{+}a$), `union` ($S_1{\cup}S_2$)
  - Produkträume (+ Mengen, Folgen, ...): Komponentenweise Komposition
  - Natürlichen Zahlen ($\mathbb{N}$): Nachfolgerfunktion

Unterstützungstechniken der Entwurfsstrategie

- **Heuristische Fixierung von g**
  - Falls D'$=D$, wähle g:=f R':=R, O':=O und I':=I
  - Falls D'$\neq D$, wähle g:=id R':=D', $O[y'z']$':=$z' = y'$ und $I'[y']$:=`true`

  Analoge Entscheidungen, falls R'=R durch Wahl von *Compose*

- **Inferenzmechanismus: Abgeleitete Vorbedingungen**
  - Bestimme Voraussetzungen für Gültigkeit einer Formel durch zielgerichteten Einsatz von Lemmas entsprechend vorkommender Begriffe
  - Voraussetzungen sind verbleibende Vorbedingungen beim Beweisversuch

- **Inferenzmechanismus: Fallanalyse**
  - Erzeugung von Alternativen über existierende Prädikate
  - Partielle Auswertung der Einzelfälle

  Liefert Programmstücke mit Fallunterscheidungen

- **Inferenzmechanismus: Operator match**
  - Synthese von Programmstücken durch Anpassung an bekannte Lösungen

Divide & Conquer Synthese von Sortieralgorithmen

`FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true`
`  RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)`

---

1. Wähle $L \succ L'$ $\equiv$ $|L|>|L'|$

$\quad$ *Decompose* $\equiv$ ListSplit $\quad (O_D[L,L_1,L_2] \equiv L=L_1\circ L_2, \quad D' \equiv \texttt{Seq}(\alpha))$

2. Konstruiere Hilfsfunktion g $\equiv$ sort $\quad \mapsto O' \equiv O, \quad I'[L'] \equiv$ true

3. Verifiziere *Decompose*

`FUNCTION` f_d`(L:Seq(ℤ)):Seq(ℤ)×Seq(ℤ) WHERE` $\neg$*primitive*[L]
`  RETURNS` L_1,L_2 `SUCH THAT` $L\succ L_1 \wedge L\succ L_2 \wedge L_1\circ L_2=L$

Vorbedingung für Korrektheit $\quad$ *primitive*[L] $\equiv$ L=[]

4. Konstruiere *Compose*

$L\succ L_1 \wedge L\succ L_2 \wedge L_1\circ L_2=L \wedge SORT(L_1,S_1) \wedge SORT(L_2,S_2) \wedge O_C[S_1,S_2,S] \Rightarrow SORT(L,S)$

liefert als Spezifikation für *Compose* $\quad \mapsto$ *Synthese folgt später*

`FUNCTION` $f_c(S_1,S_2$`:Seq(ℤ)×Seq(ℤ)):Seq(ℤ) WHERE ordered(`S_1`)` $\wedge$ `ordered(`S_2`)`
`  RETURNS S SUCH THAT ordered(S) ∧ rearranges(S,` $S_1\circ S_2$`)`

5. Konstruiere *Directly-solve*

`FUNCTION` f_p`(L:Seq(ℤ)):Seq(ℤ) WHERE L=[]`
`  RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)`

Liefert $\quad$ *Directly-solve*[L] $\equiv$ []

Divide & Conquer Synthese von Sortieralgorithmen

Ermittelte Komponenten

$$
\begin{array}{lll}
\mathit{Decompose}[L] & \equiv & \texttt{ListSplit}[L] \\
 & \equiv & \texttt{([}L_i\texttt{|i}\in\texttt{[1..|L|÷2]], [}L_i\texttt{|i}\in\texttt{[1+|L|÷2..|L|]])} \\
g & \equiv & \texttt{sort} \\
\mathit{Compose}[S_1, S_2] & \equiv & \texttt{merge}(S_1,S_2) \\
\mathit{Directly\text{-}solve}[L] & \equiv & \texttt{[]} \\
\mathit{primitive}[L] & \equiv & L\texttt{=[]}
\end{array}
$$

