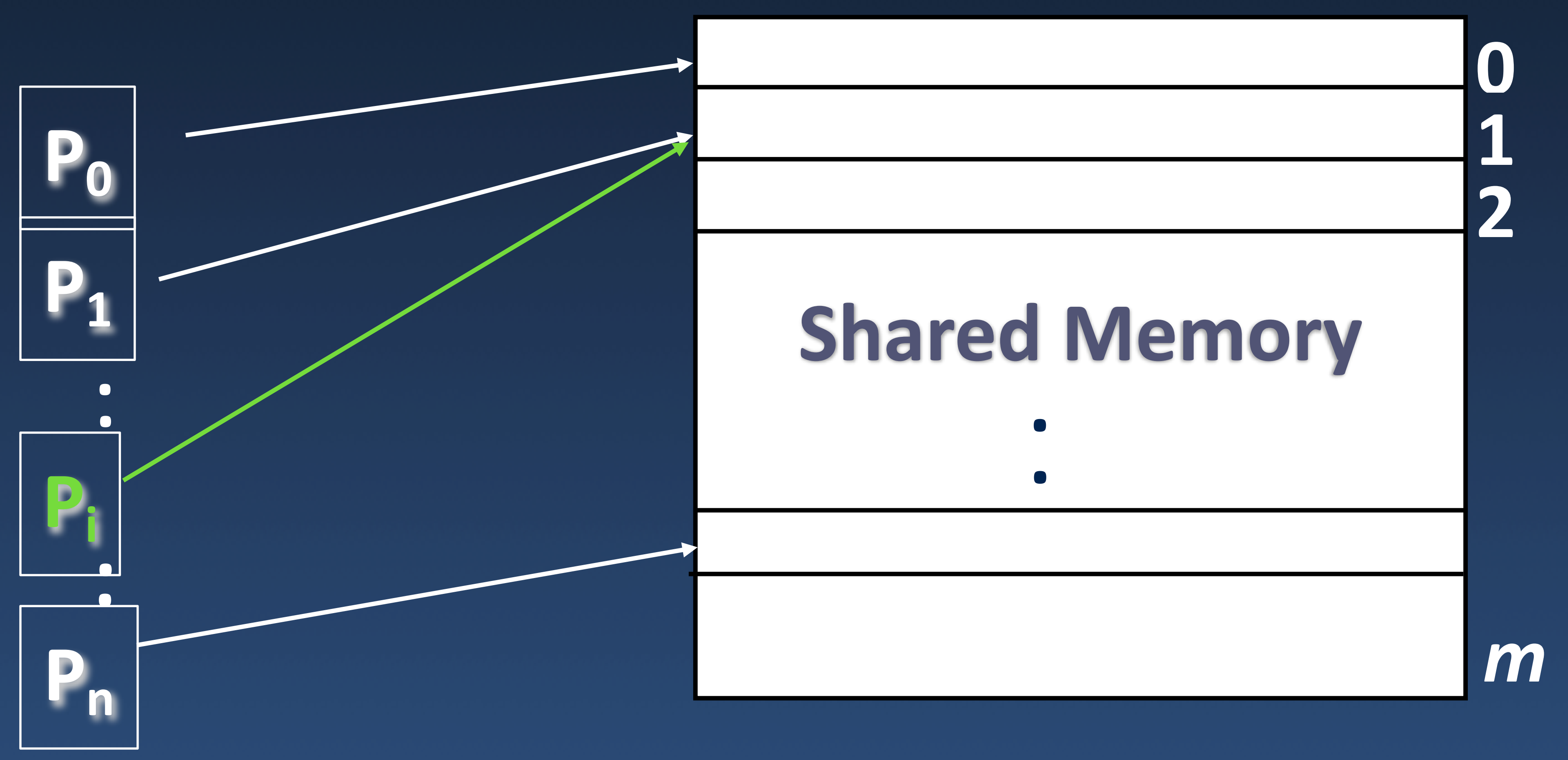
COL380

Introduction to
Parallel & Distributed Programming

PRAM Recap

Recall PRAM



Synchronous steps:

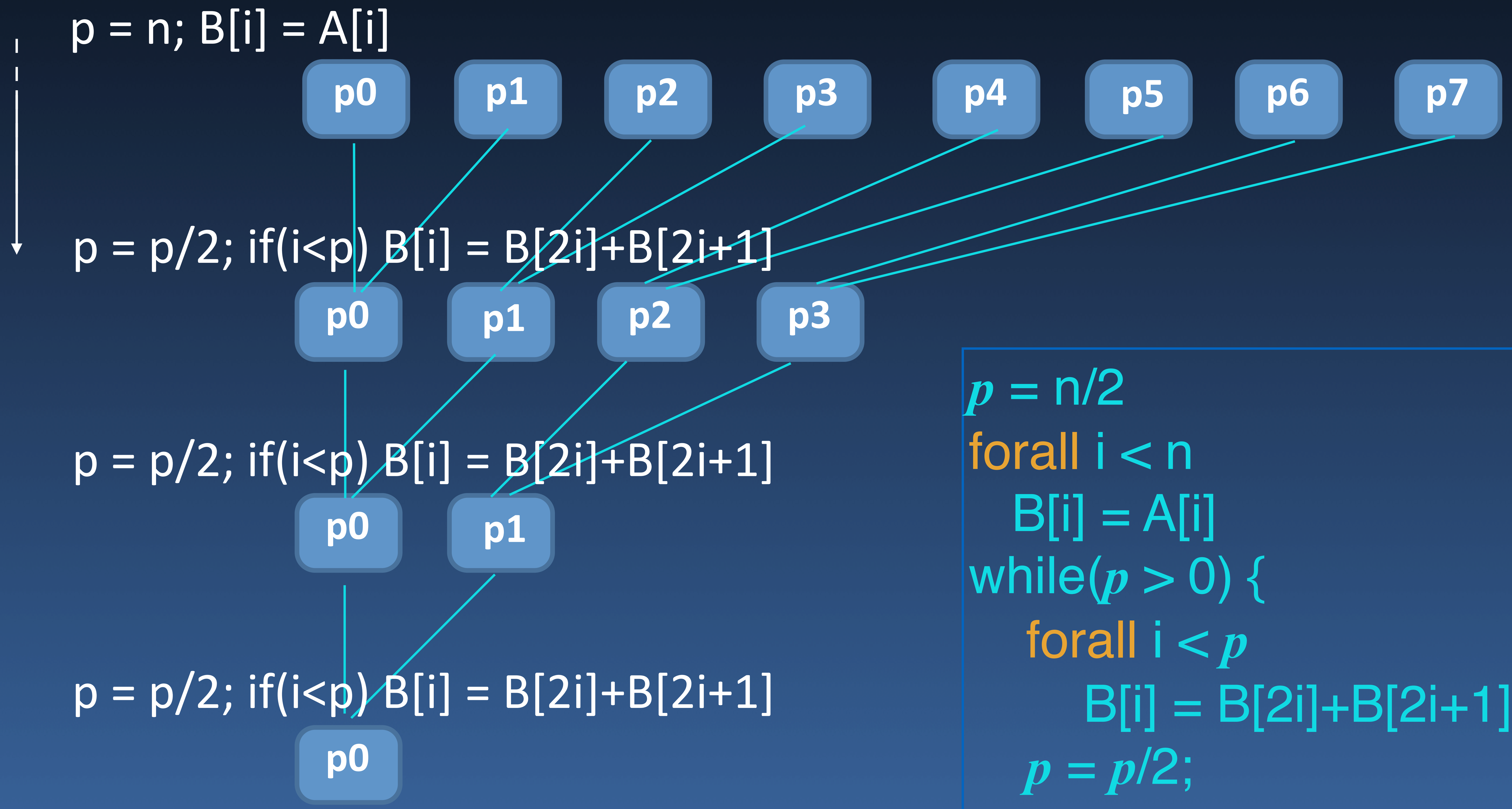
- Read
- Compute
- Write

- EREW
- CREW
- CRCW

Subodh Kumar

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

Parallel Addition




```
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;
}
```

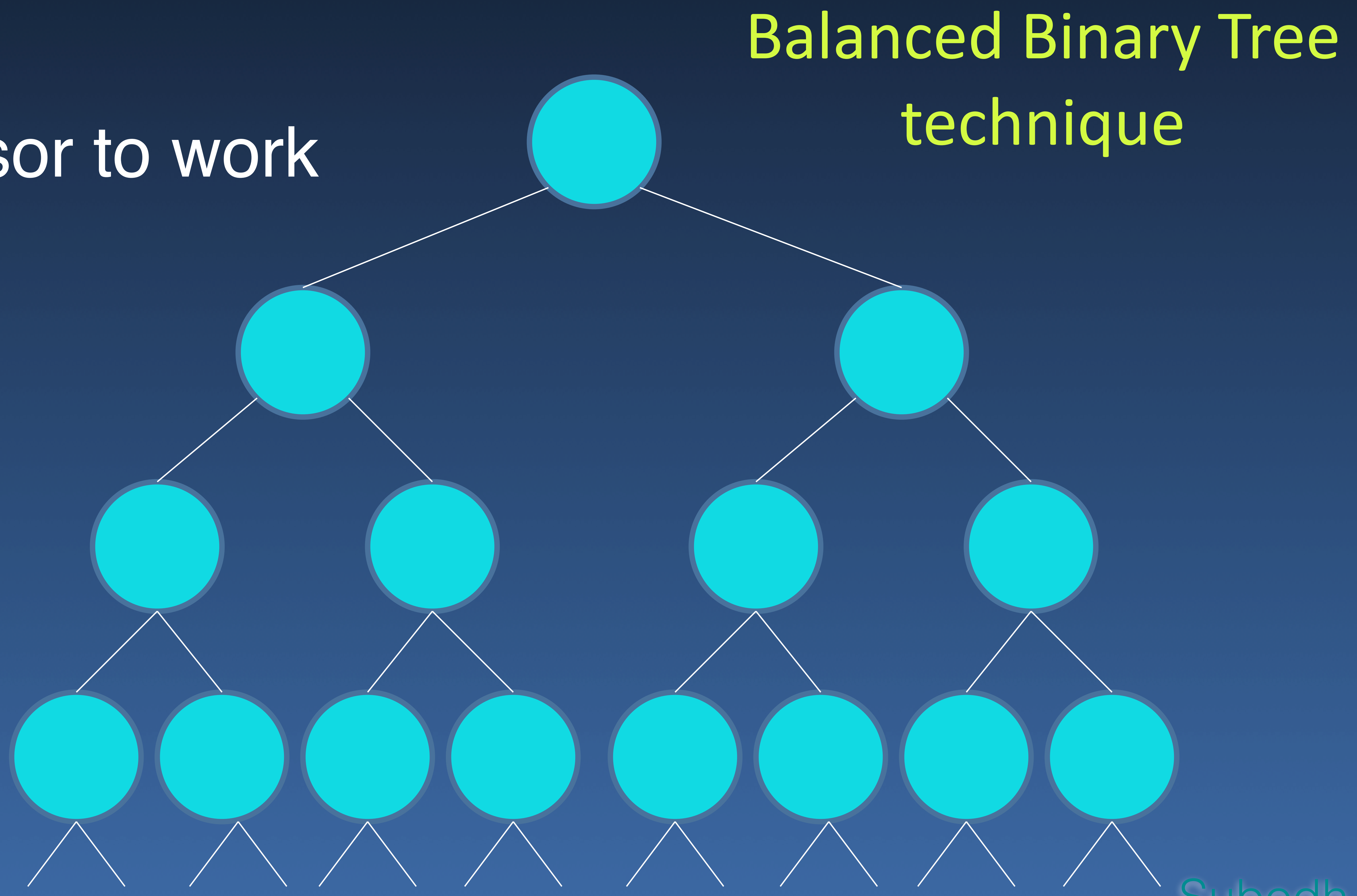
(assumes n is a power of 2)

