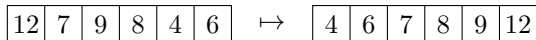


# Sortieren

Ein einfaches Problem:



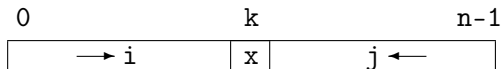
Viele Lösungen:

- Quicksort
- Insertion Sort
- Merge Sort
- Bubble Sort
- ...

# Quicksort

1. Wähle ein Element  $x$  der Eingabeliste.
2. Partitioniere die verbleibenden Elemente in zwei Teil-Listen:
  - ▶ eine mit allen Elementen kleiner als  $x$ , und
  - ▶ eine mit allen Elementen größer oder gleich  $x$ .
3. Sortiere die beiden Teil-Listen rekursiv.
4. Die Ausgabeliste ist die Konkatenation von:
  - ▶ der sortierten ersten Teil-Liste,
  - ▶ dem Element  $x$ , und
  - ▶ der sortierten zweiten Teil-Liste.

Umsetzung:



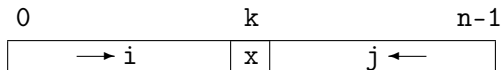
Beispiel:

2	15	7	9	12	4	11
---	----	---	---	----	---	----

# Quicksort

1. Wähle eine Element  $x$  der Eingabeliste.
2. Partitioniere die verbleibenden Elemente in zwei Teil-Listen:
  - ▶ eine mit allen Elementen kleiner als  $x$ , und
  - ▶ eine mit allen Elementen größer oder gleich  $x$ .
3. Sortiere die beiden Teil-Listen rekursiv.
4. Die Ausgabeliste ist die Konkatenation von:
  - ▶ der sortierten ersten Teil-Liste,
  - ▶ dem Element  $x$ , und
  - ▶ der sortierten zweiten Teil-Liste.

Umsetzung:



Beispiel:

2	4	7	9	12	15	11
---	---	---	---	----	----	----

## Quicksort (1)

### Aufgabe:

- gegeben eine Liste von zu ordnenden Elementen
- Konstruiere eine neue Liste, in der dieselben Elemente in **aufsteigend sortierter** Reihenfolge auftreten!

```
quicksort [] = []  
quicksort (x:xs) = quicksort [ y | y <- xs, y < x ]  
                ++ [ x ]  
                ++ quicksort [ y | y <- xs, y >= x ]
```

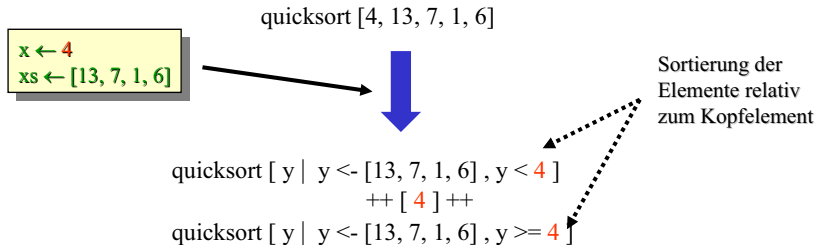
wichtig: Die Elemente müssen nicht unbedingt Zahlen sein – auch auf Objekten anderer Datentypen sind die Vergleichsoperatoren definiert.

z.B.:

```
> 'a' < 'b'  
True
```

"Quicksort at work": **Sortiere die Liste [4, 13, 7, 1, 6] !**

```
quicksort [ ] = [ ]  
quicksort (x:xs) = quicksort [ y | y <- xs, y < x ]  
                ++ [ x ]  
                ++ quicksort [ y | y <- xs, y >= x ]
```



## Quicksort (3)

```
quicksort [ ] = [ ]  
quicksort (x:xs) = quicksort [ y | y <- xs, y < x ]  
                ++ [ x ]  
                ++ quicksort [ y | y <- xs, y >= x ]
```

```
quicksort [ y | y <- [13, 7, 1, 6], y < 4 ]  
                ++ [ 4 ] ++  
quicksort [ y | y <- [13, 7, 1, 6], y >= 4 ]
```



