Prefix Sum

input: $x[1]\ldots x[n]$

output: $s[1]\ldots s[n]$ with $s[i] = \sum_{j=1}^{i} x[i]$ (w.r.t. operator $*$)

Algorithm 6 PrefixSum($n, x[1]\ldots x[n]$)

```
// compute prefixsums; n = 2^k
if n = 1 then s[1] ← x[1]; return
for 1 ≤ i ≤ n/2 pardo
        a[i] ← x[2i − 1] ∗ x[2i]
z[1], ..., z[n/2] ← PrefixSum(n/2, a[1]...a[n/2])
for 1 ≤ i ≤ n pardo
        i even   : s[i] ← z[i/2]
        i = 1    : s[1] = x[1]
        i odd    : s[i] ← z[(i − 1)/2] ∗ x[i]
```

Prefix Sum

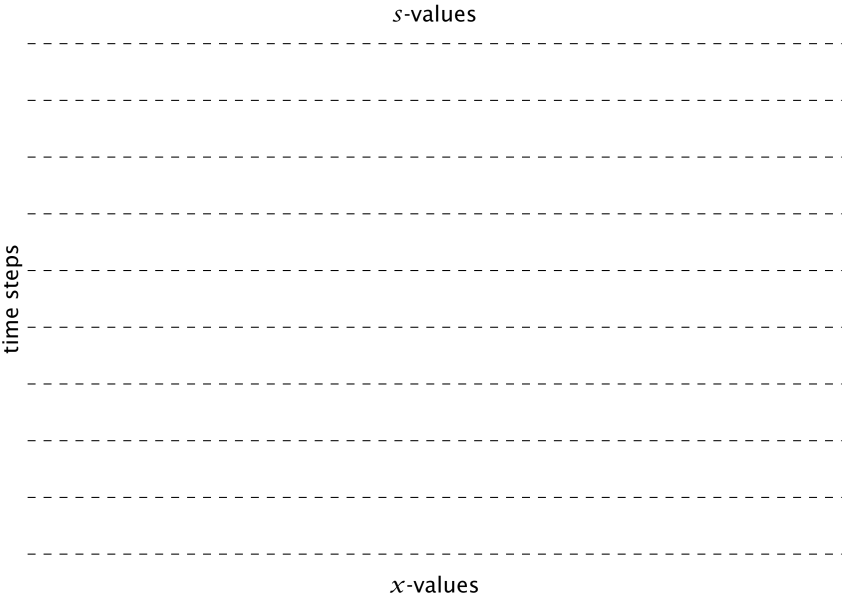
input: $x[1] \dots x[n]$

output: $s[1] \dots s[n]$ with $s[i] = \sum_{j=1}^{i} x[i]$ (w.r.t. operator $*$)

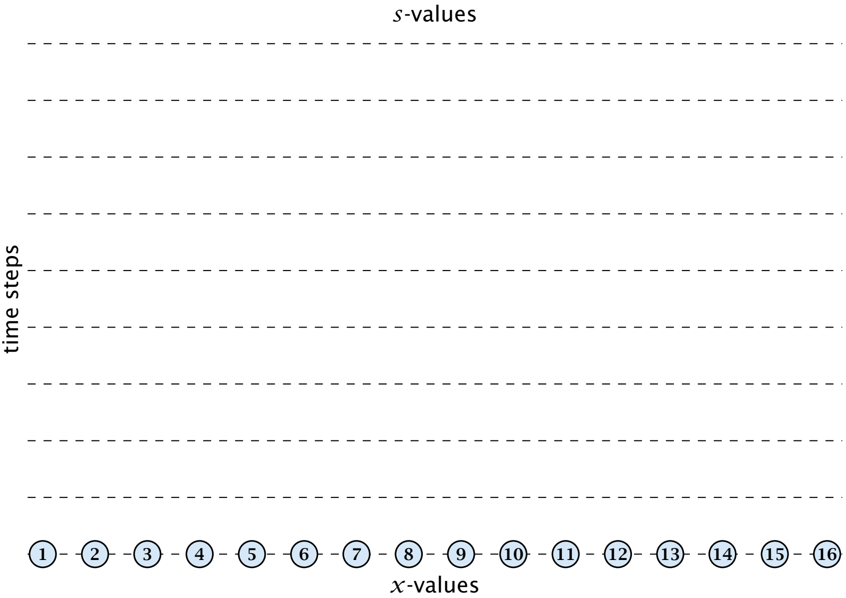
Algorithm 6 PrefixSum$(n, x[1] \dots x[n])$

```
// compute prefixsums; n = 2^k
if n = 1 then s[1] ← x[1]; return
for 1 ≤ i ≤ n/2 pardo
    a[i] ← x[2i − 1] ∗ x[2i]
z[1], ..., z[n/2] ← PrefixSum(n/2, a[1] ... a[n/2])
for 1 ≤ i ≤ n pardo
    i even : s[i] ← z[i/2]
    i = 1  : s[1] = x[1]
    i odd  : s[i] ← z[(i − 1)/2] ∗ x[i]
```

