### Very simple Algorithm (Idea)

<table>
<thead>
<tr>
<th>22</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The algorithm involves a series of comparisons and swaps, aiming to sort the elements in ascending order. Each step is designed to progressively move closer to a sorted state, with each iteration refining the arrangement of the array. The specific details of the algorithm, such as the criteria for comparisons and the logic behind the swaps, are crucial for understanding its efficiency and effectiveness.
Very simple Algorithm (Idea)

<table>
<thead>
<tr>
<th>22</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>34</td>
<td>12</td>
<td>14</td>
<td>56</td>
<td>23</td>
<td>67</td>
<td>49</td>
<td>27</td>
<td>61</td>
<td>52</td>
<td>57</td>
<td>59</td>
<td>26</td>
<td>41</td>
</tr>
</tbody>
</table>
## Very simple Algorithm (Idea)

<table>
<thead>
<tr>
<th></th>
<th>22</th>
<th>33</th>
<th>41</th>
<th>26</th>
<th>59</th>
<th>57</th>
<th>52</th>
<th>61</th>
<th>27</th>
<th>49</th>
<th>67</th>
<th>23</th>
<th>56</th>
<th>14</th>
<th>12</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

```plaintext
34 12 14 56 23 67 49 27 61 52 57 59 26 41 33 22
```
### Very simple Algorithm (Idea)

<table>
<thead>
<tr>
<th>22</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>59</td>
</tr>
<tr>
<td>59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>57</td>
</tr>
<tr>
<td>57</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>52</td>
</tr>
<tr>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>61</td>
</tr>
<tr>
<td>61</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>49</td>
</tr>
<tr>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>67</td>
</tr>
<tr>
<td>67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>56</td>
</tr>
<tr>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>34</td>
<td>34</td>
<td>12</td>
<td>14</td>
<td>56</td>
<td>23</td>
<td>67</td>
<td>49</td>
<td>27</td>
<td>61</td>
<td>52</td>
<td>57</td>
<td>59</td>
<td>26</td>
<td>41</td>
<td>33</td>
</tr>
</tbody>
</table>
Very simple Algorithm (Idea)
Very simple Algorithm (Idea)
# Very simple Algorithm (Idea)

<table>
<thead>
<tr>
<th>22</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>41</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>59</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>57</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>52</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>61</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>67</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>56</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>34</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td>12</td>
<td>14</td>
<td>56</td>
<td>23</td>
<td>67</td>
<td>49</td>
<td>27</td>
<td>61</td>
<td>52</td>
<td>57</td>
<td>59</td>
<td>26</td>
<td>41</td>
<td>33</td>
<td>22</td>
<td>10</td>
</tr>
</tbody>
</table>
**Very simple Sorting Algorithm**

- **Idea:** Compute the position for each element.
- Compare pairwise all elements and count the number of smaller elements.
- Use \( n^2 \) processors.
- **Programm:** SimpleSort
  
  **Eingabe:** \( s_1, \ldots, s_n \).
  
  for all \( P_{i,j} \) where \( 1 \leq i, j \leq n \) do in parallel
  
  if \( s_i > s_j \) then \( P_{i,j}(1) \rightarrow R_{i,j} \) else \( P_{i,j}(0) \rightarrow R_{i,j} \)
  
  for all \( i \) where \( 1 \leq i \leq n \) do in parallel
  
  for all \( P_{i,j} \) where \( 1 \leq j \leq n \) do in parallel
  
  Processors \( P_{i,j} \) bestimmen \( q_i = \sum_{l=1}^{n} R_{i,l} \).
  
  \( P_i(s_i) \rightarrow R_{q_i+1} \).

- Complexity: \( T(n) = O(\log n) \) and \( P(n) = n^2 \).
- Efficiency: \( \frac{O(n \log n)}{n^2 \cdot O(\log n)} = O\left(\frac{1}{n}\right) \).
- **Model:** CREW.
Very simple Sorting Algorithm

- **Idea:** Compute the position for each element.
- **Compare pairwise all elements and count the number of smaller elements.**
- **Use** $n^2$ processors.
- **Programm:** SimpleSort
  
  **Eingabe:** $s_1, \cdots, s_n$.
  
  for all $P_{i,j}$ where $1 \leq i, j \leq n$ do in parallel
  
  if $s_i > s_j$ then $P_{i,j}(1) \rightarrow R_{i,j}$ else $P_{i,j}(0) \rightarrow R_{i,j}$
  
  for all $i$ where $1 \leq i \leq n$ do in parallel
  
  for all $P_{i,j}$ where $1 \leq j \leq n$ do in parallel
  
  Processors $P_{i,j}$ bestimmen $q_i = \sum_{l=1}^{n} R_{i,l}$.
  
  $P_i(s_i) \rightarrow R_{q_i+1}$.

- **Complexity:** $T(n) = O(\log n)$ and $P(n) = n^2$.

- **Efficiency:** $\frac{O(n \log n)}{n^2 \cdot O(\log n)} = O\left(\frac{1}{n}\right)$.

- **Model:** CREW.
Very simple Sorting Algorithm

- Idea: Compute the position for each element.
- Compare pairwise all elements and count the number of smaller elements.
- Use $n^2$ processors.

Programm: SimpleSort

Eingabe: $s_1, \ldots, s_n$.

for all $P_{i,j}$ where $1 \leq i, j \leq n$ do in parallel
   if $s_i > s_j$ then $P_{i,j}(1) \rightarrow R_{i,j}$ else $P_{i,j}(0) \rightarrow R_{i,j}$

for all $i$ where $1 \leq i \leq n$ do in parallel
   for all $P_{i,j}$ where $1 \leq j \leq n$ do in parallel
      Processors $P_{i,j}$ bestimmen $q_i = \sum_{l=1}^{n} R_{i,l}$.
      $P_i(s_i) \rightarrow R_{q_i+1}$.

- Complexity: $T(n) = O(\log n)$ and $P(n) = n^2$.
- Efficiency: $\frac{O(n \log n)}{n^2 \cdot O(\log n)} = O\left(\frac{1}{n}\right)$.
- Model: CREW.
Very simple Sorting Algorithm

- Idea: Compute the position for each element.
- Compare pairwise all elements and count the number of smaller elements.
- Use $n^2$ processors.

Programm: SimpleSort

Eingabe: $s_1, \ldots, s_n$.

for all $P_{i,j}$ where $1 \leq i, j \leq n$ do in parallel

if $s_i > s_j$ then $P_{i,j}(1) \rightarrow R_{i,j}$ else $P_{i,j}(0) \rightarrow R_{i,j}$

for all $i$ where $1 \leq i \leq n$ do in parallel

for all $P_{i,j}$ where $1 \leq j \leq n$ do in parallel

Processors $P_{i,j}$ bestimmen $q_i = \sum_{l=1}^{n} R_{i,l}$.

$P_i(s_i) \rightarrow R_{q_i+1}$.

- Complexity: $T(n) = O(\log n)$ and $P(n) = n^2$.
- Efficiency: $\frac{O(n \log n)}{n^2 \cdot O(\log n)} = O\left(\frac{1}{n}\right)$.

- Model: CREW.
Very simple Sorting Algorithm

- Idea: Compute the position for each element.
- Compare pairwise all elements and count the number of smaller elements.
- Use \( n^2 \) processors.

Programm: SimpleSort

Eingabe: \( s_1, \ldots, s_n \).

for all \( P_{i,j} \) where \( 1 \leq i, j \leq n \) do in parallel
  if \( s_i > s_j \) then \( P_{i,j}(1) \rightarrow R_{i,j} \) else \( P_{i,j}(0) \rightarrow R_{i,j} \)

for all \( i \) where \( 1 \leq i \leq n \) do in parallel
  for all \( P_{i,j} \) where \( 1 \leq j \leq n \) do in parallel
    Processors \( P_{i,j} \) bestimmen \( q_i = \sum_{l=1}^{n} R_{i,l} \).
    \( P_i(s_i) \rightarrow R_{q_i+1} \).

Complexity: \( T(n) = O(\log n) \) and \( P(n) = n^2 \).

Efficiency: \( \frac{O(n \log n)}{n^2 \cdot O(\log n)} = O\left(\frac{1}{n}\right) \).

Model: CREW.
Very simple Sorting Algorithm

- Idea: Compute the position for each element.
- Compare pairwise all elements and count the number of smaller elements.
- Use \( n^2 \) processors.

**Programm: SimpleSort**

**Eingabe:** \( s_1, \ldots, s_n \).

**for all** \( P_{i,j} \) where \( 1 \leq i, j \leq n \) **do in parallel**

- if \( s_i > s_j \) **then** \( P_{i,j}(1) \rightarrow R_{i,j} \) **else** \( P_{i,j}(0) \rightarrow R_{i,j} \)

**for all** \( i \) where \( 1 \leq i \leq n \) **do in parallel**

- **for all** \( P_{i,j} \) where \( 1 \leq j \leq n \) **do in parallel**

  - Processors \( P_{i,j} \) bestimmen \( q_i = \sum_{l=1}^{n} R_{i,l} \).

  - \( P_i(s_i) \rightarrow R_{q_i+1} \).

**Complexity:** \( T(n) = O(\log n) \) and \( P(n) = n^2 \).

**Efficiency:** \( \frac{O(n \log n)}{n^2 \cdot O(\log n)} = O\left(\frac{1}{n}\right) \).

**Model:** CREW.
Very simple Sorting Algorithm

- **Idea:** Compute the position for each element.
- **Compare pairwise all elements and count the number of smaller elements.**
- **Use** $n^2$ processors.
- **Programm:** SimpleSort

**Eingabe:** $s_1, \ldots, s_n$.

**for all** $P_{i,j}$ where $1 \leq i, j \leq n$ **do in parallel**

- **if** $s_i > s_j$ **then** $P_{i,j}(1) \rightarrow R_{i,j}$ **else** $P_{i,j}(0) \rightarrow R_{i,j}$

**for all** $i$ where $1 \leq i \leq n$ **do in parallel**

- **for all** $P_{i,j}$ where $1 \leq j \leq n$ **do in parallel**

  **Processors** $P_{i,j}$ bestimmen $q_i = \sum_{l=1}^{n} R_{i,l}$.

  $P_i(s_i) \rightarrow R_{q_i+1}$.

- **Complexity:** $T(n) = O(\log n)$ and $P(n) = n^2$.
- **Efficiency:** $\frac{O(n \log n)}{n^2 \cdot O(\log n)} = O\left(\frac{1}{n}\right)$.
- **Model:** CREW.
Very simple Sorting Algorithm

- Idea: Compute the position for each element.
- Compare pairwise all elements and count the number of smaller elements.
- Use $n^2$ processors.

Programm: SimpleSort

Eingabe: $s_1, \cdots, s_n$.

for all $P_{i,j}$ where $1 \leq i, j \leq n$ do in parallel
  if $s_i > s_j$ then $P_{i,j}(1) \rightarrow R_{i,j}$ else $P_{i,j}(0) \rightarrow R_{i,j}$

for all $i$ where $1 \leq i \leq n$ do in parallel
  for all $P_{i,j}$ where $1 \leq j \leq n$ do in parallel
    Processors $P_{i,j}$ bestimmen $q_i = \sum_{l=1}^{n} R_{i,l}$.
    $P_i(s_i) \rightarrow R_{q_i+1}$.

- Complexity: $T(n) = O(\log n)$ and $P(n) = n^2$.
- Efficiency: $\frac{O(n \log n)}{n^2 \cdot O(\log n)} = O\left(\frac{1}{n}\right)$.
- Model: CREW.
Very simple Sorting Algorithm

- Idea: Compute the position for each element.
- Compare pairwise all elements and count the number of smaller elements.
- Use $n^2$ processors.

**Programm: SimpleSort**

**Eingabe:** $s_1, \ldots, s_n$.

*for all* $P_{i,j}$ where $1 \leq i, j \leq n$ *do in parallel*

- *if* $s_i > s_j$ *then* $P_{i,j}(1) \rightarrow R_{i,j}$ *else* $P_{i,j}(0) \rightarrow R_{i,j}$

*for all* $i$ where $1 \leq i \leq n$ *do in parallel*

- *for all* $P_{i,j}$ where $1 \leq j \leq n$ *do in parallel*

  - Processors $P_{i,j}$ bestimmen $q_i = \sum_{l=1}^{n} R_{i,l}$.

  - $P_i(s_i) \rightarrow R_{q_i+1}$.

- **Complexity:** $T(n) = O(\log n)$ and $P(n) = n^2$.

- **Efficiency:** $\frac{O(n \log n)}{n^2 \cdot O(\log n)} = O\left(\frac{1}{n}\right)$.

- **Model:** CREW.
Improved Algorithm for CREW

- Work with $P(n)$ processors ($P(n) \leq n$).
- Split the input in blocks of size $O(n/P(n))$.
- Sort parallel each block.
- Merge the blocks pairwise and parallel.

Complexity: $T(n) = O(n/P(n) \cdot \log n + \log^2 n)$.

Efficiency: $Eff(n) = \frac{O(n \log n)}{O(P(n)) \cdot O(n/P(n) \cdot \log n + \log^2 n)} = \frac{O(n \log n)}{O(n \cdot \log n + P(n) \cdot \log^2 n)}$

Is $O(1)$ for $P(n) \leq n / \log n$. 
Improved Algorithm for CREW

- Work with $P(n)$ processors ($P(n) \leq n$).
- Split the input in blocks of size $O(n/P(n))$.
- Sort parallel each block.
- Merge the blocks pairwise and parallel.

Complexity: $T(n) = O(n/P(n) \cdot \log n + \log^2 n)$.

Efficiency: $Eff(n) = \frac{O(n \log n)}{O(P(n)) \cdot O(n/P(n) \cdot \log n + \log^2 n)} = \frac{O(n \log n)}{O(n \cdot \log n + P(n) \cdot \log^2 n)}$

- Is $O(1)$ for $P(n) \leq n/ \log n$. 
Improved Algorithm for CREW

- Work with $P(n)$ processors ($P(n) \leq n$).
- Split the input in blocks of size $O(n/P(n))$.
- Sort parallel each block.
- Merge the blocks pairwise and parallel.

Complexity: $T(n) = O(n/P(n) \cdot \log n + \log^2 n)$.
Efficiency: $Eff(n) = \frac{O(n \log n)}{O(P(n)) \cdot O(n/P(n) \cdot \log n + \log^2 n)} = \frac{O(n \log n)}{O(n \cdot \log n + P(n) \cdot \log^2 n)}$

Is $O(1)$ for $P(n) \leq n / \log n$. 

Improved Algorithm for CREW

- Work with $P(n)$ processors ($P(n) \leq n$).
- Split the input in blocks of size $O(n/P(n))$.
- Sort parallel each block.
- Merge the blocks pairwise and parallel.

- Complexity: $T(n) = O(n/P(n) \cdot \log n + \log^2 n)$.
- Efficiency: $Eff(n) =$

\[
\frac{O(n \log n)}{O(P(n)) \cdot O(n/P(n) \cdot \log n + \log^2 n)} = \frac{O(n \log n)}{O(n \cdot \log n + P(n) \cdot \log^2 n)}
\]

- Is $O(1)$ for $P(n) \leq n/\log n$. 


Improved Algorithm for CREW

- Work with $P(n)$ processors ($P(n) \leq n$).
- Split the input in blocks of size $O(n/P(n))$. \(O(1)\)
- Sort parallel each block.
- Merge the blocks pairwise and parallel.

**Complexity:** \(T(n) = O(n/P(n) \cdot \log n + \log^2 n)\).

**Efficiency:** \(Eff(n) = \frac{O(n \log n)}{O(P(n)) \cdot O(n/P(n) \cdot \log n + \log^2 n)} = \frac{O(n \log n)}{O(n \cdot \log n + P(n) \cdot \log^2 n)}\)

- Is \(O(1)\) for $P(n) \leq n/\log n$. 
Improved Algorithm for CREW

- Work with $P(n)$ processors ($P(n) \leq n$).
- Split the input in blocks of size $O(n/P(n))$. $O(1)$
- Sort parallel each block. $O(n/P(n) \cdot \log(n/P(n)))$
- Merge the blocks pairwise and parallel.

- Complexity: $T(n) = O(n/P(n) \cdot \log n + \log^2 n)$.
- Efficiency: $Eff(n) =$

$$\frac{O(n \log n)}{O(P(n)) \cdot O(n/P(n) \cdot \log n + \log^2 n)} = \frac{O(n \log n)}{O(n \cdot \log n + P(n) \cdot \log^2 n)}$$

- Is $O(1)$ for $P(n) \leq n/\log n$. 

Improved Algorithm for CREW

- Work with $P(n)$ processors ($P(n) \leq n$).
- Split the input in blocks of size $O(n/P(n))$. $O(1)$
- Sort parallel each block. $O(n/P(n) \cdot \log(n/P(n)))$
- Merge the blocks pairwise and parallel. $O(n/P(n) + \log n) \cdot O(\log P(n))$

Complexity: $T(n) = O(n/P(n) \cdot \log n + \log^2 n)$.

Efficiency: $Eff(n) = \frac{O(n \log n)}{O(P(n)) \cdot O(n/P(n) \cdot \log n + \log^2 n)} = \frac{O(n \log n)}{O(n \cdot \log n + P(n) \cdot \log^2 n)}$

Is $O(1)$ for $P(n) \leq n/\log n$. 
Improved Algorithm for CREW

- Work with $P(n)$ processors ($P(n) \leq n$).
- Split the input in blocks of size $O(n/P(n))$. $O(1)$
- Sort parallel each block. $O(n/P(n) \cdot \log(n/P(n)))$
- Merge the blocks pairwise and parallel. $O(n/P(n) + \log n) \cdot O(\log P(n))$

**Complexity:** $T(n) = O(n/P(n) \cdot \log n + \log^2 n)$.

**Efficiency:** $Eff(n) = \frac{O(n \log n)}{O(P(n)) \cdot O(n/P(n) \cdot \log n + \log^2 n)} = \frac{O(n \log n)}{O(n \cdot \log n + P(n) \cdot \log^2 n)}$

Is $O(1)$ for $P(n) \leq n/ \log n$. 
Improved Algorithm for CREW

- Work with $P(n)$ processors ($P(n) \leq n$).
- Split the input in blocks of size $O(n/P(n))$. $O(1)$
- Sort parallel each block. $O(n/P(n) \cdot \log(n/P(n)))$
- Merge the blocks pairwise and parallel. $O(n/P(n) + \log n) \cdot O(\log P(n))$

Complexity: $T(n) = O(n/P(n) \cdot \log n + \log^2 n)$.

Efficiency: $Eff(n) =$

$$\frac{O(n \log n)}{O(P(n)) \cdot O(n/P(n) \cdot \log n + \log^2 n)} = \frac{O(n \log n)}{O(n \cdot \log n + P(n) \cdot \log^2 n)}$$

Is $O(1)$ for $P(n) \leq n/\log n$. 

Improved Algorithm for CREW

- Work with $P(n)$ processors ($P(n) \leq n$).
- Split the input in blocks of size $O(n/P(n))$. $O(1)$
- Sort parallel each block. $O(n/P(n) \cdot \log(n/P(n)))$
- Merge the blocks pairwise and parallel. $O(n/P(n) + \log n) \cdot O(\log P(n))$

Complexity: $T(n) = O(n/P(n) \cdot \log n + \log^2 n)$.
Efficiency: $Eff(n) = \frac{O(n \log n)}{O(P(n)) \cdot O(n/P(n) \cdot \log n + \log^2 n)} = \frac{O(n \log n)}{O(n \cdot \log n + P(n) \cdot \log^2 n)}$

- Is $O(1)$ for $P(n) \leq n/\log n$. 
Improved Algorithm EREW

- Exchange the merge algorithm.
- Recall $T_{\text{Merging}(EREW)}(n) = \log(O(n/P(n) + \log n \cdot \log P(n)))$.
- $T(n) = O(n/P(n) \cdot \log(n/P(n)) + O(n/P(n) \cdot \log P(n) + \log n \cdot \log^2 P(n))$.
- $T(n) = O((n/P(n) + \log^2 n) \cdot \log n)$.
- Efficiency:

$$\text{Eff}(n) = \frac{O(n \log n)}{O(P(n) \cdot ((n/P(n) + \log^2 n) \cdot \log n))}$$

- Is $O(1)$ if $P(n) < n/\log^2 n$. 
Improved Algorithm EREW

- Exchange the merge algorithm.
- Recall $T_{\text{Merging}(EREW)}(n) = \text{ls}O\left(\frac{n}{P(n)} + \log n \cdot \log P(n)\right)$.
- $T(n) = O\left(\frac{n}{P(n)} \cdot \log(n/P(n)) + \frac{n}{P(n)} \cdot \log P(n) + \log n \cdot \log^2 P(n)\right)$
- $T(n) = O\left(\left(\frac{n}{P(n)} + \log^2 n\right) \cdot \log n\right)$
- Efficiency:
  
  $$\text{Eff}(n) = \frac{O(n \log n)}{O(P(n) \cdot ((n/P(n) + \log^2 n) \cdot \log n))}$$

- Is $O(1)$ if $P(n) < n/\log^2 n$. 

Improved Algorithm EREW

- Exchange the merge algorithm.
- Recall $T_{\text{Merging(EREW)}}(n) = \Theta(n/P(n) + \log n \cdot \log P(n))$.
- $T(n) = O(n/P(n) \cdot \log(n/P(n)) + O(n/P(n) \cdot \log P(n) + \log n \cdot \log^2 P(n))$
- $T(n) = O((n/P(n) + \log^2 n) \cdot \log n)$
- Efficiency:

$$\text{Eff}(n) = \frac{O(n \log n)}{O(P(n) \cdot ((n/P(n) + \log^2 n) \cdot \log n))}$$

- $\Theta(1)$ if $P(n) < n/\log^2 n$.
Improved Algorithm EREW

- Exchange the merge algorithm.
- Recall $T_{\text{Merging}(\text{EREW})}(n) = \Omega(n/P(n) + \log n \cdot \log P(n))$.  
- $T(n) = O(n/P(n) \cdot \log(n/P(n)) + O(n/P(n) \cdot \log P(n) + \log n \cdot \log^2 P(n))$
- $T(n) = O((n/P(n) + \log^2 n) \cdot \log n)$
- Efficiency:

$$\text{Eff}(n) = \frac{O(n \log n)}{O(P(n) \cdot ((n/P(n) + \log^2 n) \cdot \log n))}$$

- Is $O(1)$ if $P(n) < n/\log^2 n$.  

Improved Algorithm EREW

- Exchange the merge algorithm.
- Recall $T_{\text{Merging}(\text{EREW})}(n) = \Omega(n/P(n) + \log n \cdot \log P(n))$.
- $T(n) = O(n/P(n) \cdot \log(n/P(n)) + O(n/P(n) \cdot \log P(n) + \log n \cdot \log^2 P(n))$
- $T(n) = O((n/P(n) + \log^2 n) \cdot \log n)$
- Efficiency:

$$Eff(n) = \frac{O(n \log n)}{O(P(n) \cdot ((n/P(n) + \log^2 n) \cdot \log n))}$$

- Is $O(1)$ if $P(n) < n/\log^2 n$. 

---

Sorting
Introduction to optimal Sorting
Algorithmn of Cole

Improved Algorithm EREW

- Exchange the merge algorithm.
- Recall $T_{\text{Merging(EREW)}}(n) = \Theta(n/P(n) + \log n \cdot \log P(n))$.
- $T(n) = O(n/P(n) \cdot \log(n/P(n))) + O(n/P(n) \cdot \log P(n) + \log n \cdot \log^2 P(n))$
- $T(n) = O((n/P(n) + \log^2 n) \cdot \log n)$
- Efficiency:
  
  $$\text{Eff}(n) = \frac{O(n \log n)}{O(P(n) \cdot ((n/P(n) + \log^2 n) \cdot \log n))}$$

- Is $O(1)$ if $P(n) < n/\log^2 n$. 


Lower Bound

Theorem:

For any parallel sorting algorithm \( Srt \) with \( P_{Srt}(n) = O(n) \) hold:

\[
T_{Srt}(n) = \Omega(\log(n)).
\]

Proof:

- Lower bound for sequential is \( \Theta(n \log n) \).
- One needs \( O(n \log n) \) comparisons.
- In each parallel step are at most \( o(n) \) comparisons possible.
- Thus with less steps we have a contradiction to the lower bound for sequential.

Situation at this point:

- Inefficient algorithms with: \( T(n) = O(\log n) \) and \( P(n) = n^2 \).
- Nearly efficient algorithm with: \( T(n) = O(\log^2 n) \) and \( P(n) = o(n) \).
Lower Bound

**Theorem:**
For any parallel sorting algorithm $Srt$ with $P_{Srt}(n) = O(n)$ hold:

$$T_{Srt}(n) = \Omega(\log(n)).$$

**Proof:**
- Lower bound for sequential is $\Theta(n \log n)$.
- One needs $O(n \log n)$ comparisons.
- In each parallel step are at most $o(n)$ comparisons possible.
- Thus with less steps we have a contradiction to the lower bound for sequential.

**Situation at this point:**
- Inefficient algorithms with: $T(n) = O(\log n)$ and $P(n) = n^2$.
- Nearly efficient algorithm with: $T(n) = O(\log^2 n)$ and $P(n) = o(n)$.
Lower Bound

Theorem:

For any parallel sorting algorithm $S_{rt}$ with $P_{S_{rt}}(n) = O(n)$ hold:

$$T_{S_{rt}}(n) = \Omega(\log(n)).$$

Proof:

- Lower bound for sequential is $\Theta(n \log n)$.
- One needs $O(n \log n)$ comparisons.
- In each parallel step are at most $o(n)$ comparisons possible.
- Thus with less steps we have a contradiction to the lower bound for sequential.

Situation at this point:

- Inefficient algorithms with: $T(n) = O(\log n)$ and $P(n) = n^2$.
- Nearly efficient algorithm with: $T(n) = O(\log^2 n)$ and $P(n) = o(n)$. 
**Theorem:**

For any parallel sorting algorithm $Srt$ with $P_{Srt}(n) = O(n)$ hold:

$$T_{Srt}(n) = \Omega(\log(n)).$$

**Proof:**

- Lower bound for sequential is $\Theta(n \log n)$.
- One needs $O(n \log n)$ comparisons.
- In each parallel step are at most $o(n)$ comparisons possible.
- Thus with less steps we have a contradiction to the lower bound for sequential.

**Situation at this point:**

- Inefficient algorithms with: $T(n) = O(\log n)$ and $P(n) = n^2$.
- Nearly efficient algorithm with: $T(n) = O(\log^2 n)$ and $P(n) = o(n)$. 
Lower Bound

**Theorem:**

For any parallel sorting algorithm $Srt$ with $P_{Srt}(n) = O(n)$ hold:

$$T_{Srt}(n) = \Omega(\log(n)).$$

**Proof:**

- Lower bound for sequential is $\Theta(n \log n)$.
- One needs $O(n \log n)$ comparisons.
- In each parallel step are at most $o(n)$ comparisons possible.
- Thus with less steps we have a contradiction to the lower bound for sequential.

**Situation at this point:**

- Inefficient algorithms with: $T(n) = O(\log n)$ and $P(n) = n^2$.
- Nearly efficient algorithm with: $T(n) = O(\log^2 n)$ and $P(n) = o(n)$. 
Theorem:
For any parallel sorting algorithm $Srt$ with $P_{Srt}(n) = O(n)$ hold:
$$T_{Srt}(n) = \Omega(\log(n)).$$

Proof:
- Lower bound for sequential is $\Theta(n \log n)$.
- One needs $O(n \log n)$ comparisons.
- In each parallel step are at most $o(n)$ comparisons possible.
- Thus with less steps we have a contradiction to the lower bound for sequential.

Situation at this point:
- Inefficient algorithms with: $T(n) = O(\log n)$ and $P(n) = n^2$.
- Nearly efficient algorithm with: $T(n) = O(\log^2 n)$ and $P(n) = o(n)$. 