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

Subodh Kumar

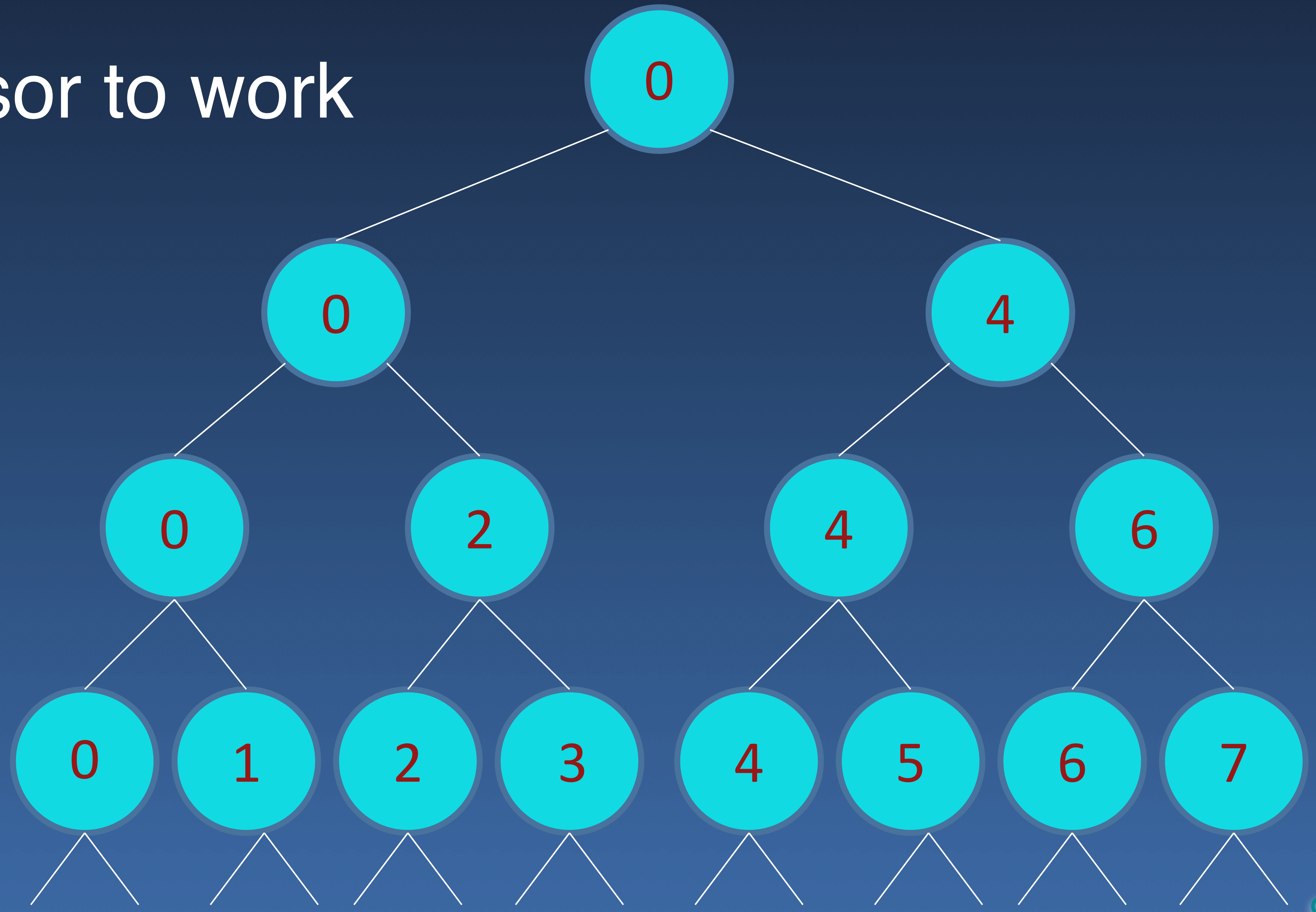
Reduction

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

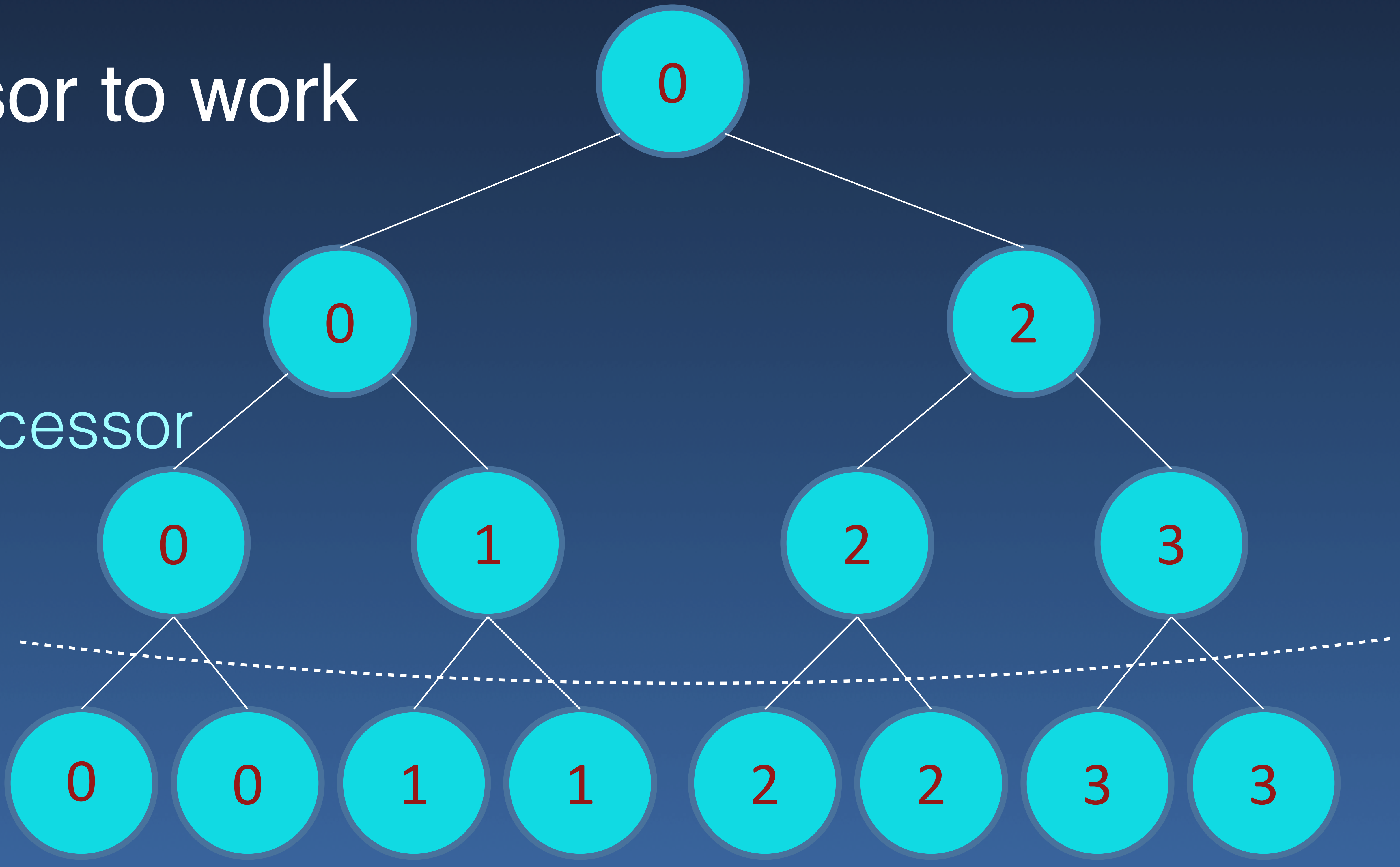


Reduction

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



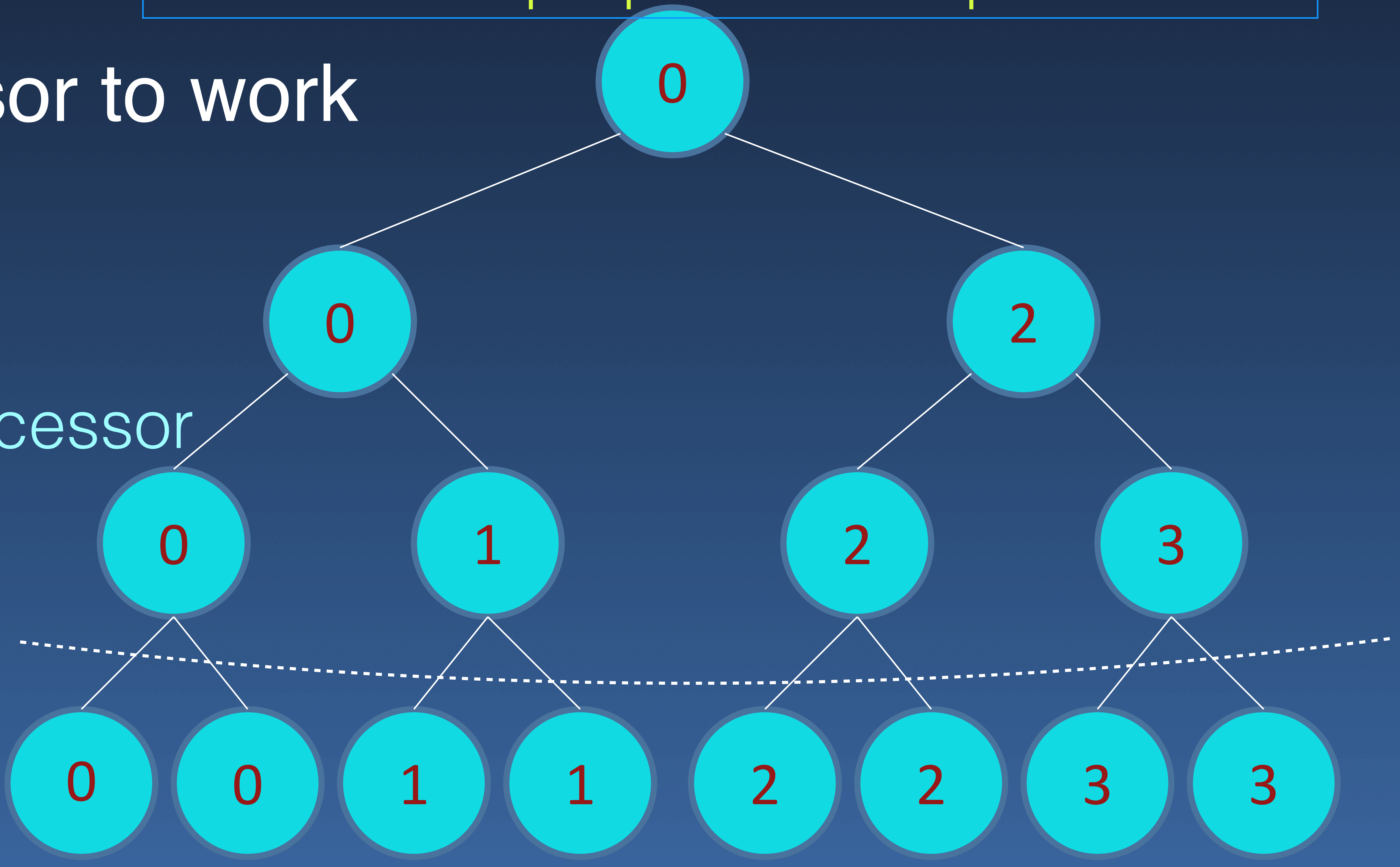
- n operands ⇒ log n steps
- Total work = O(n)
- How do you map processor to work
  - ➡ Consider $p < n$
  - ➡ Locally reduce at each processor
  - ➡ $p/2^i$ processors per step
  - ➡ step i: if $!(id\%2^i)$



Reduction

- Count the number of operations
  - ➡ Then map to p processors
- Sometime, convenient to start with p
  - ➡ And map operation to 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 $!(id\%2^i)$

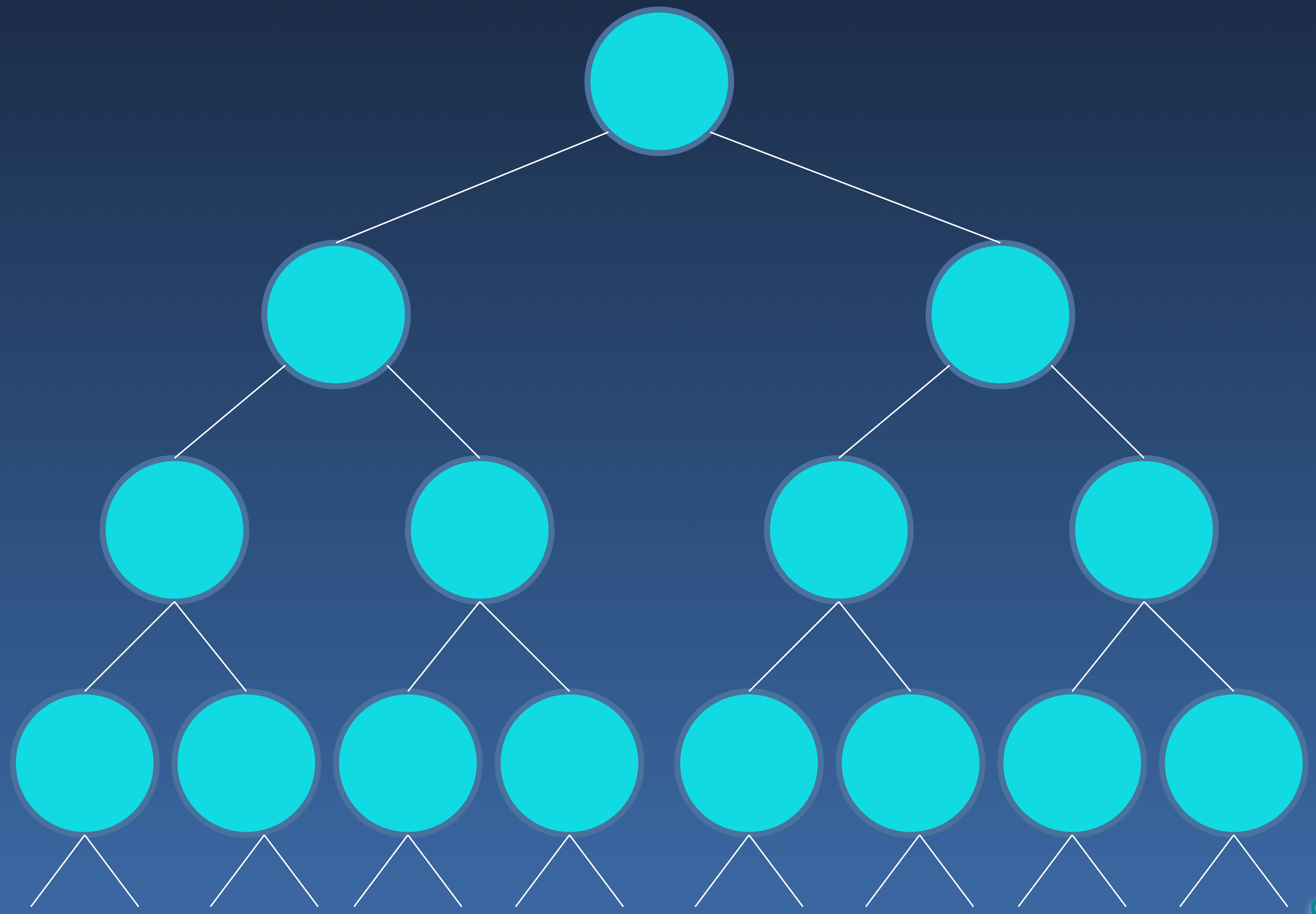


Subodh Kumar

Prefix Sums

Input: x

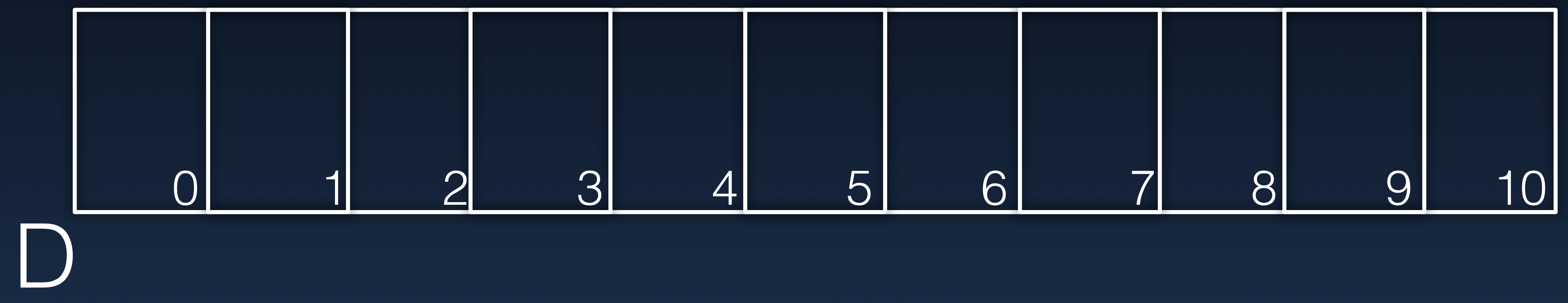
- P[0] = x[0]
- For i = 1 to n-1
  - ➡ P[i] = P[i-1] + x[i]



