COL380

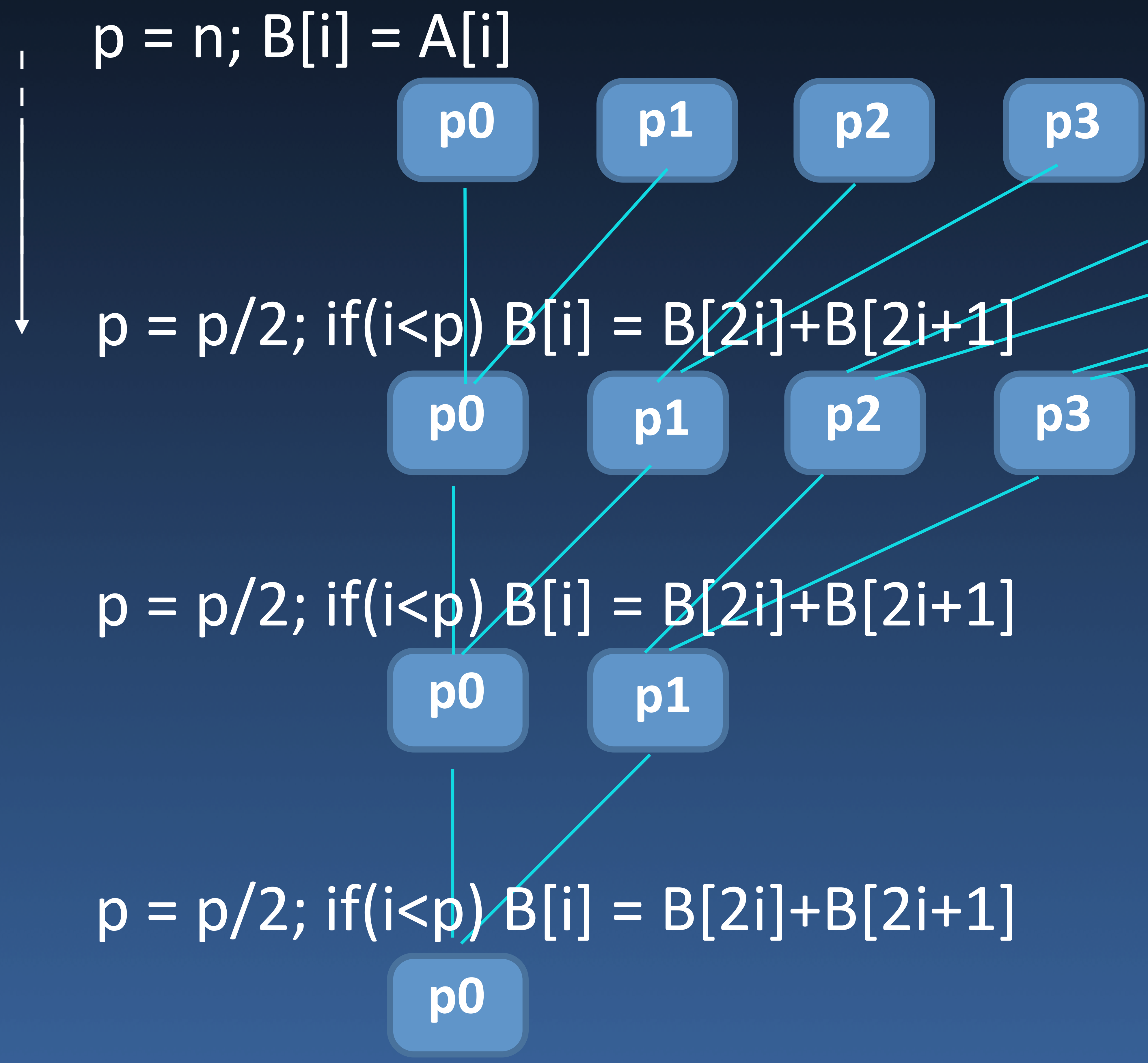
Introduction to

Parallel & Distributed Programming

Parallel Algorithms

- Maximize concurrency
  - ➡ Reduce dependency
    - ▶ OK to sometime recompute data
- Map tasks to processors
  - ➡ Statically or Dynamically
  - ➡ Reduce communication

Parallel Addition


p = n; B[i] = A[i]
p0
p1
p2
p3
p4
p5
p6
p7
p = p/2; if(i<p) B[i] = B[2i]+B[2i+1]
p0
p1
p2
p3
p = p/2; if(i<p) B[i] = B[2i]+B[2i+1]
p0
p1
p = p/2; if(i<p) B[i] = B[2i]+B[2i+1]
p0

```
p = n/2
forall i < n
    B[i] = A[i]
while(p > 0) {
    forall i < p
        B[i] = B[2i]+B[2i+1]
    p = p/2;
}
```

(assumes n is a power of 2)

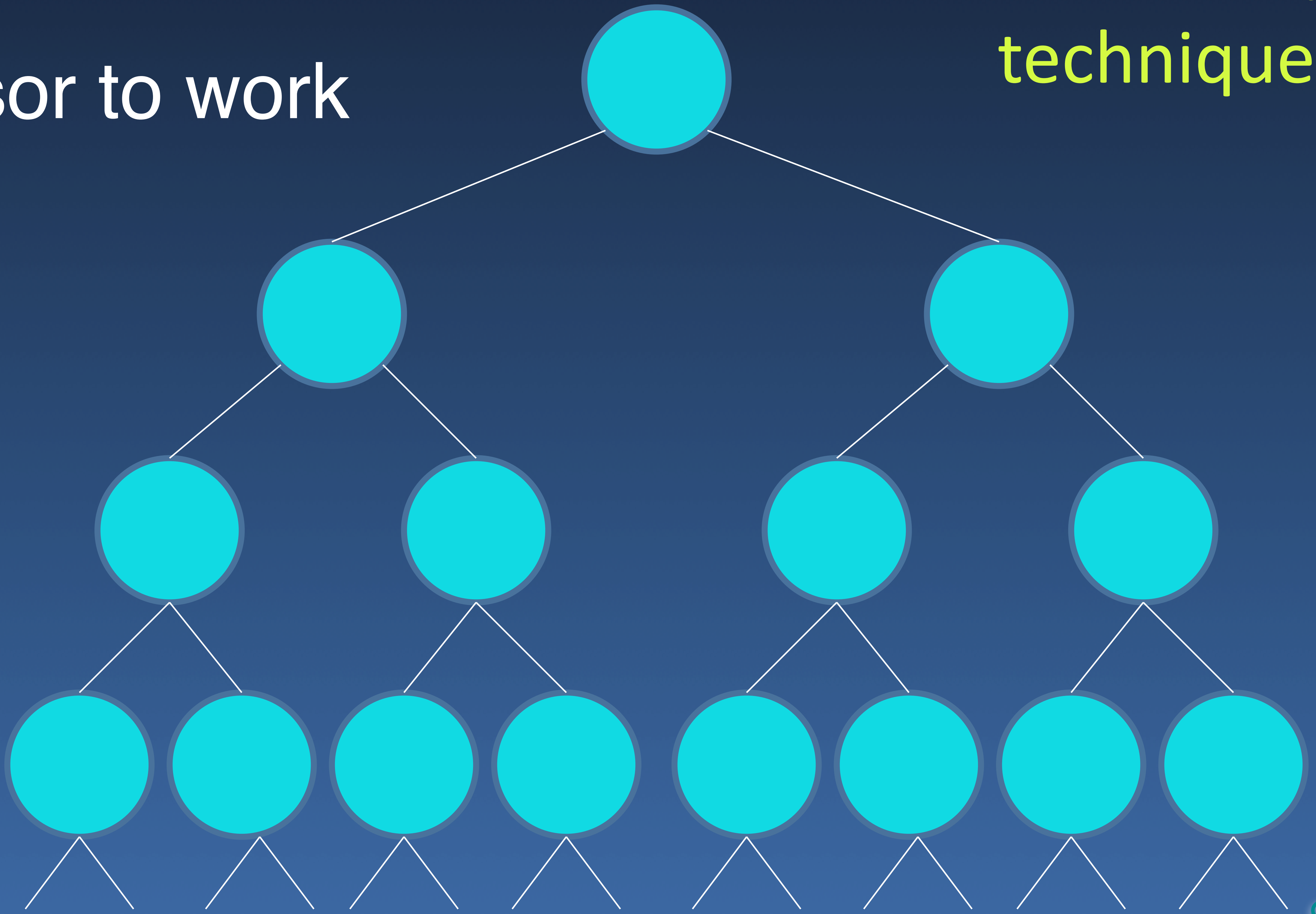
- processors: n
- time: O(log n)
- Speed-up: n/(log n)
- Efficiency: 1/log(n)
- Cost: n log n
- Work: n

Subodh Kumar

Reduction

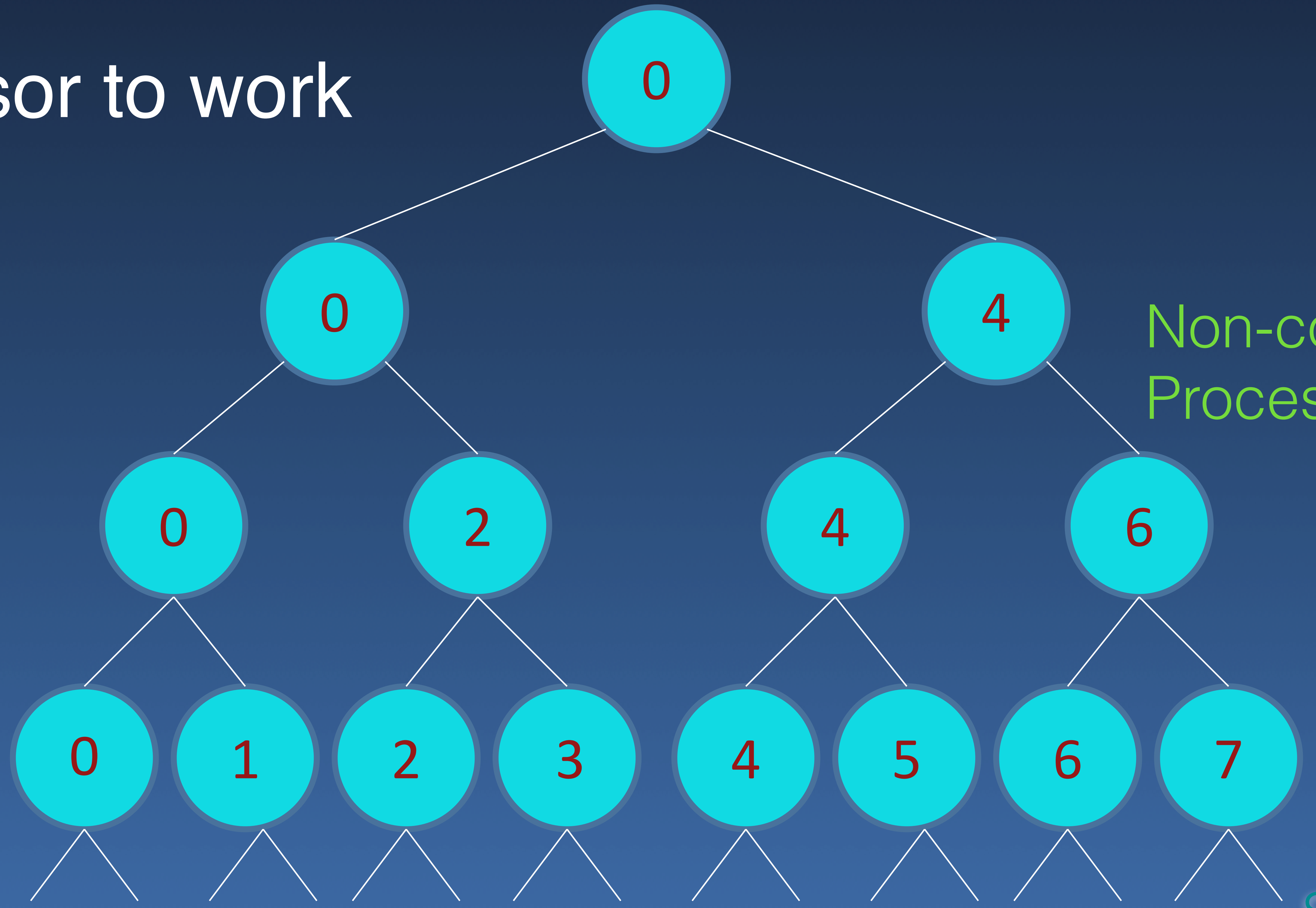
- n operands ⇒ log n steps
- Total work = O(n)
- How do you map processor to work

Balanced Binary Tree technique



Subodh Kumar

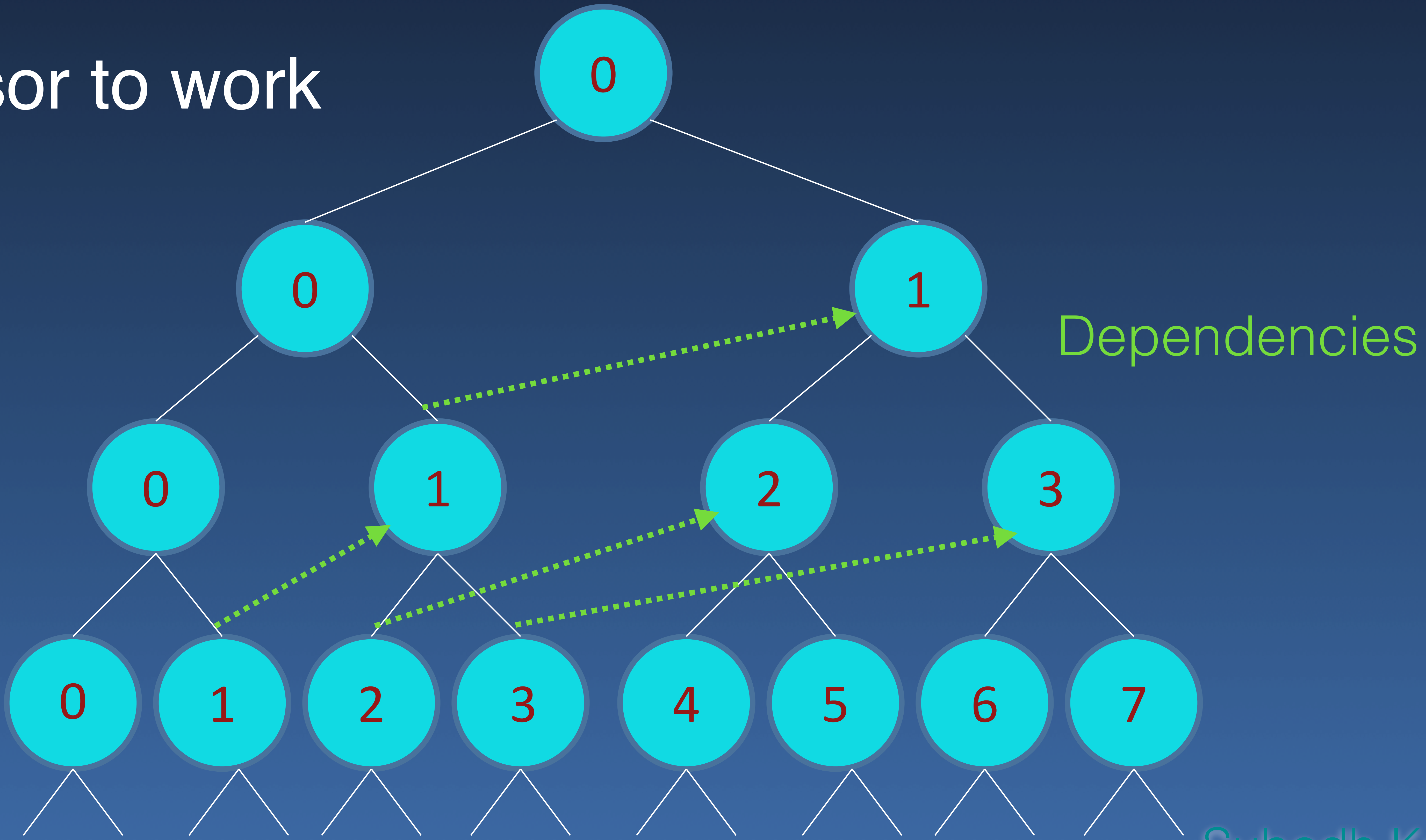
- n operands ⇒ log n steps
- Total work = O(n)
- How do you map processor to work
  - ➡ $n/2^i$ processors per step
  - ➡ step i: if $!(id\%2^i)$
    - Read: id, $id+2^{i-1}$
    - Write: id



Non-contiguous Processor IDs

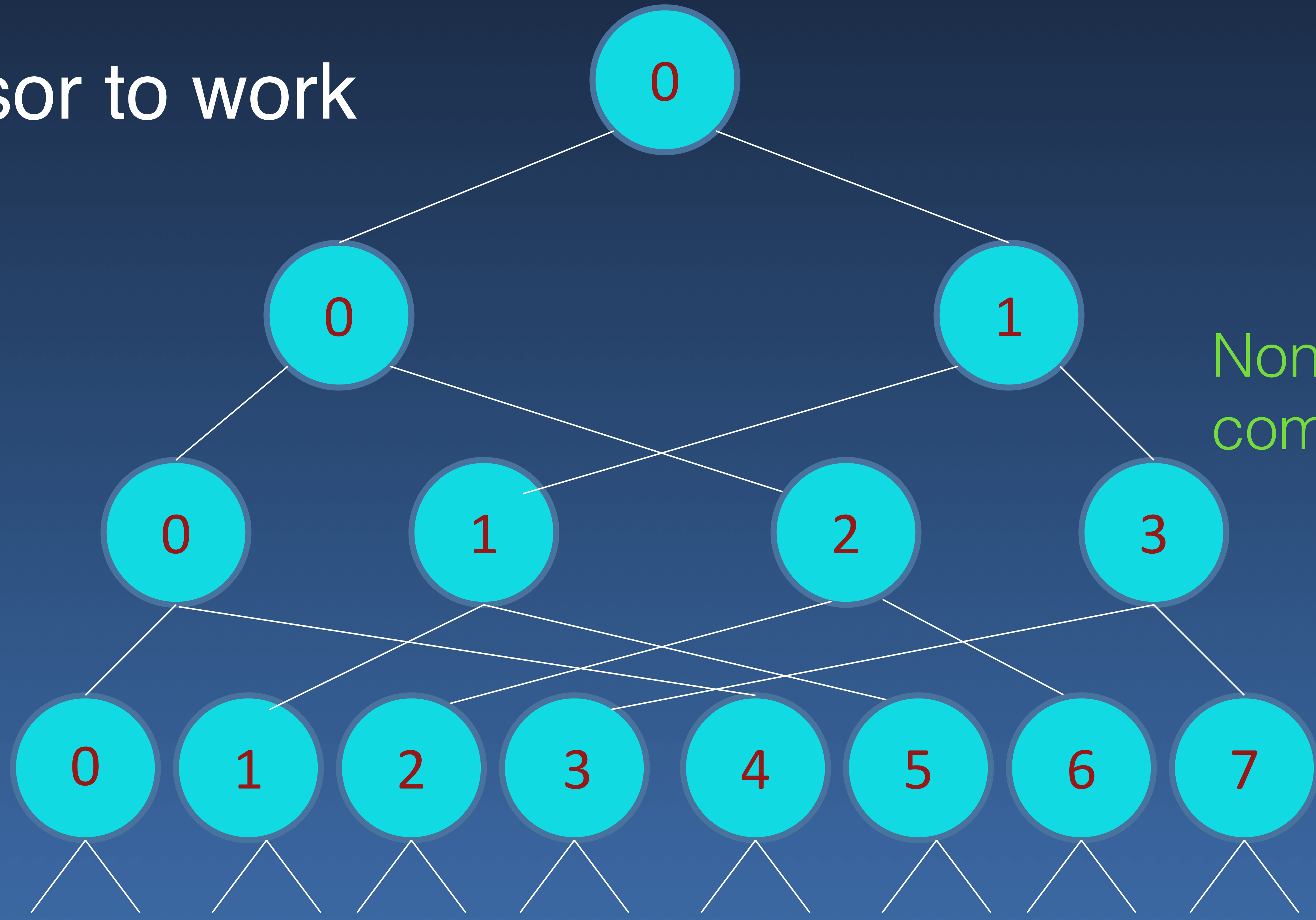
Reduction

- n operands ⇒ log n steps
- Total work = O(n)
- How do you map processor to work
  - ➡ $n/2^i$ processors per step
  - ➡ step i: forall id < $n/2^i$
    - Read: 2*id, 2*id+ 1
    - Write: id



Reduction

- n operands ⇒ log n steps
- Total work = O(n)
- How do you map processor to work
  - ➡ $n/2^i$ processors per step
  - ➡ step i: forall id < $n/2^i$
    - ▸ Read: id, id+ $n/2^i$
    - ▸ Write: id

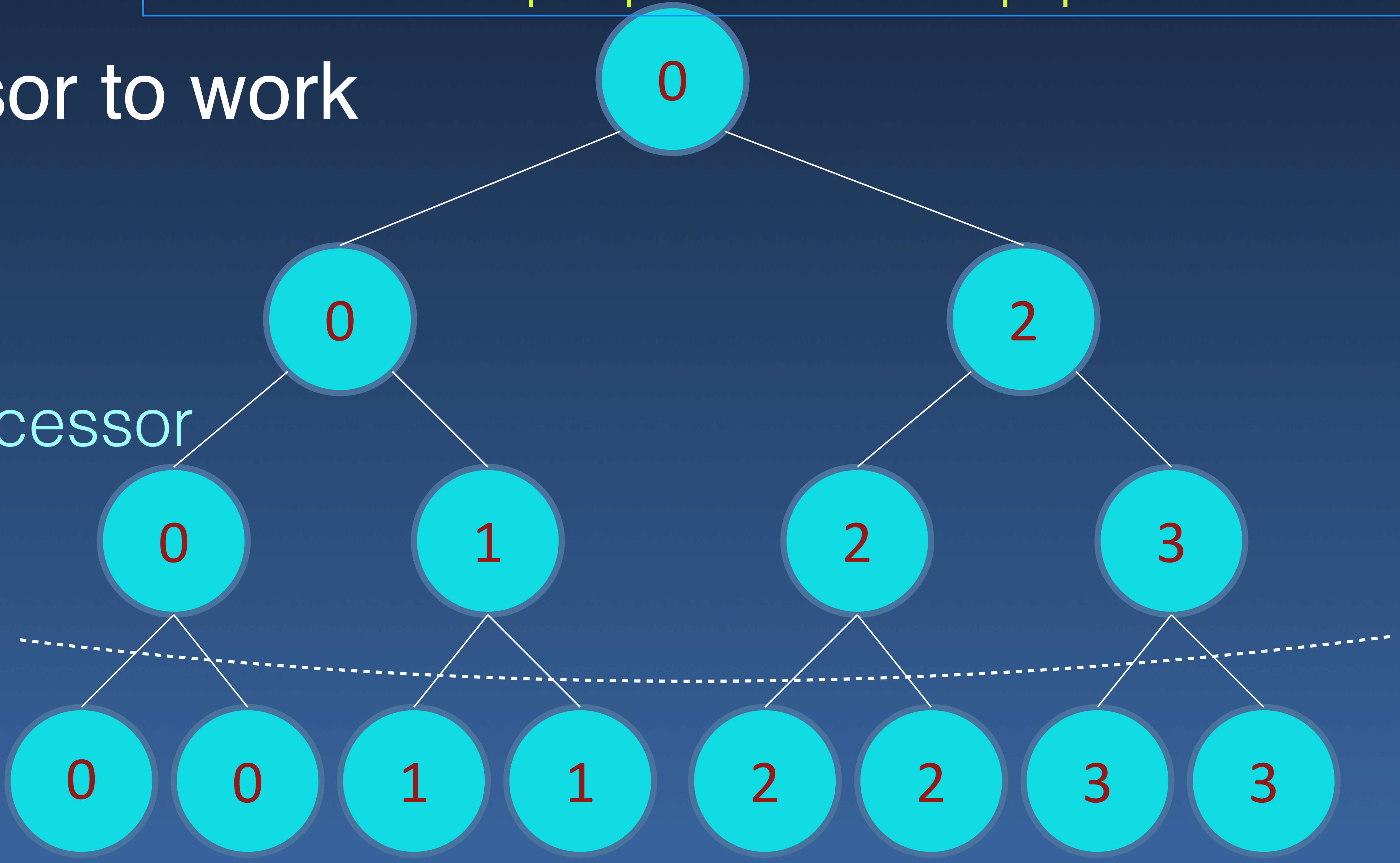


Non-proximate communication

Reduction

- Count the number of operations
  - ➡ Then allocate to p processors
- Often, convenient to start with p
  - ➡ And map operations to p processors