Basic Operation for Sorting

- Identify basic operation for sorting.
- Assume: sorting key is $s_1, \cdots, s_n$.
- Program: compare_exchange($i, j$)
  
  \[
  \text{if } s_i > s_j \text{ then exchange } s_i \leftrightarrow s_j
  \]

- Symbolic view (Batcher):

\[
\begin{array}{c}
\text{max}(x, y) \\
\text{min}(x, y)
\end{array}
\]

- Basic building block for sorting networks.
- Base for Odd-Even merge
- Form this we build the optimal algorithm by Cole
Basic Operation for Sorting

- Identify basic operation for sorting.
- Assume: sorting key is $s_1, \cdots, s_n$.
- Program: compare_exchange($i, j$)
  
  if $s_i > s_j$ then exchange $s_i \leftrightarrow s_j$

- Symbolic view (Batcher):

  $$
  \begin{array}{ccc}
  y & \text{max}(x, y) \\
  \hline \\
  x & \text{min}(x, y)
  \end{array}
  $$

- Basic building block for sorting networks.
- Base for Odd-Even merge
- Form this we build the optimal algorithm by Cole
Basic Operation for Sorting

- Identify basic operation for sorting.
- Assume: sorting key is $s_1, \ldots, s_n$.
- Programm: compare_exchange(i,j)
  
  \[
  \text{if } s_i > s_j \text{ then exchange } s_i \leftrightarrow s_j
  \]

- Symbolic view (Batcher):

  \[
  \begin{array}{c}
  y \\
  \text{max}(x, y) \\
  \end{array}
  \]

  \[
  \begin{array}{c}
  x \\
  \text{min}(x, y) \\
  \end{array}
  \]

- Basic building block for sorting networks.
- Base for Odd-Even merge
- Form this we build the optimal algorithm by Cole
Basic Operation for Sorting

- Identify basic operation for sorting.
- Assume: sorting key is $s_1, \cdots, s_n$.
- Programm: compare_exchange$(i, j)$
  
  \[
  \text{if } s_i > s_j \text{ then exchange } s_i \leftrightarrow s_j
  \]

- Symbolic view (Batcher):

  \[
  y \quad \text{max}(x, y)
  \]
  \[
  x \quad \text{min}(x, y)
  \]

- Basic building block for sorting networks.
- Base for Odd-Even merge
- Form this we build the optimal algorithm by Cole
Basic Operation for Sorting

- Identify basic operation for sorting.
- Assume: sorting key is $s_1, \ldots, s_n$.
- **Programm**: `compare_exchange(i, j)`
  
  ```
  if $s_i > s_j$ then exchange $s_i \leftrightarrow s_j$
  ```

- **Symbolic view (Batcher)**:
  
  ```
  y  \------------------\ max(x, y)
  |                |
  x  \------------------\ min(x, y)
  ```

- Basic building block for sorting networks.
- Base for Odd-Even merge
- Form this we build the optimal algorithm by Cole
Basic Operation for Sorting

- Identify basic operation for sorting.
- Assume: sorting key is $s_1, \ldots, s_n$.
- **Programm:** `compare_exchange(i,j)
  
  if $s_i > s_j$ then exchange $s_i \leftrightarrow s_j$

- **Symbolic view (Batcher):**

  \[
  \begin{array}{c}
  y \\
  \hline
  x \\
  \end{array}
  \quad \max(x, y) \\
  \quad \min(x, y)
  \]

- **Basic building block for sorting networks.**
- Base for Odd-Even merge
- Form this we build the optimal algorithm by Cole
Basic Operation for Sorting

- Identify basic operation for sorting.
- Assume: sorting key is $s_1, \ldots, s_n$.
- **Programm:** `compare_exchange(i,j)
  if $s_i > s_j$ then exchange $s_i \leftrightarrow s_j$
- **Symbolic view (Batcher):**
  \[
  \begin{array}{c}
  y \\
  \hline
  \hline
  \end{array}
  \quad \max(x, y)
  \begin{array}{c}
  \hline
  \hline
  x
  \end{array}
  \quad \min(x, y)
  
- Basic building block for sorting networks.
- **Base for Odd-Even merge**
- Form this we build the optimal algorithm by Cole
Basic Operation for Sorting

- Identify basic operation for sorting.
- Assume: sorting key is $s_1, \ldots, s_n$.
- Programm: `compare_exchange(i,j)
  if $s_i > s_j$ then exchange $s_i \leftrightarrow s_j$

- Symbolic view (Batcher):
  $y \quad \text{max}(x, y)$
  $x \quad \text{min}(x, y)$

- Basic building block for sorting networks.
- Base for Odd-Even merge
- Form this we build the optimal algorithm by Cole
Odd-even Merge (Definition)

- **Input:** Sequence $S = (s_1, s_2, \cdots, s_n)$. (O.E.d.A. $n$ even)
- Let $Odd(S)$ [$Even(S)$] be the elements of $S$ with odd [even] index.
- Let $S' = (s'_1, s'_2, \cdots, s'_n)$ be a second sequence.
- Then we define: $interleave(S, S') = (s_1, s'_1, s_2, s'_2, \cdots, s_n, s'_n)$.

$$T_{interleave}(n) = O(1) \text{ mit } P_{interleave}(n) = O(n)$$
Odd-even Merge (Definition)

- Input: Sequence $S = (s_1, s_2, \cdots, s_n)$. (O.E.d.A. $n$ even)
- Let $Odd(S)$ [$Even(S)$] be the elements of $S$ with odd [even] index.
- Let $S' = (s_1', s_2', \cdots, s_n')$ be a second sequence.
- Then we define: $\text{interleave}(S, S') = (s_1, s_1', s_2, s_2', \cdots, s_n, s_n')$.

\[T_{\text{interleave}}(n) = O(1)\; \text{mit} \; P_{\text{interleave}}(n) = O(n)\]
Odd-even Merge (Definition)

- Input: Sequence $S = (s_1, s_2, \cdots, s_n)$. (O.E.d.A. $n$ even)
- Let $Odd(S)$ [$Even(S)$] be the elements of $S$ with odd [even] index.
- Let $S' = (s'_1, s'_2, \cdots, s'_n)$ be a second sequence.
- Then we define: $\text{interleave}(S, S') = (s_1, s'_1, s_2, s'_2, \cdots, s_n, s'_n)$.

- $T_{\text{interleave}}(n) = O(1)$ mit $P_{\text{interleave}}(n) = O(n)$
Odd-even Merge (Definition)

- **Input:** Sequence $S = (s_1, s_2, \ldots, s_n)$. (O.E.d.A. $n$ even)
- Let $Odd(S)$ [$Even(S)$] be the elements of $S$ with odd [even] index.
- Let $S' = (s'_1, s'_2, \ldots, s'_n)$ be a second sequence.
- Then we define: $\text{interleave}(S, S') = (s_1, s'_1, s_2, s'_2, \ldots, s_n, s'_n)$.

![Diagram](image_url)

- $T_{\text{interleave}}(n) = O(1)$ mit $P_{\text{interleave}}(n) = O(n)$
Odd-even Merge (Definition)

- Input: Sequence $S = (s_1, s_2, \ldots, s_n)$. (O.E.d.A. $n$ even)
- Let $Odd(S)$ [$Even(S)$] be the elements of $S$ with odd [even] index.
- Let $S' = (s'_1, s'_2, \ldots, s'_n)$ be a second sequence.
- Then we define: $\text{interleave}(S, S') = (s_1, s'_1, s_2, s'_2, \ldots, s_n, s'_n)$.

$T_{\text{interleave}}(n) = O(1)$ mit $P_{\text{interleave}}(n) = O(n)$
Odd-even Merge (Definition)

- Program: `odd_even(S)`
  
  `for all i` where `1 < i < n` and `i` even do in parallel
  
  `compare_exchange(i, i + 1)`.

- `T_{compare_exchange}(n) = O(1)` mit `P_{compare_exchange}(n) = O(n)`
Odd-even Merge (Definition)

- Programm: odd_even(S)
  for all $i$ where $1 < i < n$ and $i$ even do in parallel
  compare_exchange($i, i + 1$).

- $T_{\text{compare\_exchange}}(n) = O(1)$ mit $P_{\text{compare\_exchange}}(n) = O(n)$
Odd-even Merge (Definition)

- **Programm**: `odd_even(S)`
  
  ```plaintext```
  for all i where 1 < i < n and i even do in parallel
  
  `compare_exchange(i, i + 1)`.
  ```plaintext```

- $T_{\text{compare\_exchange}}(n) = O(1)$ mit $P_{\text{compare\_exchange}}(n) = O(n)$
Odd-even Merge (Definition)

Programm: $\text{join}_1(S, S')$

$\text{odd\_even(interleave}(S, S'))$
Odd-even Merge (Definition)

- Programm: $\text{join1}(S, S')$
  
  $\text{odd\_even}(\text{interleave}(S, S'))$

- $T_{\text{join1}}(n) = O(1)$ mit $P_{\text{join1}}(n) = O(n)$
Odd-even Merge (Definition)

- Programm: \( \text{join1}(S, S') \)
  \[
  \text{odd\_even}(\text{interleave}(S, S'))
  \]

- \( T_{\text{join1}}(n) = O(1) \) mit \( P_{\text{join1}}(n) = O(n) \)
Sorting with Merging

- Programm: `odd_even_merge(S, S')`
  
  ```
  if |S| = |S'| = 1 then merge with compare_exchange.
  S_odd = odd_even_merge(odd(S), odd(S')).
  S_even = odd_even_merge(even(S), even(S')).
  return join1(S_odd, S_even).
  ```

- \( T_{odd\_even\_merge}(n) = O(\log n) \) mit \( P_{odd\_even\_merge}(n) = O(n) \)

Theorem:

The algorithm `odd_even_merge` sorts two already sorted sequences into one.

Proof follows.
Sorting with Merging

- Programm: `odd_even_merge(S, S')`
  - if $|S| = |S'| = 1$ then merge with `compare_exchange`
  - $S_{odd} = odd_even_merge(odd(S), odd(S'))$
  - $S_{even} = odd_even_merge(even(S), even(S'))$
  - `return join1(S_{odd}, S_{even})`

- $T_{odd\_even\_merge}(n) = O(\log n)$ mit $P_{odd\_even\_merge}(n) = O(n)$

**Theorem:**

The algorithm `odd\_even\_merge` sorts two already sorted sequences into one.

Proof follows.
Sorting with Merging

- Programm: odd\_even\_merge(S, S')
  - if $|S| = |S'| = 1$ then merge with compare\_exchange.
  - $S_{odd} = odd\_even\_merge(odd(S), odd(S'))$.
  - $S_{even} = odd\_even\_merge(even(S), even(S'))$.
  - return $join1(S_{odd}, S_{even})$.

- $T_{odd\_even\_merge}(n) = O(\log n)$ mit $P_{odd\_even\_merge}(n) = O(n)$

Theorem:

The algorithm odd\_even\_merge sorts two already sorted sequences into one.

Proof follows.
Sorting with Merging

- **Programm:** $\text{odd\_even\_merge}(S, S')$
  - $\text{if } |S| = |S'| = 1 \text{ then merge with } \text{compare\_exchange}.$
  - $S_{\text{odd}} = \text{odd\_even\_merge}(\text{odd}(S), \text{odd}(S')).$
  - $S_{\text{even}} = \text{odd\_even\_merge}(\text{even}(S), \text{even}(S')).$
  - $\text{return } \text{join1}(S_{\text{odd}}, S_{\text{even}}).$

- $T_{\text{odd\_even\_merge}}(n) = O(\log n) \text{ mit } P_{\text{odd\_even\_merge}}(n) = O(n)$

**Theorem:**

The algorithm $\text{odd\_even\_merge}$ sorts two already sorted sequences into one.

Proof follows.
Sorting with Merging

- **Programm: odd_even_merge(S, S')**
  
  \[
  \text{if } |S| = |S'| = 1 \text{ then merge with compare_exchange.}
  \]

  \[
  S_{odd} = odd\_even\_merge(odd(S), odd(S')).
  \]

  \[
  S_{even} = odd\_even\_merge(even(S), even(S')).
  \]

  \[
  \text{return join1}(S_{odd}, S_{even}).
  \]

- \[
  T_{odd\_even\_merge}(n) = O(\log n) \text{ mit } P_{odd\_even\_merge}(n) = O(n)
  \]

**Theorem:**

The algorithm *odd_even_merge* sorts two already sorted sequences into one.

Proof follows.
Sorting Networks

Theorem:
There exists a sorting algorithm with $T(n) = O(\log^2 n)$ and $P(n) = n$.

Proof: use divide and conquer, and merging of depth $O(\log n)$.

Theorem:
There exists a sorting network of size $O(n \log^2 n)$.

Proof: All calls to `compare_exchange` operation are independent form the input (oblivious algorithm).
Sorting Networks

**Theorem:**
There exists a sorting algorithm with $T(n) = O(\log^2 n)$ and $P(n) = n$.

Proof: use divide and conquer, and merging of depth $O(\log n)$.

**Theorem:**
There exists a sorting network of size $O(n \log^2 n)$.

Proof: All calls to `compare_exchange` operation are independent from the input (oblivious algorithm).
**Theorem:**

There exists a sorting algorithm with $T(n) = O(\log^2 n)$ and $P(n) = n$.

Proof: use divide and conquer, and merging of depth $O(\log n)$.

**Theorem:**

There exists a sorting network of size $O(n \log^2 n)$.

Proof: All calls to `compare_exchange` operation are independent form the input (oblivious algorithm).
Sorting Networks

Theorem:
There exists a sorting algorithm with $T(n) = O(\log^2 n)$ and $P(n) = n$.

Proof: use divide and conquer, and merging of depth $O(\log n)$.

Theorem:
There exists a sorting network of size $O(n \log^2 n)$.

Proof: All calls to `compare_exchange` operation are independent from the input (oblivious algorithm).
The 0-1 Principle

**Theorem:**

If a sorting network $X$, resp. sorting algorithm is correct for all 0-1 inputs, then it is also correct for any input.

**Proof (by contradiction):**

- Let $f(x)$ be non-decreasing function: $f(s_i) \leq f(s_j) \iff s_i \leq s_j$.
- If $X$ sorts the sequence $(a_1, a_2, \cdots, a_n)$ to $(b_1, b_2, \cdots, b_n)$, then if $X$ gets $(f(a_1), f(a_2), \cdots, f(a_n))$ then the output $(f(b_1), f(b_2), \cdots, f(b_n))$ is also sorted.
- Assume $b_i > b_{i+1}$ and $f(b_i) \neq f(b_{i+1})$, then we have $f(b_i) > f(b_{i+1})$ in the “sorted” sequence $(f(b_1), f(b_2), \cdots, f(b_n))$. I.e errors may be kept under the function $f$.
- Choose now $f$: $f(b_j) = 0$ for $b_j < b_i$ and $f(b_j) = 1$ otherwise.
- Thus the sequence $(f(b_1), f(b_2), \cdots, f(b_n))$ is not sorted, because of $f(b_i) = 1$ and $f(b_{i+1}) = 0$.
- This is a contradiction.
The 0-1 Principle

If a sorting network $X$, resp. sorting algorithm is correct for all 0-1 inputs, then it is also correct for any input.

Proof (by contradiction):

- Let $f(x)$ be non-decreasing function: $f(s_i) \leq f(s_j) \iff s_i \leq s_j$.
- If $X$ sorts the sequence $(a_1, a_2, \cdots, a_n)$ to $(b_1, b_2, \cdots, b_n)$, then if $X$ gets $(f(a_1), f(a_2), \cdots, f(a_n))$ then the output $(f(b_1), f(b_2), \cdots, f(b_n))$ is also sorted.
- Assume $b_i > b_{i+1}$ and $f(b_i) \neq f(b_{i+1})$, then we have $f(b_i) > f(b_{i+1})$ in the “sorted” sequence $(f(b_1), f(b_2), \cdots, f(b_n))$. I.e errors may be kept under the function $f$.
- Choose now $f$: $f(b_j) = 0$ for $b_j < b_i$ and $f(b_j) = 1$ otherwise.
- Thus the sequence $(f(b_1), f(b_2), \cdots, f(b_n))$ is not sorted, because of $f(b_i) = 1$ and $f(b_{i+1}) = 0$.
- This is a contradiction.
The 0-1 Principle

**Theorem:**
If a sorting network $X$, resp. sorting algorithm is correct for all 0-1 inputs, then it is also correct for any input.

**Proof (by contradiction):**

- Let $f(x)$ be non-decreasing function: $f(s_i) \leq f(s_j) \iff s_i \leq s_j$.
- If $X$ sorts the sequence $(a_1, a_2, \ldots, a_n)$ to $(b_1, b_2, \ldots, b_n)$, then if $X$ gets $(f(a_1), f(a_2), \ldots, f(a_n))$ then the output $(f(b_1), f(b_2), \ldots, f(b_n))$ is also sorted.
- Assume $b_i > b_{i+1}$ and $f(b_i) \neq f(b_{i+1})$, then we have $f(b_i) > f(b_{i+1})$ in the “sorted” sequence $(f(b_1), f(b_2), \ldots, f(b_n))$. I.e errors may be kept under the function $f$.
- Choose now $f$: $f(b_j) = 0$ for $b_j < b_i$ and $f(b_j) = 1$ otherwise.
- Thus the sequence $(f(b_1), f(b_2), \ldots, f(b_n))$ is not sorted, because of $f(b_i) = 1$ and $f(b_{i+1}) = 0$.
- This is a contradiction.
The 0-1 Principle

**Theorem:**

If a sorting network $X$, resp. sorting algorithm is correct for all 0-1 inputs, then it is also correct for any input.

**Proof (by contradiction):**

- Let $f(x)$ be non-decreasing function: $f(s_i) \leq f(s_j) \iff s_i \leq s_j$.
- If $X$ sorts the sequence $(a_1, a_2, \cdots, a_n)$ to $(b_1, b_2, \cdots, b_n)$, then if $X$ gets $(f(a_1), f(a_2), \cdots, f(a_n))$ then the output $(f(b_1), f(b_2), \cdots, f(b_n))$ is also sorted.
- Assume $b_i > b_{i+1}$ and $f(b_i) \neq f(b_{i+1})$, then we have $f(b_i) > f(b_{i+1})$ in the “sorted” sequence $(f(b_1), f(b_2), \cdots, f(b_n))$. I.e errors may be kept under the function $f$.
- **Choose now** $f$: $f(b_j) = 0$ for $b_j < b_i$ and $f(b_j) = 1$ otherwise.
- Thus the sequence $(f(b_1), f(b_2), \cdots, f(b_n))$ is not sorted, because of $f(b_i) = 1$ and $f(b_{i+1}) = 0$.
- This is a contradiction.
The 0-1 Principle

**Theorem:**
If a sorting network $X$, resp. sorting algorithm is correct for all 0-1 inputs, then it is also correct for any input.

**Proof (by contradiction):**

- Let $f(x)$ be non-decreasing function: $f(s_i) \leq f(s_j) \Leftrightarrow s_i \leq s_j$.
- If $X$ sorts the sequence $(a_1, a_2, \cdots, a_n)$ to $(b_1, b_2, \cdots, b_n)$, then if $X$ gets $(f(a_1), f(a_2), \cdots, f(a_n))$ then the output $(f(b_1), f(b_2), \cdots, f(b_n))$ is also sorted.
- Assume $b_i > b_{i+1}$ and $f(b_i) \neq f(b_{i+1})$, then we have $f(b_i) > f(b_{i+1})$ in the “sorted” sequence $(f(b_1), f(b_2), \cdots, f(b_n))$. I.e errors may be kept under the function $f$.
- Choose now $f$: $f(b_j) = 0$ for $b_j < b_i$ and $f(b_j) = 1$ otherwise.
- Thus the sequence $(f(b_1), f(b_2), \cdots, f(b_n))$ is not sorted, because of $f(b_i) = 1$ and $f(b_{i+1}) = 0$.
- This is a contradiction.
The 0-1 Principle

Theorem:
If a sorting network $X$, resp. sorting algorithm is correct for all 0-1 inputs, then it is also correct for any input.

Proof (by contradiction):

- Let $f(x)$ be non-decreasing function: $f(s_i) \leq f(s_j) \iff s_i \leq s_j$.
- If $X$ sorts the sequence $(a_1, a_2, \ldots, a_n)$ to $(b_1, b_2, \ldots, b_n)$, then if $X$ gets $(f(a_1), f(a_2), \ldots, f(a_n))$ then the output $(f(b_1), f(b_2), \ldots, f(b_n))$ is also sorted.
- Assume $b_i > b_{i+1}$ and $f(b_i) \neq f(b_{i+1})$, then we have $f(b_i) > f(b_{i+1})$ in the “sorted” sequence $(f(b_1), f(b_2), \ldots, f(b_n))$. I.e errors may be kept under the function $f$.
- Choose now $f$: $f(b_j) = 0$ for $b_j < b_i$ and $f(b_j) = 1$ otherwise.
- Thus the sequence $(f(b_1), f(b_2), \ldots, f(b_n))$ is not sorted, because of $f(b_i) = 1$ and $f(b_{i+1}) = 0$.
- This is a contradiction.
Correctness of the Merging

**Theorem:**
The algorithm `odd_even_merge` sorts two sorted sequences into a single one.

**Proof:**

- $S$ has the form: $S = 0^p1^{m-p}$ for some $p$ with $0 \leq p \leq m$.
- $S'$ has the form: $S' = 0^q1^{m'-q}$ for some $q$ with $0 \leq q \leq m'$.
- Thus the sequence $S_{odd}$ has the form $0^\left\lceil \frac{p}{2} \right\rceil + \left\lfloor \frac{q}{2} \right\rfloor 1^*$.
- And $S_{even}$ has the form $0^\left\lfloor \frac{p}{2} \right\rfloor + \left\lceil \frac{q}{2} \right\rceil 1^*$.
- Definiere: $d = \left\lceil \frac{p}{2} \right\rceil + \left\lfloor \frac{q}{2} \right\rfloor - \left(\left\lfloor \frac{p}{2} \right\rfloor + \left\lceil \frac{q}{2} \right\rceil\right)$
- Depending on $d$ we consider three cases: $d = 0$, $d = 1$ and $d = 2$. 
Correctness of the Merging

Theorem:
The algorithm `odd_even_merge` sorts two sorted sequences into a singel one.

Proof:

- $S$ has the form: $S = 0^p1^{m-p}$ for some $p$ with $0 \leq p \leq m$.
- $S'$ has the form: $S' = 0^q1^{m'-q}$ for some $q$ with $0 \leq q \leq m'$.
- Thus the sequence $S_{odd}$ has the form $0^{\lceil p/2 \rceil + \lceil q/2 \rceil}1^*$.
- And $S_{even}$ has the form $0^{\lfloor p/2 \rfloor + \lfloor q/2 \rfloor}1^*$.
- Definiere: $d = \lceil p/2 \rceil + \lceil q/2 \rceil - (\lfloor p/2 \rfloor + \lfloor q/2 \rfloor)$
- Depending on $d$ we consider three cases: $d = 0$, $d = 1$ and $d = 2$. 
Correctness of the Merging

Theorem:
The algorithm `odd_even_merge` sorts two sorted sequences into a single one.

Proof:

- $S$ has the form: $S = 0^p1^{m-p}$ for some $p$ with $0 \leq p \leq m$.
- $S'$ has the form: $S' = 0^q1^{m'-q}$ for some $q$ with $0 \leq q \leq m'$.
- Thus the sequence $S_{odd}$ has the form $0^\lceil p/2 \rceil + \lfloor q/2 \rfloor 1^*$.
- And $S_{even}$ has the form $0^\lfloor p/2 \rfloor + \lceil q/2 \rceil 1^*$.
- Definiere: $d = \lceil p/2 \rceil + \lfloor q/2 \rfloor - (\lfloor p/2 \rfloor + \lceil q/2 \rceil)$.
- Depending on $d$ we consider three cases: $d = 0$, $d = 1$ and $d = 2$. 
Correctness of the Merging

Theorem:
The algorithm `odd_even_merge` sorts two sorted sequences into a single one.

Proof:
- $S$ has the form: $S = 0^p1^{m-p}$ for some $p$ with $0 \leq p \leq m$.
- $S'$ has the form: $S' = 0^q1^{m'-q}$ for some $q$ with $0 \leq q \leq m'$.
- Thus the sequence $S_{odd}$ has the form $0^{\lceil p/2 \rceil + \lceil q/2 \rceil}1^*$
- And $S_{even}$ has the form $0^{\lfloor p/2 \rfloor + \lfloor q/2 \rfloor}1^*$.
- Define: $d = \lceil p/2 \rceil + \lceil q/2 \rceil - (\lfloor p/2 \rfloor + \lfloor q/2 \rfloor)$
- Depending on $d$ we consider three cases: $d = 0$, $d = 1$ and $d = 2$. 
Correctness of the Merging

**Theorem:**
The algorithm `odd_even_merge` sorts two sorted sequences into a single one.

**Proof:**

- $S$ has the form: $S = 0^p1^{m-p}$ for some $p$ with $0 \leq p \leq m$.
- $S'$ has the form: $S' = 0^q1^{m'-q}$ for some $q$ with $0 \leq q \leq m'$.
- Thus the sequence $S_{odd}$ has the form $0^{\lfloor p/2 \rfloor + \lfloor q/2 \rfloor}1^*$
- And $S_{even}$ has the form $0^{\lceil p/2 \rceil + \lceil q/2 \rceil}1^*$.
- **Define:** $d = \lceil p/2 \rceil + \lceil q/2 \rceil - (\lfloor p/2 \rfloor + \lfloor q/2 \rfloor)$
- Depending on $d$ we consider three cases: $d = 0$, $d = 1$ and $d = 2$. 
Correctness of the Merging

Theorem:
The algorithm *odd_even_merge* sorts two sorted sequences into a single one.

Proof:

- $S$ has the form: $S = 0^p 1^{m-p}$ for some $p$ with $0 \leq p \leq m$.
- $S'$ has the form: $S' = 0^q 1^{m'-q}$ for some $q$ with $0 \leq q \leq m'$.
- Thus the sequence $S_{odd}$ has the form $0^{\lfloor p/2 \rfloor + \lfloor q/2 \rfloor} 1^*$.
- And $S_{even}$ has the form $0^{\lfloor p/2 \rfloor + \lfloor q/2 \rfloor} 1^*$.
- Define: $d = \lfloor p/2 \rfloor + \lfloor q/2 \rfloor - (\lfloor p/2 \rfloor + \lfloor q/2 \rfloor)$.
- Depending on $d$ we consider three cases: $d = 0$, $d = 1$ and $d = 2$. 
Correctness of the Merging

If $d = 0$: Then we have: $p$ and $q$ are even.
- The `interleave` step of `join1` has the form:

$$interleave(S_{odd}, S_{even}) = (00)^{(p+q)/2}1^{m+m'-p-q}$$
- The resulting sequences is already sorted.
- The `compare_exchange` step keeps the order.

If $d = 1$: Then we have: $p$ is odd and $q$ is even.
- The `interleave` step of `join1` has the form:

$$interleave(S_{odd}, S_{even}) = (00)^{(p+q)/2}01^{m+m'-p-q}$$
- The resulting sequences is already sorted.

If $d = 2$: Then we have: $p$ and $q$ are odd.
- The `interleave` step of `join1` has the form:

$$interleave(S_{odd}, S_{even}) = (00)^{(p+q)/2}101^{m+m'-p-q}$$
- The `compare_exchange` step will exchange the 1 on position $2r$ with the 0 on position $2r + 1$. 


Correctness of the Merging

If $d = 0$: Then we have: $p$ and $q$ are even.
- The *interleave* step of *join1* has the form:

$$\text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{(p+q)/2} 1^{m+m'-p-q}$$
- The resulting sequences is already sorted.
- The *compare_exchange* step keeps the order.

If $d = 1$: Then we have: $p$ is odd and $q$ is even.
- The *interleave* step of *join1* has the form:

$$\text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{\lfloor(p+q)/2\rfloor} 01^{m+m'-p-q}$$
- The resulting sequences is already sorted.