Subodh Kumar

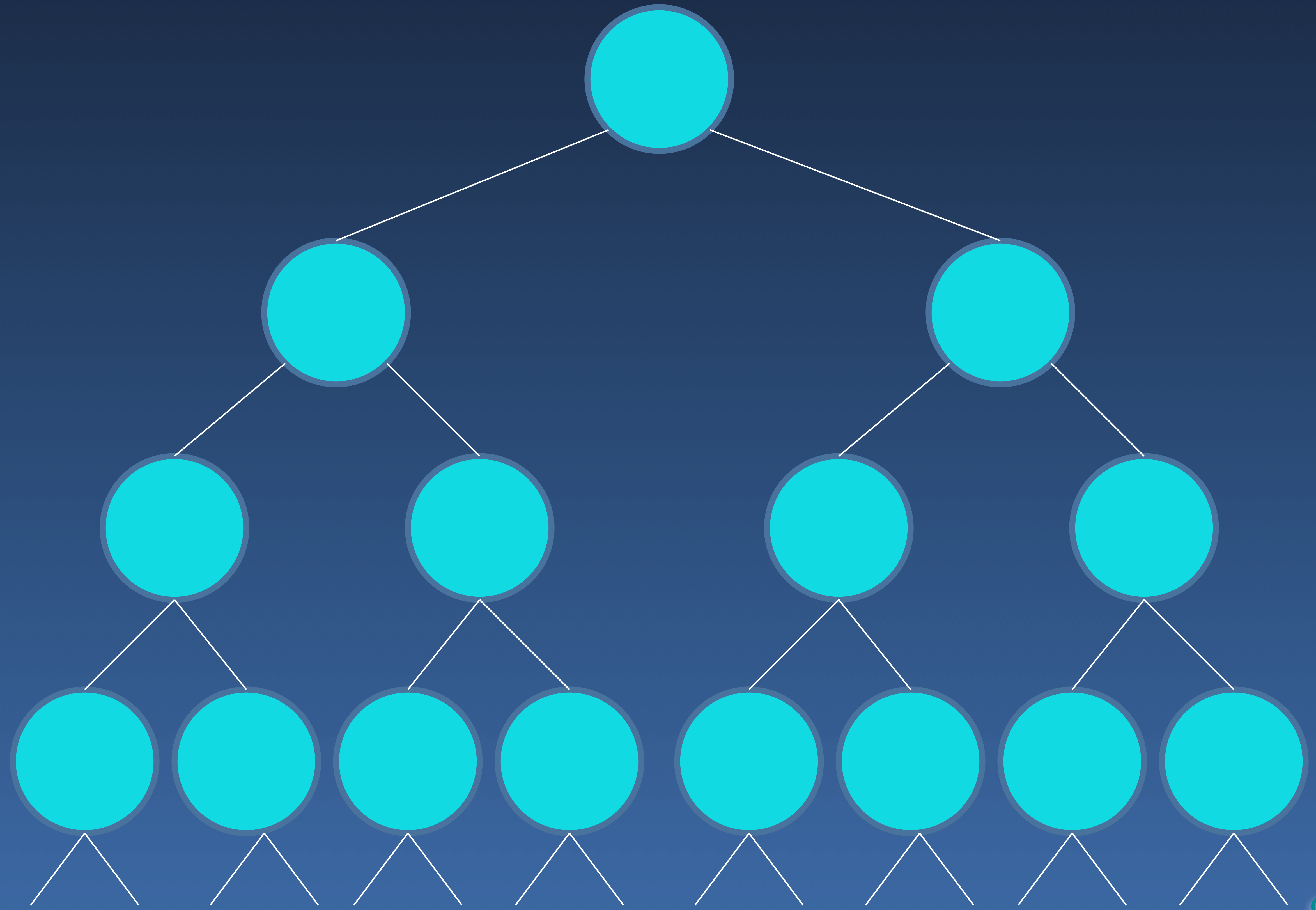
Prefix Sums

```
forall i < n
        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

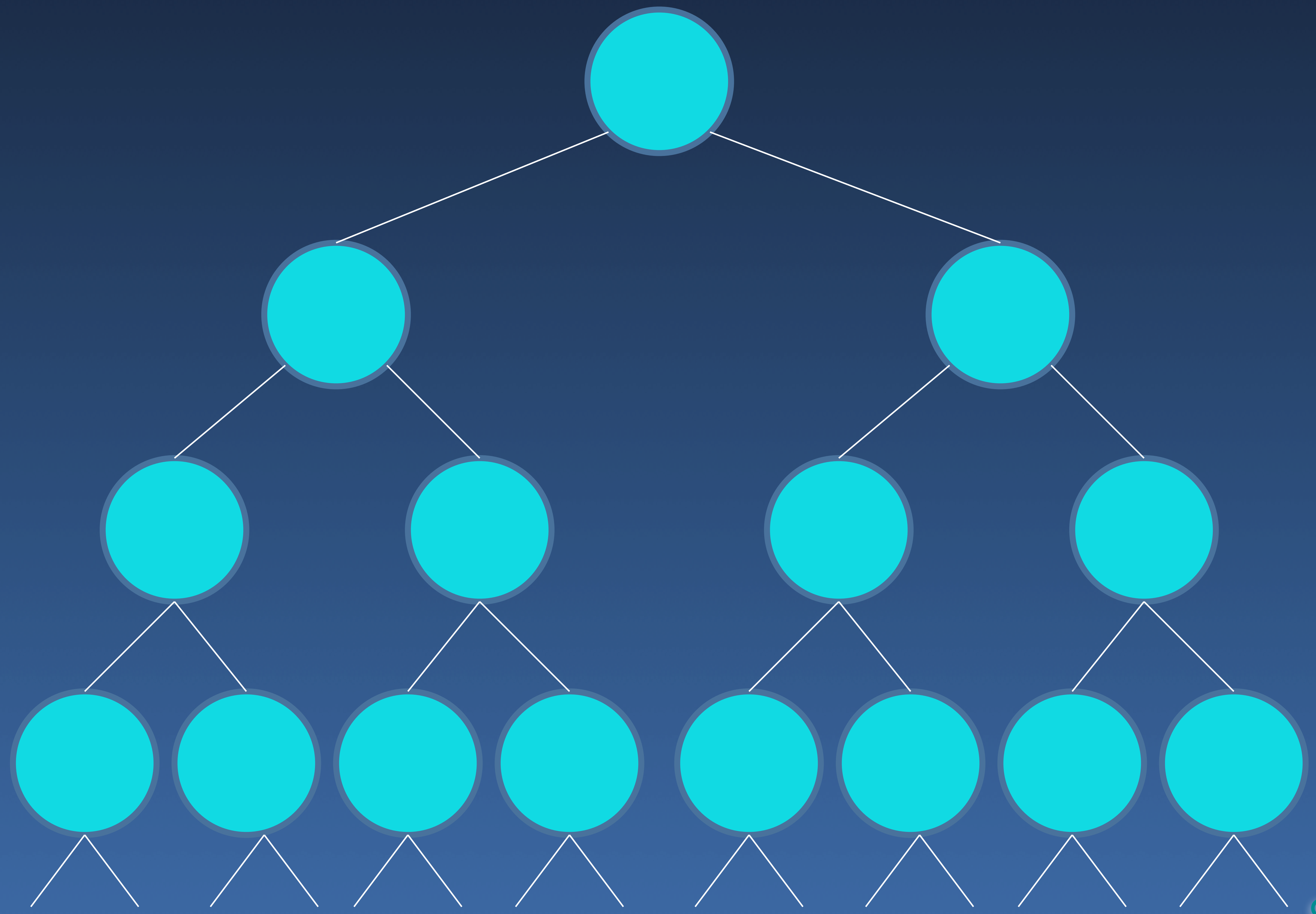
```
forall i < n
   if(filter(D[i]))
      R[i] = Map(D[i]);
```

Input: x

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

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

D



Subodh Kumar

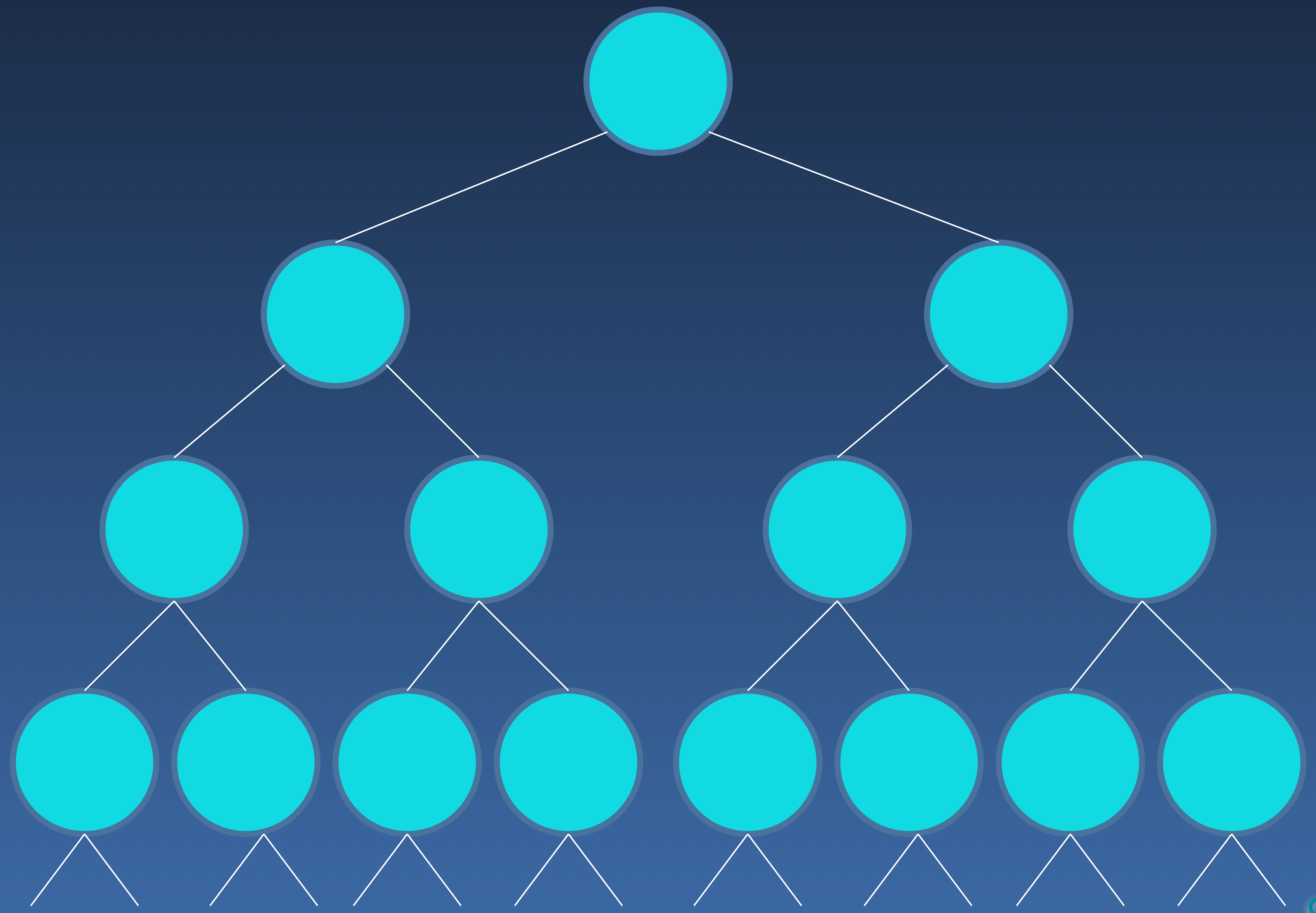
Prefix Sums

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

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

Input: x

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



Subodh Kumar

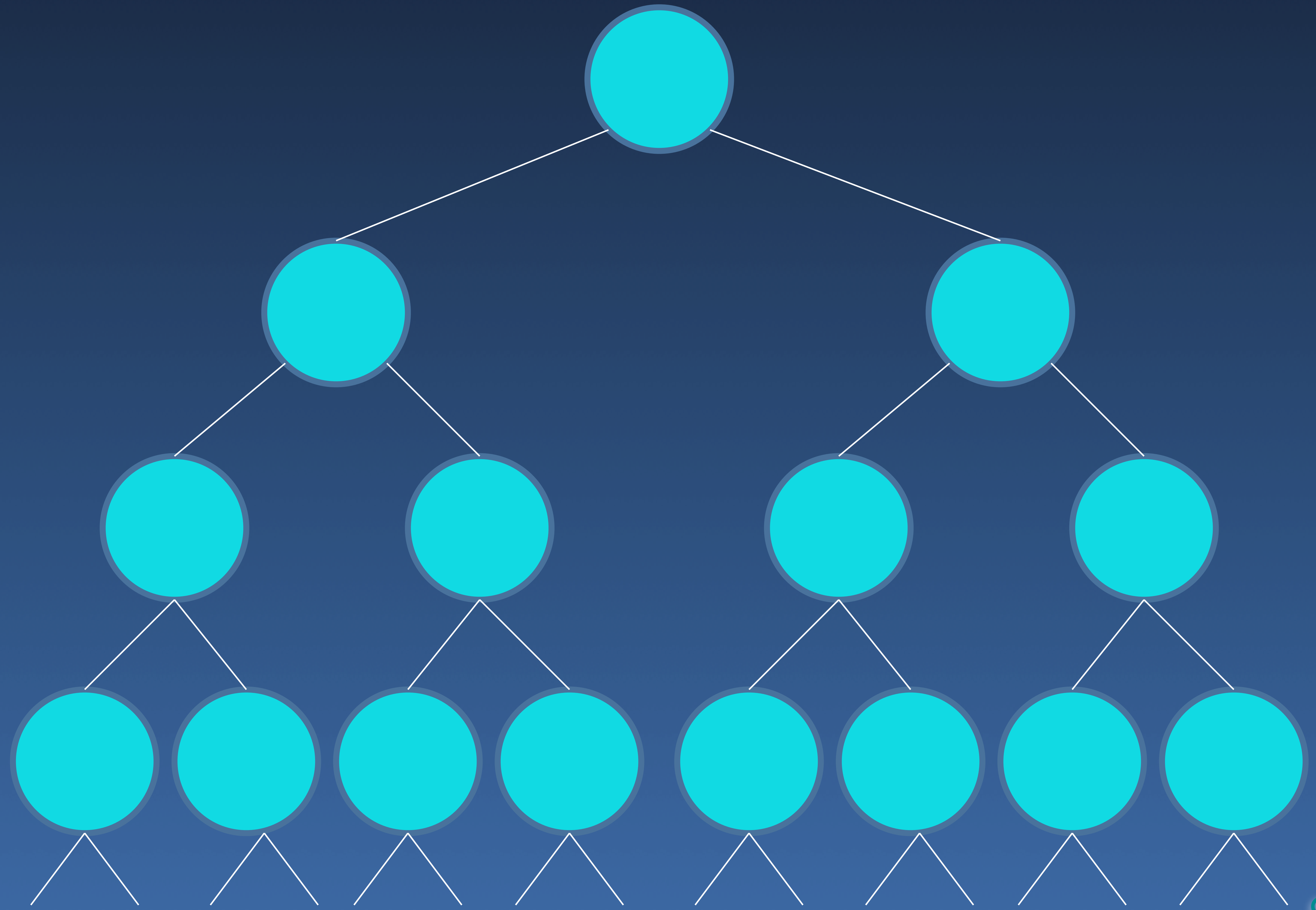
Prefix Sums

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

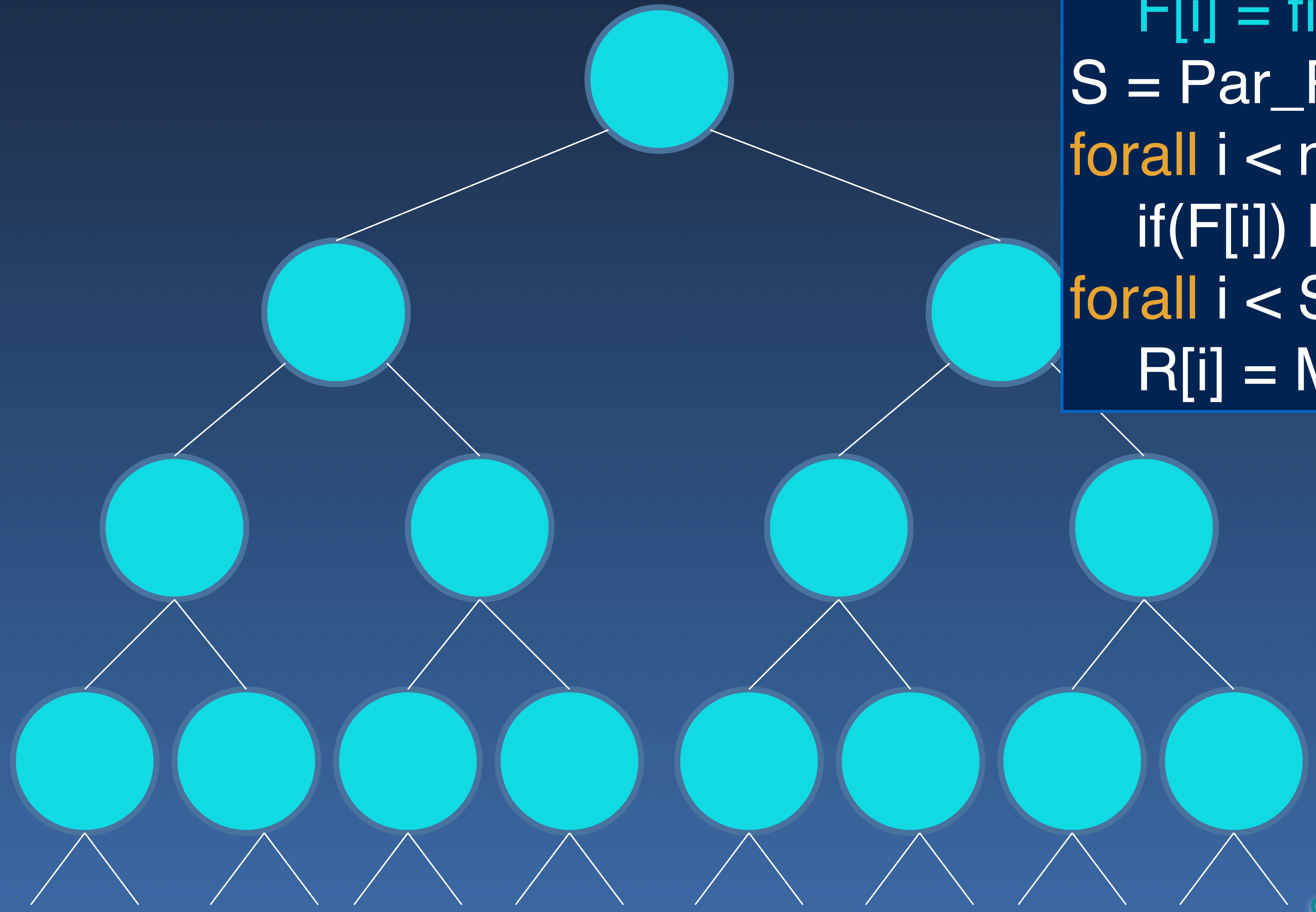
Prefix Sums

Input: x

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

P	2	3	4	6	10

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



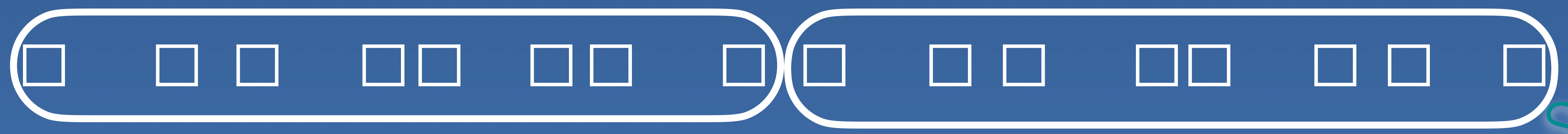
```
forall i < n
    if(filter(D[i]))
        R[i] = Map(D[i]);
```