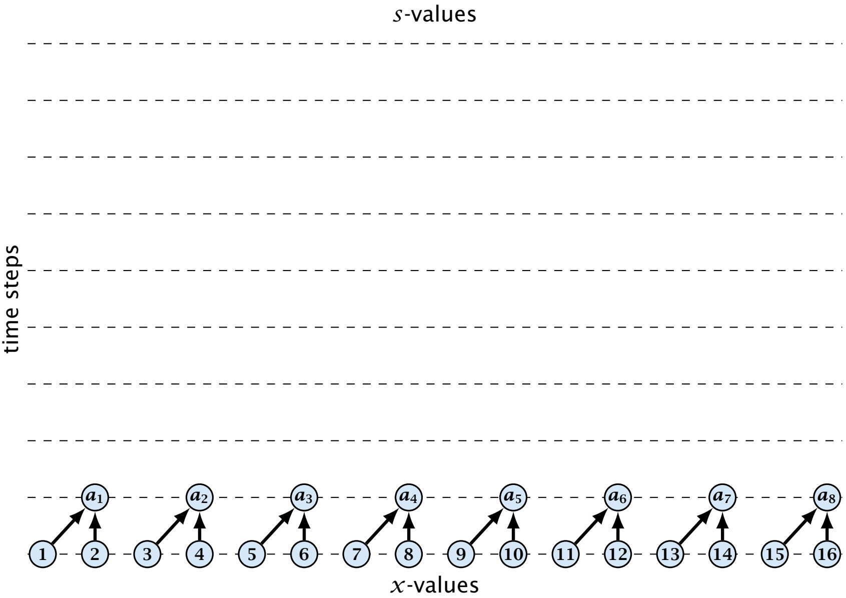
Prefix Sum



Prefix Sum

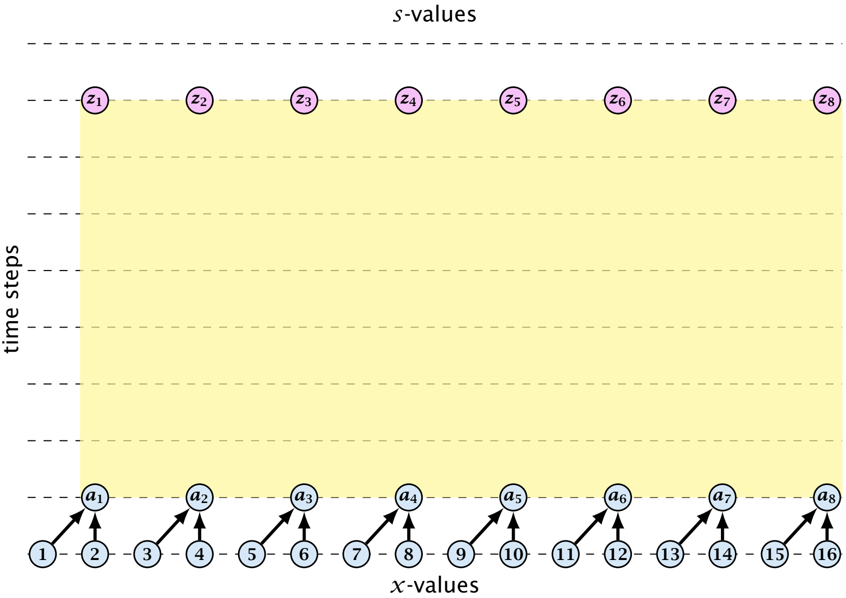


Prefix Sum



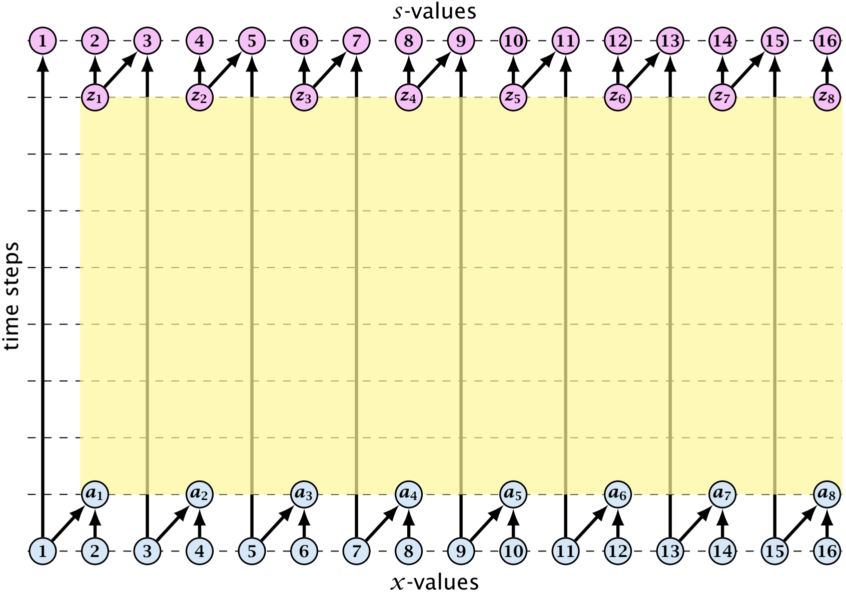
s-values
time steps
a1
a2
a3
a4
a5
a6
a7
a8
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
x-values

Prefix Sum

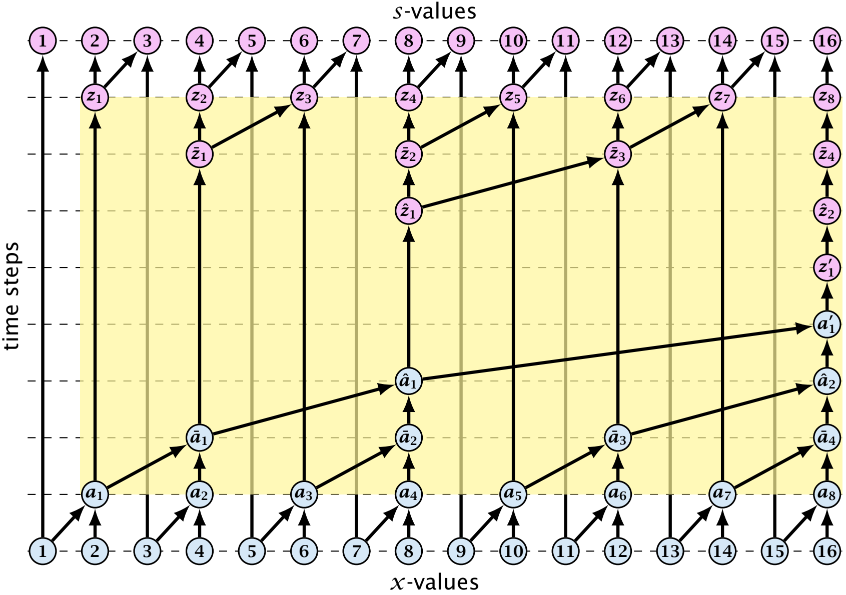


s-values
time steps
z1 z2 z3 z4 z5 z6 z7 z8
a1 a2 a3 a4 a5 a6 a7 a8
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
x-values