If $d = 2$: Then we have: $p$ and $q$ are odd.
- The *interleave* step of *join1* has the form:

$$\text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{\lfloor(p+q)/2\rfloor} 101^{m+m'-p-q}$$
- The *compare_exchange* step will exchange the 1 on position $2r$ with the 0 on position $2r + 1$. 
Correctness of the Merging

If $d = 0$: Then we have: $p$ and $q$ are even.

- The *interleave* step of *join1* has the form:
  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{(p+q)/2}1^{m+m'-p-q}
  \]

- The resulting sequences is already sorted.
- The *compare_exchange* step keeps the order.

If $d = 1$: Then we have: $p$ is odd and $q$ is even.

- The *interleave* step of *join1* has the form:
  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{(p+q)/2}01^{m+m'-p-q}
  \]

- The resulting sequences is already sorted.

If $d = 2$: Then we have: $p$ and $q$ are odd.

- The *interleave* step of *join1* has the form:
  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{(p+q)/2}101^{m+m'-p-q}
  \]

- The *compare_exchange* step will exchange the 1 on position $2r$ with the 0 on position $2r + 1$. 
Correctness of the Merging

If $d = 0$: Then we have: $p$ and $q$ are even.

- The *interleave* step of $join1$ has the form:

  $interleave(S_{odd}, S_{even}) = (00)^{(p+q)/2}1^{m+m'-p-q}$

- The resulting sequences is already sorted.
- The *compare_exchange* step keeps the order.

If $d = 1$: Then we have: $p$ is odd and $q$ is even.

- The *interleave* step of $join1$ has the form:

  $interleave(S_{odd}, S_{even}) = (00)^{\lfloor(p+q)/2\rfloor}01^{m+m'-p-q}$

- The resulting sequences is already sorted.

If $d = 2$: Then we have: $p$ and $q$ are odd.

- The *interleave* step of $join1$ has the form:

  $interleave(S_{odd}, S_{even}) = (00)^{\lfloor(p+q)/2\rfloor}101^{m+m'-p-q}$

- The *compare_exchange* step will exchange the 1 on position $2r$ with the 0 on position $2r + 1$. 
Correctness of the Merging

If $d = 0$: Then we have: $p$ and $q$ are even.

- The *interleave* step of *join1* has the form:

  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{(p+q)/2}1^{m+m'-p-q}
  \]

- The resulting sequences is already sorted.
- The *compare_exchange* step keeps the order.

If $d = 1$: Then we have: $p$ is odd and $q$ is even.

- The *interleave* step of *join1* has the form:

  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{\lfloor(p+q)/2\rfloor}01^{m+m'-p-q}
  \]

- The resulting sequences is already sorted.

If $d = 2$: Then we have: $p$ and $q$ are odd.

- The *interleave* step of *join1* has the form:

  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{\lfloor(p+q)/2\rfloor}101^{m+m'-p-q}
  \]

- The *compare_exchange* step will exchange the 1 on position $2r$ with the 0 on position $2r + 1$. 
Correctness of the Merging

If $d = 0$: Then we have: $p$ and $q$ are even.

- The *interleave* step of *join1* has the form:

  $$\text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{(p+q)/2}1^{m+m'-p-q}$$

- The resulting sequences is already sorted.
- The *compare_exchange* step keeps the order.

If $d = 1$: Then we have: $p$ is odd and $q$ is even.

- The *interleave* step of *join1* has the form:

  $$\text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{\lfloor(p+q)/2\rfloor}01^{m+m'-p-q}$$

- The resulting sequences is already sorted.

If $d = 2$: Then we have: $p$ and $q$ are odd.

- The *interleave* step of *join1* has the form:

  $$\text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{\lfloor(p+q)/2\rfloor}101^{m+m'-p-q}$$

- The *compare_exchange* step will exchange the 1 on position $2r$ with the 0 on position $2r + 1$. 
Correctness of the Merging

If \( d = 0 \): Then we have: \( p \) and \( q \) are even.

- The \textit{interleave} step of \textit{join1} has the form:
  
  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{(p+q)/2}1^{m+m'-p-q}
  \]

- The resulting sequences is already sorted.
- The \texttt{compare\_exchange} step keeps the order.

If \( d = 1 \): Then we have: \( p \) is odd and \( q \) is even.

- The \textit{interleave} step of \textit{join1} has the form:
  
  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{(p+q)/2}01^{m+m'-p-q}
  \]

- The resulting sequences is already sorted.

If \( d = 2 \): Then we have: \( p \) and \( q \) are odd.

- The \textit{interleave} step of \textit{join1} has the form:
  
  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{(p+q)/2}101^{m+m'-p-q}
  \]

- The \texttt{compare\_exchange} step will exchange the 1 on position \( 2r \) with the 0 on position \( 2r + 1 \).
Correctness of the Merging

If $d = 0$: Then we have: $p$ and $q$ are even.
- The *interleave* step of *join1* has the form:
  $$\text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{(p+q)/2} 1^{m+m'-p-q}$$
- The resulting sequences is already sorted.
- The *compare_exchange* step keeps the order.

If $d = 1$: Then we have: $p$ is odd and $q$ is even.
- The *interleave* step of *join1* has the form:
  $$\text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{\lfloor(p+q)/2\rfloor} 01^{m+m'-p-q}$$
- The resulting sequences is already sorted.

If $d = 2$: Then we have: $p$ and $q$ are odd.
- The *interleave* step of *join1* has the form:
  $$\text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{\lfloor(p+q)/2\rfloor} 01^{m+m'-p-q}$$
- The *compare_exchange* step will exchange the 1 on position $2r$ with the 0 on position $2r + 1$. 
Correctness of the Merging

If $d = 0$: Then we have: $p$ and $q$ are even.
- The *interleave* step of *join1* has the form:
  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{(p+q)/2}1^{m+m'-p-q}
  \]
- The resulting sequences is already sorted.
- The *compare_exchange* step keeps the order.

If $d = 1$: Then we have: $p$ is odd and $q$ is even.
- The *interleave* step of *join1* has the form:
  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{\lfloor(p+q)/2\rfloor}01^{m+m'-p-q}
  \]
- The resulting sequences is already sorted.

If $d = 2$: Then we have: $p$ and $q$ are odd.
- The *interleave* step of *join1* has the form:
  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{\lfloor(p+q)/2\rfloor}101^{m+m'-p-q}
  \]
- The *compare_exchange* step will exchange the 1 on position $2r$ with the 0 on position $2r + 1$. 
Correctness of the Merging

If \( d = 0 \): Then we have: \( p \) and \( q \) are even.
- The \textit{interleave} step of \textit{join}1 has the form:
  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{(p+q)/2}1^{m+m'-p-q}
  \]
- The resulting sequences is already sorted.
- The \textit{compare\_exchange} step keeps the order.

If \( d = 1 \): Then we have: \( p \) is odd and \( q \) is even.
- The \textit{interleave} step of \textit{join}1 has the form:
  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{\lfloor(p+q)/2\rfloor}01^{m+m'-p-q}
  \]
- The resulting sequences is already sorted.

If \( d = 2 \): Then we have: \( p \) and \( q \) are odd.
- The \textit{interleave} step of \textit{join}1 has the form:
  \[
  \text{interleave}(S_{\text{odd}}, S_{\text{even}}) = (00)^{\lfloor(p+q)/2\rfloor}101^{m+m'-p-q}
  \]
- The \textit{compare\_exchange} step will exchange the 1 on position \( 2r \) with the 0 on position \( 2r + 1 \).
The correctness of a merge network may be tested in time $O(n^2)$.  

Proof: Test all inputs of the form $(0^p 1^{m-p}, 0^q 1^{m'-q})$.

The test for correctness of a sorting network is NP-hard. 

Proof: Literature.
Testing the Correctness of a Network

Corollary:
The correctness of a merge network may be tested in time $O(n^2)$.

Proof: Test all inputs of the form $(0^p 1^{m-p}, 0^q 1^{m'-q})$.

Theorem:
The test for correctness of a sorting network is NP-hard.

Proof: Literature.
Corollary:
The correctness of a merge network may be tested in time $O(n^2)$.

Proof: Test all inputs of the form $(0^p 1^{m-p}, 0^q 1^{m'-q})$.

Theorem:
The test for correctness of a sorting network is NP-hard.

Proof: Literature.
Situation

- **Aim:** Fast optimal algorithm.

- So far $T(n) = \log^2 n$ bei $P(n) = O(n)$.

- So far: Two loop for merging and sorting.

- Idea: make one loop faster, i.e. the merging in $O(1)$.

- Problem: With no further information we need $\Theta(\log n)$ steps.

- Idea: compute this additional information during the sorting.

- Choose as additional information nice splitting points for merging.

- I.e choose positions which split the blocks to be merged of constants size.

- Problem: How to compute these points?

- Solution is the base for the algorithm of Cole.
Situation

- **Aim:** Fast optimal algorithm.
- **So far** $T(n) = \log^2 n$ bei $P(n) = O(n)$.
- **So far:** Two loop for merging and sorting.
- **Idea:** make one loop faster, i.e. the merging in $O(1)$.
- **Problem:** With no further information we need $\Theta(\log n)$ steps.
- **Idea:** compute this additional information during the sorting.
- **Choose** as additional information nice splitting points for merging.
- **I.e choose positions** which split the blocks to be merged of constants size.
- **Problem:** How to compute these points?
- **Solution** is the base for the algorithm of Cole.
Situation

- **Aim:** Fast optimal algorithm.
- **So far** $T(n) = \log^2 n$ bei $P(n) = O(n)$.
- **So far:** Two loop for merging and sorting.
- **Idea:** make one loop faster, i.e. the merging in $O(1)$.
- **Problem:** With no further information we need $\Theta(\log n)$ steps.
- **Idea:** compute this additional information during the sorting.
- **Choose as additional information nice splitting points for merging.**
  - i.e. choose positions which split the blocks to be merged of constants size.
- **Problem:** How to compute these points?
- **Solution is the base for the algorithm of Cole.**
Situation

- **Aim:** Fast optimal algorithm.
- **So far:** $T(n) = \log^2 n$ bei $P(n) = O(n)$.
- **So far:** Two loop for merging and sorting.
- **Idea:** make one loop faster, i.e. the merging in $O(1)$.
- **Problem:** With no further information we need $\Theta(\log n)$ steps.
- **Idea:** compute this additional information during the sorting.
- **Choose as additional information nice splitting points for merging.**
  - i.e choose positions which split the blocks to be merged of constants size.
  - **Problem:** How to compute these points?
- **Solution is the base for the algorithm of Cole.**
Situation

- **Aim**: Fast optimal algorihtm.
- So far $T(n) = \log^2 n$ bei $P(n) = O(n)$.
- So far: Two loop for merging and sorting.
- Idea: make one loop faster, i.e. the merging in $O(1)$.
- **Problem**: With no further information we need $\Theta(\log n)$ steps.
- Idea: compute this additional information during the sorting.
- Choose as additional information nice splitting points for merging.
- I.e choose positions which split the blocks to be merged of constants size.
- **Problem**: How to compute these points?
- Solution is the base for the algorithm of Cole.
Situation

- **Aim:** Fast optimal algorithm.
- **So far** \( T(n) = \log^2 n \) bei \( P(n) = O(n) \).
- **So far:** Two loop for merging and sorting.
- **Idea:** make one loop faster, i.e. the merging in \( O(1) \).
- **Problem:** With no further information we need \( \Theta(\log n) \) steps.
- **Idea:** compute this additional information during the sorting.
- Choose as additional information nice splitting points for merging.
- I.e choose positions which split the blocks to be merged of constants size.
- **Problem:** How to compute these points?
- **Solution** is the base for the algorithm of Cole.
Situation

- Aim: Fast optimal algorithm.
- So far $T(n) = \log^2 n$ bei $P(n) = O(n)$.
- So far: Two loop for merging and sorting.
- Idea: make one loop faster, i.e. the merging in $O(1)$.
- Problem: With no further information we need $\Theta(\log n)$ steps.
- Idea: compute this additional information during the sorting.
- Choose as additional information nice splitting points for merging.
- i.e choose positions which split the blocks to be merged of constants size.
- Problem: How to compute these points?
- Solution is the base for the algorithm of Cole.
Situation

- Aim: Fast optimal algorithm.
- So far \( T(n) = \log^2 n \) bei \( P(n) = O(n) \).
- So far: Two loop for merging and sorting.
- Idea: make one loop faster, i.e. the merging in \( O(1) \).
- Problem: With no further information we need \( \Theta(\log n) \) steps.
- Idea: compute this additional information during the sorting.
- Choose as additional information nice splitting points for merging.
- I.e choose positions which split the blocks to be merged of constants size.
- Problem: How to compute these points?
- Solution is the base for the algorithm of Cole.
Aim: Fast optimal algorithm.

So far $T(n) = \log^2 n$ bei $P(n) = O(n)$.

So far: Two loop for merging and sorting.

Idea: make one loop faster, i.e. the merging in $O(1)$.

Problem: With no further information we need $\Theta(\log n)$ steps.

Idea: compute this additional information during the sorting.

Choose as additional information nice splitting points for merging.

I.e choose positions which split the blocks to be merged of constants size.

Problem: How to compute these points?

Solution is the base for the algorithm of Cole.
Situation

- Aim: Fast optimal algorithm.
- So far $T(n) = \log^2 n$ bei $P(n) = O(n)$.
- So far: Two loop for merging and sorting.
- Idea: make one loop faster, i.e. the merging in $O(1)$.
- Problem: With no further information we need $\Theta(\log n)$ steps.
- Idea: compute this additional information during the sorting.
- Choose as additional information nice splitting points for merging.
- I.e choose positions which split the blocks to be merged of constants size.
- Problem: How to compute these points?
- Solution is the base for the algorithm of Cole.
The Merging-Tree, a View
The Merging-Tree, a View
The Merging-Tree, a View
The Merging-Tree, a View
The Merging-Tree, a View
The Merging-Tree, a View
The Merging-Tree, a View
The Merging-Tree, a View
The Merging-Tree, a View
The Merging-Tree, a View
The Merging-Tree, a View
The Merging-Tree, a View
The Merging-Tree, a View
The Merging-Tree, a View
The Merging-Tree, a View
Idea

- Before merging two sequences we will merge two sub-sequences.
- Choose as sub-sequence each $k$-th element of the original sequence.
- These sub-sequences will be used as crutch/support to do the final merging.
- I.e. these sub-sequences are used as a kind of “preview”.
- Using these crutch points we will be able to do the merging in $O(1)$ time.
- Total running time will be $O(\log n)$.
- The additional effort should be at most $O(1)$. 
**Idea**

- Before merging two sequences we will merge two sub-sequences.
- Choose as sub-sequence each $k$-th element of the original sequence.
- These sub-sequences will be used as crutch/support to do the final merging.
- I.e. these sub-sequences are used as a kind of “preview”.
- Using these crutch points we will be able to do the merging in $O(1)$ time.
- Total running time will be $O(\log n)$.
- The additional effort should be at most $O(1)$. 
Idea

- Before merging two sequences we will merge two sub-sequences.
- Choose as sub-sequence each \( k \)-th element of the original sequence.
- These sub-sequences will be used as crutch/support to do the final merging.
- I.e. these sub-sequences are used as a kind of “preview”.
- Using these crutch points we will be able to do the merging in \( O(1) \) time.
- Total running time will be \( O(\log n) \).
- The additional effort should be at most \( O(1) \).
Idea

- Before merging two sequences we will merge two sub-sequences.
- Choose as sub-sequence each $k$-th element of the original sequence.
- These sub-sequences will be used as crutch/support to do the final merging.
- I.e. these sub-sequences are used as a kind of “preview”.
- Using these crutch points we will be able to do the merging in $O(1)$ time.
- Total running time will be $O(\log n)$.
- The additional effort should be at most $O(1)$. 
Idea

- Before merging two sequences we will merge two sub-sequences.
- Choose as sub-sequence each $k$-th element of the original sequence.
- These sub-sequences will be used as crutch/support to do the final merging.
- I.e. these sub-sequences are used as a kind of “preview”.
- Using these crutch points we will be able to do the merging in $O(1)$ time.
- Total running time will be $O(\log n)$.
- The additional effort should be at most $O(1)$. 
Idea

- Before merging two sequences we will merge two sub-sequences.
- Choose as sub-sequence each $k$-th element of the original sequence.
- These sub-sequences will be used as crutch/support to do the final merging.
- I.e. these sub-sequences are used as a kind of “preview”.
- Using these crutch points we will be able to do the merging in $O(1)$ time.
- **Total running time will be $O(\log n)$**.
- The additional effort should be at most $O(1)$.
Idea

- Before merging two sequences we will merge two sub-sequences.
- Choose as sub-sequence each $k$-th element of the original sequence.
- These sub-sequences will be used as crutch/support to do the final merging.
- I.e. these sub-sequences are used as a kind of “preview”.
- Using these crutch points we will be able to do the merging in $O(1)$ time.
- Total running time will be $O(\log n)$.
- The additional effort should be at most $O(1)$. 
The Merging-Tree, a View

Each processor starts with 256 elements.
The Merging-Tree, a View

Each Processor starts with 256 elements.
The Merging-Tree, a View

Each Prozessor starts with 256 elements
The Merging-Tree, a View

Each Processor starts with 256 elements
The Merging-Tree, a View

Each Processor starts with 256 elements
The Merging-Tree, a View

Each Processor starts with 256 elements
The Merging-Tree, a View

Each Processor starts with 256 elements
The Merging-Tree, a View

Each Processor starts with 256 elements
The Merging-Tree, a View

Each Processor starts with 256 elements
The Merging-Tree, a View

Each processor starts with 256 elements.
The Merging-Tree, a View

Each Processor starts with 256 elements
The Merging-Tree, a View

Each Prozessor starts with 256 elements
The Merging-Tree, a View

Each Processor starts with 256 elements.

↑ each ↑


The Merging-Tree, a View
The Merging-Tree, a View

Each processor starts with 256 elements
The Merging-Tree, a View

Each Processor starts with 256 elements.
The Merging-Tree, a View

Each Processor starts with 256 elements

↑ each ↑
The Merging-Tree, a View

Each Prozessor starts with 256 elements
The Merging-Tree, a View

Each Prozessor starts with 256 elements
The Merging-Tree, a View

Each Processor starts with 256 elements
The Merging-Tree, a View

Each processor starts with 256 elements.
The Merging-Tree, a View

Each processor starts with 256 elements.
The Merging-Tree, a View

Each Prozessor starts with 256 elements
The Merging-Tree, a View

Each processor starts with 256 elements.
The Merging-Tree, a View

Each Prozessor starts with 256 elements
The Merging-Tree, a View

Each processor starts with 256 elements.
The Merging-Tree, a View

Each Processor starts with 256 elements.
Definition

- Let $J$ and $K$ be two sorted sequences.
- Note: without additional information we could not merge $J$ and $K$ in $O(1)$ time with $O(n)$ processors.
- Let $L$ be a third sequence, which will be called in the following good sampler for $J$ and $K$.
- Informal: $|L| < |J|$ and the elements of $L$ are evenly spread in $J$.
- Let $a < b$, $c$ is between $a$ and $b$ iff $a < c \leq b$.
- The rank of $e$ in $S$ is $\text{rng}(e, S) = |\{x \in S \mid x < e\}|$.
- Notation: $\text{Rng}_{A, B}$ is the function $\text{Rng}_{A, B} : A \mapsto \mathbb{N}^{\mid A\mid}$ with $\text{Rng}_{A, B}(e) = \text{rng}(e, B)$ for all $e \in A$.
- $\text{Rng}_{A, B}$ is called the rank between $A$ and $B$.
- Depending on the context $\text{Rng}_{A, B}$ could also be an array with $\mid A\mid$ elements.
Definition

- Let $J$ and $K$ be two sorted sequences.

- **Note:** without additional information we could not merge $J$ and $K$ in $O(1)$ time with $O(n)$ processors.

- Let $L$ be a third sequence, which will be called in the following good sampler for $J$ and $K$.

- Informal: $|L| < |J|$ and the elements of $L$ are evenly spread in $J$.

- Let $a < b$, $c$ is between $a$ and $b$ iff $a < c \leq b$.

- The rank of $e$ in $S$ is $rng(e, S) = |\{x \in S \mid x < e\}|$.

- **Notation:** $Rng_{A,B}$ is the function $Rng_{A,B} : A \mapsto \mathbb{N}^{|A|}$ with $Rng_{A,B}(e) = rng(e, B)$ for all $e \in A$.

- $Rng_{A,B}$ is called the rank between $A$ and $B$.

- Depending on the context $Rng_{A,B}$ could also be an array with $|A|$ elements.
Definition

- Let $J$ and $K$ be two sorted sequences.
- Note: without additional information we could not merge $J$ and $K$ in $O(1)$ time with $O(n)$ processors.
- Let $L$ be a third sequence, which will be called in the following **good sampler** for $J$ and $K$.
  - Informal: $|L| < |J|$ and the elements of $L$ are evenly spread in $J$.
  - Let $a < b$, $c$ is between $a$ and $b$ iff $a < c \leq b$.
  - The rank of $e$ in $S$ is $\text{rng}(e, S) = |\{x \in S \mid x < e\}|$.
  - Notation: $Rng_{A,B}$ is the function $Rng_{A,B} : A \mapsto \mathbb{N}^{|A|}$ with $Rng_{A,B}(e) = \text{rng}(e, B)$ for all $e \in A$.
  - $Rng_{A,B}$ is called the rank between $A$ and $B$.
  - Depending on the context $Rng_{A,B}$ could also be an array with $|A|$ elements.
Definition

- Let $J$ and $K$ be two sorted sequences.
- Note: without additional information we could not merge $J$ and $K$ in $O(1)$ time with $O(n)$ processors.
- Let $L$ be a third sequence, which will be called in the following good sampler for $J$ and $K$.
- Informal: $|L| < |J|$ and the elements of $L$ are evenly spread in $J$.
- Let $a < b$, $c$ is between $a$ and $b$ iff $a < c \leq b$.
- The rank of $e$ in $S$ is $rng(e, S) = |\{x \in S \mid x < e\}|$.
- Notation: $Rng_{A,B}$ is the function $Rng_{A,B} : A \mapsto \mathbb{N}^{|A|}$ with $Rng_{A,B}(e) = rng(e, B)$ for all $e \in A$.
- $Rng_{A,B}$ is called the rank between $A$ and $B$.
- Depending on the context $Rng_{A,B}$ could also be an array with $|A|$ elements.
Definition

- Let $J$ and $K$ be two sorted sequences.
- Note: without additional information we could not merge $J$ and $K$ in $O(1)$ time with $O(n)$ processors.
- Let $L$ be a third sequence, which will be called in the following good sampler for $J$ and $K$.
- Informal: $|L| < |J|$ and the elements of $L$ are evenly spread in $J$.
- Let $a < b$, $c$ is between $a$ and $b$ iff $a < c \leq b$.
- The rank of $e$ in $S$ is $\text{rng}(e, S) = |\{x \in S \mid x < e\}|$.
- Notation: $\text{Rng}_{A, B}$ is the function $\text{Rng}_{A, B} : A \mapsto \mathbb{N}^{|A|}$ with $\text{Rng}_{A, B}(e) = \text{rng}(e, B)$ for all $e \in A$.
- $\text{Rng}_{A, B}$ is called the rank between $A$ and $B$.
- Depending on the context $\text{Rng}_{A, B}$ could also be an array with $|A|$ elements.
Let $J$ and $K$ be two sorted sequences.

Note: without additional information we could not merge $J$ and $K$ in $O(1)$ time with $O(n)$ processors.

Let $L$ be a third sequence, which will be called in the following good sampler for $J$ and $K$.

Informal: $|L| < |J|$ and the elements of $L$ are evenly spread in $J$.

Let $a < b$, $c$ is between $a$ and $b$ iff $a < c \leq b$.

The rank of $e$ in $S$ is $\text{rng}(e, S) = |\{x \in S \mid x < e\}|$.

Notation: $\text{Rng}_{A, B}$ is the function $\text{Rng}_{A, B} : A \mapsto \mathbb{N}^{|A|}$ with $\text{Rng}_{A, B}(e) = \text{rng}(e, B)$ for all $e \in A$.

$\text{Rng}_{A, B}$ is called the rank between $A$ and $B$.

Depending on the context $\text{Rng}_{A, B}$ could also be an array with $|A|$ elements.
**Definition**

- Let $J$ and $K$ be two sorted sequences.
- Note: without additional information we could not merge $J$ and $K$ in $O(1)$ time with $O(n)$ processors.
- Let $L$ be a third sequence, which will be called in the following **good sampler** for $J$ and $K$.
- Informal: $|L| < |J|$ and the elements of $L$ are evenly spread in $J$.
- Let $a < b$, $c$ is between $a$ and $b$ iff $a < c \leq b$.
- The rank of $e$ in $S$ is $\text{rng}(e, S) = |\{x \in S \mid x < e\}|$.
- **Notation:** $\text{Rng}_{A,B}$ is the function $\text{Rng}_{A,B} : A \mapsto \mathbb{N}^{|A|}$ with $\text{Rng}_{A,B}(e) = \text{rng}(e, B)$ for all $e \in A$.
- $\text{Rng}_{A,B}$ is called the rank between $A$ and $B$.
- Depending on the context $\text{Rng}_{A,B}$ could also be an array with $|A|$ elements.
Definition

- Let $J$ and $K$ be two sorted sequences.
- Note: without additional information we could not merge $J$ and $K$ in $O(1)$ time with $O(n)$ processors.
- Let $L$ be a third sequence, which will be called in the following good sampler for $J$ and $K$.
- Informal: $|L| < |J|$ and the elements of $L$ are evenly spread in $J$.
- Let $a < b$, $c$ is between $a$ and $b$ iff $a < c \leq b$.
- The rank of $e$ in $S$ is $\text{rng}(e, S) = |\{x \in S \mid x < e\}|$.
- Notation: $Rng_{A, B}$ is the function $Rng_{A, B} : A \mapsto \mathbb{N}^{|A|}$ with $Rng_{A, B}(e) = \text{rng}(e, B)$ for all $e \in A$.
- $Rng_{A, B}$ is called the rank between $A$ and $B$.
- Depending on the context $Rng_{A, B}$ could also be an array with $|A|$ elements.
Definition

- Let $J$ and $K$ be two sorted sequences.
- Note: without additional information we could not merge $J$ and $K$ in $O(1)$ time with $O(n)$ processors.
- Let $L$ be a third sequence, which will be called in the following good sampler for $J$ and $K$.
- Informal: $|L| < |J|$ and the elements of $L$ are evenly spread in $J$.
- Let $a < b$, $c$ is between $a$ and $b$ iff $a < c \leq b$.
- The rank of $e$ in $S$ is $\text{rng}(e, S) = |\{x \in S \mid x < e\}|$.
- Notation: $Rng_{A,B}$ is the function $Rng_{A,B} : A \mapsto \mathbb{N}^{|A|}$ with $Rng_{A,B}(e) = \text{rng}(e, B)$ for all $e \in A$.
- $Rng_{A,B}$ is called the rank between $A$ and $B$.
- Depending on the context $Rng_{A,B}$ could also be an array with $|A|$ elements.
Good Sampler

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad \text{Rng}_{A,B} : A \mapsto \mathbb{N}^{|A|} \quad \text{with} \quad \text{Rng}_{A,B}(e) = \text{rng}(e, B) \]

**Definition:**

We call \( L \) a good sampler of \( J \), iff:

- \( L \) and \( J \) are sorted.
- Between any \( k + 1 \) succeeding elements of \( \{-\infty\} \cup L \cup \{+\infty\} \) are at most \( 2 \cdot k + 1 \) many elements in \( J \).

**Example:**

- Let \( S \) be a sorted sequence
- Let \( S_1 \) be the sequence consisting of each fourth element of \( S \).
- Then \( S_1 \) is a good sampler of \( S \).
- Let \( S_2 \) be the sequence consisting of each second element of \( S \).
- Then \( S_1 \) is a good sampler of \( S_2 \).
- Example \((k = 1)\): 1, 2, 3, 4.
- Example \((k = 3)\): 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
Good Sampler

Definition:

We call $L$ a good sampler of $J$, iff:

- $L$ and $J$ are sorted.
- Between any $k + 1$ succeeding elements of $\{-\infty\} \cup L \cup \{+\infty\}$ are at most $2 \cdot k + 1$ many elements in $J$.