$\mapsto$ *nächste Folie*

6. Instantiiere das Divide & Conquer Schema

```
FUNCTION sort(L:Seq(ℤ)):Seq(ℤ) WHERE true
  RETURNS S SUCH THAT rearranges(L,S) ∧ ordered(S)
≡ if L=[] then []
          else let L1.L2 = ( [Li|i∈[1..|L|÷2] ],
                             [Li|i∈[1+|L|÷2..|L|] ] )
               in merge(sort(L1),sort(L2))
```

Algorithmus ist korrekt durch Konstruktion

Divide & Conquer Synthese der merge-Funktion

$\texttt{FUNCTION merge}(S_1,S_2{:}\texttt{Seq}(\mathbb{Z})\times\texttt{Seq}(\mathbb{Z}))\texttt{:Seq}(\mathbb{Z})\ \texttt{WHERE ordered}(S_1)\wedge\texttt{ordered}(S_2)$
$\quad\texttt{RETURNS S SUCH THAT ordered(S)}\wedge\texttt{rearranges}(S, S_1\circ S_2)$

---

1. Wähle *Compose* $\equiv$ `cons` $\qquad (O_C[a,S',S]\equiv S{=}a\,.\,S',\ \ R'\equiv\mathbb{Z})$

2. g **kann nicht** `merge` **sein Wähle** g $\equiv$ `id` $\qquad (O'[a,a']\equiv a{=}a')$

3. Konstruiere $\succ$ **auf** $\texttt{Seq}(\mathbb{Z})\times\texttt{Seq}(\mathbb{Z})$**:**

– $(S_1,S_2)\succ(S_1,S_2)\ \equiv\ |S_1|>|S_1'|\ \vee\ (|S_1|=|S_1'|\ \wedge\ |S_2|>|S_2'|)$

– Kombination Längenordnung für Listen + lexikographische Ordnung für Produkte

4. Konstruiere *Decompose* **mit Axiom 2 und 3**

$O_D(S_1,S_2,a',S_1',S_2)\ \wedge\ a{=}a'\wedge\texttt{MERGE}(S_1',S_2',S')\ \wedge\ S{=}a.S'\ \ \Rightarrow\ \texttt{MERGE}(S_1,S_2,S)$

liefert als Spezifikation und Lösung für *Decompose*

$\quad\texttt{FUNCTION}\ f_d(S_1,S_2{:}\texttt{Seq}(\mathbb{Z})\times\texttt{Seq}(\mathbb{Z}))\texttt{:}\mathbb{Z}\times\texttt{Seq}(\mathbb{Z})\times\texttt{Seq}(\mathbb{Z})$
$\quad\quad\texttt{WHERE ordered}(S_1)\ \wedge\ \texttt{ordered}(S_2)\ \wedge\ S_1\neq[]\ \wedge\ S_2\neq[]$
$\quad\quad\texttt{RETURNS}\ a',S_1',S_2'$
$\quad\quad\texttt{SUCH THAT rearranges}(a.(S_1'\circ S_2'),S_1\circ S_2)$
$\quad\quad\quad\quad\wedge\ \forall x\in S_1'\circ S_2'.\ a\leq x\ \wedge\ (S_1,S_2)\succ(S_1',S_2')$
$\quad=\texttt{let}\ x_1.S_1'{=}S_1\ \texttt{and}\ x_2.S_2'{=}S_2\ \texttt{in if}\ x_1\leq x_2\ \texttt{then}\ (x_1,S_1',S_2)\ \texttt{else}\ (x_2,S_1,S_2')$

(Lösung durch Fallanalyse: `a'` muß Minimum von $\texttt{hd}(S_1)$ und $\texttt{hd}(S_2)$ sein)