Prefix Sum


s-values
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
z1
z2
z3
z4
z5
z6
z7
z8
time steps
a1
a2
a3
a4
a5
a6
a7
a8
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
x-values

Prefix Sum


s-values
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
time steps
z_1
z_2
z_3
z_4
z_5
z_6
z_7
z_8
$\bar{z}_1$
$\bar{z}_2$
$\bar{z}_3$
$\bar{z}_4$
$\hat{z}_1$
$\hat{z}_2$
z'_1
a'_1
$\hat{a}_1$
$\hat{a}_2$
$\bar{a}_1$
$\bar{a}_2$
$\bar{a}_3$
$\bar{a}_4$
a_1
a_2
a_3
a_4
a_5
a_6
a_7
a_8
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
x-values

Prefix Sum

The algorithm uses work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$ for solving Prefix Sum on an EREW-PRAM with n processors.

It is clearly work-optimal.

Theorem 1

On a CREW PRAM a Prefix Sum requires running time $\Omega(\log n)$ regardless of the number of processors.

Prefix Sum

The algorithm uses work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$ for solving Prefix Sum on an EREW-PRAM with n processors.

It is clearly work-optimal.

Theorem 1

On a CREW PRAM a Prefix Sum requires running time $\Omega(\log n)$ regardless of the number of processors.

Prefix Sum

The algorithm uses work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$ for solving Prefix Sum on an EREW-PRAM with n processors.

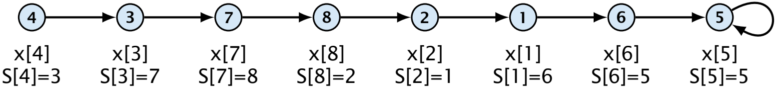
It is clearly work-optimal.

Theorem 1
On a CREW PRAM a Prefix Sum requires running time $\Omega(\log n)$ regardless of the number of processors.

Parallel Prefix

Input: a linked list given by successor pointers; a value $x[i]$ for every list element; an operator $*$;

Output: for every list position ℓ the sum (w.r.t. $*$) of elements after ℓ in the list (including ℓ)



Parallel Prefix

Algorithm 7 ParallelPrefix

for $1 \leq i \leq n$ **pardo**
    $P[i] \leftarrow S[i]$
    while $S[i] \neq S[S[i]]$ **do**
        $x[i] \leftarrow x[i] * x[S[i]]$
        $S[i] \leftarrow S[S[i]]$
    if $P[i] \neq i$ **then** $S[i] \leftarrow x[S(i)]$

The algorithm runs in time $\mathcal{O}(\log n)$.

It has work requirement $\mathcal{O}(n \log n)$. non-optimal

This technique is also known as pointer jumping

Parallel Prefix

Algorithm 7 ParallelPrefix

for $1 \le i \le n$ **pardo**
    $P[i] \leftarrow S[i]$
    while $S[i] \neq S[S[i]]$ **do**
        $x[i] \leftarrow x[i] * x[S[i]]$
        $S[i] \leftarrow S[S[i]]$
    if $P[i] \neq i$ **then** $S[i] \leftarrow x[S(i)]$

The algorithm runs in time $\mathcal{O}(\log n)$.

It has work requirement $\mathcal{O}(n \log n)$. non-optimal

This technique is also known as pointer jumping

Parallel Prefix

Algorithm 7 ParallelPrefix

```
for 1 ≤ i ≤ n pardo
    P[i] ← S[i]
    while S[i] ≠ S[S[i]] do
        x[i] ← x[i] ∗ x[S[i]]
        S[i] ← S[S[i]]
    if P[i] ≠ i then S[i] ← x[S(i)]
```

The algorithm runs in time $\mathcal{O}(\log n)$.

It has work requirement $\mathcal{O}(n \log n)$. non-optimal

This technique is also known as pointer jumping

Parallel Prefix

Algorithm 7 ParallelPrefix

```
for 1 ≤ i ≤ n pardo
    P[i] ← S[i]
    while S[i] ≠ S[S[i]] do
        x[i] ← x[i] * x[S[i]]
        S[i] ← S[S[i]]
    if P[i] ≠ i then S[i] ← x[S(i)]
```

The algorithm runs in time $\mathcal{O}(\log n)$.

It has work requirement $\mathcal{O}(n \log n)$. non-optimal

This technique is also known as pointer jumping

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, compute the sorted squence $C = (c_1, \ldots, c_n)$.

Definition 2

Let $X = (x_1, \ldots, x_t)$ be a sequence. The rank $\mathrm{rank}(y : X)$ of y in X is

$$\mathrm{rank}(y : X) = |\{x \in X \mid x \le y\}|$$

For a sequence $Y = (y_1, \ldots, y_s)$ we define $\mathrm{rank}(Y : X) := (r_1, \ldots, r_s)$ with $r_i = \mathrm{rank}(y_i : X)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, compute the sorted squence $C = (c_1, \ldots, c_n)$.

Definition 2

Let $X = (x_1, \ldots, x_t)$ be a sequence. The rank $\mathrm{rank}(y : X)$ of y in X is

$$\mathrm{rank}(y : X) = |\{x \in X \mid x \le y\}|$$

For a sequence $Y = (y_1, \ldots, y_s)$ we define $\mathrm{rank}(Y : X) := (r_1, \ldots, r_s)$ with $r_i = \mathrm{rank}(y_i : X)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1, \dots, a_n)$ and $B = (b_1, \dots, b_n)$, compute the sorted squence $C = (c_1, \dots, c_n)$.