Example:

- Let $S$ be a sorted sequence
- Let $S_1$ be the sequence consisting of each forth element of $S$.
- Then $S_1$ is a good sampler of $S$.
- Let $S_2$ be the sequence consisting of each second element of $S$.
- Then $S_1$ is a good sampler of $S_2$.
- Example ($k = 1$): 1, 2, 3, 4.
- Example ($k = 3$): 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
Good Sampler

\[ rng(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad Rng_{A,B} : A \mapsto \mathbb{N}^{|A|} \quad \text{with} \quad Rng_{A,B}(e) = rng(e, B) \]

**Definition:**

We call \( L \) a good sampler of \( J \), iff:

- \( L \) and \( J \) are sorted.
- Between any \( k + 1 \) succeeding elements of \( \{-\infty\} \cup L \cup \{+\infty\} \) are at most \( 2 \cdot k + 1 \) many elements in \( J \).

**Example:**

- Let \( S \) be a sorted sequence
- Let \( S_1 \) be the sequence consisting of each forth element of \( S \).
- Then \( S_1 \) is a good sampler of \( S \).
- Let \( S_2 \) be the sequence consisting of each second element of \( S \).
- Then \( S_1 \) is a good sampler of \( S_2 \).
- Example \((k = 1)\): 1, 2, 3, 4.
- Example \((k = 3)\): 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
Good Sampler

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \text{ and } \text{Rng}_{A,B} : A \mapsto \mathbb{N}^{|A|} \text{ with } \text{Rng}_{A,B}(e) = \text{rng}(e, B) \]

**Definition:**

We call \( L \) a good sampler of \( J \), iff:

- \( L \) and \( J \) are sorted.
- Between any \( k + 1 \) succeeding elements of \( \{-\infty\} \cup L \cup \{+\infty\} \) are at most \( 2 \cdot k + 1 \) many elements in \( J \).

**Example:**

- Let \( S \) be a sorted sequence
- Let \( S_1 \) be the sequence consisting of each forth element of \( S \).
- Then \( S_1 \) is a good sampler of \( S \).
- Let \( S_2 \) be the sequence consisting of each second element of \( S \).
- Then \( S_1 \) is a good sampler of \( S_2 \).
- Example \((k = 1)\): 1, 2, 3, 4.
- Example \((k = 3)\): 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
Good Sampler

\[ \text{rng}(e, S) = |\{ x \in S \mid x < e \}| \quad \text{and} \quad \text{Rng}_{A, B} : A \mapsto \mathbb{N}^{|A|} \quad \text{with} \quad \text{Rng}_{A, B}(e) = \text{rng}(e, B) \]

**Definition:**
We call \( L \) a good sampler of \( J \), iff:

- \( L \) and \( J \) are sorted.
- Between any \( k + 1 \) succeeding elements of \( \{-\infty\} \cup L \cup \{+\infty\} \) are at most \( 2 \cdot k + 1 \) many elements in \( J \).

**Example:**
- Let \( S \) be a sorted sequence.
- Let \( S_1 \) be the sequence consisting of each fourth element of \( S \).
- Then \( S_1 \) is a good sampler of \( S \).
- Let \( S_2 \) be the sequence consisting of each second element of \( S \).
- Then \( S_1 \) is a good sampler of \( S_2 \).
- Example (\( k = 1 \)): 1, 2, 3, 4.
- Example (\( k = 3 \)): 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
Good Sampler

Definition:
We call $L$ a good sampler of $J$, iff:
- $L$ and $J$ are sorted.
- Between any $k + 1$ succeeding elements of $\{-\infty\} \cup L \cup \{+\infty\}$ are at most $2 \cdot k + 1$ many elements in $J$.

Example:
- Let $S$ be a sorted sequence
  - Let $S_1$ be the sequence consisting of each forth element of $S$.
  - Then $S_1$ is a good sampler of $S$.
- Let $S_2$ be the sequence consisting of each second element of $S$.
  - Then $S_1$ is a good sampler of $S_2$.
- Example ($k = 1$): $1, 2, 3, 4$.
- Example ($k = 3$): $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$.
Good Sampler

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \text{ and } R_{A,B} : A \mapsto \mathbb{N}^{\lvert A\rvert} \text{ with } R_{A,B}(e) = \text{rng}(e, B) \]

Definition:
We call \( L \) a good sampler of \( J \), iff:

- \( L \) and \( J \) are sorted.
- Between any \( k + 1 \) succeeding elements of \( \{-\infty\} \cup L \cup \{+\infty\} \) are at most \( 2 \cdot k + 1 \) many elements in \( J \).

Example:
- Let \( S \) be a sorted sequence
- Let \( S_1 \) be the sequence consisting of each fourth element of \( S \).
- Then \( S_1 \) is a good sampler of \( S \).
- Let \( S_2 \) be the sequence consisting of each second element of \( S \).
- Then \( S_1 \) is a good sampler of \( S_2 \).
- Example \((k = 1)\): 1, 2, 3, 4.
- Example \((k = 3)\): 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
Good Sampler

\[ rng(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad Rng_{A,B} : A \mapsto \mathbb{N}^{|A|} \quad \text{with} \quad Rng_{A,B}(e) = rng(e, B) \]

**Definition:**

We call \( L \) a good sampler of \( J \), iff:

- \( L \) and \( J \) are sorted.
- Between any \( k + 1 \) succeeding elements of \( \{-\infty\} \cup L \cup \{+\infty\} \) are at most \( 2 \cdot k + 1 \) many elements in \( J \).

**Example:**

- Let \( S \) be a sorted sequence
- Let \( S_1 \) be the sequence consisting of each forth element of \( S \).
- Then \( S_1 \) is a good sampler of \( S \).
- Let \( S_2 \) be the sequence consisting of each second element of \( S \).
- **Then** \( S_1 \) **is a good sampler of** \( S_2 \).
- Example \((k = 1)\): \( 1, 2, 3, 4 \).
- Example \((k = 3)\): \( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \).
Good Sampler

Definition:

We call $L$ a good sampler of $J$, iff:

- $L$ and $J$ are sorted.
- Between any $k + 1$ succeeding elements of $\{-\infty\} \cup L \cup \{+\infty\}$ are at most $2 \cdot k + 1$ many elements in $J$.

Example:

- Let $S$ be a sorted sequence
- Let $S_1$ be the sequence consisting of each forth element of $S$.
- Then $S_1$ is a good sampler of $S$.
- Let $S_2$ be the sequence consisting of each second element of $S$.
- Then $S_1$ is a good sampler of $S_2$.

Example ($k = 1$): $1, 2, 3, 4$.

Example ($k = 3$): $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$. 

$$rng(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad Rng_{A,B} : A \mapsto \mathbb{N}|^A| \quad \text{with} \quad Rng_{A,B}(e) = rng(e, B)$$
Good Sampler

\[ \text{rng}(e, S) = \left|\{x \in S \mid x < e\}\right| \quad \text{and} \quad R_{A,B} : A \mapsto \mathbb{N}^{|A|} \quad \text{with} \quad R_{A,B}(e) = \text{rng}(e, B) \]

**Definition:**

We call \( L \) a good sampler of \( J \), iff:

- \( L \) and \( J \) are sorted.
- Between any \( k + 1 \) succeeding elements of \( \{-\infty\} \cup L \cup \{+\infty\} \) are at most \( 2 \cdot k + 1 \) many elements in \( J \).

**Example:**

- Let \( S \) be a sorted sequence
- Let \( S_1 \) be the sequence consisting of each forth element of \( S \).
- Then \( S_1 \) is a good sampler of \( S \).
- Let \( S_2 \) be the sequence consisting of each second element of \( S \).
- Then \( S_1 \) is a good sampler of \( S_2 \).
- Example \((k = 1)\): 1, 2, 3, 4.
- Example \((k = 3)\): 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
Merging using a Good Sampler

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad Rng_{A,B} : A \mapsto \mathbb{N}^{\lfloor |A| \rfloor} \quad \text{with} \quad Rng_{A,B}(e) = \text{rng}(e, B) \]

- Let \( J, K \) and \( L \) be sorted sequences.
- Let \( L \) be a good sampler of both \( J \) and \( K \).
- Let \( L = (l_1, l_2, \cdots, l_s) \).
- Programm: `merge_with_help(J, K, L)`
  
  for all \( i \) where \( 1 \leq i \leq s \) do in parallel
  
  Assign \( J_i = \{x \in J \mid l_{i-1} < x \leq l_i\} \).
  
  Assign \( K_i = \{x \in K \mid l_{i-1} < x \leq l_i\} \).
  
  Assign \( res_i = \text{merge}(J_i, K_i) \).
  
  return \((res_1, res_2, \cdots, res_s)\).

- Situation:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L₁</td>
<td>L₂</td>
<td>L₃</td>
<td>L₄</td>
<td>L₅</td>
<td>L₆</td>
<td>L₇</td>
<td>L₈</td>
</tr>
<tr>
<td>l₁</td>
<td></td>
<td>l₂</td>
<td>l₃</td>
<td>l₄</td>
<td>l₅</td>
<td>l₆</td>
<td>l₇</td>
<td>l₈</td>
</tr>
<tr>
<td>K₁</td>
<td>K₂</td>
<td>K₃</td>
<td>K₄</td>
<td>K₅</td>
<td>K₆</td>
<td>K₇</td>
<td>K₈</td>
<td>K₉</td>
</tr>
</tbody>
</table>
Merging using a Good Sampler

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \text{ and } \text{Rng}_{A,B} : A \mapsto \mathbb{N}^{|A|} \text{ with } \text{Rng}_{A,B}(e) = \text{rng}(e, B) \]

- Let \( J, K \) and \( L \) be sorted sequences.
- Let \( L \) be a good sampler of both \( J \) and \( K \).
- Let \( L = (l_1, l_2, \ldots, l_s) \).
- Program: \text{merge\_with\_help}(J, K, L)
  
  for all \( i \) where \( 1 \leq i \leq s \) do in parallel
  
  Assign \( J_i = \{x \in J \mid l_{i-1} < x \leq l_i\} \).
  
  Assign \( K_i = \{x \in K \mid l_{i-1} < x \leq l_i\} \).
  
  Assign \( \text{res}_i = \text{merge}(J_i, K_i) \).

return \( (\text{res}_1, \text{res}_2, \ldots, \text{res}_s) \).

Situation:

<table>
<thead>
<tr>
<th>( L_1 )</th>
<th>( L_2 )</th>
<th>( L_3 )</th>
<th>( L_4 )</th>
<th>( L_5 )</th>
<th>( L_6 )</th>
<th>( L_7 )</th>
<th>( L_8 )</th>
<th>( L_9 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l_1 )</td>
<td>( l_2 )</td>
<td>( l_3 )</td>
<td>( l_4 )</td>
<td>( l_5 )</td>
<td>( l_6 )</td>
<td>( l_7 )</td>
<td>( l_8 )</td>
<td></td>
</tr>
<tr>
<td>( K_1 )</td>
<td>( K_2 )</td>
<td>( K_3 )</td>
<td>( K_4 )</td>
<td>( K_5 )</td>
<td>( K_6 )</td>
<td>( K_7 )</td>
<td>( K_8 )</td>
<td>( K_9 )</td>
</tr>
</tbody>
</table>
Merging using a Good Sampler

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \text{ and } R_{A,B} : A \mapsto \mathbb{N}^{|A|} \text{ with } R_{A,B}(e) = \text{rng}(e, B) \]

- Let \( J, K \) and \( L \) be sorted sequences.
- Let \( L \) be a good sampler of both \( J \) and \( K \).
- Let \( L = (l_1, l_2, \ldots, l_s) \).
- Program: merge_with_help(\( J, K, L \))
  for all \( i \) where \( 1 \leq i \leq s \) do in parallel
    Assign \( J_i = \{x \in J \mid l_{i-1} < x \leq l_i\} \).
    Assign \( K_i = \{x \in K \mid l_{i-1} < x \leq l_i\} \).
    Assign \( \text{res}_i = \text{merge}(J_i, K_i) \).
  return \( (\text{res}_1, \text{res}_2, \ldots, \text{res}_s) \).
- Situation:

\[
\begin{array}{cccccccc}
L_1 & L_2 & L_3 & L_4 & L_5 & L_6 & L_7 & L_8 & L_9 \\
| & | & | & | & | & | & | & |
| l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & |
| K_1 & K_2 & K_3 & K_4 & K_5 & K_6 & K_7 & K_8 & K_9 |
\end{array}
\]
Merging using a Good Sampler

\[\operatorname{rng}(e, S) = \left| \{ x \in S \mid x < e \} \right| \text{ and } \operatorname{Rng}_{A,B} : A \mapsto \mathbb{N}^{\left| A \right|} \text{ with } \operatorname{Rng}_{A,B}(e) = \operatorname{rng}(e, B)\]

- Let \( J \), \( K \) and \( L \) be sorted sequences.
- Let \( L \) be a good sampler of both \( J \) and \( K \).
- Let \( L = (l_1, l_2, \cdots, l_s) \).

Program: \text{merge\_with\_help}(J, K, L)

\begin{align*}
\text{for all } i \text{ where } 1 \leq i \leq s \text{ do in parallel} \\
&\text{Assign } J_i = \{ x \in J \mid l_{i-1} < x \leq l_i \}. \\
&\text{Assign } K_i = \{ x \in K \mid l_{i-1} < x \leq l_i \}. \\
&\text{Assign } \text{res}_i = \text{merge}(J_i, K_i). \\
\end{align*}

\text{return } (\text{res}_1, \text{res}_2, \cdots, \text{res}_s).

Situation:

\[
\begin{array}{cccccccccc}
L_1 & L_2 & L_3 & L_4 & L_5 & L_6 & L_7 & L_8 & L_9 \\
\hline
l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 & \text{ } \\
K_1 & K_2 & K_3 & K_4 & K_5 & K_6 & K_7 & K_8 & K_9
\end{array}
\]
Merging using a Good Sampler

\[ \text{rng}(e, S) = |\{ x \in S \mid x < e \}| \text{ and } \text{Rng}_{A,B} : A \mapsto \mathbb{N}^{|A|} \text{ with } \text{Rng}_{A,B}(e) = \text{rng}(e, B) \]

- Let \( J, K \) and \( L \) be sorted sequences.
- Let \( L \) be a good sampler of both \( J \) and \( K \).
- Let \( L = (l_1, l_2, \cdots, l_s) \).
- Programm: \( \text{merge\_with\_help}(J, K, L) \)
  
  for all \( i \) where \( 1 \leq i \leq s \) do in parallel
  
  Assign \( J_i = \{ x \in J \mid l_{i-1} < x \leq l_i \} \).
  Assign \( K_i = \{ x \in K \mid l_{i-1} < x \leq l_i \} \).
  Assign \( \text{res}_i = \text{merge}(J_i, K_i) \).

 return \( (\text{res}_1, \text{res}_2, \cdots, \text{res}_s) \).

- Situation:

<table>
<thead>
<tr>
<th></th>
<th>( L_1 )</th>
<th>( L_2 )</th>
<th>( L_3 )</th>
<th>( L_4 )</th>
<th>( L_5 )</th>
<th>( L_6 )</th>
<th>( L_7 )</th>
<th>( L_8 )</th>
<th>( L_9 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( l_1 )</td>
<td>( l_2 )</td>
<td>( l_3 )</td>
<td>( l_4 )</td>
<td>( l_5 )</td>
<td>( l_6 )</td>
<td>( l_7 )</td>
<td>( l_8 )</td>
<td>( l_9 )</td>
</tr>
<tr>
<td>( K_1 )</td>
<td>( K_2 )</td>
<td>( K_3 )</td>
<td>( K_4 )</td>
<td>( K_5 )</td>
<td>( K_6 )</td>
<td>( K_7 )</td>
<td>( K_8 )</td>
<td>( K_9 )</td>
<td></td>
</tr>
</tbody>
</table>
Merging using a Good Sampler

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad \text{Rng}_{A,B} : A \mapsto \mathbb{N}^{\left|A\right|} \quad \text{with} \quad \text{Rng}_{A,B}(e) = \text{rng}(e, B) \]

- Let \( J, K \) and \( L \) be sorted sequences.
- Let \( L \) be a good sampler of both \( J \) and \( K \).
- Let \( L = (l_1, l_2, \cdots, l_s) \).
- Program: merge_with_help\( (J, K, L) \)
  \[ \text{for all } i \text{ where } 1 \leq i \leq s \text{ do in parallel} \]
  - Assign \( J_i = \{ x \in J \mid l_{i-1} < x \leq l_i \} \).
  - Assign \( K_i = \{ x \in K \mid l_{i-1} < x \leq l_i \} \).
  - Assign \( \text{res}_i = \text{merge}(J_i, K_i) \).
  \[ \text{return } (\text{res}_1, \text{res}_2, \cdots, \text{res}_s). \]

- Situation:

\[
\begin{array}{cccccccccc}
L_1 & L_2 & L_3 & L_4 & L_5 & L_6 & L_7 & L_8 & L_9 \\
\hline
l_1 & l_2 & l_3 & l_4 & l_5 & l_6 & l_7 & l_8 \\
K_1 & K_2 & K_3 & K_4 & K_5 & K_6 & K_7 & K_8 & K_9
\end{array}
\]
**Introduction to optimal Sorting Algorithmn of Cole**

**Idea**

**Merging using a Good Sampler (Example)**

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad Rng_{A,B} : A \mapsto \mathbb{N}^{|A|} \ \text{with} \ Rng_{A,B}(e) = \text{rng}(e, B) \]

- \( K = (1, 4, 6, 9, 11, 12, 13, 16, 19, 20) \)
- \( J = (2, 3, 7, 8, 10, 14, 15, 17, 18, 21) \)
- \( L = (5, 10, 12, 17) \)

Then we have:

<table>
<thead>
<tr>
<th>( i )</th>
<th>( K_i )</th>
<th>( J_i )</th>
<th>merge(( K_i, J_i ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Result: \((1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21)\)
Merging using a Good Sampler (Example)

\[ rng(e, S) = |\{ x \in S \mid x < e \} | \text{ and } Rng_{A,B} : A \mapsto \mathbb{N}^{|A|} \text{ with } Rng_{A,B}(e) = rng(e, B) \]

- \( K = (1, 4, 6, 9, 11, 12, 13, 16, 19, 20) \)
- \( J = (2, 3, 7, 8, 10, 14, 15, 17, 18, 21) \)
- \( L = (5, 10, 12, 17) \)

Then we have:

<table>
<thead>
<tr>
<th>( i )</th>
<th>( K_i )</th>
<th>( J_i )</th>
<th>merge(( K_i, J_i ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Result: \((1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21)\)
Merging using a Good Sampler (Example)

\[ \text{rng}(e, S) = |\{ x \in S \mid x < e \}| \quad \text{and} \quad Rng_{A,B} : A \mapsto \mathbb{N}^{|A|} \quad \text{with} \quad Rng_{A,B}(e) = \text{rng}(e, B) \]

- \( K = (1, 4, 6, 9, 11, 12, 13, 16, 19, 20) \)
- \( J = (2, 3, 7, 8, 10, 14, 15, 17, 18, 21) \)
- \( L = (5, 10, 12, 17) \)

Then we have:

<table>
<thead>
<tr>
<th>i</th>
<th>( K_i )</th>
<th>( J_i )</th>
<th>( \text{merge}(K_i, J_i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Result: \((1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21)\)
Merging using a Good Sampler (Example)

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad Rng_{A,B} : A \mapsto \mathbb{N}^{|A|} \quad \text{with} \quad Rng_{A,B}(e) = \text{rng}(e, B) \]

- \( K = (1, 4, 6, 9, 11, 12, 13, 16, 19, 20) \)
- \( J = (2, 3, 7, 8, 10, 14, 15, 17, 18, 21) \)
- \( L = (5, 10, 12, 17) \)

Then we have:

<table>
<thead>
<tr>
<th>( i )</th>
<th>( K_i )</th>
<th>( J_i )</th>
<th>merge((K_i, J_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\text{merge}(K_1, J_1)</td>
<td>\text{merge}(K_1, J_1)</td>
<td>\text{merge}(K_1, J_1)</td>
</tr>
<tr>
<td>2</td>
<td>\text{merge}(K_2, J_2)</td>
<td>\text{merge}(K_2, J_2)</td>
<td>\text{merge}(K_2, J_2)</td>
</tr>
<tr>
<td>3</td>
<td>\text{merge}(K_3, J_3)</td>
<td>\text{merge}(K_3, J_3)</td>
<td>\text{merge}(K_3, J_3)</td>
</tr>
<tr>
<td>4</td>
<td>\text{merge}(K_4, J_4)</td>
<td>\text{merge}(K_4, J_4)</td>
<td>\text{merge}(K_4, J_4)</td>
</tr>
<tr>
<td>5</td>
<td>\text{merge}(K_5, J_5)</td>
<td>\text{merge}(K_5, J_5)</td>
<td>\text{merge}(K_5, J_5)</td>
</tr>
</tbody>
</table>

Result: \((1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21)\)
Merging using a Good Sampler (Example)

\[ \text{rng}(e, S) = |\{ x \in S \mid x < e \}| \text{ and } \text{Rng}_{A,B} : A \mapsto \mathbb{N}^{|A|} \text{ with } \text{Rng}_{A,B}(e) = \text{rng}(e, B) \]

- \( K = (1, 4, 6, 9, 11, 12, 13, 16, 19, 20) \)
- \( J = (2, 3, 7, 8, 10, 14, 15, 17, 18, 21) \)
- \( L = (5, 10, 12, 17) \)
- Then we have:

<table>
<thead>
<tr>
<th>( i )</th>
<th>( K_i )</th>
<th>( J_i )</th>
<th>merge(( K_i, J_i ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1, 4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(6, 9)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(11, 12)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(13, 16)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(19, 20)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Result: \( (1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21) \)
Merging using a Good Sampler (Example)

\[
\text{rng}(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad \text{Rng}_{A,B} : A \mapsto \mathbb{N}^{\mid A\mid} \quad \text{with} \quad \text{Rng}_{A,B}(e) = \text{rng}(e, B)
\]

- \( K = (1, 4, 6, 9, 11, 12, 13, 16, 19, 20) \)
- \( J = (2, 3, 7, 8, 10, 14, 15, 17, 18, 21) \)
- \( L = (5, 10, 12, 17) \)
- Then we have:

<table>
<thead>
<tr>
<th>( i )</th>
<th>( K_i )</th>
<th>( J_i )</th>
<th>( \text{merge}(K_i, J_i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1, 4)</td>
<td>(2, 3)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(6, 9)</td>
<td>(7, 8, 10)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(11, 12)</td>
<td>( \emptyset )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(13, 16)</td>
<td>(14, 15, 17)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(19, 20)</td>
<td>(18, 21)</td>
<td></td>
</tr>
</tbody>
</table>

- Result: \( (1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21) \)
**Merging using a Good Sampler (Example)**

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad R_{A,B} : A \mapsto \mathbb{N}^{\lvert A \rvert} \quad \text{with} \quad R_{A,B}(e) = \text{rng}(e, B) \]

- \( K = (1, 4, 6, 9, 11, 12, 13, 16, 19, 20) \)
- \( J = (2, 3, 7, 8, 10, 14, 15, 17, 18, 21) \)
- \( L = (5, 10, 12, 17) \)

Then we have:

<table>
<thead>
<tr>
<th>( i )</th>
<th>( K_i )</th>
<th>( J_i )</th>
<th>( \text{merge}(K_i, J_i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1, 4)</td>
<td>(2, 3)</td>
<td>(1, 2, 3, 4)</td>
</tr>
<tr>
<td>2</td>
<td>(6, 9)</td>
<td>(7, 8, 10)</td>
<td>(6, 7, 8, 9, 10)</td>
</tr>
<tr>
<td>3</td>
<td>(11, 12)</td>
<td>( \emptyset )</td>
<td>(11, 12)</td>
</tr>
<tr>
<td>4</td>
<td>(13, 16)</td>
<td>(14, 15, 17)</td>
<td>(13, 14, 15, 16, 17)</td>
</tr>
<tr>
<td>5</td>
<td>(19, 20)</td>
<td>(18, 21)</td>
<td>(18, 19, 20, 21)</td>
</tr>
</tbody>
</table>

Result: \((1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21)\)
Merging using a Good Sampler (Example)

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad Rng_{A,B} : A \mapsto \mathbb{N}^{|A|} \quad \text{with} \quad Rng_{A,B}(e) = \text{rng}(e, B) \]

- \( K = (1, 4, 6, 9, 11, 12, 13, 16, 19, 20) \)
- \( J = (2, 3, 7, 8, 10, 14, 15, 17, 18, 21) \)
- \( L = (5, 10, 12, 17) \)

Then we have:

<table>
<thead>
<tr>
<th>( i )</th>
<th>( K_i )</th>
<th>( J_i )</th>
<th>merge((K_i, J_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1, 4)</td>
<td>(2, 3)</td>
<td>(1, 2, 3, 4)</td>
</tr>
<tr>
<td>2</td>
<td>(6, 9)</td>
<td>(7, 8, 10)</td>
<td>(6, 7, 8, 9, 10)</td>
</tr>
<tr>
<td>3</td>
<td>(11, 12)</td>
<td>( \emptyset )</td>
<td>(11, 12)</td>
</tr>
<tr>
<td>4</td>
<td>(13, 16)</td>
<td>(14, 15, 17)</td>
<td>(13, 14, 15, 16, 17)</td>
</tr>
<tr>
<td>5</td>
<td>(19, 20)</td>
<td>(18, 21)</td>
<td>(18, 19, 20, 21)</td>
</tr>
</tbody>
</table>

Result: \((1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21)\)
Merging using a Good Sampler (Example)

rng(e, S) = |\{x \in S \mid x < e\}| and Rng_{A,B} : A \mapsto \mathbb{N}^{\mid A\mid} with Rng_{A,B}(e) = rng(e, B)

\bullet K = (1, 4, 6, 9, 11, 12, 13, 16, 19, 20)
\bullet J = (2, 3, 7, 8, 10, 14, 15, 17, 18, 21)
\bullet L = (5, 10, 12, 17)

Then we have:

<table>
<thead>
<tr>
<th>i</th>
<th>(K_i)</th>
<th>(J_i)</th>
<th>merge((K_i, J_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1, 4)</td>
<td>(2, 3)</td>
<td>(1, 2, 3, 4)</td>
</tr>
<tr>
<td>2</td>
<td>(6, 9)</td>
<td>(7, 8, 10)</td>
<td>(6, 7, 8, 9, 10)</td>
</tr>
<tr>
<td>3</td>
<td>(11, 12)</td>
<td>(14, 15, 17)</td>
<td>(13, 14, 15, 16, 17)</td>
</tr>
<tr>
<td>4</td>
<td>(13, 16)</td>
<td>(18, 21)</td>
<td>(18, 19, 20, 21)</td>
</tr>
</tbody>
</table>

\bullet Result: (1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21)
Merging with good sampler (running time)

\[ \text{rng}(e, S) = |\{ x \in S \mid x < e \}| \text{ and } \text{Rng}_{A,B} : A \mapsto \mathbb{N}^{|A|} \text{ with } \text{Rng}_{A,B}(e) = \text{rng}(e, B) \]

Lemma:

If \( L \) is a good sampler for \( K \) and \( J \).
If \( \text{Rng}_{L,J}, \text{Rng}_{L,K}, \text{Rng}_{K,L} \) and \( \text{Rng}_{J,L} \) is known, then we have:

\[ T_{\text{merge\_with\_help}(J,K,L)} = O(1) \text{ with } P_{\text{merge\_with\_help}(J,K,L)} = O(|J| + |K|). \]

Proof:

- The same way as in the merging introduced in the last chapter.
- Each processor uses \( \text{Rng}_{L,J} \) resp. \( \text{Rng}_{L,K} \) to know the area to read its input sequences.
- Each processor uses \( \text{Rng}_{J,L} \) and \( \text{Rng}_{K,L} \) to know the area to write its output sequence.
Merging with good sampler (running time)

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad Rng_{A,B} : A \mapsto \mathbb{N}^{\left|A\right|} \quad \text{with} \quad Rng_{A,B}(e) = \text{rng}(e, B) \]

Lemma:
If \( L \) is a good sampler for \( K \) and \( J \).
If \( Rng_{L,J}, Rng_{L,K}, Rng_{K,L} \) and \( Rng_{J,L} \) is known, then we have:

\[ T_{\text{merge\_with\_help}(J,K,L)} = O(1) \quad \text{with} \quad P_{\text{merge\_with\_help}(J,K,L)} = O(|J| + |K|). \]

Proof:
- The same way as in the merging introduced in the last chapter.
- Each processor uses \( Rng_{L,J} \) resp. \( Rng_{L,K} \) to know the area to read its input sequences.
- Each processor uses \( Rng_{J,L} \) and \( Rng_{K,L} \) to know the area to write its output sequence.
Merging with good sampler (running time)

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \] and \( \text{Rng}_{A,B} : A \mapsto \mathbb{N}^{|A|} \) with \( \text{Rng}_{A,B}(e) = \text{rng}(e, B) \)

**Lemma:**

If \( L \) is a good sampler for \( K \) and \( J \).

