Recap: Prefix Sums

- Given **A**: set of *n* integers
- Find **B**: *prefix sums*

$$B[i] = \sum_{k=1}^{i} A[k]$$

A:	3	1	1	7	2	5	9	2	4	3	3

B:	3	4	5	12	14	19	28	30	34	37	40

Recap: Parallel Prefix Sums

- Recursive algorithm
  - Recursively computes sums
  - Use partial sums to get prefix sums
- $T(n) = O(\log n)$
- $W(n) = O(n)$
- Hard to get intuition
- Iterative algorithm easier to grasp?

Iterative prefix sum

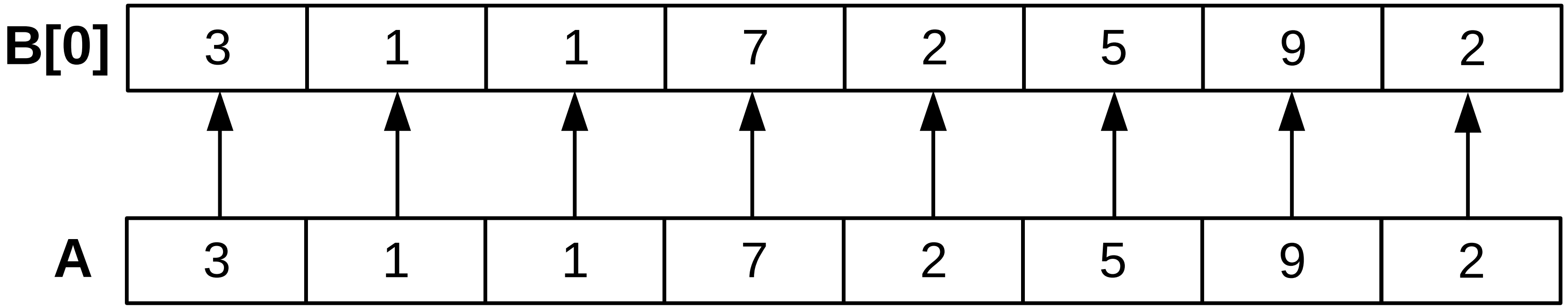
- 2 phases: up-sweep, down-sweep
- Up-sweep pseudocode:

$\textsc{PrefixSum}(A[0, ..., n-1])$

for $i = 0$ to $n-1$ **in parallel do**
    $B[0][i] = A[i]$
end for
for $h = 1$ to $\log n$ **do**
    for $i = 0$ to $\frac{n}{2^h} - 1$ **in parallel do**
        $B[h][i] = B[h-1][2i] + B[h-1][2i+1]$
    end for
end for

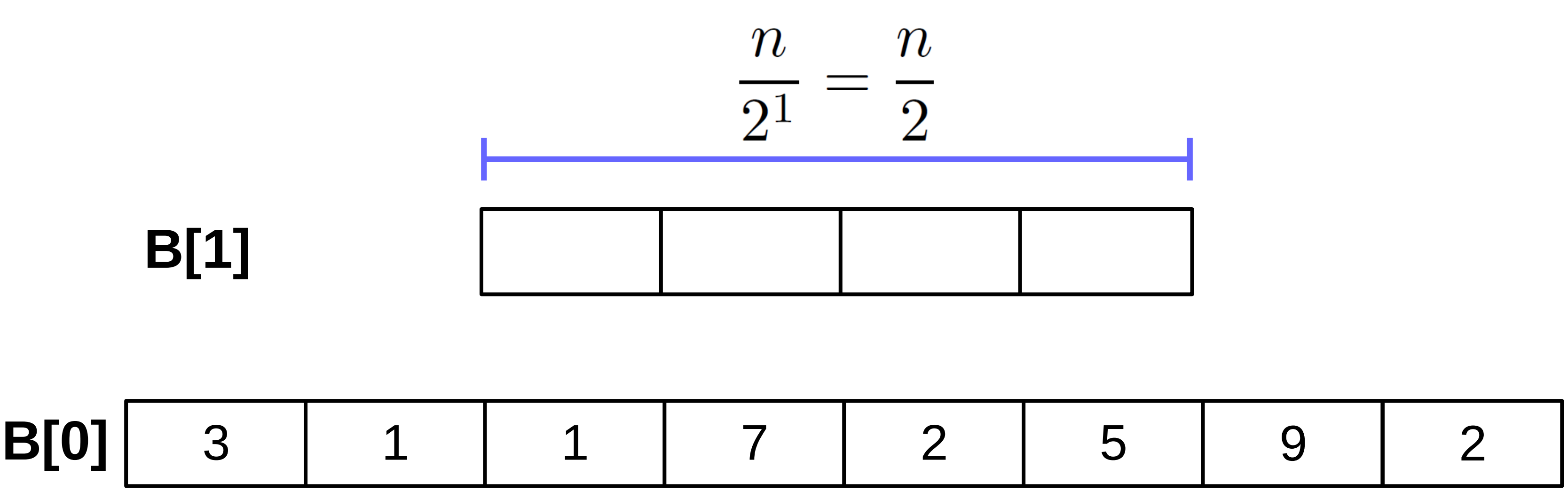
Up-sweep phase

for $i = 0$ to $n-1$ **in parallel do**
  $B[0][i] = A[i]$
end for



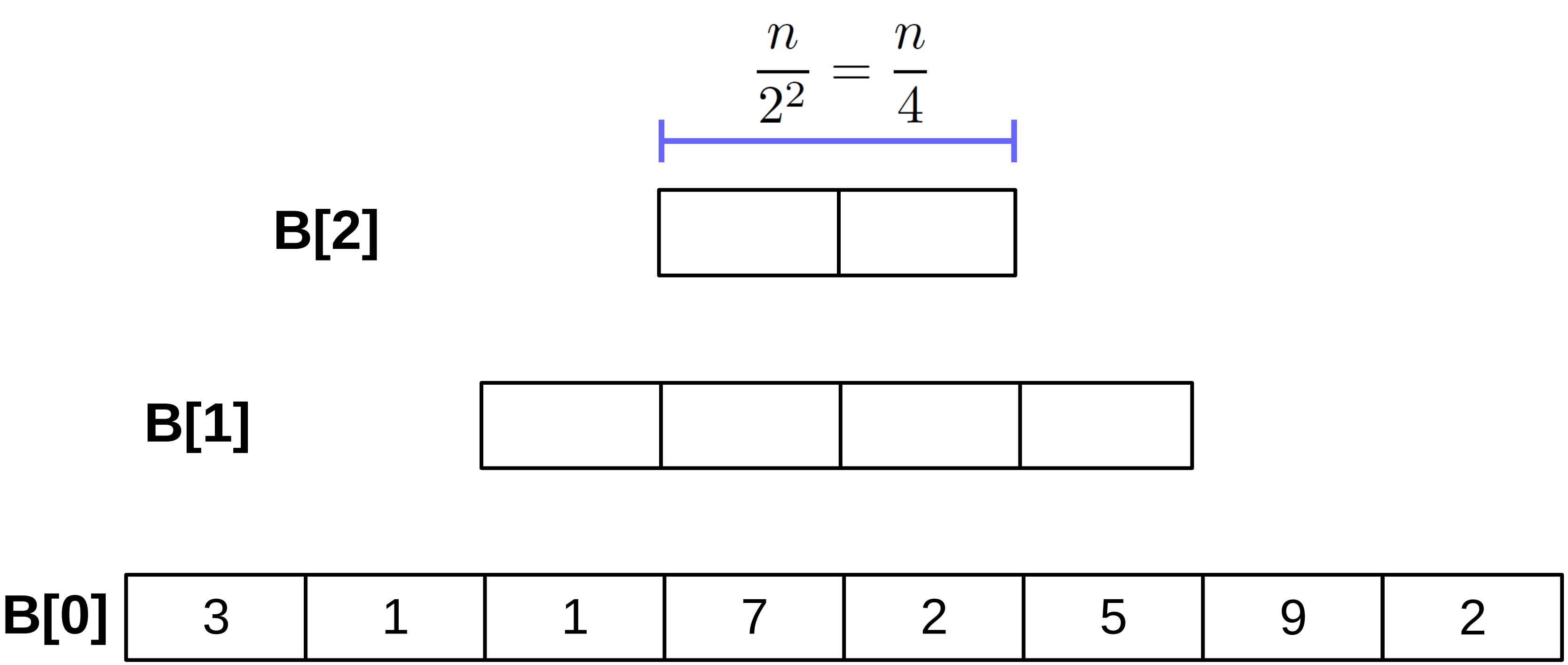
Up-sweep phase

for $h = 1$ to $\log n$ **do**
  for $\boxed{i = 0 \text{ to } \frac{n}{2^h} - 1}$ **in parallel do**
    $B[h][i] = B[h-1][2i] + B[h-1][2i+1]$
  end for
end for