Definition 2

Let $X = (x_1, \dots, x_t)$ be a sequence. The rank $\mathrm{rank}(y : X)$ of y in X is

$$\mathrm{rank}(y : X) = |\{x \in X \mid x \le y\}|$$

For a sequence $Y = (y_1, \dots, y_s)$ we define $\mathrm{rank}(Y : X) := (r_1, \dots, r_s)$ with $r_i = \mathrm{rank}(y_i : X)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, compute the sorted squence $C = (c_1, \ldots, c_n)$.

Definition 2

Let $X = (x_1, \ldots, x_t)$ be a sequence. The rank $\mathrm{rank}(y : X)$ of y in X is

$$\mathrm{rank}(y : X) = |\{x \in X \mid x \le y\}|$$

For a sequence $Y = (y_1, \ldots, y_s)$ we define $\mathrm{rank}(Y : X) := (r_1, \ldots, r_s)$ with $r_i = \mathrm{rank}(y_i : X)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \dots a_n)$ and $B = (b_1 \dots b_n)$, compute the sorted squence $C = (c_1 \dots c_n)$.

Observation:

We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \mathrm{rank}(c_i : A \cup B)$.

This means we just need to determine rank($x : A \cup B$) for all elements!

Observe, that $\mathrm{rank}(x : A \cup B) = \mathrm{rank}(x : A) + \mathrm{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\mathrm{rank}(x : A)$, and for $x \in B$ we know $\mathrm{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \dots a_n)$ and $B = (b_1 \dots b_n)$, compute the sorted squence $C = (c_1 \dots c_n)$.

Observation:

We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \text{rank}(c_i : A \cup B)$.

This means we just need to determine rank$(x : A \cup B)$ for all elements!

Observe, that $\text{rank}(x : A \cup B) = \text{rank}(x : A) + \text{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\text{rank}(x : A)$, and for $x \in B$ we know $\text{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \dots a_n)$ and $B = (b_1 \dots b_n)$, compute the sorted squence $C = (c_1 \dots c_n)$.

Observation:
We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \text{rank}(c_i : A \cup B)$.

This means we just need to determine $\text{rank}(x : A \cup B)$ for all elements!

Observe, that $\text{rank}(x : A \cup B) = \text{rank}(x : A) + \text{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\text{rank}(x : A)$, and for $x \in B$ we know $\text{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \ldots a_n)$ and $B = (b_1 \ldots b_n)$, compute the sorted squence $C = (c_1 \ldots c_n)$.

Observation:
We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \text{rank}(c_i : A \cup B)$.

This means we just need to determine $\text{rank}(x : A \cup B)$ for all elements!

Observe, that $\text{rank}(x : A \cup B) = \text{rank}(x : A) + \text{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\text{rank}(x : A)$, and for $x \in B$ we know $\text{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \dots a_n)$ and $B = (b_1 \dots b_n)$, compute the sorted squence $C = (c_1 \dots c_n)$.

Observation:
We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \text{rank}(c_i : A \cup B)$.

This means we just need to determine $\text{rank}(x : A \cup B)$ for all elements!

Observe, that $\text{rank}(x : A \cup B) = \text{rank}(x : A) + \text{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\text{rank}(x : A)$, and for $x \in B$ we know $\text{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \ldots a_n)$ and $B = (b_1 \ldots b_n)$, compute the sorted squence $C = (c_1 \ldots c_n)$.

Observation:

We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \text{rank}(c_i : A \cup B)$.

This means we just need to determine $\text{rank}(x : A \cup B)$ for all elements!

Observe, that $\text{rank}(x : A \cup B) = \text{rank}(x : A) + \text{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\text{rank}(x : A)$, and for $x \in B$ we know $\text{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \dots a_n)$ and $B = (b_1 \dots b_n)$, compute the sorted squence $C = (c_1 \dots c_n)$.

Observation:
We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \text{rank}(c_i : A \cup B)$.

This means we just need to determine $\text{rank}(x : A \cup B)$ for all elements!

Observe, that $\text{rank}(x : A \cup B) = \text{rank}(x : A) + \text{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\text{rank}(x : A)$, and for $x \in B$ we know $\text{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Compute $\text{rank}(x : A)$ for all $x \in B$ and $\text{rank}(x : B)$ for all $x \in A$.

can be done in $\mathcal{O}(\log n)$ time with $2n$ processors by binary search

Lemma 3

On a CREW PRAM, Merging can be done in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n \log n)$ work.

not optimal

4.3 Divide & Conquer — Merging

Compute $\text{rank}(x : A)$ for all $x \in B$ and $\text{rank}(x : B)$ for all $x \in A$.

can be done in $\mathcal{O}(\log n)$ time with $2n$ processors by binary search

Lemma 3

On a CREW PRAM, Merging can be done in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n \log n)$ work.

not optimal

4.3 Divide & Conquer — Merging

Compute $\text{rank}(x : A)$ for all $x \in B$ and $\text{rank}(x : B)$ for all $x \in A$.
can be done in $\mathcal{O}(\log n)$ time with $2n$ processors by binary search

Lemma 3
On a CREW PRAM, Merging can be done in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n \log n)$ work.

not optimal

4.3 Divide & Conquer — Merging

Compute $\text{rank}(x : A)$ for all $x \in B$ and $\text{rank}(x : B)$ for all $x \in A$.
can be done in $\mathcal{O}(\log n)$ time with $2n$ processors by binary search

Lemma 3
On a CREW PRAM, Merging can be done in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n \log n)$ work.

not optimal

4.3 Divide & Conquer — Merging

$A = (a_1, \ldots, a_n)$; $B = (b_1, \ldots, b_n)$;
$\log n$ integral; $k := n/\log n$ integral;

Algorithm 8 GenerateSubproblems

```
j_0 ← 0
j_k ← n
for 1 ≤ i ≤ k − 1 pardo
        j_i ← rank(b_{i log n} : A)
for 0 ≤ i ≤ k − 1 pardo
        B_i ← (b_{i log n+1}, ..., b_{(i+1) log n})
        A_i ← (a_{j_i+1}, ..., a_{j_{i+1}})
```

If C_i is the merging of A_i and B_i then the sequence $C_0 \ldots C_{k-1}$ is a sorted sequence.