```
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]]);
```

Recursive Prefix Sums

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

Psum two halves

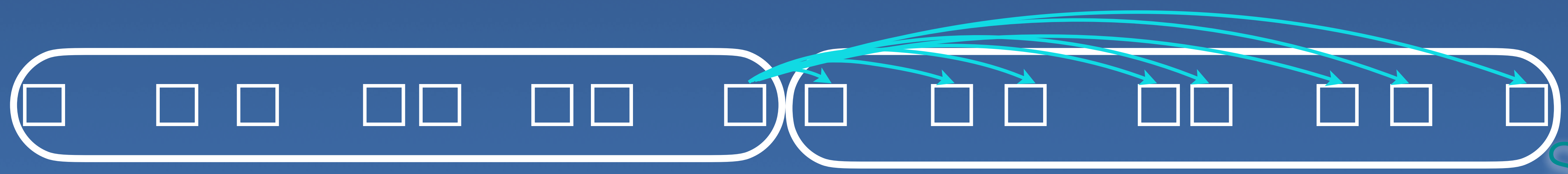


Subodh Kumar

Recursive Prefix Sums

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

Complete original Psum



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]

□ □ □ □□ □□ □ □ □ □ □□ □ □ □

Subodh Kumar

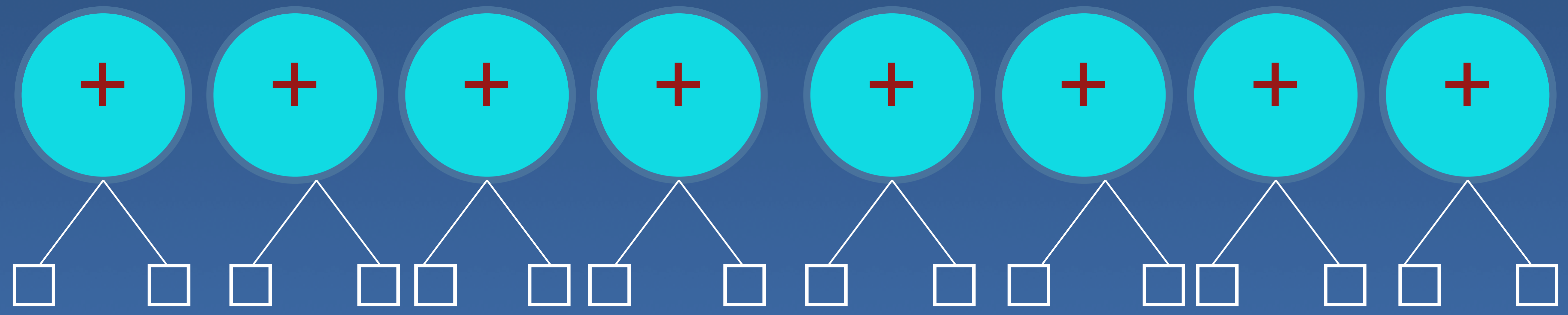
Recursive Prefix Sums

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

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

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

Pair-wise sum



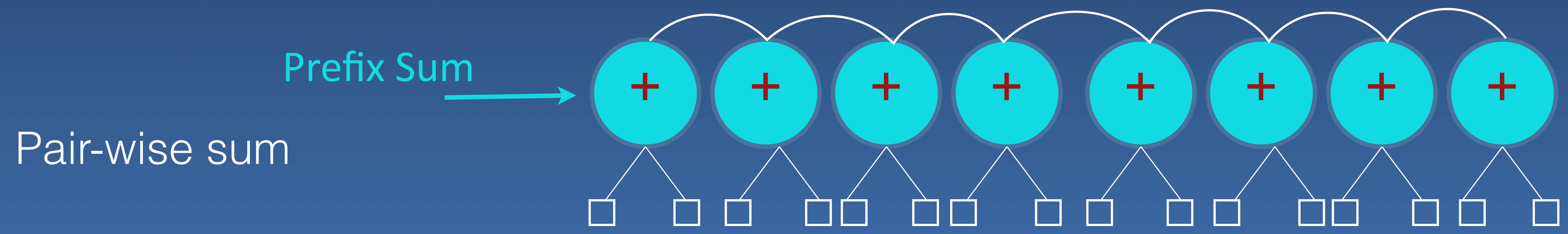
Subodh Kumar

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]

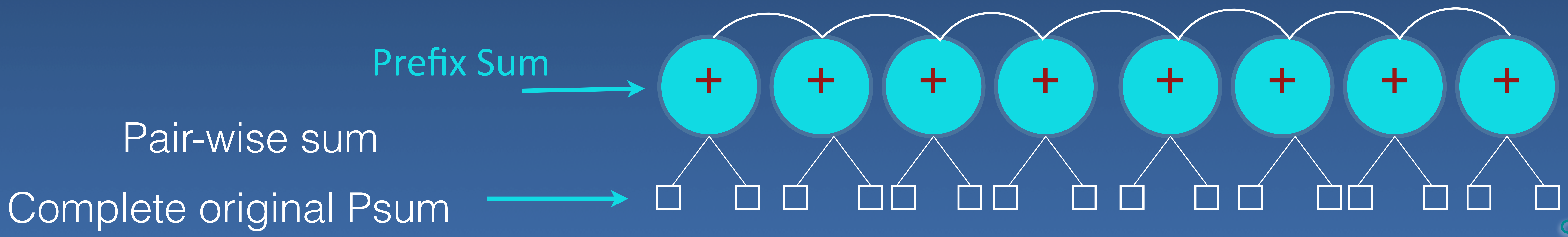


Recursive Prefix Sums

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

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

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



Subodh Kumar

Recursive Prefix Sums

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

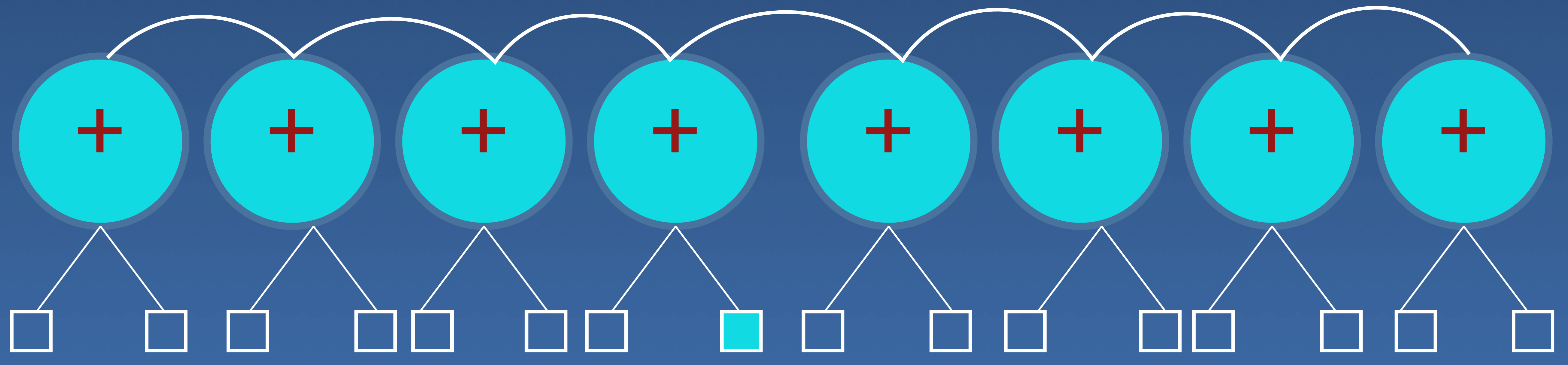
$W(n) = O(n \log n)$

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

Prefix Sum

Pair-wise sum

Complete original Psum



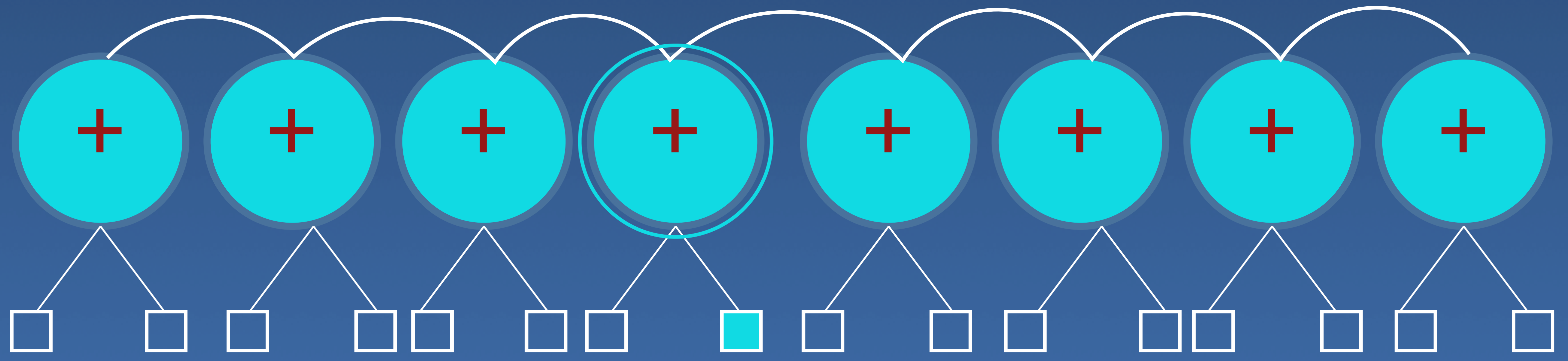
Subodh Kumar

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]



Subodh Kumar

Recursive Prefix Sums

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

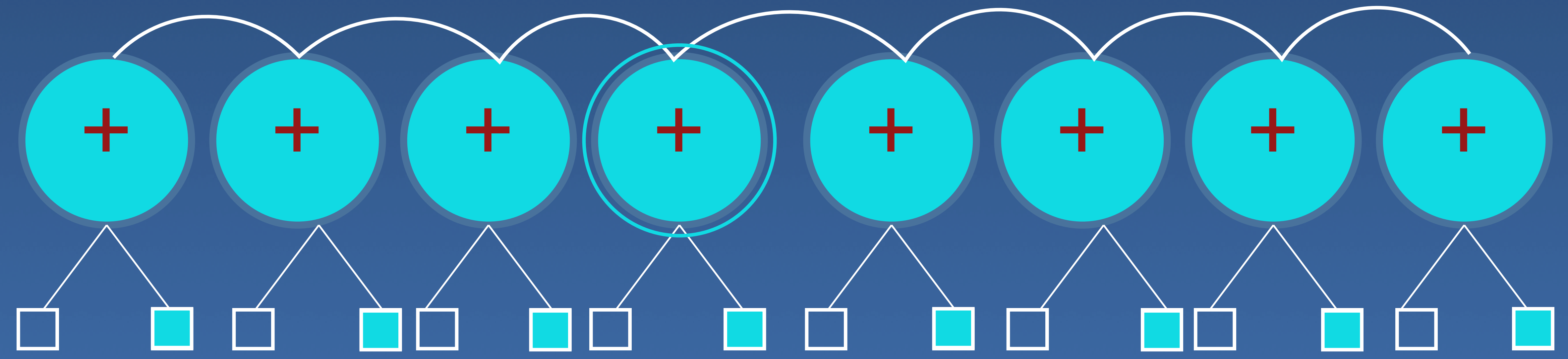
$W(n) = O(n \log n)$

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

Prefix Sum

Pair-wise sum

Complete original Psum



known by recursion

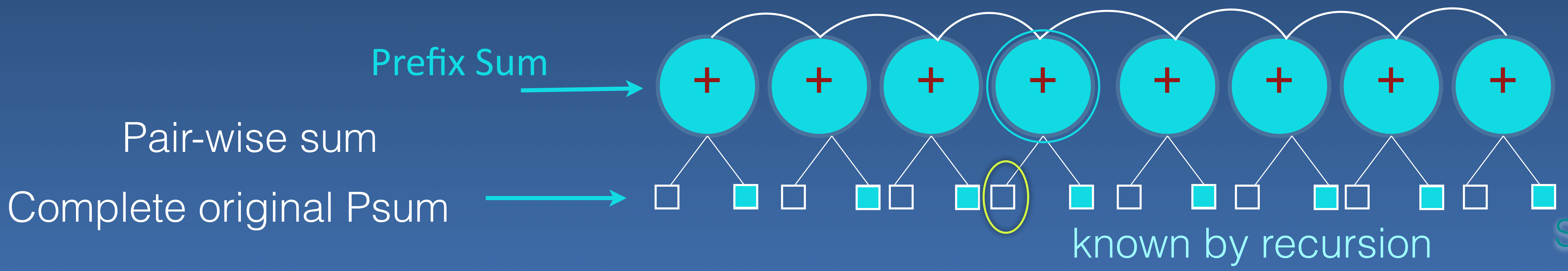
Subodh Kumar

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]



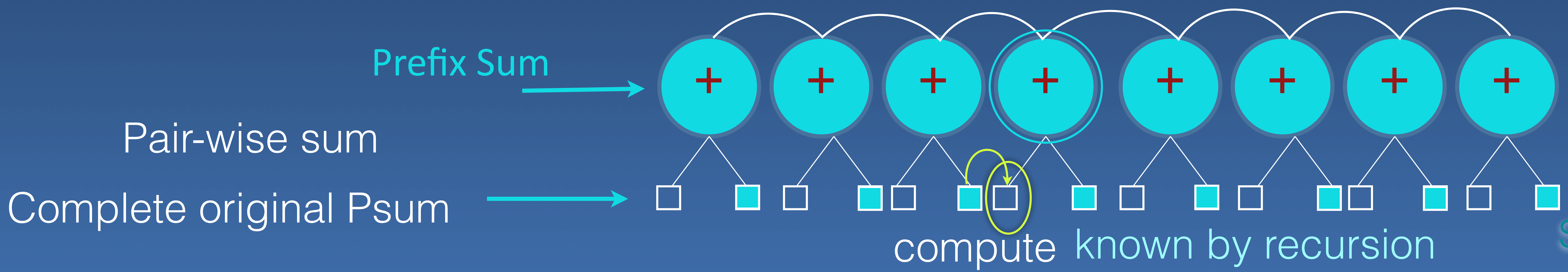
Subodh Kumar

Recursive Prefix Sums

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

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

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



Subodh Kumar

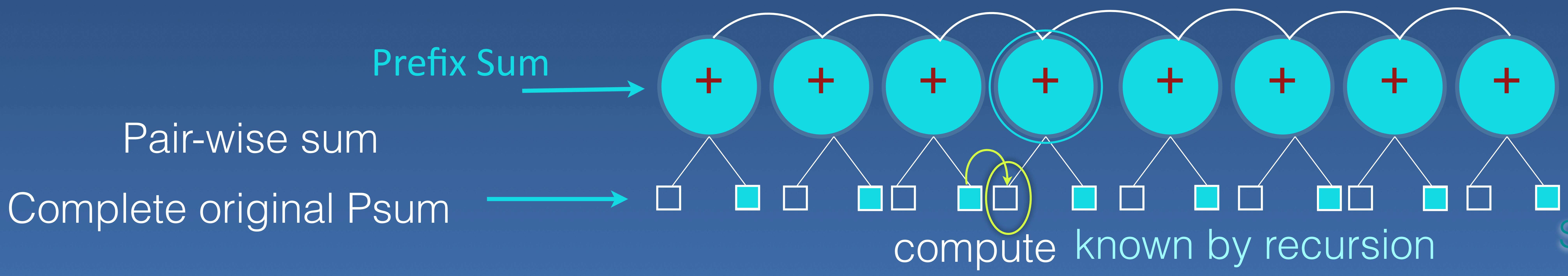
Recursive Prefix Sums

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

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

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

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



Subodh Kumar

Recursive Prefix Sums

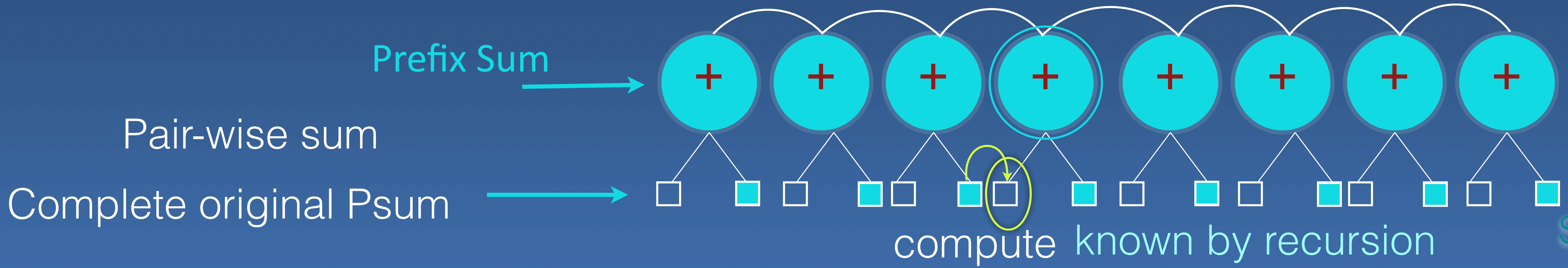
$T(n) = T(n/2) + O(1)$
$W(n) = 2W(n/2)+Kn/2$

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

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

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

$W(n) = O(n)$



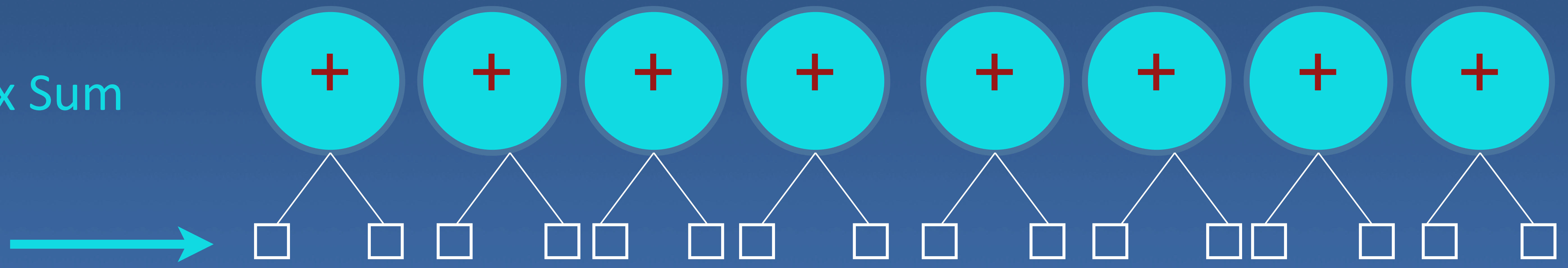
Subodh Kumar

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



Subodh Kumar

Prefix Sum Binary Tree

(Non recursive)

forall i = 0 to n

B[0][i] = A[i]

for h = 1 to log n

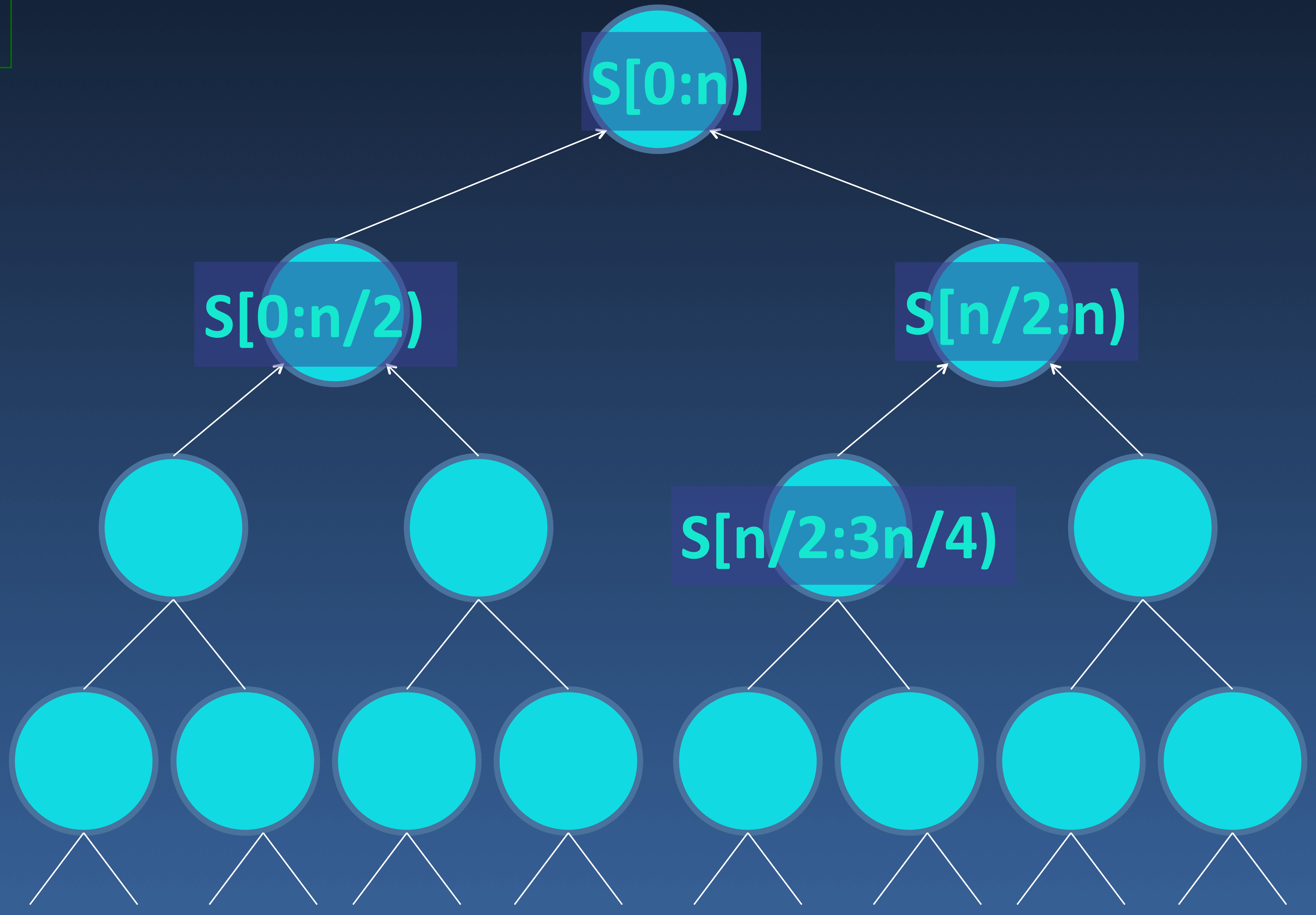
forall i in 0:n/2^h

B[h][i] = B[h-1][2i] **OP** B[h-1][2i+1]

P[0] = x[0]

For i = 1 to n-1

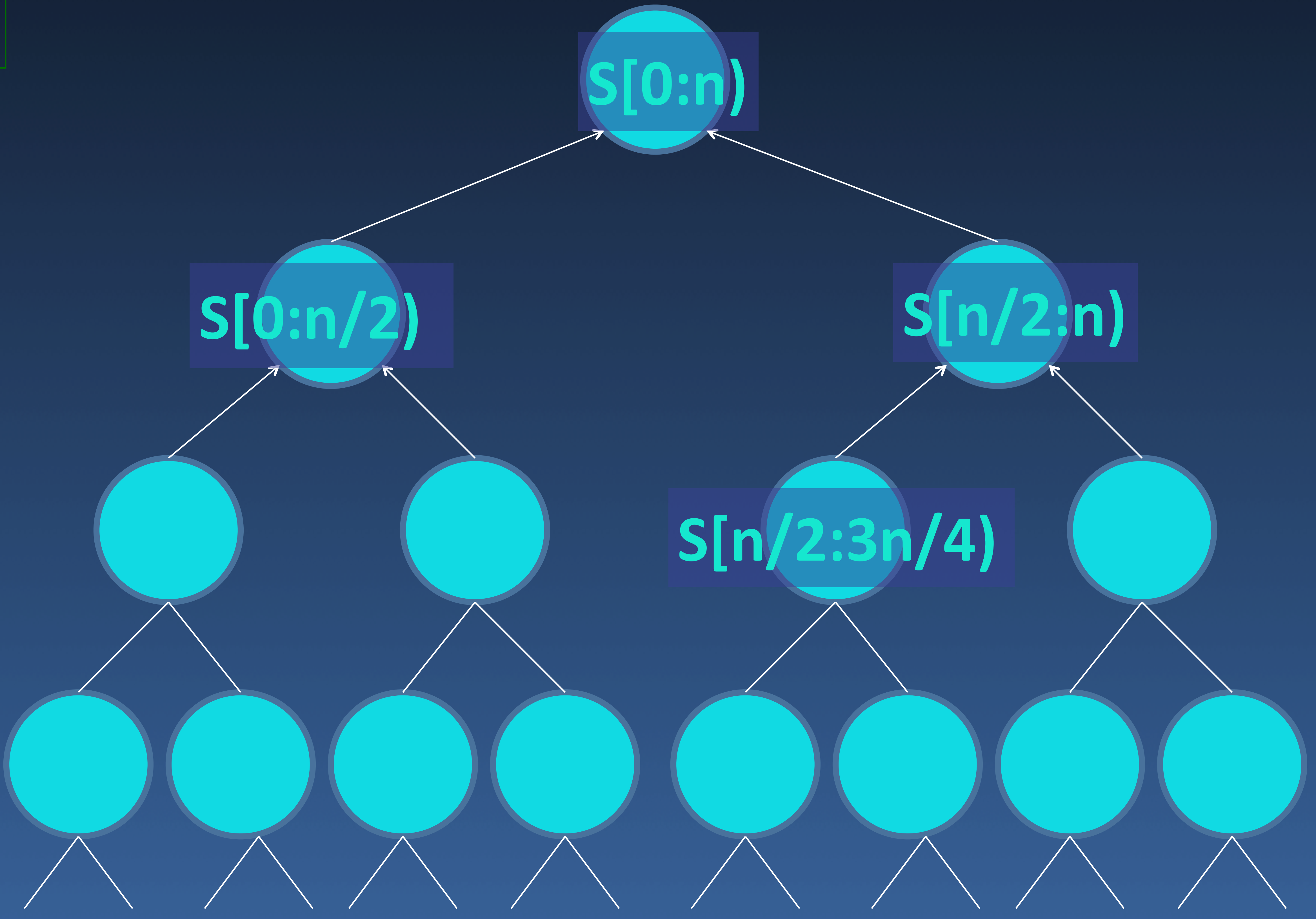
P[i] = P[i-1] + x[i]



Prefix Sum Binary Tree
(Non recursive)