Up-sweep phase

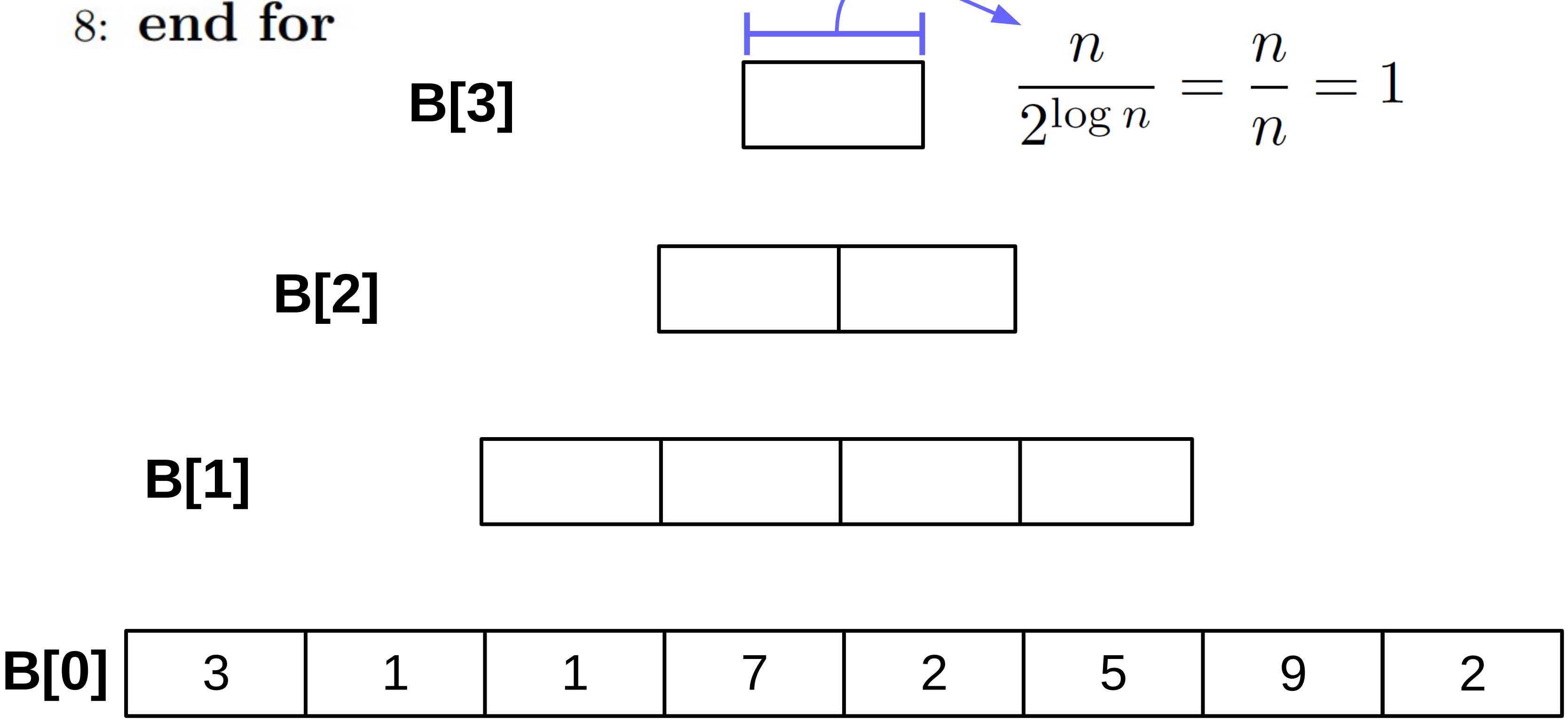
for $h = 1$ to $\log n$ **do**
  for $\boxed{i = 0 \text{ to } \frac{n}{2^h} - 1}$ **in parallel do**
    $B[h][i] = B[h-1][2i] + B[h-1][2i+1]$
  end for
end for



Up-sweep phase

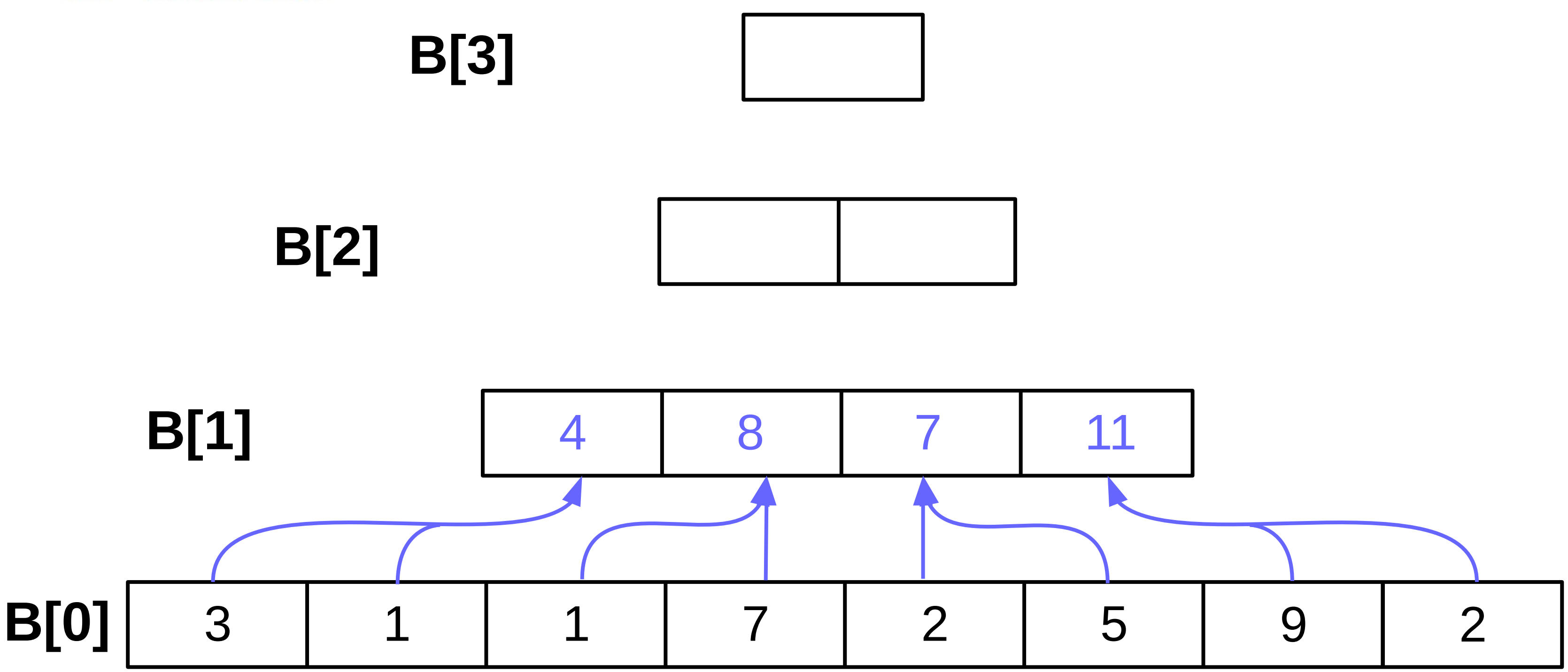
for $h = 1$ to $\log n$ **do**
    for $\boxed{i = 0 \text{ to } \frac{n}{2^h} - 1}$ **in parallel do**
        $B[h][i] = B[h-1][2i] + B[h-1][2i+1]$
    end for
end for