DIVIDE & CONQUER SYNTHESE DER `merge`-FUNKTION

$\texttt{FUNCTION merge}(S_1,S_2{:}\,\texttt{Seq}(\mathbb{Z})\times\texttt{Seq}(\mathbb{Z}))\texttt{:Seq}(\mathbb{Z})\ \texttt{WHERE ordered}(S_1)\wedge\texttt{ordered}(S_2)$
$\quad\texttt{RETURNS S SUCH THAT ordered(S)}\wedge\texttt{rearranges}(S,S_1\circ S_2)$

---

5. Vorbedingung für Korrektheit von *Decompose* **ist** $S_1\neq[]\ \wedge\ S_2\neq[]$

– Liefert *primitive* und Spezifikation für *Directly-solve*

$\texttt{FUNCTION}\ f_p(S_1,S_2{:}\,\texttt{Seq}(\mathbb{Z})\times\texttt{Seq}(\mathbb{Z}))\texttt{:Seq}(\mathbb{Z})$
$\quad\texttt{WHERE ordered}(S_1)\wedge\texttt{ordered}(S_2)\wedge\ (S_1=[]\ \vee\ S_2=[])$
$\quad\texttt{RETURNS S SUCH THAT ordered(S)}\wedge\texttt{rearranges}(S,S_1\circ S_2)$
$=\ \texttt{if}\ S_1=[]\ \texttt{then}\ S_2\ \texttt{else}\ S_1$

6. Instantiierter Divide & Conquer Algorithmus

$\texttt{FUNCTION merge}(S_1,S_2{:}\,\texttt{Seq}(\mathbb{Z})\times\texttt{Seq}(\mathbb{Z}))\texttt{:Seq}(\mathbb{Z})$
$\quad\texttt{WHERE ordered}(S_1)\wedge\texttt{ordered}(S_2)$
$\quad\texttt{RETURNS S SUCH THAT ordered(S)}\wedge\texttt{rearranges}(S,S_1\circ S_2)$
$=\ \texttt{if}\ S_1=[]\ \texttt{then}\ S_2$
$\qquad\texttt{elseif}\ S_2=[]\ \texttt{then}\ S_1$
$\qquad\quad\texttt{else let}\ x_1.S_1'=S_1\ \texttt{and}\ x_2.S_2'=S_2$
$\qquad\qquad\texttt{in if}\ x_1\leq x_2\ \texttt{then}\ x_1.\texttt{merge}(S'_1,S_2)\ \texttt{else}\ x_2.\texttt{merge}(S_1,S_2')$

Erzeugung Alternativer Sortieralgorithmen

- **Wähle einfachere Dekomposition**
  - *Decompose* $\equiv$ `HdTl`, $g \equiv$ `id`
  - *primitive*[`L`] $\equiv$ `L=[]`, *Directly-solve*[`L`] $\equiv$ `[]`
  - *Compose*[`a`, `S`] $\equiv$ `ordered_insert(a,S)`

Sortieren durch Einfügen

- **Wähle einfache Komposition**
  - *Compose*[`a`, `S`] $\equiv$ `a.S`, $g \equiv$ `id`
  - *primitive*[`L`] $\equiv$ `L=[]`, *Directly-solve*[`L`] $\equiv$ `[]`
  - *Decompose*[`L`] $\equiv$ `let m=min(L) in (m, L-m)`

Sortieren durch Auswahl

- **Wähle binare Komposition**
  - *Compose*[S_1, S_2] $\equiv$ $S_1 \circ S_2$, $g \equiv$ `sort`
  - *primitive*[`L`] $\equiv$ `L=[]`, *Directly-solve*[`L`] $\equiv$ `[]`
  - *Decompose*[`L`] $\equiv$ let let $a = L_{[|L|/2]}$ in $(L_{<a}, L_{\geq a}])$

Naives Quicksort

$\Downarrow$

Flexible Strategie mit vielfältigen Anwendungen