Sortieren

Ein einfaches Problem:

Sortieren

Ein einfaches Problem:

Viele Lösungen:

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

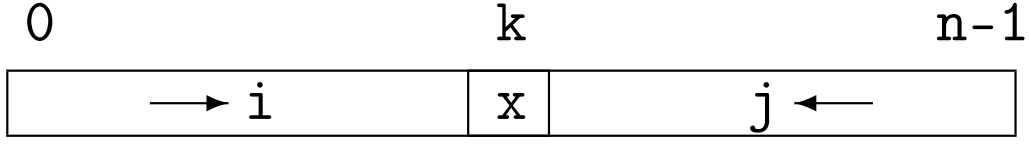
Quicksort

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.

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.

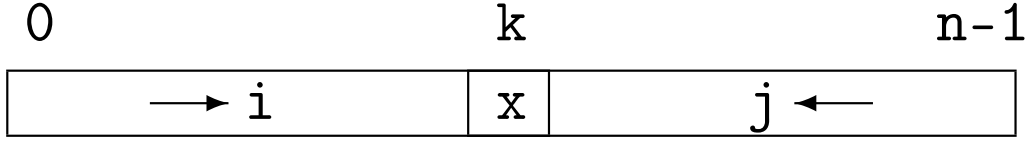
Umsetzung:



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:



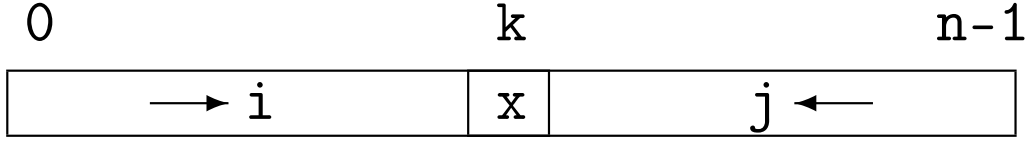
Beispiel:

2	15	7	9	12	4	11

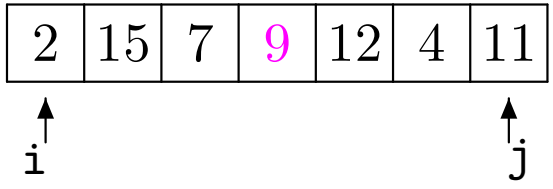
Quicksort

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:



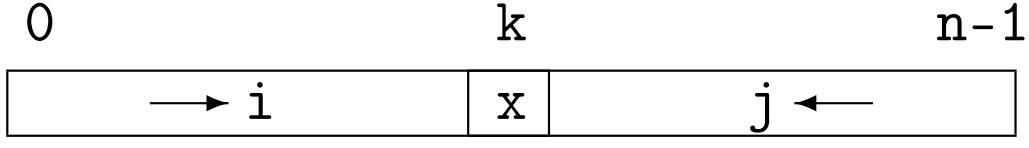
Beispiel:



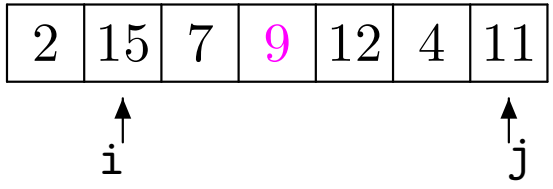
Quicksort

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:



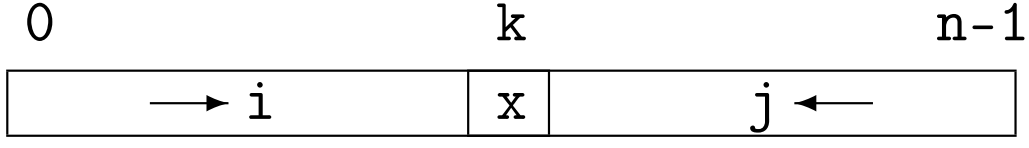
Beispiel:



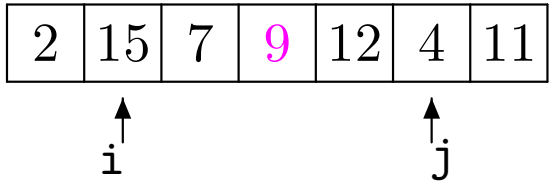
Quicksort

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:



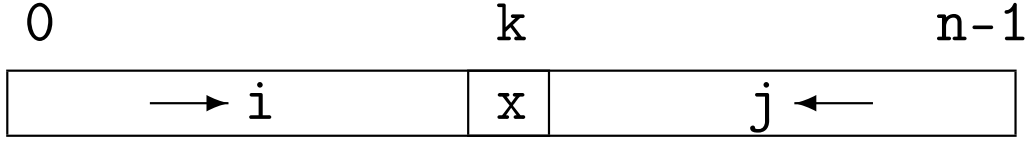
Beispiel:



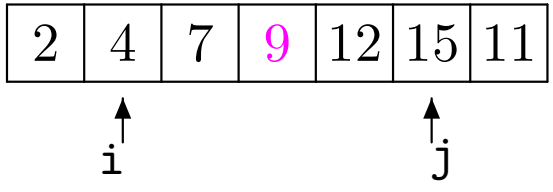
Quicksort

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:



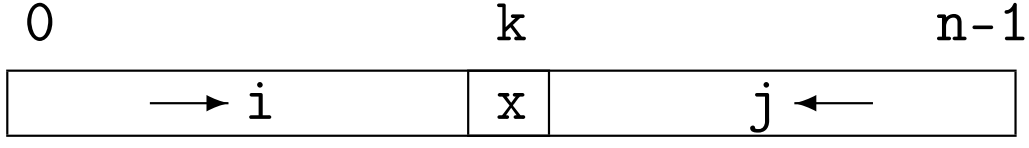
Beispiel:



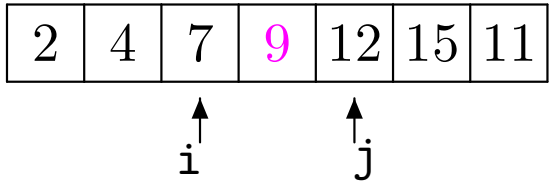
Quicksort

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:



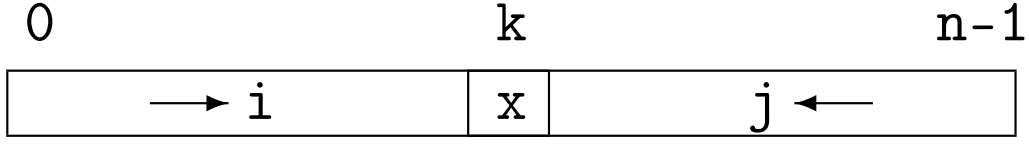
Beispiel:



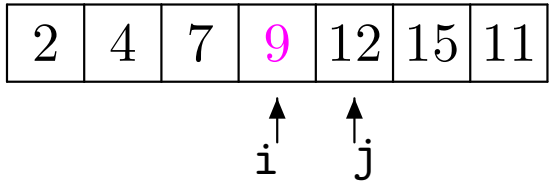
Quicksort

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:



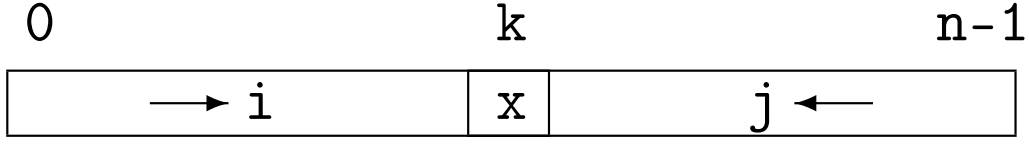
Beispiel:



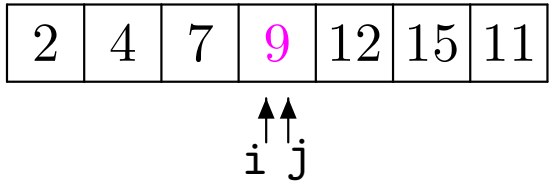
Quicksort

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:



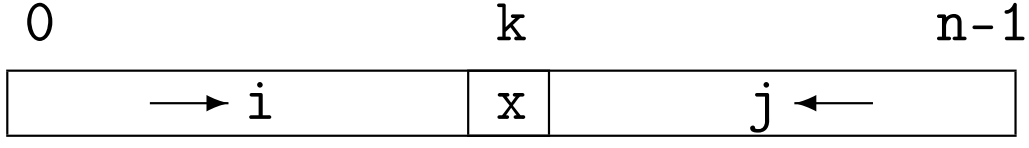
Beispiel:



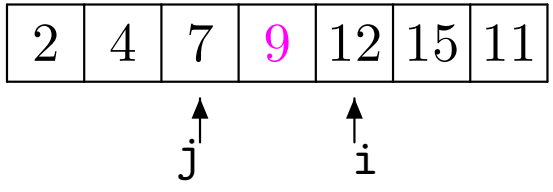
Quicksort

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:



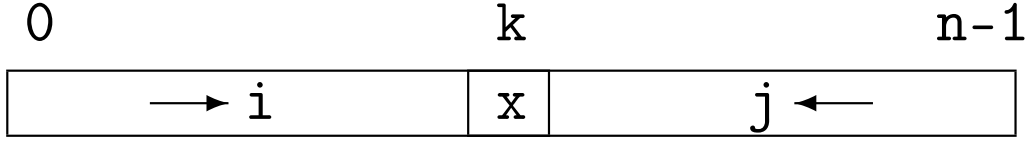
Beispiel:



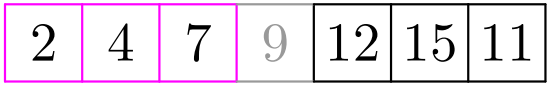
Quicksort

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:



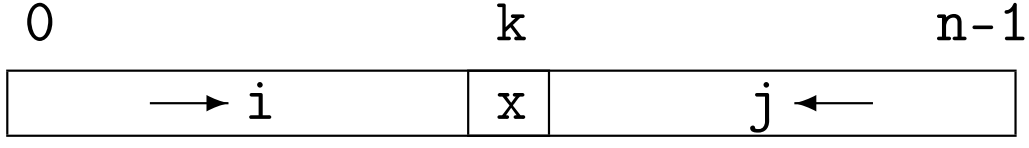
Beispiel:



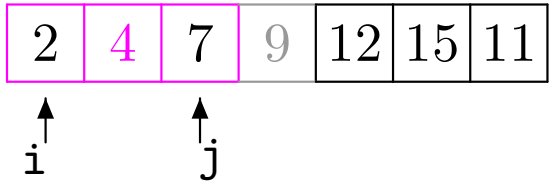
Quicksort

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:



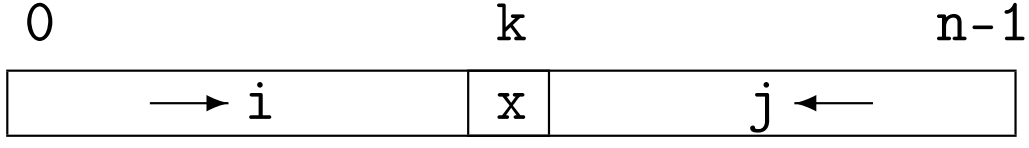
Beispiel:



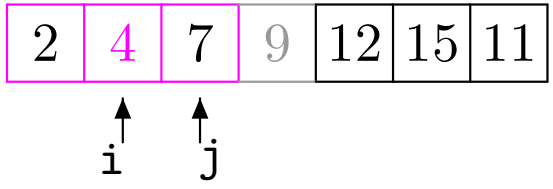
Quicksort

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:



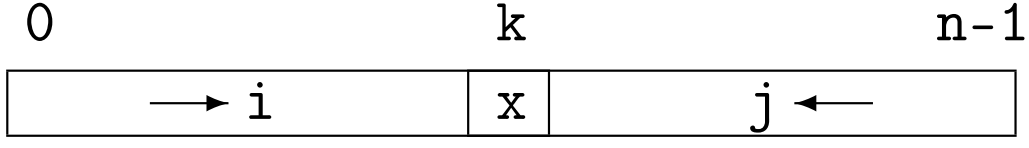
Beispiel:



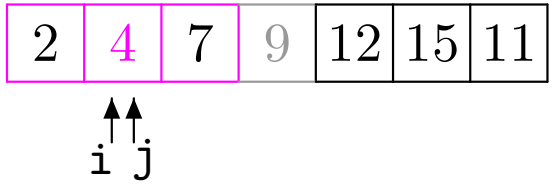
Quicksort

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:



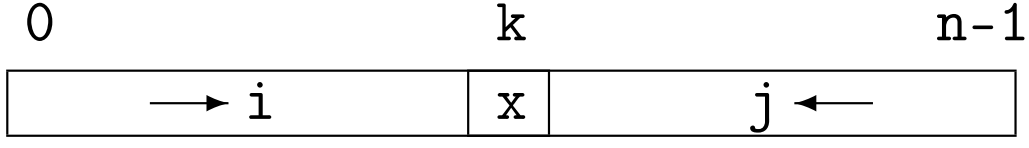
Beispiel:



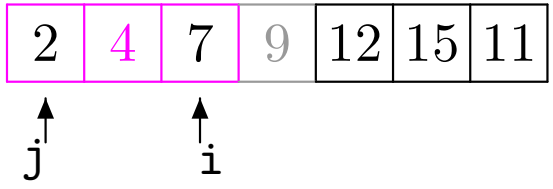
Quicksort

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:



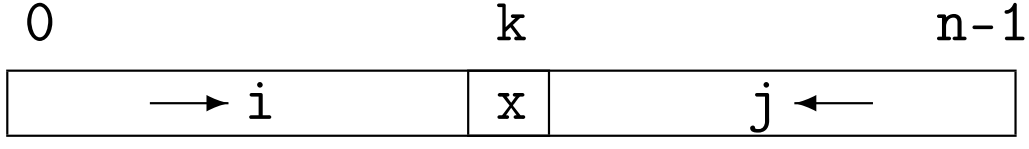
Beispiel:



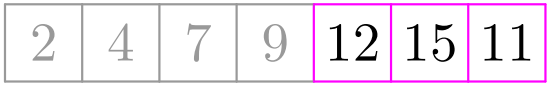
Quicksort

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:



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 Vergleichsoperatorendefiniert.

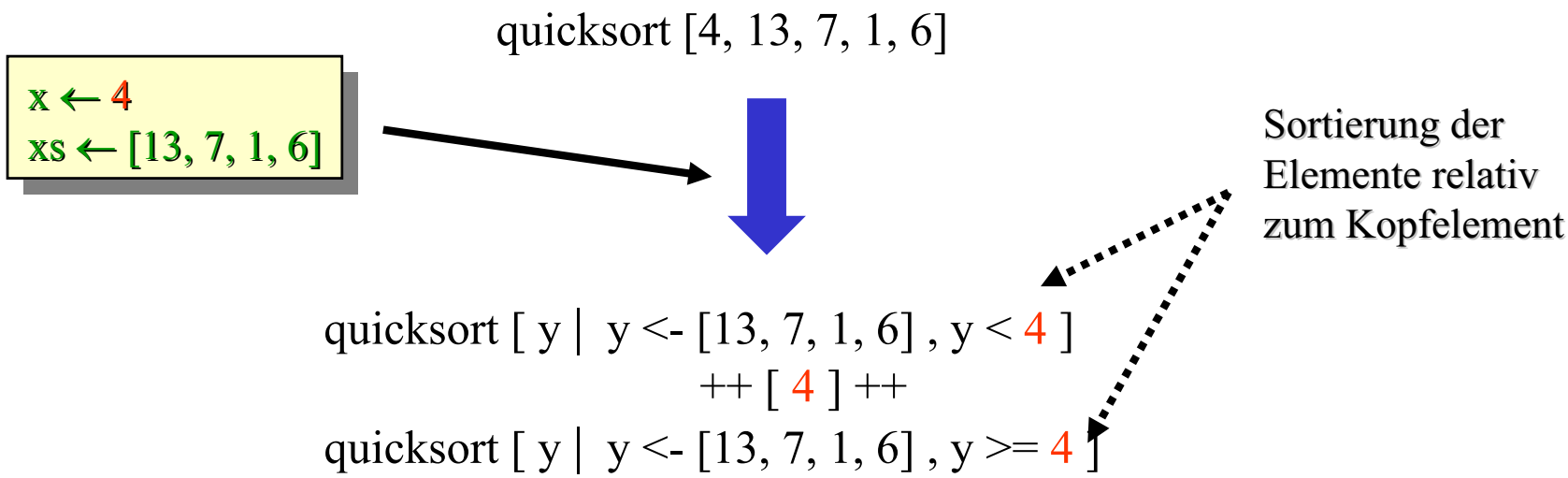
z.B.:

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

Quicksort (2)

"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]