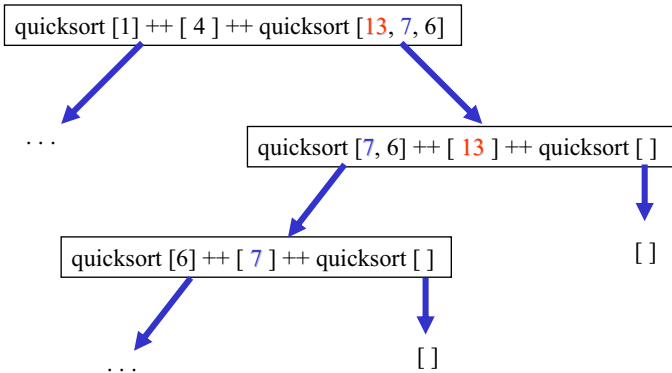
```
quicksort [1] ++ [ 4 ] ++ quicksort [13, 7, 6]
```

**Aufspalten** der Aufgabe in zwei **Unterprobleme**, die für sich jeweils "leichter" zu lösen sind als das Ausgangsproblem:

**"Divide and Conquer"-Prinzip**

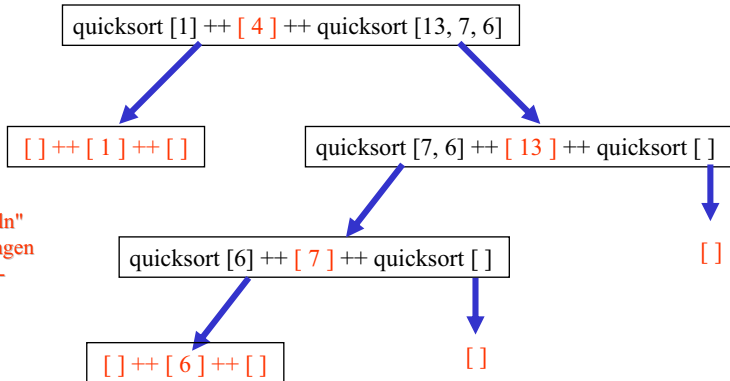
## Quicksort (4)

```
quicksort [] = []  
quicksort (x:xs) = quicksort [ y | y <- xs, y < x ]  
                  ++ [ x ]  
                  ++ quicksort [ y | y <- xs, y >= x ]
```



## Quicksort (6)

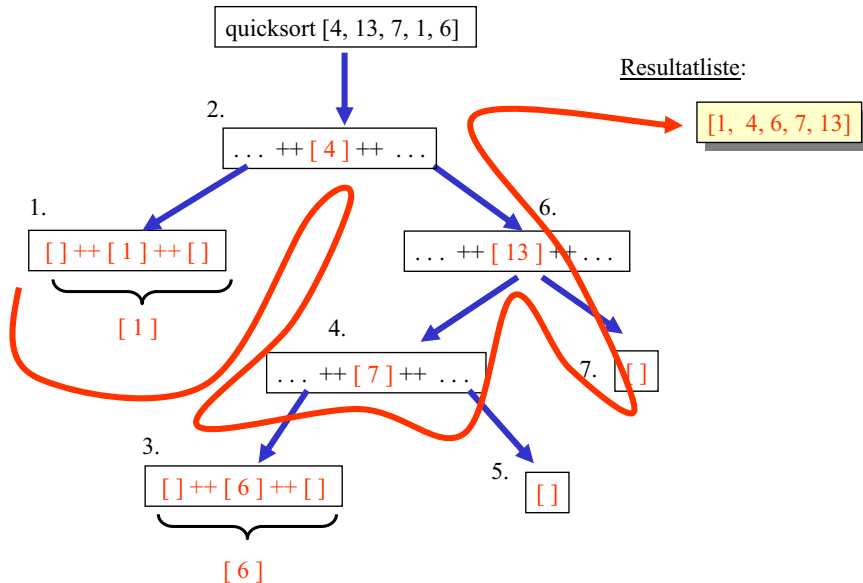
```
quicksort [] = []  
quicksort (x:xs) = quicksort [ y | y <- xs, y < x ]  
                ++ [ x ]  
                ++ quicksort [ y | y <- xs, y >= x ]
```



"Aufsammeln"  
aller Lösungen  
aller Unter-  
probleme



## Quicksort (7)



## Alternativen

- Beachte:
- Der Algorithmus Quicksort benutzt folgende Schlüsseloperation (zur Partitionsbildung):

*compare* ::  $\tau \rightarrow \tau \rightarrow \text{Bool}$

- Das Gleiche gilt für Algorithmen wie Insertion Sort, Merge Sort, ...