4.3 Divide & Conquer — Merging

$A = (a_1, \ldots, a_n)$; $B = (b_1, \ldots, b_n)$;
$\log n$ integral; $k := n/\log n$ integral;

Algorithm 8 GenerateSubproblems

$j_0 \leftarrow 0$
$j_k \leftarrow n$
for $1 \le i \le k-1$ **pardo**
  $j_i \leftarrow \mathrm{rank}(b_{i \log n} : A)$
for $0 \le i \le k-1$ **pardo**
  $B_i \leftarrow (b_{i \log n + 1}, \ldots, b_{(i+1)\log n})$
  $A_i \leftarrow (a_{j_i + 1}, \ldots, a_{j_{i+1}})$

If C_i is the merging of A_i and B_i then the sequence $C_0 \ldots C_{k-1}$ is a sorted sequence.

4.3 Divide & Conquer — Merging

$A = (a_1, \dots, a_n)$; $B = (b_1, \dots, b_n)$;
$\log n$ integral; $k := n/\log n$ integral;

Algorithm 8 GenerateSubproblems

```
j_0 ← 0
j_k ← n
for 1 ≤ i ≤ k − 1 pardo
    j_i ← rank(b_{i log n} : A)
for 0 ≤ i ≤ k − 1 pardo
    B_i ← (b_{i log n+1}, ..., b_{(i+1) log n})
    A_i ← (a_{j_i+1}, ..., a_{j_{i+1}})
```

If C_i is the merging of A_i and B_i then the sequence $C_0 \dots C_{k-1}$ is a sorted sequence.

4.3 Divide & Conquer — Merging

$A = (a_1, \ldots, a_n)$; $B = (b_1, \ldots, b_n)$;
$\log n$ integral; $k := n/\log n$ integral;

Algorithm 8 GenerateSubproblems

$j_0 \leftarrow 0$
$j_k \leftarrow n$
for $1 \le i \le k-1$ **pardo**
    $j_i \leftarrow \mathrm{rank}(b_{i \log n} : A)$
for $0 \le i \le k-1$ **pardo**
    $B_i \leftarrow (b_{i \log n + 1}, \ldots, b_{(i+1)\log n})$
    $A_i \leftarrow (a_{j_i + 1}, \ldots, a_{j_{i+1}})$

If C_i is the merging of A_i and B_i then the sequence $C_0 \ldots C_{k-1}$ is a sorted sequence.

4.3 Divide & Conquer — Merging

We can generate the subproblems in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Note that in a sub-problem B_i has length $\log n$.

If we run the algorithm again for every subproblem, (where A_i takes the role of B) we can in time $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n)$ generate subproblems where A_j and B_j have both length at most $\log n$.

Such a subproblem can be solved by a single processor in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|A_i| + |B_i|)$.

Parallelizing the last step gives total work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$.

the resulting algorithm is work optimal

4.3 Divide & Conquer — Merging

We can generate the subproblems in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Note that in a sub-problem B_i has length $\log n$.

If we run the algorithm again for every subproblem, (where A_i takes the role of B) we can in time $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n)$ generate subproblems where A_j and B_j have both length at most $\log n$.

Such a subproblem can be solved by a single processor in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|A_i| + |B_i|)$.

Parallelizing the last step gives total work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$.

the resulting algorithm is work optimal

4.3 Divide & Conquer — Merging

We can generate the subproblems in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Note that in a sub-problem B_i has length $\log n$.

If we run the algorithm again for every subproblem, (where A_i takes the role of B) we can in time $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n)$ generate subproblems where A_j and B_j have both length at most $\log n$.

Such a subproblem can be solved by a single processor in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|A_i| + |B_i|)$.

Parallelizing the last step gives total work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$.

the resulting algorithm is work optimal

4.3 Divide & Conquer — Merging

We can generate the subproblems in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Note that in a sub-problem B_i has length $\log n$.

If we run the algorithm again for every subproblem, (where A_i takes the role of B) we can in time $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n)$ generate subproblems where A_j and B_j have both length at most $\log n$.

Such a subproblem can be solved by a single processor in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|A_i| + |B_i|)$.

Parallelizing the last step gives total work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$.

the resulting algorithm is work optimal

4.3 Divide & Conquer — Merging

We can generate the subproblems in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Note that in a sub-problem B_i has length $\log n$.

If we run the algorithm again for every subproblem, (where A_i takes the role of B) we can in time $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n)$ generate subproblems where A_j and B_j have both length at most $\log n$.

Such a subproblem can be solved by a single processor in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|A_i| + |B_i|)$.

Parallelizing the last step gives total work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$.

the resulting algorithm is work optimal

4.3 Divide & Conquer — Merging

We can generate the subproblems in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Note that in a sub-problem B_i has length $\log n$.

If we run the algorithm again for every subproblem, (where A_i takes the role of B) we can in time $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n)$ generate subproblems where A_j and B_j have both length at most $\log n$.

Such a subproblem can be solved by a single processor in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|A_i| + |B_i|)$.

Parallelizing the last step gives total work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$.

the resulting algorithm is work optimal

4.4 Maximum Computation

Lemma 4

On a CRCW PRAM the maximum of n numbers can be computed in time $\mathcal{O}(1)$ with n^2 processors.

proof on board...