$$\frac{n}{2^{\log n}} = \frac{n}{n} = 1$$



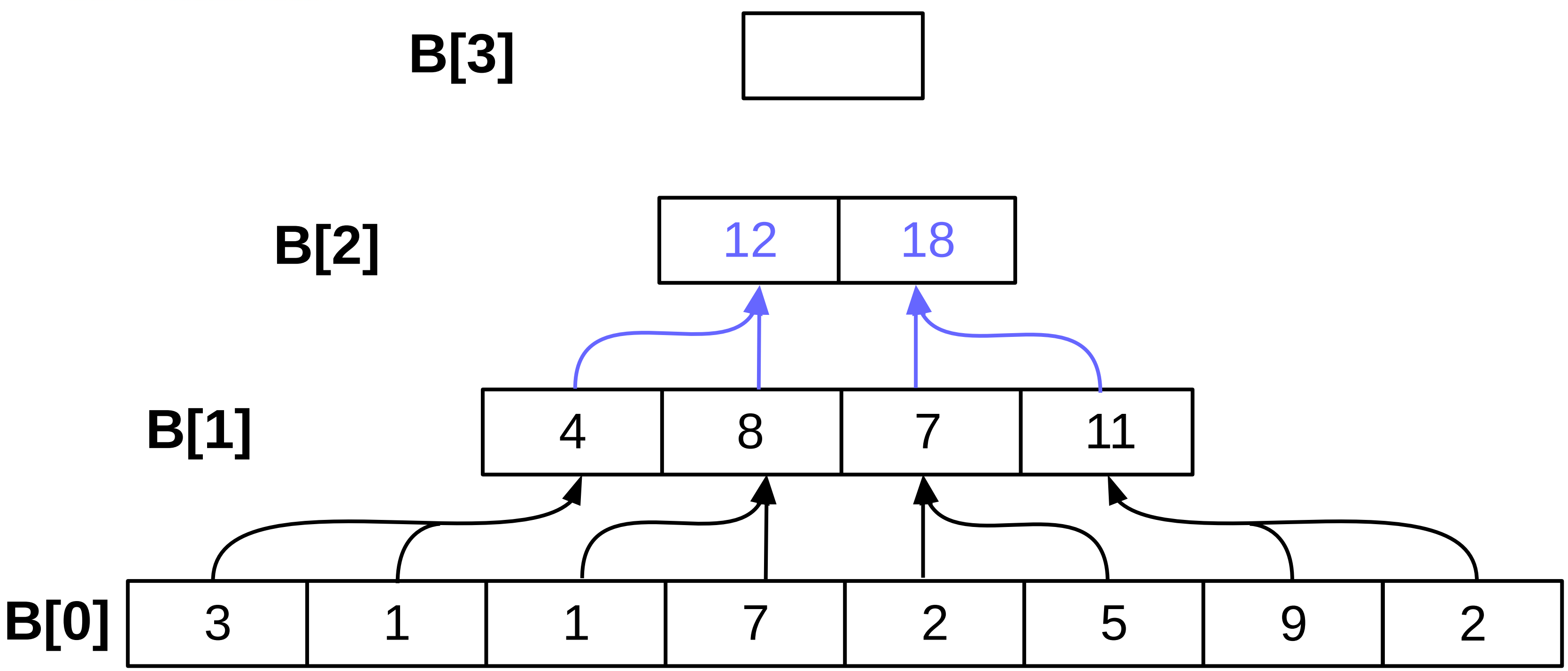
Up-sweep phase

> **for** $h = 1$ to $\log n$ **do**
>
>   **for** $i = 0$ to $\frac{n}{2^h} - 1$ **in parallel do**
>
> $B[h][i] = B[h-1][2i] + B[h-1][2i+1]$
>
>   **end for**
>
> **end for**



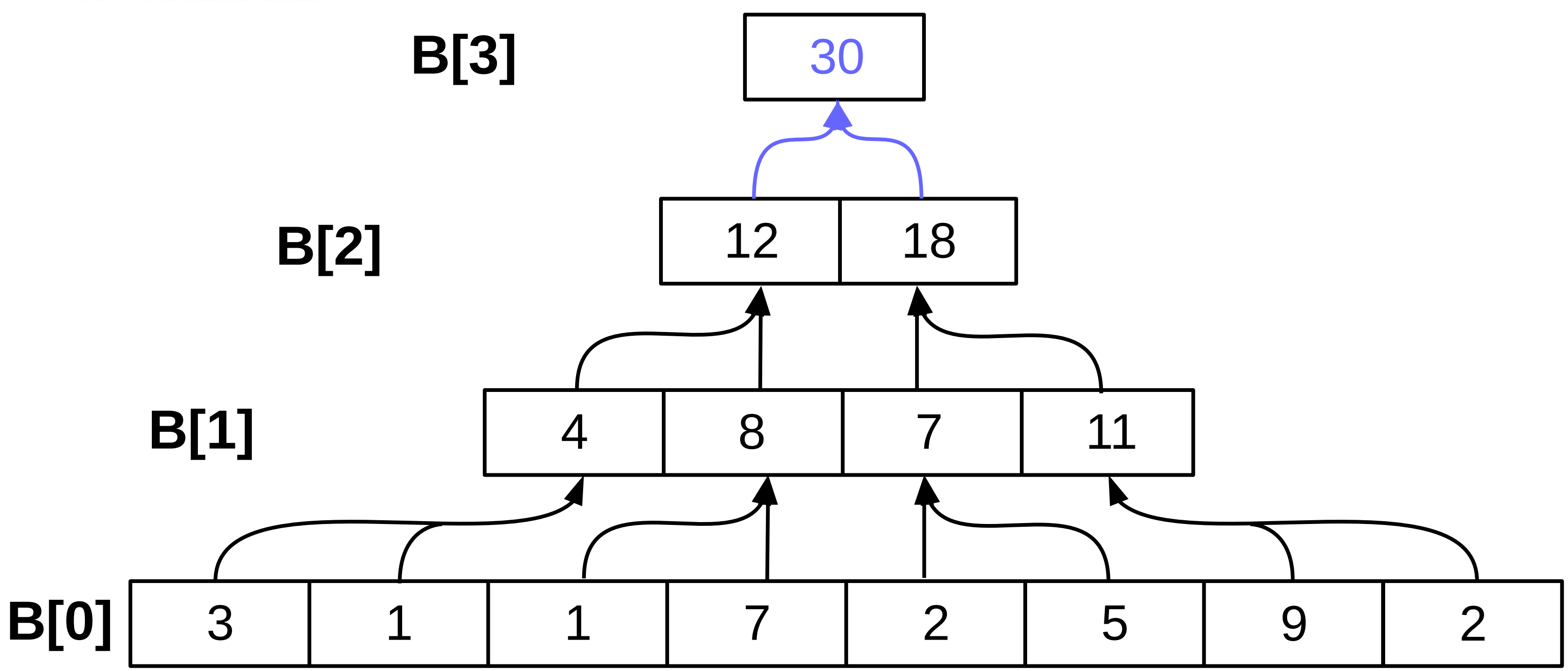
Up-sweep phase

for $h = 1$ to $\log n$ **do**
  for $i = 0$ to $\frac{n}{2^h} - 1$ **in parallel do**
    $B[h][i] = B[h-1][2i] + B[h-1][2i+1]$
  end for
end for



Up-sweep phase

for $h = 1$ to $\log n$ **do**
  for $i = 0$ to $\frac{n}{2^h} - 1$ **in parallel do**
    $B[h][i] = B[h-1][2i] + B[h-1][2i+1]$
  end for
end for



Down-sweep phase

$C[\log n][0] = 0$
for $h = \log n - 1$ down to 0 **do**
  for $i = 0$ to $\frac{n}{2^h} - 1$ **in parallel do**
    if i % $2 == 0$ **then**
      $C[h][i] = C[h+1][i/2]$
    else
      $C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]$
    end if
  end for
end for
for $i = 0$ to $n - 1$ **in parallel do**
  $A[i] = A[i] + C[0, i]$
end for

Down-sweep phase

$$9: \; C[\log n][0] = 0$$

C[3]