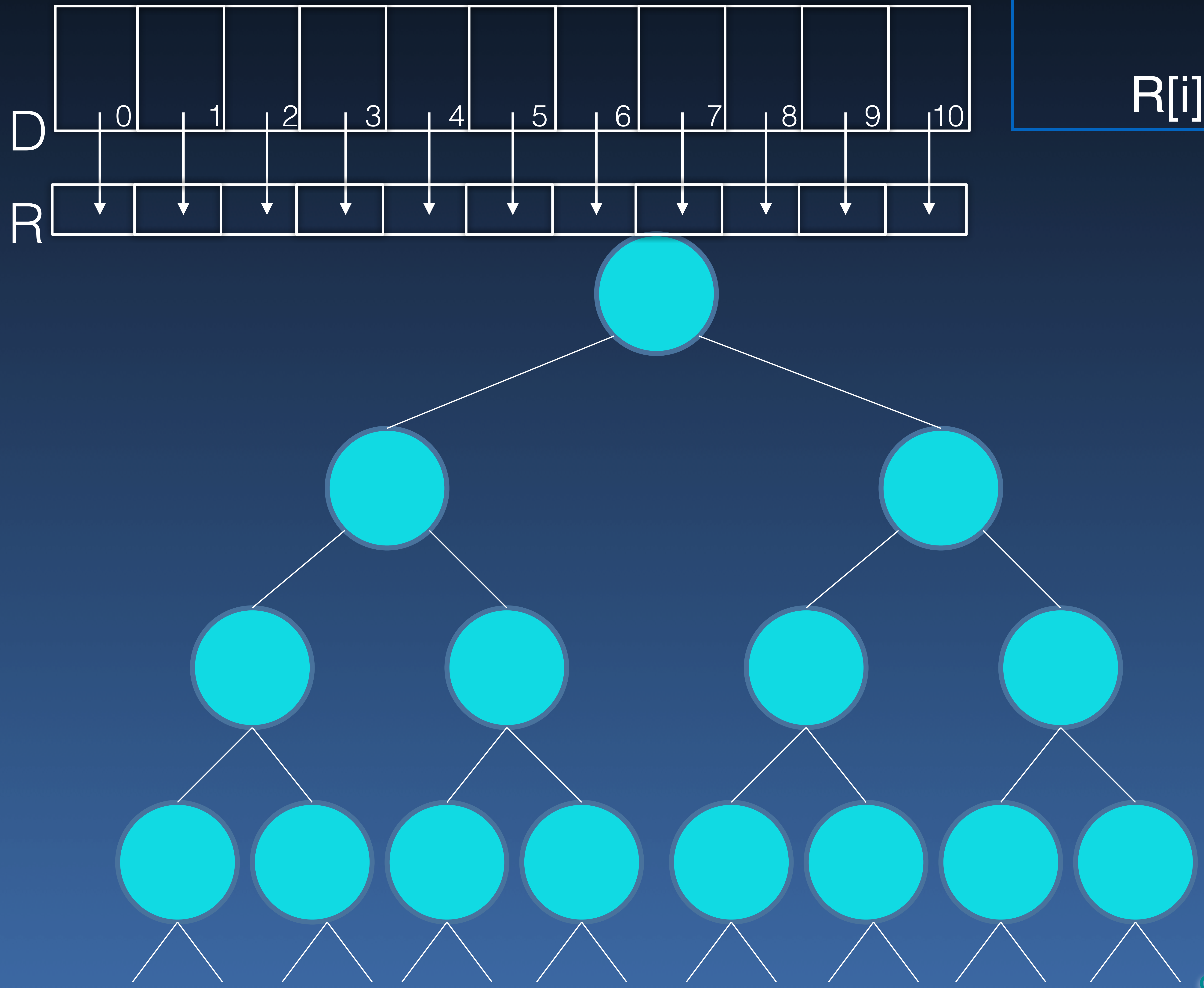
- n operands ⇒ log n steps
- Total work = O(n)
- How do you map processor to work
  - ➡ Consider $p < n$
  - ➡ Locally reduce at each processor
  - ➡ $p/2^i$ processors per step
  - ➡ step i: if $!(id\%2^i)$



Prefix Sums

Input: x

- P[0] = x[0]
- For i = 1 to n-1
  - ➡ P[i] = P[i-1] + x[i]



```
forall i < n
        R[i] = Map(D[i]);
```

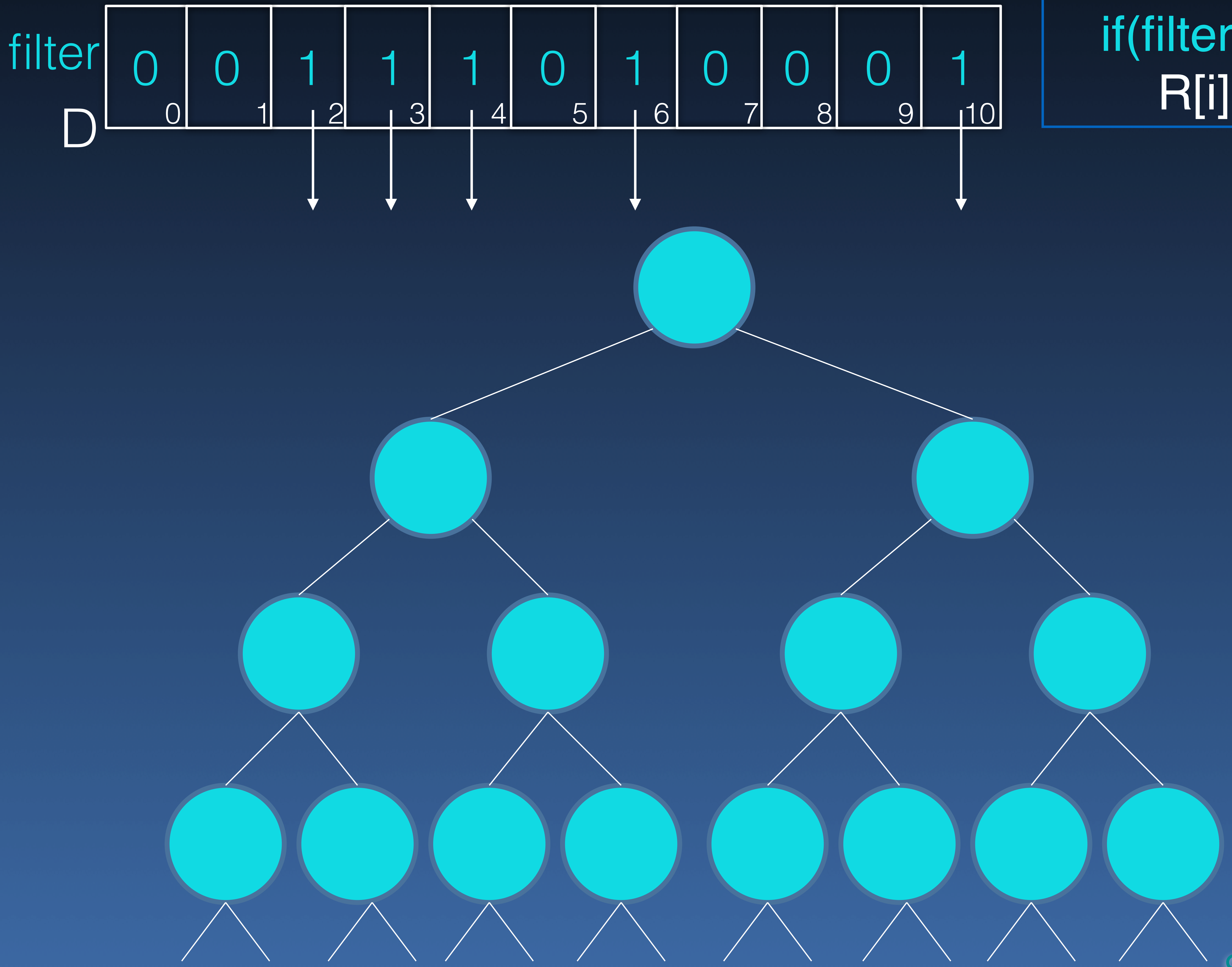
Subodh Kumar

Prefix Sums

```
forall i < n
    if(filter(D[i]))
        R[i] = Map(D[i]);
```

Input: x

- P[0] = x[0]
- For i = 1 to n-1
    - ➡ P[i] = P[i-1] + x[i]



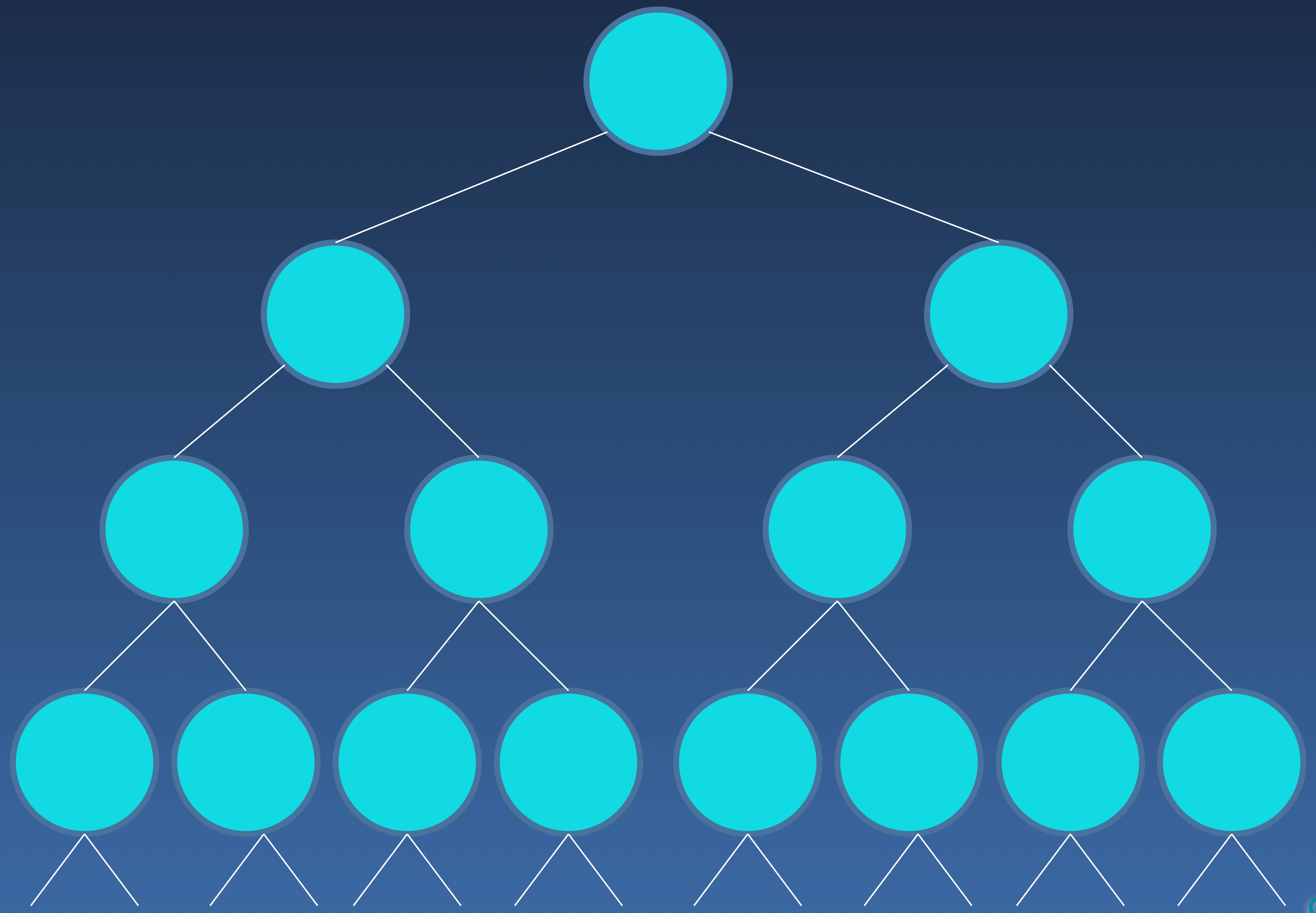
Prefix Sums

S	0	0	1	2	3	3	4	4	4	4	5
filter	0	0	1	1	1	0	1	0	0	0	1
D	0	1	2	3	4	5	6	7	8	9	10

```
forall i < n
    if(filter(D[i]))
        R[i] = Map(D[i]);
```

Input: x

- P[0] = x[0]
- For i = 1 to n-1
  - ➡ P[i] = P[i-1] + x[i]

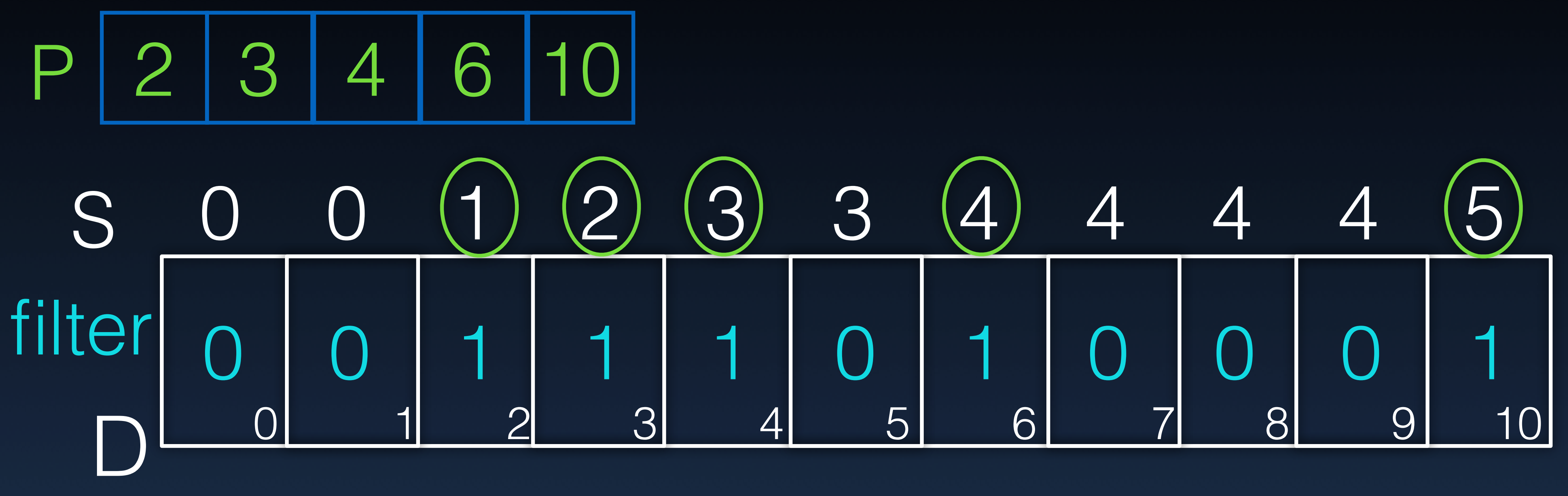


Subodh Kumar

Prefix Sums

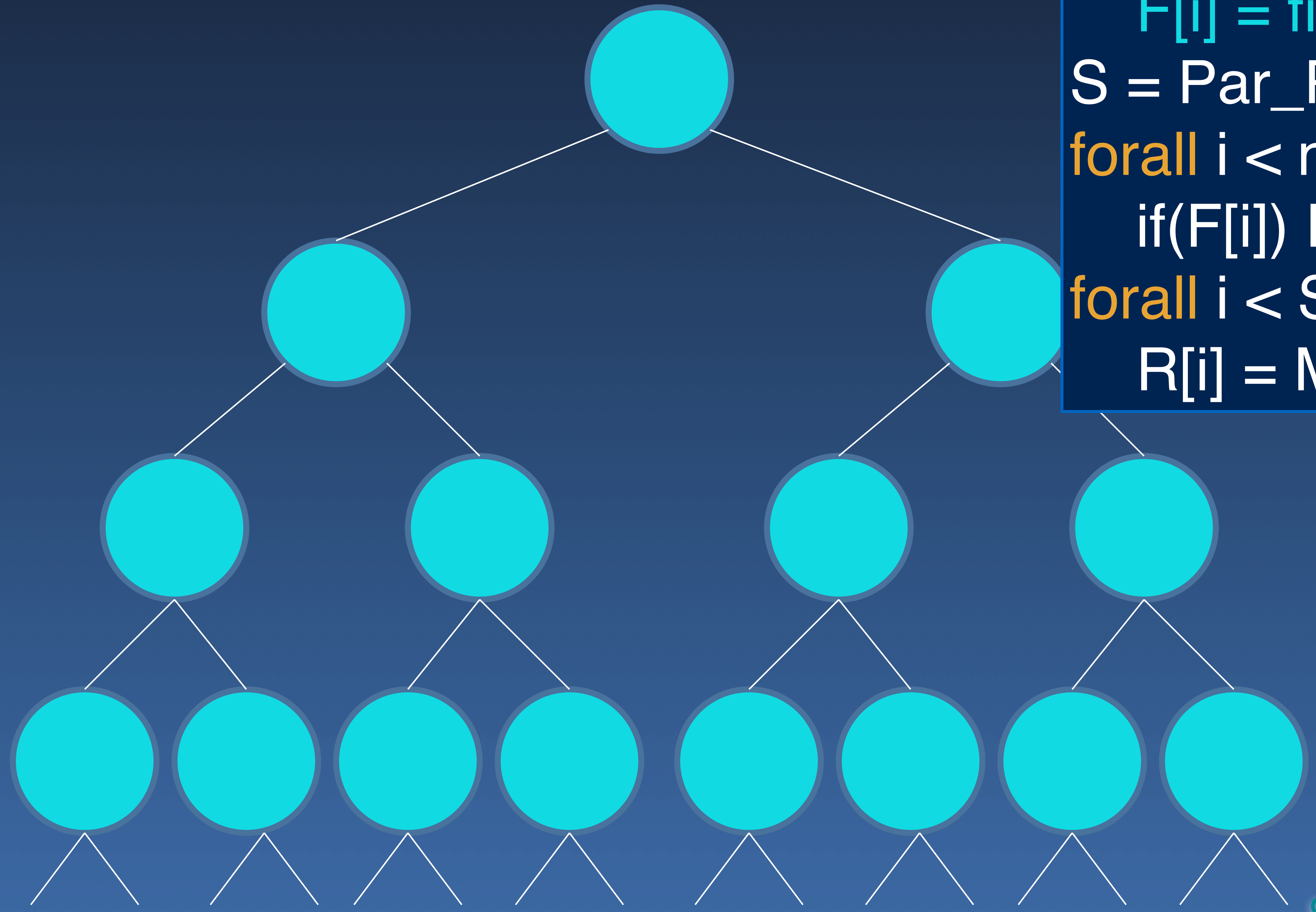
Input: x

- $P[0] = x[0]$
- For $i = 1$ to $n-1$
  - ➡ $P[i] = P[i-1] + x[i]$



```
forall i < n
    if(filter(D[i]))
        R[i] = Map(D[i]);
```

```
forall i < n
    F[i] = filter(D[i]);
S = Par_PrefixSum(F)
forall i < n
    if(F[i]) P[S[i]-1] = i;
forall i < S[n]
    R[i] = Map(D[P[i]]);
```

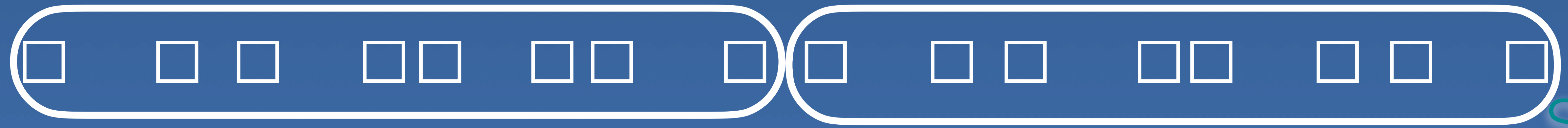


Subodh Kumar

Recursive Prefix Sums

- P[0] = x[0]
- For i = 1 to n-1
  - ➡ P[i] = P[i-1] + x[i]

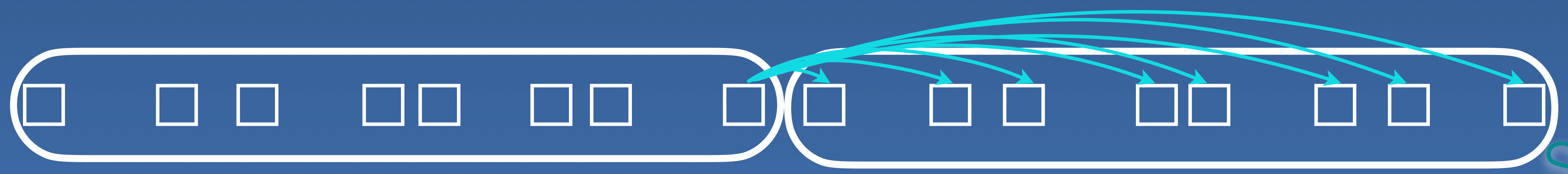
Psum two halves



Recursive Prefix Sums

- P[0] = x[0]
- For i = 1 to n-1
  - ➡ P[i] = P[i-1] + x[i]

Complete original Psum

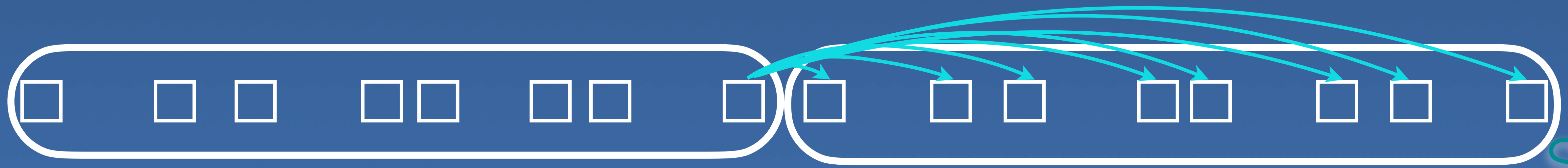


Recursive Prefix Sums

- P[0] = x[0]
- For i = 1 to n-1
  - ➡ P[i] = P[i-1] + x[i]

$T(n) = T(n/2) + O(1)$
$W(n) = 2W(n/2)+Kn/2$

Complete original Psum



Subodh Kumar

Recursive Prefix Sums

$T(n) = T(n/2) + O(1)$
$W(n) = 2W(n/2)+Kn/2$

$\boxed{W(n) = O(n \log n)}$

```
• P[0] = x[0]
• For i = 1 to n-1
    ➡ P[i] = P[i-1] + x[i]
```