```
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]
```

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

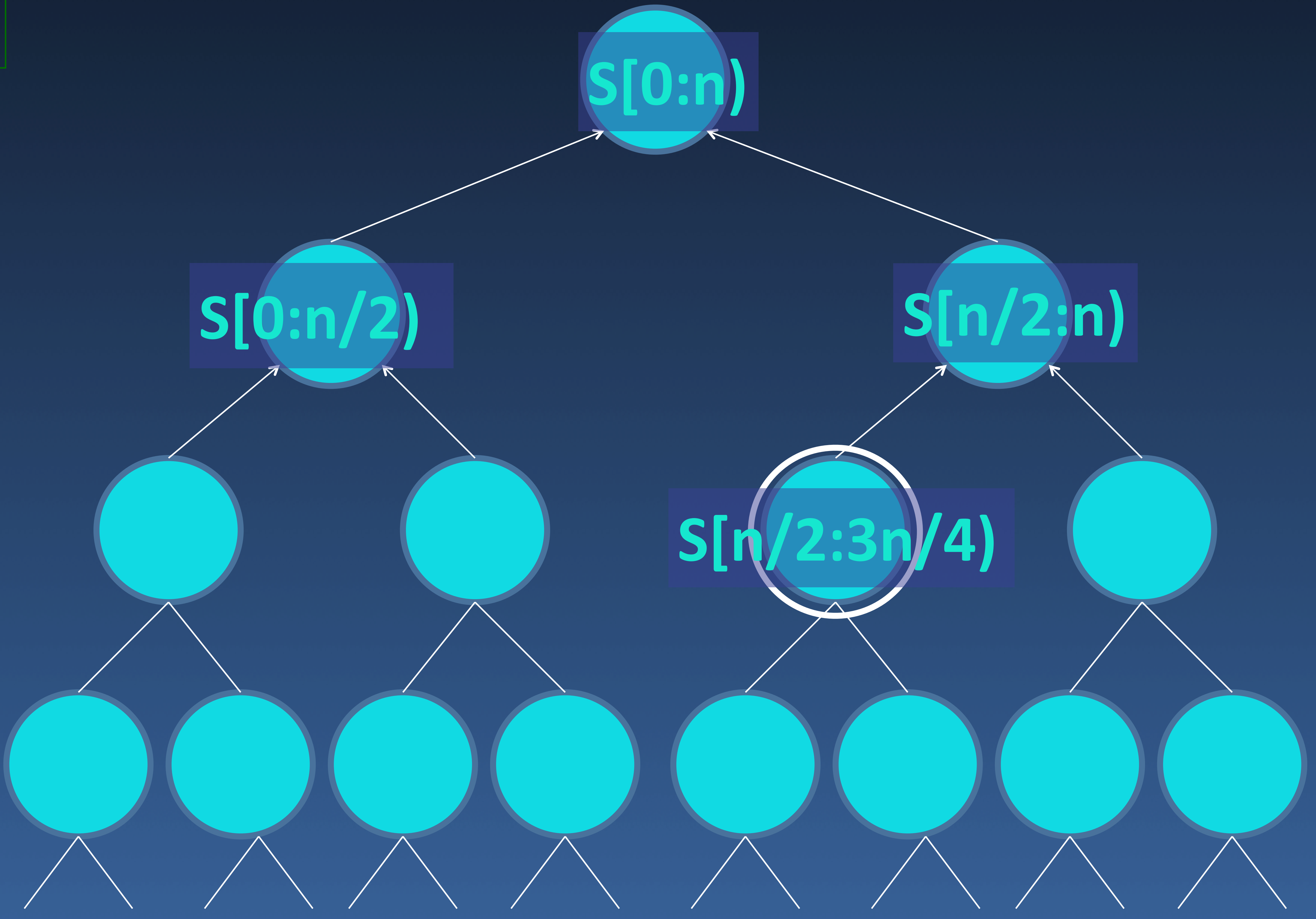


Subodh Kumar

Prefix Sum Binary Tree
(Non recursive)

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]

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



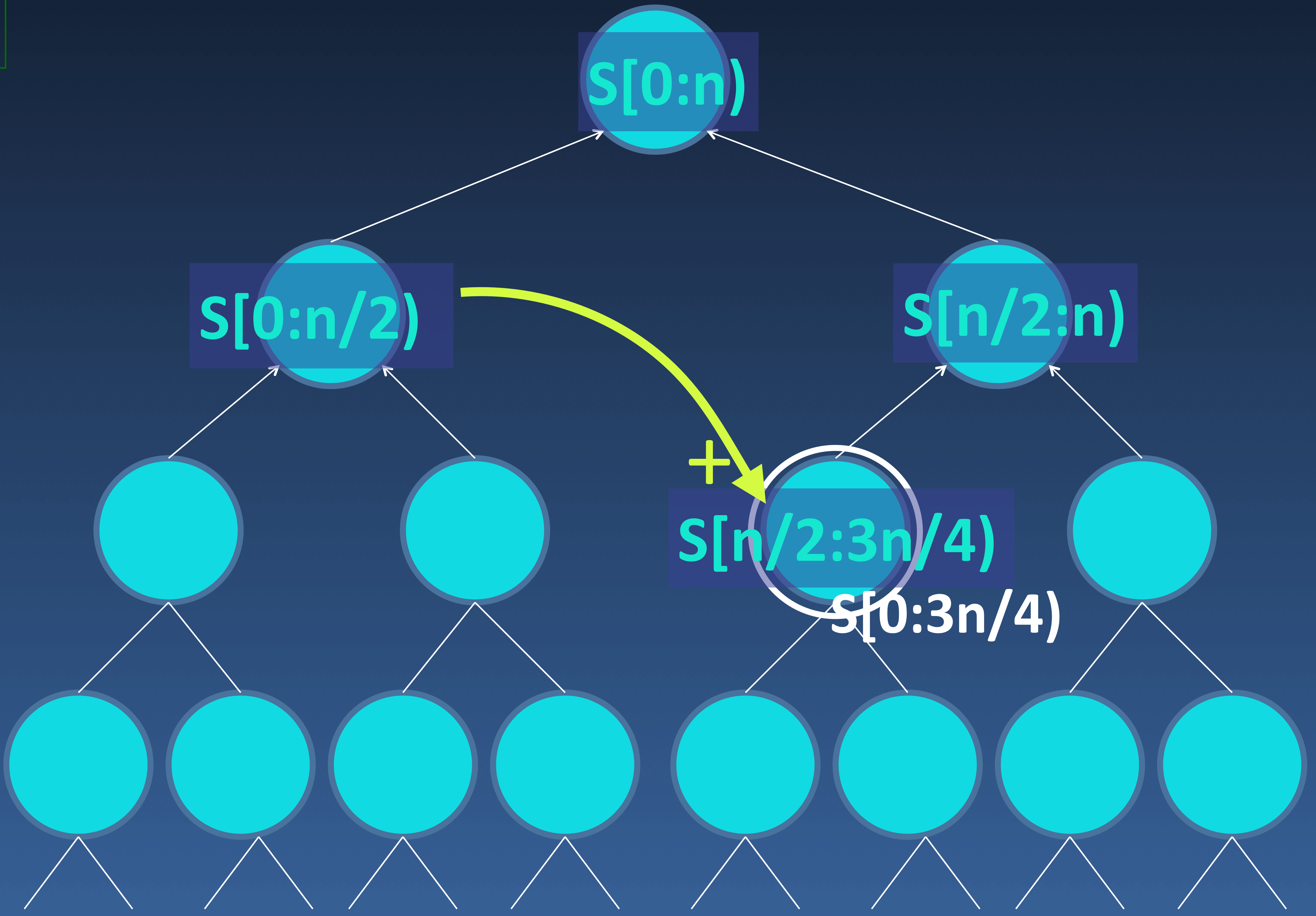
Subodh Kumar

Prefix Sum Binary Tree

(Non recursive)