If \( \text{Rng}_{L,J}, \text{Rng}_{L,K}, \text{Rng}_{K,L} \) and \( \text{Rng}_{J,L} \) is known, then we have:

\[ T_{\text{merge with help}(J,K,L)} = O(1) \text{ with } P_{\text{merge with help}(J,K,L)} = O(|J| + |K|). \]

**Proof:**

- The same way as in the merging introduced in the last chapter.
- Each processor uses \( \text{Rng}_{L,J} \) resp. \( \text{Rng}_{L,K} \) to know the area to read its input sequences.
- Each processor uses \( \text{Rng}_{J,L} \) and \( \text{Rng}_{K,L} \) to know the area to write its output sequence.
Merging with good sampler (running time)

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad \text{Rng}_{A,B} : A \mapsto \mathbb{N} |A| \quad \text{with} \quad \text{Rng}_{A,B}(e) = \text{rng}(e, B) \]

**Lemma:**

If \( L \) is a good sampler for \( K \) and \( J \).
If \( \text{Rng}_{L,J}, \text{Rng}_{L,K}, \text{Rng}_{K,L} \) and \( \text{Rng}_{J,L} \) is known, then we have:
\[
T_{\text{merge\_with\_help}(J,K,L)} = O(1) \quad \text{with} \quad P_{\text{merge\_with\_help}(J,K,L)} = O(|J| + |K|).
\]

**Proof:**

- The same way as in the merging introduced in the last chapter.
- Each processor uses \( \text{Rng}_{L,J} \) resp. \( \text{Rng}_{L,K} \) to know the area to read its input sequences.
- Each processor uses \( \text{Rng}_{J,L} \) and \( \text{Rng}_{K,L} \) to know the area to write its output sequence.
Properties of Good Samplers

Lemma:

If \( X \) is a good sampler for \( X' \) and \( Y \) is a good sampler for \( Y' \), then \( \text{merge}(X, Y) \) is a good sampler for \( X' \) [resp. \( Y' \)].

Proof:

- Consider \( X \) as a good sampler for \( X' \).
- Any additional element make the good sampler just "better".

Note:

\( \text{merge}(X, Y) \) is not necessary a sampler for \( \text{merge}(X', Y') \).

- \( X = (2, 7) \) and \( X' = (2, 5, 6, 7) \).
- \( Y = (1, 8) \) and \( Y' = (1, 3, 4, 8) \).
- \( \text{merge}(X, Y) = (1, 2, 7, 8) \) and \( \text{merge}(X', Y') = (1, 2, 3, 4, 5, 6, 7, 8) \).
- There are 5 elements between 2 and 7.
Properties of Good Samplers

rng(e, S) = |{x ∈ S | x < e}| and Rng_{A,B} : A ↦→ \mathbb{N}^{|A|} with Rng_{A,B}(e) = rng(e, B)

Lemma:

If X is a good sampler for X’ and Y is a good sampler for Y’, then merge(X, Y) is a good sampler for X’ [resp. Y’].

Proof:

- Consider X as a good sampler for X’.
- Any additional element make the good sampler just “better”.

Note:

merge(X, Y) is not necessary a sampler for merge(X’, Y’).

- X = (2, 7) and X’ = (2, 5, 6, 7).
- Y = (1, 8) and Y’ = (1, 3, 4, 8).
- merge(X, Y) = (1, 2, 7, 8) and merge(X’, Y’) = (1, 2, 3, 4, 5, 6, 7, 8).
- There are 5 elements between 2 and 7.
Properties of Good Samplers

Lemma:
If $X$ is a good sampler for $X'$ and $Y$ is a good sampler for $Y'$, then $\text{merge}(X, Y)$ is a good sampler for $X'$ [resp. $Y'$].

Proof:
- Consider $X$ as a good sampler for $X'$.
- Any additional element make the good sampler just "better".

Note:
$\text{merge}(X, Y)$ is not necessary a sampler for $\text{merge}(X', Y')$.
- $X = (2, 7)$ and $X' = (2, 5, 6, 7)$.
- $Y = (1, 8)$ and $Y' = (1, 3, 4, 8)$.
- $\text{merge}(X, Y) = (1, 2, 7, 8)$ and $\text{merge}(X', Y') = (1, 2, 3, 4, 5, 6, 7, 8)$.
- There are 5 elements between 2 and 7.
Properties of Good Samplers

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad R_{A,B} : A \mapsto \mathbb{N}^{|A|} \quad \text{with} \quad R_{A,B}(e) = \text{rng}(e, B) \]

**Lemma:**

If \( X \) is a good sampler for \( X' \) and \( Y \) is a good sampler for \( Y' \), then \( \text{merge}(X, Y) \) is a good sampler for \( X' \) [resp. \( Y' \)].

**Proof:**

- Consider \( X \) as a good sampler for \( X' \).
- Any additional element make the good sampler just "better".

**Note:**

\( \text{merge}(X, Y) \) is not necessary a sampler for \( \text{merge}(X', Y') \).

- \( X = (2,7) \) and \( X' = (2,5,6,7) \).
- \( Y = (1,8) \) and \( Y' = (1,3,4,8) \).
- \( \text{merge}(X, Y) = (1,2,7,8) \) and \( \text{merge}(X', Y') = (1,2,3,4,5,6,7,8) \).
- There are 5 elements between 2 and 7.
**Properties of Good Samplers**

\[ \text{rng}(e, S) = | \{x \in S \mid x < e \} | \text{ and } R_{A,B} : A \mapsto \mathbb{N}^{|A|} \text{ with } R_{A,B}(e) = \text{rng}(e, B) \]

**Lemma:**

If \( X \) is a good sampler for \( X' \) and \( Y \) is a good sampler for \( Y' \), then merge\((X, Y)\) is a good sampler for \( X' \) [resp. \( Y' \)].

**Proof:**

- Consider \( X \) as a good sampler for \( X' \).
- Any additional element make the good sampler just "better".

**Note:**

merge\((X, Y)\) is not necessary a sampler for merge\((X', Y')\).

- \( X = (2, 7) \) and \( X' = (2, 5, 6, 7) \).
- \( Y = (1, 8) \) and \( Y' = (1, 3, 4, 8) \).
- merge\((X, Y) = (1, 2, 7, 8) \) and merge\((X', Y') = (1, 2, 3, 4, 5, 6, 7, 8) \).
- There are 5 elements between 2 and 7.
Properties of Good Samplers

rng(e, S) = |{x ∈ S | x < e}| and Rng_{A,B} : A → N^{|A|} with Rng_{A,B}(e) = rng(e, B)

Lemma:
If X is a good sampler for X' and Y is a good sampler for Y', then
merge(X, Y) is a good sampler for X' [resp. Y'].

Proof:
- Consider X as a good sampler for X'.
- Any additional element make the good sampler just "better".

Note:
merge(X, Y) is not necessary a sampler for merge(X', Y').

- X = (2, 7) and X' = (2, 5, 6, 7).
- Y = (1, 8) and Y' = (1, 3, 4, 8).
- merge(X, Y) = (1, 2, 7, 8) and merge(X', Y') = (1, 2, 3, 4, 5, 6, 7, 8).
- There are 5 elements between 2 and 7.
Properties of Good Samplers

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \] and \( R_{A,B} : A \mapsto \mathbb{N}^{\lfloor A \rfloor} \) with \( R_{A,B}(e) = \text{rng}(e, B) \)

Lemma:

If \( X \) is a good sampler for \( X' \) and \( Y \) is a good sampler for \( Y' \), then \( \text{merge}(X, Y) \) is a good sampler for \( X' \) [resp. \( Y' \)].

Proof:

- Consider \( X \) as a good sampler for \( X' \).
- Any additional element make the good sampler just ’’better’’.

Note:

\( \text{merge}(X, Y) \) is not necessary a sampler for \( \text{merge}(X', Y') \).

- \( X = (2, 7) \) and \( X' = (2, 5, 6, 7) \).
- \( Y = (1, 8) \) and \( Y' = (1, 3, 4, 8) \).
- \( \text{merge}(X, Y) = (1, 2, 7, 8) \) and \( \text{merge}(X', Y') = (1, 2, 3, 4, 5, 6, 7, 8) \).
- There are 5 elements between 2 and 7.
Properties of Good Samplers

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \] \text{ and } \text{Rng}_{A,B} : A \mapsto \mathbb{N}^{\lfloor A \rfloor} \text{ with } \text{Rng}_{A,B}(e) = \text{rng}(e, B) \]

**Lemma:**

If \( X \) is a good sampler for \( X' \) and \( Y \) is a good sampler for \( Y' \), then \( \text{merge}(X, Y) \) is a good sampler for \( X' \) [resp. \( Y' \)].

**Proof:**

- Consider \( X \) as a good sampler for \( X' \).
- Any additional element make the good sampler just "better".

**Note:**

\( \text{merge}(X, Y) \) is not necessary a sampler for \( \text{merge}(X', Y') \).

- \( X = (2, 7) \) and \( X' = (2, 5, 6, 7) \).
- \( Y = (1, 8) \) and \( Y' = (1, 3, 4, 8) \).
- \( \text{merge}(X, Y) = (1, 2, 7, 8) \) and \( \text{merge}(X', Y') = (1, 2, 3, 4, 5, 6, 7, 8) \).
- **There are 5 elements between 2 and 7.**
Properties of Good Samplers

\[ \text{rng}(e, S) = |\{ x \in S \mid x < e \}| \quad \text{and} \quad R_{n}^{A,B} : A \mapsto \mathbb{N}^{|A|} \quad \text{with} \quad R_{n}^{A,B}(e) = \text{rng}(e, B) \]

**Lemma:**

Let X be a good sampler for \( X' \) and let Y be a good sampler for \( Y' \).

Then there are at most \( 2 \cdot r + 2 \) elements of merge\((X', Y')\) between \( r \) successive elements of merge\((X, Y)\).

**Proof:**

- W.l.o.g. contain \( X \) and \( Y \) elements \(-\infty\) and \(+\infty\).
- Let \((e_1, e_2, \cdots, e_r)\) successive elements of merge\((X, Y)\).
- W.l.o.g. let \( e_1 \in X \).
- Consider now two cases: \( e_r \in X \) and \( e_r \in Y \).
- Let in the following be

\[
x = |X \cap \{e_1, e_2, \cdots, e_r\}| \quad \text{and} \quad y = |Y \cap \{e_1, e_2, \cdots, e_r\}|.
\]
Properties of Good Samplers

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \text{ and } Rng_{A,B} : A \mapsto \mathbb{N}^{|A|} \text{ with } Rng_{A,B}(e) = \text{rng}(e, B) \]

Lemma:

Let \( X \) be a good sampler for \( X' \) and let \( Y \) be a good sampler for \( Y' \). Then there are at most \( 2 \cdot r + 2 \) elements of \( \text{merge}(X', Y') \) between \( r \) successive elements of \( \text{merge}(X, Y) \).

Proof:

- W.l.o.g. contain \( X \) and \( Y \) elements \(-\infty\) and \(+\infty\).
- Let \((e_1, e_2, \cdots, e_r)\) successive elements of \( \text{merge}(X, Y) \).
- W.l.o.g. let \( e_1 \in X \).
- Consider now two cases: \( e_r \in X \) and \( e_r \in Y \).
- Let in the following be

\[
\begin{align*}
x &= |X \cap \{e_1, e_2, \cdots, e_r\}| \
y &= |Y \cap \{e_1, e_2, \cdots, e_r\}|.
\end{align*}
\]
Properties of Good Samplers

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad \text{Rng}_{A,B} : A \mapsto \mathbb{N}^{\mid A\mid} \quad \text{with} \quad \text{Rng}_{A,B}(e) = \text{rng}(e, B) \]

**Lemma:**

Let \( X \) be a good sampler for \( X' \) and let \( Y \) be a good sampler for \( Y' \). Then there are at most \( 2 \cdot r + 2 \) elements of merge\((X', Y')\) between \( r \) successive elements of merge\((X, Y)\).

**Proof:**

- W.l.o.g. contain \( X \) and \( Y \) elements \(-\infty\) and \(+\infty\).
- Let \((e_1, e_2, \cdots, e_r)\) successive elements of merge\((X, Y)\).
- W.l.o.g. let \( e_1 \in X \).
- Consider now two cases: \( e_r \in X \) and \( e_r \in Y \).
- Let in the following be

\[
\begin{align*}
x &= |X \cap \{e_1, e_2, \cdots, e_r\}| \quad \text{and} \\
y &= |Y \cap \{e_1, e_2, \cdots, e_r\}|.
\end{align*}
\]
Properties of Good Samplers

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \text{ and } \text{Rng}_{A,B} : A \mapsto \mathbb{N}^{|A|} \text{ with } \text{Rng}_{A,B}(e) = \text{rng}(e, B) \]

Lemma:
Let \( X \) be a good sampler for \( X' \) and let \( Y \) be a good sampler for \( Y' \). Then there are at most \( 2 \cdot r + 2 \) elements of \( \text{merge}(X', Y') \) between \( r \) successive elements of \( \text{merge}(X, Y) \).

Proof:
- W.l.o.g. contain \( X \) and \( Y \) elements \(-\infty \) and \(+\infty\).
- Let \((e_1, e_2, \cdots, e_r)\) successive elements of \( \text{merge}(X, Y) \).
- W.l.o.g. let \( e_1 \in X \).
- Consider now two cases: \( e_r \in X \) and \( e_r \in Y \).
- Let in the following be
  \[
  x = |X \cap \{e_1, e_2, \cdots, e_r\}| \quad \text{and} \quad y = |Y \cap \{e_1, e_2, \cdots, e_r\}|.
  \]
Properties of Good Samplers

Lemma:
Let $X$ be a good sampler for $X'$ and let $Y$ be a good sampler for $Y'$.
Then there are at most $2 \cdot r + 2$ elements of $\text{merge}(X', Y')$ between $r$ successive elements of $\text{merge}(X, Y)$.

Proof:
- W.l.o.g. contain $X$ and $Y$ elements $-\infty$ and $+\infty$.
- Let $(e_1, e_2, \cdots, e_r)$ successive elements of $\text{merge}(X, Y)$.
- W.l.o.g. let $e_1 \in X$.
- Consider now two cases: $e_r \in X$ and $e_r \in Y$.
- Let in the following be
  \[
  x = |X \cap \{e_1, e_2, \cdots, e_r\}| \quad \text{and} \quad
  y = |Y \cap \{e_1, e_2, \cdots, e_r\}|.
  \]
Properties of Good Samplers

\[ \text{rng}(e, S) = |\{x \in S \mid x < e\}| \quad \text{and} \quad \text{Rng}_{A,B} : A \mapsto \mathbb{N}^{|A|} \quad \text{with} \quad \text{Rng}_{A,B}(e) = \text{rng}(e, B) \]

**Lemma:**

Let \( X \) be a good sampler for \( X' \) and let \( Y \) be a good sampler for \( Y' \). Then there are at most \( 2 \cdot r + 2 \) elements of \( \text{merge}(X', Y') \) between \( r \) successive elements of \( \text{merge}(X, Y) \).

**Proof:**

- W.l.o.g. contain \( X \) and \( Y \) elements \(-\infty\) and \( +\infty\).
- Let \((e_1, e_2, \cdots, e_r)\) successive elements of \( \text{merge}(X, Y) \).
- W.l.o.g. let \( e_1 \in X \).
- Consider now two cases: \( e_r \in X \) and \( e_r \in Y \).
- Let in the following be

\[
\begin{align*}
x &= |X \cap \{e_1, e_2, \cdots, e_r\}| \quad \text{and} \\
y &= |Y \cap \{e_1, e_2, \cdots, e_r\}|.
\end{align*}
\]
Properties of Good Samplers

\((e_1, e_2, \cdots, e_r)\) successive elements of merge\((X, Y)\) and \(x = |X \cap \{e_1, e_2, \cdots, e_r\}|\) and \(y = |Y \cap \{e_1, e_2, \cdots, e_r\}|\) and

**Lemma:**

Let \(X\) be a good sampler for \(X'\) and let \(Y\) be a good sampler for \(Y'\). Then there are at most \(2 \cdot r + 2\) elements of merge\((X', Y')\) between \(r\) successive elements of merge\((X, Y)\).

Proof: W.l.o.g. let \(e_1 \in X\).

If: \(e_r \in X\)

- Between \(e_1\) and \(e_r\) are at most \(2(x - 1) + 1\) elements of \(X'\).
- Between \(e_1\) and \(e_r\) are at most \(2(y + 1) + 1\) elements of \(Y'\), because they are between \(y + 2\) elements of \(Y\).

Thus we get: \(2(x - 1) + 1 + 2(y + 1) + 1 = 2 \cdot r + 2\).

Example \(x = 3\) and \(y = 2\):

\[ e_1 \in X \quad e_2 \in Y \quad e_3 \in X \quad e_4 \in Y \quad e_5 \in X \]
Properties of Good Samplers

\((e_1, e_2, \cdots, e_r)\) successive elements of \(\text{merge}(X, Y)\) and \(x = |X \cap \{e_1, e_2, \cdots, e_r\}|\) and \(y = |Y \cap \{e_1, e_2, \cdots, e_r\}|\) and

**Lemma:**

Let \(X\) be a good sampler for \(X'\) and let \(Y\) be a good sampler for \(Y'\). Then there are at most \(2 \cdot r + 2\) elements of \(\text{merge}(X', Y')\) between \(r\) successive elements of \(\text{merge}(X, Y)\).

Proof: W.l.o.g. let \(e_1 \in X\).

If: \(e_r \in X\)

- Between \(e_1\) and \(e_r\) are at most \(2(x - 1) + 1\) elements of \(X'\).
- Between \(e_1\) and \(e_r\) are at most \(2(y + 1) + 1\) elements of \(Y'\), because they are between \(y + 2\) elements of \(Y\).

Thus we get: \(2(x - 1) + 1 + 2(y + 1) + 1 = 2 \cdot r + 2\).

Example \(x = 3\) and \(y = 2\):

\[
e_1 \in X \quad e_2 \in Y \quad e_3 \in X \quad e_4 \in Y \quad e_5 \in X
\]
Properties of Good Samplers

\((e_1, e_2, \cdots, e_r)\) successive elements of \(\text{merge}(X, Y)\) and
\(x = |X \cap \{e_1, e_2, \cdots, e_r\}|\) and \(y = |Y \cap \{e_1, e_2, \cdots, e_r\}|\) and

**Lemma:**

Let \(X\) be a good sampler for \(X'\) and let \(Y\) be a good sampler for \(Y'\).
Then there are at most \(2 \cdot r + 2\) elements of \(\text{merge}(X', Y')\) between \(r\) successive elements of \(\text{merge}(X, Y)\).

**Proof:** W.l.o.g. let \(e_1 \in X\).
If: \(e_r \in X\)

- Between \(e_1\) and \(e_r\) are at most \(2(x - 1) + 1\) elements of \(X'\).
- Between \(e_1\) and \(e_r\) are at most \(2(y + 1) + 1\) elements of \(Y'\), because they are between \(y + 2\) elements of \(Y\).

Thus we get: \(2(x - 1) + 1 + 2(y + 1) + 1 = 2 \cdot r + 2\).

**Example** \(x = 3\) and \(y = 2\):

\[ a \in Y \quad e_1 \in X \quad e_2 \in Y \quad e_3 \in X \quad e_4 \in Y \quad e_5 \in X \quad b \in Y \]
Properties of Good Samplers

\((e_1, e_2, \ldots, e_r)\) successive elements of merge\((X, Y)\) and \(x = |X \cap \{e_1, e_2, \ldots, e_r\}|\) and \(y = |Y \cap \{e_1, e_2, \ldots, e_r\}|\)

**Lemma:**

Let \(X\) be a good sampler for \(X'\) and let \(Y\) be a good sampler for \(Y'\). Then there are at most \(2 \cdot r + 2\) elements of merge\((X', Y')\) between \(r\) successive elements of merge\((X, Y)\).

**Proof:** W.l.o.g. let \(e_1 \in X\). If: \(e_r \in Y\)

- Add \(e_0 \in Y\) with \(e_0 < e_1\) to the good sampler.
- Add \(e_{r+1} \in X\) with \(e_r < e_{r+1}\) to the good sampler.
- The elements from \(X'\) between \((e_1, e_2, \ldots, e_r)\) are between \(x + 1\) elements from \(X\).
- The elements from \(Y'\) between \((e_1, e_2, \ldots, e_r)\) are between \(y + 1\) elements from \(Y\).
- Thus we get: \(2x + 1 + 2y + 1 = 2r + 2\).

**Example** \(x = 2\) and \(y = 2\):

\[
e_1 \in X \quad e_2 \in Y \quad e_3 \in X \quad e_4 \in Y
\]
Properties of Good Samplers

\[(e_1, e_2, \ldots, e_r)\] successive elements of merge\((X, Y)\) and \(x = |X \cap \{e_1, e_2, \ldots, e_r\}|\) and \(y = |Y \cap \{e_1, e_2, \ldots, e_r\}|\) and

**Lemma:**

Let \(X\) be a good sampler for \(X'\) and let \(Y\) be a good sampler for \(Y'\). Then there are at most \(2 \cdot r + 2\) elements of merge\((X', Y')\) between \(r\) successive elements of merge\((X, Y)\).

**Proof:** W.l.o.g. let \(e_1 \in X\). If: \(e_r \in Y\)

- Add \(e_0 \in Y\) with \(e_0 < e_1\) to the good sampler.
- Add \(e_{r+1} \in X\) with \(e_r < e_{r+1}\) to the good sampler.
- The elements from \(X'\) between \((e_1, e_2, \ldots, e_r)\) are between \(x + 1\) elements from \(X\).
- The elements from \(Y'\) between \((e_1, e_2, \ldots, e_r)\) are between \(y + 1\) elements from \(Y\).
- Thus we get: \(2x + 1 + 2y + 1 = 2r + 2\).

**Example** \(x = 2\) and \(y = 2\):

\[
\begin{align*}
e_0 & \in Y \\
e_1 & \in X \\
e_2 & \in Y \\
e_3 & \in X \\
e_4 & \in Y
\end{align*}
\]
Properties of Good Samplers

\((e_1, e_2, \cdots, e_r)\) successive elements of \(\text{merge}(X, Y)\) and \(x = |X \cap \{e_1, e_2, \cdots, e_r\}|\) and \(y = |Y \cap \{e_1, e_2, \cdots, e_r\}|\) and

**Lemma:**

Let \(X\) be a good sampler for \(X'\) and let \(Y\) be a good sampler for \(Y'\). Then there are at most \(2 \cdot r + 2\) elements of \(\text{merge}(X', Y')\) between \(r\) successive elements of \(\text{merge}(X, Y)\).

**Proof:** W.l.o.g. let \(e_1 \in X\). If: \(e_r \in Y\)

- Add \(e_0 \in Y\) with \(e_0 < e_1\) to the good sampler.
- Add \(e_{r+1} \in X\) with \(e_r < e_{r+1}\) to the good sampler.

The elements from \(X'\) between \((e_1, e_2, \cdots, e_r)\) are between \(x + 1\) elements from \(X\).

The elements from \(Y'\) between \((e_1, e_2, \cdots, e_r)\) are between \(y + 1\) elements from \(Y\).

Thus we get: \(2x + 1 + 2y + 1 = 2r + 2\).

**Example** \(x = 2\) and \(y = 2\):

\[e_0 \in Y \quad e_1 \in X \quad e_2 \in Y \quad e_3 \in X \quad e_4 \in Y \quad e_5 \in X\]
Properties of Good Samplers

\((e_1, e_2, \ldots, e_r)\) successive elements of merge\((X, Y)\) and \(x = |X \cap \{e_1, e_2, \ldots, e_r\}|\) and \(y = |Y \cap \{e_1, e_2, \ldots, e_r\}|\) and

**Lemma:**

Let \(X\) be a good sampler for \(X'\) and let \(Y\) be a good sampler for \(Y'\). Then there are at most \(2 \cdot r + 2\) elements of merge\((X', Y')\) between \(r\) successive elements of merge\((X, Y)\).

**Proof:** W.l.o.g. let \(e_1 \in X\). If: \(e_r \in Y\)

- Add \(e_0 \in Y\) with \(e_0 < e_1\) to the good sampler.
- Add \(e_{r+1} \in X\) with \(e_r < e_{r+1}\) to the good sampler.
- The elements from \(X'\) between \((e_1, e_2, \ldots, e_r)\) are between \(x + 1\) elements from \(X\).
- **The elements from \(Y'\) between \((e_1, e_2, \ldots, e_r)\) are between \(y + 1\) elements from \(Y\).**
- Thus we get: \(2x + 1 + 2y + 1 = 2r + 2\).

**Example** \(x = 2\) and \(y = 2\):

\[e_0 \in Y \quad e_1 \in X \quad e_2 \in Y \quad e_3 \in X \quad e_4 \in Y \quad e_5 \in X\]
Properties of Good Samplers

\[(e_1, e_2, \cdots, e_r)\] successive elements of merge\((X, Y)\) and \(x = |X \cap \{e_1, e_2, \cdots, e_r\}|\) and \(y = |Y \cap \{e_1, e_2, \cdots, e_r\}|\) and

**Lemma:**

Let \(X\) be a good sampler for \(X'\) and let \(Y\) be a good sampler for \(Y'\). Then there are at most \(2 \cdot r + 2\) elements of merge\((X', Y')\) between \(r\) successive elements of merge\((X, Y)\).

**Proof:** W.l.o.g. let \(e_1 \in X\). If: \(e_r \in Y\)

- Add \(e_0 \in Y\) with \(e_0 < e_1\) to the good sampler.
- Add \(e_{r+1} \in X\) with \(e_r < e_{r+1}\) to the good sampler.
- The elements from \(X'\) between \((e_1, e_2, \cdots, e_r)\) are between \(x + 1\) elements from \(X\).
- The elements from \(Y'\) between \((e_1, e_2, \cdots, e_r)\) are between \(y + 1\) elements from \(Y\).

Thus we get: \(2x + 1 + 2y + 1 = 2r + 2\).

Example \(x = 2\) and \(y = 2\):

\[e_0 \in Y \quad e_1 \in X \quad e_2 \in Y \quad e_3 \in X \quad e_4 \in Y \quad e_5 \in X\]
Properties of good sampler

**At most** $2 \cdot r + 2$ elements of $\text{merge}(X', Y')$ between $r$ successive elements of $\text{merge}(X, Y)$

**Definition**

Let $\text{reduce}(X)$ be the operation, which chooses from $X$ every forth element.

**Lemma:**

If $X$ is a good sampler for $X'$ and $Y$ is a good sampler for $Y'$, then $\text{reduce}(\text{merge}(X, Y))$ is a good sampler for $\text{reduce}(\text{merge}(X', Y'))$.