Aber: Knuth betrachtet auch eine **stärker eingeschränkte** Klasse von Sortier-Algorithmen, stattdessen auf folgender Operation basierend:

*cswap* ::  $(\tau, \tau) \rightarrow (\tau, \tau)$

## Bitonic Sort

1. Teile die Eingabeliste in zwei Teil-Listen gleicher Länge.
2. Sortiere die beiden Teil-Listen rekursiv, die zweite in umgekehrter Richtung.
3. Füge die sortierten Teil-Listen wie folgt zusammen:
  - 3.1 Wende *cswap* auf Paare von Elementen an einander entsprechenden Positionen an.
  - 3.2 Teile jede der resultierenden Listen in der Mitte.
  - 3.3 Füge die resultierenden Paare von Listen rekursiv zusammen.
  - 3.4 Konkateniere die Ergebnisse.

### Beachte:

- Funktioniert so nur für Listen von Zweierpotenz-Länge.
- Komplexität ist  $O(n \cdot \log(n)^2)$ .
- Besonders geeignet für Hardware- und parallele Implementierungen.
- Korrektheit ist nicht offensichtlich!

## Bitonic Sort in Haskell

```
bitonic :: ((a,a) -> (a,a)) -> [a] -> [a]
bitonic cswap [] = []
bitonic cswap [x] = [x]
bitonic cswap xs =
  let k          = length xs `div` 2
      (ys,zs)   = splitAt k xs
      ys'       = bitonic cswap ys
      zs'       = bitonic ((\ (u,v) -> (v,u)) . cswap) zs
      merge [u] [v] = case cswap (u,v) of (p,q) -> [p,q]
      merge us vs = let (us',vs') = unzip (map cswap
                                                (zip us vs))
                      l           = length us' `div` 2
                      (us1,us2) = splitAt l us'
                      (vs1,vs2) = splitAt l vs'
                      in (merge us1 us2) ++ (merge vs1 vs2)
  in merge ys' zs'
```

## Knuths 0-1-Prinzip

**Informell:** Wenn ein “comparison-swap Algorithmus” Booleans korrekt sortiert, dann auch ganze Zahlen.

**Formal:** Verwende Haskell. Es seien

`sort` :: (( $\alpha$ ,  $\alpha$ )  $\rightarrow$  ( $\alpha$ ,  $\alpha$ ))  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]

`f` :: (Int, Int)  $\rightarrow$  (Int, Int)

`f` (x, y) = *if* x > y *then* (y, x) *else* (x, y)

`g` :: (Bool, Bool)  $\rightarrow$  (Bool, Bool)

`g` (x, y) = (x && y, x || y)

Wenn für jedes `xs :: [Bool]`, `sort g xs` das korrekte Ergebnis liefert, dann liefert für jedes `xs :: [Int]`, `sort f xs` das korrekte Ergebnis.

# Knuths 0-1-Prinzip

**Informell:** Wenn ein “comparison-swap Algorithmus” Booleans korrekt sortiert, dann auch ganze Zahlen.

**Formal:** Verwende Haskell. Es seien

`sort` :: (( $\alpha$ ,  $\alpha$ )  $\rightarrow$  ( $\alpha$ ,  $\alpha$ ))  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]

`f` :: (Int, Int)  $\rightarrow$  (Int, Int)

`f` (x, y) = if x > y then (y, x) else (x, y)

`g` :: (Bool, Bool)  $\rightarrow$  (Bool, Bool)

`g` (x, y) = (x && y, x || y)

Wenn  $\forall xs :: [\text{Bool}], ys = \text{sort } g \text{ } xs. P(xs, ys) \wedge Q(ys)$ ,  
dann  $\forall xs :: [\text{Int}], ys = \text{sort } f \text{ } xs. P(xs, ys) \wedge Q(ys)$ ,  
wobei  $P(xs, ys) := xs$  und  $ys$  enthalten die gleichen  
Elemente in gleicher Anzahl