```
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]
```

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

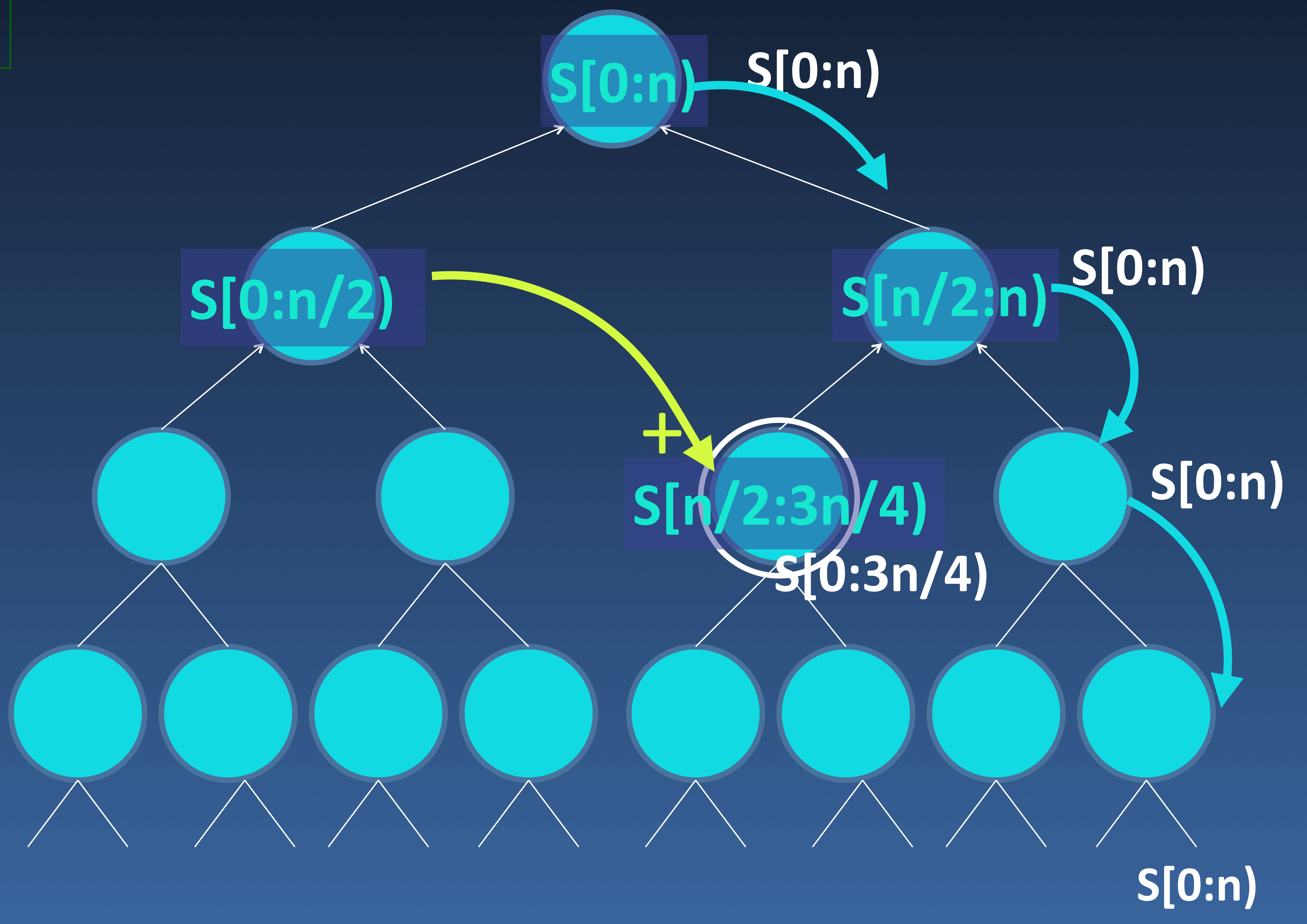


Subodh Kumar

Prefix Sum Binary Tree
(Non recursive)

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]

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



Subodh Kumar

Balanced Tree Approach

- Build binary tree on the input
- Hierarchically divide into groups
  - ➡ and groups of groups..
- Traverse tree upwards/downwards
- Useful to think of “tree” network topology
  - ➡ Only for algorithm design
  - ➡ Later map sub-trees to processors

Merge Sorted Arrays A,B

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

Subodh Kumar

Merge Sorted Arrays A,B

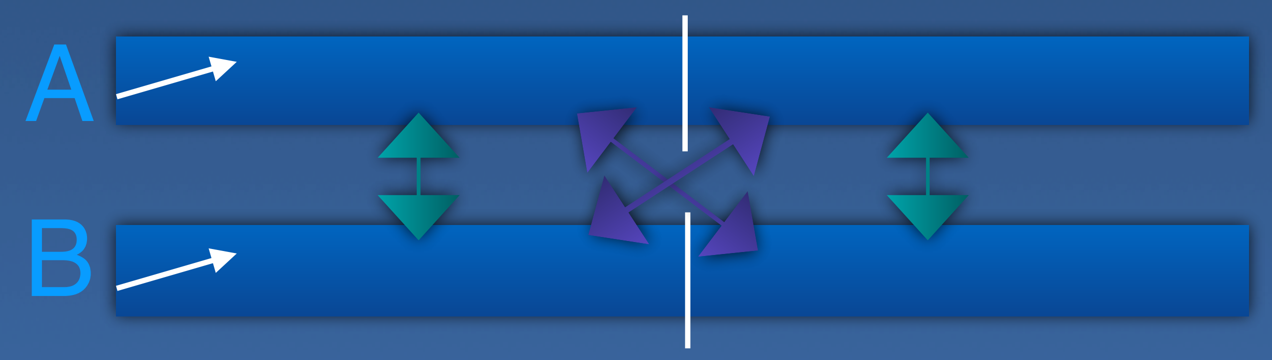
```
if(A[i] <= B[j])
   C[k++] = A[i++]
else
   C[k++] = B[j++]
```

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

Subodh Kumar

Merge Sorted Arrays A,B

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

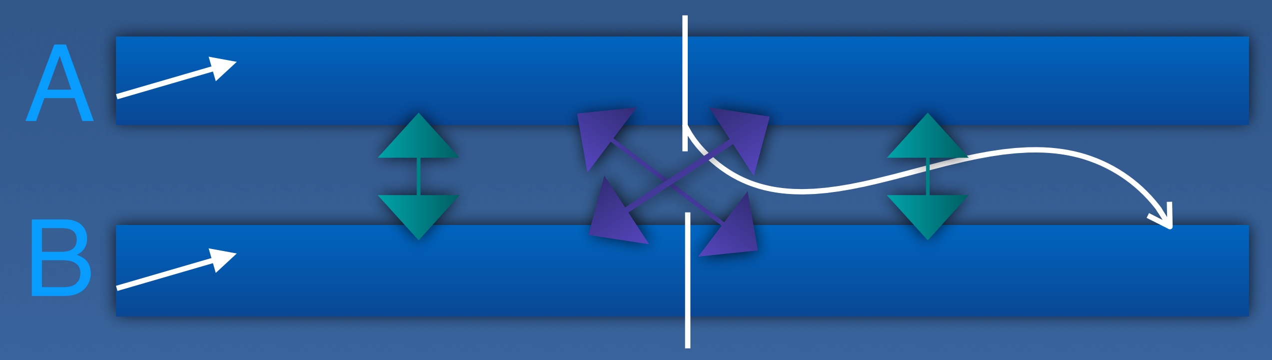


$T(2n) = T(n) + ?$

Subodh Kumar

Merge Sorted Arrays A,B

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

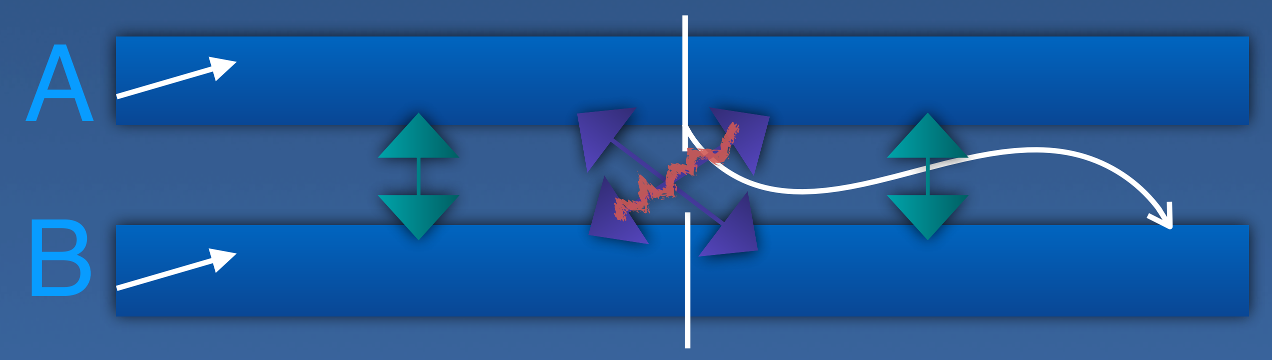


$T(2n) = T(n) + ?$

Subodh Kumar

Merge Sorted Arrays A,B

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

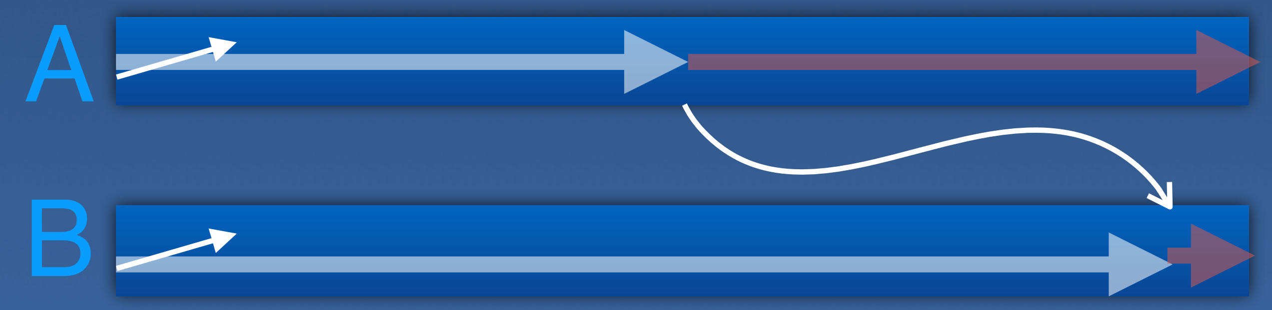


$T(2n) = T(n) + ?$

Subodh Kumar

Merge Sorted Arrays A,B

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



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

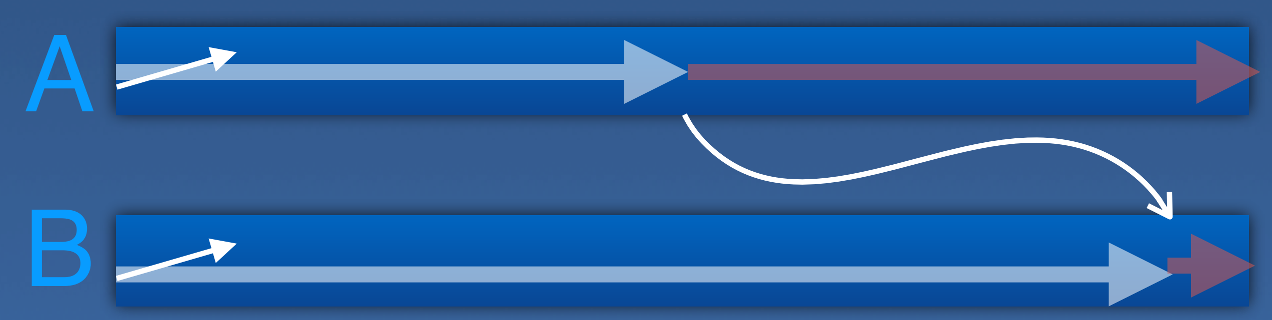
$T(2n) = T(3n/2) + \log n$

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

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



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

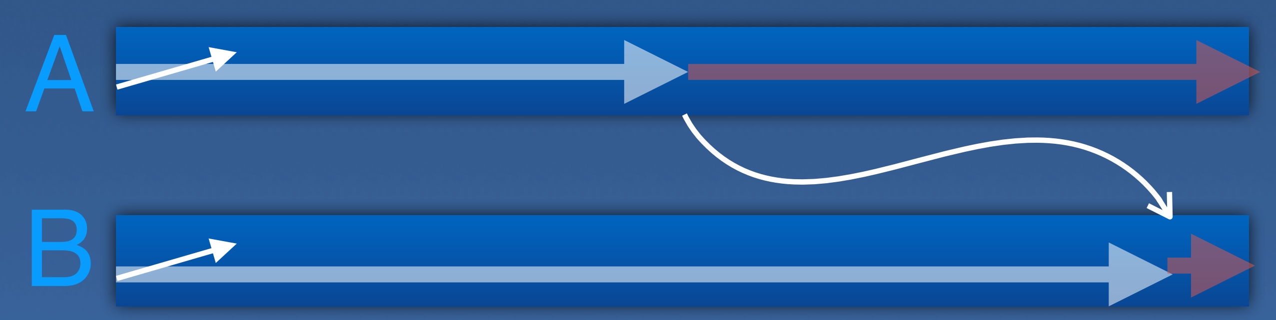
$T(2n) = T(3n/2) + \log n$

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 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++]
```



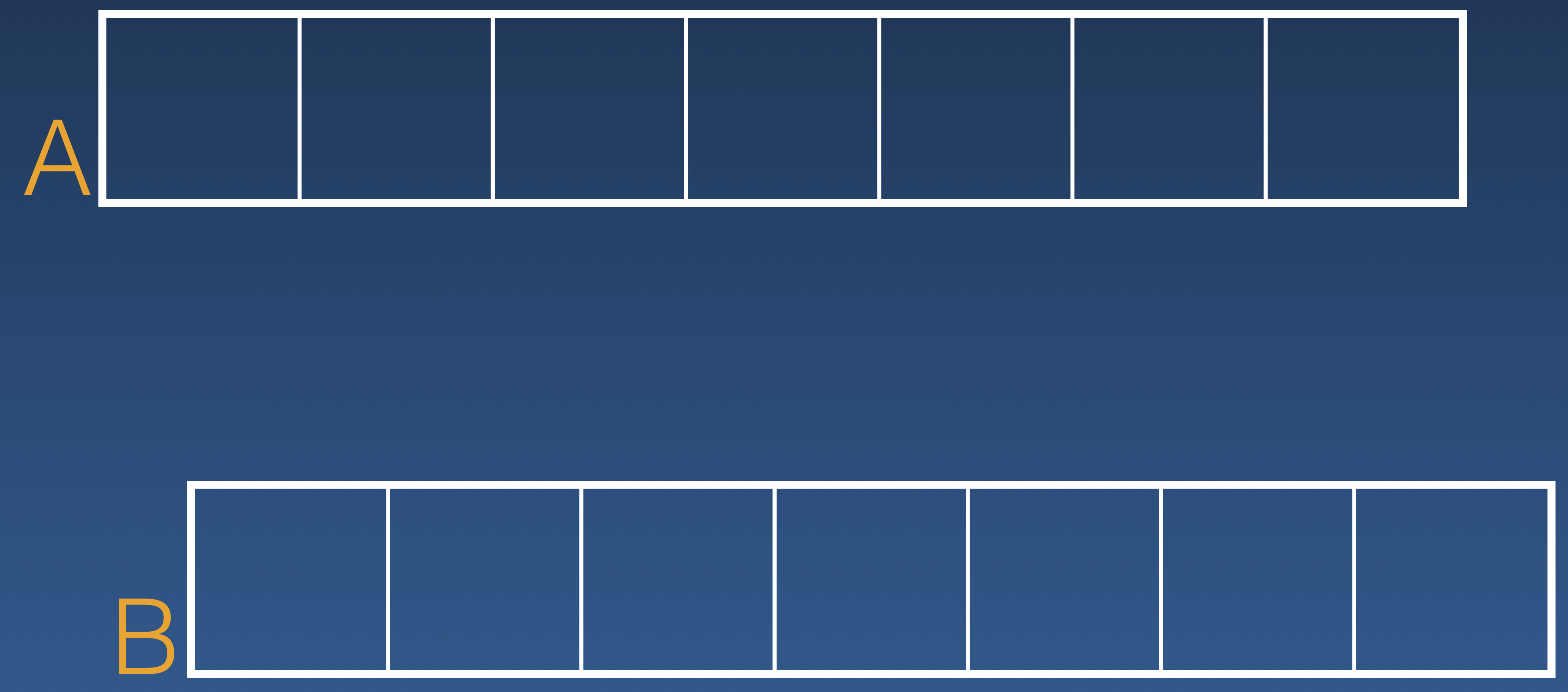
$T(2n) = T(n) + ?$

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

Towards Optimal Merge(A,B)