C[3]	0

B[2]	12	18

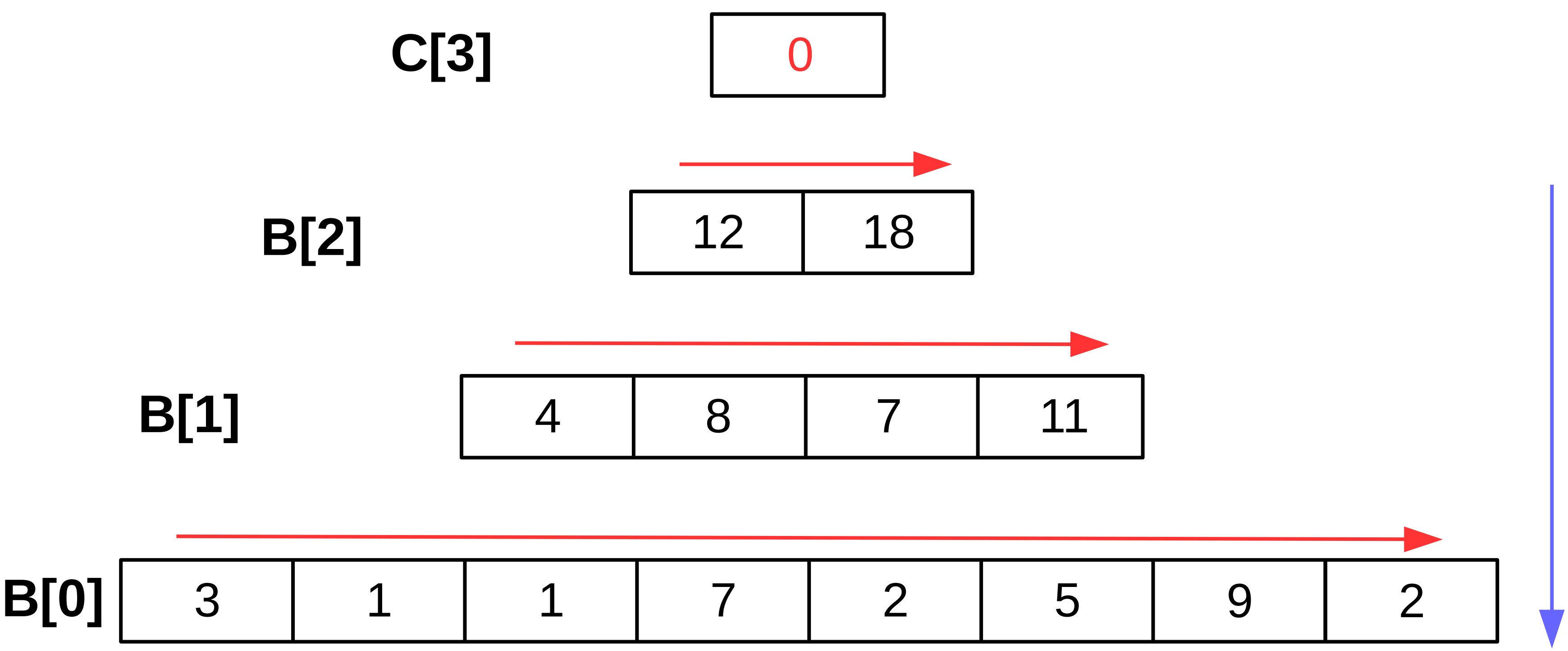
B[1]	4	8	7	11

B[0]	3	1	1	7	2	5	9	2

Down-sweep phase

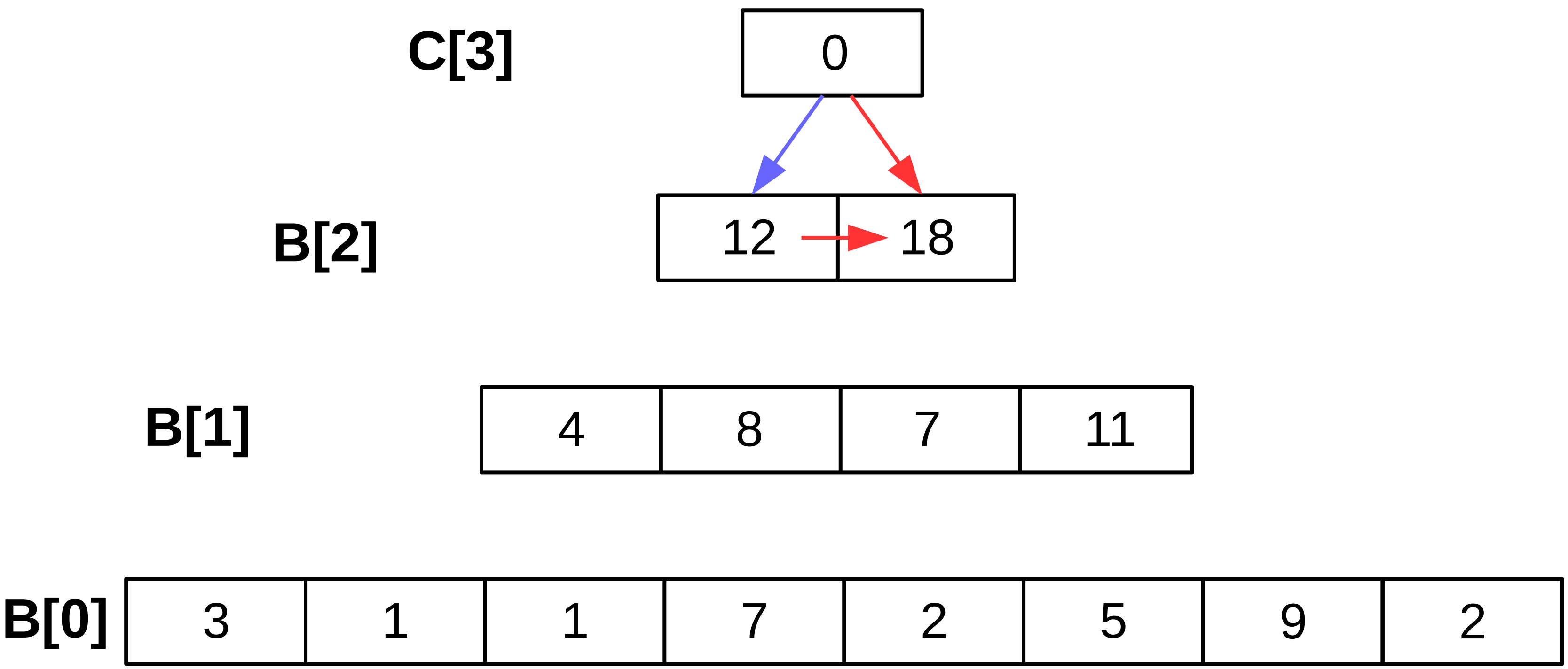
for $h = \log n - 1$ down to 0 **do**

for $i = 0$ to $\frac{n}{2^h} - 1$ **in parallel do**



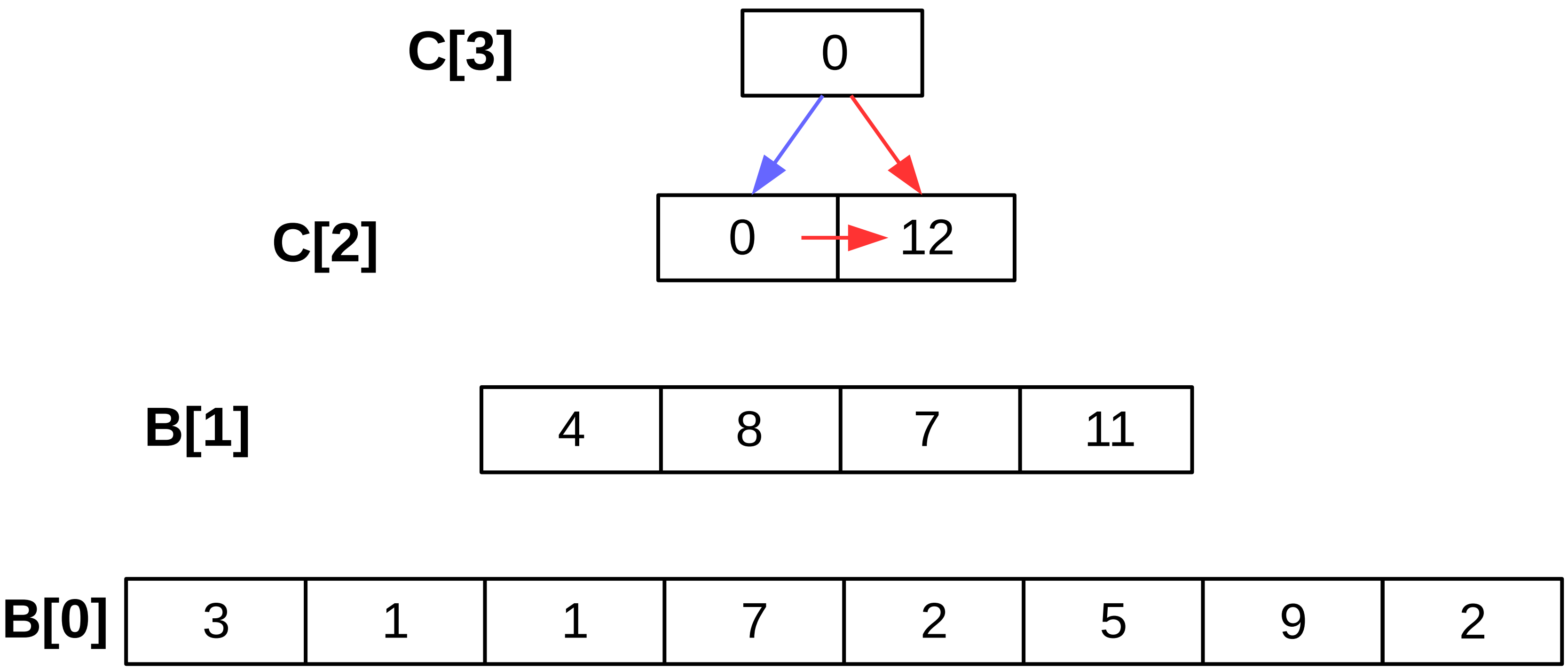
Down-sweep phase

if $i \ \% \ 2 == 0$ **then**
$\quad C[h][i] = C[h+1][i/2]$
else
$\quad C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]$