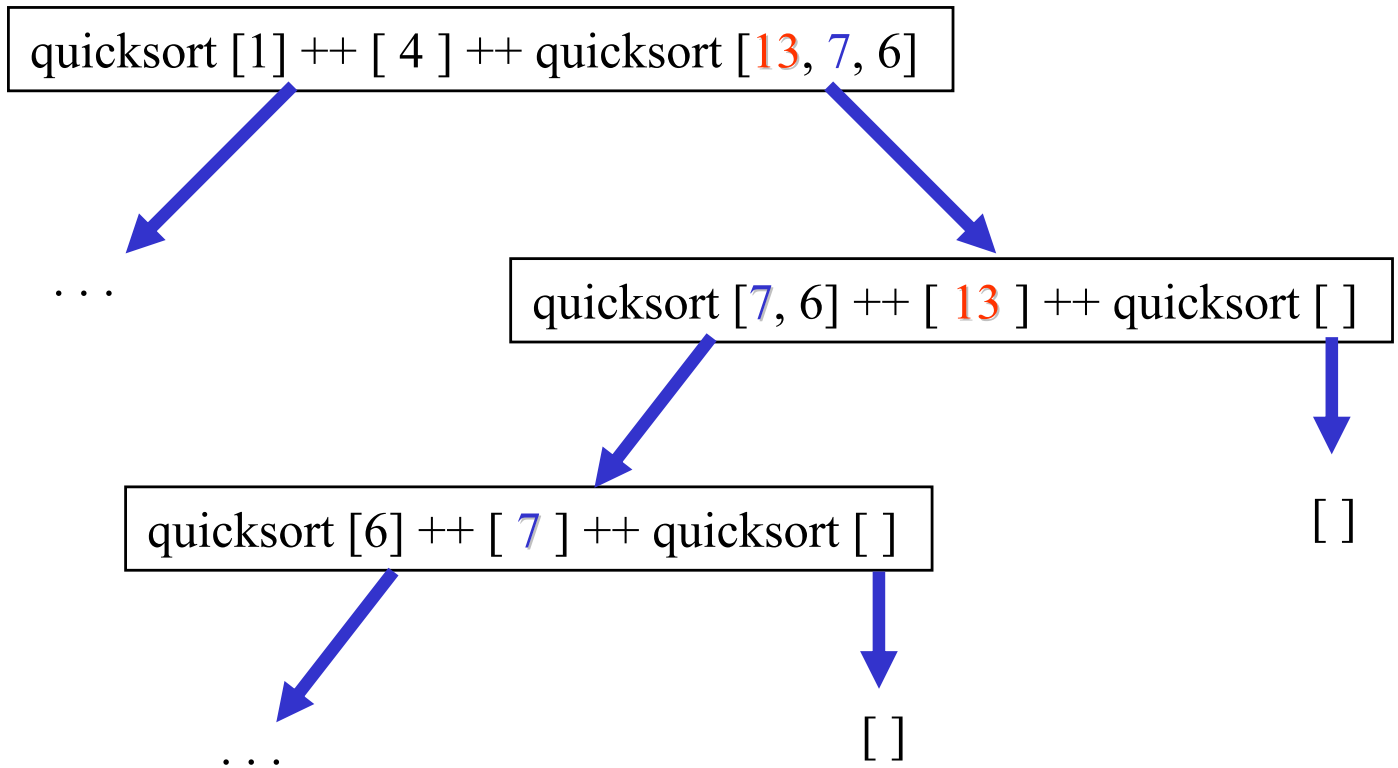
quicksort [1] ++ [4] ++ quicksort [13, 7, 6]

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

"Divide and Conquer"-Prinzip

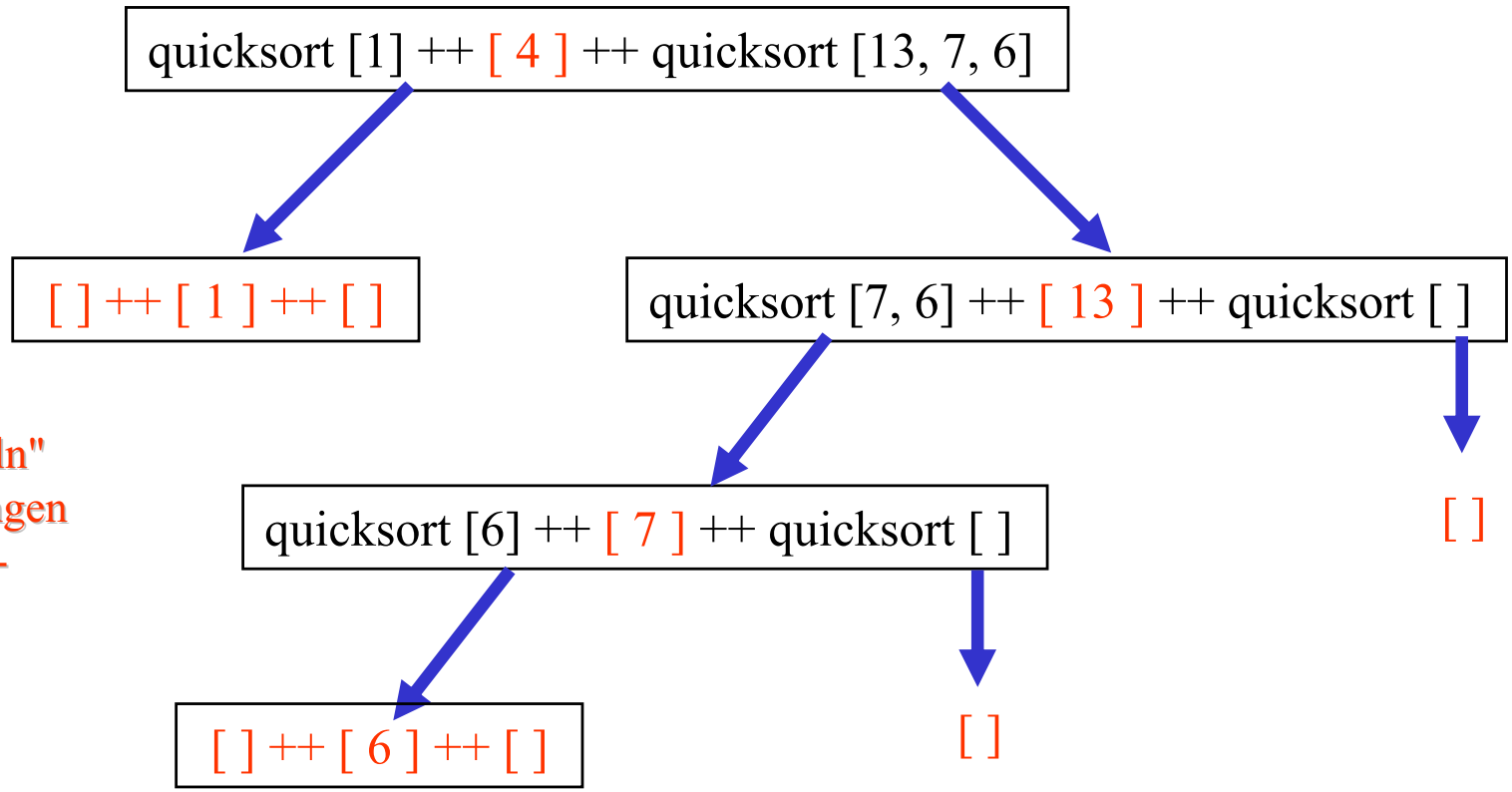
Quicksort (4)

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



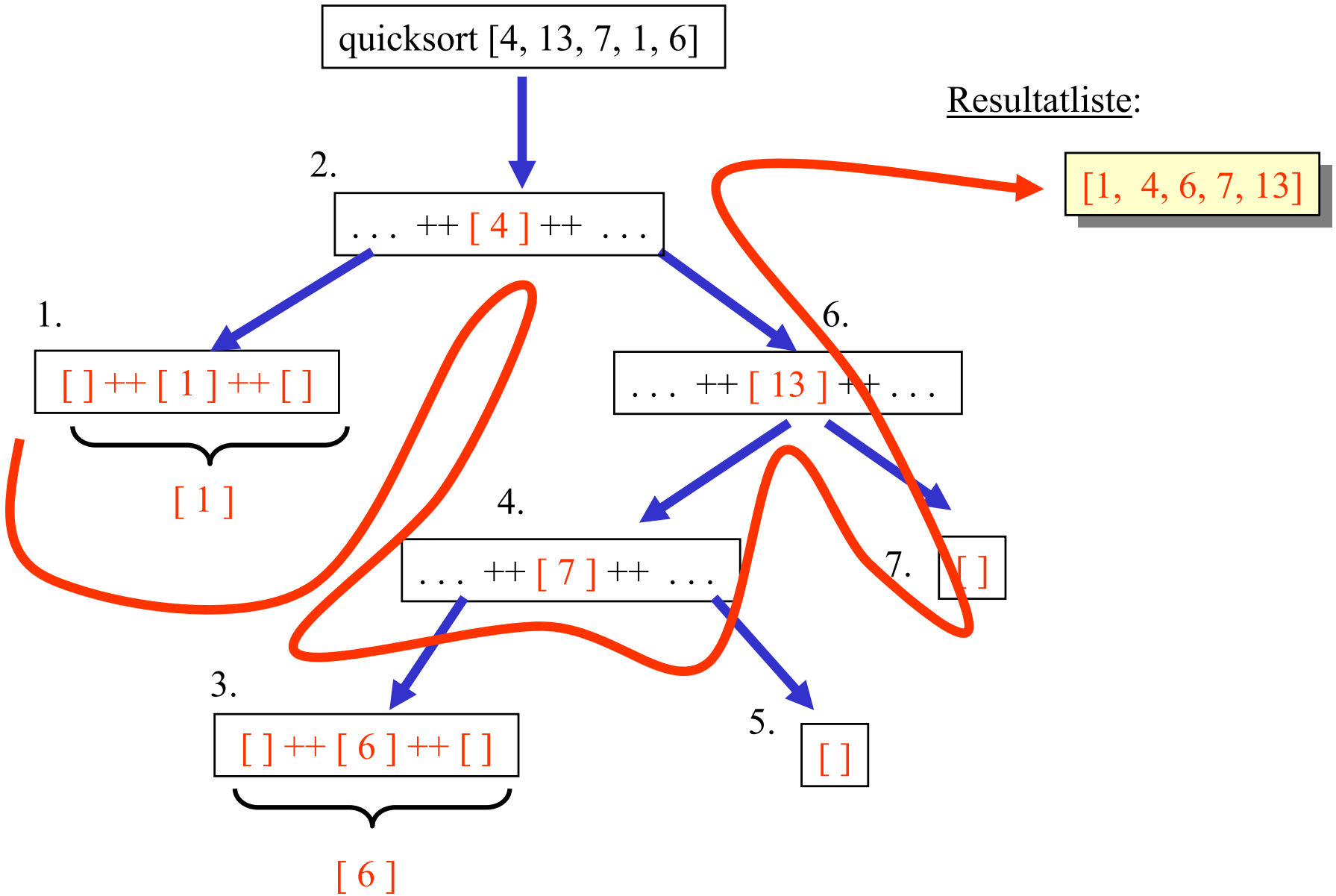
Quicksort (6)

```
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):

$$\mathit{compare} :: \tau \to \tau \to \mathrm{Bool}$$

Alternativen

Beachte:

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

$$\mathit{compare} :: \tau \to \tau \to \mathrm{Bool}$$

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

Alternativen

Beachte:

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

$$\mathit{compare} :: \tau \to \tau \to \mathrm{Bool}$$

- 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:

$$\mathit{cswap} :: (\tau, \tau) \to (\tau, \tau)$$

Alternativen

Beachte:

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

$$\mathit{compare} :: \tau \to \tau \to \mathrm{Bool}$$

- 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:

$$\mathit{cswap} :: (\tau, \tau) \to (\tau, \tau)$$

Bitonic Sort

1. Teile die Eingabeliste in zwei Teil-Listen gleicher Länge.

Bitonic Sort

1. Teile die Eingabeliste in zwei Teil-Listen gleicher Länge.
2. Sortiere die beiden Teil-Listen rekursiv

Bitonic Sort

1. Teile die Eingabeliste in zwei Teil-Listen gleicher Länge.
2. Sortiere die beiden Teil-Listen rekursiv, die zweite in umgekehrter Richtung.

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.

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.

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.

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.

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.

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 $O(n \cdot \log(n)^2)$.

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.

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.

Knuths 0-1-Prinzip

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

Formal: ???

Knuths 0-1-Prinzip

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

Formal: Verwende Haskell.

Knuths 0-1-Prinzip

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

Formal: Verwende Haskell. Es seien

$$\texttt{sort} :: ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$$

Knuths 0-1-Prinzip

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

Formal: Verwende Haskell. Es seien

$$\texttt{sort} :: ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$$

$$f :: (\mathsf{Int}, \mathsf{Int}) \to (\mathsf{Int}, \mathsf{Int})$$
$$f\ (x, y) = \mathit{if}\ x > y\ \mathit{then}\ (y, x)\ \mathit{else}\ (x, y)$$

$$g :: (\mathsf{Bool}, \mathsf{Bool}) \to (\mathsf{Bool}, \mathsf{Bool})$$
$$g\ (x, y) = (x\ \&\&\ y, x\ ||\ y)$$

Knuths 0-1-Prinzip

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

Formal: Verwende Haskell. Es seien

$$\texttt{sort} :: ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$$

$$f :: (\mathsf{Int}, \mathsf{Int}) \to (\mathsf{Int}, \mathsf{Int})$$
$$f\ (x, y) = \mathit{if}\ x > y\ \mathit{then}\ (y, x)\ \mathit{else}\ (x, y)$$

$$g :: (\mathsf{Bool}, \mathsf{Bool}) \to (\mathsf{Bool}, \mathsf{Bool})$$
$$g\ (x, y) = (x\ \&\&\ y, x\ ||\ y)$$

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

$$\texttt{sort} :: ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$$

$$f :: (\mathsf{Int}, \mathsf{Int}) \to (\mathsf{Int}, \mathsf{Int})$$
$$f\ (x, y) = \mathit{if}\ x > y\ \mathit{then}\ (y, x)\ \mathit{else}\ (x, y)$$

$$g :: (\mathsf{Bool}, \mathsf{Bool}) \to (\mathsf{Bool}, \mathsf{Bool})$$
$$g\ (x, y) = (x\ \&\&\ y, x\ ||\ y)$$

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

Verwendung des Theorem-Generators

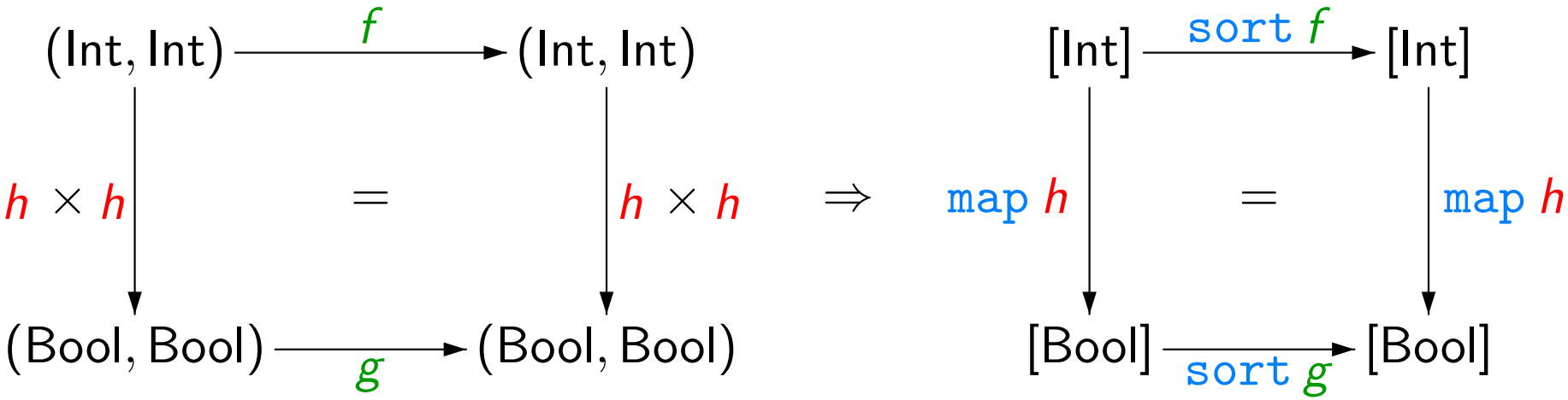
Eingabe:
```
sort::((a,a)->(a,a))->[a]->[a]
```

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)}
```

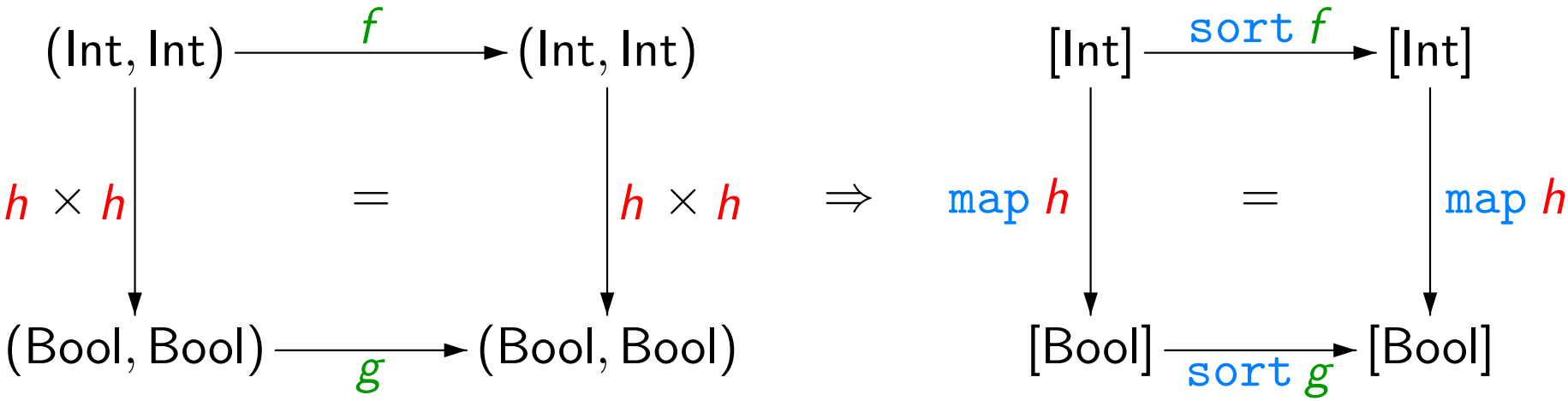
Spezifischer (und intuitiver)