**Proof:**

- Consider $k + 1$ successive elements $(e_1, e_2, \cdots, e_{k+1})$ of $\text{reduce}(\text{merge}(X, Y))$.
- At most $4k + 1$ elements of $\text{merge}(X, Y)$ are between $e_1, e_2, \cdots, e_{k+1}$ including $e_1, e_{k+1}$.
- At most $8k + 4$ elements of $\text{merge}(X', Y')$ are between these $4k + 1$ elements.
- At most $2k + 1$ elements of $\text{reduce}(\text{merge}(X', Y'))$ are between $(e_1, e_2, \cdots, e_{k+1})$. 
Properties of good sampler

At most \(2 \cdot r + 2\) elements of \(\text{merge}(X', Y')\) between \(r\) successive elements of \(\text{merge}(X, Y)\)

**Definition**

Let \(\text{reduce}(X)\) be the operation, which chooses from \(X\) every forth element.

**Lemma:**

If \(X\) is a good sampler for \(X'\) and \(Y\) is a good sampler for \(Y'\), then \(\text{reduce(merge}(X, Y))\) is a good sampler for \(\text{reduce(merge}(X', Y'))\).

**Proof:**

- Consider \(k + 1\) successive elements \((e_1, e_2, \ldots , e_{k+1})\) of \(\text{reduce(merge}(X, Y))\).
- At most \(4k + 1\) elements of \(\text{merge}(X, Y)\) are between \(e_1, e_2, \ldots , e_{k+1}\) including \(e_1, e_{k+1}\).
- At most \(8k + 4\) elements of \(\text{merge}(X', Y')\) are between these \(4k + 1\) elements.
- At most \(2k + 1\) elements of \(\text{reduce(merge}(X', Y'))\) are between \((e_1, e_2, \ldots , e_{k+1})\).
Properties of good sampler

At most $2 \cdot r + 2$ elements of $\text{merge}(X', Y')$ between $r$ successive elements of $\text{merge}(X, Y)$

**Definition**

Let $\text{reduce}(X)$ be the operation, which chooses from $X$ every forth element.

**Lemma:**

If $X$ is a good sampler for $X'$ and $Y$ is a good sampler for $Y'$, then $\text{reduce}(\text{merge}(X, Y))$ is a good sampler for $\text{reduce}(\text{merge}(X', Y'))$.

**Proof:**

- Consider $k + 1$ successive elements $(e_1, e_2, \cdots, e_{k+1})$ of $\text{reduce}(\text{merge}(X, Y))$.
- At most $4k + 1$ elements of $\text{merge}(X, Y)$ are between $e_1, e_2, \cdots, e_{k+1}$ including $e_1, e_{k+1}$.
- At most $8k + 4$ elements of $\text{merge}(X', Y')$ are between these $4k + 1$ elements.
- At most $2k + 1$ elements of $\text{reduce}(\text{merge}(X', Y'))$ are between $(e_1, e_2, \cdots, e_{k+1})$. 
Properties of good sampler

At most $2 \cdot r + 2$ elements of $\text{merge}(X', Y')$ between $r$ successive elements of $\text{merge}(X, Y)$

**Definition**

Let reduce($X$) be the operation, which chooses from $X$ every forth element.

**Lemma:**

If $X$ is a good sampler for $X'$ and $Y$ is a good sampler for $Y'$, then reduce(merge($X, Y$)) is a good sampler for reduce(merge($X', Y'$)).

**Proof:**

- Consider $k + 1$ successive elements ($e_1, e_2, \cdots, e_{k+1}$) of reduce(merge($X, Y$)).
- At most $4k + 1$ elements of $\text{merge}(X, Y)$ are between $e_1, e_2, \cdots, e_{k+1}$ including $e_1, e_{k+1}$.
- At most $8k + 4$ elements of $\text{merge}(X', Y')$ are between these $4k + 1$ elements.
- At most $2k + 1$ elements of reduce(merge($X', Y'$)) are between ($e_1, e_2, \cdots, e_{k+1}$).
Properties of good sampler

At most $2 \cdot r + 2$ elements of $\text{merge}(X', Y')$ between $r$ successive elements of $\text{merge}(X, Y)$

Definition

Let $\text{reduce}(X)$ be the operation, which chooses from $X$ every forth element.

Lemma:

If $X$ is a good sampler for $X'$ and $Y$ is a good sampler for $Y'$, then $\text{reduce}(\text{merge}(X, Y))$ is a good sampler for $\text{reduce}(\text{merge}(X', Y'))$.

Proof:

1. Consider $k + 1$ successive elements $(e_1, e_2, \cdots, e_{k+1})$ of $\text{reduce}(\text{merge}(X, Y))$.
2. At most $4k + 1$ elements of $\text{merge}(X, Y)$ are between $e_1, e_2, \cdots, e_{k+1}$ including $e_1, e_{k+1}$.
3. At most $8k + 4$ elements of $\text{merge}(X', Y')$ are between these $4k + 1$ elements.
4. At most $2k + 1$ elements of $\text{reduce}(\text{merge}(X', Y'))$ are between $(e_1, e_2, \cdots, e_{k+1})$. 
Properties of good sampler

At most $2 \cdot r + 2$ elements of $\text{merge}(X', Y')$ between $r$ successive elements of $\text{merge}(X, Y)$

**Definition**

Let $\text{reduce}(X)$ be the operation, which chooses from $X$ every forth element.

**Lemma:**

If $X$ is a good sampler for $X'$ and $Y$ is a good sampler for $Y'$, then $\text{reduce}(\text{merge}(X, Y))$ is a good sampler for $\text{reduce}(\text{merge}(X', Y'))$.

Proof:

- Consider $k + 1$ successive elements $(e_1, e_2, \cdots, e_{k+1})$ of $\text{reduce}(\text{merge}(X, Y))$.
- At most $4k + 1$ elements of $\text{merge}(X, Y)$ are between $e_1, e_2, \cdots, e_{k+1}$ including $e_1, e_{k+1}$.
- At most $8k + 4$ elements of $\text{merge}(X', Y')$ are between these $4k + 1$ elements.
- At most $2k + 1$ elements of $\text{reduce}(\text{merge}(X', Y'))$ are between $(e_1, e_2, \cdots, e_{k+1})$. 
Overview to the Algorithm of Cole

- We start with an explanation using a complete binary tree.
- The leaves contain the elements to be sorted.
- Interior nodes $v$ “cares” about as many elements as the number of leaves below $v$.
- A node $v$ receives from its sons sequences of already sorted sequences.
- The “length” of the sequences doubles each time.
- Node $v$ receives sequences $X_1, X_2, \ldots, X_r$ and $Y_1, Y_2, \ldots, Y_r$.
- Node $v$ sends to his father sequences $Z_1, Z_2, \ldots, Z_r, Z_{r+1}$.
- Node $v$ updates an interior help-sequence $val_v$.
- It holds: $|X_1| = |Y_1| = |Z_1| = 1$.
- It holds: $|X_i| = 2 \cdot |X_{i-1}|$, $|Y_i| = 2 \cdot |Y_{i-1}|$ and $|Z_i| = 2 \cdot |Z_{i-1}|$. 
Overview to the Algorithm of Cole

- We start with an explanation using a complete binary tree.
- The leaves contain the elements to be sorted.
- Interior nodes $v$ “cares” about as many elements as the number of leaves below $v$.
- A node $v$ receives from its sons sequences of already sorted sequences.
- The “length” of the sequences doubles each time.
- Node $v$ receives sequences $X_1, X_2, \cdots, X_r$ and $Y_1, Y_2, \cdots, Y_r$.
- Node $v$ sends to his father sequences $Z_1, Z_2, \cdots, Z_r, Z_{r+1}$.
- Node $v$ updates a interior help-sequence $val_v$.
- It holds: $|X_1| = |Y_1| = |Z_1| = 1$.
- It holds: $|X_i| = 2 \cdot |X_{i-1}|$, $|Y_i| = 2 \cdot |Y_{i-1}|$ and $|Z_i| = 2 \cdot |Z_{i-1}|$. 
Overview to the Algorithm of Cole

- We start with an explanation using a complete binary tree.
- The leaves contain the elements to be sorted.
- Interior nodes $v$ “cares” about as many elements as the number of leaves below $v$.
- A node $v$ receives from its sons sequences of already sorted sequences.
- The “length” of the sequences doubles each time.
- Node $v$ receives sequences $X_1, X_2, \ldots, X_r$ and $Y_1, Y_2, \ldots, Y_r$.
- Node $v$ sends to his father sequences $Z_1, Z_2, \ldots, Z_r, Z_{r+1}$.
- Node $v$ updates a interior help-sequence $val_v$.
- It holds: $|X_1| = |Y_1| = |Z_1| = 1$.
- It holds: $|X_i| = 2 \cdot |X_{i-1}|$, $|Y_i| = 2 \cdot |Y_{i-1}|$ and $|Z_i| = 2 \cdot |Z_{i-1}|$. 
Overview to the Algorithm of Cole

- We start with an explanation using a complete binary tree.
- The leaves contain the elements to be sorted.
- Interior nodes \( v \) "cares" about as many elements as the number of leaves below \( v \).
- A node \( v \) receives from its sons sequences of already sorted sequences.
- The "length" of the sequences doubles each time.
- Node \( v \) receives sequences \( X_1, X_2, \ldots, X_r \) and \( Y_1, Y_2, \ldots, Y_r \).
- Node \( v \) sends to his father sequences \( Z_1, Z_2, \ldots, Z_r, Z_{r+1} \).
- Node \( v \) updates a interior help-sequence \( \text{val}_v \).
- It holds: \( |X_1| = |Y_1| = |Z_1| = 1 \).
- It holds: \( |X_i| = 2 \cdot |X_{i-1}|, |Y_i| = 2 \cdot |Y_{i-1}| \) and \( |Z_i| = 2 \cdot |Z_{i-1}| \).
Overview to the Algorithm of Cole

- We start with an explanation using a complete binary tree.
- The leaves contain the elements to be sorted.
- Interior nodes \( v \) “cares” about as many elements as the number of leaves below \( v \).
- A node \( v \) receives from its sons sequences of already sorted sequences.
- The “length” of the sequences doubles each time.
- Node \( v \) receives sequences \( X_1, X_2, \cdots, X_r \) and \( Y_1, Y_2, \cdots, Y_r \).
- Node \( v \) sends to his father sequences \( Z_1, Z_2, \cdots, Z_r, Z_{r+1} \).
- Node \( v \) updates a interior help-sequence \( \text{val}_v \).
- It holds: \( |X_1| = |Y_1| = |Z_1| = 1 \).
- It holds: \( |X_i| = 2 \cdot |X_{i-1}|, |Y_i| = 2 \cdot |Y_{i-1}| \) and \( |Z_i| = 2 \cdot |Z_{i-1}| \).
Overview to the Algorithm of Cole

- We start with an explanation using a complete binary tree.
- The leaves contain the elements to be sorted.
- Interior nodes \( v \) “cares” about as many elements as the number of leaves below \( v \).
- A node \( v \) receives from its sons sequences of already sorted sequences.
- The “length” of the sequences doubles each time.
- Node \( v \) receives sequences \( X_1, X_2, \cdots, X_r \) and \( Y_1, Y_2, \cdots, Y_r \).
- Node \( v \) sends to his father sequences \( Z_1, Z_2, \cdots, Z_r, Z_{r+1} \).
- Node \( v \) updates an interior help-sequence \( \text{val}_v \).
- It holds: \( |X_1| = |Y_1| = |Z_1| = 1 \).
- It holds: \( |X_i| = 2 \cdot |X_{i-1}|, \ |Y_i| = 2 \cdot |Y_{i-1}| \) and \( |Z_i| = 2 \cdot |Z_{i-1}| \).
Overview to the Algorithm of Cole

- We start with an explanation using a complete binary tree.
- The leaves contain the elements to be sorted.
- Interior nodes $v$ "cares" about as many elements as the number of leaves below $v$.
- A node $v$ receives from its sons sequences of already sorted sequences.
- The "length" of the sequences doubles each time.
- Node $v$ receives sequences $X_1, X_2, \ldots, X_r$ and $Y_1, Y_2, \ldots, Y_r$.
- Node $v$ sends to his father sequences $Z_1, Z_2, \ldots, Z_r, Z_{r+1}$.
- Node $v$ updates an interior help-sequence $v a l_v$.
- It holds: $|X_1| = |Y_1| = |Z_1| = 1$.
- It holds: $|X_i| = 2 \cdot |X_{i-1}|$, $|Y_i| = 2 \cdot |Y_{i-1}|$ and $|Z_i| = 2 \cdot |Z_{i-1}|$. 
Overview to the Algorithm of Cole

- We start with an explanation using a complete binary tree.
- The leaves contain the elements to be sorted.
- Interior nodes $v$ “cares” about as many elements as the number of leaves below $v$.
- A node $v$ receives from its sons sequences of already sorted sequences.
- The “length” of the sequences doubles each time.
- Node $v$ receives sequences $X_1, X_2, \cdots, X_r$ and $Y_1, Y_2, \cdots, Y_r$.
- Node $v$ sends to his father sequences $Z_1, Z_2, \cdots, Z_r, Z_{r+1}$.
- Node $v$ updates an interior help-sequence $val_v$.
- It holds: $|X_1| = |Y_1| = |Z_1| = 1$.
- It holds: $|X_i| = 2 \cdot |X_{i-1}|$, $|Y_i| = 2 \cdot |Y_{i-1}|$ and $|Z_i| = 2 \cdot |Z_{i-1}|$. 
Overview to the Algorithm of Cole

- We start with an explanation using a complete binary tree.
- The leaves contain the elements to be sorted.
- Interior nodes \( v \) “cares” about as many elements as the number of leaves below \( v \).
- A node \( v \) receives from its sons sequences of already sorted sequences.
- The “length” of the sequences doubles each time.
- Node \( v \) receives sequences \( X_1, X_2, \ldots, X_r \) and \( Y_1, Y_2, \ldots, Y_r \).
- Node \( v \) sends to his father sequences \( Z_1, Z_2, \ldots, Z_r, Z_{r+1} \).
- Node \( v \) updates an interior help-sequence \( \text{val}_v \).

**It holds:** \( |X_1| = |Y_1| = |Z_1| = 1 \).

**It holds:** \( |X_i| = 2 \cdot |X_{i-1}|, \  |Y_i| = 2 \cdot |Y_{i-1}| \) and \( |Z_i| = 2 \cdot |Z_{i-1}| \).
Overview to the Algorithm of Cole

- We start with an explanation using a complete binary tree.
- The leaves contain the elements to be sorted.
- Interior nodes \( v \) “cares” about as many elements as the number of leaves below \( v \).
- A node \( v \) receives from its sons sequences of already sorted sequences.
- The “length” of the sequences doubles each time.
- Node \( v \) receives sequences \( X_1, X_2, \ldots, X_r \) and \( Y_1, Y_2, \ldots, Y_r \).
- Node \( v \) sends to his father sequences \( Z_1, Z_2, \ldots, Z_r, Z_{r+1} \).
- Node \( v \) updates an interior help-sequence \( \text{val}_v \).
- It holds: \( |X_1| = |Y_1| = |Z_1| = 1 \).
- It holds: \( |X_i| = 2 \cdot |X_{i-1}|, |Y_i| = 2 \cdot |Y_{i-1}| \) and \( |Z_i| = 2 \cdot |Z_{i-1}| \).
One basic Operation of an interior Node \( v \)

- Receives from its sons the two sequences \( X \) and \( Y \).
- Computes: \( val_v = \text{merge\_with\_help}(X, Y, val_v) \).
- Sends to its father: \( \text{reduce}(val_v) \) till \( v \) has sorted all received sequences.
- Sends to its father each second element from \( val_v \), if \( v \) is done with sorting.
- Sends to its father \( val_v \), if \( v \) finishes sorting two steps before.
- **Example:**

<table>
<thead>
<tr>
<th>Step</th>
<th>Left</th>
<th>Right</th>
<th>( val_v )</th>
<th>Father</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>7,8</td>
<td>∅</td>
</tr>
<tr>
<td>2</td>
<td>3,7</td>
<td>5,8</td>
<td>3,5,7,8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>4,8</td>
</tr>
<tr>
<td>4</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>2,4,6,8</td>
</tr>
<tr>
<td>5</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
</tbody>
</table>
One basic Operation of an interior Node $v$

- Receives from its sons the two sequences $X$ and $Y$.
- Computes: $val_v = \text{merge}_\text{with}_\text{help}(X, Y, val_v)$.
- Sends to its father: $\text{reduce}(val_v)$ till $v$ has sorted all received sequences.
- Sends to its father each second element from $val_v$, if $v$ is done with sorting.
- Sends to its father $val_v$, if $v$ finishes sorting two steps before.
- Example:

<table>
<thead>
<tr>
<th>Step</th>
<th>Left</th>
<th>Right</th>
<th>$val_v$</th>
<th>Father</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>7,8</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>2</td>
<td>3,7</td>
<td>5,8</td>
<td>3,5,7,8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>4,8</td>
</tr>
<tr>
<td>4</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>2,4,6,8</td>
</tr>
<tr>
<td>5</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
</tbody>
</table>
One basic Operation of an interior Node $v$

- Receives from its sons the two sequences $X$ and $Y$.
- Computes: $val_v = merge\_with\_help(X, Y, val_v)$.
- Sends to its father: reduce($val_v$) till $v$ has sorted all received sequences.
- Sends to its father each second element from $val_v$, if $v$ is done with sorting.
- Sends to its father $val_v$, if $v$ finishes sorting two steps before.

Example:

<table>
<thead>
<tr>
<th>Step</th>
<th>Left</th>
<th>Right</th>
<th>$val_v$</th>
<th>Father</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>7,8</td>
<td>∅</td>
</tr>
<tr>
<td>2</td>
<td>3,7</td>
<td>5,8</td>
<td>3,5,7,8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>4,8</td>
</tr>
<tr>
<td>4</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>2,4,6,8</td>
</tr>
<tr>
<td>5</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
</tbody>
</table>
One basic Operation of an interior Node $v$

- Receives from its sons the two sequences $X$ and $Y$.
- Computes: $val_v = \text{merge\_with\_help}(X, Y, val_v)$.
- Sends to its father: reduce($val_v$) till $v$ has sorted all received sequences.
- Sends to its father each second element from $val_v$, if $v$ is done with sorting.
- Sends to its father $val_v$, if $v$ finishes sorting two steps before.

**Example:**

<table>
<thead>
<tr>
<th>Step</th>
<th>Left</th>
<th>Right</th>
<th>$val_v$</th>
<th>Father</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>7,8</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>2</td>
<td>3,7</td>
<td>5,8</td>
<td>3,5,7,8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>4,8</td>
</tr>
<tr>
<td>4</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>2,4,6,8</td>
</tr>
<tr>
<td>5</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
</tbody>
</table>
One basic Operation of an interior Node $v$

- Receives from its sons the two sequences $X$ and $Y$.
- Computes: $val_v = \text{merge\_with\_help}(X, Y, val_v)$.
- Sends to its father: $\text{reduce}(val_v)$ till $v$ has sorted all received sequences.
- Sends to its father each second element from $val_v$, if $v$ is done with sorting.
- Sends to its father $val_v$, if $v$ finishes sorting two steps before.
- Example:

<table>
<thead>
<tr>
<th>Step</th>
<th>Left</th>
<th>Right</th>
<th>$val_v$</th>
<th>Father</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>7,8</td>
<td>∅</td>
</tr>
<tr>
<td>2</td>
<td>3,7</td>
<td>5,8</td>
<td>3,5,7,8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>4,8</td>
</tr>
<tr>
<td>4</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>2,4,6,8</td>
</tr>
<tr>
<td>5</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
</tbody>
</table>
One basic Operation of an interior Node $v$

- Receives from its sons the two sequences $X$ and $Y$.
- Computes: $val_v = \text{merge\_with\_help}(X, Y, val_v)$.
- Sends to its father: $\text{reduce}(val_v)$ till $v$ has sorted all received sequences.
- Sends to its father each second element from $val_v$, if $v$ is done with sorting.
- Sends to its father $val_v$, if $v$ finishes sorting two steps before.

**Example:**

<table>
<thead>
<tr>
<th>Step</th>
<th>Left</th>
<th>Right</th>
<th>$val_v$</th>
<th>Father</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>7,8</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>3,7</td>
<td>5,8</td>
<td>3,5,7,8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>4,8</td>
</tr>
<tr>
<td>4</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>2,4,6,8</td>
</tr>
<tr>
<td>5</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
</tbody>
</table>
One basic Operation of an interior Node $v$

- Receives from its sons the two sequences $X$ and $Y$.
- Computes: $val_v = \textit{merge\_with\_help}(X, Y, val_v)$.
- Sends to its father: $\text{reduce}(val_v)$ till $v$ has sorted all received sequences.
- Sends to its father each second element from $val_v$, if $v$ is done with sorting.
- Sends to its father $val_v$, if $v$ finishes sorting two steps before.
- Example:

<table>
<thead>
<tr>
<th>Step</th>
<th>Left</th>
<th>Right</th>
<th>$val_v$</th>
<th>Father</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>7,8</td>
<td>∅</td>
</tr>
<tr>
<td>2</td>
<td>3,7</td>
<td>5,8</td>
<td>3,5,7,8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>4,8</td>
</tr>
<tr>
<td>4</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>2,4,6,8</td>
</tr>
<tr>
<td>5</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
</tbody>
</table>
One basic Operation of an interior Node $v$

- Receives from its sons the two sequences $X$ and $Y$.
- Computes: $val_v = merge\_with\_help(X, Y, val_v)$.
- Sends to its father: $reduce(val_v)$ till $v$ has sorted all received sequences.
- Sends to its father each second element from $val_v$, if $v$ is done with sorting.
- Sends to its father $val_v$, if $v$ finishes sorting two steps before.

Example:

<table>
<thead>
<tr>
<th>Step</th>
<th>Left</th>
<th>Right</th>
<th>$val_v$</th>
<th>Father</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>7,8</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>2</td>
<td>3,7</td>
<td>5,8</td>
<td>3,5,7,8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>4,8</td>
</tr>
<tr>
<td>4</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>2,4,6,8</td>
</tr>
<tr>
<td>5</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
</tbody>
</table>
One basic Operation of an interior Node $v$

- Receives from its sons the two sequences $X$ and $Y$.
- Computes: $val_v = \text{merge\_with\_help}(X, Y, val_v)$.
- Sends to its father: $\text{reduce}(val_v)$ till $v$ has sorted all received sequences.
- Sends to its father each second element from $val_v$, if $v$ is done with sorting.
- Sends to its father $val_v$, if $v$ finishes sorting two steps before.

Example:

<table>
<thead>
<tr>
<th>Step</th>
<th>Left</th>
<th>Right</th>
<th>$val_v$</th>
<th>Father</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>7,8</td>
<td>⌀</td>
</tr>
<tr>
<td>2</td>
<td>3,7</td>
<td>5,8</td>
<td>3,5,7,8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>4,8</td>
</tr>
<tr>
<td>4</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>2,4,6,8</td>
</tr>
<tr>
<td>5</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
</tbody>
</table>
One basic Operation of an interior Node $v$

- Receives from its sons the two sequences $X$ and $Y$.
- Computes: $val_v = \text{merge\_with\_help}(X, Y, val_v)$.
- Sends to its father: $\text{reduce}(val_v)$ till $v$ has sorted all received sequences.
- Sends to its father each second element from $val_v$, if $v$ is done with sorting.
- Sends to its father $val_v$, if $v$ finishes sorting two steps before.
- Example:

<table>
<thead>
<tr>
<th>Step</th>
<th>Left</th>
<th>Right</th>
<th>$val_v$</th>
<th>Father</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>7,8</td>
<td>∅</td>
</tr>
<tr>
<td>2</td>
<td>3,7</td>
<td>5,8</td>
<td>3,5,7,8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>4,8</td>
</tr>
<tr>
<td>4</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>2,4,6,8</td>
</tr>
<tr>
<td>5</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
</tbody>
</table>
One basic Operation of an interior Node $v$

- Receives from its sons the two sequences $X$ and $Y$.
- Computes: $val_v = merge\_with\_help(X, Y, val_v)$.
- Sends to its father: reduce($val_v$) till $v$ has sorted all received sequences.
- Sends to its father each second element from $val_v$, if $v$ is done with sorting.
- Sends to its father $val_v$, if $v$ finishes sorting two steps before.

Example:

<table>
<thead>
<tr>
<th>Step</th>
<th>Left</th>
<th>Right</th>
<th>$val_v$</th>
<th>Father</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>7,8</td>
<td>∅</td>
</tr>
<tr>
<td>2</td>
<td>3,7</td>
<td>5,8</td>
<td>3,5,7,8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>4,8</td>
</tr>
<tr>
<td>4</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>2,4,6,8</td>
</tr>
<tr>
<td>5</td>
<td>1,3,4,7</td>
<td>2,5,6,8</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
</tbody>
</table>
Basic operation of a interior Node $v$

- Receives from its sons the two sequences $X$ and $Y$.
- Computes: $val_v = \text{merge\_with\_help}(X, Y, val_v)$.
- Sends to its father: $\text{reduce}(val_v)$ till $v$ has sorted all received sequences.
- Sends to its father each second element from $val_v$, if $v$ is done with sorting.
- Sends to its father $val_v$, if $v$ finishes sorting two steps before.
- Thus we get the following pattern:

\[
\begin{array}{cccccccc}
X_1 & X_2 & X_3 & X_4 & \cdots & X_r \\
Z_1 & Z_2 & \cdots & Z_r & Z_{r+1} & Z_{r+2}
\end{array}
\]

- If a node $x$ is finished after $t$ steps, then will the father of $x$ be finished after $t + 3$ steps.
- Thus we get a running time of $3\log n$. 
Basic operation of a interior Node \( v \)

- Receives from its sons the two sequences \( X \) and \( Y \).
- Computes: \( val_v = \text{merge\_with\_help}(X, Y, val_v) \).
- Sends to its father: \( \text{reduce}(val_v) \) till \( v \) has sorted all received sequences.
- Sends to its father each second element from \( val_v \), if \( v \) is done with sorting.
- Sends to its father \( val_v \), if \( v \) finishes sorting two steps before.
- Thus we get the following pattern:
  \[
  X_1 \quad X_2 \quad X_3 \quad X_4 \quad \cdots \quad X_r
  
  Z_1 \quad Z_2 \quad \cdots \quad Z_r \quad Z_{r+1} \quad Z_{r+2}
  \]

- If a node \( x \) is finished after \( t \) steps, then will the father of \( x \) be finished after \( t + 3 \) steps.
- Thus we get a running time of \( 3 \log n \).
Basic operation of a interior Node $v$

- Receives from its sons the two sequences $X$ and $Y$.
- Computes: $val_v = merge\_with\_help(X, Y, val_v)$.
- Sends to its father: $\text{reduce}(val_v)$ till $v$ has sorted all received sequences.
- Sends to its father each second element from $val_v$, if $v$ is done with sorting.
- Sends to its father $val_v$, if $v$ finishes sorting two steps before.
- Thus we get the following pattern:
  
  $X_1 \ X_2 \ X_3 \ X_4 \ \cdots \ X_r$
  
  $Z_1 \ Z_2 \ \cdots \ Z_r \ Z_{r+1} \ Z_{r+2}$

- If a node $x$ is finished after $t$ steps, then will the father of $x$ be finished after $t + 3$ steps.
- Thus we get a running time of $3 \log n$. 
Invariant:

- Each $X_i$ is a good sampler of $X_{i+1}$.
- Each $Y_i$ is a good sampler of $Y_{i+1}$.
- Each $Z_i$ is a good sampler of $Z_{i+1}$.
- Each $X_i$ is half as big as $X_{i+1}$.
- Each $Y_i$ is half as big as $Y_{i+1}$.
- Each $Z_i$ is half as big as $Z_{i+1}$.
- $|X_1| = |Y_1| = |Z_1| = 1$. 