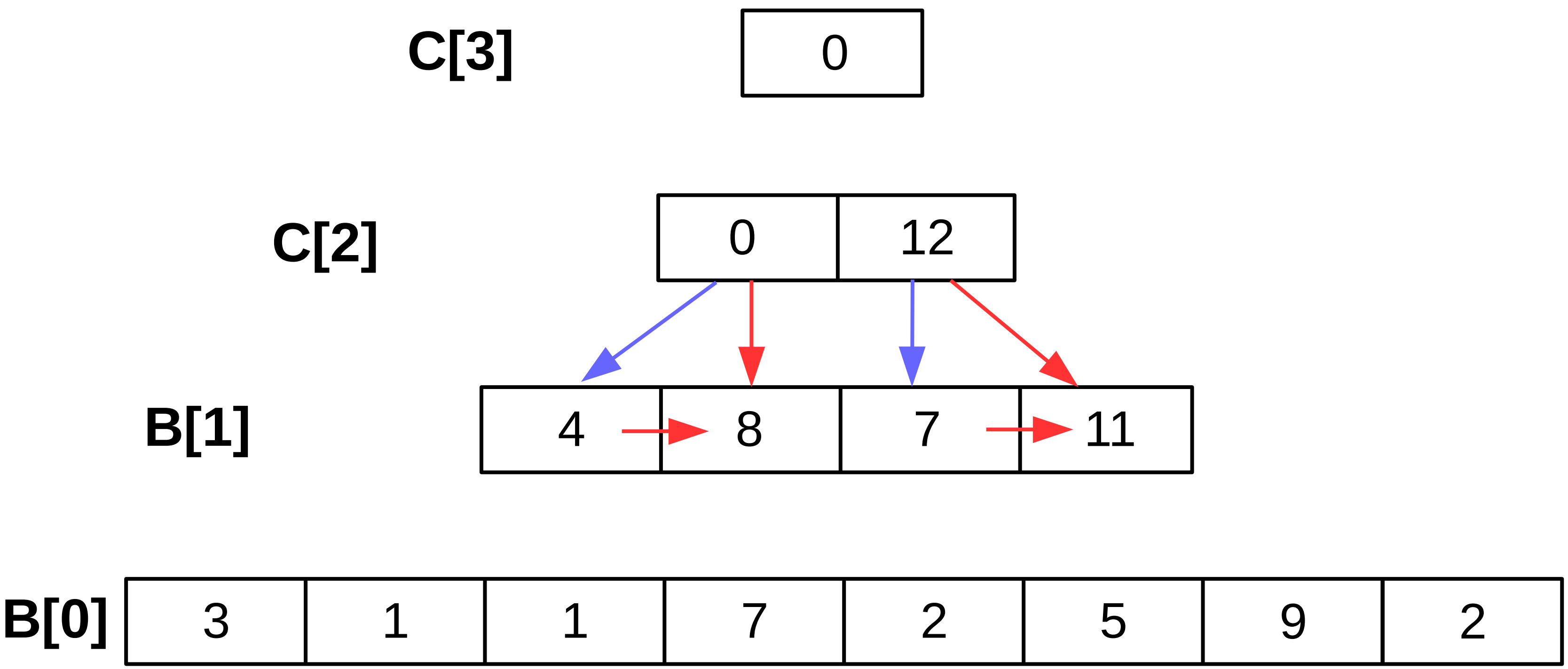
Down-sweep phase

if $i \ \% \ 2 == 0$ **then**
  $C[h][i] = C[h+1][i/2]$
else
  $C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]$



Down-sweep phase

if $i \;\%\; 2 == 0$ **then**
$\quad C[h][i] = C[h+1][i/2]$
else
$\quad C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]$