□ □ □ □□ □□ □ □ □ □ □□ □ □ □

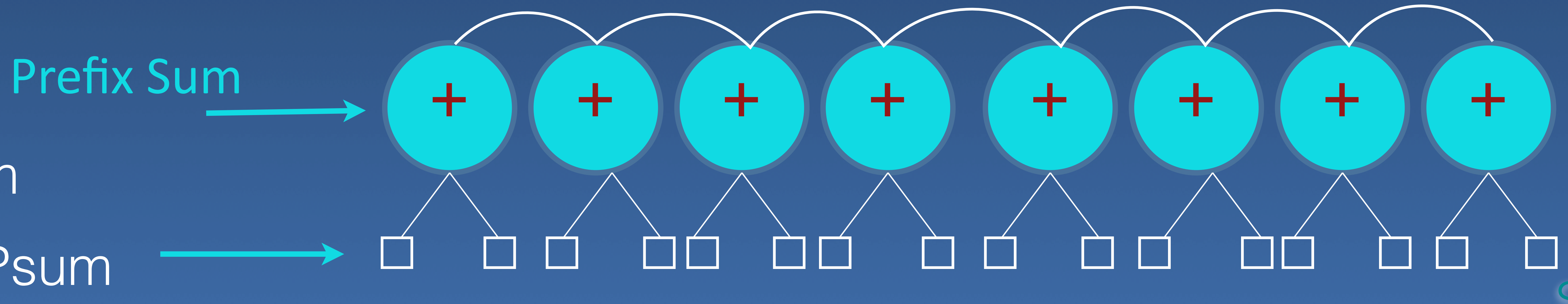
Subodh Kumar

Recursive Prefix Sums

$T(n) = T(n/2) + O(1)$
$W(n) = 2W(n/2)+Kn/2$

$\boxed{W(n) = O(n \log n)}$

- P[0] = x[0]
- For i = 1 to n-1
  - ➡ P[i] = P[i-1] + x[i]

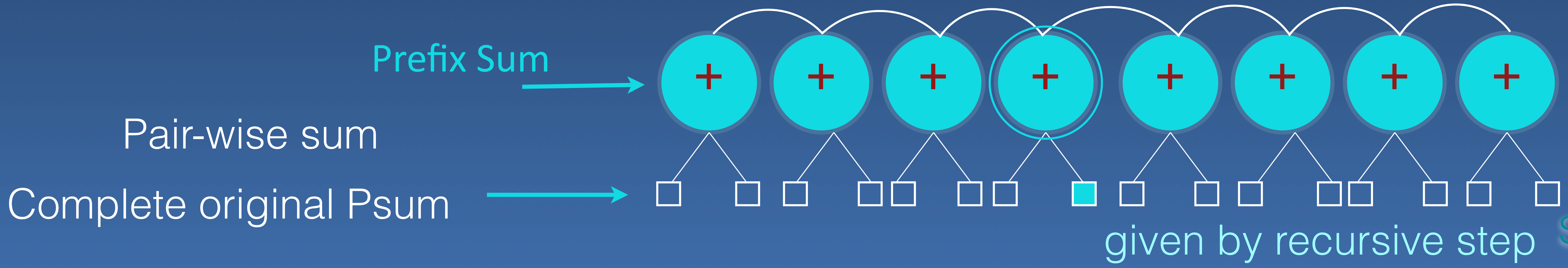


Recursive Prefix Sums

$T(n) = T(n/2) + O(1)$
$W(n) = 2W(n/2)+Kn/2$

$\boxed{W(n) = O(n \log n)}$

- P[0] = x[0]
- For i = 1 to n-1
  - ➡ P[i] = P[i-1] + x[i]

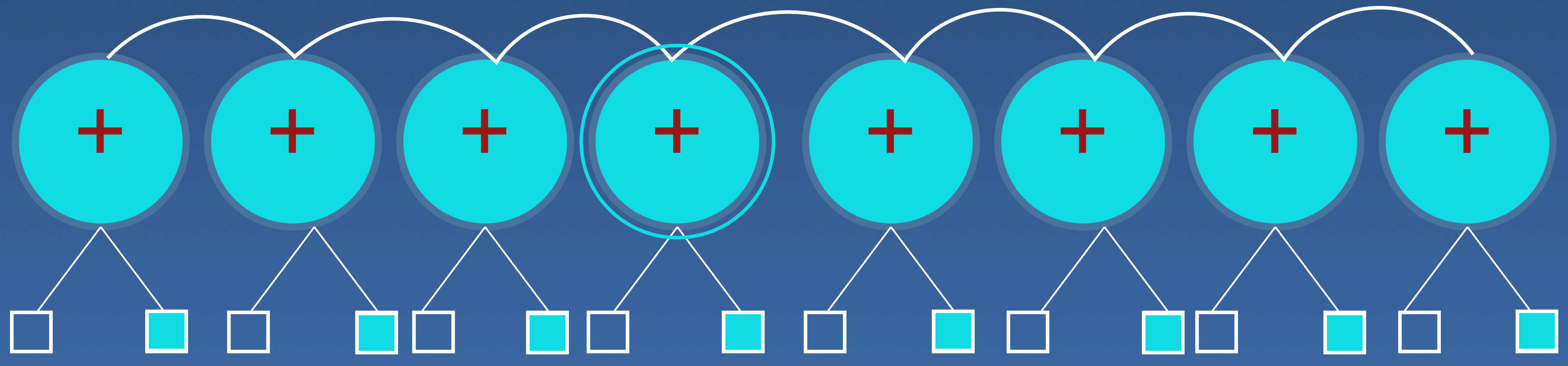


Recursive Prefix Sums

$T(n) = T(n/2) + O(1)$
$W(n) = 2W(n/2)+Kn/2$

$W(n) = O(n \log n)$

- $P[0] = x[0]$
- For $i = 1$ to $n-1$
  - ➡ $P[i] = P[i-1] + x[i]$



Prefix Sum

Pair-wise sum

Complete original Psum

given by recursive step

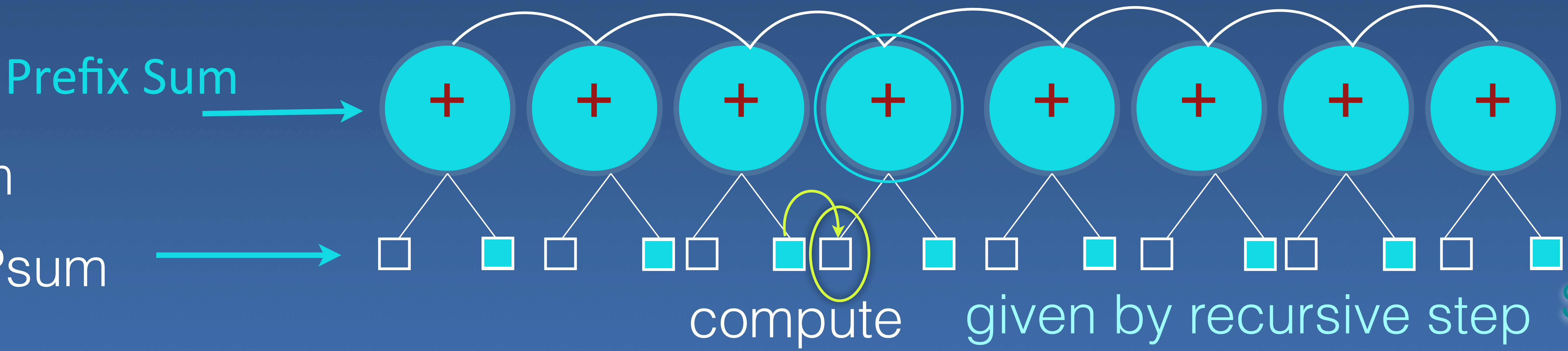
Subodh Kumar

Recursive Prefix Sums

$T(n) = T(n/2) + O(1)$
$W(n) = 2W(n/2)+Kn/2$

$W(n) = O(n \log n)$

- P[0] = x[0]
- For i = 1 to n-1
  - ➡ P[i] = P[i-1] + x[i]



Recursive Prefix Sums

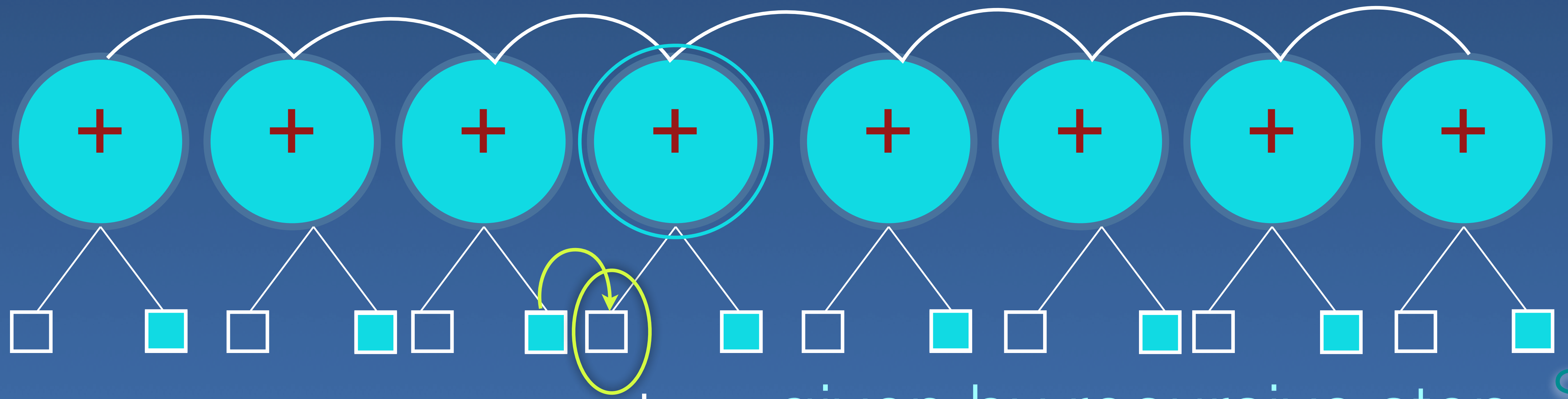
$\boxed{\begin{array}{l} T(n) = T(n/2) + O(1) \\ W(n) = 2W(n/2)+Kn/2 \end{array}}$

$\boxed{W(n) = O(n \log n)}$

- P[0] = x[0]
- For i = 1 to n-1
  - ➡ P[i] = P[i-1] + x[i]

$\boxed{\begin{array}{l} T(n) = T(n/2) + O(1) \\ W(n) = W(n/2)+Kn/2 \end{array}}$

$\boxed{W(n) = O(n)}$

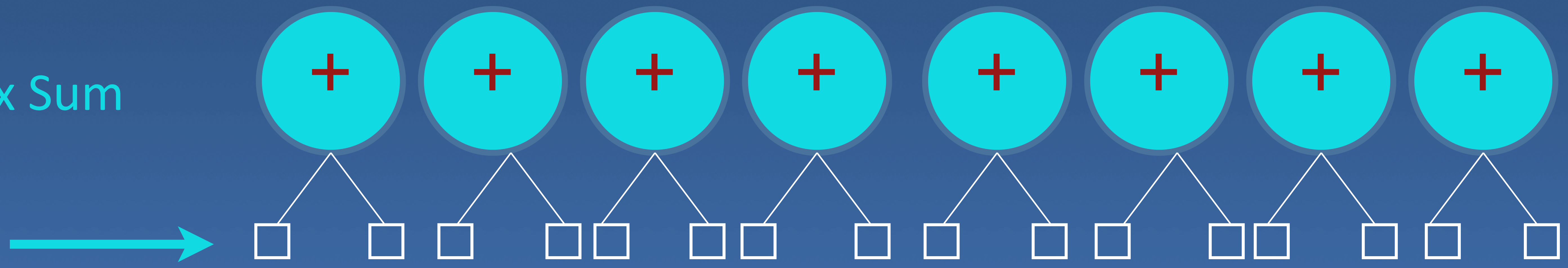


Subodh Kumar

Recursive Prefix Sums

```
prefixSums(P, x, [0:n))
{
    forall i in [0:n/2)
        y[i] =  OP(x[2*i], x[2*i+1])
    prefixSum(z, y, [0:n/2))
    P[0] = x[0]
    forall i in [1:n)
        if(i&1) P[i] = z[i/2]
        else    P[i] = OP(z[i/2-1 ], x[i])
}
```

Or OP^{-1} (z[i/2], x[i]), if op invertible



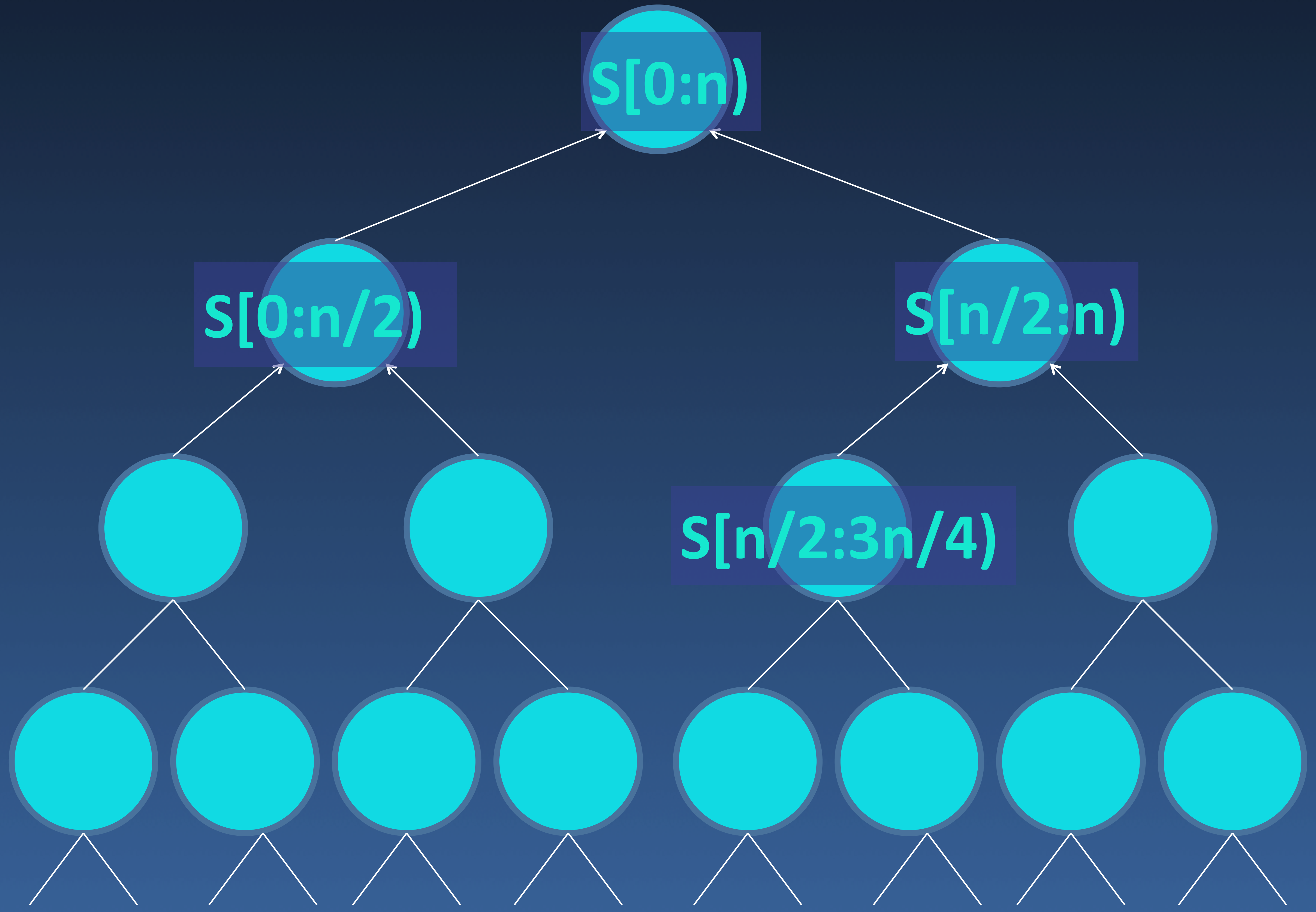
Prefix Sum Binary Tree

(Non recursive)

```
P[0] = x[0]
For i = 1 to n-1
    P[i] = P[i-1] + x[i]
```

```
forall i = 0 to n
    B[0][i] = A[i]
for h = 1 to log n
    forall i in 0:n/2^h
        B[h][i] = B[h-1][2i] OP B[h-1][2i+1]
```

Upward-pass



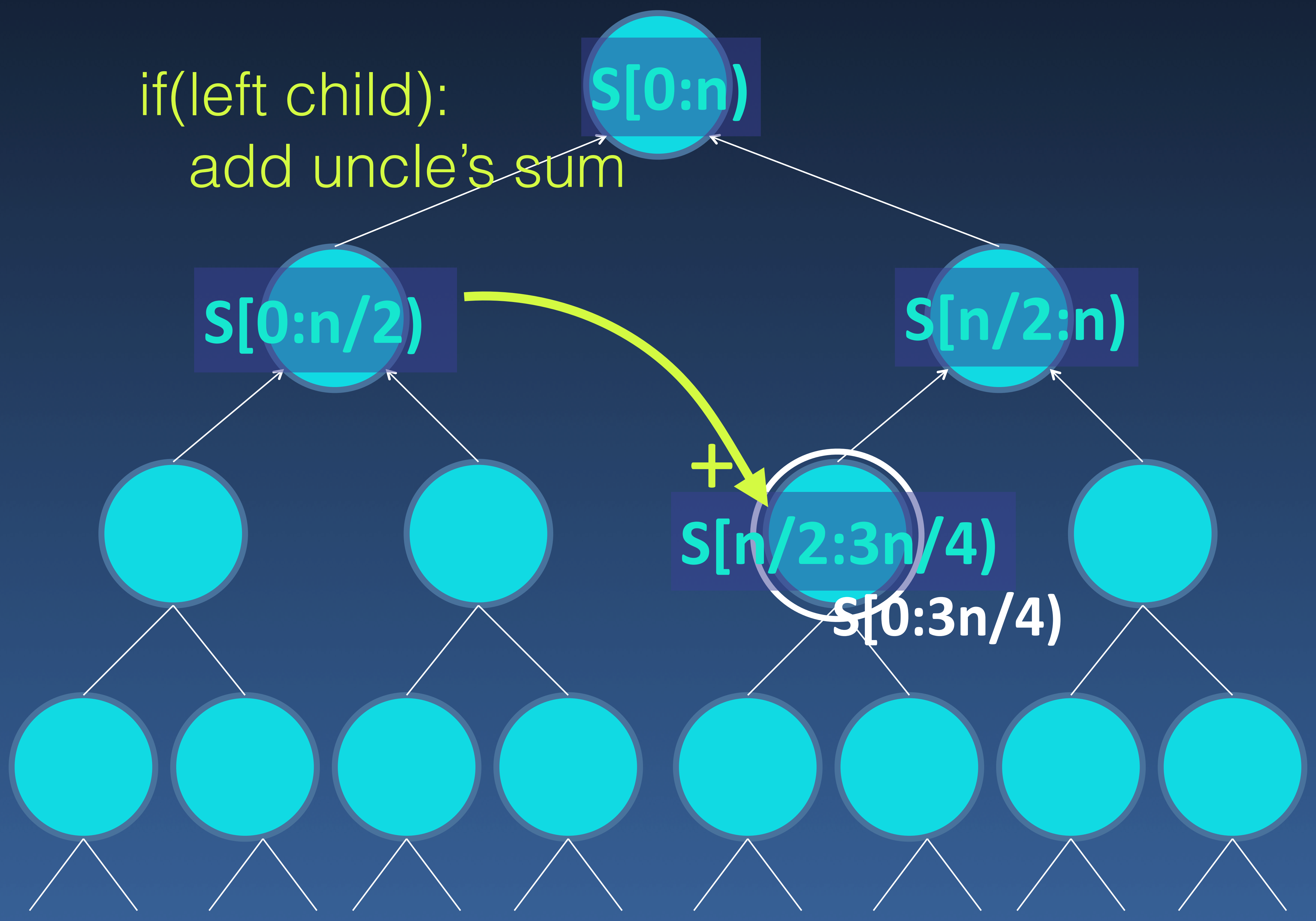
Prefix Sum Binary Tree

(Non recursive)

```
P[0] = x[0]
For i = 1 to n-1
    P[i] = P[i-1] + x[i]
```

```
forall i = 0 to n
    B[0][i] = A[i]
for h = 1 to log n
    forall i in 0:n/2^h
        B[h][i] = B[h-1][2i] OP B[h-1][2i+1]
for h = log n to 0
    C[h][0] = B[h][0]
    forall i in 1:n/2^h
        Odd i: C[h][i] = C[h+1][i/2]
        Even i: C[h][i] = C[h+1][i/2-1] OP B[h][i]
```

Downward-pass

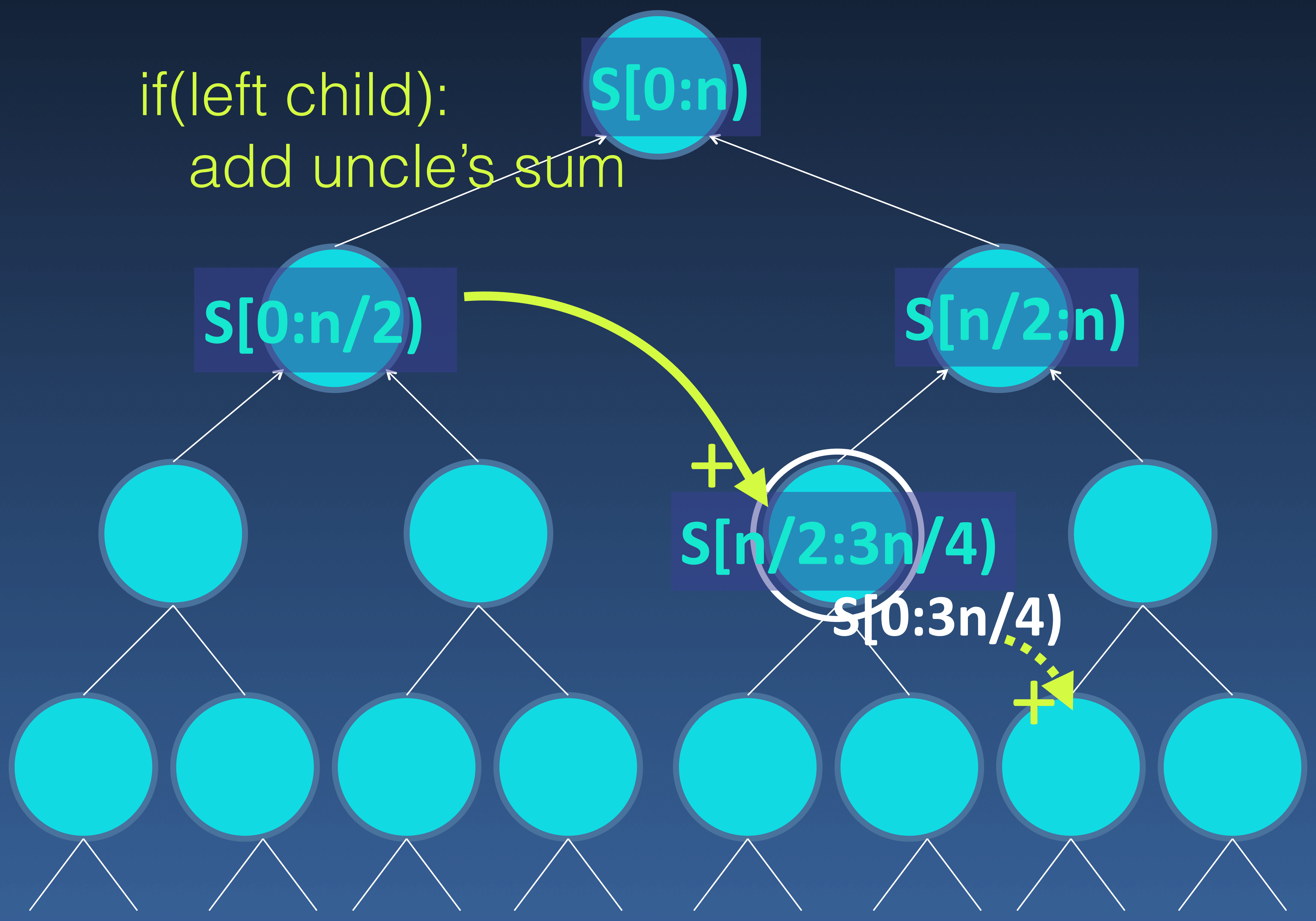


Subodh Kumar

Prefix Sum Binary Tree
(Non recursive)

```
P[0] = x[0]
For i = 1 to n-1
    P[i] = P[i-1] + x[i]
```

```
forall i = 0 to n
    B[0][i] = A[i]
for h = 1 to log n
    forall i in 0:n/2^h
        B[h][i] = B[h-1][2i] OP B[h-1][2i+1]
for h = log n to 0
    C[h][0] = B[h][0]          Downward-pass
    forall i in 1:n/2^h
        Odd i: C[h][i] = C[h+1][i/2]
        Even i: C[h][i] = C[h+1][i/2-1] OP B[h][i]
```