4.4 Maximum Computation

Lemma 4

On a CRCW PRAM the maximum of n numbers can be computed in time $\mathcal{O}(1)$ with n^2 processors.

proof on board...

4.4 Maximum Computation

Lemma 5

On a CRCW PRAM the maximum of n numbers can be computed in time $\mathcal{O}(\log\log n)$ with n processors and work $\mathcal{O}(n\log\log n)$.

proof on board...

4.4 Maximum Computation

Lemma 5
On a CRCW PRAM the maximum of n numbers can be computed in time $\mathcal{O}(\log\log n)$ with n processors and work $\mathcal{O}(n\log\log n)$.

proof on board...

4.4 Maximum Computation

Lemma 6

On a CRCW PRAM the maximum of n numbers can be computed in time $\mathcal{O}(\log\log n)$ with n processors and work $\mathcal{O}(n)$.

proof on board...

4.4 Maximum Computation

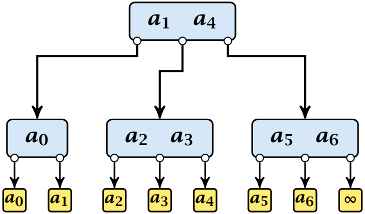
Lemma 6
On a CRCW PRAM the maximum of n numbers can be computed in time $\mathcal{O}(\log\log n)$ with n processors and work $\mathcal{O}(n)$.

proof on board...

4.5 Inserting into a $(2,3)$-tree

Given a $(2,3)$-tree with n elements, and a sequence $x_0 < x_1 < x_2 < \cdots < x_k$ of elements. We want to insert elements $x_1, \ldots, x_k$ into the tree ($k \ll n$).

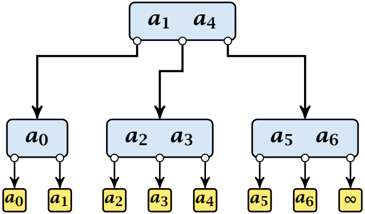
time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$



4.5 Inserting into a $(2,3)$-tree

Given a $(2,3)$-tree with n elements, and a sequence $x_0 < x_1 < x_2 < \cdots < x_k$ of elements. We want to insert elements $x_1, \ldots, x_k$ into the tree ($k \ll n$).

time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$



4.5 Inserting into a $(2,3)$-tree

1. determine for every x_i the leaf element before which it has to be inserted
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$; CREW PRAM

   all x_i's that have to be inserted before the same element form a chain

2. determine the largest/smallest/middle element of every chain
   time: $\mathcal{O}(1)$; work: $\mathcal{O}(k)$;

3. insert the middle element of every chain
   compute new chains
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k_i \log n)$; k_i= #inserted elements
   (computing new chains is constant time)

4. repeat Step 3 for logarithmically many rounds
   time: $\mathcal{O}(\log n \log k)$; work: $\mathcal{O}(k \log n)$;

4.5 Inserting into a $(2,3)$-tree

1. determine for every x_i the leaf element before which it has to be inserted
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$; CREW PRAM

   all x_i's that have to be inserted before the same element form a chain
2. determine the largest/smallest/middle element of every chain
   time: $\mathcal{O}(1)$; work: $\mathcal{O}(k)$;
3. insert the middle element of every chain
   compute new chains
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k_i \log n)$; k_i= #inserted elements
   (computing new chains is constant time)
4. repeat Step 3 for logarithmically many rounds
   time: $\mathcal{O}(\log n \log k)$; work: $\mathcal{O}(k \log n)$;

4.5 Inserting into a $(2,3)$-tree

1. determine for every x_i the leaf element before which it has to be inserted
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$; CREW PRAM

   all x_i's that have to be inserted before the same element form a chain
2. determine the largest/smallest/middle element of every chain
   time: $\mathcal{O}(1)$; work: $\mathcal{O}(k)$;
3. insert the middle element of every chain
   compute new chains
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k_i \log n)$; k_i= #inserted elements
   (computing new chains is constant time)
4. repeat Step 3 for logarithmically many rounds
   time: $\mathcal{O}(\log n \log k)$; work: $\mathcal{O}(k \log n)$;

4.5 Inserting into a $(2,3)$-tree

1. determine for every x_i the leaf element before which it has to be inserted
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$; CREW PRAM

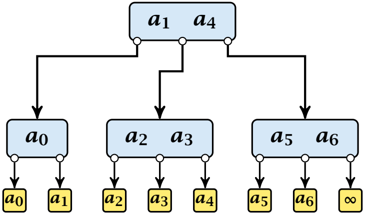
   all x_i's that have to be inserted before the same element form a chain
2. determine the largest/smallest/middle element of every chain
   time: $\mathcal{O}(1)$; work: $\mathcal{O}(k)$;
3. insert the middle element of every chain
   compute new chains
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k_i \log n)$; k_i= #inserted elements
   (computing new chains is constant time)
4. repeat Step 3 for logarithmically many rounds
   time: $\mathcal{O}(\log n \log k)$; work: $\mathcal{O}(k \log n)$;

4.5 Inserting into a $(2,3)$-tree

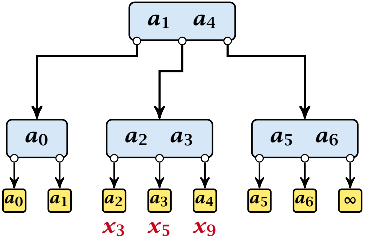
1. determine for every x_i the leaf element before which it has to be inserted
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$; CREW PRAM

   all x_i's that have to be inserted before the same element form a chain
2. determine the largest/smallest/middle element of every chain
   time: $\mathcal{O}(1)$; work: $\mathcal{O}(k)$;
3. insert the middle element of every chain
   compute new chains
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k_i \log n)$; k_i= #inserted elements
   (computing new chains is constant time)
4. repeat Step 3 for logarithmically many rounds
   time: $\mathcal{O}(\log n \log k)$; work: $\mathcal{O}(k \log n)$;