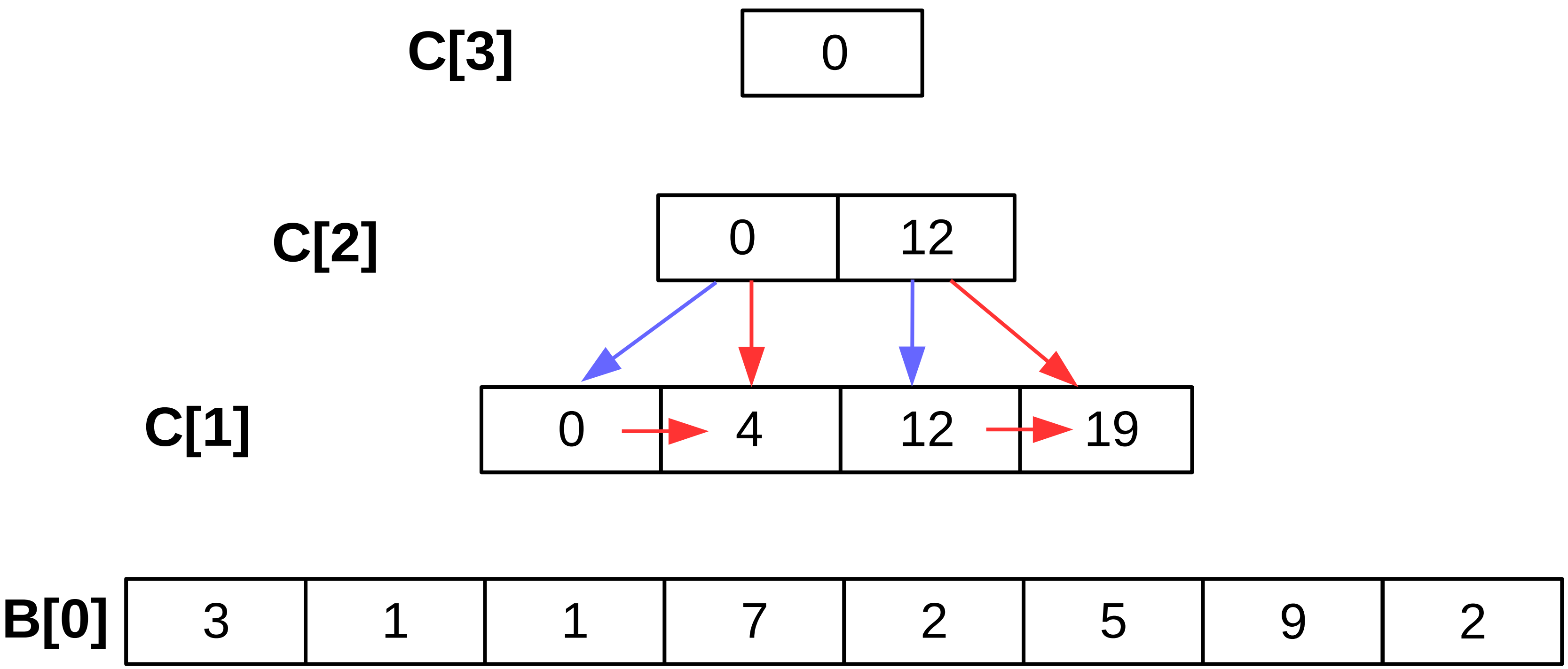
Down-sweep phase

if $i \,\%\, 2 == 0$ **then**

$\quad C[h][i] = C[h+1][i/2]$

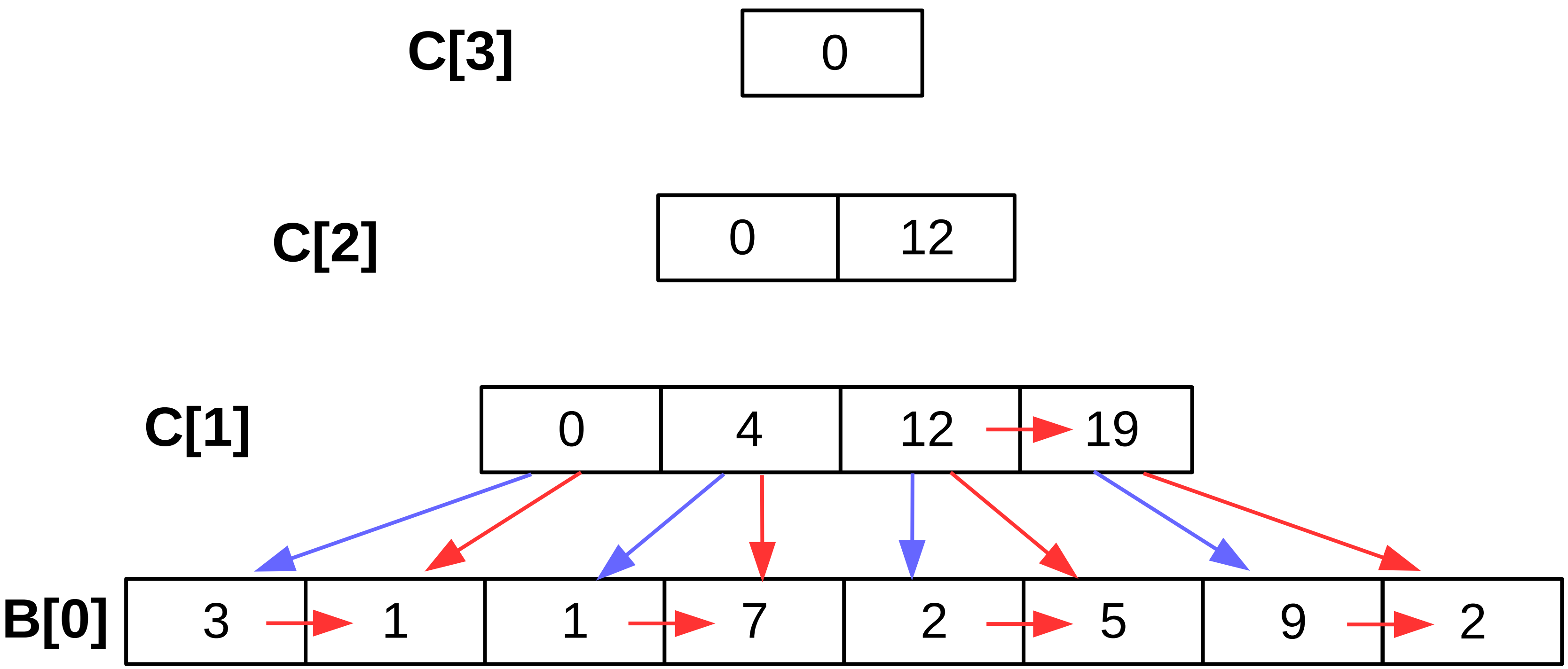
else

$\quad C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]$



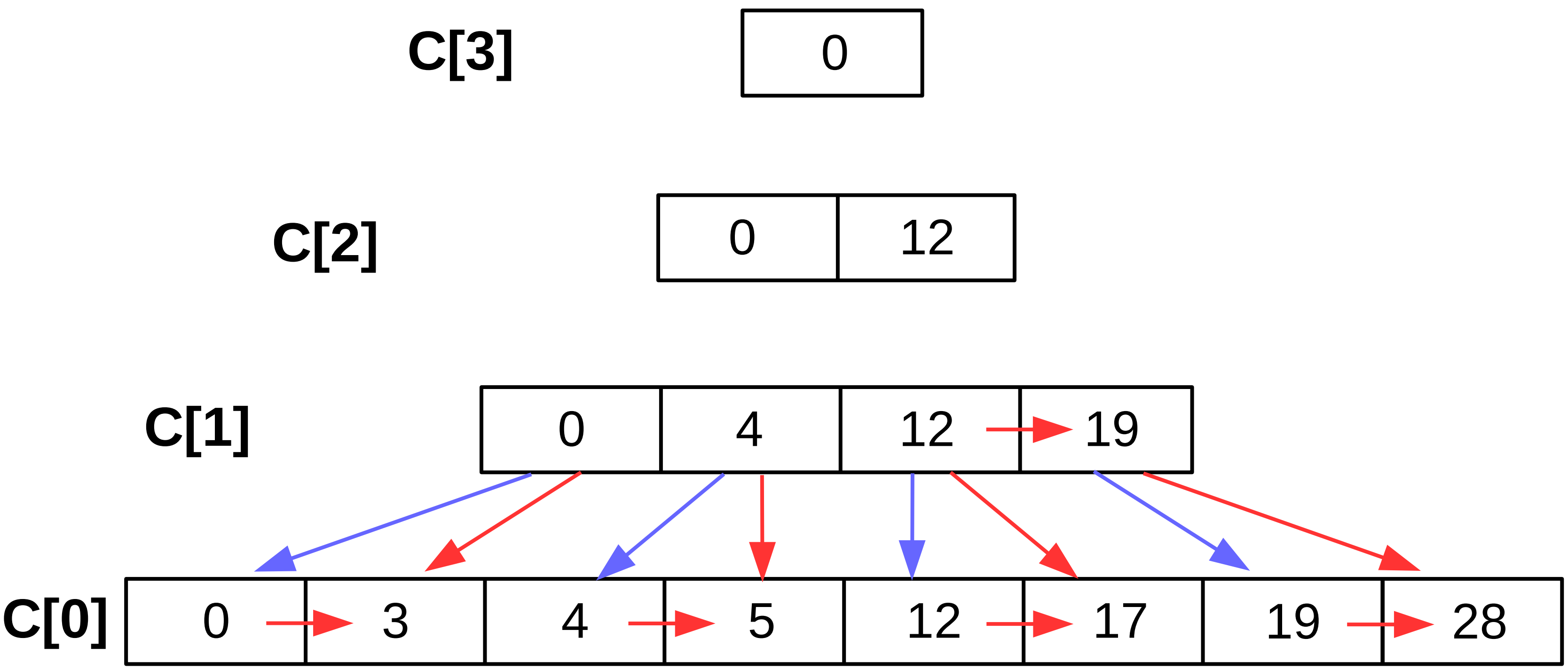
Down-sweep phase

if i % $2 == 0$ **then**
  $C[h][i] = C[h+1][i/2]$
else
  $C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]$



Down-sweep phase

if $i \ \% \ 2 == 0$ **then**
  $C[h][i] = C[h+1][i/2]$
else
  $C[h][i] = C[h+1][\frac{i-1}{2}] + B[h][i-1]$



Down-sweep phase

for $i = 0$ to $n - 1$ **in parallel do**
    $A[i] = A[i] + C[0, i]$
end for

C[0]	0	3	4	5	12	17	19	28
A	3	1	1	7	2	5	9	2

Down-sweep phase

for $i = 0$ to $n - 1$ **in parallel do**

$\quad A[i] = A[i] + C[0, i]$

end for

C[0]	0	3	4	5	12	17	19	28
A	3	4	5	12	14	22	28	30

Applications of prefix sums

- More useful than it seems:
  - Create an array of 1s and 0s
  - Prefix sums gives # of 1s up to each point
  - Used to ***separate*** an array into 2
    - Using almost **any** criteria!
- Examples:
  - separate array into upper-case and lower-case letters
  - separate array into numbers >x and <x

Example: string separation

- Separate array **A** into lower-case and upper-case:

A	a	P	r	e	R	E	c	F	l	o	X	o	S	U	l	M	S

Example: string separation

- Create bitstring B:
- 1 if upper-case, 0 otherwise

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S

Example: string separation

- Create bitstring B:
- 1 if upper-case, 0 otherwise

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	0	0	1	1	0	1	1	0	1	0	1	1	0	1	1

- Time/work to do this in parallel?

Example: string separation

- Create bitstring B:
- 1 if upper-case, 0 otherwise

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	0	0	1	1	0	1	1	0	1	0	1	1	0	1	1

- Time/work to do this in parallel?

$$W(n) = O(n)$$
$$T(n) = O(1)$$

Example: string separation

- Perform **prefix sums** on B

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	0	0	1	1	0	1	1	0	1	0	1	1	0	1	1

Example: string separation

- Perform **prefix sums** on B

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	1	1	2	3	3	4	5	5	6	6	7	8	8	9	10

- What is B[i]?

Example: string separation

- Perform **prefix sums** on B

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	1	1	2	3	3	4	5	5	6	6	7	8	8	9	10

- What is B[i]?
  - The number of capital letters with index ≤ i

Example: string separation

- Copy capital letters into C

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	1	1	2	3	3	4	5	5	6	6	7	8	8	9	10
C																	

- How can we use B to write **only capitals** into C?

Example: string separation

- Copy capital letters into C

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	1	1	2	3	3	4	5	5	6	6	7	8	8	9	10
C																	

- How can we use B to write **only capitals** into C?
  - B[i] is the **index** of each capital in C!

Example: string separation

- Copy capital letters into C

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B	0	1	1	1	2	3	3	4	5	5	6	6	7	8	8	9	10
C	P	R	E	F	I	X	S	U	M	S							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

- How can we use B to write **only capitals** into C?
  - B[i] is the **index** of each capital in C!

Example: string separation

- Create **B‘**
- 1 for lower-case, 0 otherwise

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B‘	1	0	1	1	0	0	1	0	0	1	0	1	0	0	1	0	0
C	P	R	E	F	I	X	S	U	M	S							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Example: string separation

- Prefix sums on **B'**

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B'	1	1	2	3	3	3	4	4	4	5	5	6	6	6	7	7	7
C	P	R	E	F	I	X	S	U	M	S							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Example: string separation

- Copy lower-case into the rest of C

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B'	1	1	2	3	3	3	4	4	4	5	5	6	6	6	7	7	7
C	P	R	E	F	I	X	S	U	M	S							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Example: string separation

- Copy lower-case into the rest of C

A	a	P	r	e	R	E	c	F	I	o	X	o	S	U	l	M	S
B‘	1	1	2	3	3	3	4	4	4	5	5	6	6	6	7	7	7
C	P	R	E	F	I	X	S	U	M	S	a	r	e	c	o	o	l
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
											1	2	3	4	5	6	7

- A[i] = C[j]
  - where j = B[n] + B‘[i] = 10 + B‘[i]

Example: string separation

	$W(n)$	$T(n)$
Create **B** and **B'**	$O(n)$	$O(1)$
Prefix sums		
Copy into **C**		
Total algorithm		

Example: string separation

	$W(n)$	$T(n)$
Create **B** and **B'**	$O(n)$	$O(1)$
Prefix sums	$O(n)$	$O(\log n)$
Copy into **C**		
Total algorithm		