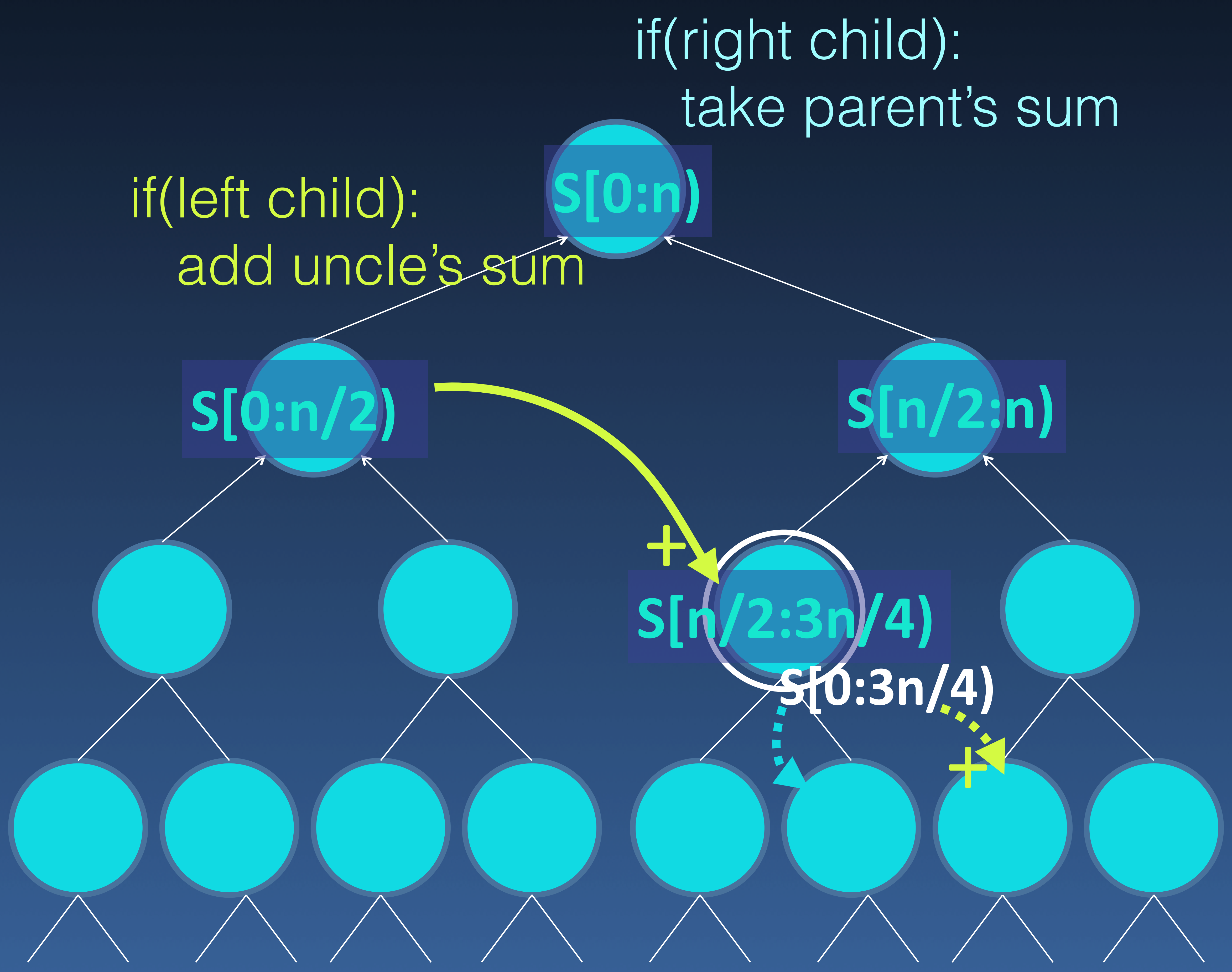
Subodh Kumar

Prefix Sum Binary Tree
(Non recursive)

```
P[0] = x[0]
For i = 1 to n-1
    P[i] = P[i-1] + x[i]
```

```
forall i = 0 to n
   B[0][i] = A[i]
for h = 1 to log n
   forall i in 0:n/2^h
      B[h][i] = B[h-1][2i] OP B[h-1][2i+1]
for h = log n to 0
   C[h][0] = B[h][0]
   forall i in 1:n/2^h
      Odd i: C[h][i] = C[h+1][i/2]
      Even i: C[h][i] = C[h+1][i/2-1] OP B[h][i]
```

Downward-pass



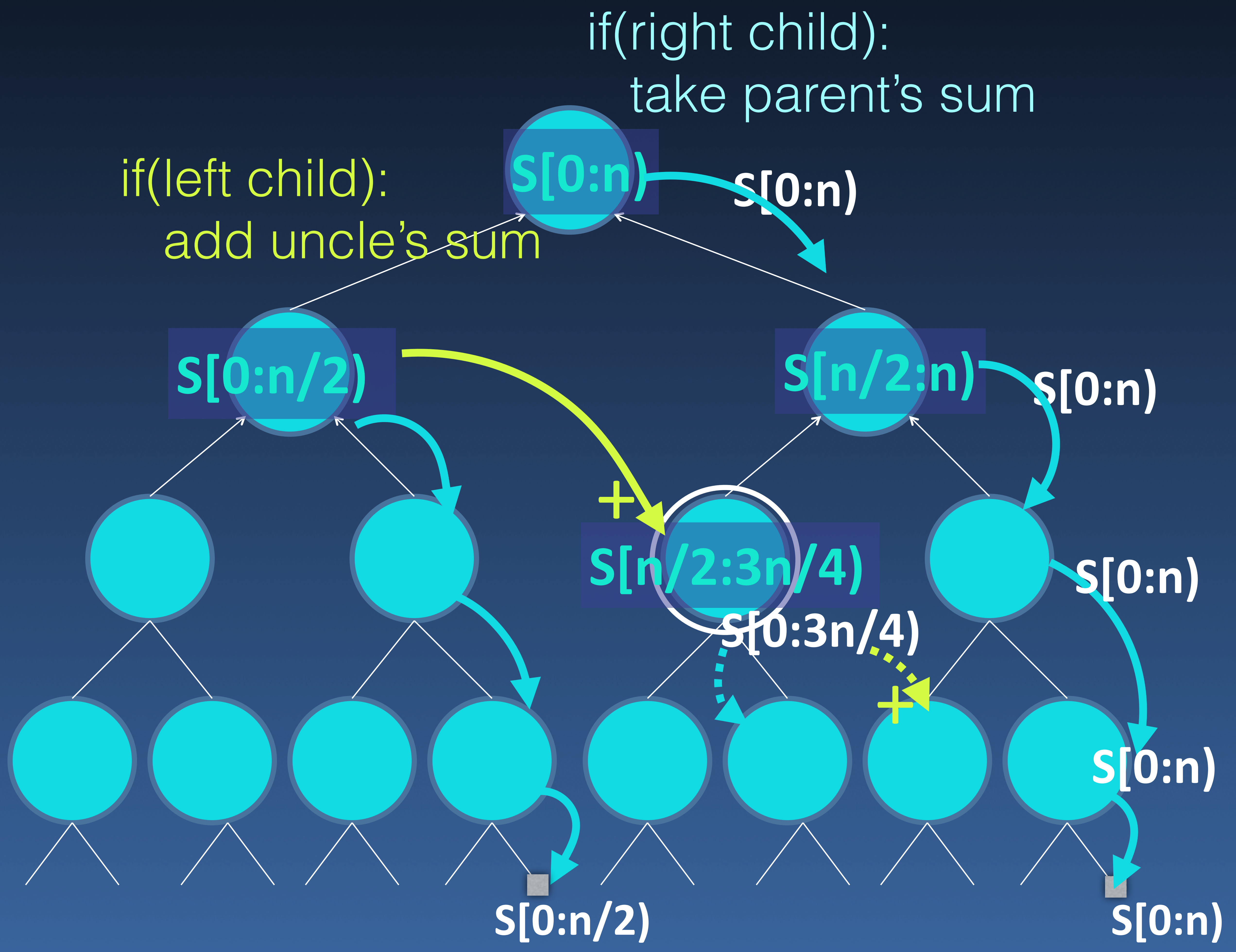
Subodh Kumar

Prefix Sum Binary Tree
(Non recursive)

```
P[0] = x[0]
For i = 1 to n-1
    P[i] = P[i-1] + x[i]
```

```
forall i = 0 to n
    B[0][i] = A[i]
for h = 1 to log n
    forall i in 0:n/2^h
        B[h][i] = B[h-1][2i] OP B[h-1][2i+1]
for h = log n to 0
    C[h][0] = B[h][0]
    forall i in 1:n/2^h
        Odd i: C[h][i] = C[h+1][i/2]
        Even i: C[h][i] = C[h+1][i/2-1] OP B[h][i]
```

Downward-pass



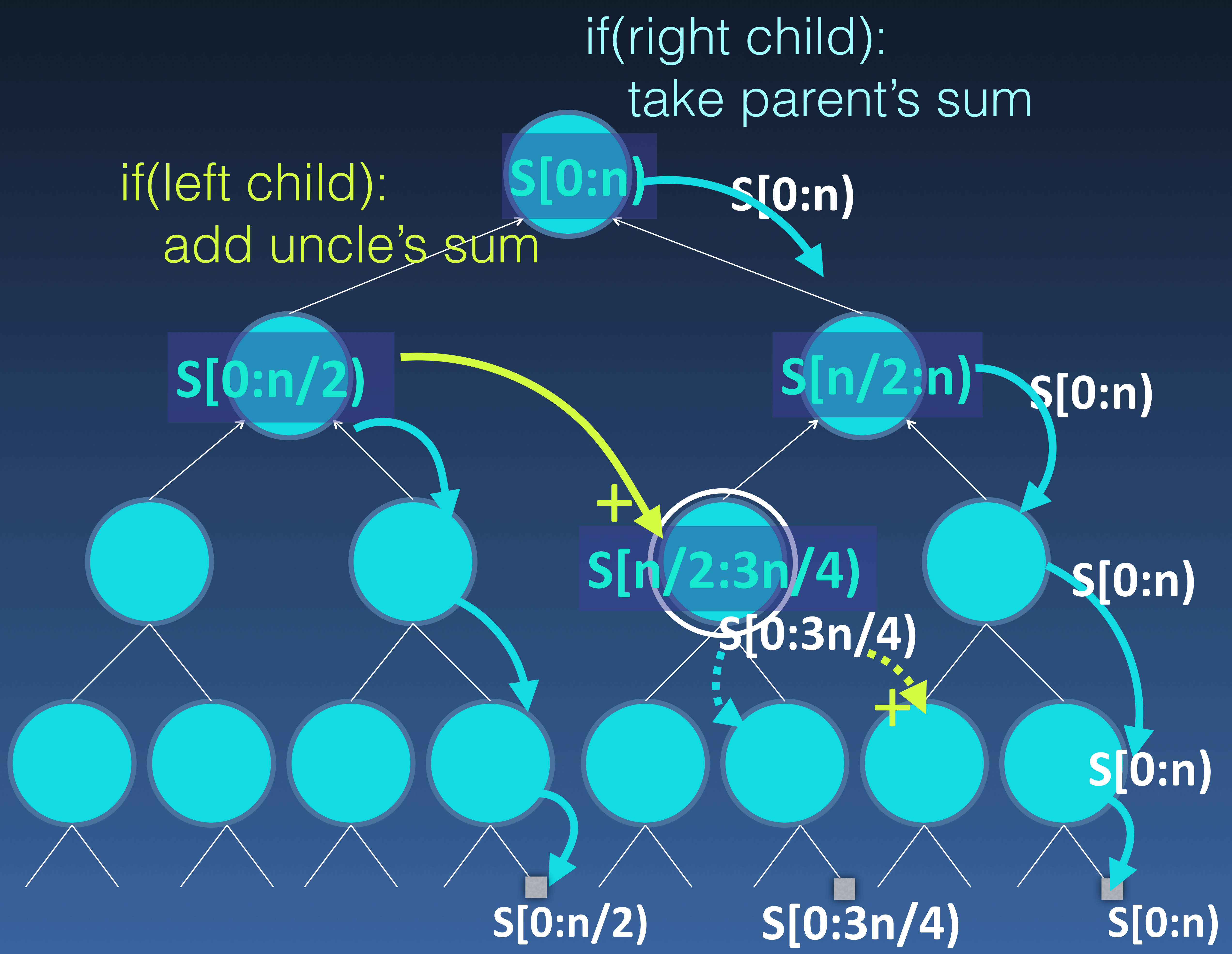
Subodh Kumar

Prefix Sum Binary Tree

(Non recursive)

```
P[0] = x[0]
For i = 1 to n-1
    P[i] = P[i-1] + x[i]
```

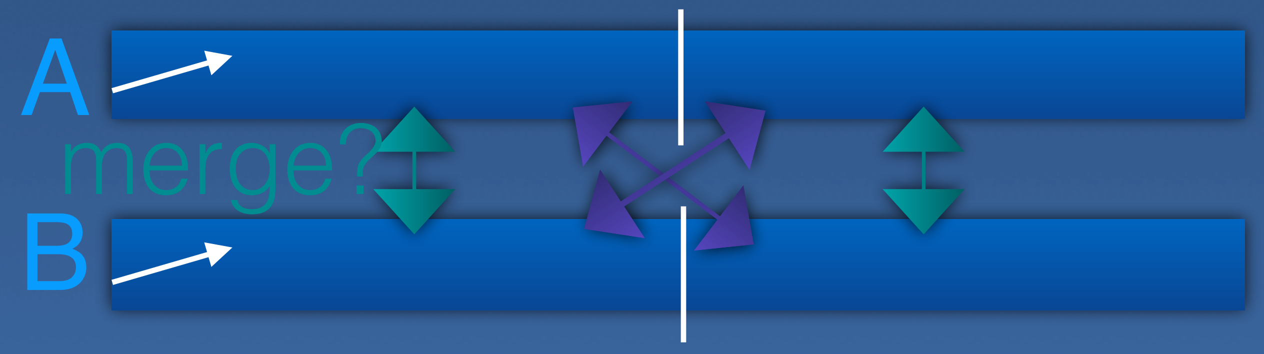
```
forall i = 0 to n
   B[0][i] = A[i]
for h = 1 to log n
   forall i in 0:n/2^h
      B[h][i] = B[h-1][2i] OP B[h-1][2i+1]
for h = log n to 0
   C[h][0] = B[h][0]          Downward-pass
   forall i in 1:n/2^h
      Odd i: C[h][i] = C[h+1][i/2]
      Even i: C[h][i] = C[h+1][i/2-1] OP B[h][i]
```



Subodh Kumar

Merge Sorted Arrays A,B

```
if(A[i] <= B[j])
    C[k++] = A[i++]
else
    C[k++] = B[j++]
```

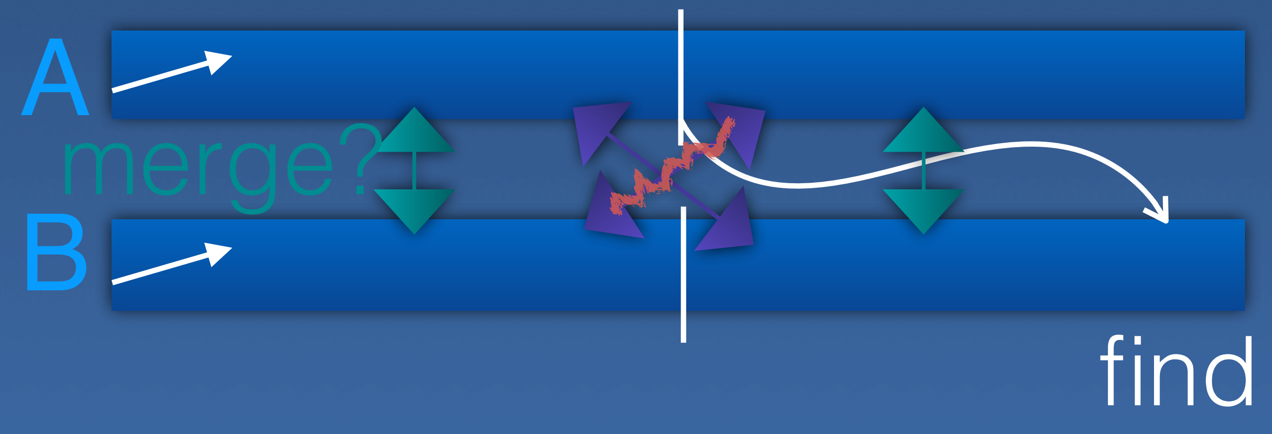


$T(2n) = T(n) + ?$

Subodh Kumar

Merge Sorted Arrays A,B

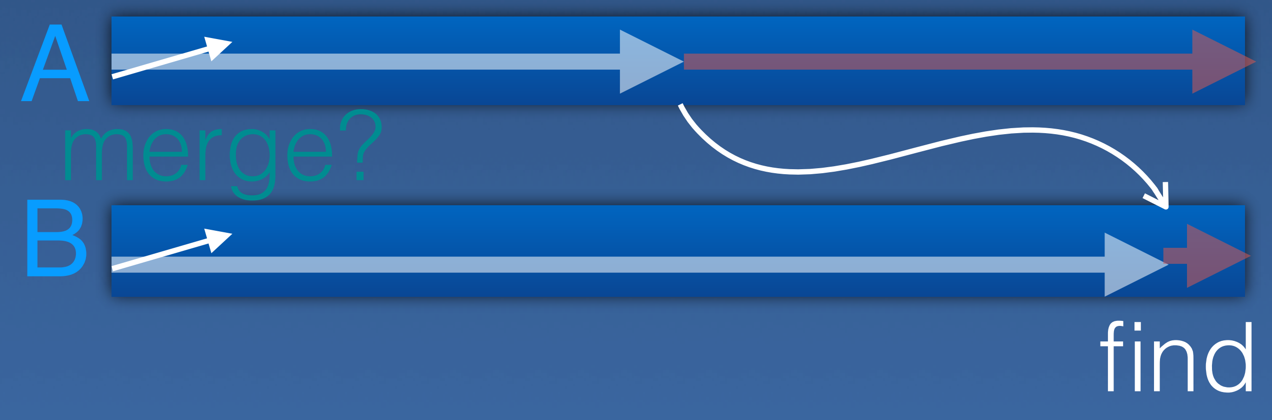
```
if(A[i] <= B[j])
    C[k++] = A[i++]
else
    C[k++] = B[j++]
```



$T(2n) = T(n) + ?$

Merge Sorted Arrays A,B

```
if(A[i] <= B[j])
    C[k++] = A[i++]
else
    C[k++] = B[j++]
```



$T(2n) = T(n) + ?$

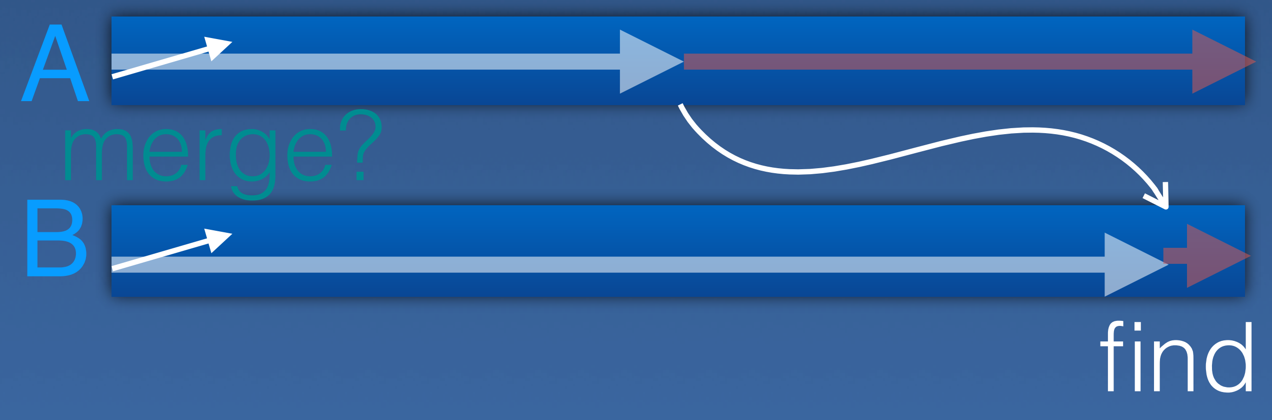
$T(2n) = T(3n/2) + \log n$

Subodh Kumar

Merge Sorted Arrays A,B

- Determine Rank of each element in A U B
- Rank(x, A U B) = Rank(x, A) + Rank(x, B)
  - ➡ A and B are each sorted; only need to compute the ranks in the other list
- Find Rank(A[i], B) ∀i and Rank(B[j], A) ∀j
  - ➡ Find each rank by binary search
  - ➡ O(log n) time
- O(n log n) work

```
if(A[i] <= B[j])
    C[k++] = A[i++]
else
    C[k++] = B[j++]
```



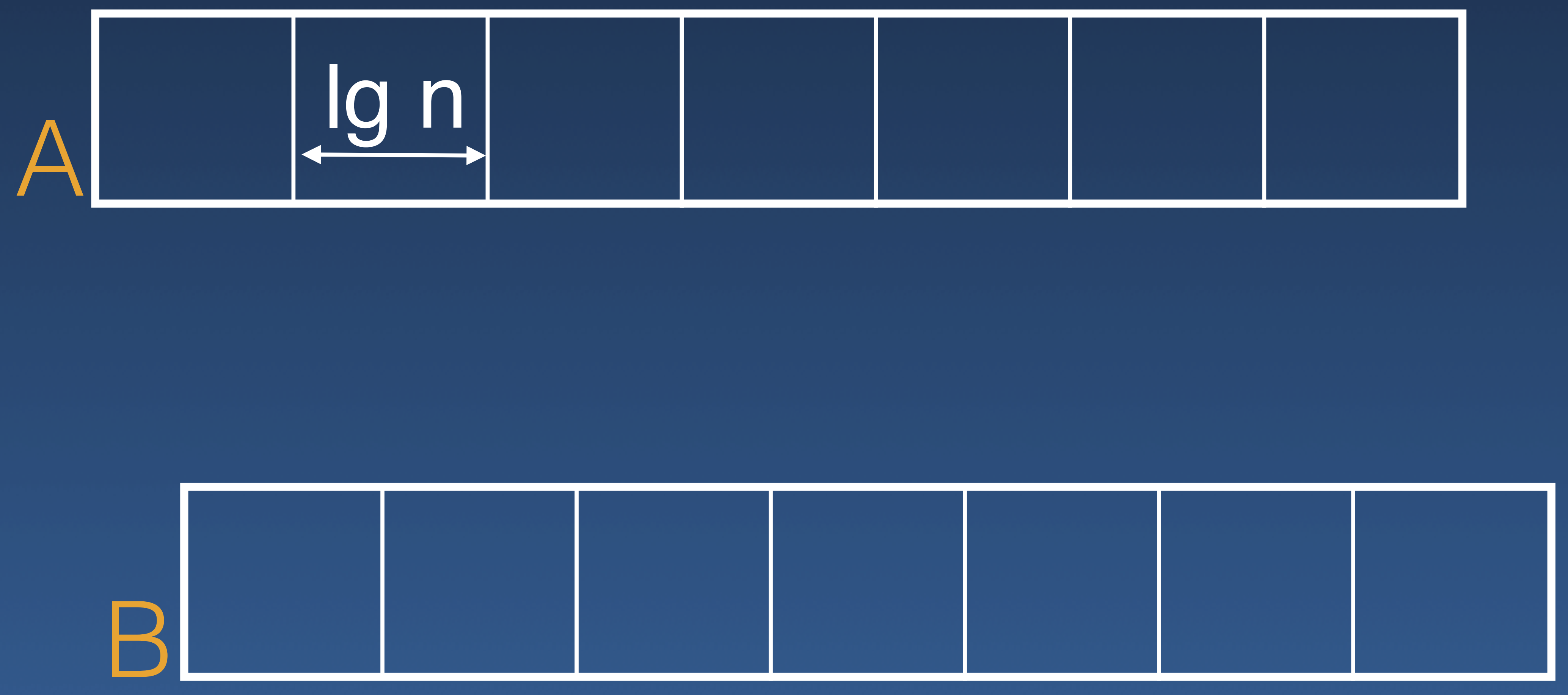
$T(2n) = T(n) + ?$

$T(2n) = T(3n/2) + \log n$

Towards Optimal Merge(A,B)