$Q(ys) := ys$  ist sortiert

## Verwendung des Theorem-Generators

Eingabe: `sort :: ((a,a)->(a,a))->[a]->[a]`

Ausgabe: `forall t1,t2 in TYPES, h::t1->t2.`

`forall f::(t1,t1)->(t1,t1).`

`forall g::(t2,t2)->(t2,t2).`

`(forall (x,y) in lift_{(,)}(h,h).`

`(f x,g y) in lift_{(,)}(h,h))`

`==> (forall xs::[t1].`

`map h (sort f xs) = sort g (map h xs))`

`lift_{(,)}(h,h)`

`= {(x1,x2),(y1,y2)} | (h x1 = y1)`

`&& (h x2 = y2)}`

## Spezifischer (und intuitiver)

Für alle  $\text{sort} :: ((\alpha, \alpha) \rightarrow (\alpha, \alpha)) \rightarrow [\alpha] \rightarrow [\alpha]$ ,

$f :: (\text{Int}, \text{Int}) \rightarrow (\text{Int}, \text{Int})$ ,  $g :: (\text{Bool}, \text{Bool}) \rightarrow (\text{Bool}, \text{Bool})$  und

$h :: \text{Int} \rightarrow \text{Bool}$ :

$$\begin{array}{ccc} (\text{Int}, \text{Int}) & \xrightarrow{f} & (\text{Int}, \text{Int}) \\ \downarrow h \times h & & \downarrow h \times h \\ (\text{Bool}, \text{Bool}) & \xrightarrow{g} & (\text{Bool}, \text{Bool}) \end{array} \quad = \quad \Rightarrow \quad \begin{array}{ccc} [\text{Int}] & \xrightarrow{\text{sort } f} & [\text{Int}] \\ \downarrow \text{map } h & & \downarrow \text{map } h \\ [\text{Bool}] & \xrightarrow{\text{sort } g} & [\text{Bool}] \end{array}$$

Wenn  $f$  und  $g$  wie zuvor definiert, dann ist die Vorbedingung erfüllt für jedes  $h$  der Form  $h\ x = n < x$  für irgendein  $n :: \text{Int}$ .



# Knuths 0-1-Prinzip

**Informell:** Wenn ein “comparison-swap Algorithmus” Booleans korrekt sortiert, dann auch ganze Zahlen.

**Formal:** Verwende Haskell. Es seien

`sort` :: (( $\alpha$ ,  $\alpha$ )  $\rightarrow$  ( $\alpha$ ,  $\alpha$ ))  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]

`f` :: (Int, Int)  $\rightarrow$  (Int, Int)

`f` (x, y) = *if* x > y *then* (y, x) *else* (x, y)

`g` :: (Bool, Bool)  $\rightarrow$  (Bool, Bool)

`g` (x, y) = (x && y, x || y)

Wenn  $\forall xs :: [\text{Bool}], ys = \text{sort } g \text{ } xs. P(xs, ys) \wedge Q(ys)$ ,  
dann  $\forall xs :: [\text{Int}], ys = \text{sort } f \text{ } xs. P(xs, ys) \wedge Q(ys)$ ,  
wobei  $P(xs, ys) := xs$  und  $ys$  enthalten die gleichen  
Elemente in gleicher Anzahl

$Q(ys) := ys$  ist sortiert

## Beweis von „ $P$ auf $[\text{Bool}]$ impliziert $P$ auf $[\text{Int}]$ “

Zur Erinnerung:  $P(xs, ys) := xs$  und  $ys$  enthalten die gleichen Elemente in gleicher Anzahl

Gegeben:  $\forall xs :: [\text{Bool}], ys = \text{sort } g \text{ } xs. P(xs, ys)$

Zu zeigen:  $\forall xs :: [\text{Int}], ys = \text{sort } f \text{ } xs. P(xs, ys)$

Angenommen, es gäbe  $us :: [\text{Int}], vs = \text{sort } f \text{ } us$  mit  $\neg P(us, vs)$ . Dann gäbe es ein kleinstes  $n$  so dass die Anzahlen von  $n$  in  $us$  und  $vs$  nicht gleich sind. Dann wären für  $h \ x = n < x$  die Anzahlen von False in  $(\text{map } h \ us)$  und  $(\text{map } h \ vs)$  verschieden. Dies steht jedoch im Widerspruch zur Vorbedingung mit:

$$xs = \text{map } h \ us$$

$$ys = \text{sort } g \ (\text{map } h \ us) = \text{map } h \ (\text{sort } f \ us) = \text{map } h \ vs$$

## Beweis von „Q auf [Bool] impliziert Q auf [Int]“

Zur Erinnerung:  $Q(ys) := ys$  ist sortiert

Gegeben:  $\forall xs :: [Bool], ys = \text{sort } g \text{ } xs. Q(ys)$

Zu zeigen:  $\forall xs :: [Int], ys = \text{sort } f \text{ } xs. Q(ys)$

Angenommen, es gäbe  $us :: [Int], vs = \text{sort } f \text{ } us$  mit  $\neg Q(vs)$ .