Invariant:

- Each $X_i$ is a good sampler of $X_{i+1}$.
- Each $Y_i$ is a good sampler of $Y_{i+1}$.
- Each $Z_i$ is a good sampler of $Z_{i+1}$.
- Each $X_i$ is half as big as $X_{i+1}$.
- Each $Y_i$ is half as big as $Y_{i+1}$.
- Each $Z_i$ is half as big as $Z_{i+1}$.
- $|X_1| = |Y_1| = |Z_1| = 1$. 
Invariant

- Each $X_i$ is a good sampler of $X_{i+1}$.
- Each $Y_i$ is a good sampler of $Y_{i+1}$.
- Each $Z_i$ is a good sampler of $Z_{i+1}$.
- Each $X_i$ is half as big as $X_{i+1}$.
- Each $Y_i$ is half as big as $Y_{i+1}$.
- Each $Z_i$ is half as big as $Z_{i+1}$.
- $|X_1| = |Y_1| = |Z_1| = 1$. 
Invariant:

- Each $X_i$ is a good sampler of $X_{i+1}$.
- Each $Y_i$ is a good sampler of $Y_{i+1}$.
- Each $Z_i$ is a good sampler of $Z_{i+1}$.
- Each $X_i$ is half as big as $X_{i+1}$.
- Each $Y_i$ is half as big as $Y_{i+1}$.
- Each $Z_i$ is half as big as $Z_{i+1}$.
- $|X_1| = |Y_1| = |Z_1| = 1$. 
Invariant:

- Each $X_i$ is a good sampler of $X_{i+1}$.
- Each $Y_i$ is a good sampler of $Y_{i+1}$.
- Each $Z_i$ is a good sampler of $Z_{i+1}$.
- Each $X_i$ is half as big as $X_{i+1}$.
- Each $Y_i$ is half as big as $Y_{i+1}$.
- Each $Z_i$ is half as big as $Z_{i+1}$.
- $|X_1| = |Y_1| = |Z_1| = 1.$
Invariant:

- Each $X_i$ is a good sampler of $X_{i+1}$.
- Each $Y_i$ is a good sampler of $Y_{i+1}$.
- Each $Z_i$ is a good sampler of $Z_{i+1}$.
- Each $X_i$ is half as big as $X_{i+1}$.
- Each $Y_i$ is half as big as $Y_{i+1}$.
- Each $Z_i$ is half as big as $Z_{i+1}$.
- $|X_1| = |Y_1| = |Z_1| = 1$. 
Invariant:

- Each $X_i$ is a good sampler of $X_{i+1}$.
- Each $Y_i$ is a good sampler of $Y_{i+1}$.
- Each $Z_i$ is a good sampler of $Z_{i+1}$.
- Each $X_i$ is half as big as $X_{i+1}$.
- Each $Y_i$ is half as big as $Y_{i+1}$.
- Each $Z_i$ is half as big as $Z_{i+1}$.
- $|X_1| = |Y_1| = |Z_1| = 1$. 
Situation

- Running time is $O(\log n)$.
- The inner nodes $v$ need $|val_v|$ many processors.
- We still have to proof that the number of processors is in $O(n)$.
- PRAM Model has to be verified.
- Important: The computation of the values $Rng_{X,Y}$ has to be shown.
- These values will be in the following also transmitted and updated.
Running time is $O(\log n)$.

The inner nodes $v$ need $|val_v|$ many processors.

We still have to proof that the number of processors is in $O(n)$.

PRAM Model has to be verified.

Important: The computation of the values $Rng_{X,Y}$ has to be shown.

These values will be in the following also transmitted and updated.
Running time is $O(\log n)$.

The inner nodes $v$ need $|val_v|$ many processors.

We still have to proof that the number of processors is in $O(n)$.

PRAM Model has to be verified.

Important: The computation of the values $Rng_{X,Y}$ has to be shown.

These values will be in the following also transmitted and updated.
Running time is $O(\log n)$.

The inner nodes $v$ need $|val_v|$ many processors.

We still have to proof that the number of processors is in $O(n)$.

PRAM Model has to be verified.

Important: The computation of the values $Rng_{X,Y}$ has to be shown.

These values will be in the following also transmitted and updated.
Situation

- Running time is $O(\log n)$.
- The inner nodes $v$ need $|val_v|$ many processors.
- We still have to proof that the number of processors is in $O(n)$.
- PRAM Model has to be verified.
- Important: The computation of the values $Rng_{X,Y}$ has to be shown.
- These values will be in the following also transmitted and updated.
Running time is $O(\log n)$.

The inner nodes $v$ need $|val_v|$ many processors.

We still have to proof that the number of processors is in $O(n)$.

PRAM Model has to be verified.

Important: The computation of the values $Rng_{X,Y}$ has to be shown.

These values will be in the following also transmitted and updated.
Computing the Ranks

- In each step will compute: \texttt{merge\_with\_help}(X_{i+1}, Y_{i+1}, \text{merge}(X_i, Y_i)).
- Using the Lemma from above we have: \text{merge}(X_i, Y_i) is a good sampler of $X_{i+1}$ and $Y_{i+1}$.
- Let $L = \text{merge}(X_i, Y_i)$, $J = X_{i+1}$ and $K = Y_{i+1}$.
- We have to compute: $\text{Rng}_{L,J}$, $\text{Rng}_{L,K}$, $\text{Rng}_{J,L}$ and $\text{Rng}_{K,L}$.

Invariant:

- Let $S_1, S_2, \ldots, S_p$ be a sequence of sequences at node $v$.
- Then node $c$ also knows: $\text{Rng}_{S_{i+1}, S_i}$ for $1 \leq i < p$.
- Furthermore for each sequence $S$ is known: $\text{Rng}_{S,S}$. 
Computing the Ranks

- In each step will compute: \textit{merge\_with\_help}(X_{i+1}, Y_{i+1}, \text{merge}(X_i, Y_i))

- Using the Lemma from above we have: \text{merge}(X_i, Y_i) is a good sampler of \textit{X}_{i+1} and \textit{Y}_{i+1}

- Let \textit{L} = \text{merge}(X_i, Y_i), J = X_{i+1} and K = Y_{i+1}

- We have to compute: \text{Rng}_{\textit{L}, \textit{J}}, \text{Rng}_{\textit{L}, \textit{K}}, \text{Rng}_{\textit{J}, \textit{L}} and \text{Rng}_{\textit{K}, \textit{L}}

**Invariant:**

- Let \textit{S}_1, \textit{S}_2, \cdots, \textit{S}_p be a sequence of sequences at node \textit{v}

- Then node \textit{c} also knows: \text{Rng}_{\textit{S}_{i+1}, \textit{S}_i} for 1 \leq i < p.

- Furthermore for each sequence \textit{S} is known: \text{Rng}_S, \textit{S}.
Computing the Ranks

- In each step will compute: \( \text{merge\_with\_help}(X_{i+1}, Y_{i+1}, \text{merge}(X_i, Y_i)) \).
- Using the Lemma from above we have: \( \text{merge}(X_i, Y_i) \) is a good sampler of \( X_{i+1} \) and \( Y_{i+1} \).
- Let \( L = \text{merge}(X_i, Y_i) \), \( J = X_{i+1} \) and \( K = Y_{i+1} \).
- We have to compute: \( \text{Rng}_L, J \), \( \text{Rng}_L, K \), \( \text{Rng}_J, L \) and \( \text{Rng}_K, L \).

Invariant:

- Let \( S_1, S_2, \ldots, S_p \) be a sequence of sequences at node \( \nu \).
- Then node \( c \) also knows: \( \text{Rng}_{S_{i+1}, S_i} \) for \( 1 \leq i < p \).
- Furthermore for each sequence \( S \) is known: \( \text{Rng}_S, S \).
Computing the Ranks

- In each step will compute: $\text{merge\_with\_help}(X_{i+1}, Y_{i+1}, \text{merge}(X_i, Y_i))$.
- Using the Lemma from above we have: $\text{merge}(X_i, Y_i)$ is a good sampler of $X_{i+1}$ and $Y_{i+1}$.
- Let $L = \text{merge}(X_i, Y_i)$, $J = X_{i+1}$ and $K = Y_{i+1}$.
- We have to compute: $\text{Rng}_{L,J}$, $\text{Rng}_{L,K}$, $\text{Rng}_{J,L}$ and $\text{Rng}_{K,L}$.

Invariant:

- Let $S_1, S_2, \ldots, S_p$ be a sequence of sequences at node $v$.
- Then node $c$ also knows: $\text{Rng}_{S_{i+1}, S_i}$ for $1 \leq i < p$.
- Furthermore for each sequence $S$ is known: $\text{Rng}_{S,S}$.
Computing the Ranks

- In each step will compute: $merge\_with\_help(X_{i+1}, Y_{i+1}, merge(X_i, Y_i))$.
- Using the Lemma from above we have: $merge(X_i, Y_i)$ is a good sampler of $X_{i+1}$ and $Y_{i+1}$.
- Let $L = merge(X_i, Y_i)$, $J = X_{i+1}$ and $K = Y_{i+1}$.
- We have to compute: $Rng_{L,J}$, $Rng_{L,K}$, $Rng_{J,L}$ and $Rng_{K,L}$.

**Invariant:**

- Let $S_1, S_2, \cdots, S_p$ be a sequence of sequences at node $v$.
- Then node $c$ also knows: $Rng_{S_{i+1}, S_i}$ for $1 \leq i < p$.
- Furthermore for each sequence $S$ is known: $Rng_{S,S}$.
Computing the Ranks

- In each step will compute: $merge\_with\_help(X_{i+1}, Y_{i+1}, merge(X_i, Y_i))$.
- Using the Lemma from above we have: $merge(X_i, Y_i)$ is a good sampler of $X_{i+1}$ and $Y_{i+1}$.
- Let $L = merge(X_i, Y_i)$, $J = X_{i+1}$ and $K = Y_{i+1}$.
- We have to compute: $Rng_{L,J}$, $Rng_{L,K}$, $Rng_{J,L}$ and $Rng_{K,L}$.

Invariant:

- Let $S_1, S_2, \ldots, S_p$ be a sequence of sequences at node $v$.
- Then node $c$ also knows: $Rng_{S_{i+1}, S_i}$ for $1 \leq i < p$.
- Furthermore for each sequence $S$ is known: $Rng_{S,S}$. 
Computing the Ranks

- In each step will compute: \( \text{merge \_with \_help}(X_{i+1}, Y_{i+1}, \text{merge}(X_i, Y_i)) \).
- Using the Lemma from above we have: \( \text{merge}(X_i, Y_i) \) is a good sampler of \( X_{i+1} \) and \( Y_{i+1} \).
- Let \( L = \text{merge}(X_i, Y_i) \), \( J = X_{i+1} \) and \( K = Y_{i+1} \).
- We have to compute: \( \text{Rng}_L,J \), \( \text{Rng}_L,K \), \( \text{Rng}_J,L \) and \( \text{Rng}_K,L \).

Invariant:

- Let \( S_1, S_2, \ldots, S_p \) be a sequence of sequences at node \( v \).
- Then node \( c \) also knows: \( \text{Rng}_{S_{i+1},S_i} \) for \( 1 \leq i < p \).
- Furthermore for each sequence \( S \) is known: \( \text{Rng}_S,S \).
Computing the Ranks

- In each step will compute: \( merge\_with\_help(X_{i+1}, Y_{i+1}, merge(X_i, Y_i)) \).
- Using the Lemma from above we have: \( merge(X_i, Y_i) \) is a good sampler of \( X_{i+1} \) and \( Y_{i+1} \).
- Let \( L = merge(X_i, Y_i), J = X_{i+1} \) and \( K = Y_{i+1} \).
- We have to compute: \( Rng_{L,J}, Rng_{L,K}, Rng_{J,L} \) and \( Rng_{K,L} \).

Invariant:

- Let \( S_1, S_2, \cdots, S_p \) be a sequence of sequences at node \( v \).
- Then node \( c \) also knows: \( Rng_{s_{i+1}, s_i} \) for \( 1 \leq i < p \).
- Furthermore for each sequence \( S \) is known: \( Rng_{S,S} \).
Lemma:

Let $S = (b_1, b_2, \cdots, b_k)$ be a sortierted sequence, then we may compute the rank of $a \in S$ in time $O(1)$ using $k$ processors.

Proof:

- Programm: rng1(a,S)
  for all $P_i$ where $1 \leq i \leq k$ do in parallel
    if $b_i < a \leq b_{i+1}$ then return $i$

- Note, the program has no write-conflicts.
- Note, it could be changed, to avoid read-conflicts.
Computing the Ranks

**Lemma:**
Let \( S = (b_1, b_2, \cdots, b_k) \) be a sorted sequence, then we may compute the rank of \( a \in S \) in time \( O(1) \) using \( k \) processors.

**Proof:**
- **Program:** rng1(a,S)
  - for all \( P_i \) where \( 1 \leq i \leq k \) do in parallel
    - if \( b_i < a \leq b_{i+1} \) then return \( i \)

  - Note, the program has no write-conflicts.
  - Note, it could be changed, to avoid read-conflicts.
Computing the Ranks

Lemma:
Let \( S = (b_1, b_2, \ldots, b_k) \) be a sorted sequence, then we may compute the rank of \( a \in S \) in time \( O(1) \) using \( k \) processors.

Proof:

- **Programm:** \texttt{rng1(a,S)}
  
  for all \( P_i \) where \( 1 \leq i \leq k \) do in parallel
  
  if \( b_i < a \leq b_{i+1} \) then return \( i \)

- Note, the program has no write-conflicts.
- Note, it could be changed, to avoid read-conflicts.
Computing the Ranks

Lemma:
Let \( S = (b_1, b_2, \cdots, b_k) \) be a sorted sequence, then we may compute the rank of \( a \in S \) in time \( O(1) \) using \( k \) processors.

Proof:

- **Program:** \( \text{rng1}(a, S) \)
  
  \[
  \text{for all } P_i \text{ where } 1 \leq i \leq k \text{ do in parallel} \\
  \quad \text{if } b_i < a \leq b_{i+1} \text{ then return } i
  \]

- **Note:** the program has no write-conflicts.

- **Note:** it could be changed, to avoid read-conflicts.
Computing the Ranks

Lemma:
Let \( S = (b_1, b_2, \cdots, b_k) \) be a sorted sequence, then we may compute the rank of \( a \in S \) in time \( O(1) \) using \( k \) processors.

Proof:

- **Programm**: rng1(a,S) for all \( P_i \) where \( 1 \leq i \leq k \) do in parallel
  - if \( b_i < a \leq b_{i+1} \) then return \( i \)

  - Note, the program has no write-conflicts.
  - Note, it could be changed, to avoid read-conflicts.
Computing the Ranks

**Lemma:**

Let $S_1, S_2, S$ be two sorted sequences with $S = \text{merge}(S_1, S_2)$ and $S_1 \cap S_2 = \emptyset$. Then we may compute $\text{rnk}_{S_1, S_2}$ and $\text{rnk}_{S_2, S_1}$ in time $O(1)$ using $O(|S|)$ processors.

**Proof:**

- We do know $\text{rnk}_{S, S}$, $\text{rnk}_{S_1, S_1}$ and $\text{rnk}_{S_2, S_2}$.
- Furthermore we have: $\text{rnk}(a, S_2) = \text{rnk}(a, \text{merge}(S_1, S_2)) - \text{rnk}(a, S_1)$.
- The claim follows directly.
Computing the Ranks

Lemma:
Let $S_1, S_2, S$ be two sorted sequences with $S = \text{merge}(S_1, S_2)$ and $S_1 \cap S_2 = \emptyset$. Then we may compute $\text{rnk}_{S_1, S_2}$ and $\text{rnk}_{S_2, S_1}$ in time $O(1)$ using $O(|S|)$ processors.

Proof:

- We do know $\text{rnk}_{S, S}$, $\text{rnk}_{S_1, S_1}$ and $\text{rnk}_{S_2, S_2}$.
- Furthermore we have: $\text{rnk}(a, S_2) = \text{rnk}(a, \text{merge}(S_1, S_2)) - \text{rnk}(a, S_1)$.
- The claim follows directly.
Computing the Ranks

**Lemma:**

Let $S_1, S_2, S$ be two sorted sequences with $S = \text{merge}(S_1, S_2)$ and $S_1 \cap S_2 = \emptyset$. Then we may compute $\text{Rnk}_{S_1,S_2}$ and $\text{Rnk}_{S_2,S_1}$ in time $O(1)$ using $O(|S|)$ processors.

**Proof:**

- We do know $\text{Rnk}_{S,S}$, $\text{Rnk}_{S_1,S_1}$ and $\text{Rnk}_{S_2,S_2}$.
- Furthermore we have: $\text{rnk}(a, S_2) = \text{rnk}(a, \text{merge}(S_1, S_2)) - \text{rnk}(a, S_1)$.
- The claim follows directly.
Computing the Ranks

Lemma:

Let $S_1, S_2, S$ be two sorted sequences with $S = \text{merge}(S_1, S_2)$ and $S_1 \cap S_2 = \emptyset$. Then we may compute $\text{Rnk}_{S_1, S_2}$ and $\text{Rnk}_{S_2, S_1}$ in time $O(1)$ using $O(|S|)$ processors.

Proof:

- We do know $\text{Rnk}_{S, S}$, $\text{Rnk}_{S_1, S_1}$ and $\text{Rnk}_{S_2, S_2}$.
- Furthermore we have: $\text{rk}(a, S_2) = \text{rk}(a, \text{merge}(S_1, S_2)) - \text{rk}(a, S_1)$.
- The claim follows directly.
**Lemma:**

- Let $X$ be a good sampler of $X'$.
- Let $Y$ be a good sampler of $Y'$.
- Let $U = \text{merge}(X, Y)$.
- Assume $\text{Rnk}_{X',X}$ and $\text{Rnk}_{Y',Y}$ are known.

Then we may compute in time $O(1)$ using $O(|X| + |Y|)$ processors $\text{Rnk}_{X',U}$, $\text{Rnk}_{Y',U}$, $\text{Rnk}_{U,X'}$, and $\text{Rnk}_{U,Y'}$.

**Proof:**

- First we compute $\text{Rnk}_{X',U}$ and $\text{Rnk}_{Y',U}$.
- Then we compute $\text{Rnk}_{X,X'}$ and $\text{Rnk}_{Y,Y'}$.
- Finally we compute $\text{Rnk}_{U,X'}$ and $\text{Rnk}_{U,Y'}$. 

We have $\text{rnk}(a, S)$ and $\text{Rnk}_{S_1,S_2}$ and $\text{Rnk}_{S_2,S_1}$.
Computing the Ranks

Lemma:

1. Let $X$ be a good sampler of $X'$.
2. Let $Y$ be a good sampler of $Y'$.
3. Let $U = \text{merge}(X, Y)$.
4. Assume $\text{Rnk}_{X',X}$ and $\text{Rnk}_{Y',Y}$ are known.

Then we may compute in time $O(1)$ using $O(|X| + |Y|)$ processors $\text{Rnk}_{X',U}$, $\text{Rnk}_{Y',U}$, $\text{Rnk}_{U,X'}$ and $\text{Rnk}_{U,Y'}$.

Proof:

1. First we compute $\text{Rnk}_{X',U}$ and $\text{Rnk}_{Y',U}$.
2. Then we compute $\text{Rnk}_{X,X'}$ and $\text{Rnk}_{Y,Y'}$.
3. Finally we compute $\text{Rnk}_{U,X'}$ and $\text{Rnk}_{U,Y'}$. 
Computing the Ranks

**Lemma:**

- Let $X$ be a good sampler of $X'$.
- Let $Y$ be a good sampler of $Y'$.
- Let $U = \text{merge}(X, Y)$.
- Assume $\text{Rnk}_{X',X}$ and $\text{Rnk}_{Y',Y}$ are known.

Then we may compute in time $O(1)$ using $O(|X| + |Y|)$ processors $\text{Rnk}_{X',U}$, $\text{Rnk}_{Y',U}$, $\text{Rnk}_{U,X'}$ and $\text{Rnk}_{U,Y'}$.

**Proof:**

- First we compute $\text{Rnk}_{X',U}$ and $\text{Rnk}_{Y',U}$.
- Then we compute $\text{Rnk}_{X,X'}$ and $\text{Rnk}_{Y,Y'}$.
- Finally we compute $\text{Rnk}_{U,X'}$ and $\text{Rnk}_{U,Y'}$. 

we have $\text{rk}(a, S)$ and $\text{Rnk}_{S_1, S_2}$ and $\text{Rnk}_{S_2, S_1}$
Computing the Ranks

Lemma:

- Let $X$ be a good sampler of $X'$.
- Let $Y$ be a good sampler of $Y'$.
- Let $U = \text{merge}(X, Y)$.
- Assume $\text{Rnk}_{X', X}$ and $\text{Rnk}_{Y', Y}$ are known.

Then we may compute in time $O(1)$ using $O(|X| + |Y|)$ processors $\text{Rnk}_{X', U}$, $\text{Rnk}_{Y', U}$, $\text{Rnk}_{U, X'}$ and $\text{Rnk}_{U, Y'}$.

Proof:

- First we compute $\text{Rnk}_{X', U}$ and $\text{Rnk}_{Y', U}$.
- Then we compute $\text{Rnk}_{X, X'}$ and $\text{Rnk}_{Y, Y'}$.
- Finally we compute $\text{Rnk}_{U, X'}$ and $\text{Rnk}_{U, Y'}$. 

we have $\text{rnk}(a, S)$ and $\text{Rnk}_{S_1, S_2}$ and $\text{Rnk}_{S_2, S_1}$
Computing the Ranks

we have \( \text{rnk}(a, S) \) and \( \text{Rnk}_{S_1, S_2} \) and \( \text{Rnk}_{S_2, S_1} \)

Lemma:

- Let \( X \) be a good sampler of \( X' \).
- Let \( Y \) be a good sampler of \( Y' \).
- Let \( U = \text{merge}(X, Y) \).
- Assume \( \text{Rnk}_{X', X} \) and \( \text{Rnk}_{Y', Y} \) are known.

Then we may compute in time \( O(1) \) using \( O(|X| + |Y|) \) processors \( \text{Rnk}_{X', U} \), \( \text{Rnk}_{Y', U} \), \( \text{Rnk}_{U, X'} \) and \( \text{Rnk}_{U, Y'} \).

Proof:

- First we compute \( \text{Rnk}_{X', U} \) and \( \text{Rnk}_{Y', U} \).
- Then we compute \( \text{Rnk}_{X, X'} \) and \( \text{Rnk}_{Y, Y'} \).
- Finally we compute \( \text{Rnk}_{U, X'} \) and \( \text{Rnk}_{U, Y'} \).
Computing the Ranks

Lemma:

- Let $X$ be a good sampler of $X'$.
- Let $Y$ be a good sampler of $Y'$.
- Let $U = \text{merge}(X, Y)$.
- Assume $\text{Rnk}_{X',X}$ and $\text{Rnk}_{Y',Y}$ are known.

Then we may compute in time $O(1)$ using $O(|X| + |Y|)$ processors $\text{Rnk}_{X',U}$, $\text{Rnk}_{Y',U}$, $\text{Rnk}_{U,X'}$ and $\text{Rnk}_{U,Y'}$.

Proof:

- First we compute $\text{Rnk}_{X',U}$ and $\text{Rnk}_{Y',U}$.
- Then we compute $\text{Rnk}_{X,X'}$ and $\text{Rnk}_{Y,Y'}$.
- Finally we compute $\text{Rnk}_{U,X'}$ and $\text{Rnk}_{U,Y'}$. 

we have $\text{rnk}(a, S)$ and $\text{Rnk}_{S_1,S_2}$ and $\text{Rnk}_{S_2,S_1}$
Computing the Ranks

**Lemma:**

- Let $X$ be a good sampler of $X'$.
- Let $Y$ be a good sampler of $Y'$.
- Let $U = \text{merge}(X, Y)$.
- Assume $\text{Rnk}_{X',X}$ and $\text{Rnk}_{Y',Y}$ are known.

Then we may compute in time $O(1)$ using $O(|X| + |Y|)$ processors $\text{Rnk}_{X',U}$, $\text{Rnk}_{Y',U}$, $\text{Rnk}_{U,X'}$ and $\text{Rnk}_{U,Y'}$.

**Proof:**

- **First** we compute $\text{Rnk}_{X',U}$ and $\text{Rnk}_{Y',U}$.
- Then we compute $\text{Rnk}_{X,X'}$ and $\text{Rnk}_{Y,Y'}$.
- Finally we compute $\text{Rnk}_{U,X'}$ and $\text{Rnk}_{U,Y'}$.
Computing the Ranks

Lemma:

- Let \( X \) be a good sampler of \( X' \).
- Let \( Y \) be a good sampler of \( Y' \).
- Let \( U = \text{merge}(X, Y) \).
- Assume \( \text{Rnk}_{X',X} \) and \( \text{Rnk}_{Y',Y} \) are known.

Then we may compute in time \( O(1) \) using \( O(|X| + |Y|) \) processors \( \text{Rnk}_{X',U} \), \( \text{Rnk}_{Y',U} \), \( \text{Rnk}_{U,X'} \) and \( \text{Rnk}_{U,Y'} \).

Proof:

- First we compute \( \text{Rnk}_{X',U} \) and \( \text{Rnk}_{Y',U} \).
- Then we compute \( \text{Rnk}_{X,X'} \) and \( \text{Rnk}_{Y,Y'} \).
- Finally we compute \( \text{Rnk}_{U,X'} \) and \( \text{Rnk}_{U,Y'} \).
Lemma:

- Let $X$ be a good sampler of $X'$.
- Let $Y$ be a good sampler of $Y'$.
- Let $U = \text{merge}(X, Y)$.
- Assume $\text{rnk}_{X', X}$ and $\text{rnk}_{Y', Y}$ are known.

Then we may compute in time $O(1)$ using $O(|X| + |Y|)$ processors $\text{rnk}_{X', U}$, $\text{rnk}_{Y', U}$, $\text{rnk}_{U, X'}$ and $\text{rnk}_{U, Y'}$.