- Partition A and B into log n sized blocks

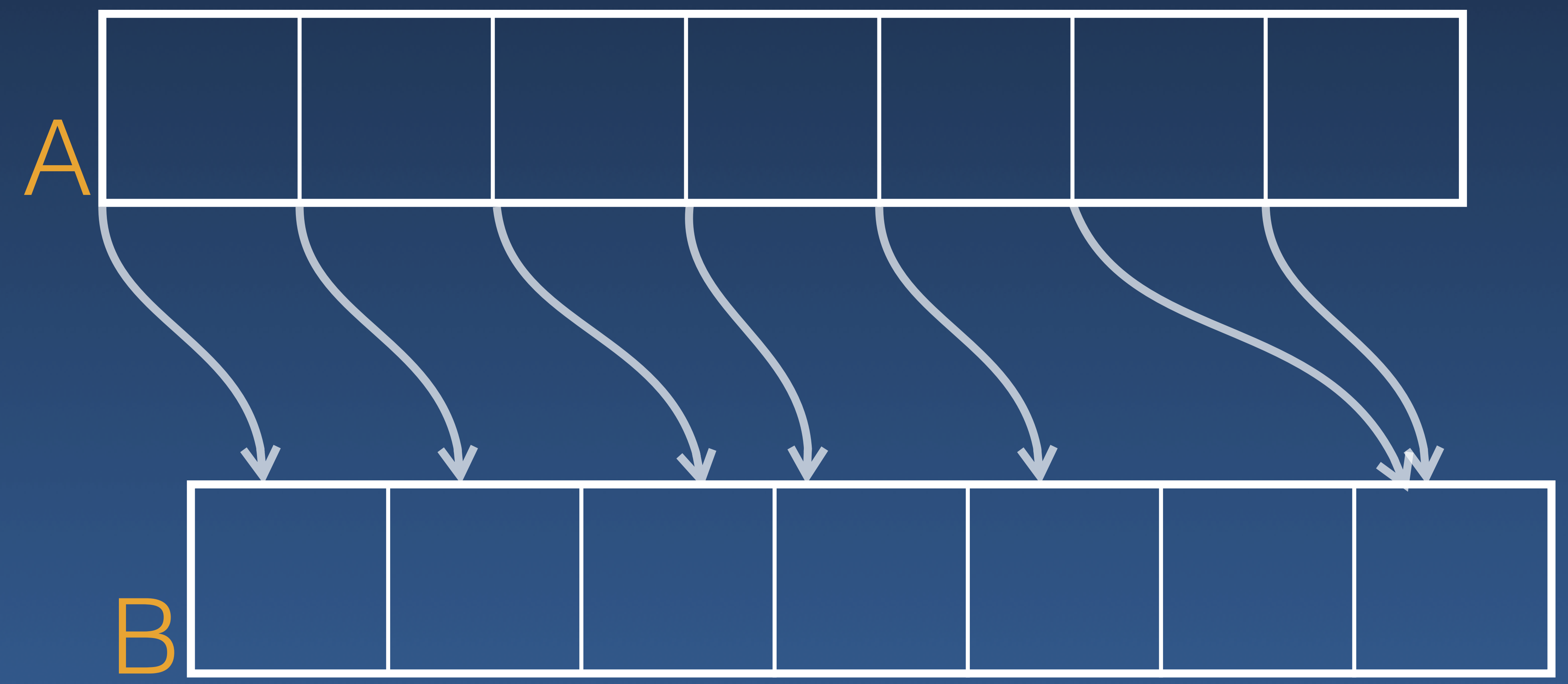
Data Partitioning Technique



Towards Optimal Merge(A,B)

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

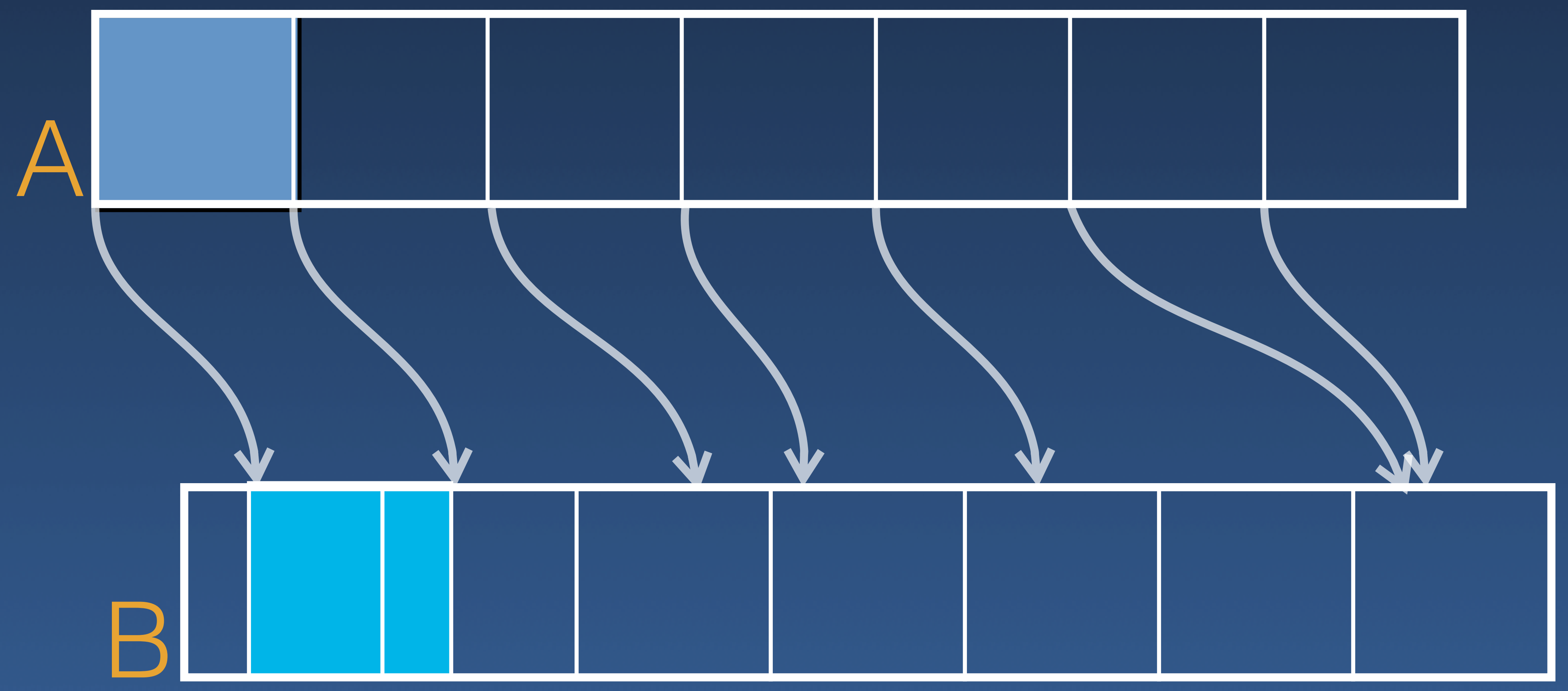
Data Partitioning Technique



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

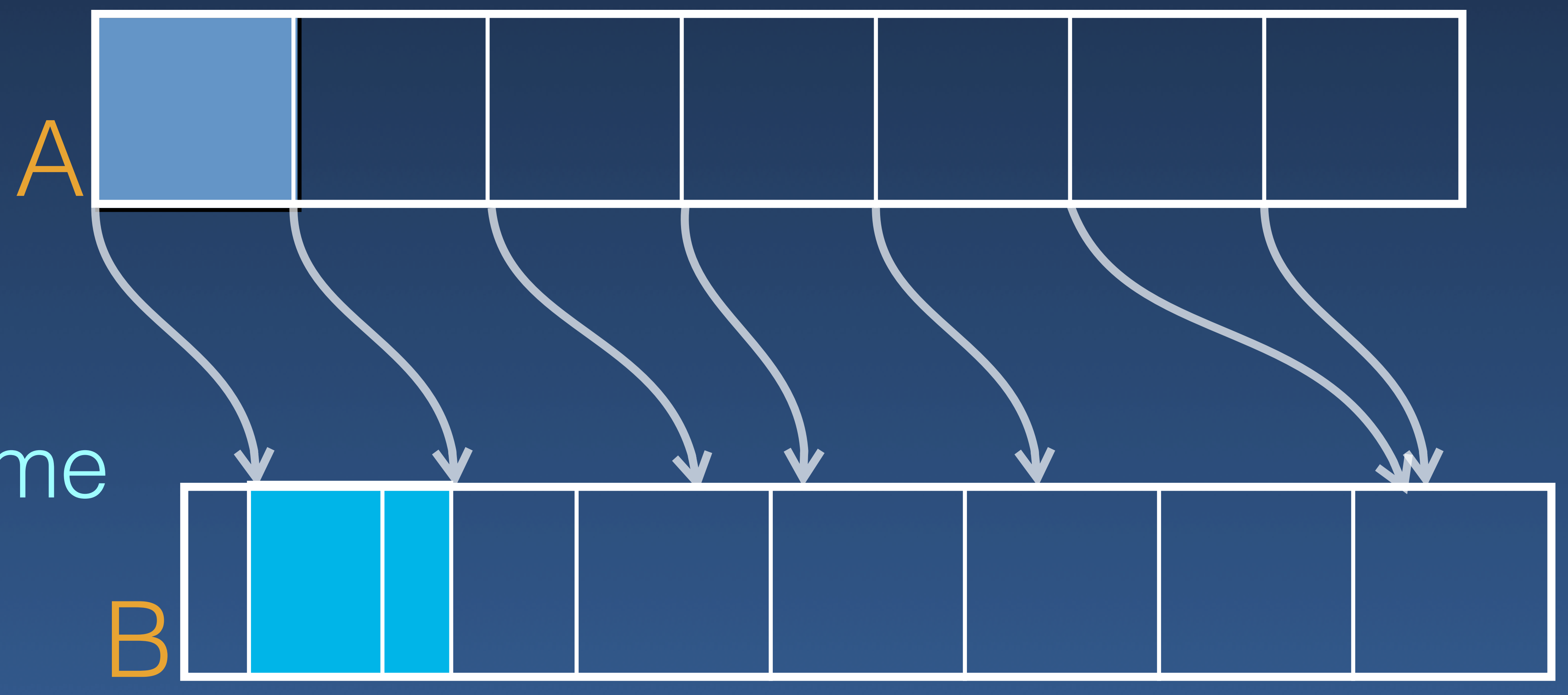


Subodh Kumar

Towards Optimal Merge(A,B)

- Partition A and B into log n sized blocks
- Select from A, elements i * log n, $i \in 0{:}n/\log n$
- 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

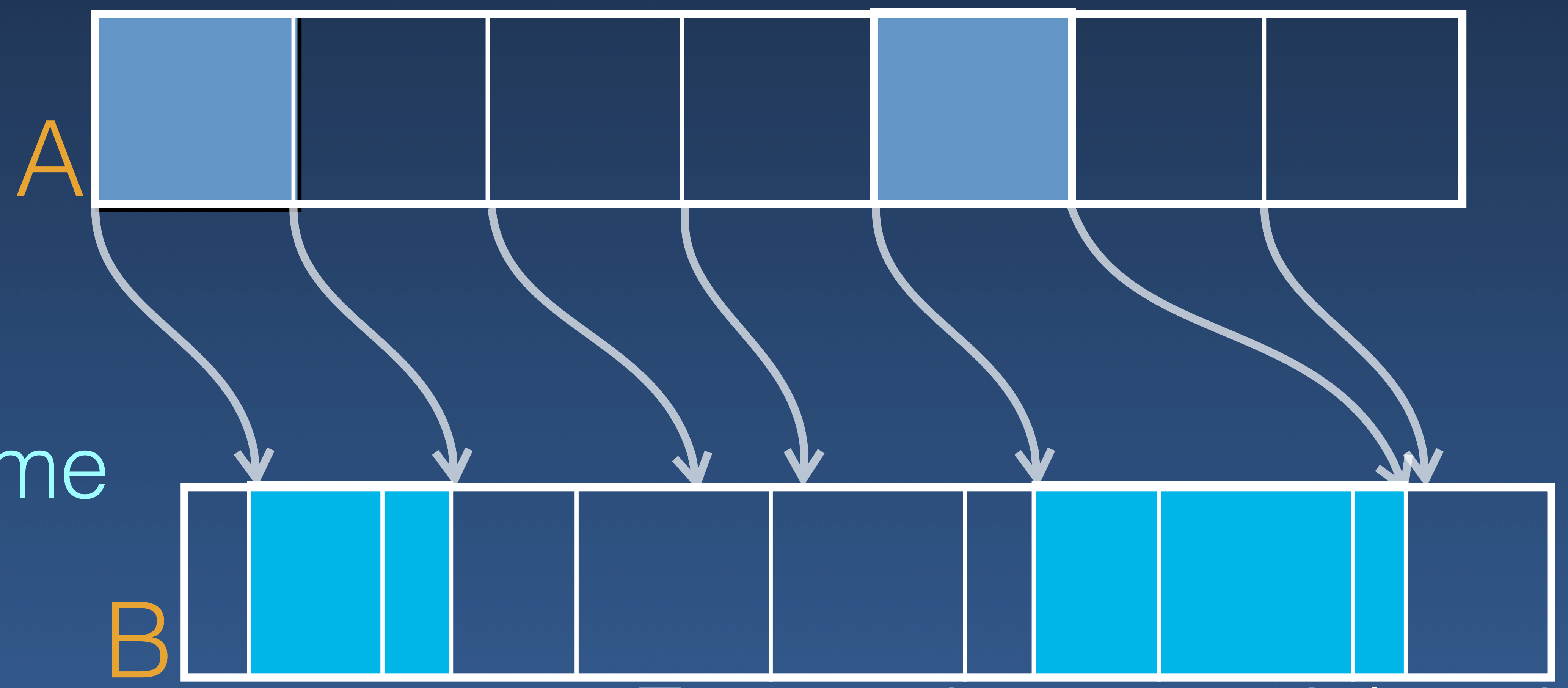
Data Partitioning Technique



Towards Optimal Merge(A,B)

- Partition A and B into log n sized blocks
- Select from A, elements i * log n, $i \in 0{:}n/\log n$
- 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 Recursively subdivide A_i into sub-sub-sequences

Data Partitioning Technique



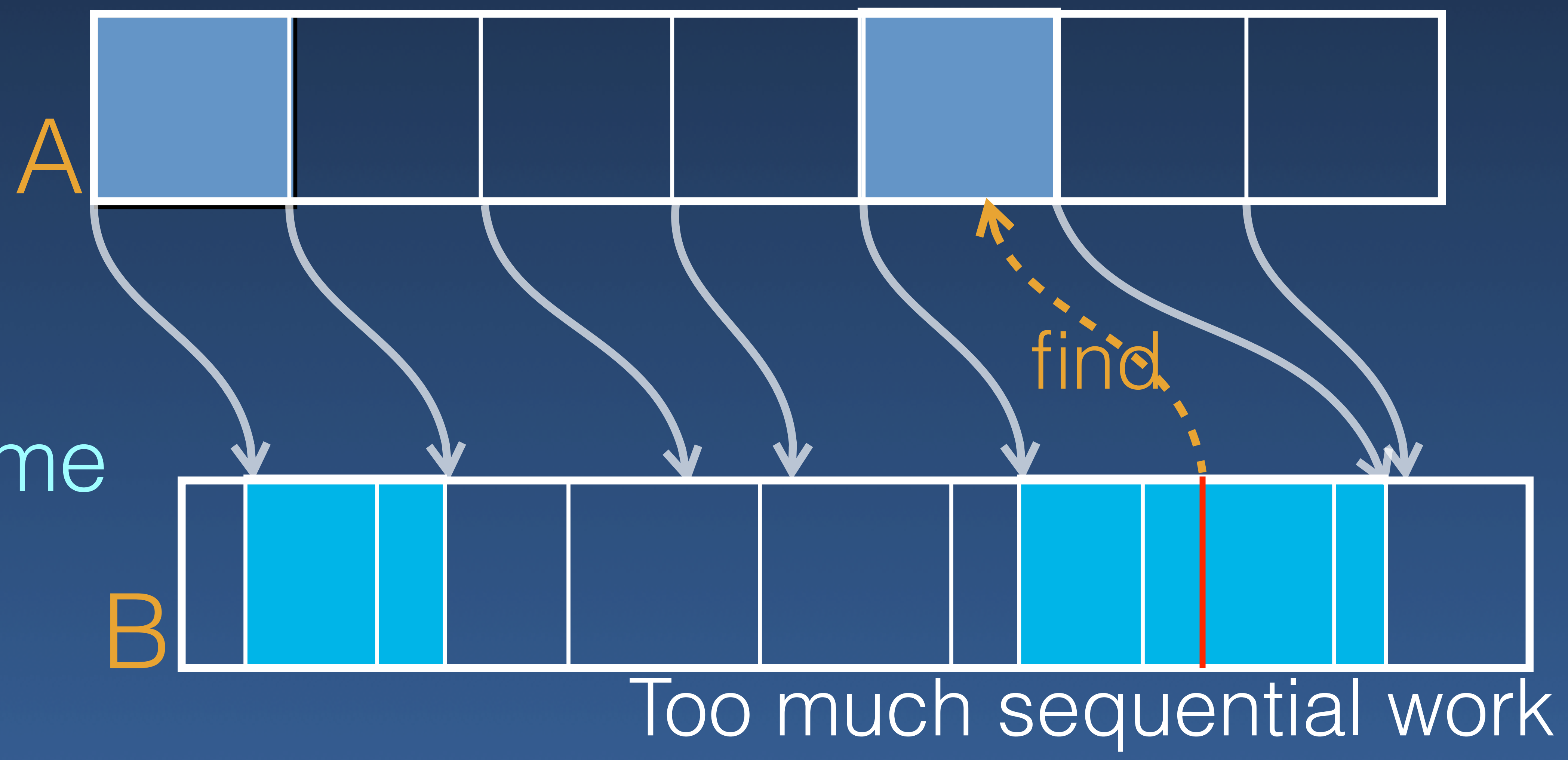
Too much sequential work

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 $|B_i| \le \log(n)$, Sequential merge in O(log n) time
  - ➡ Otherwise, partition B_i into log n blocks
    - ▸ And Recursively subdivide A_i into sub-sub-sequences