- Partition A and B into log n sized blocks

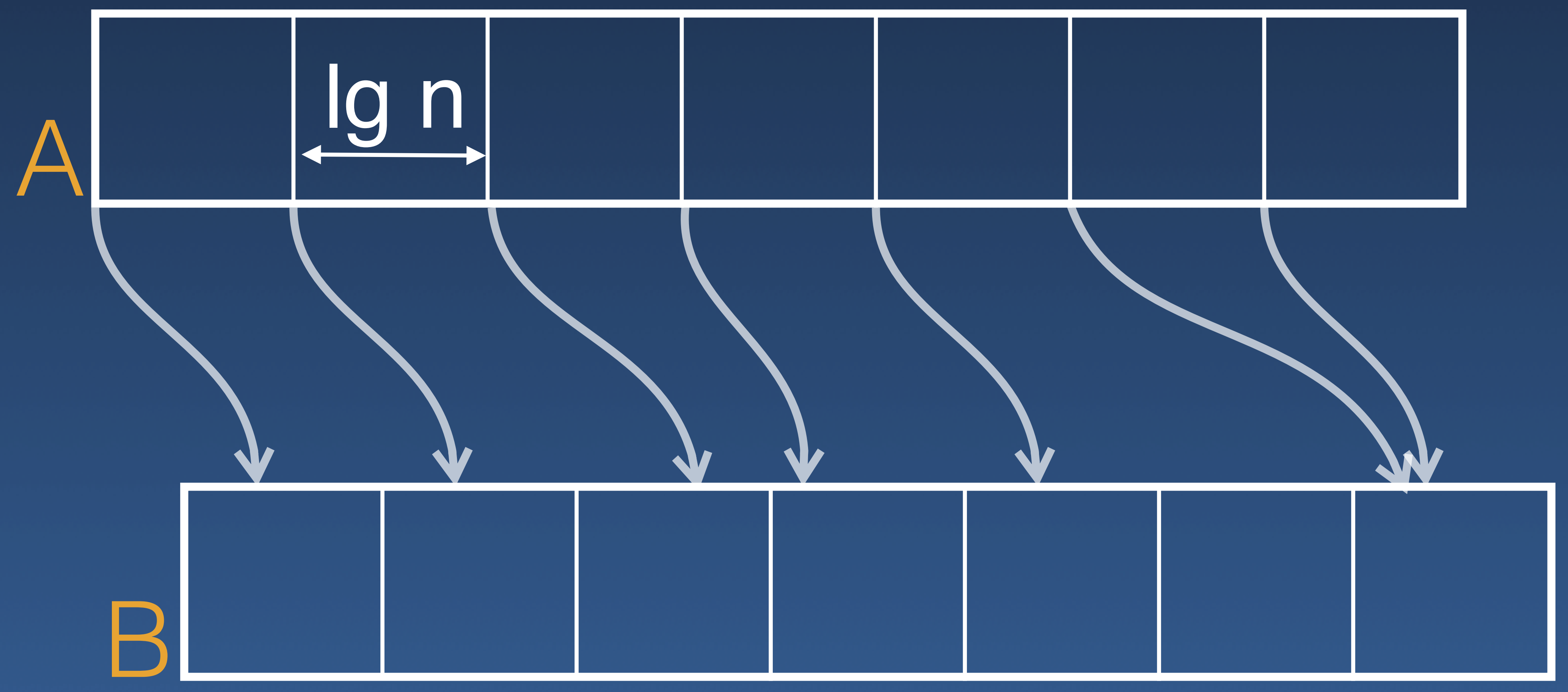
Data Partitioning Technique



Towards Optimal Merge(A,B)

- Partition A and B into log n sized blocks
- Select from A, elements i * log n, i ∈ 0:n/log n
- Rank each selected element of A in B
    - ➡ Binary search

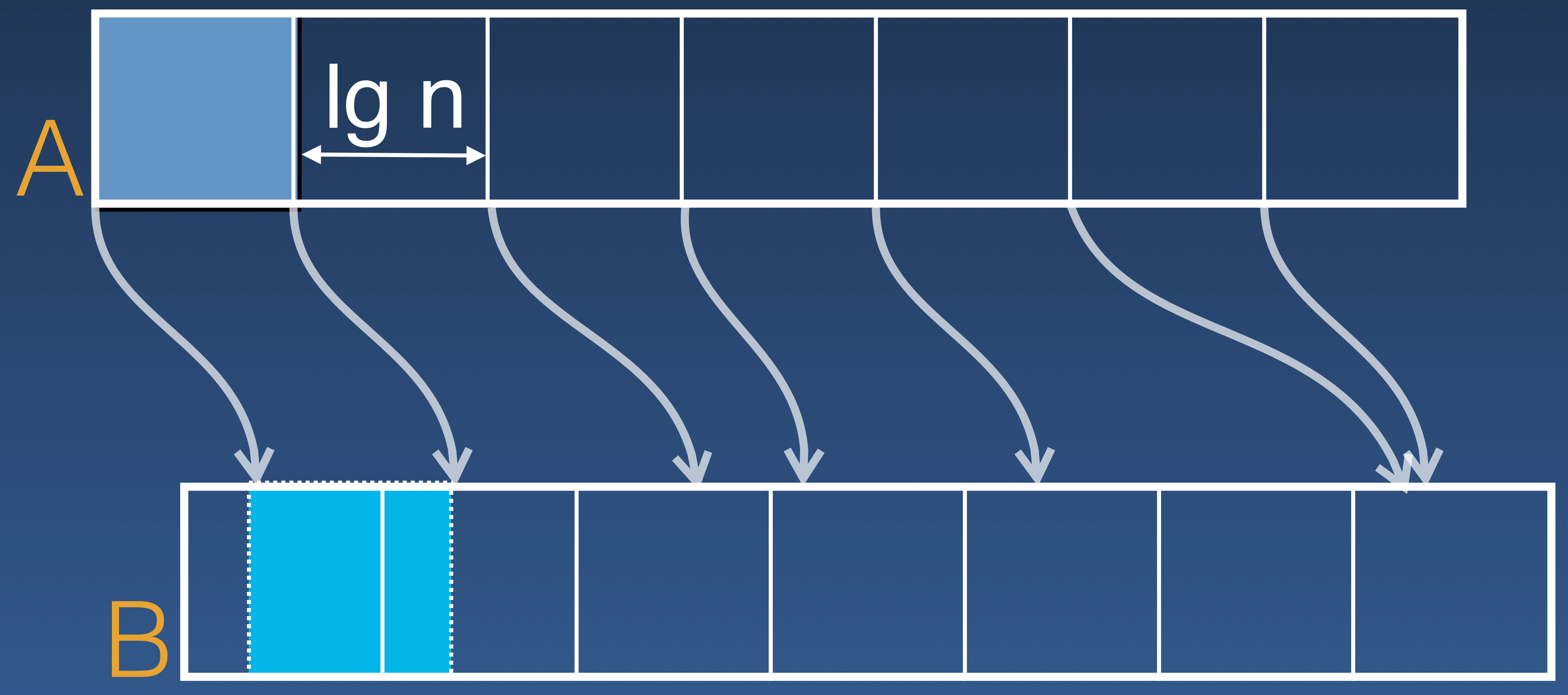
Data Partitioning Technique



Towards Optimal Merge(A,B)

- Partition A and B into log n sized blocks
- Select from A, elements i * log n, i ∈ 0:n/log n
- Rank each selected element of A in B
  - ➡ Binary search
- Merge pairs of sub-sequences

Data Partitioning Technique

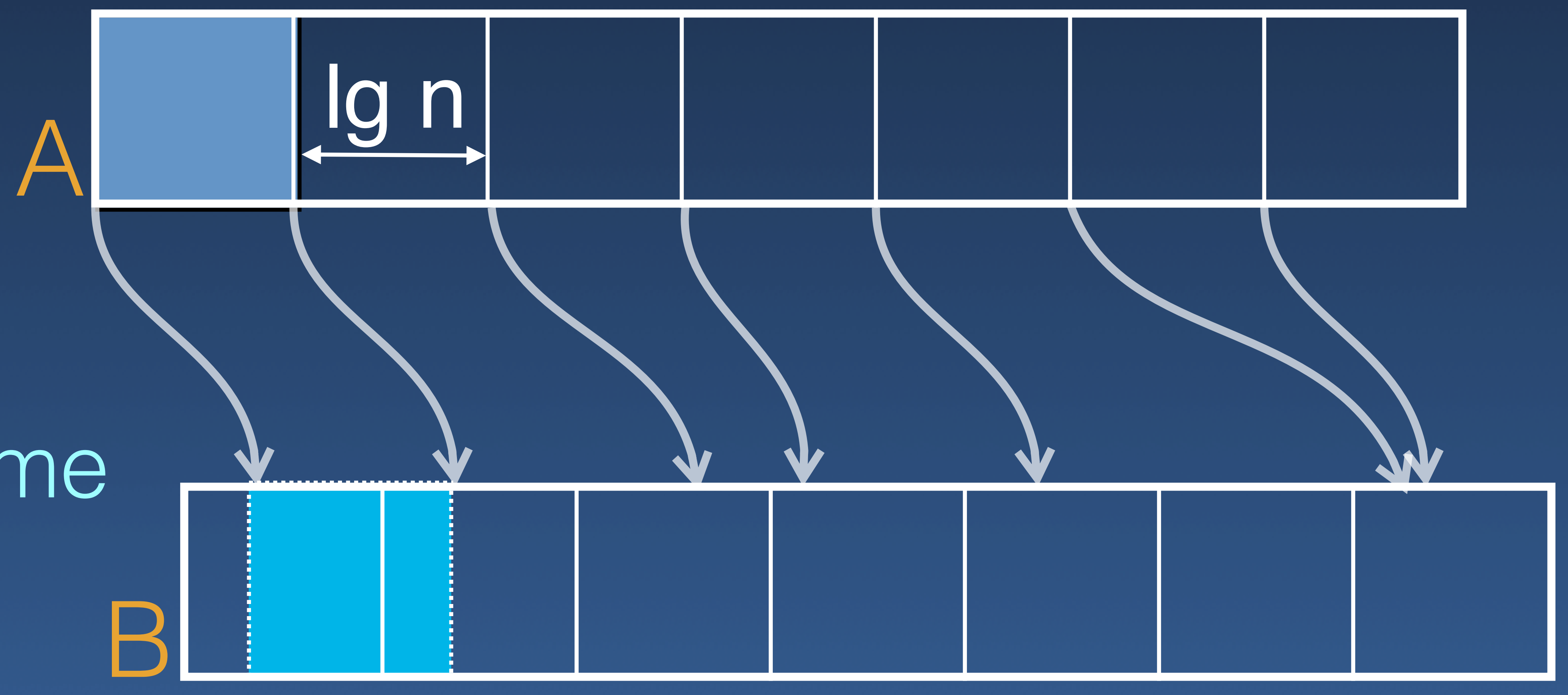


Subodh Kumar

Towards Optimal Merge(A,B)

- Partition A and B into log n sized blocks
- Select from A, elements i * log n, i ∊ 0:n/log n
- Rank each selected element of A in B
    - ➡ Binary search
- Merge pairs of sub-sequences
    - ➡ If $|B_i| \leq \log(n)$, Sequential merge in O(log n) time

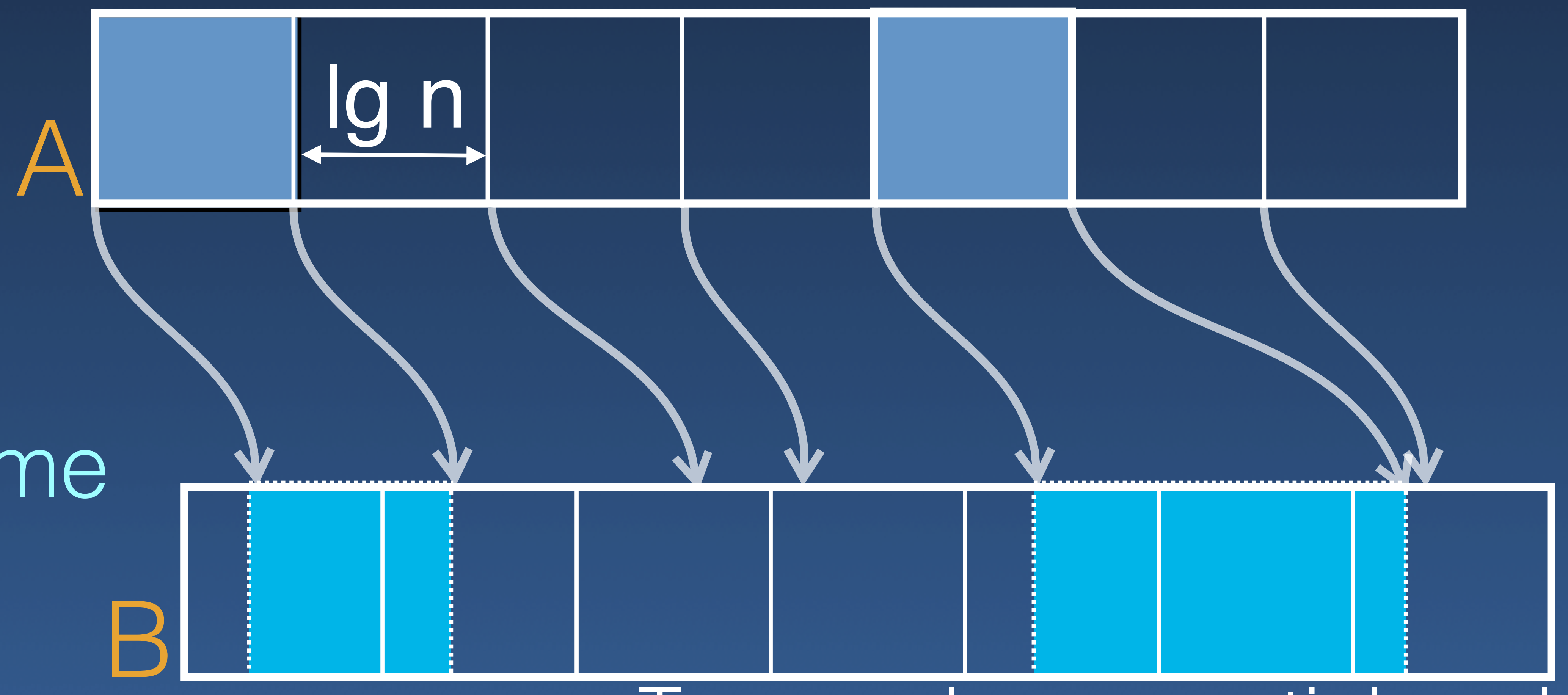
Data Partitioning Technique



Towards Optimal Merge(A,B)

- Partition A and B into log n sized blocks
- Select from A, elements i * log n, i ∈ 0:n/log n
- Rank each selected element of A in B
  - ➡ Binary search
- Merge pairs of sub-sequences
  - ➡ If $|B_i| \leq \log(n)$, Sequential merge in O(log n) time

Data Partitioning Technique

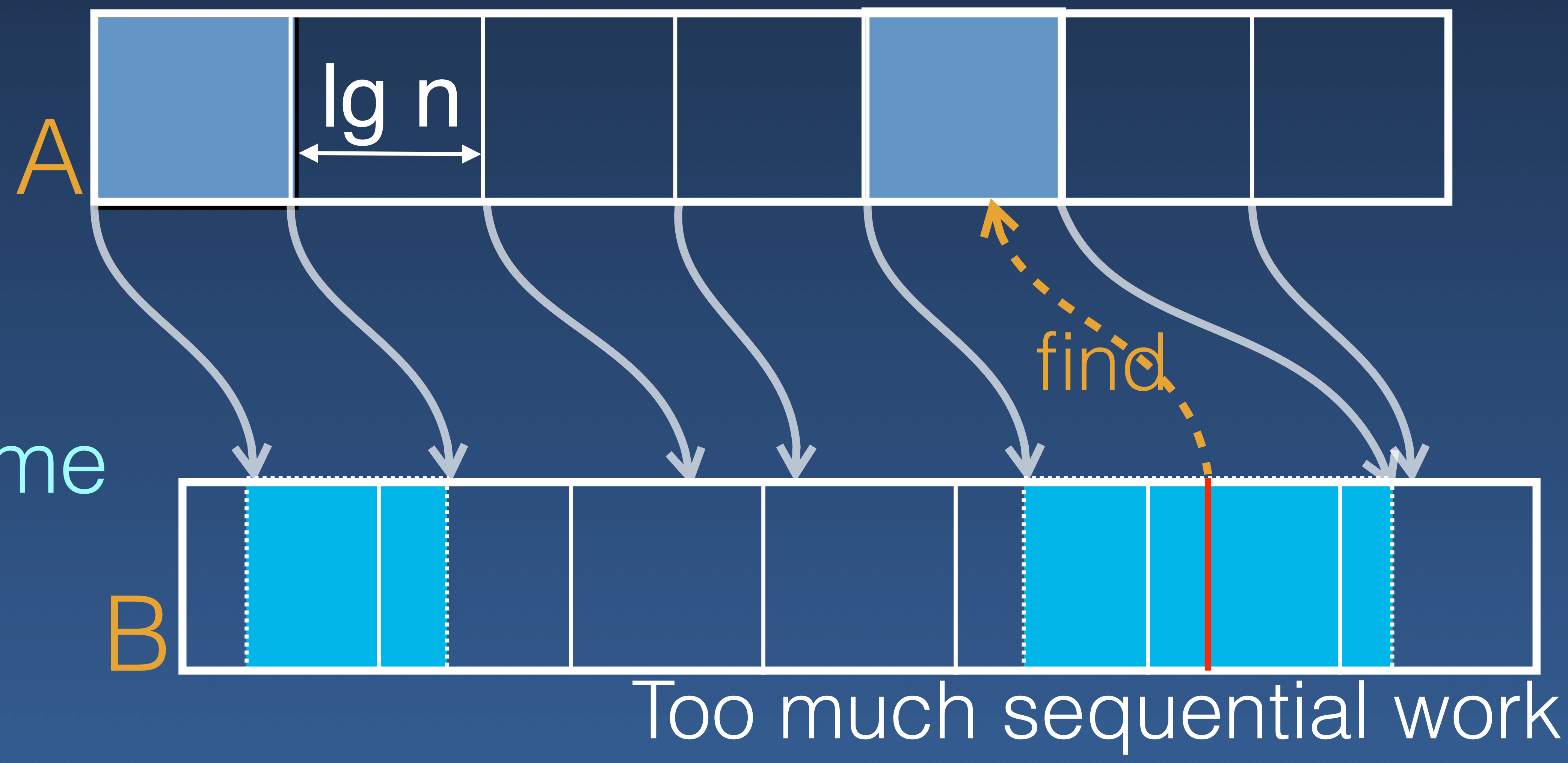


Too much sequential work

Towards Optimal Merge(A,B)

- Partition A and B into log n sized blocks
- Select from A, elements i * log n, i ∈ 0:n/log n
- Rank each selected element of A in B
    - ➡ Binary search
- Merge pairs of sub-sequences
    - ➡ If $|B_i| \leq \log(n)$, Sequential merge in O(log n) time
    - ➡ Otherwise, partition B_i into log n blocks
        - ▸ And also subdivide A_i into sub-sub-sequences

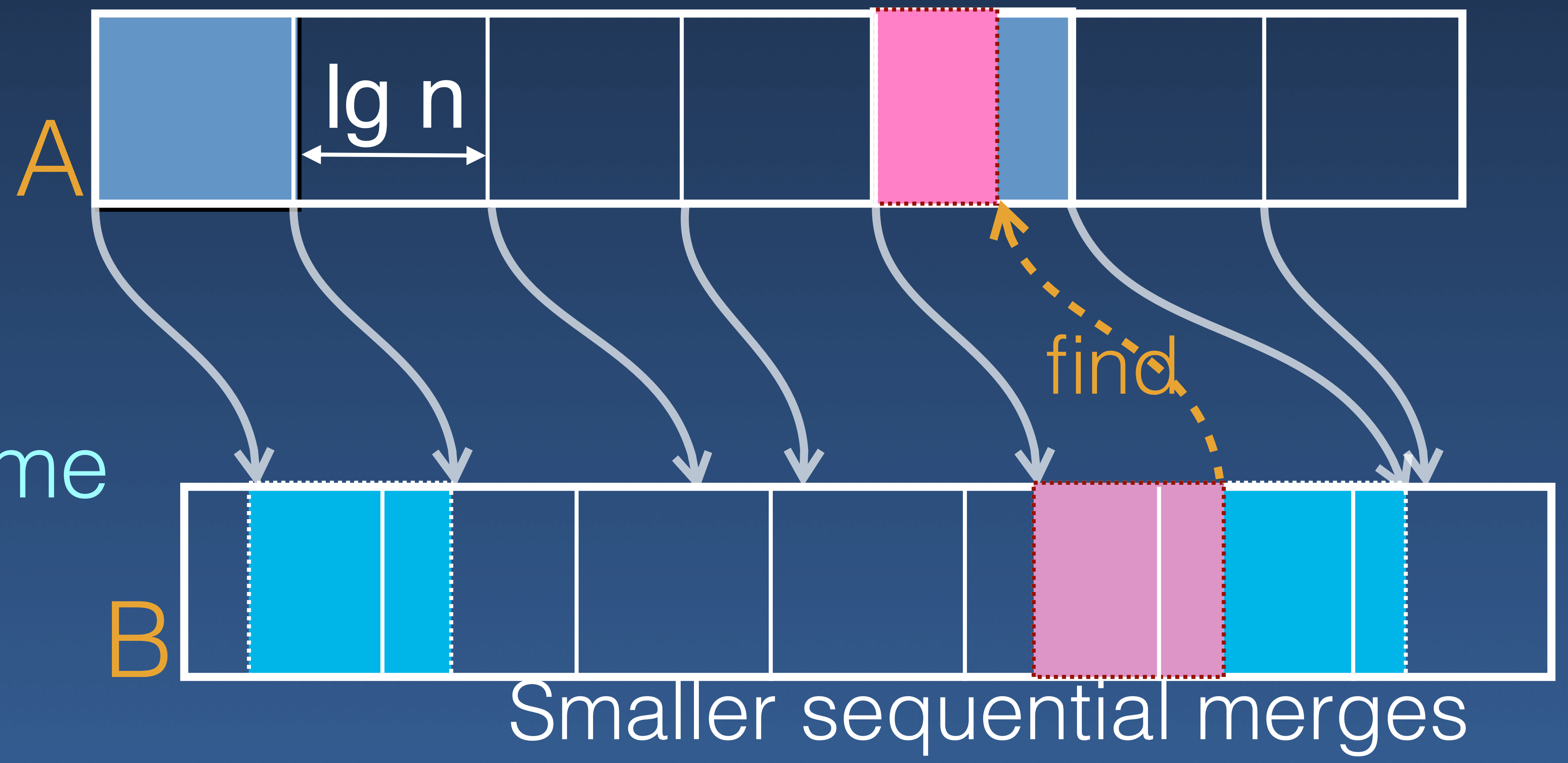
Data Partitioning Technique



Towards Optimal Merge(A,B)