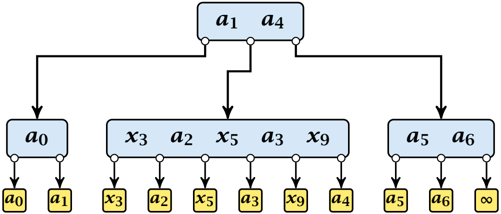
Step 3



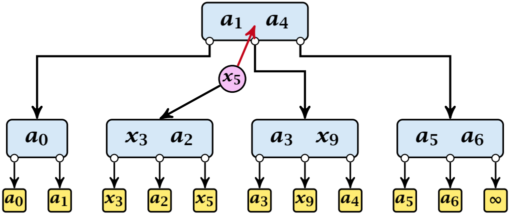
Step 3



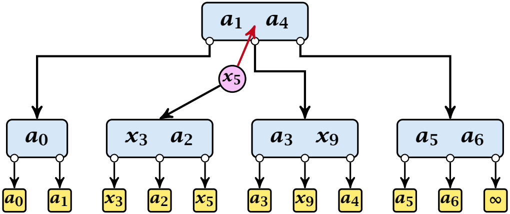
Step 3



Step 3

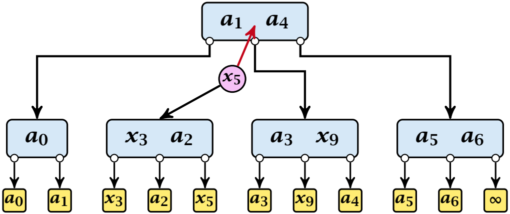


Step 3



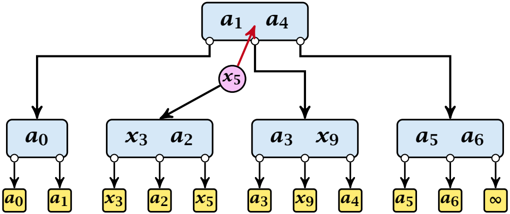
- each internal node is split into at most two parts

Step 3



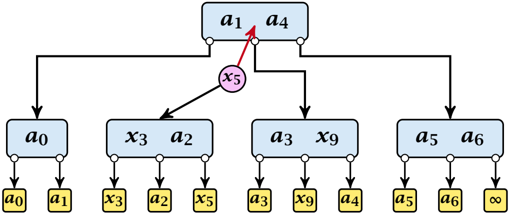
- each internal node is split into at most two parts
- each split operation promotes at most one element

Step 3



- each internal node is split into at most two parts
- each split operation promotes at most one element
- hence, on every level we want to insert at most one element per successor pointer

Step 3



- each internal node is split into at most two parts
- each split operation promotes at most one element
- hence, on every level we want to insert at most one element per successor pointer
- we can use the same routine for every level

4.5 Inserting into a $(2,3)$-tree

- Step 3, works in phases; one phase for every level of the tree
- Step 4, works in rounds; in each round a different set of elements is inserted

Observation

We can start with phase i of round r as long as phase i of round $r-1$ and (of course), phase $i-1$ of round r has finished.

This is called Pipelining. Using this technique we can perform all rounds in Step 4 in just $\mathcal{O}(\log k + \log n)$ many parallel steps.

4.5 Inserting into a $(2, 3)$-tree

- Step 3, works in phases; one phase for every level of the tree
- Step 4, works in rounds; in each round a different set of elements is inserted

Observation

We can start with phase i of round r as long as phase i of round $r - 1$ and (of course), phase $i - 1$ of round r has finished.

This is called Pipelining. Using this technique we can perform all rounds in Step 4 in just $\mathcal{O}(\log k + \log n)$ many parallel steps.

4.5 Inserting into a $(2,3)$-tree

- Step 3, works in phases; one phase for every level of the tree
- Step 4, works in rounds; in each round a different set of elements is inserted

Observation

We can start with phase i of round r as long as phase i of round $r-1$ and (of course), phase $i-1$ of round r has finished.

This is called Pipelining. Using this technique we can perform all rounds in Step 4 in just $\mathcal{O}(\log k + \log n)$ many parallel steps.

4.5 Inserting into a $(2, 3)$-tree

- Step 3, works in phases; one phase for every level of the tree
- Step 4, works in rounds; in each round a different set of elements is inserted

Observation

We can start with phase i of round r as long as phase i of round $r - 1$ and (of course), phase $i - 1$ of round r has finished.

This is called Pipelining. Using this technique we can perform all rounds in Step 4 in just $\mathcal{O}(\log k + \log n)$ many parallel steps.

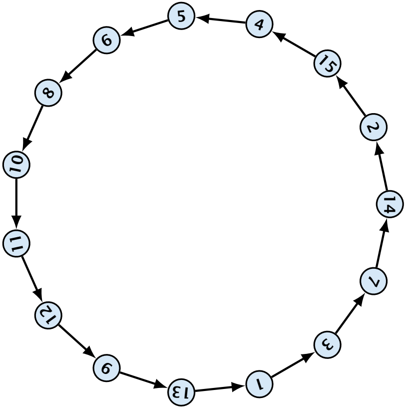
4.6 Symmetry Breaking

The following algorithm colors an n-node cycle with $\lceil \log n \rceil$ colors.