Dann gäbe es  $n < m$  so dass ein  $m$  in  $vs$  vor einem  $n$  vorkommt.

Dann würde für  $h \ x = n < x$  ein True in  $(\text{map } h \ vs)$  vor einem False vorkommen. Dies steht jedoch im Widerspruch zur Vorbedingung mit:

$$xs = \text{map } h \ us$$

$$ys = \text{sort } g \ (\text{map } h \ us) = \text{map } h \ (\text{sort } f \ us) = \text{map } h \ vs$$

## Knuths 0-1-Prinzip

**Informell:** Wenn ein “comparison-swap Algorithmus” Booleans korrekt sortiert, dann auch ganze Zahlen.

**Formal:** Verwende Haskell. Es seien

`sort` :: (( $\alpha$ ,  $\alpha$ )  $\rightarrow$  ( $\alpha$ ,  $\alpha$ ))  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]

`f` :: (Int, Int)  $\rightarrow$  (Int, Int)

`f` (x, y) = *if* x > y *then* (y, x) *else* (x, y)

`g` :: (Bool, Bool)  $\rightarrow$  (Bool, Bool)

`g` (x, y) = (x && y, x || y)

Wenn für jedes `xs` :: [Bool], `sort g xs` das korrekte Ergebnis liefert, dann liefert für jedes `xs` :: [Int], `sort f xs` das korrekte Ergebnis.

## War's das?

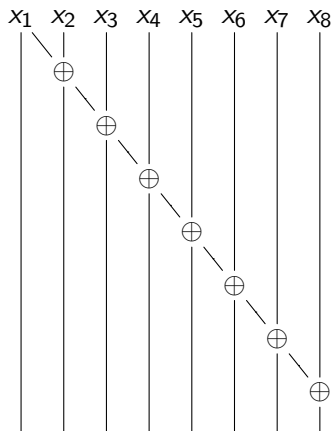
- Knuths 0-1-Prinzip erlaubt eine Reduktion der Korrektheit von Algorithmen, zum comparison-swap Sortieren, für Eingaben über einem unendlichen Bereich auf die Korrektheit über einem endlichen Bereich von Werten.
- Freie Theoreme erlauben einen besonders eleganten Beweis dieses Prinzips [Day et al. 1999].
- Geht Ähnliches für andere Klassen von Algorithmen?
- Gute Kandidaten: Algorithmen, die über irgendeiner Operation parametrisiert sind, so wie  $cswap :: (\alpha, \alpha) \rightarrow (\alpha, \alpha)$  im Fall des Sortierens.

# Parallele Präfixberechnung

**Gegeben:** Eingaben  $x_1, \dots, x_n$  und eine assoziative Operation  $\oplus$

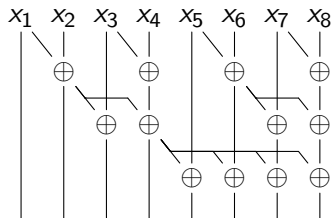
**Aufgabe:** Berechne die Werte  $x_1, x_1 \oplus x_2, \dots, x_1 \oplus x_2 \oplus \dots \oplus x_n$

**Lösung:**

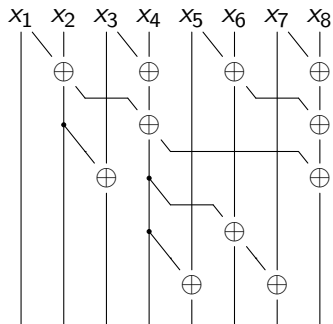


# Parallele Präfixberechnung

Alternativ:



Oder:



Oder: ...