- Partition A and B into log n sized blocks
- Select from A, elements i * log n, i ∈ 0:n/log n
- Rank each selected element of A in B
  - ➡ Binary search
- Merge pairs of sub-sequences
  - ➡ If $|B_i| \leq \log(n)$, Sequential merge in O(log n) time
  - ➡ Otherwise, partition B_i into log n blocks
    - ▸ And also subdivide A_i into sub-sub-sequences

Data Partitioning Technique



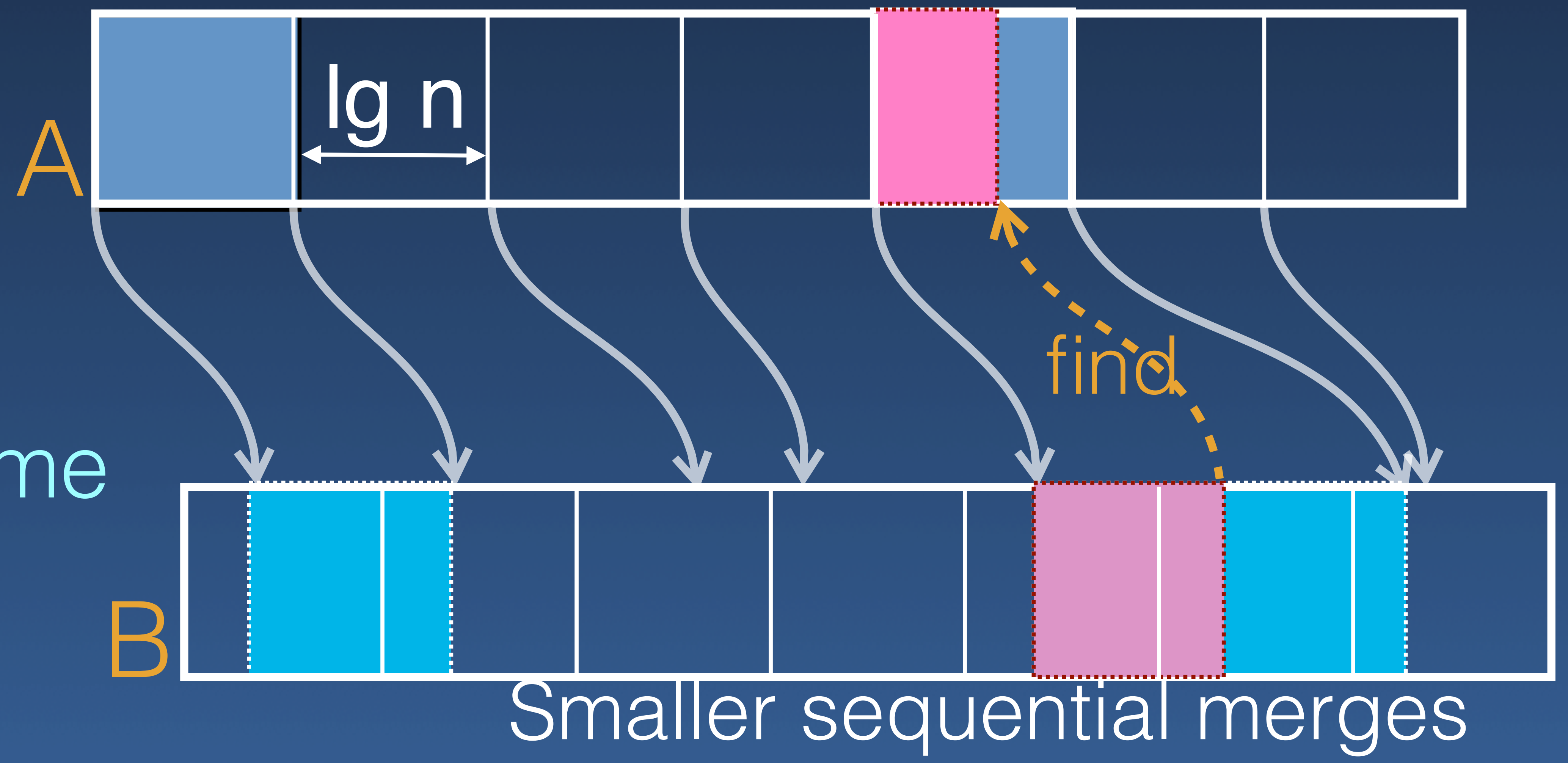
Towards Optimal Merge(A,B)

- Partition A and B into log n sized blocks
- Select from A, elements i * log n, i ∈ 0:n/log n
- Rank each selected element of A in B
  - ➡ Binary search
- Merge pairs of sub-sequences
  - ➡ If $|B_i| \le \log(n)$, Sequential merge in O(log n) time
  - ➡ Otherwise, partition B_i into log n blocks
    - ▸ And also subdivide A_i into sub-sub-sequences

Total time is O(log n)

Total work is O(n)

Data Partitioning Technique



Towards Optimal Merge(A,B)

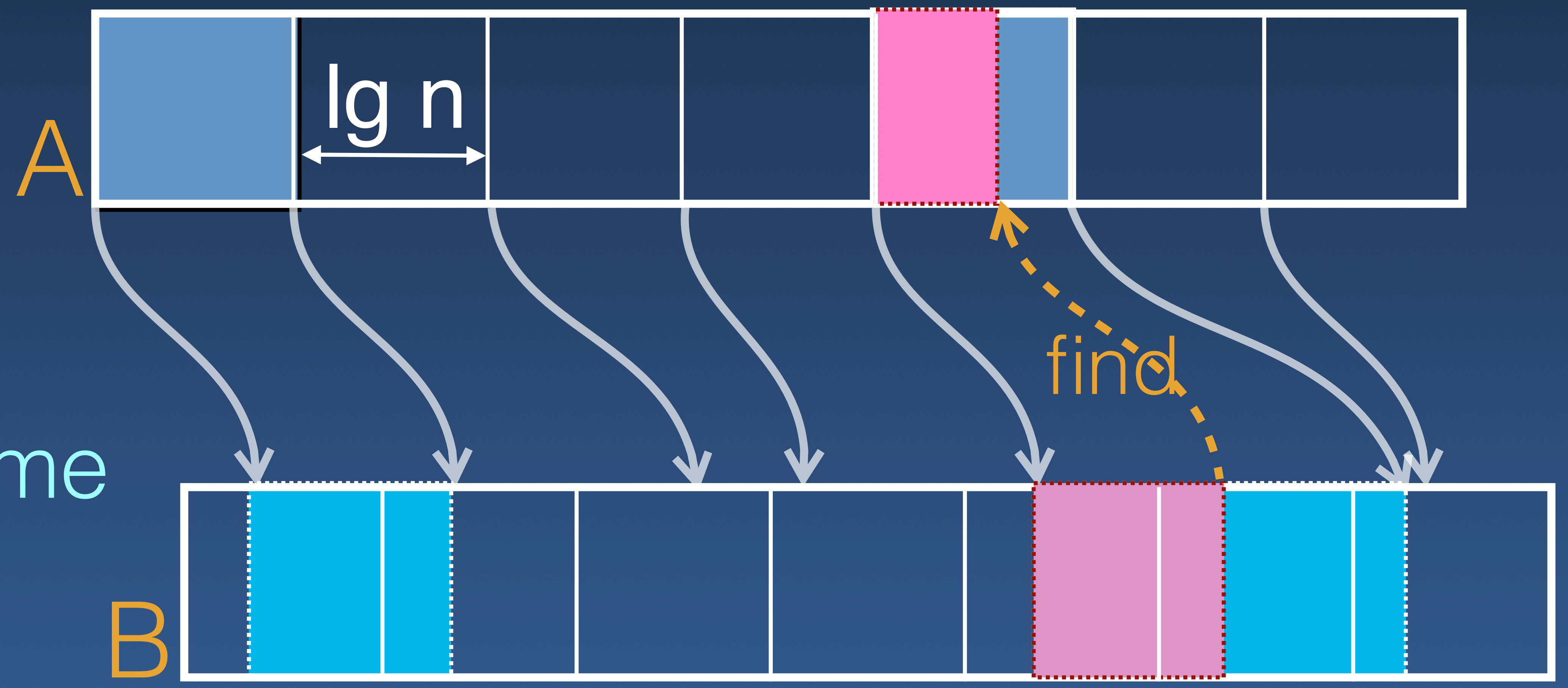
- Partition A and B into log n sized blocks
- Select from A, elements i * log n, i ∈ 0:n/log n
- Rank each selected element of A in B
  - ➡ Binary search
- Merge pairs of sub-sequences
  - ➡ If $|B_i| \leq \log(n)$, Sequential merge in O(log n) time
  - ➡ Otherwise, partition B_i into log n blocks
    - ▸ And also subdivide A_i into sub-sub-sequences

limited by

Total time is O(log n)

Total work is O(n)

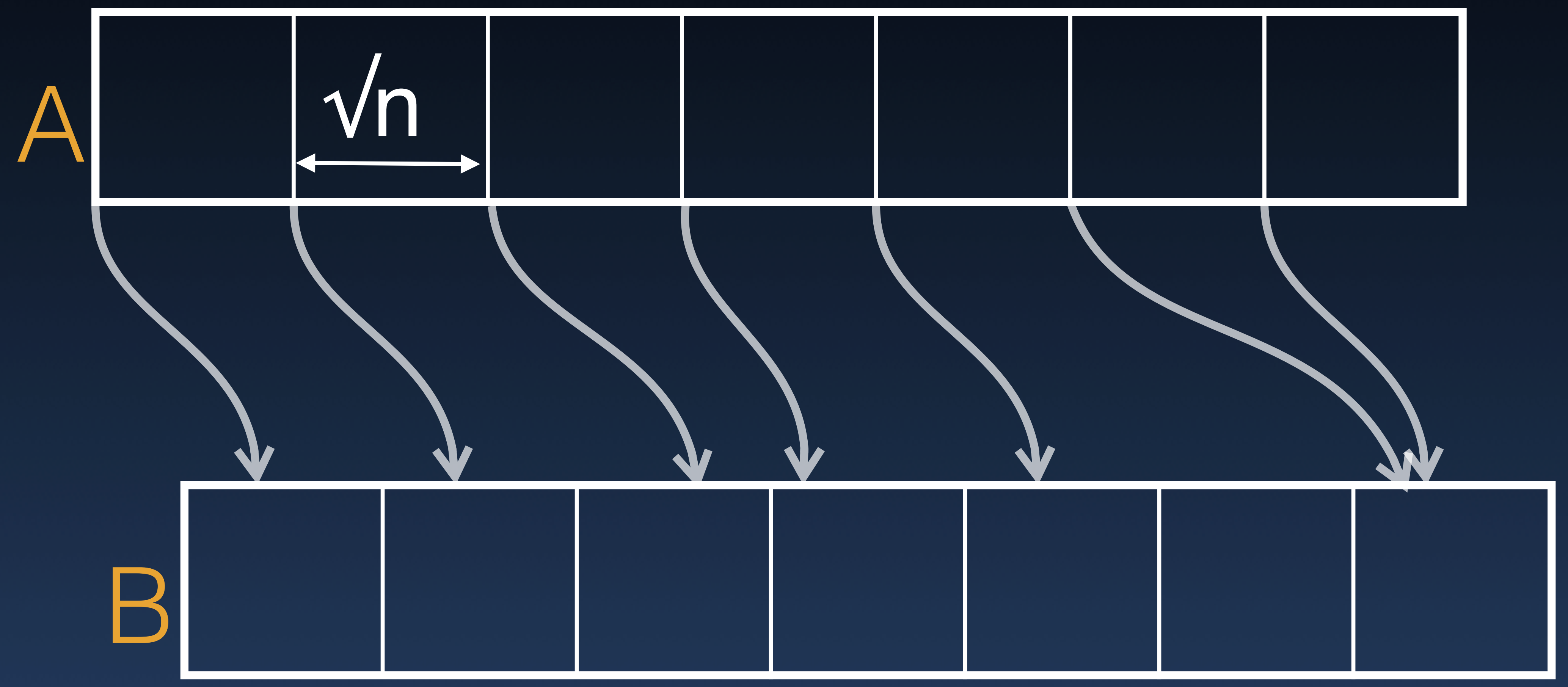
Data Partitioning Technique



Can we do better?

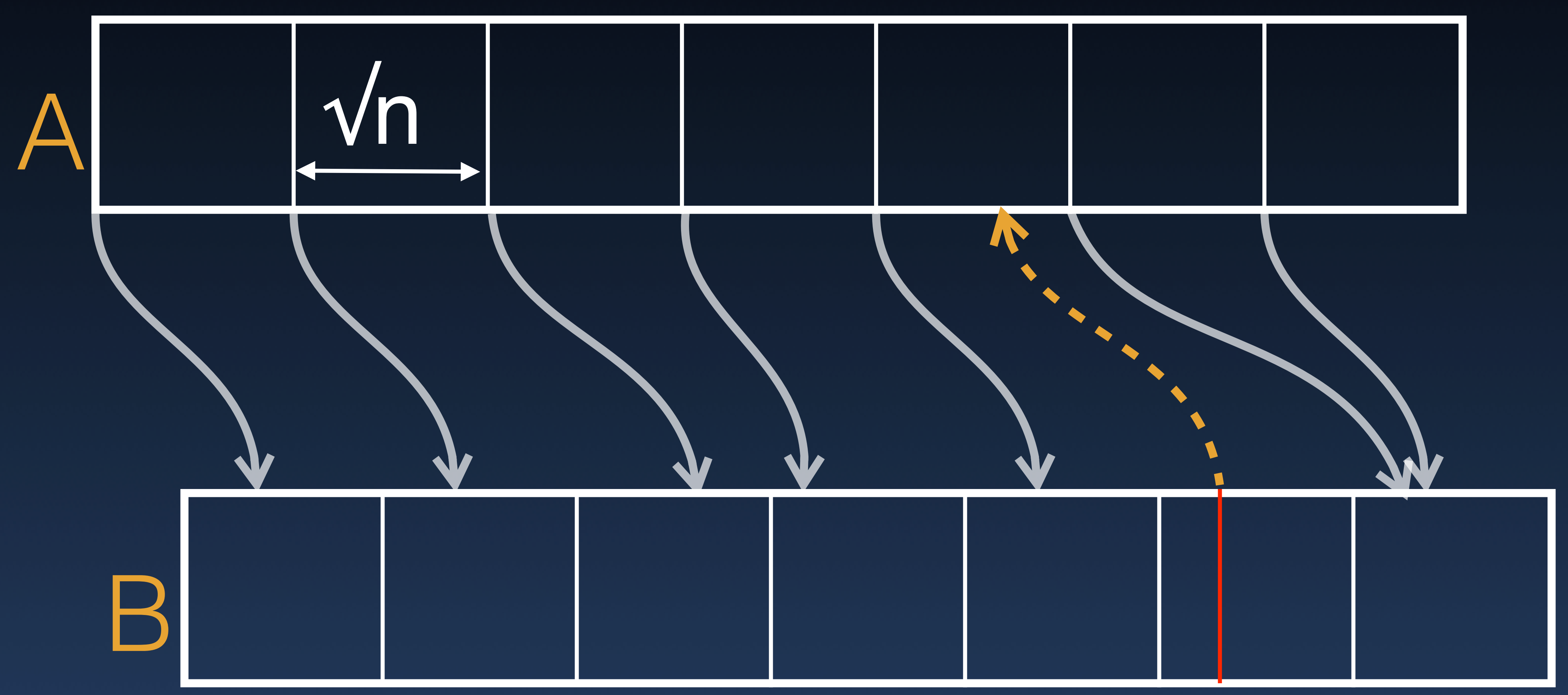
Fast Merge (A,B)

- Partition A and B into ***√n*** blocks each
- Select from A, elements i√n, i ∈ [0: √n)
- Rank each selected element of A in B
  ➡ √n Parallel searches, use √n processors for each search

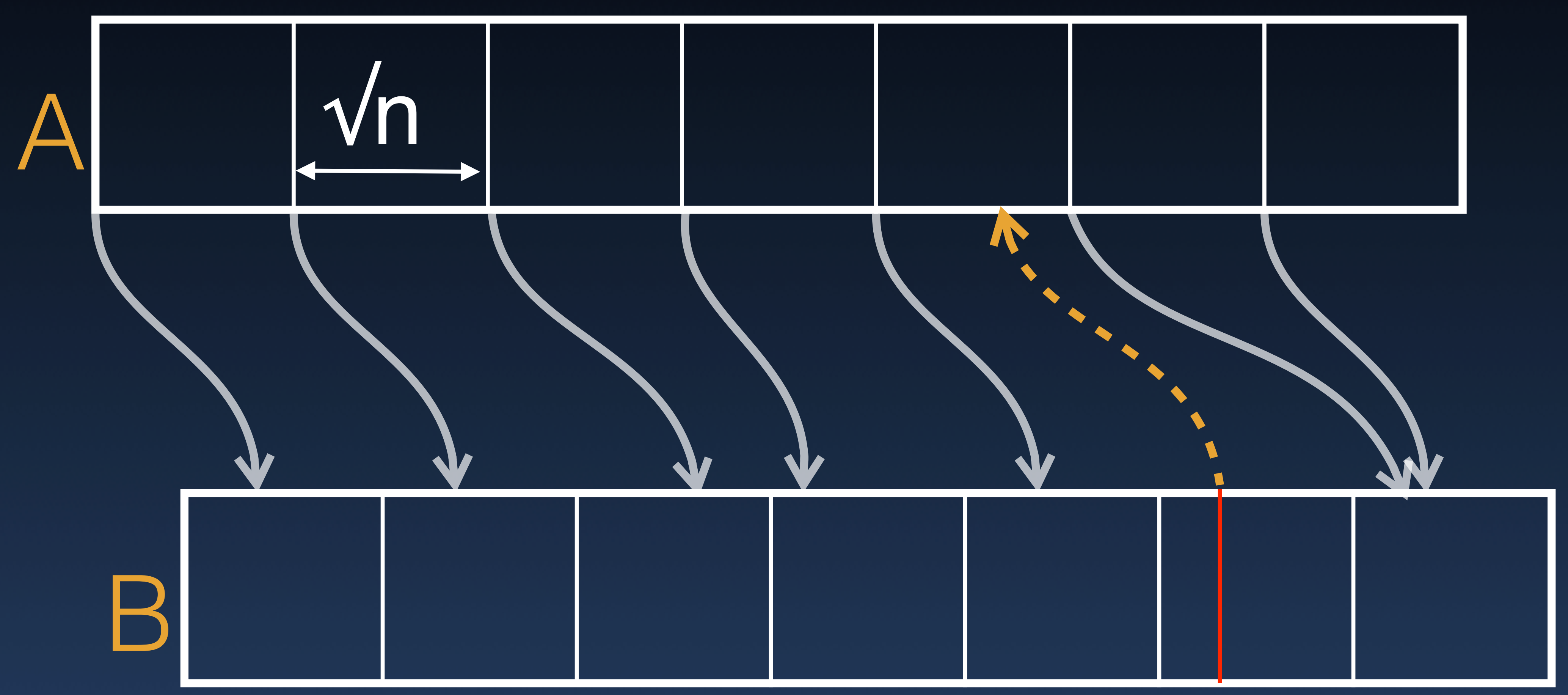


Fast Merge (A,B)

- Partition A and B into ***√n*** blocks each
- Select from A, elements i√n, i ∈ [0: √n)
- Rank each selected element of A in B
  - ➡ √n Parallel searches, use √n processors for each search
- Similarly rank √n selected elements from B in A



Fast Merge (A,B)



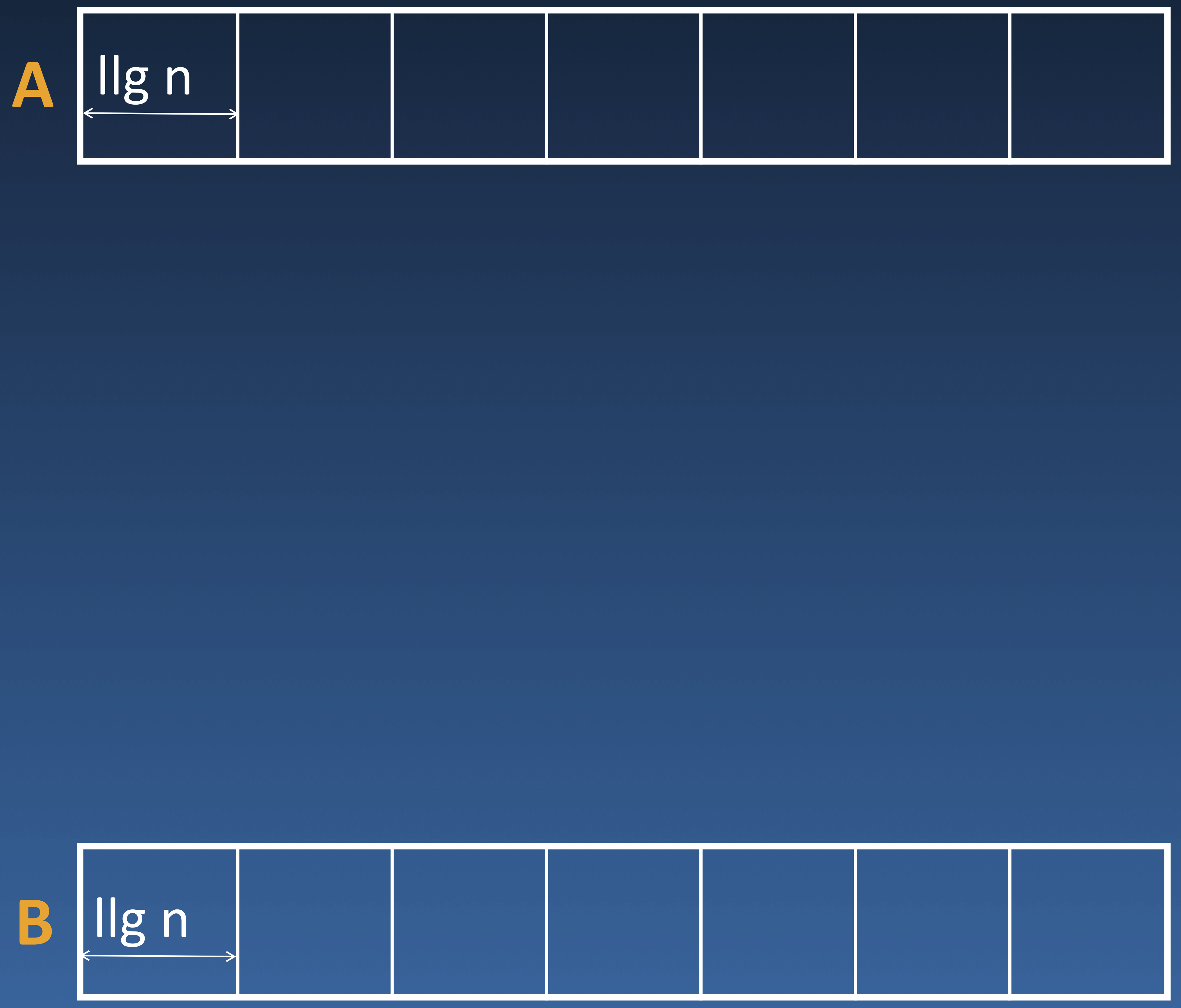
- Partition A and B into ***√n*** blocks each
- Select from A, elements i√n, i ∈ [0: √n)
- Rank each selected element of A in B
  - ➡ √n Parallel searches, use √n processors for each search
- Similarly rank √n selected elements from B in A
- Recursively merge pairs of sub-sequences
  - ➡ Total time: $T(n) = O(1)+T(\sqrt{n}) = O(\log \log n)$
  - ➡ Total work: $W(n) = O(n)+ \sqrt{n}\, W(\sqrt{n}) = O(n \log \log n)$