Data Partitioning Technique

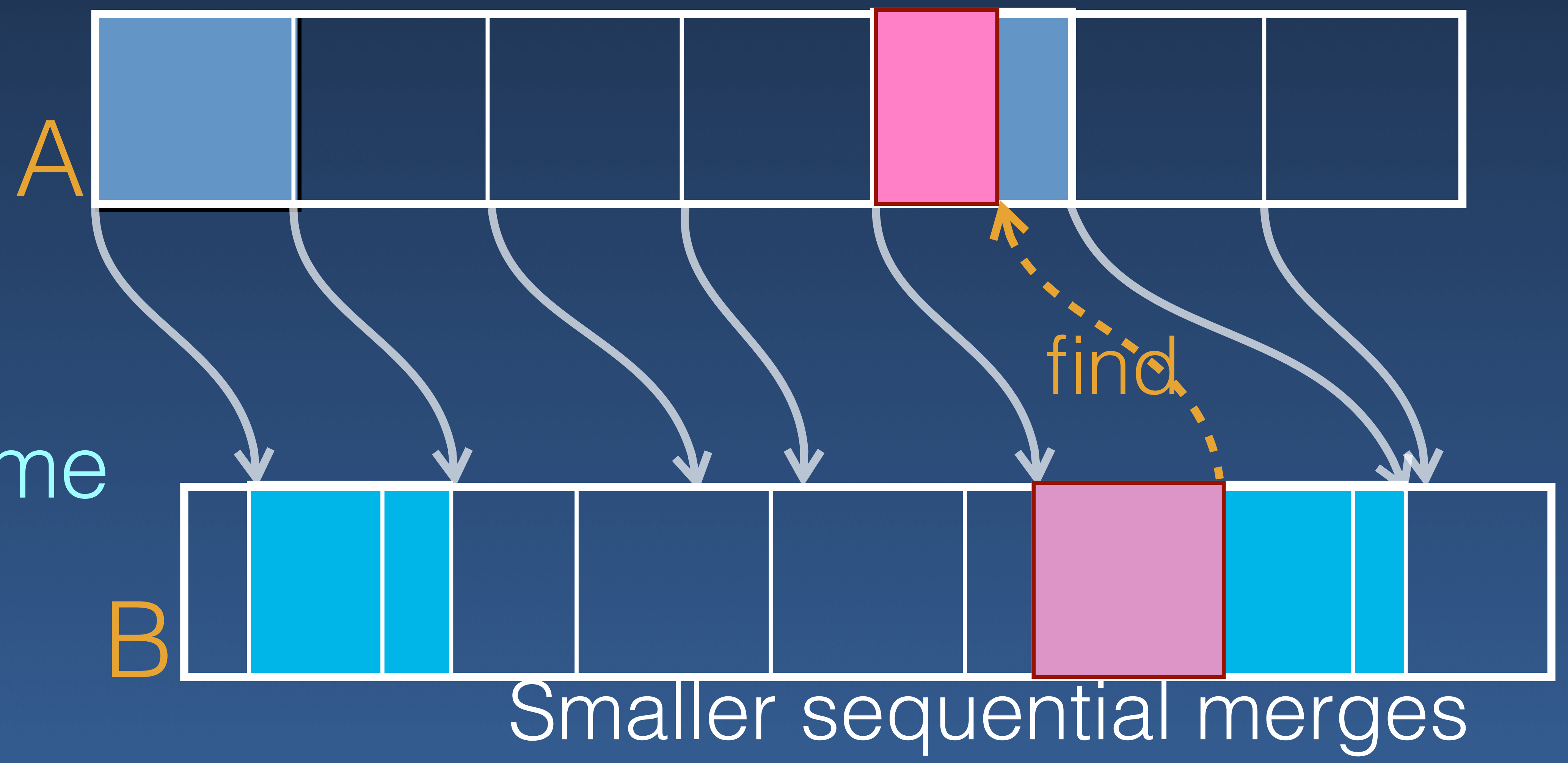


Subodh Kumar

Towards Optimal Merge(A,B)

- Partition A and B into log n sized blocks
- Select from A, elements i * log n, $i \in 0{:}n/\log n$
- 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 Recursively subdivide A_i into sub-sub-sequences

Data Partitioning Technique



Subodh Kumar

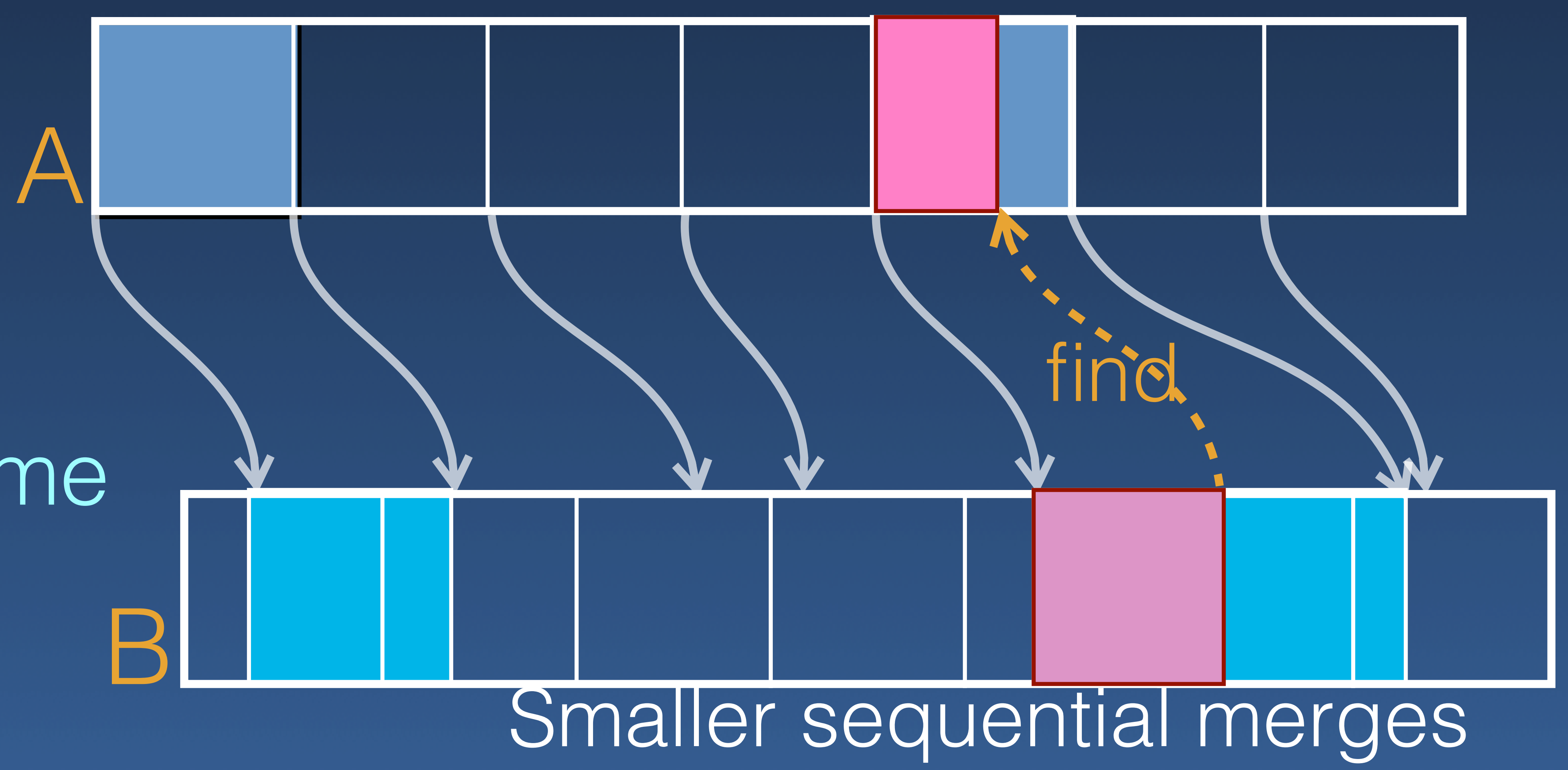
Towards Optimal Merge(A,B)

- 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 $|B_i| \le \log(n)$, Sequential merge in O(log n) time
  - ➡ Otherwise, partition B_i into log n blocks
    - And Recursively subdivide A_i into sub-sub-sequences

Total time is O(log n)

Total work is O(n)

Data Partitioning Technique



Subodh Kumar

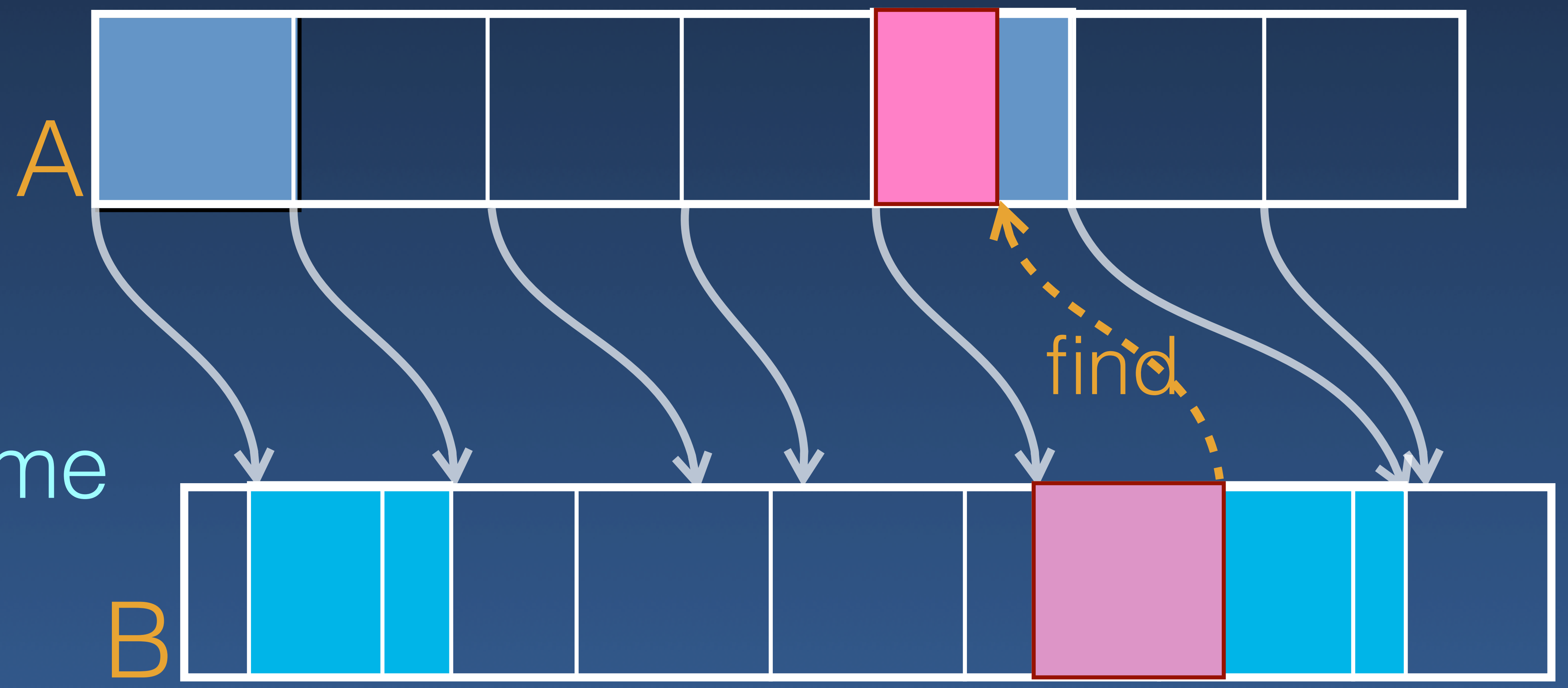
Towards Optimal Merge(A,B)

- Partition A and B into log n sized blocks
- Select from A, elements i * log n, $i \in 0{:}n/\log n$
- 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 Recursively subdivide A_i into sub-sub-sequences

Total time is O(log n)

Total work is O(n)

Data Partitioning Technique



Can we do better?

Subodh Kumar

Towards Optimal Merge(A,B)

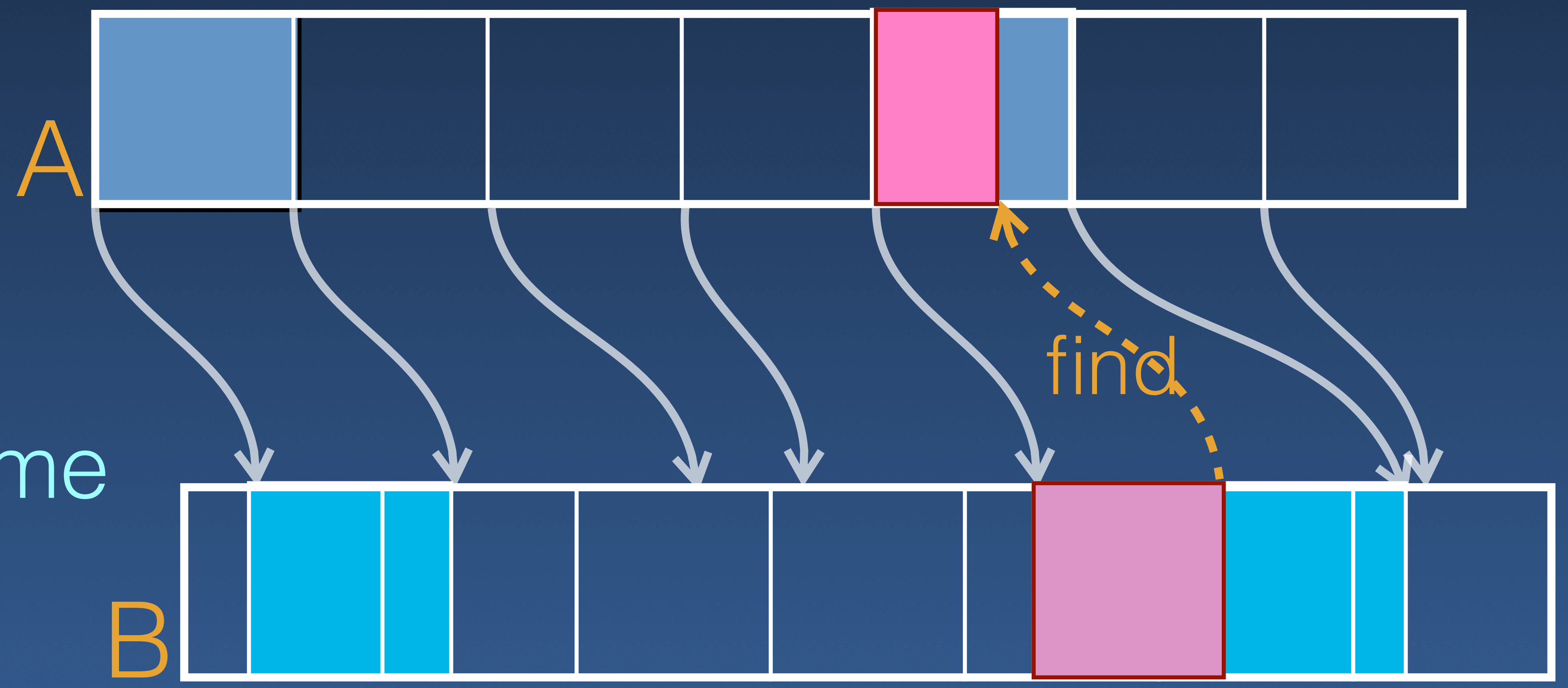
- Partition A and B into log n sized blocks
- Select from A, elements $i * \log n$, $i \in 0{:}n/\log n$
- Rank each selected element of A in B
  - ➡ Binary search
- Merge pairs of sub-sequences
  - ➡ If $|B_i| \le \log(n)$, Sequential merge in O(log n) time
  - ➡ Otherwise, partition B_i into log n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