# In Haskell

Funktionen des Typs:

```
scanl1 :: ( $\alpha \rightarrow \alpha \rightarrow \alpha$ )  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]
```

Zum Beispiel, à la [Sklansky 1960]:

```
sklansky :: ( $\alpha \rightarrow \alpha \rightarrow \alpha$ )  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]
```

```
sklansky ( $\oplus$ ) [x] = [x]
```

```
sklansky ( $\oplus$ ) xs = us ++ vs
```

```
  where t      = ((length xs) + 1) 'div' 2
```

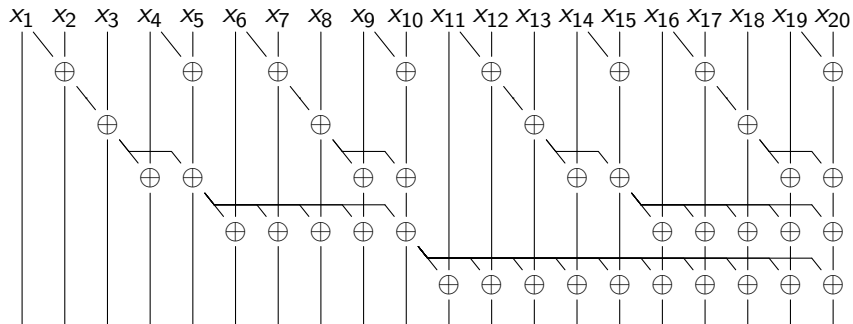
```
        (ys, zs) = splitAt t xs
```

```
        us      = sklansky ( $\oplus$ ) ys
```

```
        vs      = [(last us)  $\oplus$  v | v  $\leftarrow$  sklansky ( $\oplus$ ) zs]
```



## Sklanskys Methode:



## Oder à la [Brent & Kung 1980]

`brentKung` ::  $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha]$

`brentKung`  $(\oplus)$   $[x] = [x]$

`brentKung`  $(\oplus)$   $xs = odds (riffle (par (unriffle (evens xs))))$

**where**  $evens [] = []$

$evens [x] = [x]$

$evens (x : y : zs) = [x, x \oplus y] ++ evens zs$

$unriffle [] = ([], [])$

$unriffle [x] = ([x], [])$

$unriffle (x : y : zs) = (x : xs, y : ys)$

**where**  $(xs, ys) = unriffle zs$

$par (xs, ys) = (xs, brentKung (\oplus) ys)$

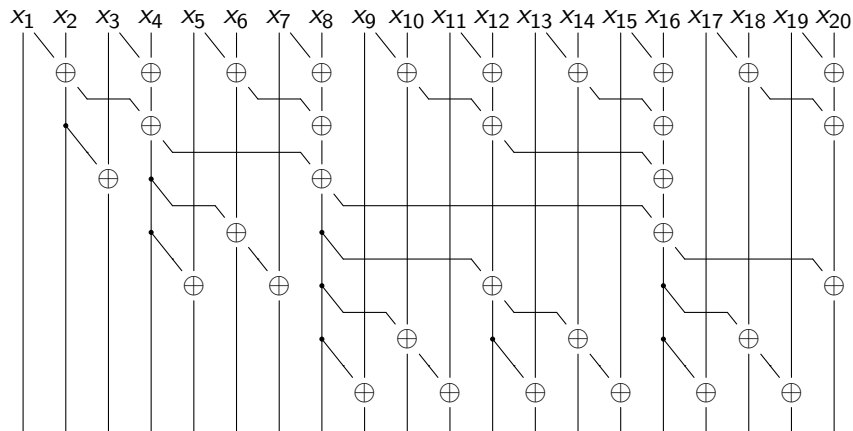
$riffle ([], []) = []$

$riffle ([x], []) = [x]$

$riffle (x : xs, y : ys) = x : y : riffle (xs, ys)$

$odds (x : xs) = x : evens xs$

## Brent & Kungs Methode:



**Gesucht:** Analyse- und Verifikationstechniken,  
systematischer Ansatz zum Testen, ...

# Untersuchung nur gewisser Instanzen

## Knuths 0-1-Prinzip

Ist ein comparison-swap Sortieralgorithmus auf den Booleans korrekt, so ist er es auf beliebigen total geordneten Wertemengen.

## Ein Knuth-ähnliches 0-1-Prinzip ?

Ist ein Algorithmus zur parallelen Präfixberechnung auf den Booleans korrekt, für assoziative Operationen, so ist er es auf beliebigen Wertemengen.

Leider nicht !