Fast, but too much work
Not work optimal

Subodh Kumar

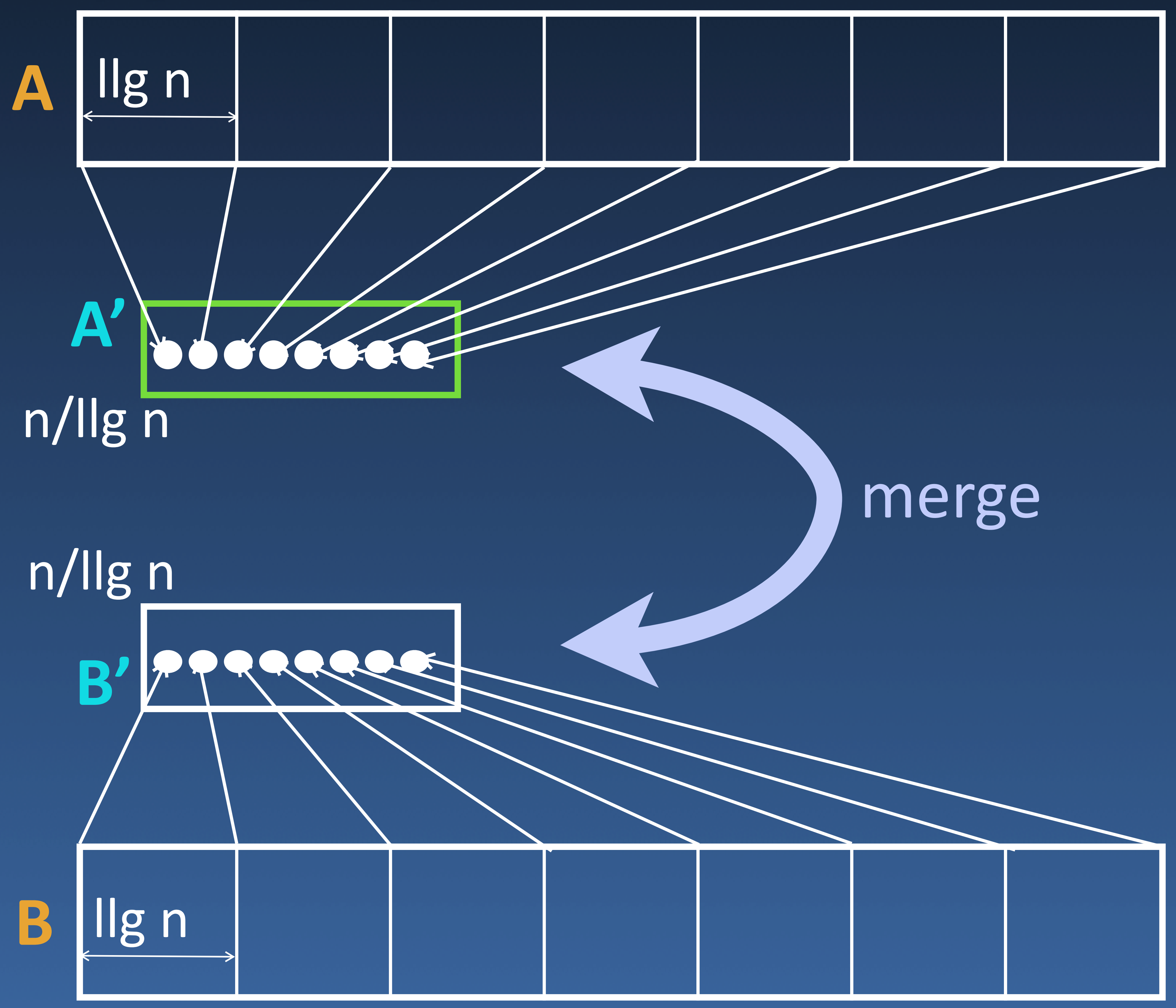
Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..



Optimal Merge (A,B)

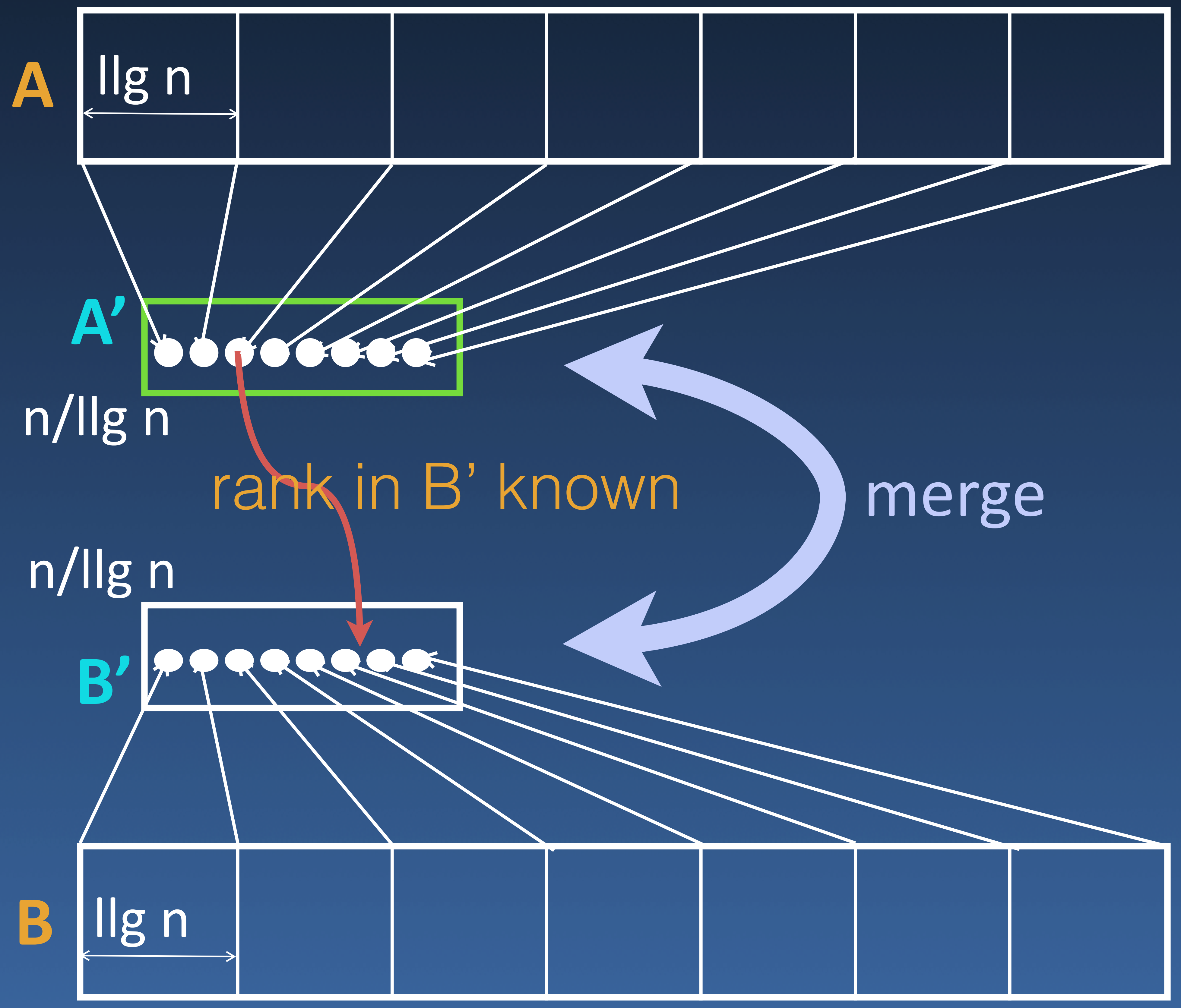
- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ $A_1, A_2, ..$
  - ➡ $B_1, B_2, ..$
- Select first element of each block
  - ➡ $A' = p_1, p_2 ..$
  - ➡ $B' = q_1, q_2 ..$



Subodh Kumar

Optimal Merge (A,B)

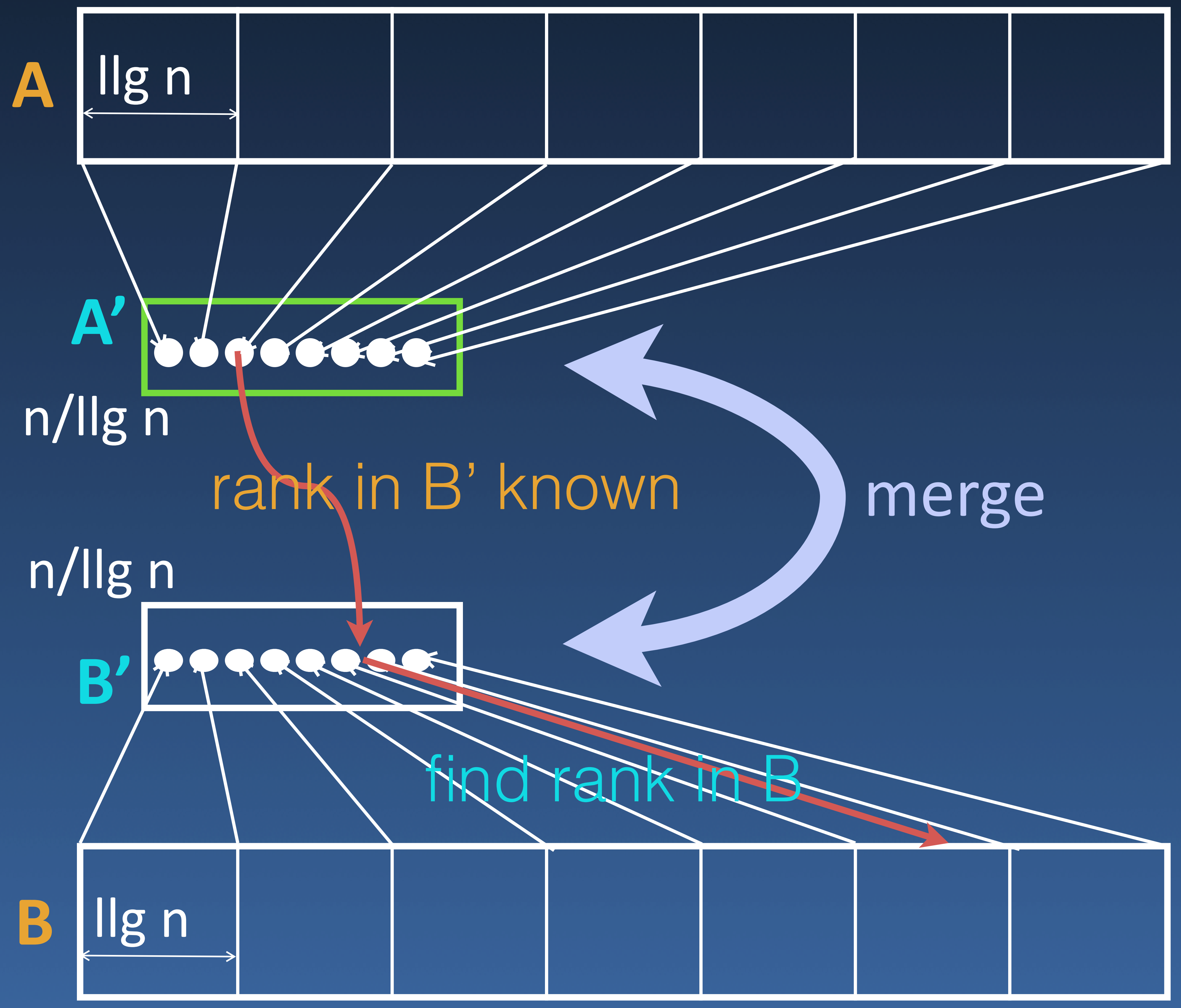
- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..
- Select first element of each block
  - ➡ A' = p_1, p_2 ..
  - ➡ B' = q_1, q_2 ..



Subodh Kumar

Optimal Merge (A,B)

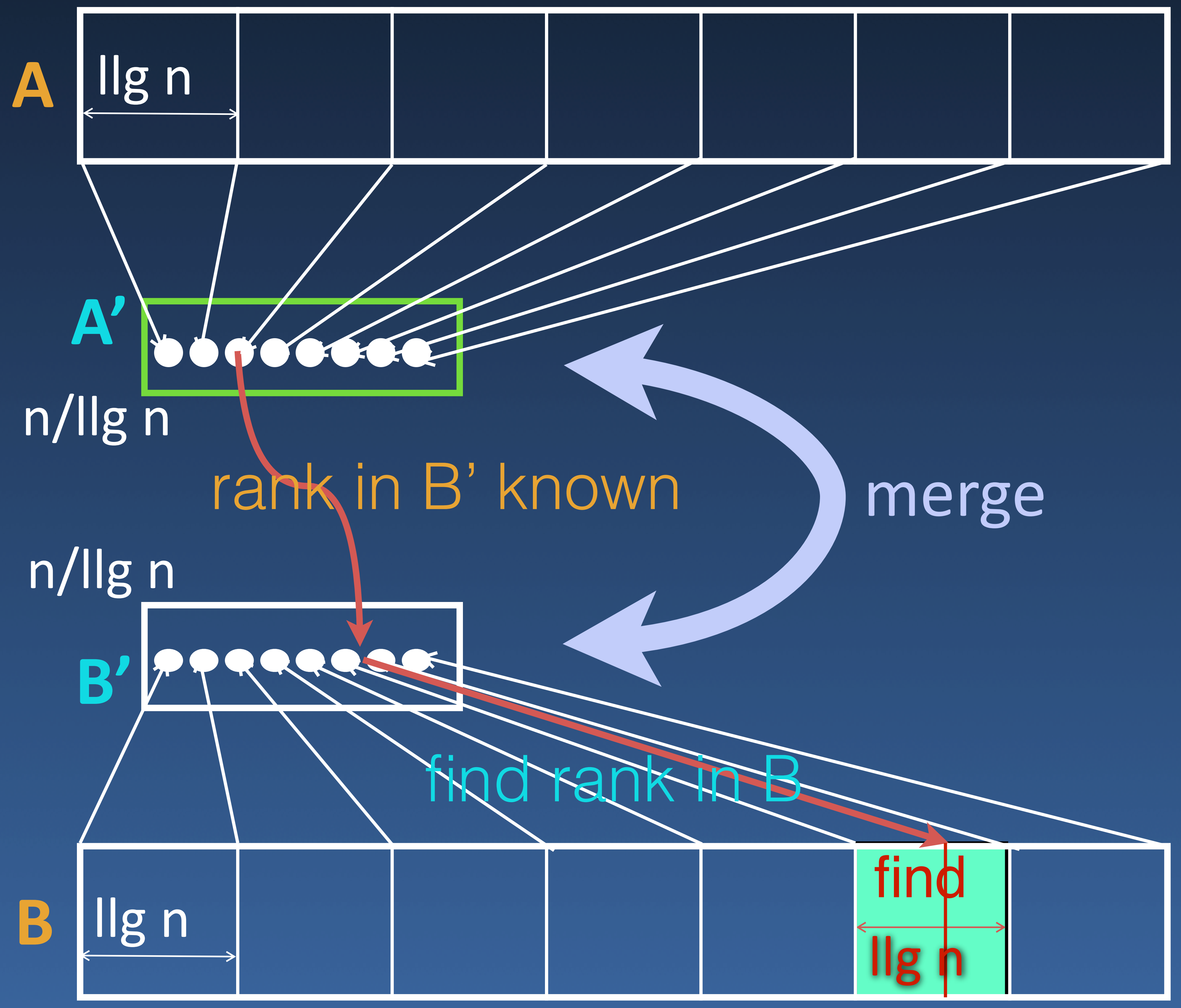
- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..
- Select first element of each block
  - ➡ A' = p_1, p_2 ..
  - ➡ B' = q_1, q_2 ..



Subodh Kumar

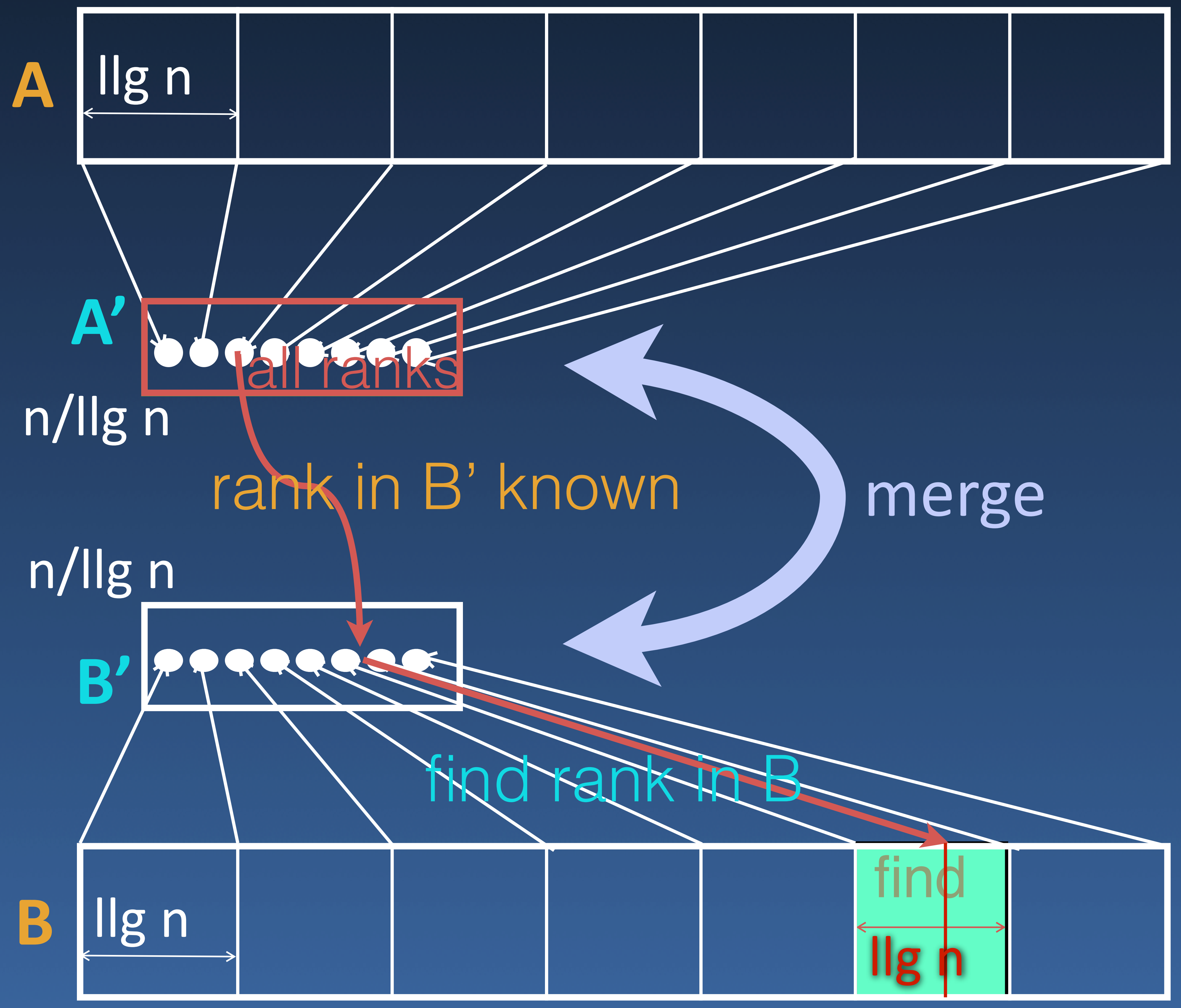
Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
    - ➡ $A_1, A_2, ..$
    - ➡ $B_1, B_2, ..$
- Select first element of each block
    - ➡ $A' = p_1, p_2 ..$
    - ➡ $B' = q_1, q_2 ..$



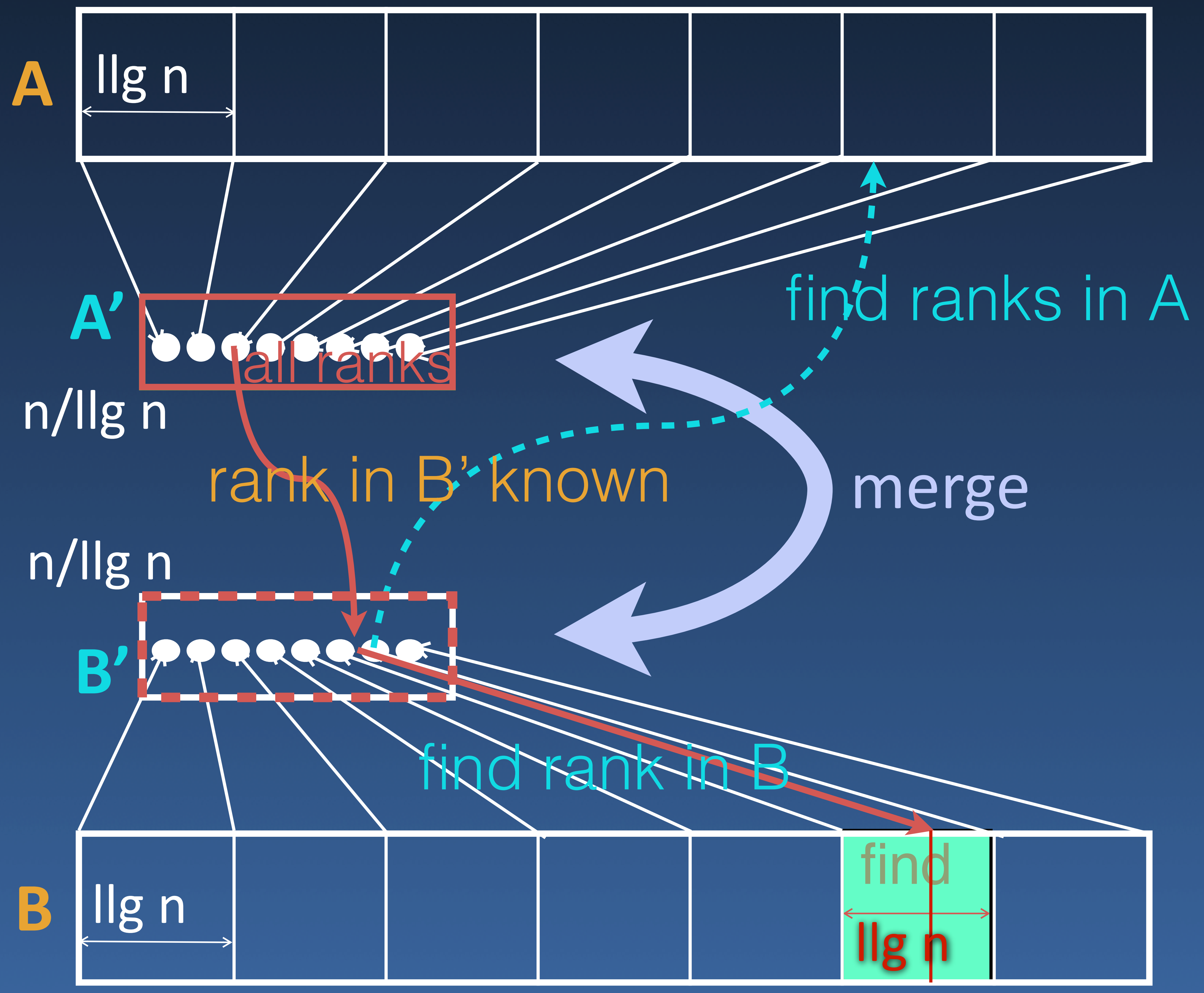
Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ $A_1, A_2, ..$
  - ➡ $B_1, B_2, ..$
- Select first element of each block
  - ➡ $A' = p_1, p_2 ..$
  - ➡ $B' = q_1, q_2 ..$