Algorithm 9 BasicColoring

```
for 1 ≤ i ≤ n pardo
    col(i) ← i
    k_i ← smallest bitpos where col(i) and col(S(i)) differ
    col'(i) ← 2k + col(i)_k
```

4.6 Symmetry Breaking



v	col	k	col′
1	0001	1	2
3	0011	2	4
7	0111	0	1
14	1110	2	5
2	0010	0	0
15	1111	0	1
4	0100	0	0
5	0101	0	1
6	0110	1	3
8	1000	1	2
10	1010	0	0
11	1011	0	1
12	1100	0	0
9	1001	2	4
13	1101	2	5

4.6 Symmetry Breaking

Applying the algorithm to a coloring with bit-length t generates a coloring with largest color at most

$$2(t-1)+1$$

and bit-length at most

$$\lceil \log_2(2(t-1)+1) \rceil \le \lceil \log_2(t-1) \rceil + 1 \le \lceil \log_2(t) \rceil + 1$$

Applying the algorithm repeatedly generates a constant number of colors after $\log^* n$ operations.

4.6 Symmetry Breaking

Applying the algorithm to a coloring with bit-length t generates a coloring with largest color at most

$$2(t-1)+1$$

and bit-length at most

$$\lceil \log_2(2(t-1)+1) \rceil \le \lceil \log_2(t-1) \rceil + 1 \le \lceil \log_2(t) \rceil + 1$$

Applying the algorithm repeatedly generates a constant number of colors after $\log^* n$ operations.

4.6 Symmetry Breaking

Applying the algorithm to a coloring with bit-length t generates a coloring with largest color at most

$$2(t-1)+1$$

and bit-length at most

$$\lceil \log_2(2(t-1)+1) \rceil \le \lceil \log_2(t-1) \rceil + 1 \le \lceil \log_2(t) \rceil + 1$$

Applying the algorithm repeatedly generates a constant number of colors after $\log^* n$ operations.

4.6 Symmetry Breaking

Applying the algorithm to a coloring with bit-length t generates a coloring with largest color at most

$$2(t-1)+1$$

and bit-length at most

$$\lceil \log_2(2(t-1)+1)\rceil \le \lceil \log_2(t-1)\rceil + 1 \le \lceil \log_2(t)\rceil + 1$$

Applying the algorithm repeatedly generates a constant number of colors after $\log^* n$ operations.

4.6 Symmetry Breaking

Applying the algorithm to a coloring with bit-length t generates a coloring with largest color at most

$$2(t-1)+1$$

and bit-length at most

$$\lceil \log_2(2(t-1)+1)\rceil \le \lceil \log_2(t-1)\rceil + 1 \le \lceil \log_2(t)\rceil + 1$$

Applying the algorithm repeatedly generates a constant number of colors after $\log^* n$ operations.

4.6 Symmetry Breaking

As long as the bit-length $t \geq 4$ the bit-length decreases.

Applying the algorithm with bit-length 3 gives a coloring with colors in the range $0, \ldots, 5 = 2t - 1$.

We can improve to a 3-coloring by successively re-coloring nodes from a color-class:

Algorithm 10 ReColor

```
for ℓ ← 5 to 3
    for 1 ≤ i ≤ n pardo
        if col(i) = ℓ then
        col(i) ← min{{0,1,2} \ {col(P[i]), col(S[i])}}
```

This requires time $\mathcal{O}(1)$ and work $\mathcal{O}(n)$.

4.6 Symmetry Breaking

As long as the bit-length $t \geq 4$ the bit-length decreases.

Applying the algorithm with bit-length 3 gives a coloring with colors in the range $0, \ldots, 5 = 2t - 1$.

We can improve to a 3-coloring by successively re-coloring nodes from a color-class:

Algorithm 10 ReColor

```
for ℓ ← 5 to 3
    for 1 ≤ i ≤ n pardo
        if col(i) = ℓ then
            col(i) ← min{{0,1,2} \ {col(P[i]), col(S[i])}}
```

This requires time $\mathcal{O}(1)$ and work $\mathcal{O}(n)$.

4.6 Symmetry Breaking

As long as the bit-length $t \geq 4$ the bit-length decreases.

Applying the algorithm with bit-length 3 gives a coloring with colors in the range $0, \dots, 5 = 2t - 1$.

We can improve to a 3-coloring by successively re-coloring nodes from a color-class:

Algorithm 10 ReColor

```
for ℓ ← 5 to 3
    for 1 ≤ i ≤ n pardo
        if col(i) = ℓ then
            col(i) ← min{{0,1,2} \ {col(P[i]), col(S[i])}}
```

This requires time $\mathcal{O}(1)$ and work $\mathcal{O}(n)$.

4.6 Symmetry Breaking

As long as the bit-length $t \geq 4$ the bit-length decreases.

Applying the algorithm with bit-length 3 gives a coloring with colors in the range $0, \ldots, 5 = 2t - 1$.

We can improve to a 3-coloring by successively re-coloring nodes from a color-class:

Algorithm 10 ReColor

```
for ℓ ← 5 to 3
    for 1 ≤ i ≤ n pardo
        if col(i) = ℓ then
            col(i) ← min{{0,1,2} \ {col(P[i]), col(S[i])}}
```

This requires time $\mathcal{O}(1)$ and work $\mathcal{O}(n)$.

4.6 Symmetry Breaking

Lemma 7

We can color vertices in a ring with three colors in $\mathcal{O}(\log^* n)$ *time and with* $\mathcal{O}(n \log^* n)$ *work.*

not work optimal

4.6 Symmetry Breaking

Lemma 8

Given n integers in the range $0, \ldots, \mathcal{O}(\log n)$, there is an algorithm that sorts these numbers in $\mathcal{O}(\log n)$ time using a linear number of operations.

Proof: Exercise!

4.6 Symmetry Breaking

Algorithm 11 OptColor

```
for 1 ≤ i ≤ n pardo
    col(i) ← i
apply BasicColoring once
sort vertices by colors
for ℓ = 2⌈log n⌉ to 3 do
    for all vertices i of color ℓ pardo
        col(i) ← min{{0,1,2} \ {col(P[i]), col(S[i])}}
```

Lemma 9

A ring can be colored with 3 colors in time $\mathcal{O}(\log n)$ and with work $\mathcal{O}(n)$.

work optimal but not too fast