Example: string separation

	$W(n)$	$T(n)$
Create **B** and **B'**	$O(n)$	$O(1)$
Prefix sums	$O(n)$	$O(\log n)$
Copy into **C**	$O(n)$	$O(1)$
Total algorithm	$O(n)$	$O(\log n)$

Quicksort Review

- Quicksort is a popular sorting algorithm
  - Works **in-place**
  - $O(n^2)$ worst-case
  - BUT $O(n \log n)$ **expected**

- Each recursive call:
  - Find pivot
  - Partition around pivot

Sequential Quicksort

Quicksort$(A[0, \cdots, n-1])$

```
pivot = random(1 ··· n)
swap(A[0], A[pivot])
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then
        swap(A[i], A[part])
        part++
    end
end
if part > 2 then
    Quicksort(A[0,··· ,part-1])
end
if part < n-1 then
    Quicksort(A[part,··· ,n-1])
end
```

Select pivot

$\text{pivot} = \text{random}(1 \cdots n)$

$\text{swap}(\text{A}[0], \text{A}[\text{pivot}])$

A	3	9	1	7	4	5	8	2

pivot

Select pivot

$\text{pivot} = \text{random}(1 \cdots n)$

$\text{swap}(A[0], A[\text{pivot}])$

A	4	9	1	7	3	5	8	2

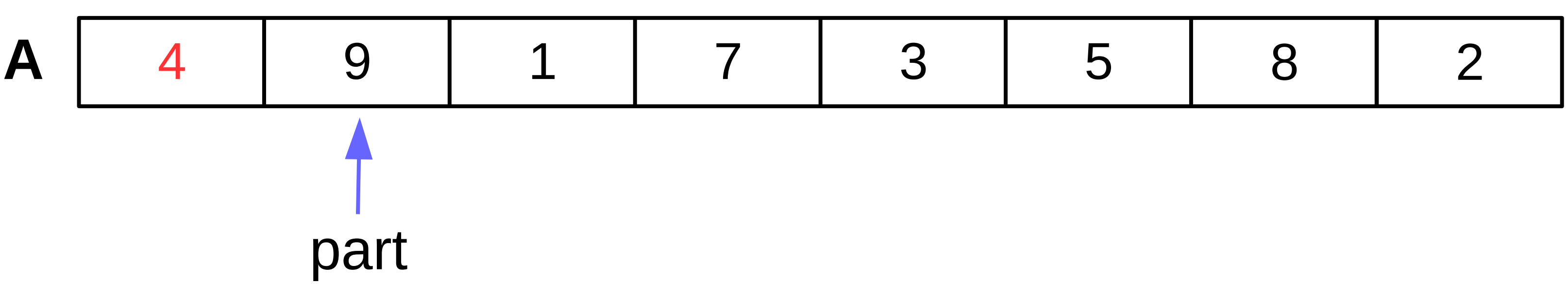
Partition elements

```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then
        swap(A[i], A[part])
        part++
    end
end
```

A	4	9	1	7	3	5	8	2

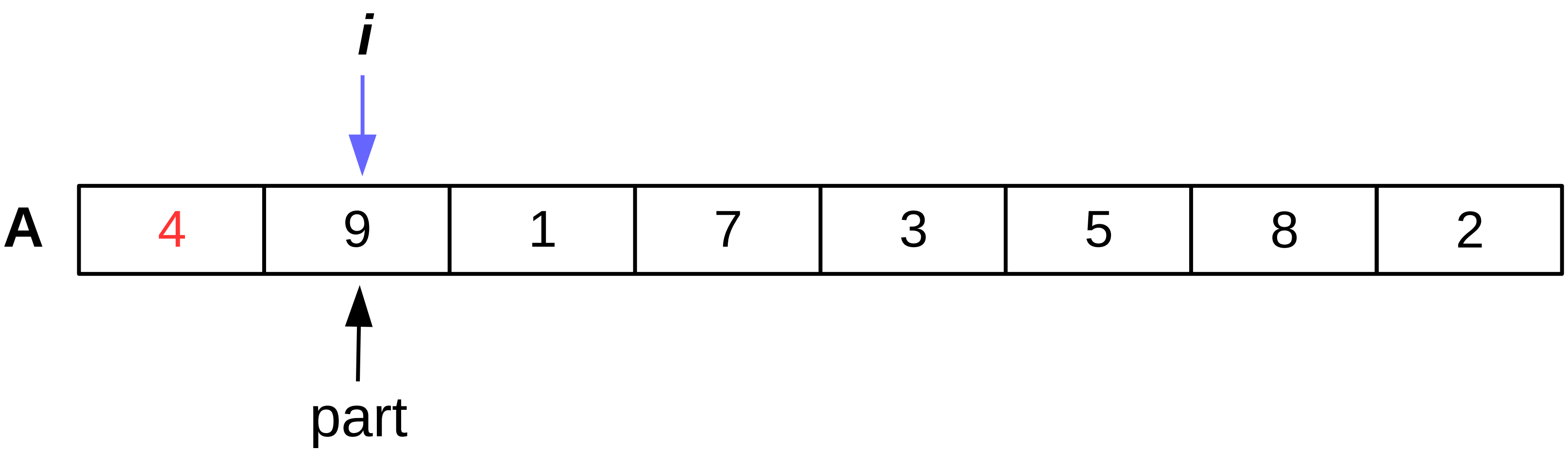
Partition elements

```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then
        swap(A[i], A[part])
        part++
    end
end
```



Partition elements

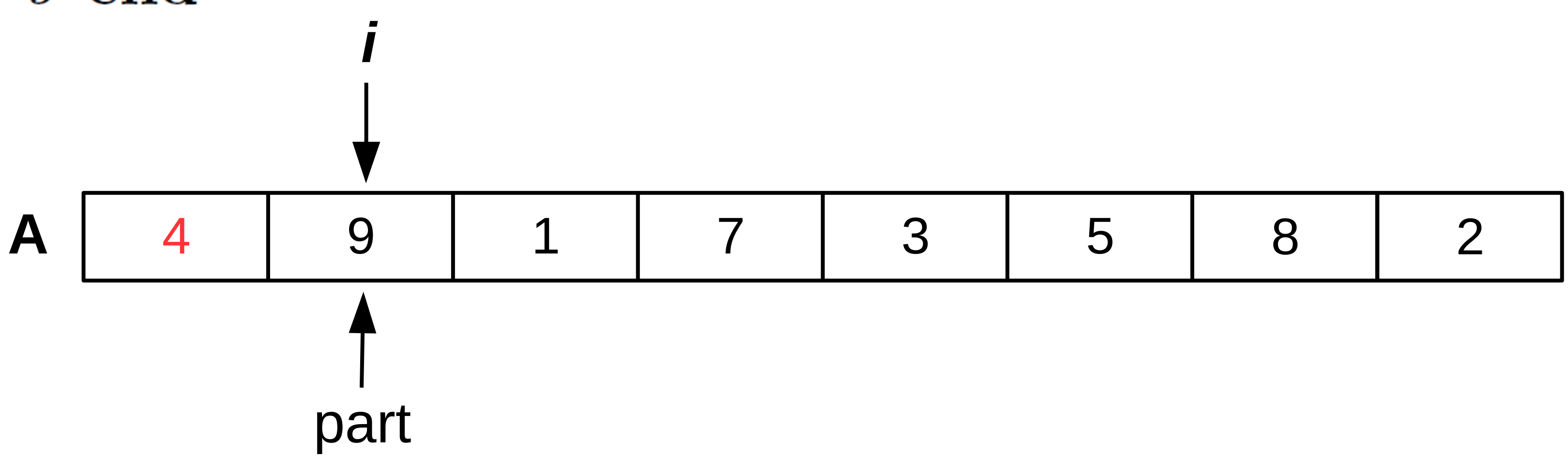
```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then
        swap(A[i], A[part])
        part++
    end
end
```