Proof:

- First we compute $\text{rnk}_{X', U}$ and $\text{rnk}_{Y', U}$.
- Then we compute $\text{rnk}_{X, X'}$ and $\text{rnk}_{Y, Y'}$.
- Finally we compute $\text{rnk}_{U, X'}$ and $\text{rnk}_{U, Y'}$. 
Computing the Ranks \((\text{Rnk}_{X'}, U)\)

- Let \(X = (a_1, a_2, \cdots, a_k)\).
- Let w.l.o.g. \(a_0 = -\infty\) and \(a_{k+1} = +\infty\).
- Using a good sampler \(X\) we split \(X'\) into \(X'_1, X'_2, \cdots, X'_k, X'_{k+1}\).
- Note: \(\text{Rnk}_{X', X}\) is known.
- Splitting may be done in time \(O(1)\) using \(O(|X|)\) processors.
- Let \(U_i\) be the sequence of elements of \(Y\) which are between \(a_{i-1}\) and \(a_i\).
- Thus we get:

  Programm: \(\text{Rnk}_{X', U}\)
  
  for all \(i\) where \(1 \leq i \leq k + 1\) do in parallel
    
    for all \(x \in X'_i\) do
      
      \(\text{rnk}(x, U) = \text{rnk}(a_{i-1}, U) + \text{rnk}(x, U_i)\)

- Running time \(O(1)\) using \(\sum_{i=1}^{k+1} |U_i|\) processors.
Computing the Ranks (\(\text{Rnk}_{X', U}\))

- Let \(X = (a_1, a_2, \cdots, a_k)\).
- Let w.l.o.g. \(a_0 = -\infty\) and \(a_{k+1} = +\infty\).
- Using a good sampler \(X\) we split \(X'\) into \(X'_1, X'_2, \cdots, X'_k, X'_{k+1}\).
- Note: \(\text{Rnk}_{X', X}\) is known.
- Splitting may be done in time \(O(1)\) using \(O(|X|)\) processors.
- Let \(U_i\) be the sequence of elements of \(Y\) which are between \(a_{i-1}\) and \(a_i\).
- Thus we get:

  Programm: \(\text{Rnk}_{X', U}\)
  
  for all \(i\) where \(1 \leq i \leq k + 1\) do in parallel
  
  for all \(x \in X'_i\) do
  
  \(\text{rnk}(x, U) = \text{rnk}(a_{i-1}, U) + \text{rnk}(x, U_i)\)

  Running time \(O(1)\) using \(\sum_{i=1}^{k+1} |U_i|\) processors.
Computing the Ranks \( \text{Rnk}_{X',U} \)

- Let \( X = (a_1, a_2, \ldots, a_k) \).
- Let w.l.o.g. \( a_0 = -\infty \) and \( a_{k+1} = +\infty \).
- Using a good sampler \( X \) we split \( X' \) into \( X'_1, X'_2, \ldots, X'_k, X'_{k+1} \).
- Note: \( \text{Rnk}_{X',X} \) is known.
- Splitting may be done in time \( O(1) \) using \( O(|X|) \) processors.
- Let \( U_i \) be the sequence of elements of \( Y \) which are between \( a_{i-1} \) and \( a_i \).
- Thus we get:

Programm: \( \text{Rnk}_{X',U} \)

for all \( i \) where \( 1 \leq i \leq k + 1 \) do in parallel

\[ \text{for all } x \in X'_i \text{ do} \]

\[ \text{rnk}(x, U) = \text{rnk}(a_{i-1}, U) + \text{rnk}(x, U_i) \]

- Running time \( O(1) \) using \( \sum_{i=1}^{k+1} |U_i| \) processors.
Computing the Ranks \( \text{Rnk}_{X', U} \)

- Let \( X = (a_1, a_2, \cdots, a_k) \).
- Let w.l.o.g. \( a_0 = -\infty \) and \( a_{k+1} = +\infty \).
- Using a good sampler \( X \) we split \( X' \) into \( X'_1, X'_2, \cdots, X'_k, X'_{k+1} \).
- Note: \( \text{Rnk}_{X', X} \) is known.
- Splitting may be done in time \( O(1) \) using \( O(|X|) \) processors.
- Let \( U_i \) be the sequence of elements of \( Y \) which are between \( a_{i-1} \) and \( a_i \).
- Thus we get:

  Programm: \( \text{Rnk}_{X', U} \)
  for all \( i \) where \( 1 \leq i \leq k + 1 \) do in parallel
    for all \( x \in X'_i \) do
      \( \text{rnk}(x, U) = \text{rnk}(a_{i-1}, U) + \text{rnk}(x, U_i) \)

- Running time \( O(1) \) using \( \sum_{i=1}^{k+1} |U_i| \) processors.
Computing the Ranks \( \text{Rnk}_{X', U} \)

- Let \( X = (a_1, a_2, \cdots, a_k) \).
- Let w.l.o.g. \( a_0 = -\infty \) and \( a_{k+1} = +\infty \).
- Using a good sampler \( X \) we split \( X' \) into \( X'_1, X'_2, \cdots, X'_k, X'_{k+1} \).
- Note: \( \text{Rnk}_{X', X} \) is known.
- Splitting may be done in time \( O(1) \) using \( O(|X|) \) processors.
- Let \( U_i \) be the sequence of elements of \( Y \) which are between \( a_{i-1} \) and \( a_i \).
- Thus we get:

  
  Programm: \( \text{Rnk}_{X', U} \)
  
  for all \( i \) where \( 1 \leq i \leq k + 1 \) do in parallel
    
    for all \( x \in X'_i \) do
      
      \( \text{rnk}(x, U) = \text{rnk}(a_{i-1}, U) + \text{rnk}(x, U_i) \)
  
  
- Running time \( O(1) \) using \( \sum_{i=1}^{k+1} |U_i| \) processors.
Computing the Ranks \((\text{Rnk}_{X'},U)\)

- Let \(X = (a_1, a_2, \cdots, a_k)\).
- Let w.l.o.g. \(a_0 = -\infty\) and \(a_{k+1} = +\infty\).
- Using a good sampler \(X\) we split \(X'\) into \(X'_1, X'_2, \cdots, X'_k, X'_{k+1}\).
- Note: \(\text{Rnk}_{X',X}\) is known.
- Splitting may be done in time \(O(1)\) using \(O(|X|)\) processors.
- Let \(U_i\) be the sequence of elements of \(Y\) which are between \(a_{i-1}\) and \(a_i\).
- Thus we get:

Programm: \(\text{Rnk}_{X',U}\)

\[
\text{for all } i \text{ where } 1 \leq i \leq k + 1 \text{ do in parallel} \\
\text{for all } x \in X'_i \text{ do} \\
\quad \text{rnk}(x, U) = \text{rnk}(a_{i-1}, U) + \text{rnk}(x, U_i)
\]

- Running time \(O(1)\) using \(\sum_{i=1}^{k+1} |U_i|\) processors.
Computing the Ranks \((\text{Rnk}_{X', U})\)

- Let \(X = (a_1, a_2, \cdots, a_k)\).
- Let w.l.o.g. \(a_0 = -\infty\) and \(a_{k+1} = +\infty\).
- Using a good sampler \(X\) we split \(X'\) into \(X_1', X_2', \cdots, X_k', X_{k+1}'\).
- Note: \(\text{Rnk}_{X', X}\) is known.
- Splitting may be done in time \(O(1)\) using \(O(|X|)\) processors.
- Let \(U_i\) be the sequence of elements of \(Y\) which are between \(a_{i-1}\) and \(a_i\).
- Thus we get:
  
  **Programm: \(\text{Rnk}_{X', U}\)**

  for all \(i\) where \(1 \leq i \leq k + 1\) do in parallel
  
  for all \(x \in X_i'\) do
    \[
    \text{rnk}(x, U) = \text{rnk}(a_{i-1}, U) + \text{rnk}(x, U_i)
    \]

- Running time \(O(1)\) using \(\sum_{i=1}^{k+1} |U_i|\) processors.

We have \(\text{rnk}(a, S)\) and \(\text{Rnk}_{S_1, S_2}\) and \(\text{Rnk}_{S_2, S_1}\)
Computing the Ranks ($\text{Rnk}_{X',U}$)

- Let $X = (a_1, a_2, \cdots, a_k)$.
- Let w.l.o.g. $a_0 = -\infty$ and $a_{k+1} = +\infty$.
- Using a good sampler $X$ we split $X'$ into $X'_1, X'_2, \cdots, X'_k, X'_{k+1}$.
- Note: $\text{Rnk}_{X',X}$ is known.
- Splitting may be done in time $O(1)$ using $O(|X|)$ processors.
- Let $U_i$ be the sequence of elements of $Y$ which are between $a_{i-1}$ and $a_i$.
- Thus we get:

Programm: $\text{Rnk}_{X',U}$

for all $i$ where $1 \leq i \leq k + 1$ do in parallel
  for all $x \in X'_i$ do
    $\text{rnk}(x, U) = \text{rnk}(a_{i-1}, U) + \text{rnk}(x, U_i)$

- Running time $O(1)$ using $\sum_{i=1}^{k+1} |U_i|$ processors.
Computing the Ranks \((Rnk_{X',U})\)

- Let \(X = (a_1, a_2, \ldots, a_k)\).
- Let \(w.l.o.g.\) \(a_0 = -\infty\) and \(a_{k+1} = +\infty\).
- Using a good sampler \(X\) we split \(X'\) into \(X'_1, X'_2, \ldots, X'_k, X'_{k+1}\).
- Note: \(Rnk_{X',X}\) is known.
- Splitting may be done in time \(O(1)\) using \(O(|X|)\) processors.
- Let \(U_i\) be the sequence of elements of \(Y\) which are between \(a_{i-1}\) and \(a_i\).
- Thus we get:

  \[
  \text{Programm: } Rnk_{X',U} \\
  \text{for all } i \text{ where } 1 \leq i \leq k + 1 \text{ do in parallel} \\
  \quad \text{for all } x \in X'_i \text{ do} \\
  \quad \quad \text{rnk}(x, U) = \text{rnk}(a_{i-1}, U) + \text{rnk}(x, U_i) \\
  \]

- Running time \(O(1)\) using \(\sum_{i=1}^{k+1} |U_i|\) processors.
Computing the Ranks \( (Rnk_{X,X'}) \)

- Let \( a_i \in X \).
- Let \( a' \) minimal element in \( X'_{i+1} \).
- The rank of \( a_i \) in \( X' \) is the same as the rank of \( a' \) in \( X' \).
- This rank is already known.
- This may be computed in time \( O(1) \) using one processor.
Computing the Ranks \((\text{Rnk}_{X, X'})\)

- Let \(a_i \in X\).
- Let \(a'\) minimal element in \(X_{i+1}'\).
- The rank of \(a_i\) in \(X'\) is the same as the rank of \(a'\) in \(X'\).
- This rank is already known.
- This may be computed in time \(O(1)\) using one processor.

\[\text{we have rnk}(a, S) \text{ and Rnk}_{S_1, S_2} \text{ and Rnk}_{S_2, S_1}\]
Computing the Ranks \((\text{Rnk}_X, X')\)

- Let \(a_i \in X\).
- Let \(a'\) minimal element in \(X_{i+1}'\).
- The rank of \(a_i\) in \(X'\) is the same as the rank of \(a'\) in \(X'\).
- This rank is already known.
- This may be computed in time \(O(1)\) using one processor.

we have \(\text{rnk}(a, S)\) and \(\text{Rnk}_{S_1, S_2}\) and \(\text{Rnk}_{S_2, S_1}\)
Computing the Ranks \((\text{Rnk}_{X,X'})\)

- Let \(a_i \in X\).
- Let \(a'\) minimal element in \(X_{i+1}'\).
- The rank of \(a_i\) in \(X'\) is the same as the rank of \(a'\) in \(X'\).
- This rank is already known.
- This may be computed in time \(O(1)\) using one processor.
Computing the Ranks ($\text{Rnk}_{X,X'}$)

- Let $a_i \in X$.
- Let $a'$ minimal element in $X'_{i+1}$.
- The rank of $a_i$ in $X'$ is the same as the rank of $a'$ in $X'$.
- This rank is already known.
- This may be computed in time $O(1)$ using one processor.
Computing the Ranks ($\text{Rnk}_{U,X'}$)

- **Note:** $\text{Rnk}_{U,X'}$ consists of $\text{Rnk} X, X'$ and $\text{Rnk} Y, X'$.
- $\text{Rnk} X, X'$ is already known.
- Still to compute: $\text{Rnk} Y, X'$.
- $\text{Rnk} Y, X$ may be computed using the previous lemma.
- We compute $\text{rnk}(a, X')$ using $\text{rnk}(a, X)$ and $\text{Rnk}_{X,X'}$.
- Thus we compute $\text{Rnk}_{U,X'}$ with $O(|U|)$ processors and time $O(1)$.
Computing the Ranks ($\text{Rnk}_{U,X'}$)

- **Note:** $\text{Rnk}_{U,X'}$ consists of $\text{Rnk} \; X, X'$ and $\text{Rnk} \; Y, X'$.
- $\text{Rnk} \; X, X'$ is already known.
- Still to compute: $\text{Rnk} \; Y, X'$.
- $\text{Rnk} \; Y, X$ may be computed using the previous lemma.
- We compute $\text{rnk}(a, X')$ using $\text{rnk}(a, X)$ and $\text{Rnk}_{X,X'}$.
- Thus we compute $\text{Rnk}_{U,X'}$ with $O(|U|)$ processors and time $O(1)$.
Computing the Ranks \((\text{Rnk}_{U,X'})\)

- Note: \(\text{Rnk}_{U,X'}\) consists of \(\text{Rnk} X, X'\) and \(\text{Rnk} Y, X'\).
- \(\text{Rnk} X, X'\) is already known.
- **Still to compute:** \(\text{Rnk} Y, X'\).
- \(\text{Rnk} Y, X\) may be computed using the previous lemma.
- We compute \(\text{rnk}(a, X')\) using \(\text{rnk}(a, X)\) and \(\text{Rnk}_{X,X'}\).
- Thus we compute \(\text{Rnk}_{U,X'}\) with \(O(|U|)\) processors and time \(O(1)\).
Computing the Ranks \((\text{Rnk}_{U,X'})\)

- Note: \(\text{Rnk}_{U,X'}\) consists of \(\text{Rnk } X, X'\) and \(\text{Rnk } Y, X'\).
- \(\text{Rnk } X, X'\) is already known.
- Still to compute: \(\text{Rnk } Y, X'\).
- \(\text{Rnk } Y, X\) may be computed using the previous lemma.
- We compute \(\text{rnk}(a, X')\) using \(\text{rnk}(a, X)\) and \(\text{Rnk}_{X,X'}\).
- Thus we compute \(\text{Rnk}_{U,X'}\) with \(O(|U|)\) processors and time \(O(1)\).
Computing the Ranks \((\text{Rnk}_{U,X'})\)

- **Note:** \(\text{Rnk}_{U,X'}\) consists of \(\text{Rnk} X, X'\) and \(\text{Rnk} Y, X'\).
- \(\text{Rnk} X, X'\) is already known.
- Still to compute: \(\text{Rnk} Y, X'\).
- \(\text{Rnk} Y, X\) may be computed using the previous lemma.
- **We compute** \(\text{rnk}(a, X')\) using \(\text{rnk}(a, X)\) and \(\text{Rnk}_{X,X'}\).
- Thus we compute \(\text{Rnk}_{U,X'}\) with \(O(|U|)\) processors and time \(O(1)\).
Computing the Ranks \((\text{Rnk}_{U,X'})\)

- Note: \(\text{Rnk}_{U,X'}\) consists of \(\text{Rnk } X, X'\) and \(\text{Rnk } Y, X'\).
- \(\text{Rnk } X, X'\) is already known.
- Still to compute: \(\text{Rnk } Y, X'\).
- \(\text{Rnk } Y, X\) may be computed using the previous lemma.
- We compute \(\text{rnk}(a, X')\) using \(\text{rnk}(a, X)\) and \(\text{Rnk}_{X,X'}\).
- Thus we compute \(\text{Rnk}_{U,X'}\) with \(O(|U|)\) processors and time \(O(1)\).
Computing the Ranks

Consider the step

\[ \text{merge\_with\_help}(J = X_{i+1}, K = Y_{i+1}, L = \text{merge}(X_i, Y_i)) \]

Using the invariant we know: \( \text{Rnk}_{J, X_i} \) and \( \text{Rnk}_{K, Y_i} \).

Using the above considerations we may compute: \( \text{Rnk}_{L, J} \), \( \text{Rnk}_{L, K} \), \( \text{Rnk}_{J, L} \) and \( \text{Rnk}_{K, L} \).

Still to be computed: \( \text{Rnk}_{\text{reduce}(\text{merge}(X_{i+1}, Y_{i+1})), \text{reduce}(\text{merge}(X_i, Y_i))} \)

Known: \( \text{Rnk}_{X_{i+1}, \text{merge}(X_i, Y_i)} \) and \( \text{Rnk}_{Y_{i+1}, \text{merge}(X_i, Y_i)} \).

It is now easy to compute: \( \text{Rnk}_{X_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \) and \( \text{Rnk}_{Y_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \).

Also easy to compute: \( \text{Rnk}_{\text{merge}(X_{i+1}, Y_{i+1}), \text{reduce}(\text{merge}(X_i, Y_i))} \).
Computing the Ranks

- Consider the step
  \[ merge\_with\_help(J = X_{i+1}, K = Y_{i+1}, L = \text{merge}(X_i, Y_i)) \]

- Using the invariant we know: \( \text{Rnk}_{J,X_i} \) and \( \text{Rnk}_{K,Y_i} \).

- Using the above considerations we may compute: \( \text{Rnk}_{L,J} \), \( \text{Rnk}_{L,K} \), \( \text{Rnk}_{J,L} \) and \( \text{Rnk}_{K,L} \).

- Still to be computed: \( \text{Rnk}_{\text{reduce}(\text{merge}(X_{i+1}, Y_{i+1})), \text{reduce}(\text{merge}(X_i, Y_i))} \)

- Known: \( \text{Rnk}_{X_{i+1}, \text{merge}(X_i, Y_i)} \) and \( \text{Rnk}_{Y_{i+1}, \text{merge}(X_i, Y_i)} \).

- It is now easy to compute: \( \text{Rnk}_{X_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \) and \( \text{Rnk}_{Y_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \).

- Also easy to compute: \( \text{Rnk}_{\text{merge}(X_{i+1}, Y_{i+1}), \text{reduce}(\text{merge}(X_i, Y_i))} \).

we have \( \text{rnk}(a, S) \) and \( \text{Rnk}_{S_1,S_2} \) and \( \text{Rnk}_{S_2,S_1} \)
Computing the Ranks

- Consider the step
  \( \text{merge\_with\_help}(J = X_{i+1}, K = Y_{i+1}, L = \text{merge}(X_i, Y_i)) \):

- Using the invariant we know: \( \text{Rnk}_{J,X_i} \) and \( \text{Rnk}_{K,Y_i} \).

- Using the above considerations we may compute: \( \text{Rnk}_{L,J} \), \( \text{Rnk}_{L,K} \), \( \text{Rnk}_{J,L} \), and \( \text{Rnk}_{K,L} \).

- Still to be computed: \( \text{Rnk}_{\text{reduce}(\text{merge}(X_{i+1}, Y_{i+1})), \text{reduce}(\text{merge}(X_i, Y_i))} \)

- Known: \( \text{Rnk}_{X_{i+1}, \text{merge}(X_i, Y_i)} \) and \( \text{Rnk}_{Y_{i+1}, \text{merge}(X_i, Y_i)} \).

- It is now easy to compute: \( \text{Rnk}_{X_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \) and \( \text{Rnk}_{Y_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \).

- Also easy to compute: \( \text{Rnk}_{\text{merge}(X_{i+1}, Y_{i+1}), \text{reduce}(\text{merge}(X_i, Y_i))} \).
Computing the Ranks

- Consider the step
  \[ \text{merge}_\text{with}_\text{help}(J = X_{i+1}, K = Y_{i+1}, L = \text{merge}(X_i, Y_i)) \]
- Using the invariant we know: \( \text{Rnk}_{J, X_i} \) and \( \text{Rnk}_{K, Y_i} \).
- Using the above considerations we may compute: \( \text{Rnk}_{L, J}, \text{Rnk}_{L, K}, \text{Rnk}_{J, L} \) and \( \text{Rnk}_{K, L} \).
- Still to be computed: \( \text{Rnk}_{\text{reduce}(\text{merge}(X_{i+1}, Y_{i+1})), \text{reduce}(\text{merge}(X_i, Y_i))} \)
- Known: \( \text{Rnk}_{X_{i+1}, \text{merge}(X_i, Y_i)} \) and \( \text{Rnk}_{Y_{i+1}, \text{merge}(X_i, Y_i)} \).
- It is now easy to compute: \( \text{Rnk}_{X_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \) and \( \text{Rnk}_{Y_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \).
- Also easy to compute: \( \text{Rnk}_{\text{merge}(X_{i+1}, Y_{i+1}), \text{reduce}(\text{merge}(X_i, Y_i))} \).
Computing the Ranks

we have \( \text{rnk}(a, S) \) and \( \text{Rnk}_{S_1, S_2} \) and \( \text{Rnk}_{S_2, S_1} \)

- Consider the step
  \[ \text{merge\_with\_help}(J = X_{i+1}, K = Y_{i+1}, L = \text{merge}(X_i, Y_i)) \]

- Using the invariant we know: \( \text{Rnk}_{J, X_i} \) and \( \text{Rnk}_{K, Y_i} \).

- Using the above considerations we may compute: \( \text{Rnk}_{L, J} \), \( \text{Rnk}_{L, K} \), \( \text{Rnk}_{J, L} \) and \( \text{Rnk}_{K, L} \).

- Still to be computed: \( \text{Rnk}_{\text{reduce}(\text{merge}(X_{i+1}, Y_{i+1})), \text{reduce}(\text{merge}(X_i, Y_i))} \)

- Known: \( \text{Rnk}_{X_{i+1}, \text{merge}(X_i, Y_i)} \) and \( \text{Rnk}_{Y_{i+1}, \text{merge}(X_i, Y_i)} \).

- It is now easy to compute: \( \text{Rnk}_{X_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \) and \( \text{Rnk}_{Y_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \).

- Also easy to compute: \( \text{Rnk}_{\text{merge}(X_{i+1}, Y_{i+1}), \text{reduce}(\text{merge}(X_i, Y_i))} \).
Computing the Ranks

we have \( \text{rnk}(a, S) \) and \( \text{Rnk}_{S_1, S_2} \) and \( \text{Rnk}_{S_2, S_1} \)

- Consider the step
  \[
  \text{merge\_with\_help}(J = X_{i+1}, K = Y_{i+1}, L = \text{merge}(X_i, Y_i)):
  \]
- Using the invariant we know: \( \text{Rnk}_{J,X_i} \) and \( \text{Rnk}_{K,Y_i} \).
- Using the above considerations we may compute: \( \text{Rnk}_{L,J} \), \( \text{Rnk}_{L,K} \), \( \text{Rnk}_{J,L} \)
  and \( \text{Rnk}_{K,L} \).
- Still to be computed: \( \text{Rnk}_{\text{reduce}(\text{merge}(X_{i+1}, Y_{i+1})), \text{reduce}(\text{merge}(X_i, Y_i))} \)
- Known: \( \text{Rnk}_{X_{i+1}, \text{merge}(X_i, Y_i)} \) and \( \text{Rnk}_{Y_{i+1}, \text{merge}(X_i, Y_i)} \).
- It is now easy to compute: \( \text{Rnk}_{X_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \) and \( \text{Rnk}_{Y_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \).
- Also easy to compute: \( \text{Rnk}_{\text{merge}(X_{i+1}, Y_{i+1}), \text{reduce}(\text{merge}(X_i, Y_i))} \).
Computing the Ranks

Consider the step
\[ \text{merge\_with\_help}(J = X_{i+1}, K = Y_{i+1}, L = \text{merge}(X_i, Y_i)) \]

Using the invariant we know: \( \text{Rnk}_{J, X_i} \) and \( \text{Rnk}_{K, Y_i} \).

Using the above considerations we may compute: \( \text{Rnk}_{L, J}, \text{Rnk}_{L, K}, \text{Rnk}_{J, L} \) and \( \text{Rnk}_{K, L} \).

Still to be computed: \( \text{Rnk}_{\text{reduce}(\text{merge}(X_{i+1}, Y_{i+1})), \text{reduce}(\text{merge}(X_i, Y_i))} \)

Known: \( \text{Rnk}_{X_{i+1}, \text{merge}(X_i, Y_i)} \) and \( \text{Rnk}_{Y_{i+1}, \text{merge}(X_i, Y_i)} \).

It is now easy to compute: \( \text{Rnk}_{X_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \) and \( \text{Rnk}_{Y_{i+1}, \text{reduce}(\text{merge}(X_i, Y_i))} \).

Also easy to compute: \( \text{Rnk}_{\text{merge}(X_{i+1}, Y_{i+1}), \text{reduce}(\text{merge}(X_i, Y_i))} \).
Algorithmn of Cole

Theorem:
We may sort $n$ values on a CREW PRAM using $O(n)$ processors in time $O(\log n)$.

Proof: discussed before.

Theorem:
We may sort $n$ values on a EREW PRAM using $O(n)$ processors in time $O(\log n)$.

Proof: see literature.

Theorem:
There exists a sorting network with $O(n)$ processors and depth $O(\log n)$.

Proof: see literature.
Theorem:
We may sort $n$ values on a CREW PRAM using $O(n)$ processors in time $O(\log n)$. 

Proof: discussed before.

Theorem:
We may sort $n$ values on a EREW PRAM using $O(n)$ processors in time $O(\log n)$. 

Proof: see literature.

Theorem:
There exists a sorting network with $O(n)$ processors and depth $O(\log n)$. 

Proof: see literature.
Algorithm of Cole

Theorem:
We may sort $n$ values on a CREW PRAM using $O(n)$ processors in time $O(\log n)$.

Proof: discussed before.

Theorem:
We may sort $n$ values on a EREW PRAM using $O(n)$ processors in time $O(\log n)$.

Proof: see literature.

Theorem:
There exists a sorting network with $O(n)$ processors and depth $O(\log n)$.

Proof: see literature.
Algorithm of Cole

Theorem:
We may sort $n$ values on a CREW PRAM using $O(n)$ processors in time $O(\log n)$.

Proof: discussed before.

Theorem:
We may sort $n$ values on a EREW PRAM using $O(n)$ processors in time $O(\log n)$.

Proof: see literature.

Theorem:
There exists a sorting network with $O(n)$ processors and depth $O(\log n)$.

Proof: see literature.
Algorithm of Cole

Theorem:
We may sort $n$ values on a CREW PRAM using $O(n)$ processors in time $O(\log n)$.

Proof: discussed before.

Theorem:
We may sort $n$ values on a EREW PRAM using $O(n)$ processors in time $O(\log n)$.

Proof: see literature.

Theorem:
There exists a sorting network with $O(n)$ processors and depth $O(\log n)$.

Proof: see literature.
Algorithmn of Cole

Theorem:

We may sort \( n \) values on a CREW PRAM using \( O(n) \) processors in time \( O(\log n) \).

Proof: discussed before.

Theorem:

We may sort \( n \) values on a EREW PRAM using \( O(n) \) processors in time \( O(\log n) \).

Proof: see literature.

Theorem:

There exists a sorting network with \( O(n) \) processors and depth \( O(\log n) \).

Proof: see literature.
we have \( \text{rnk}(a, S) \) and \( \text{rnk}_{S_1, S_2} \) and \( \text{rnk}_{S_2, S_1} \)

Literatur:

Questions

- Explain the motivation behind parallel systems.
- Explain the ideas of the different sorting algorithms.
- Explain the different running times of these sorting algorithms.
- Explain the different efficiency of these sorting algorithms.
- Explain the idea of the algorithm of Cole.
- Explain the running time of the algorithm of Cole.
- Explain the number of processors used in the algorithm of Cole.
Questions

- Explain the motivation behind parallel systems.
- Explain the ideas of the different sorting algorithms.
- Explain the different running times of these sorting algorithms.
- Explain the different efficiency of these sorting algorithms.
- Explain the idea of the algorithm of Cole.
- Explain the running time of the algorithm of Cole.
- Explain the number of processors used in the algorithm of Cole.
Questions

- Explain the motivation behind parallel systems.
- Explain the ideas of the different sorting algorithms.
- Explain the different running times of these sorting algorithms.
- Explain the different efficiency of these sorting algorithms.
- Explain the idea of the algorithm of Cole.
- Explain the running time of the algorithm of Cole.
- Explain the number of processors used in the algorithm of Cole.
Questions

- Explain the motivation behind parallel systems.
- Explain the ideas of the different sorting algorithms.
- Explain the different running times of these sorting algorithms.
- Explain the different efficiency of these sorting algorithms.
- Explain the idea of the algorithm of Cole.
- Explain the running time of the algorithm of Cole.
- Explain the number of processors used in the algorithm of Cole.
Questions

- Explain the motivation behind parallel systems.
- Explain the ideas of the different sorting algorithms.
- Explain the different running times of these sorting algorithms.
- Explain the different efficiency of these sorting algorithms.
- Explain the idea of the algorithm of Cole.
- Explain the running time of the algorithm of Cole.
- Explain the number of processors used in the algorithm of Cole.
Questions

- Explain the motivation behind parallel systems.
- Explain the ideas of the different sorting algorithms.
- Explain the different running times of these sorting algorithms.
- Explain the different efficiency of these sorting algorithms.
- Explain the idea of the algorithm of Cole.
- **Explain the running time of the algorithm of Cole.**
- Explain the number of processors used in the algorithm of Cole.
Questions

- Explain the motivation behind parallel systems.
- Explain the ideas of the different sorting algorithms.
- Explain the different running times of these sorting algorithms.
- Explain the different efficiency of these sorting algorithms.
- Explain the idea of the algorithm of Cole.
- Explain the running time of the algorithm of Cole.
- Explain the number of processors used in the algorithm of Cole.
Legend

- Not of relevance
- Implicitly used basics
- Idea of proof or algorithm
- Structure of proof or algorithm
- Full knowledge