## Ein Knuth-ähnliches 0-1-2-Prinzip

Gegeben: `scanl1` ::  $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha]$   
`scanl1` ( $\oplus$ ) (x : xs) = go x xs  
    **where** go x [] = [x]  
          go x (y : ys) = x : go (x  $\oplus$  y) ys

`candidate` ::  $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha]$

`data` Three = Zero | One | Two

**Theorem:** Wenn für jedes `xs` :: [Three] und assoziatives  
 $(\oplus)$  :: Three  $\rightarrow$  Three  $\rightarrow$  Three,

`candidate` ( $\oplus$ ) xs = `scanl1` ( $\oplus$ ) xs,

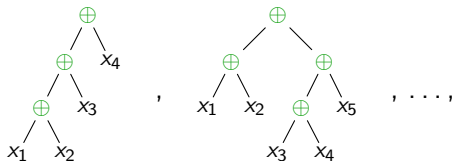
dann gilt das Gleiche für jeden Typ  $\tau$ , `xs` :: [ $\tau$ ], und  
assoziatives  $(\oplus)$  ::  $\tau \rightarrow \tau \rightarrow \tau$ .

## Warum 0-1-2? Und wie?

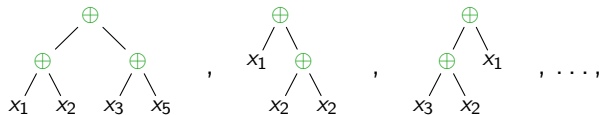
Eine Frage: Was kann **candidate**  $:: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha]$  mit einer Operation  $\oplus$  und Liste  $[x_1, \dots, x_n]$  tun ?

Antwort: Eine neue Liste erzeugen, bestehend aus mittels  $\oplus$  and  $x_1, \dots, x_n$  gebildeten Ausdrücken.  
**Und zwar unabhängig vom  $\alpha$ -Typ !**

Unter diesen Ausdrücken gibt es **gute**:

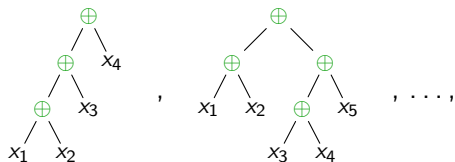


**schlechte:**

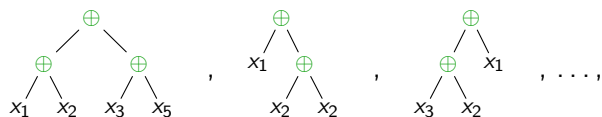


# Warum 0-1-2? Und wie?

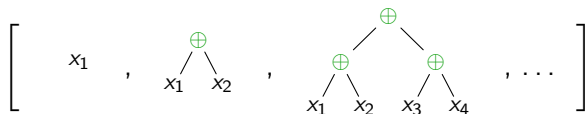
Unter diesen Ausdrücken gibt es **gute**:



**schlechte:**



und welche an der falschen **Position**:



## Wie? So!

Es seien

$\oplus_1$	Zero	One	Two		$\oplus_2$	Zero	One	Two
Zero	Zero	One	Two	und	Zero	Zero	One	Two
One	One	Two	Two		One	One	One	Two
Two	Two	Two	Two		Two	Two	One	Two

Ist `candidate` ( $\oplus_1$ ) korrekt auf jeder Liste der Form

$$[(\text{Zero}, )^* \text{One} (, \text{Zero})^* (, \text{Two})^*]$$

und `candidate` ( $\oplus_2$ ) ist korrekt auf jeder Liste der Form

$$[(\text{Zero}, )^* \text{One}, \text{Two} (, \text{Zero})^*]$$

dann ist `candidate` korrekt für assoziatives  $\oplus$  auf beliebigem Typ.



## Ein Knuth-ähnliches 0-1-2-Prinzip

Gegeben: `scanl1` ::  $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha]$   
`scanl1` ( $\oplus$ ) (x : xs) = go x xs  
    **where** go x [] = [x]  
          go x (y : ys) = x : go (x  $\oplus$  y) ys

`candidate` ::  $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha]$

`data` Three = Zero | One | Two

**Theorem:** Wenn für jedes `xs` :: [Three] und assoziatives  
 $(\oplus)$  :: Three  $\rightarrow$  Three  $\rightarrow$  Three,

`candidate` ( $\oplus$ ) xs = `scanl1` ( $\oplus$ ) xs,

dann gilt das Gleiche für jeden Typ  $\tau$ , `xs` :: [ $\tau$ ], und  
assoziatives  $(\oplus)$  ::  $\tau \rightarrow \tau \rightarrow \tau$ .