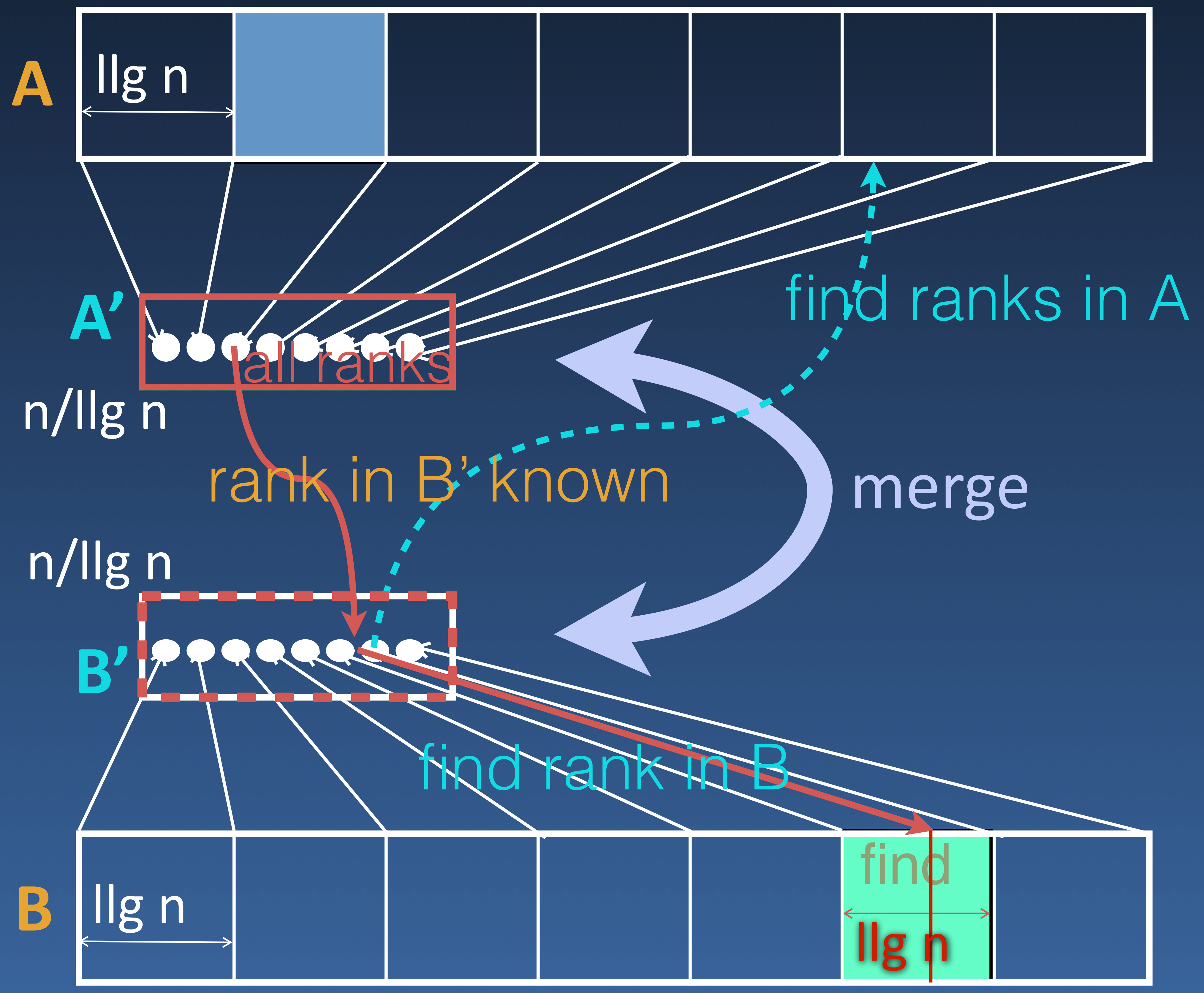
Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..
- Select first element of each block
  - ➡ A' = p_1, p_2 ..
  - ➡ B' = q_1, q_2 ..



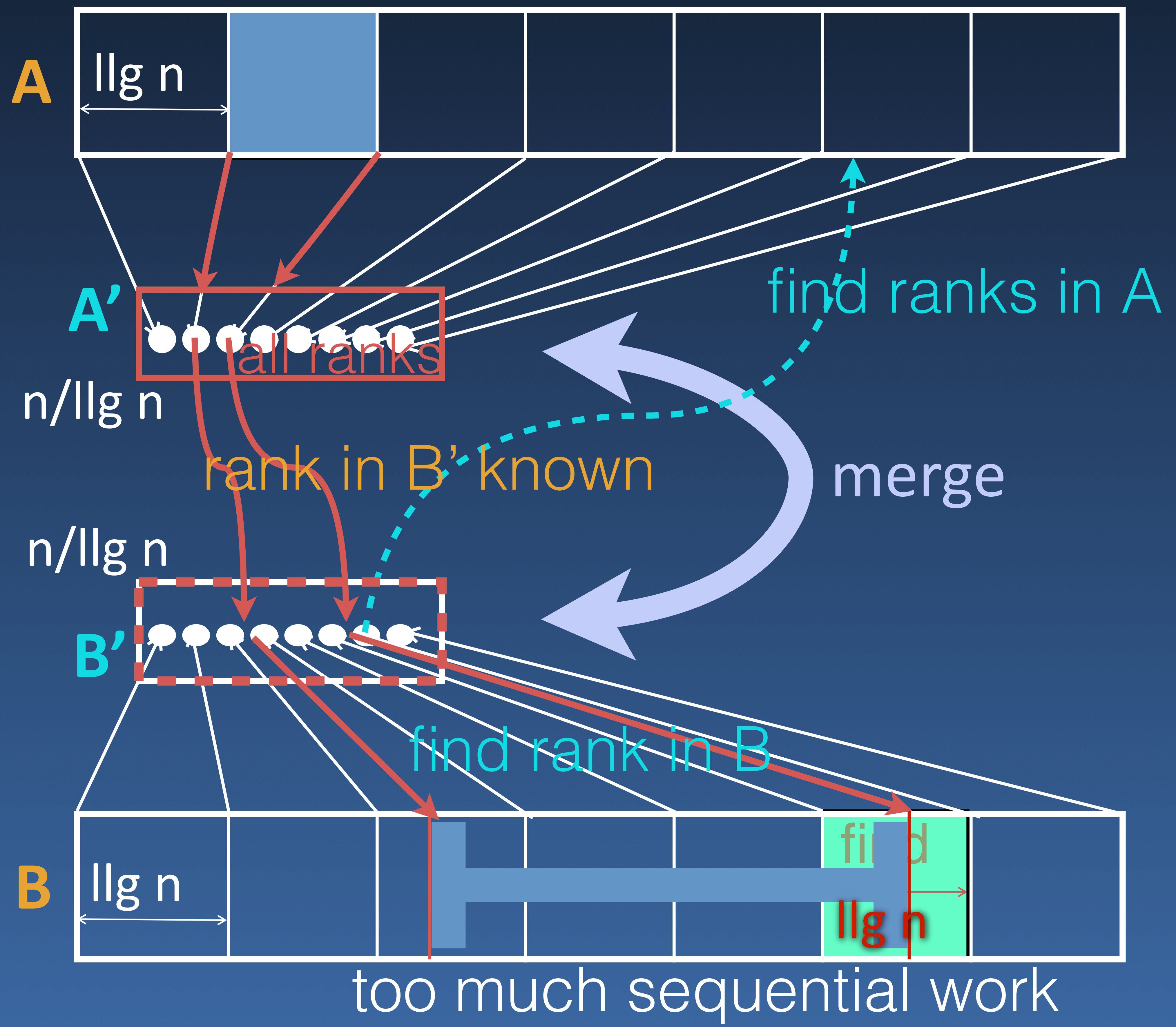
Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..
- Select first element of each block
  - ➡ A' = p_1, p_2 ..
  - ➡ B' = q_1, q_2 ..
- Now merge separated blocks of A and B



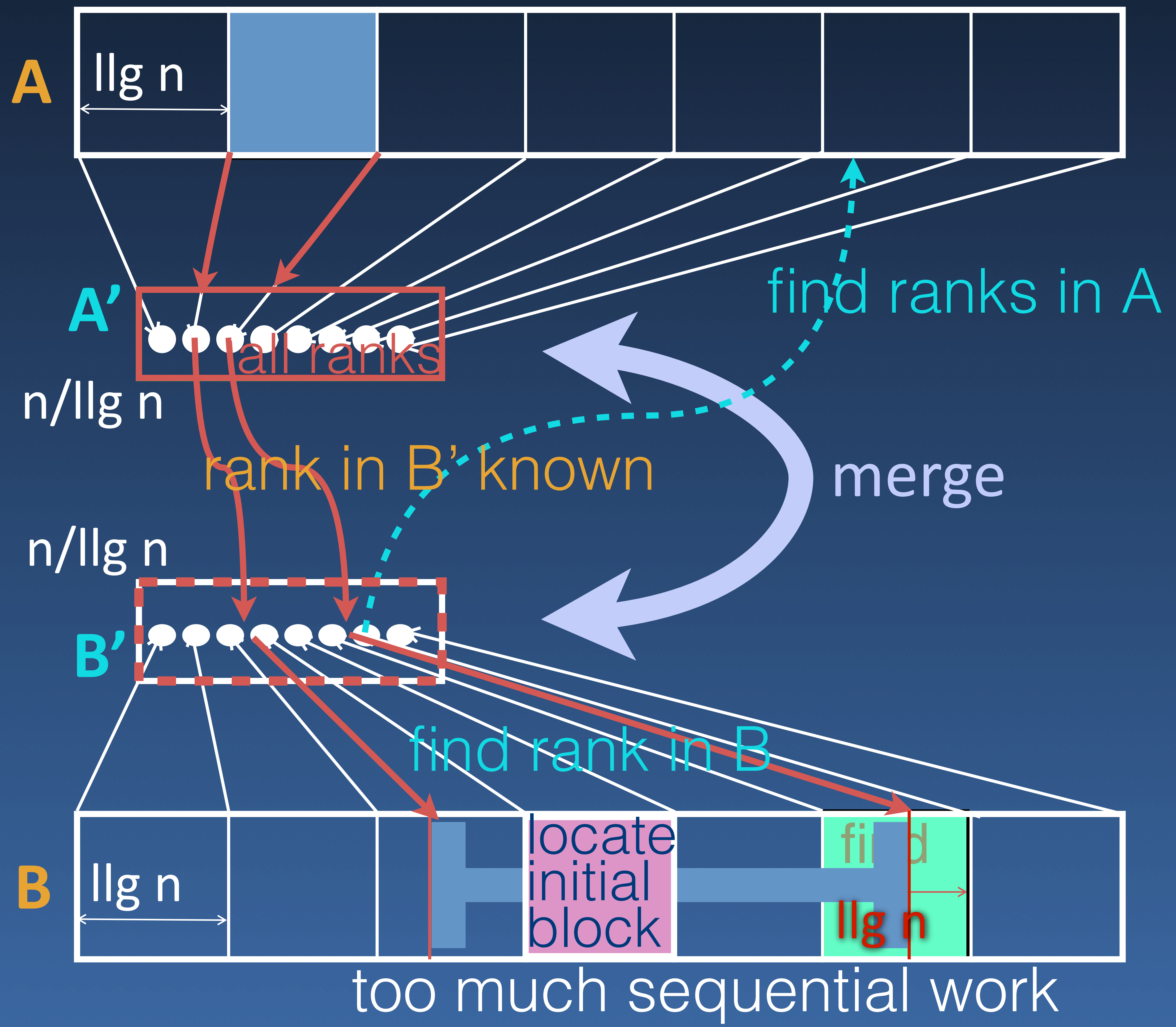
Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..
- Select first element of each block
  - ➡ A' = p_1, p_2 ..
  - ➡ B' = q_1, q_2 ..
- Now merge separated blocks of A and B



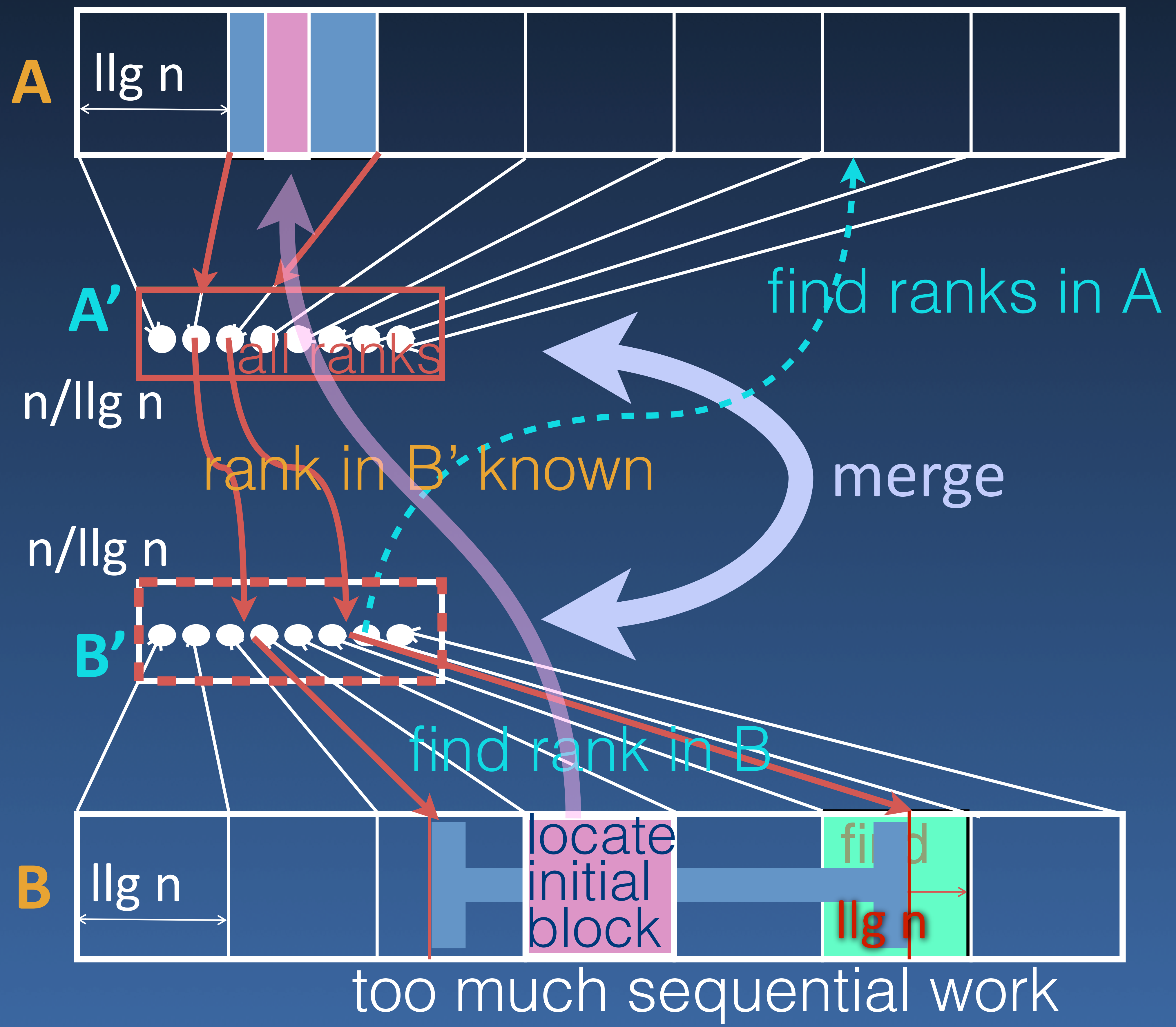
Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..
- Select first element of each block
  - ➡ A' = p_1, p_2 ..
  - ➡ B' = q_1, q_2 ..
- Now merge separated blocks of A and B



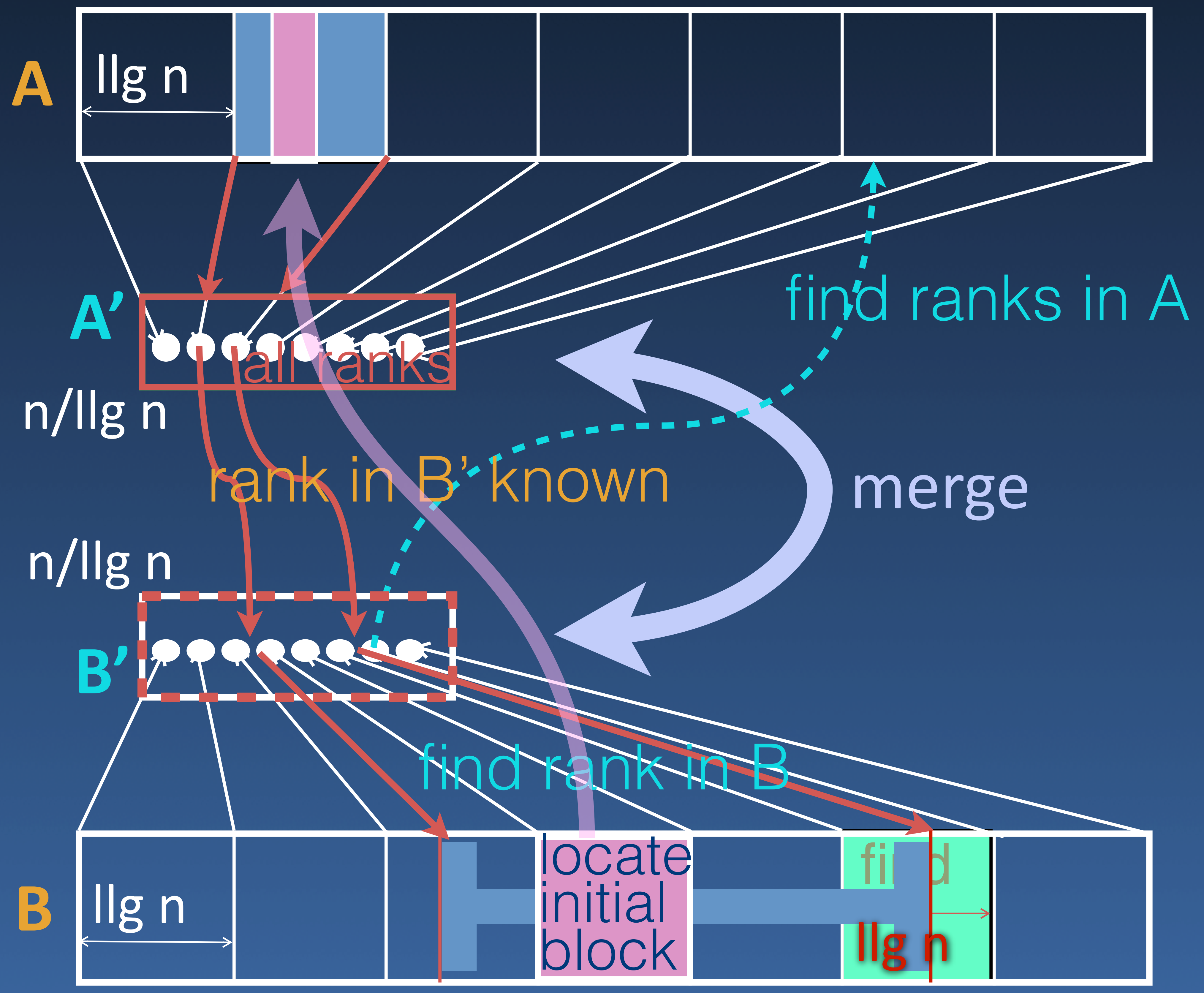
Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..
- Select first element of each block
  - ➡ A' = p_1, p_2 ..
  - ➡ B' = q_1, q_2 ..
- Now merge separated blocks of A and B



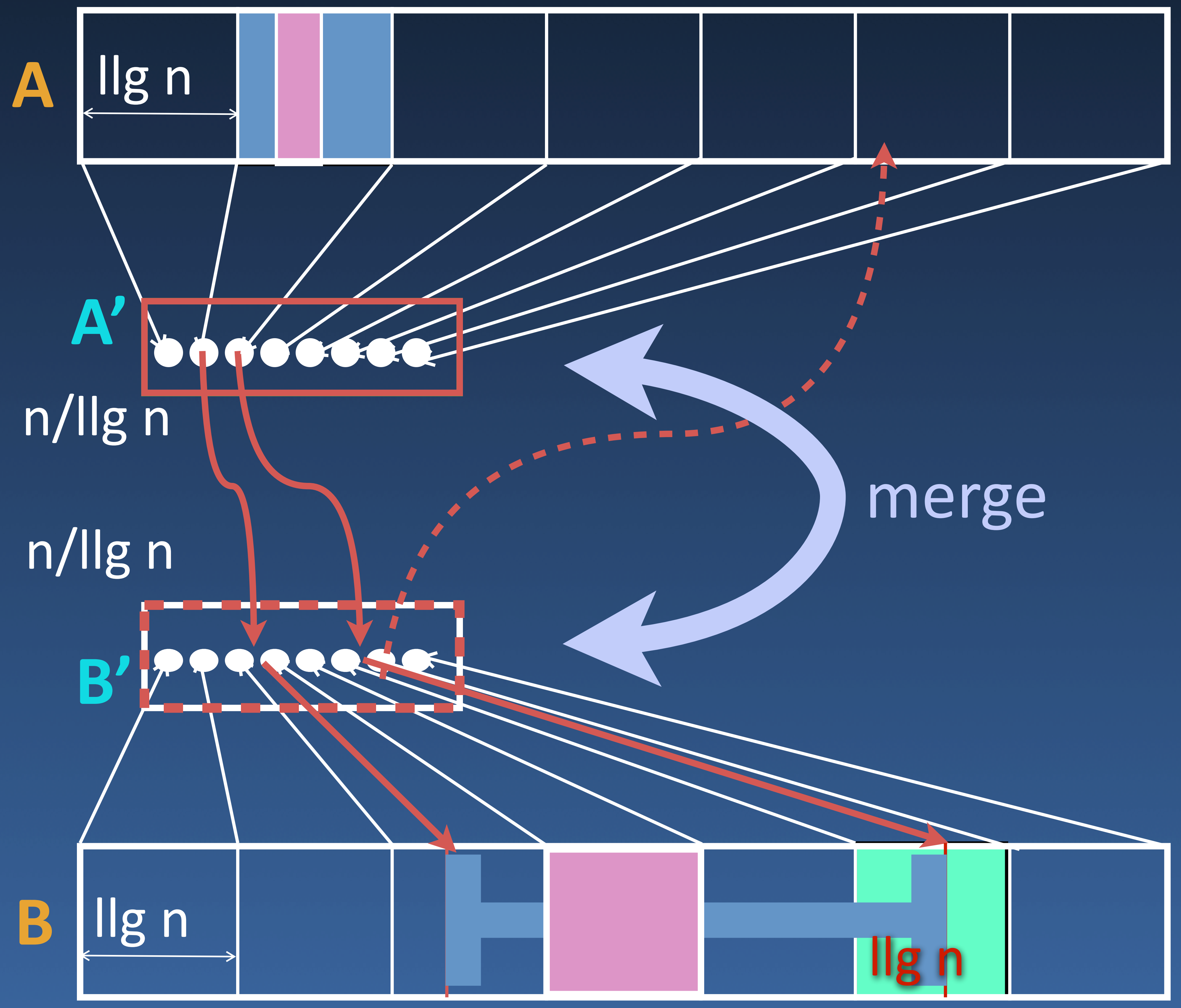
Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..
- Select first element of each block
  - ➡ A' = p_1, p_2 ..
  - ➡ B' = q_1, q_2 ..
- Merge O(llg n) sized blocks O(n/llg n) times



Optimal Merge (A,B)
(Analysis)

1. Merge A' and B' – find Rank(A':B'), Rank(B':A')
   - ➡ Use fast non-optimal algorithm
     - Time = O(log log n), Work = O(n)
2. Compute Rank(A':B) and Rank(B':A)
   - ➡ If Rank(p_i, B) is r_i, p_i lies in block B_{r_i}
     - Sequentially: Time = O(log log n), Work = O(n)
3. Compute ranks of remaining elements
   - Sequentially: Time = O(log log n), Work = O(n)


A
llg n
A'
n/llg n
merge
n/llg n
B'
B
llg n
llg n