Für alle `sort` :: $((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$,
f :: (Int, Int) $\to$ (Int, Int), g :: (Bool, Bool) $\to$ (Bool, Bool) und
h :: Int $\to$ Bool:



Spezifischer (und intuitiver)

Für alle `sort` :: $((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$,
f :: (Int, Int) $\to$ (Int, Int), g :: (Bool, Bool) $\to$ (Bool, Bool) und
h :: Int $\to$ Bool:



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

Knuths 0-1-Prinzip

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

Formal: Verwende Haskell. Es seien

$$\texttt{sort} :: ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$$

$$f :: (\mathsf{Int}, \mathsf{Int}) \to (\mathsf{Int}, \mathsf{Int})$$
$$f\ (x, y) = \mathit{if}\ x > y\ \mathit{then}\ (y, x)\ \mathit{else}\ (x, y)$$

$$g :: (\mathsf{Bool}, \mathsf{Bool}) \to (\mathsf{Bool}, \mathsf{Bool})$$
$$g\ (x, y) = (x\ \&\&\ y, x\ ||\ y)$$

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

Beweis von „P auf [Bool] impliziert P auf [Int]“

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

Beweis von „P auf [Bool] impliziert P auf [Int]“

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

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

Beweis von „P auf [Bool] impliziert P auf [Int]“

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

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

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

Beweis von „P auf [Bool] impliziert P auf [Int]“

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

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

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

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

Beweis von „P auf [Bool] impliziert P auf [Int]“

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

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

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

Angenommen, es gäbe $us :: [\text{Int}]$, $vs = \texttt{sort}\ f\ us$ mit $\neg P(us, vs)$.
Dann gäbe es ein kleinstes n so dass die Anzahlen von n in us und vs nicht gleich sind.

Beweis von „P auf [Bool] impliziert P auf [Int]“

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

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

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

Angenommen, es gäbe $us :: [\text{Int}]$, $vs = \texttt{sort}\ f\ us$ 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 ($\texttt{map}\ h\ us$) und ($\texttt{map}\ h\ vs$) verschieden.

Beweis von „P auf [Bool] impliziert P auf [Int]“

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

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

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

Angenommen, es gäbe $us :: [\text{Int}]$, $vs = \texttt{sort}\ f\ us$ 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 $(\texttt{map}\ h\ us)$ und $(\texttt{map}\ h\ vs)$ verschieden.
Dies steht jedoch im Widerspruch zur Vorbedingung mit:

$xs = \texttt{map}\ h\ us$
$ys = \texttt{sort}\ g\ (\texttt{map}\ h\ us)$

Beweis von „P auf [Bool] impliziert P auf [Int]“

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

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

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

Angenommen, es gäbe $us :: [\text{Int}]$, $vs = \texttt{sort}\ f\ us$ 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 $(\texttt{map}\ h\ us)$ und $(\texttt{map}\ h\ vs)$ verschieden.
Dies steht jedoch im Widerspruch zur Vorbedingung mit:

$xs = \texttt{map}\ h\ us$
$ys = \texttt{sort}\ g\ (\texttt{map}\ h\ us) = \texttt{map}\ h\ (\texttt{sort}\ f\ us)$

Beweis von „P auf [Bool] impliziert P auf [Int]“

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

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

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

Angenommen, es gäbe $us :: [\text{Int}]$, $vs = \texttt{sort}\ f\ us$ 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 $(\texttt{map}\ h\ us)$ und $(\texttt{map}\ h\ vs)$ verschieden.
Dies steht jedoch im Widerspruch zur Vorbedingung mit:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Angenommen, es gäbe $us :: [\text{Int}]$, $vs = \texttt{sort}\ f\ us$ mit $\neg Q(vs)$.
Dann gäbe es $n < m$ so dass ein m in vs vor einem n vorkommt.

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

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

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

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

Angenommen, es gäbe $us :: [\text{Int}]$, $vs = \texttt{sort}\ f\ us$ 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 $(\texttt{map}\ h\ vs)$ vor einem False vorkommen.

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

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

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

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

Angenommen, es gäbe $us :: [\text{Int}]$, $vs = \texttt{sort}\ f\ us$ 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 $(\texttt{map}\ h\ vs)$ vor einem False vorkommen. Dies steht jedoch im Widerspruch zur Vorbedingung mit:

$xs = \texttt{map}\ h\ us$
$ys = \texttt{sort}\ g\ (\texttt{map}\ h\ us)$

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

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

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

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

Angenommen, es gäbe $us :: [\text{Int}]$, $vs = \texttt{sort}\ f\ us$ 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 $(\texttt{map}\ h\ vs)$ vor einem False vorkommen. Dies steht jedoch im Widerspruch zur Vorbedingung mit:

$$xs = \texttt{map}\ h\ us$$
$$ys = \texttt{sort}\ g\ (\texttt{map}\ h\ us) = \texttt{map}\ h\ (\texttt{sort}\ f\ us)$$

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

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

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

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

Angenommen, es gäbe $us :: [\text{Int}]$, $vs = \texttt{sort}\ f\ us$ 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 $(\texttt{map}\ h\ vs)$ vor einem False vorkommen. Dies steht jedoch im Widerspruch zur Vorbedingung mit:

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

Knuths 0-1-Prinzip

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

Formal: Verwende Haskell. Es seien

$$\texttt{sort} :: ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$$

$$f :: (\mathsf{Int}, \mathsf{Int}) \to (\mathsf{Int}, \mathsf{Int})$$
$$f\ (x, y) = \mathit{if}\ x > y\ \mathit{then}\ (y, x)\ \mathit{else}\ (x, y)$$

$$g :: (\mathsf{Bool}, \mathsf{Bool}) \to (\mathsf{Bool}, \mathsf{Bool})$$
$$g\ (x, y) = (x\ \&\&\ y, x\ ||\ y)$$

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.

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 besonderes eleganten Beweis dieses Prinzips [Day et al. 1999].

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 besonderes eleganten Beweis dieses Prinzips [Day et al. 1999].

- Geht Ähnliches für andere Klassen von Algorithmen?

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 besonderes 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* $:: (\alpha, \alpha) \to (\alpha, \alpha)$ im Fall des Sortierens.

Parallele Präfixberechnung

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

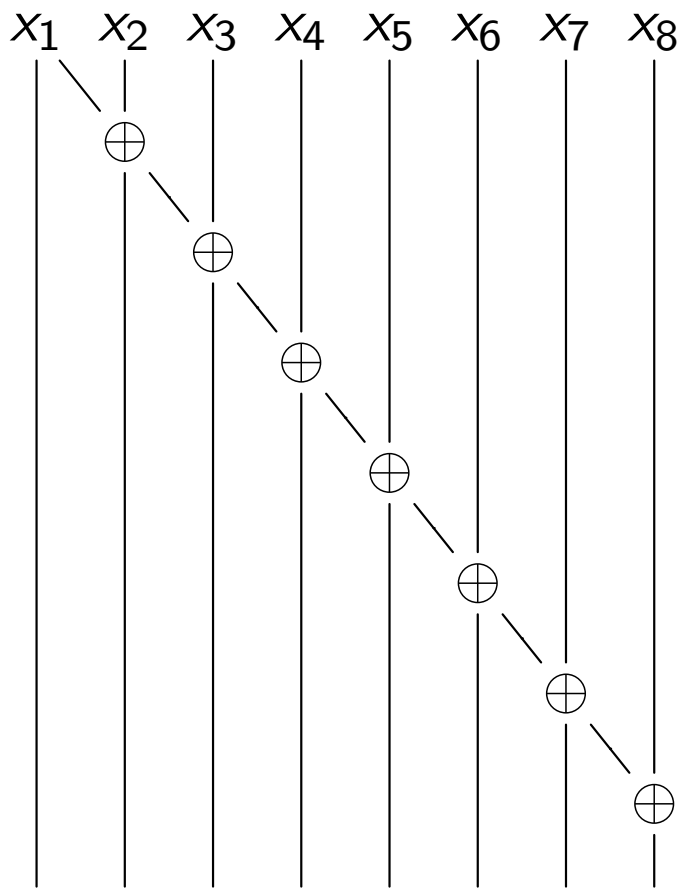
Aufgabe: Berechne die Werte $x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n$

Parallele Präfixberechnung

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

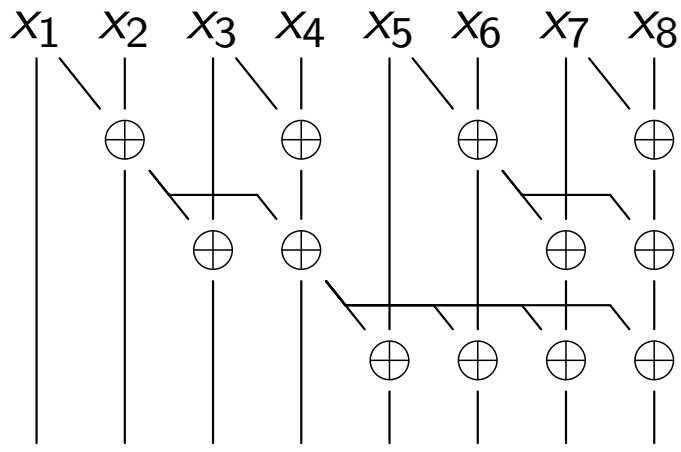
Aufgabe: Berechne die Werte $x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n$

Lösung:



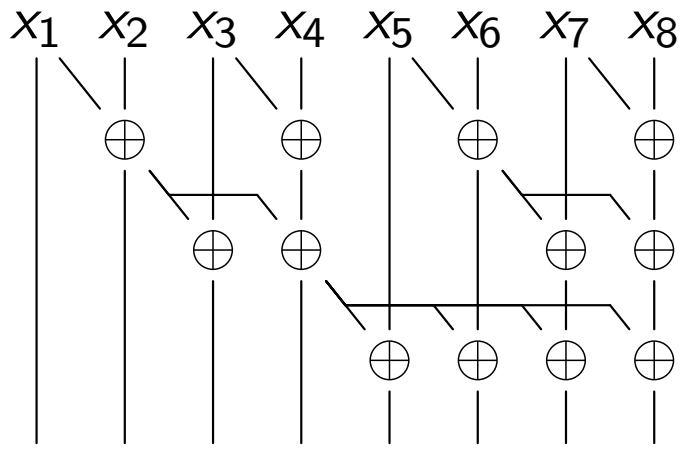
Parallele Präfixberechnung

Alternativ:

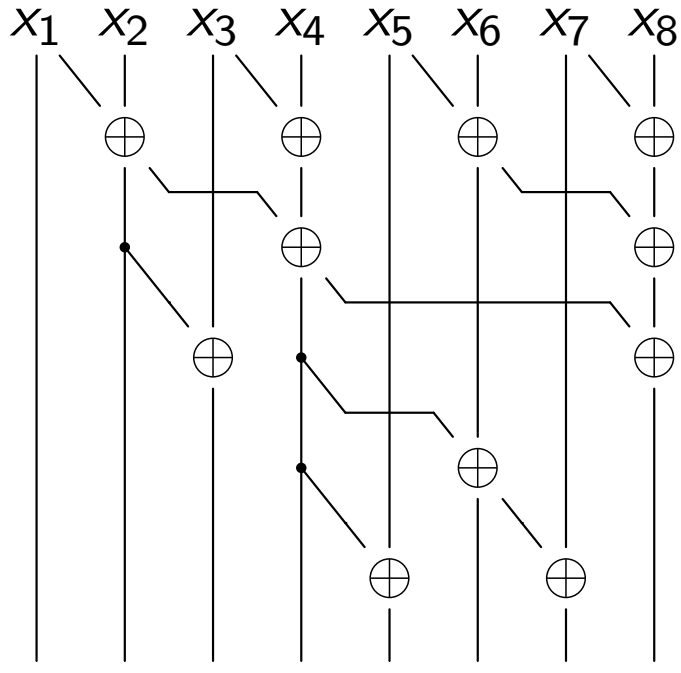


Parallele Präfixberechnung

Alternativ:



Oder:



Oder: ...

In Haskell

Funktionen des Typs:

$$\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

In Haskell

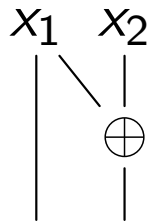
Funktionen des Typs:

$$\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

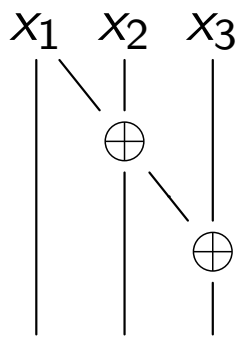
Zum Beispiel, à la [Sklansky 1960]:

```
sklansky :: (α → α → α) → [α] → [α]
sklansky (⊕) [x] = [x]
sklansky (⊕) xs  = us ++ vs
    where t       = ((length xs) + 1) `div` 2
          (ys, zs) = splitAt t xs
          us      = sklansky (⊕) ys
          vs      = [(last us) ⊕ v | v ← sklansky (⊕) zs]
```

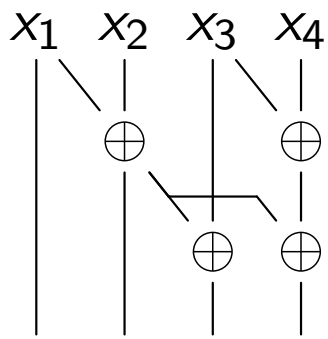
Sklanskys Methode:



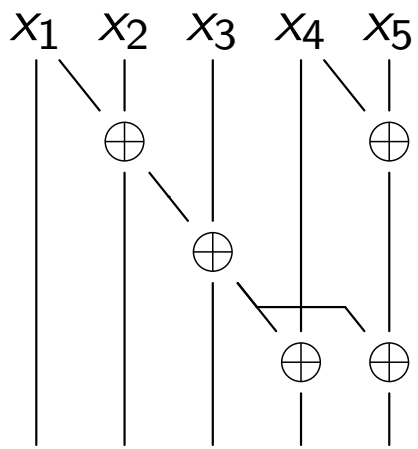
Sklanskys Methode:



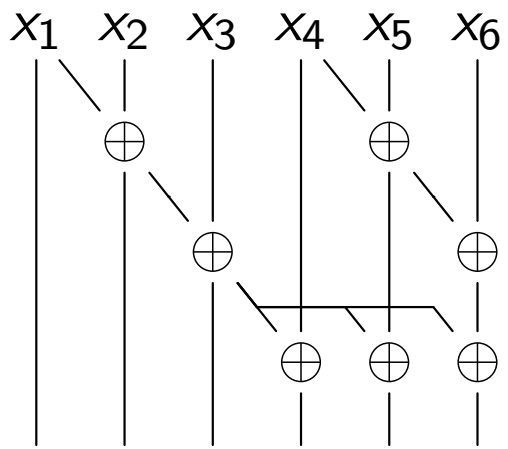
Sklanskys Methode:



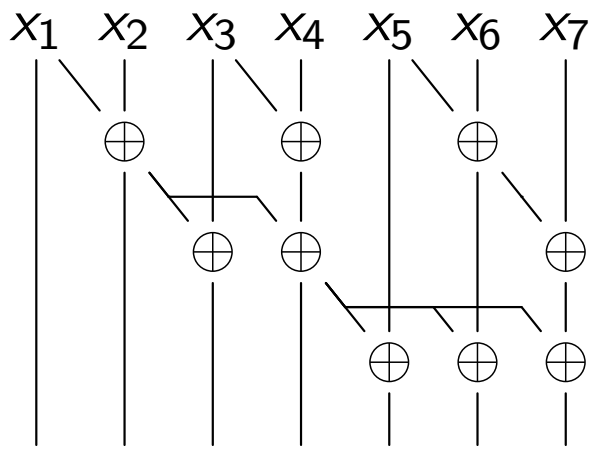
Sklanskys Methode:



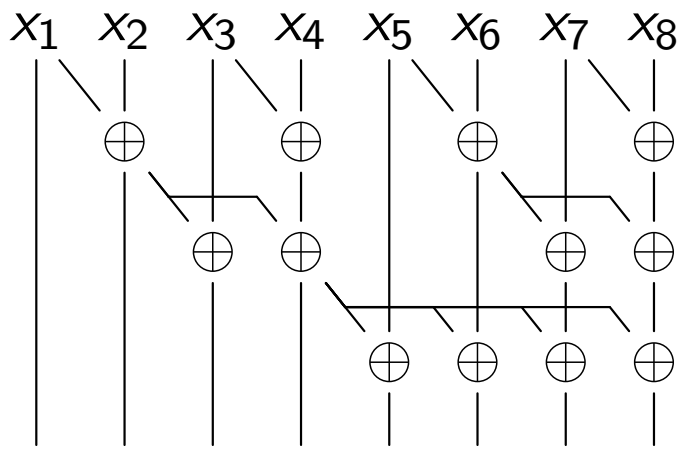
Sklanskys Methode:



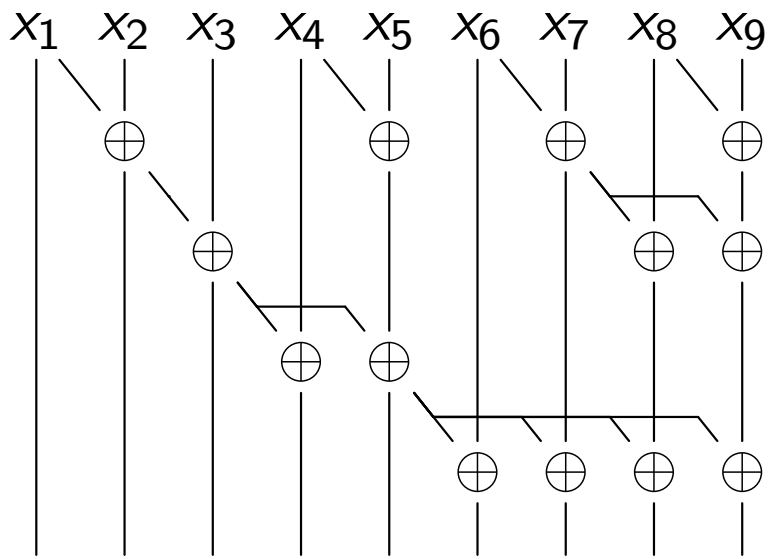
Sklanskys Methode:



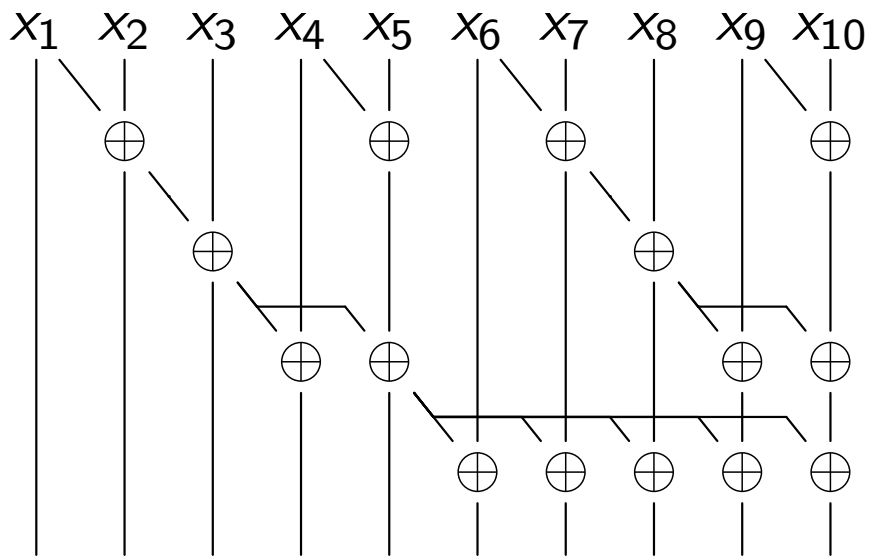
Sklanskys Methode:



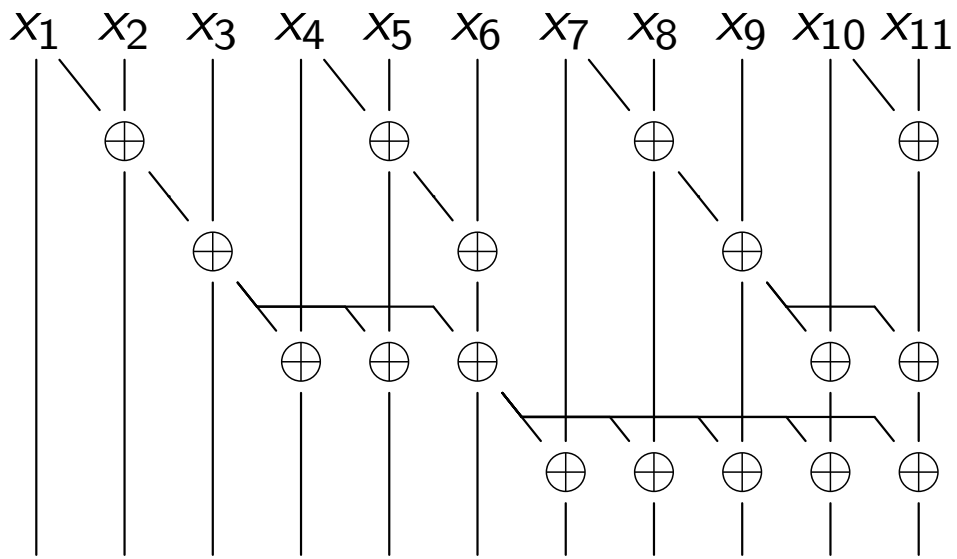
Sklanskys Methode:



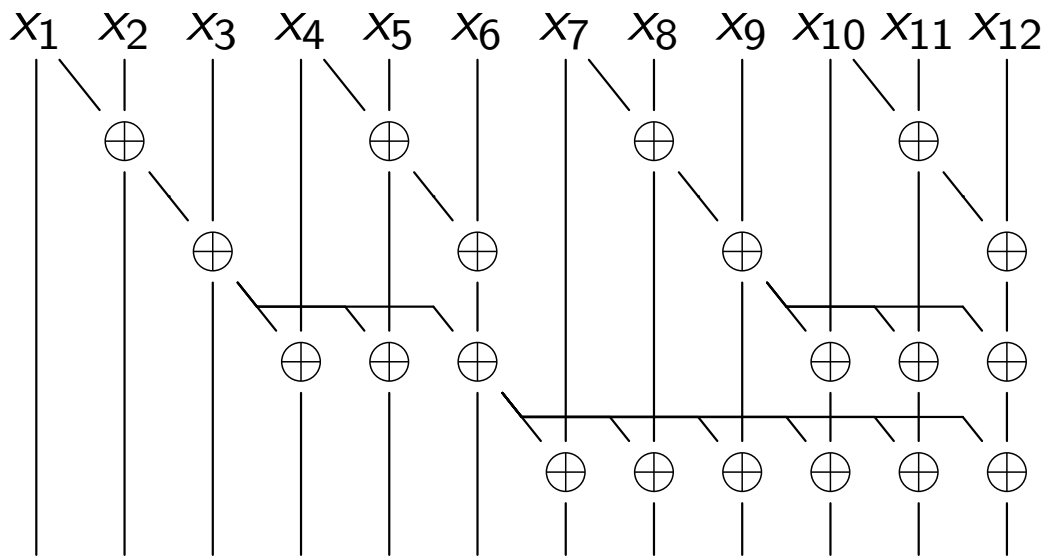
Sklanskys Methode:



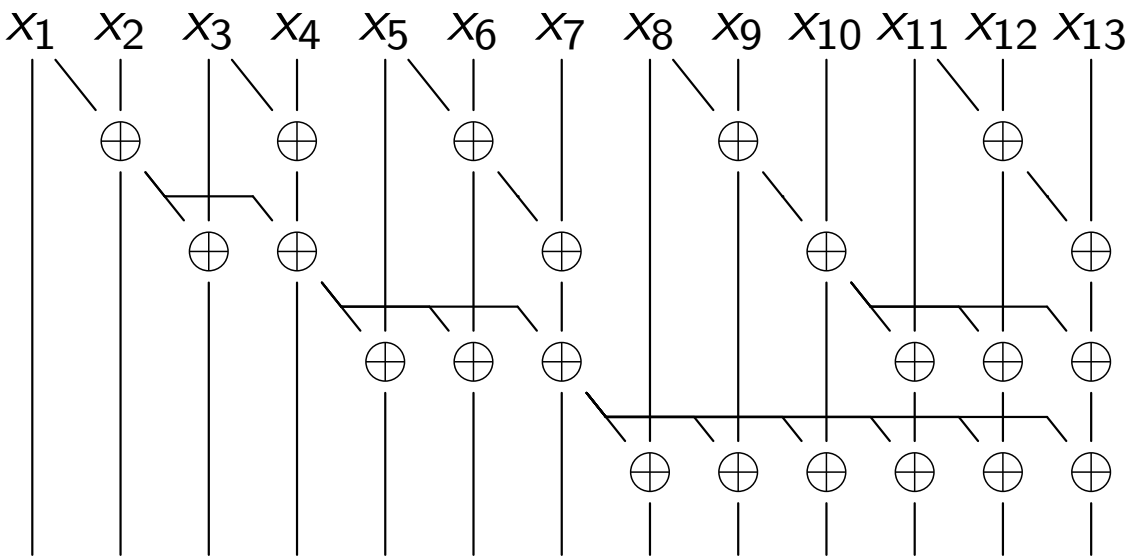
Sklanskys Methode:



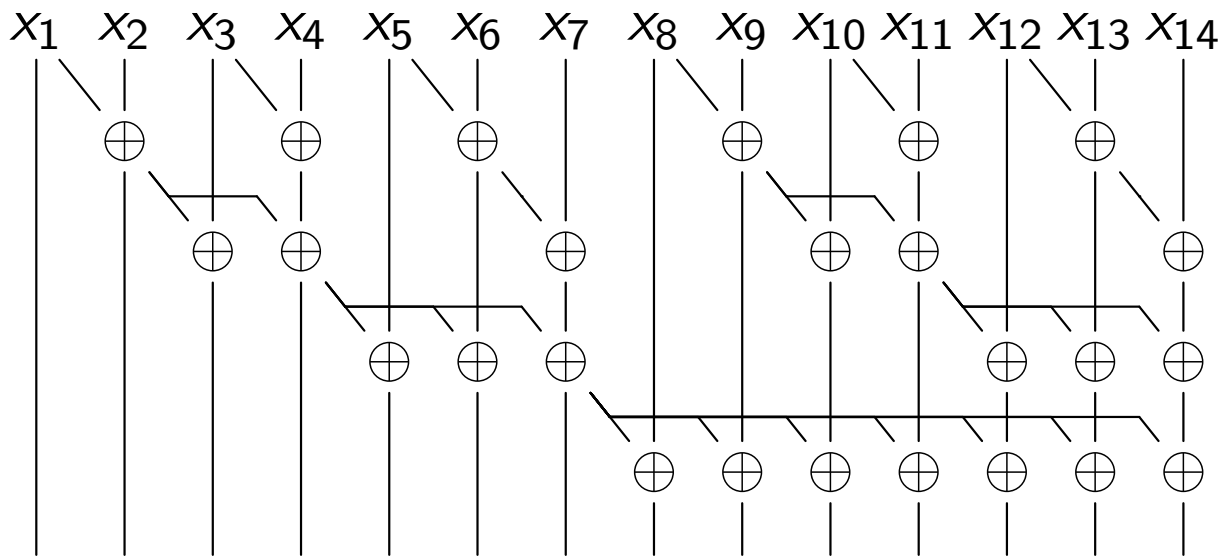
Sklanskys Methode:



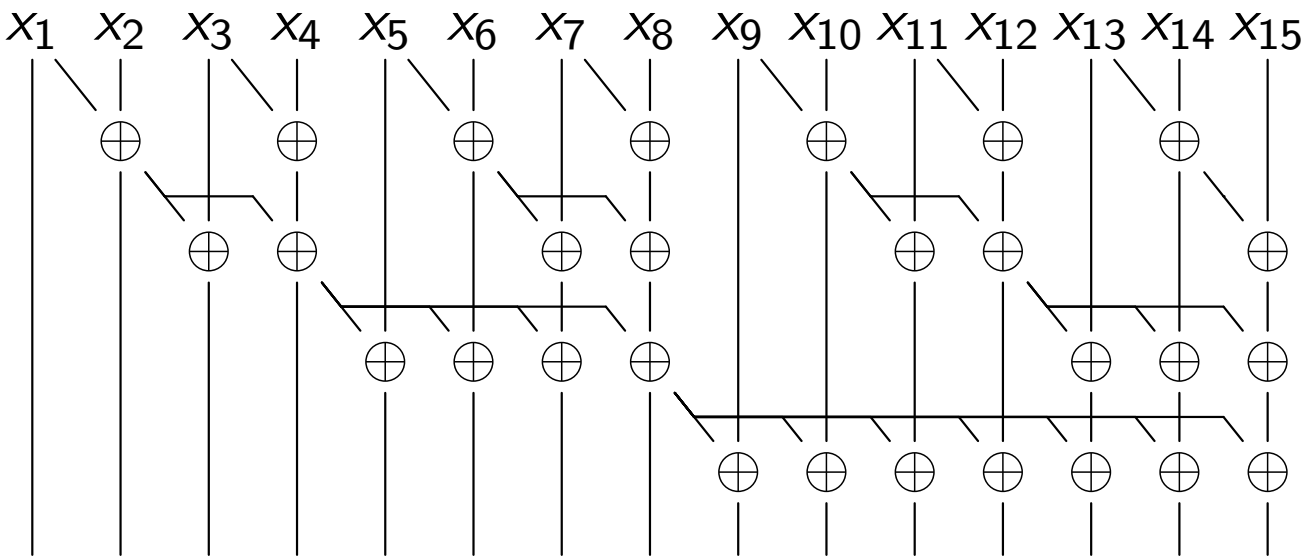
Sklanskys Methode:



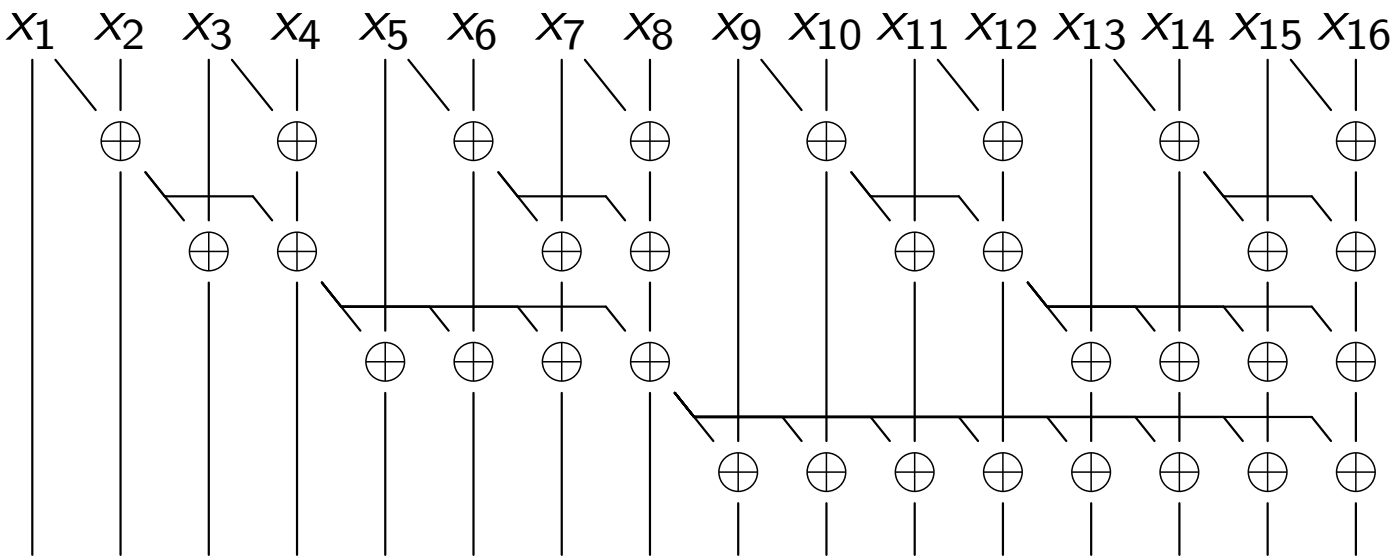
Sklanskys Methode:



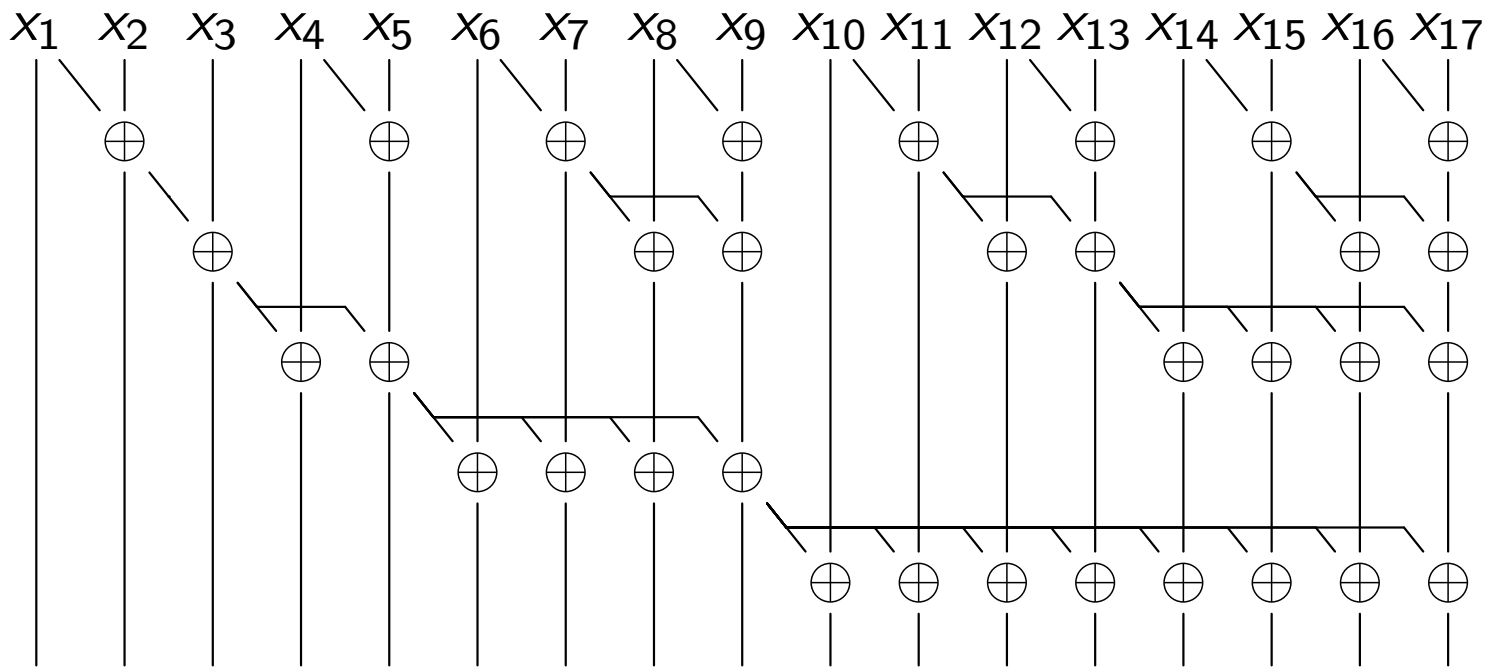
Sklanskys Methode:



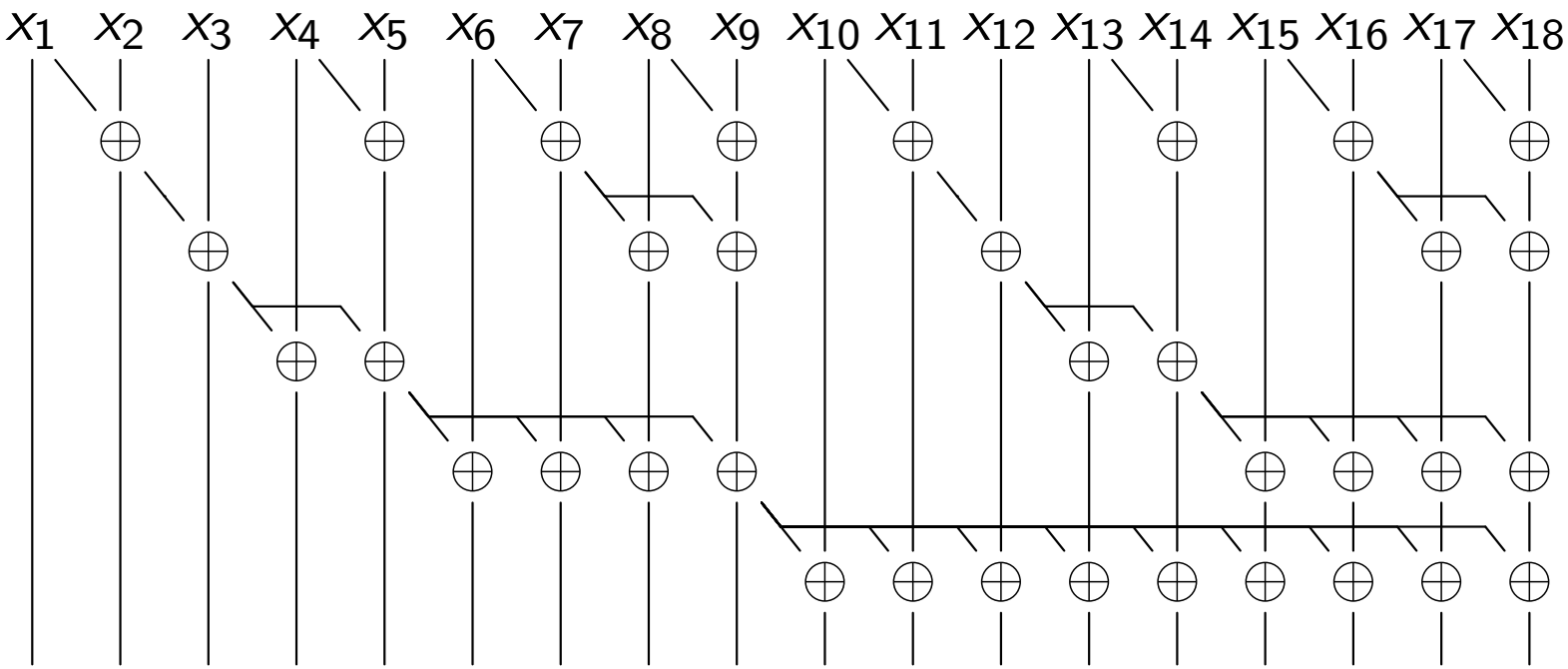
Sklanskys Methode:



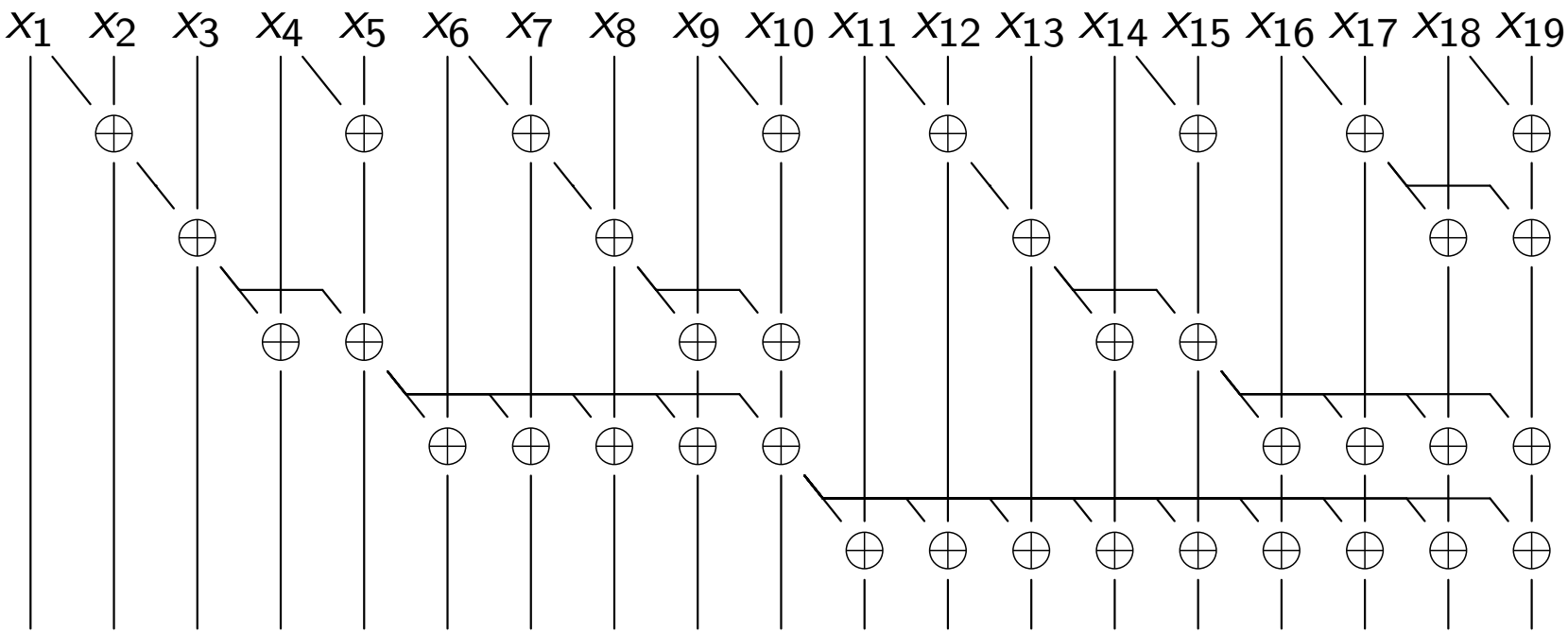
Sklanskys Methode:



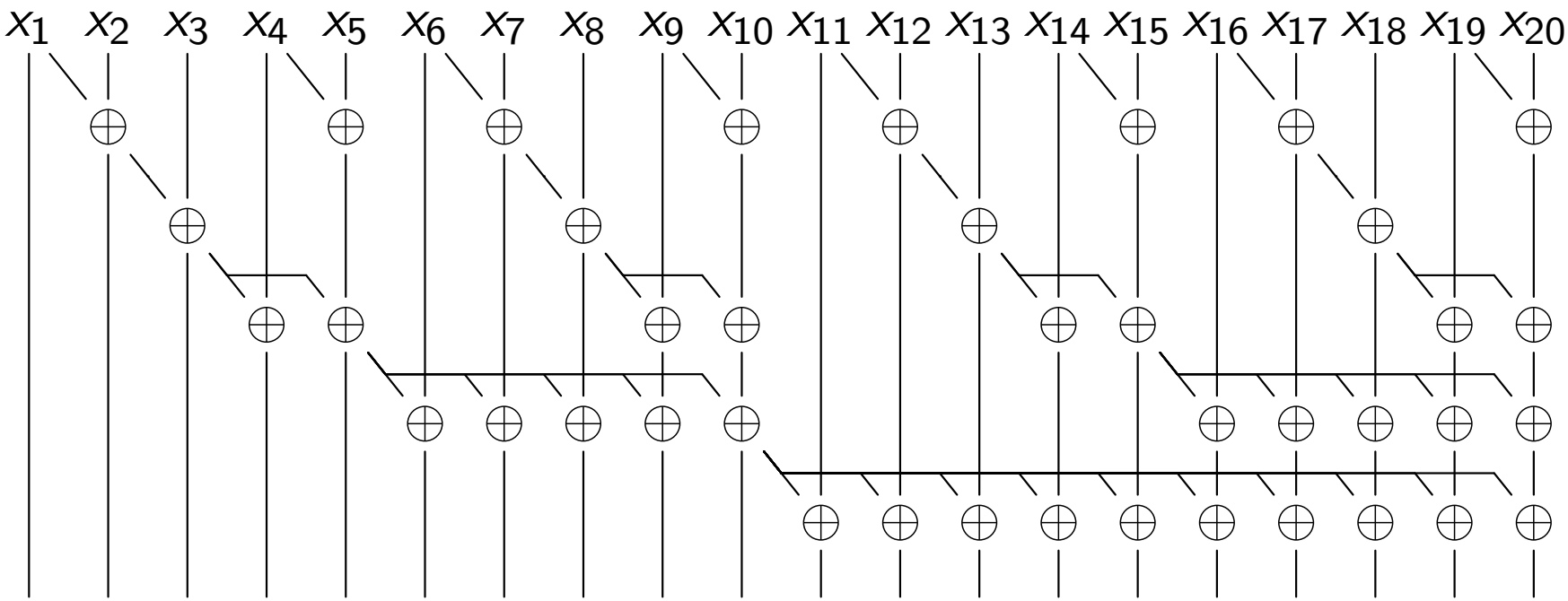
Sklanskys Methode:



Sklanskys Methode:



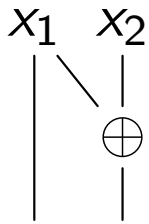
Sklanskys Methode:



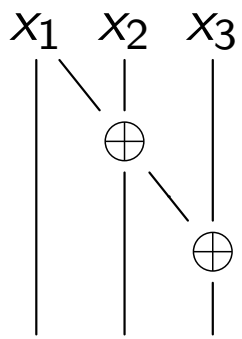
Oder à la [Brent & Kung 1980]

$$
\begin{aligned}
&\texttt{brentKung} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\
&\texttt{brentKung}\ (\oplus)\ [x] = [x] \\
&\texttt{brentKung}\ (\oplus)\ xs\ = odds\ (riffle\ (par\ (unriffle\ (evens\ xs)))) \\
&\quad \textbf{where}\ evens\ [\,] = [\,] \\
&\qquad\qquad evens\ [x] = [x] \\
&\qquad\qquad evens\ (x : y : zs) = [x, x \oplus y] +\!\!+ evens\ zs \\
&\qquad\qquad unriffle\ [\,] = ([\,], [\,]) \\
&\qquad\qquad unriffle\ [x] = ([x], [\,]) \\
&\qquad\qquad unriffle\ (x : y : zs) = (x : xs, y : ys) \\
&\qquad\qquad\quad \textbf{where}\ (xs, ys) = unriffle\ zs \\
&\qquad\qquad par\ (xs, ys) = (xs, \texttt{brentKung}\ (\oplus)\ ys) \\
&\qquad\qquad riffle\ ([\,], [\,]) = [\,] \\
&\qquad\qquad riffle\ ([x], [\,]) = [x] \\
&\qquad\qquad riffle\ (x : xs, y : ys) = x : y : riffle\ (xs, ys) \\
&\qquad\qquad odds\ (x : xs) = x : evens\ xs
\end{aligned}
$$

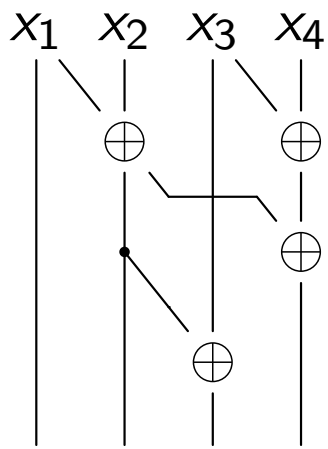
Brent & Kungs Methode:



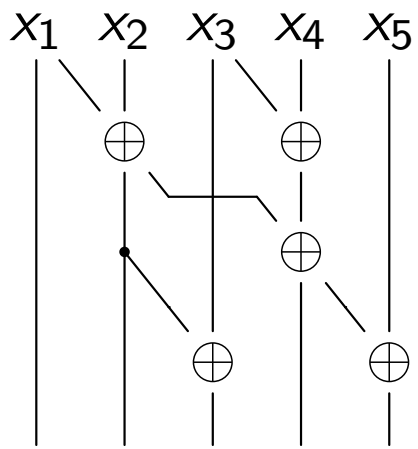
Brent & Kungs Methode:



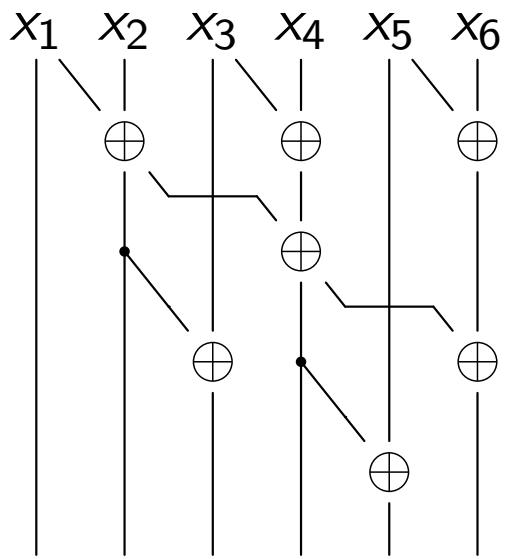
Brent & Kungs Methode:



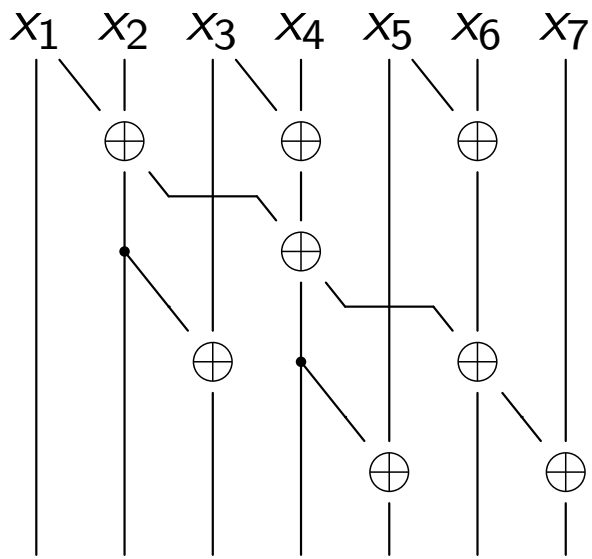
Brent & Kungs Methode:



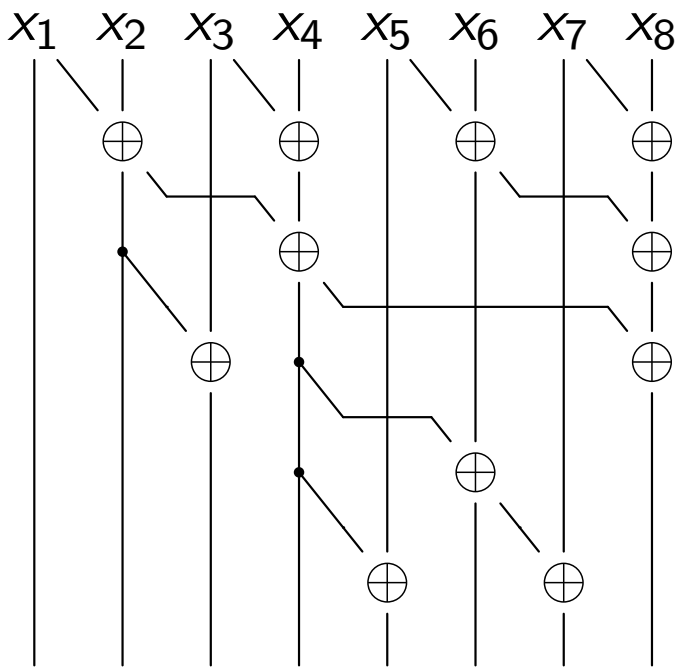
Brent & Kungs Methode:



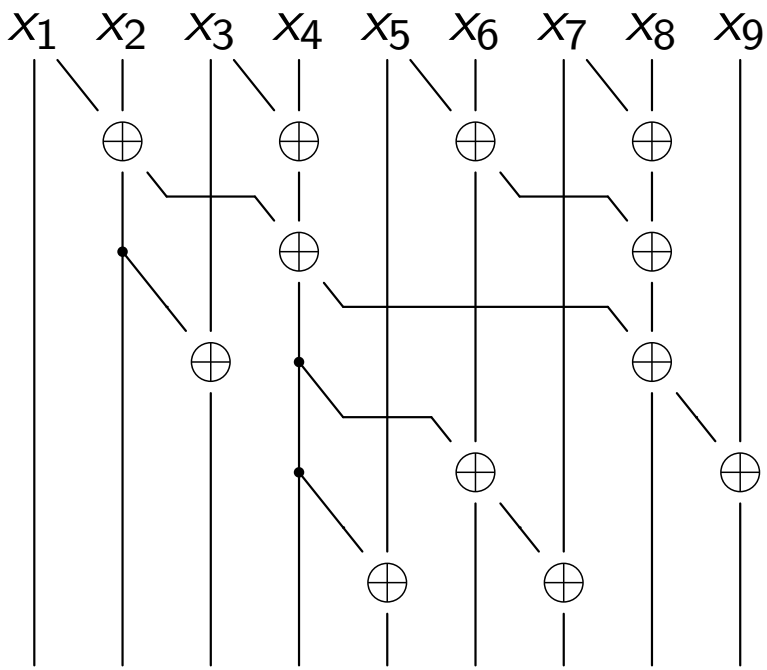
Brent & Kungs Methode:



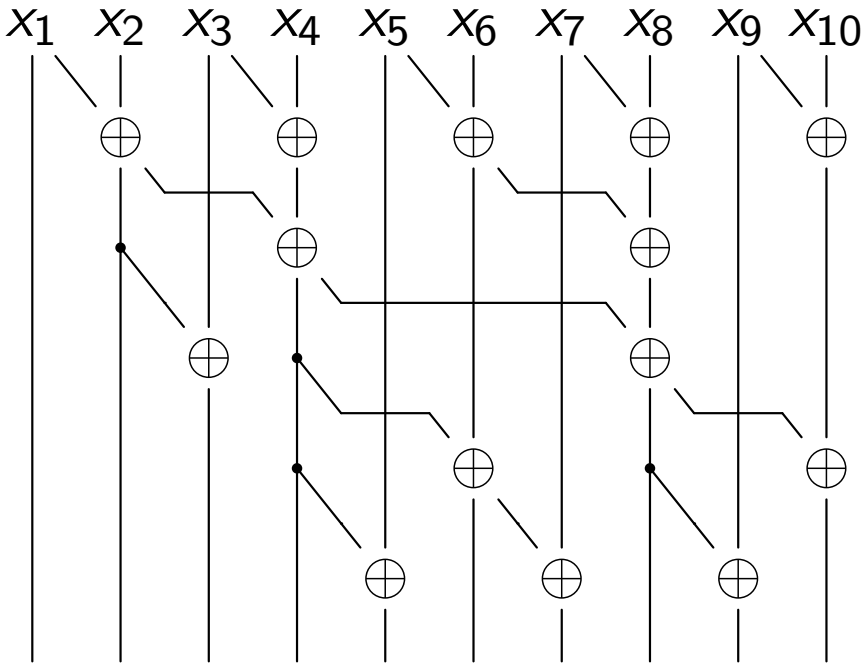
Brent & Kungs Methode:



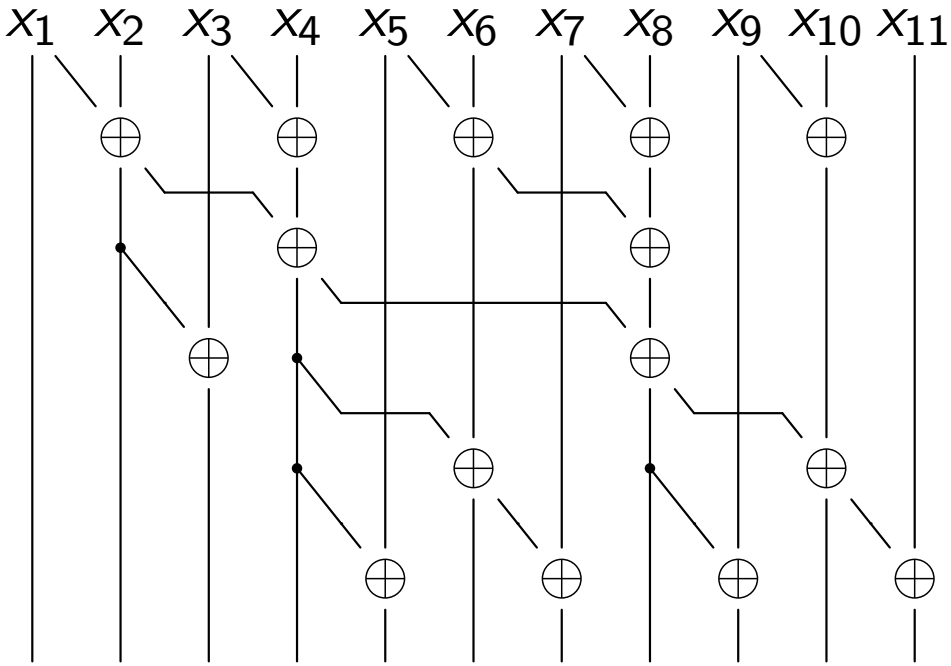
Brent & Kungs Methode:



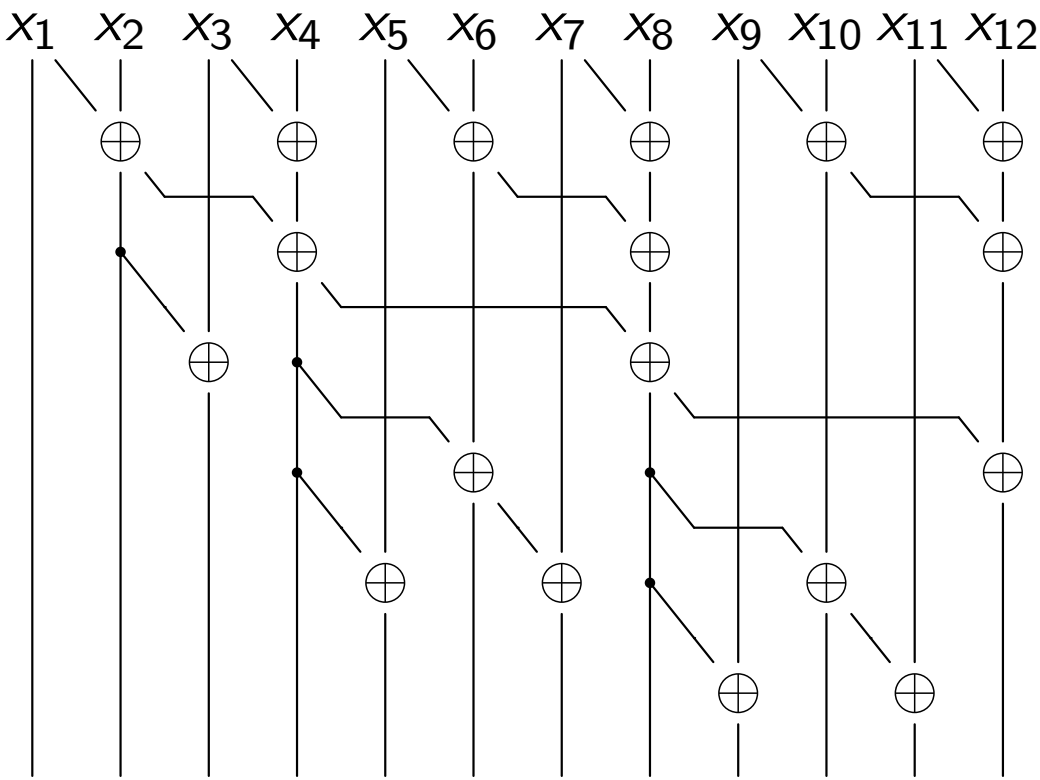
Brent & Kungs Methode:



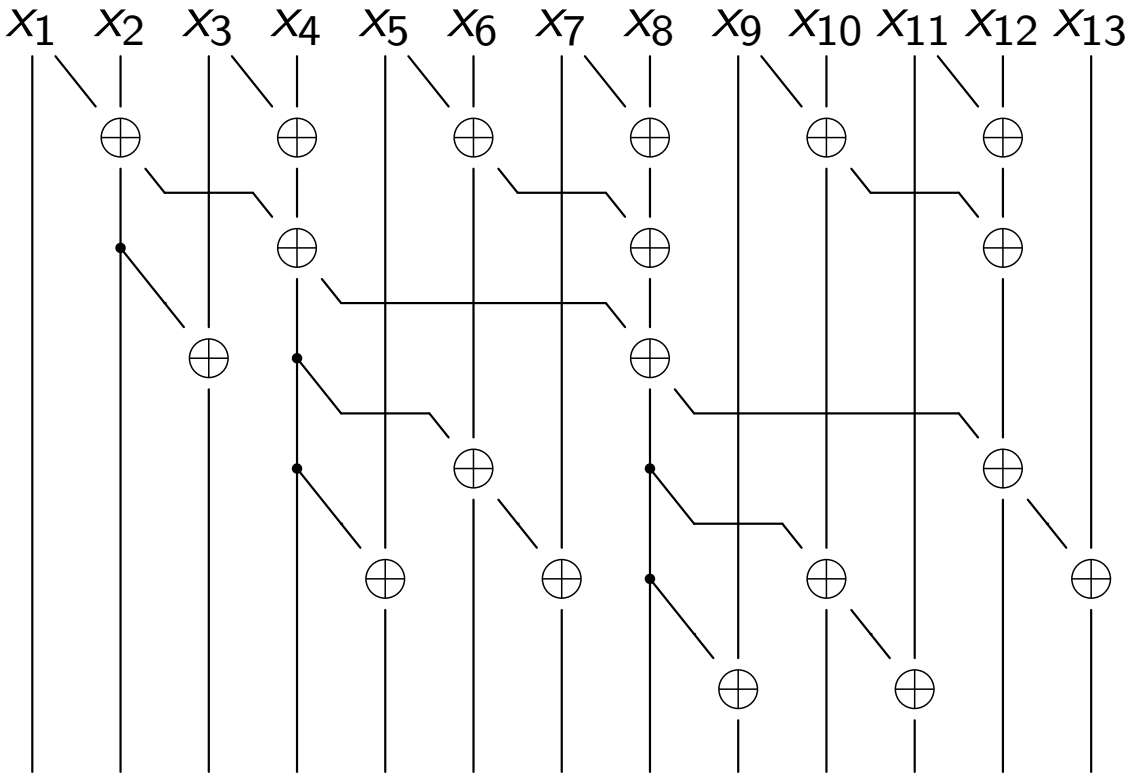
Brent & Kungs Methode:



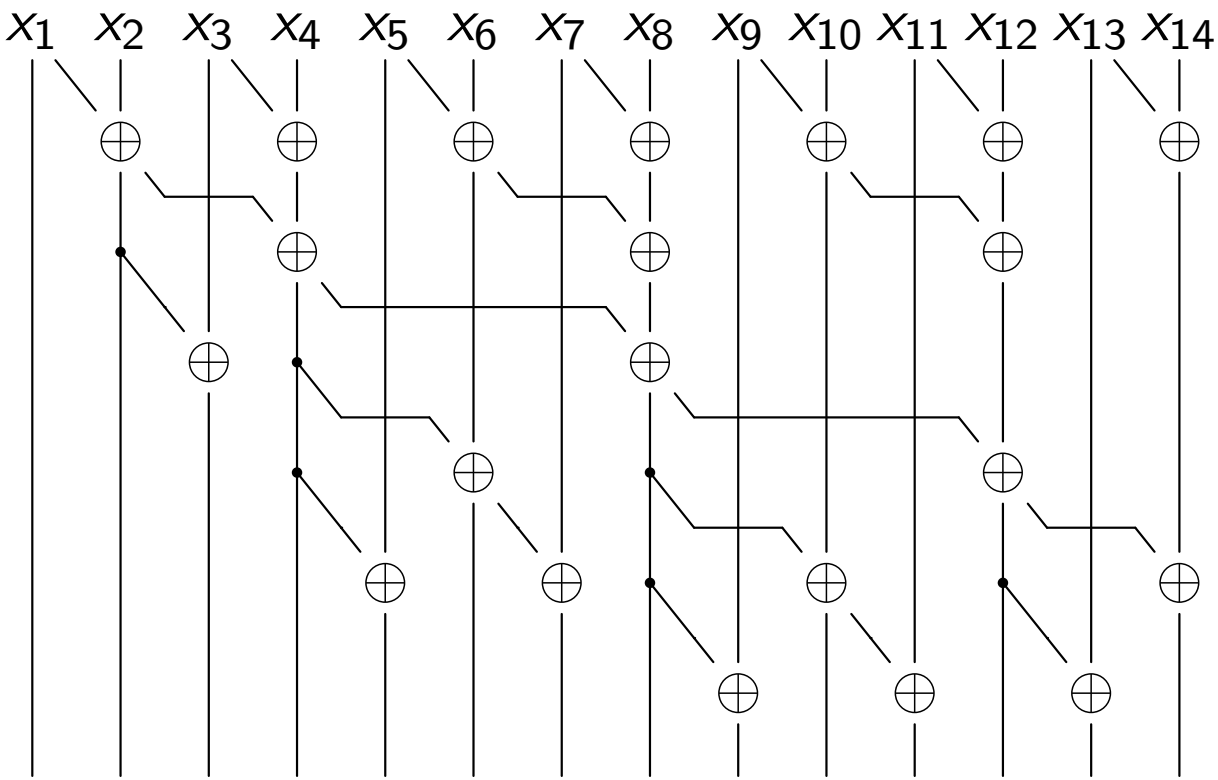
Brent & Kungs Methode:



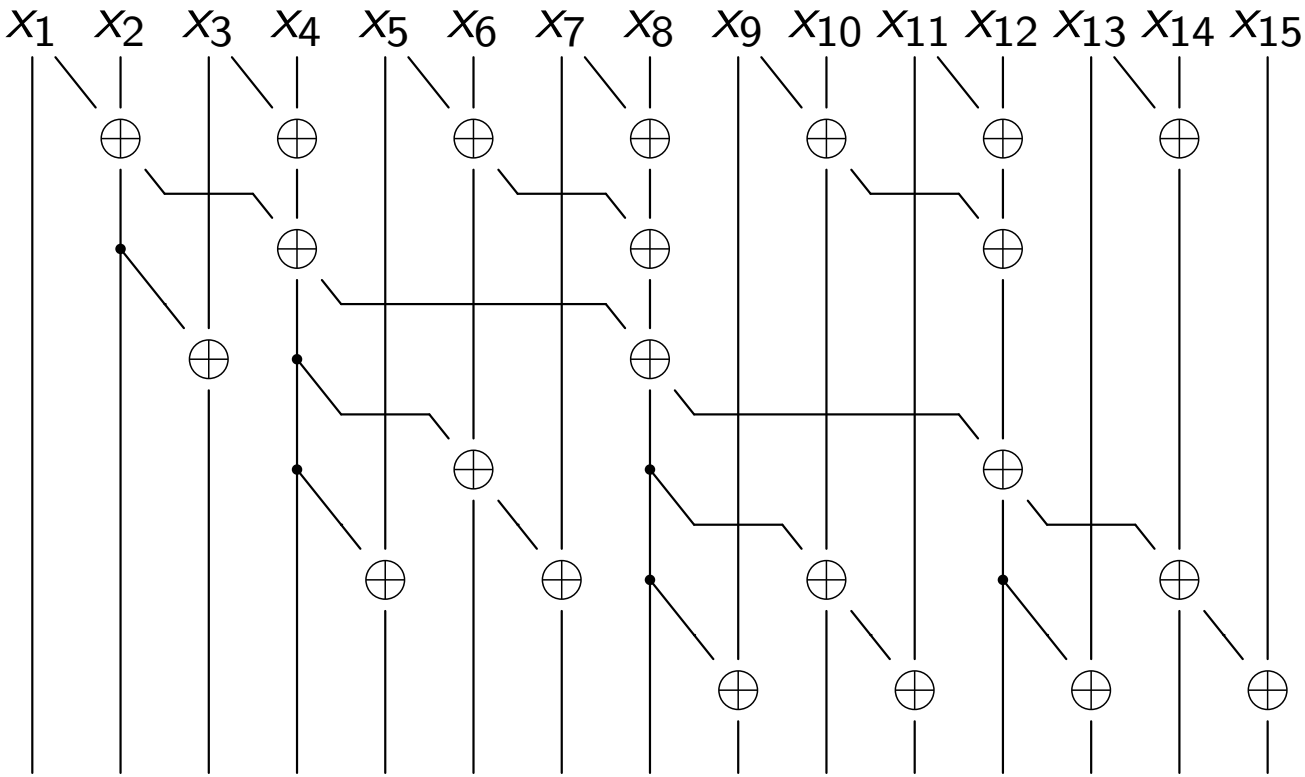
Brent & Kungs Methode:



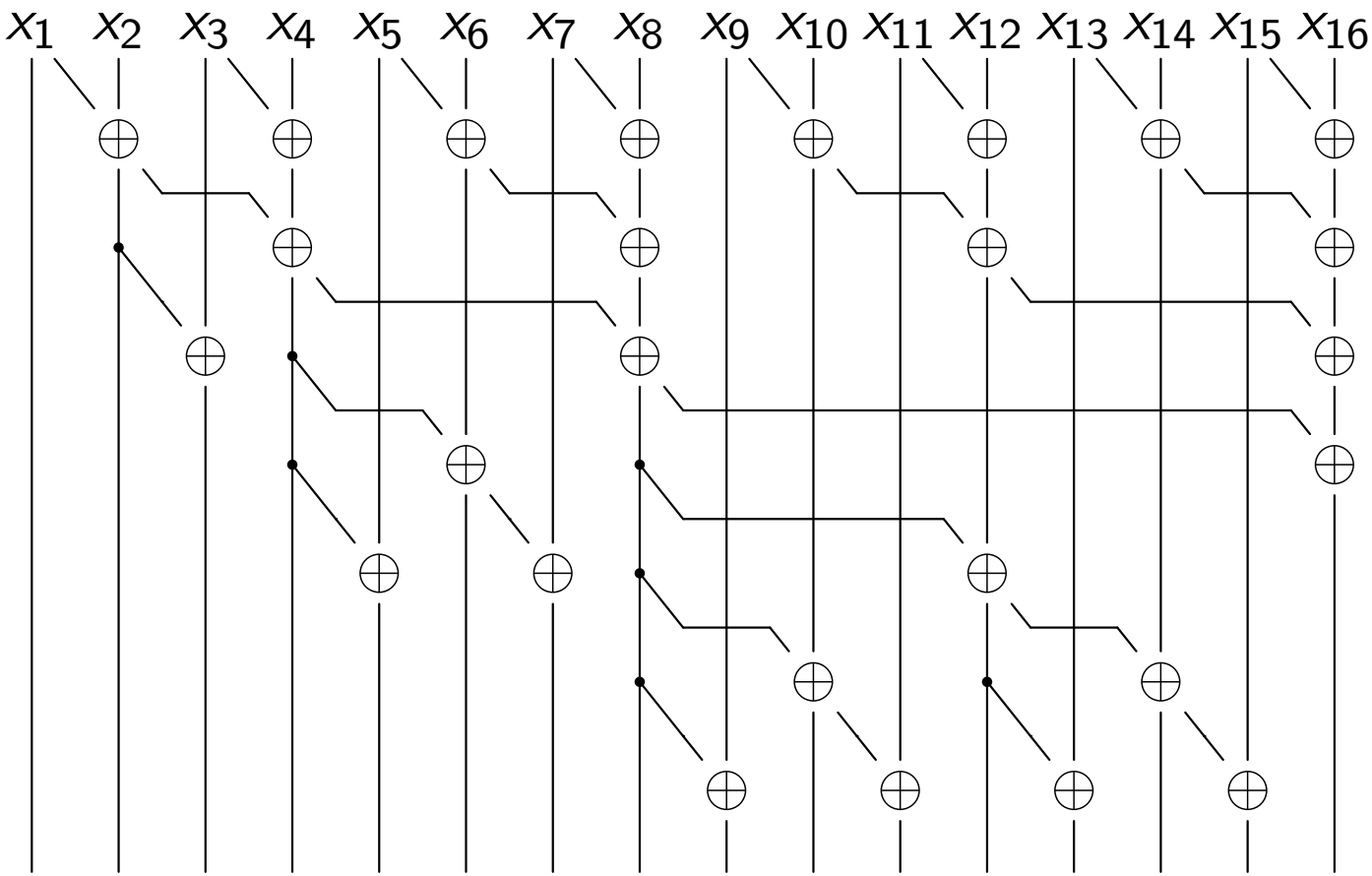
Brent & Kungs Methode:



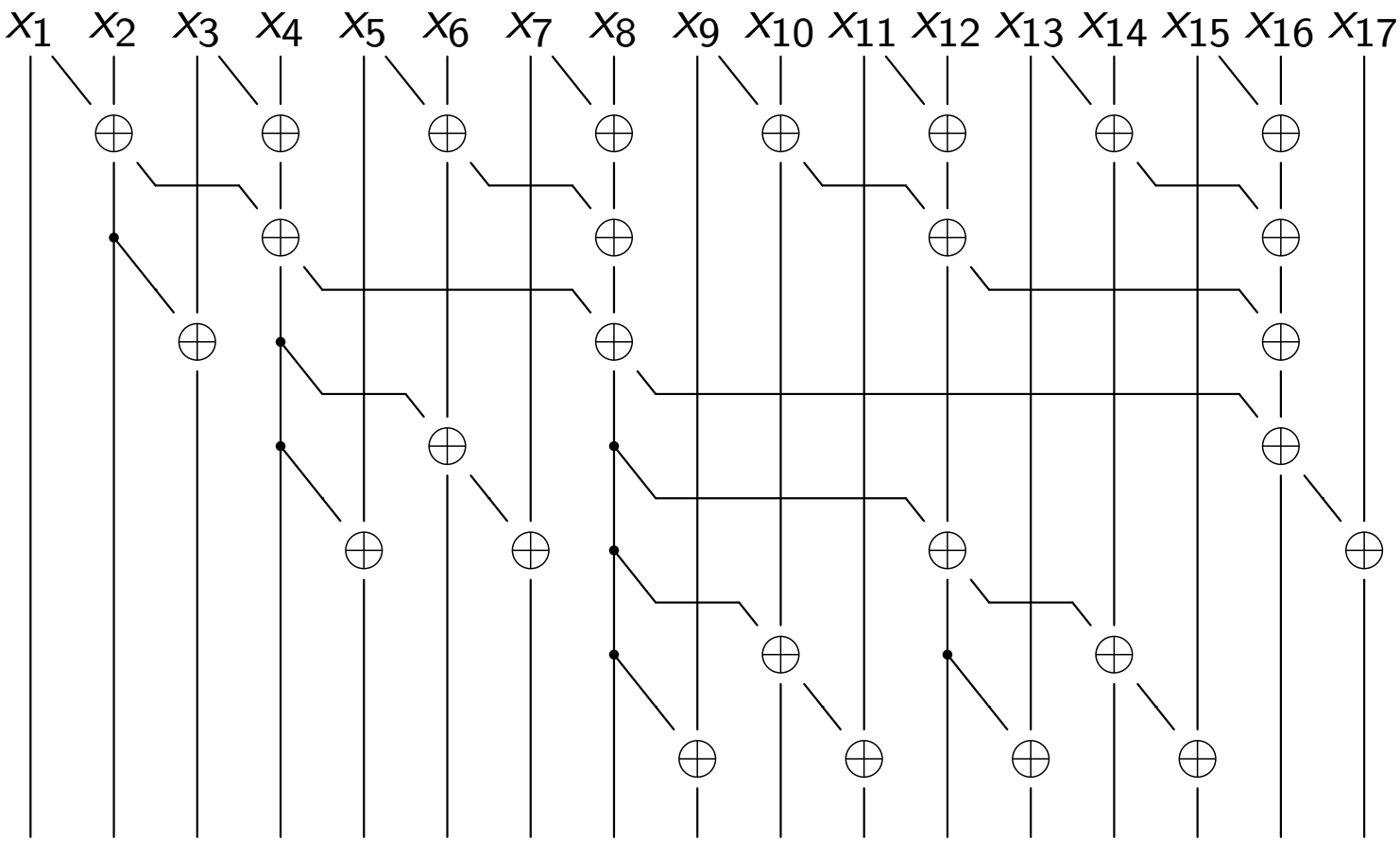
Brent & Kungs Methode:



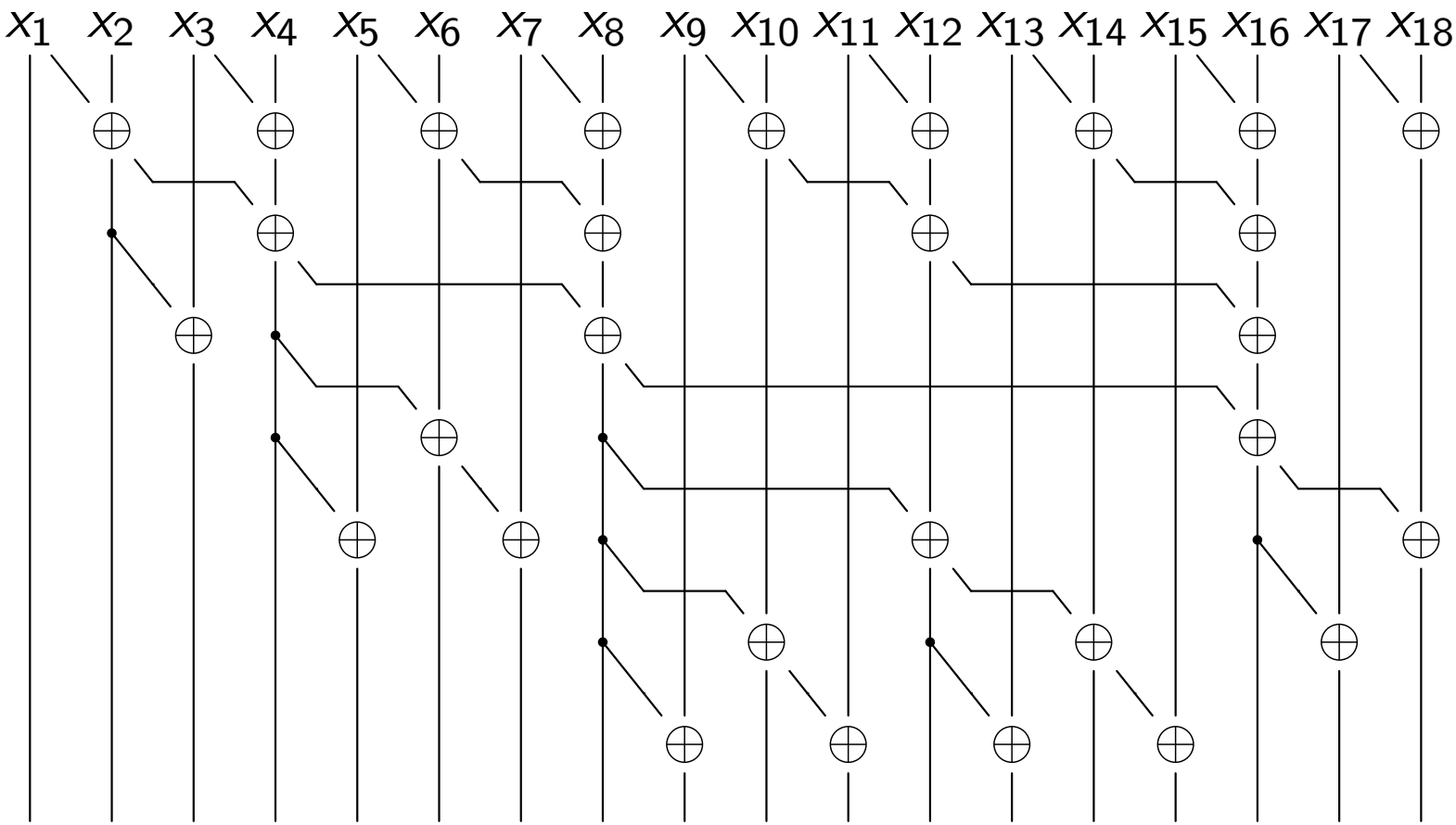
Brent & Kungs Methode:



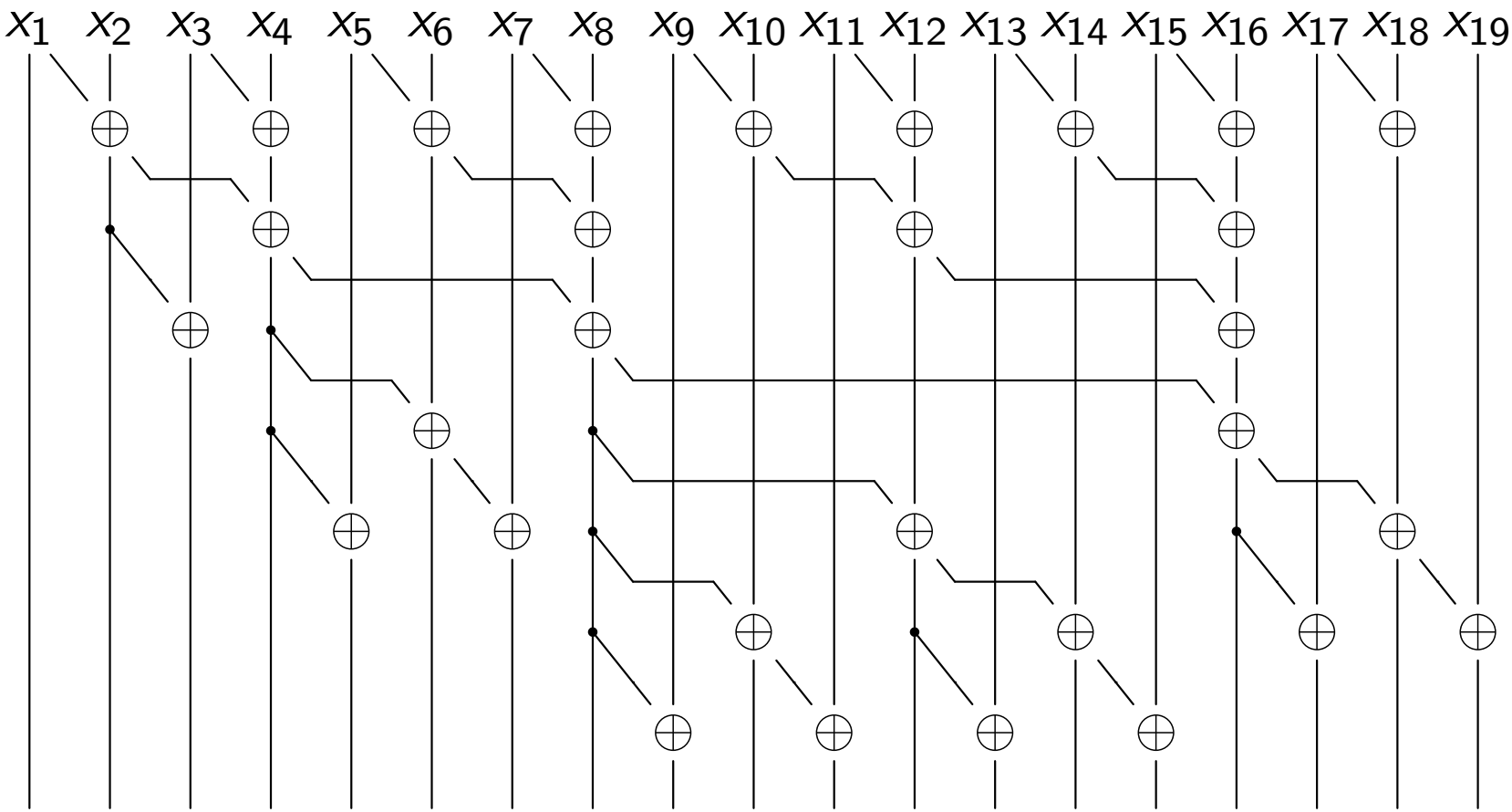
Brent & Kungs Methode:



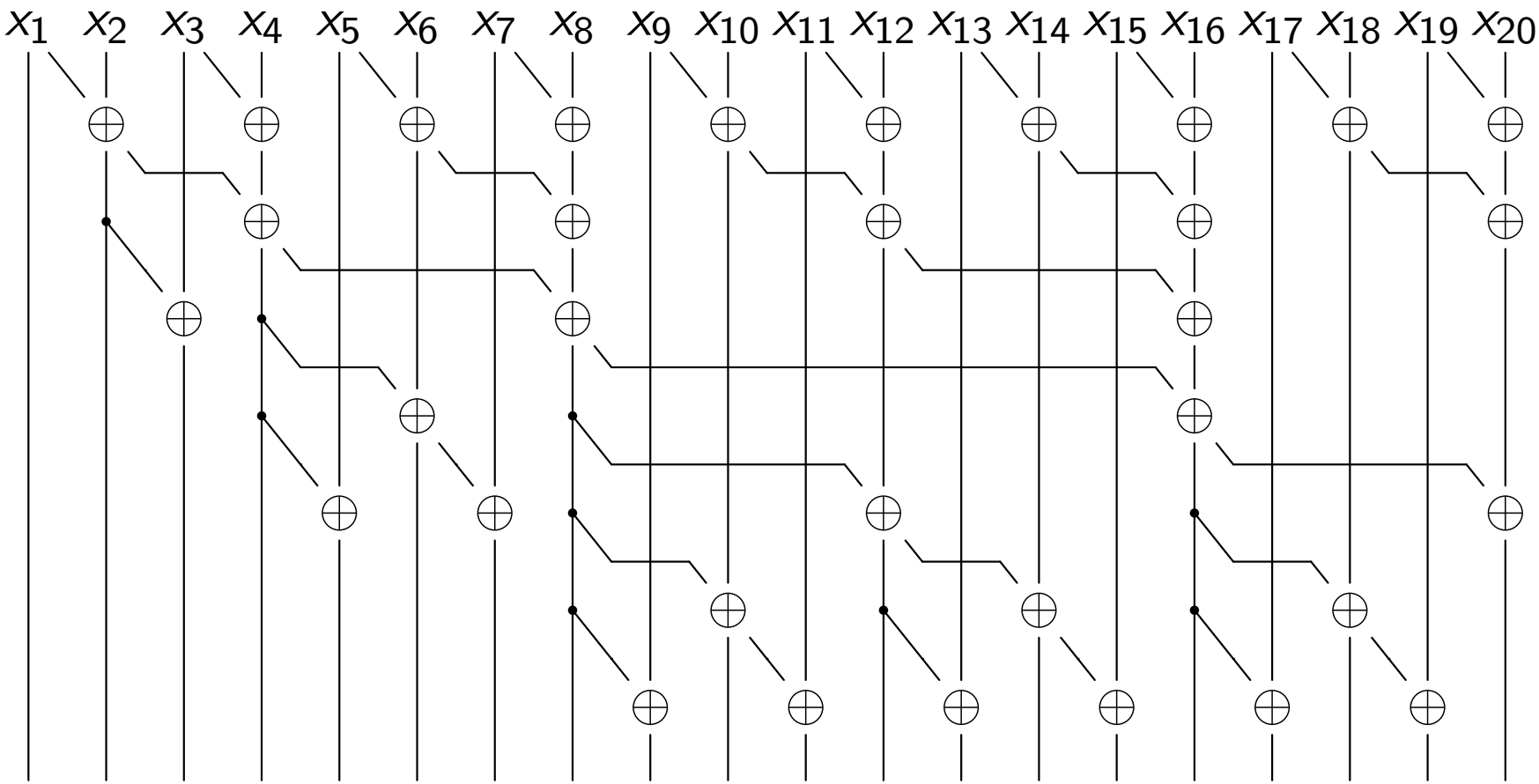
Brent & Kungs Methode:



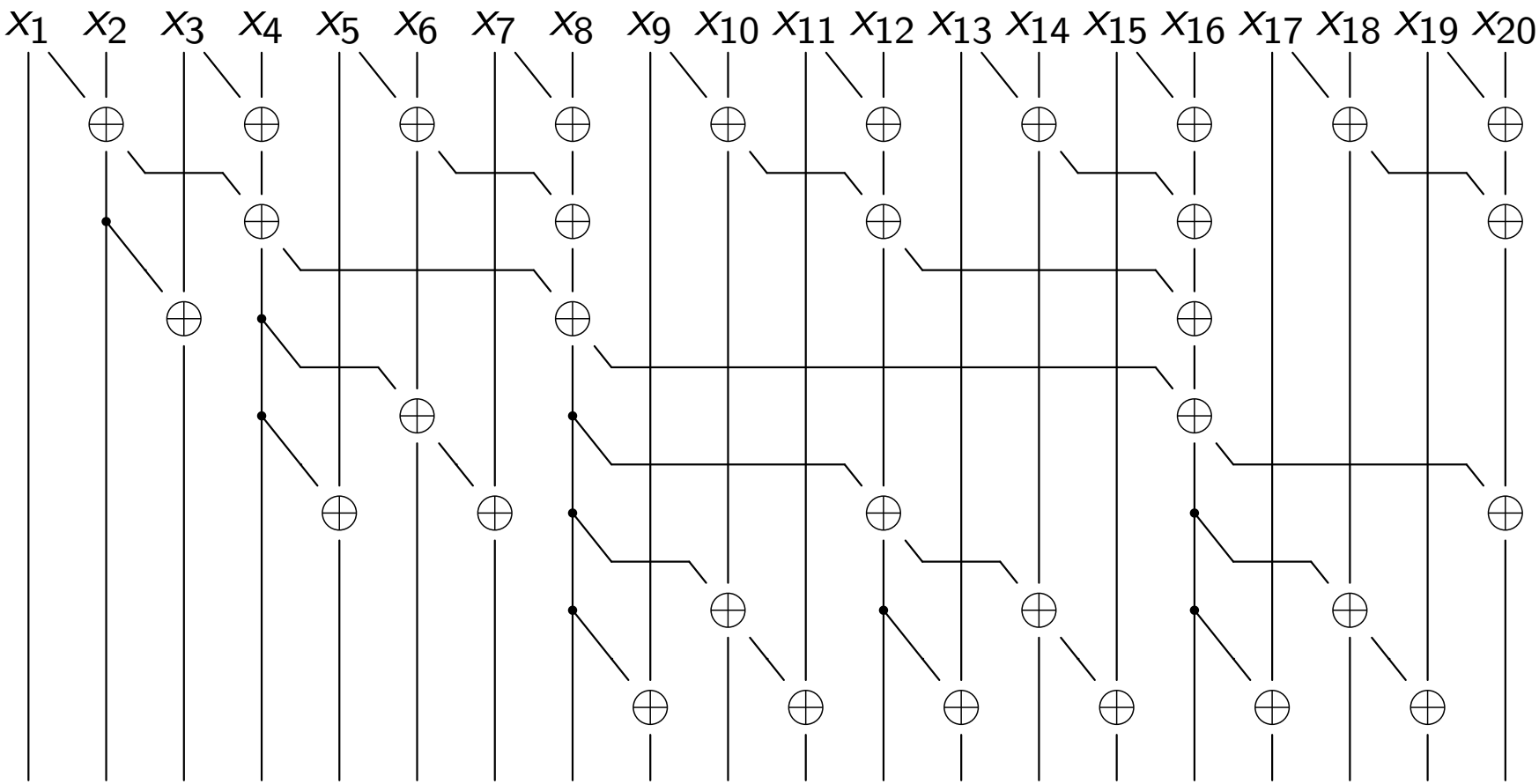
Brent & Kungs Methode:



Brent & Kungs Methode:



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.

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.

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: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$\begin{aligned}
\texttt{scanl1}\ (\oplus)\ (x : xs) &= go\ x\ xs \\
\textbf{where}\ go\ x\ [\,] &= [x] \\
go\ x\ (y : ys) &= x : go\ (x \oplus y)\ ys
\end{aligned}$$

Ein Knuth-ähnliches 0-1-2-Prinzip

Gegeben: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$\begin{aligned}
\texttt{scanl1}\ (\oplus)\ (x : xs) &= go\ x\ xs \\
\textbf{where}\ go\ x\ [\,] &= [x] \\
go\ x\ (y : ys) &= x : go\ (x \oplus y)\ ys
\end{aligned}$$

$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

Ein Knuth-ähnliches 0-1-2-Prinzip

Gegeben:

$$\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

$$\texttt{scanl1}\ (\oplus)\ (x : xs) = go\ x\ xs$$

$$\textbf{where}\ go\ x\ [\,] = [x]$$

$$go\ x\ (y : ys) = x : go\ (x \oplus y)\ ys$$

$$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

$$\textbf{data}\ \mathsf{Three} = \mathsf{Zero} \mid \mathsf{One} \mid \mathsf{Two}$$

Ein Knuth-ähnliches 0-1-2-Prinzip

Gegeben: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$\begin{aligned}\texttt{scanl1}\ (\oplus)\ (x : xs) &= go\ x\ xs \\ \textbf{where}\ go\ x\ [\,] &= [x] \\ go\ x\ (y : ys) &= x : go\ (x \oplus y)\ ys\end{aligned}$$

$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

data Three = Zero | One | Two

Theorem: Wenn für jedes *xs* :: [Three] und assoziatives $(\oplus) :: \text{Three} \to \text{Three} \to \text{Three}$,

$$\texttt{candidate}\ (\oplus)\ xs \quad = \quad \texttt{scanl1}\ (\oplus)\ xs\,,$$

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

Warum 0-1-2? Und wie?

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

Warum 0-1-2? Und wie?

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

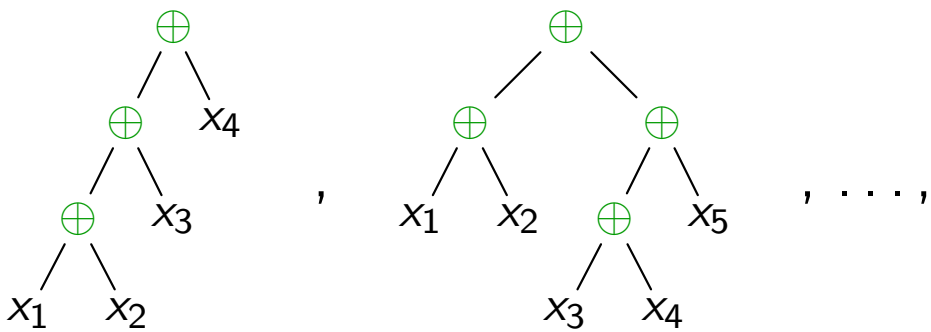
Antwort: Eine neue Liste erzeugen, bestehend aus mittels $\oplus$ and $x_1, \ldots, x_n$ gebildeten Ausdrücken.
Und zwar unabhängig vom α-Typ !

Warum 0-1-2? Und wie?

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

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

Unter diesen Ausdrücken gibt es gute:

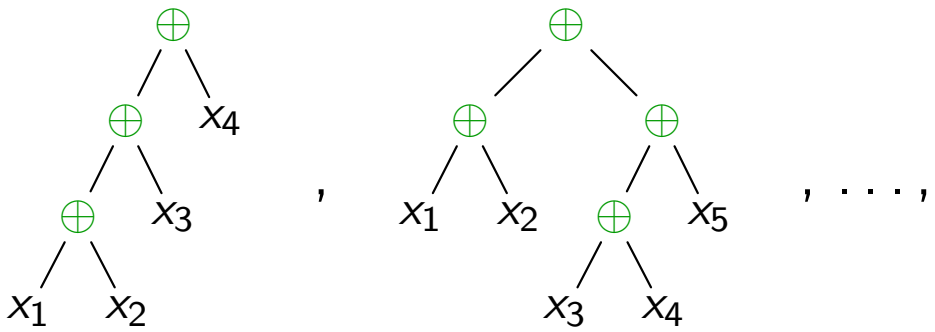


Warum 0-1-2? Und wie?

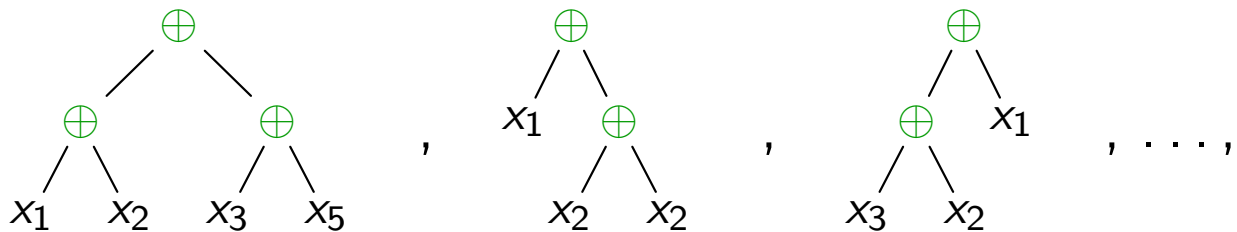
Eine Frage: Was kann `candidate` :: $(\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$
mit einer Operation $\oplus$ und Liste $[x_1, \ldots, x_n]$ tun ?

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

Unter diesen Ausdrücken gibt es gute:

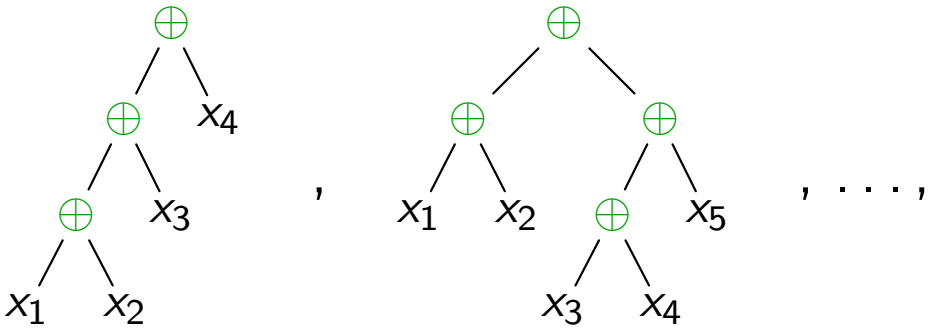


schlechte:

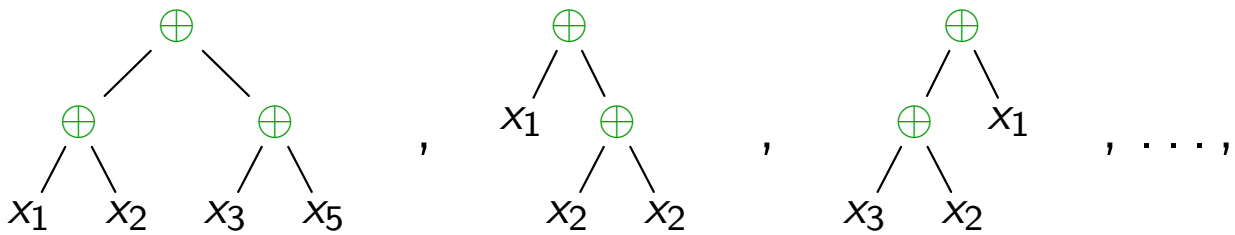


Warum 0-1-2? Und wie?

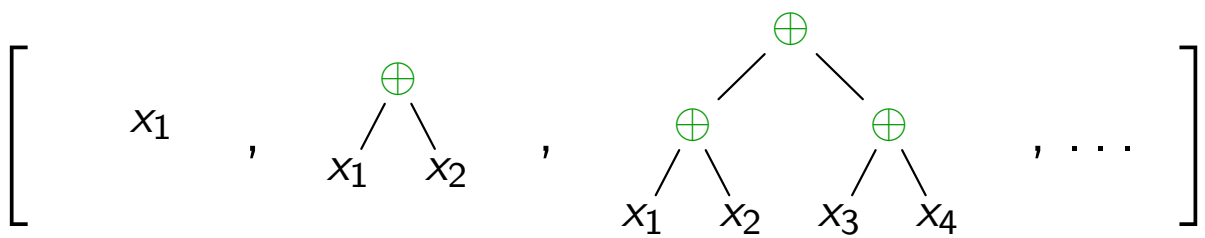
Unter diesen Ausdrücken gibt es gute:



schlechte:



und welche an der falschen Position:



Wie? So!

Es seien

$\oplus_1$	Zero	One	Two
Zero	Zero	One	Two
One	One	Two	Two
Two	Two	Two	Two

und

$\oplus_2$	Zero	One	Two
Zero	Zero	One	Two
One	One	One	Two
Two	Two	One	Two

Wie? So!

Es seien

$\oplus_1$	Zero	One	Two
Zero	Zero	One	Two
One	One	Two	Two
Two	Two	Two	Two

und

$\oplus_2$	Zero	One	Two
Zero	Zero	One	Two
One	One	One	Two
Two	Two	One	Two

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

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

Wie? So!

Es seien

$\oplus_1$	Zero	One	Two
Zero	Zero	One	Two
One	One	Two	Two
Two	Two	Two	Two

und

$\oplus_2$	Zero	One	Two
Zero	Zero	One	Two
One	One	One	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:
$$\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$
$$\texttt{scanl1}\ (\oplus)\ (x : xs) = go\ x\ xs$$
$$\textbf{where}\ go\ x\ [\,] = [x]$$
$$go\ x\ (y : ys) = x : go\ (x \oplus y)\ ys$$

$$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$$

data Three = Zero | One | Two

Theorem: Wenn für jedes *xs* :: [Three] und assoziatives $(\oplus) :: \text{Three} \to \text{Three} \to \text{Three}$,

$$\texttt{candidate}\ (\oplus)\ xs \quad = \quad \texttt{scanl1}\ (\oplus)\ xs\,,$$

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