Partition elements

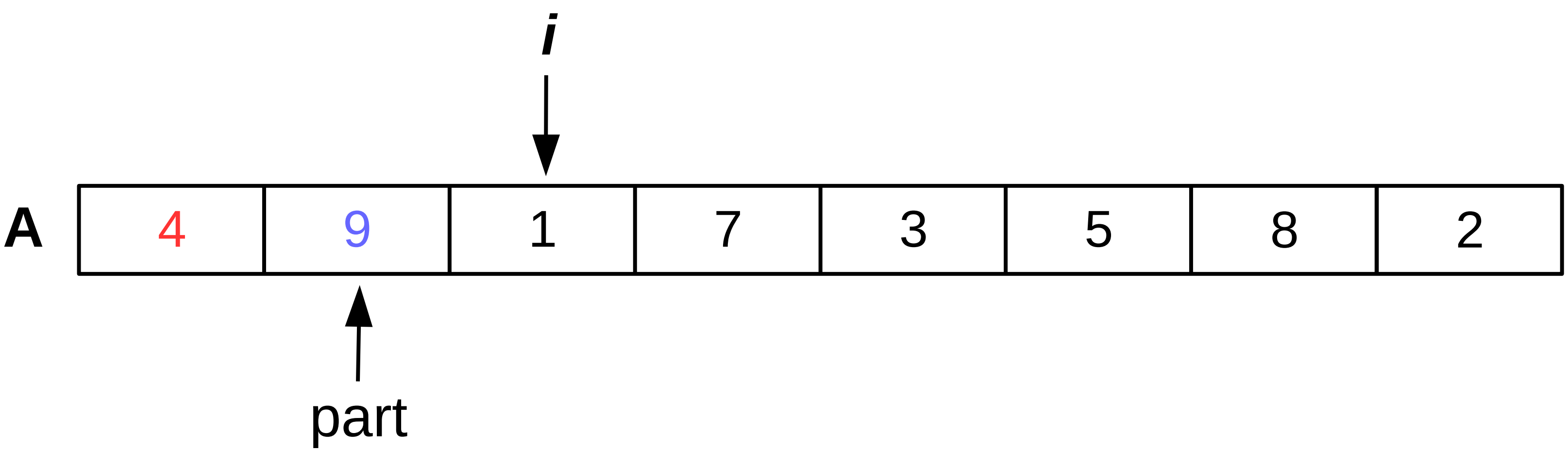
```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then
        swap(A[i], A[part])
        part++
    end
end
```

FALSE



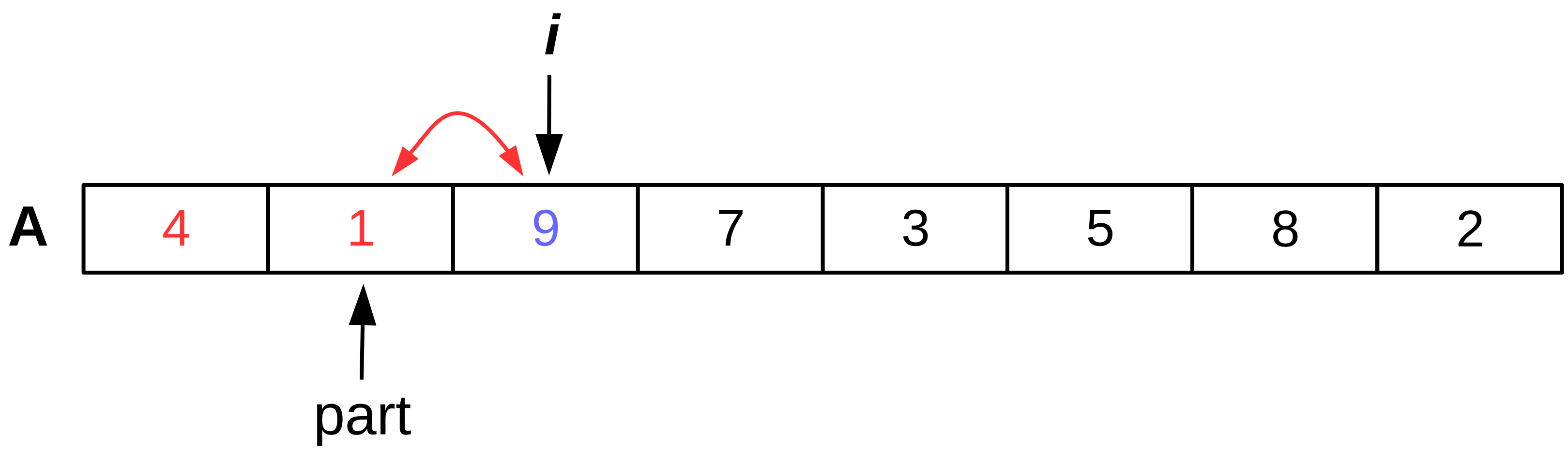
Partition elements

```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then        TRUE
        swap(A[i], A[part])
        part++
    end
end
```



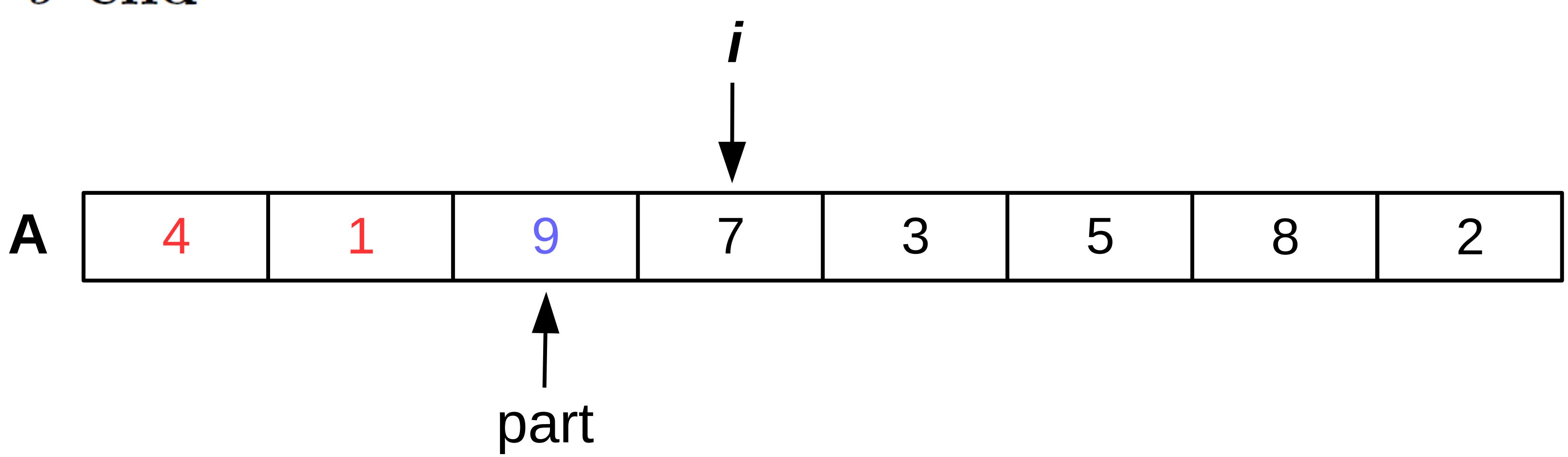
Partition elements

```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then    TRUE
        swap(A[i], A[part])
        part++
    end
end
```



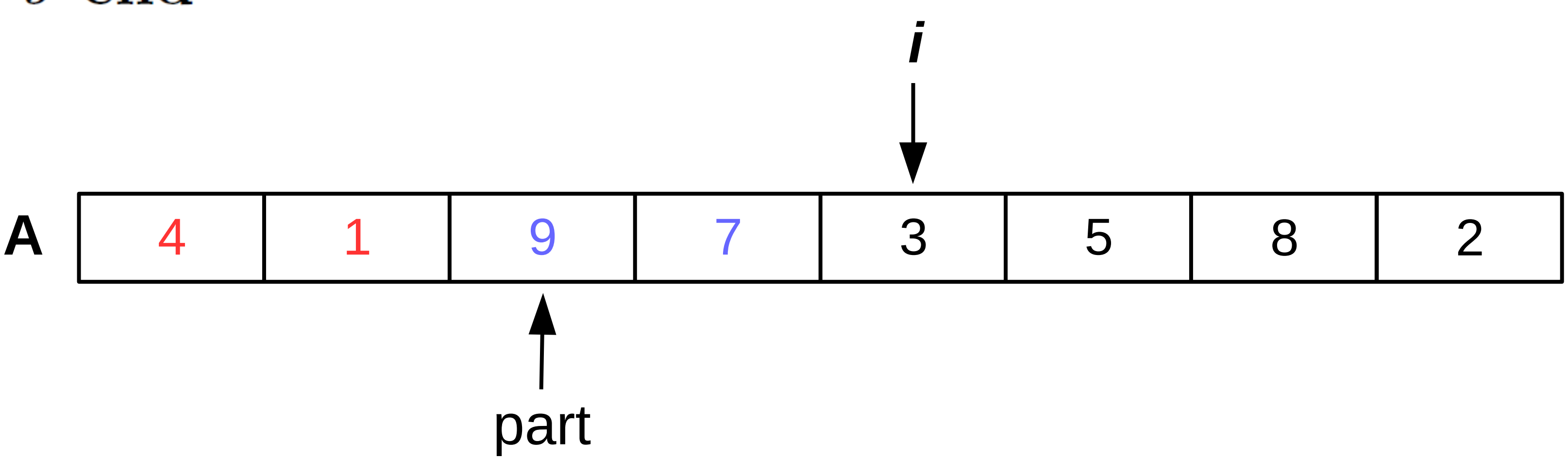
Partition elements

```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then    FALSE
        swap(A[i], A[part])
        part++
    end
end
```



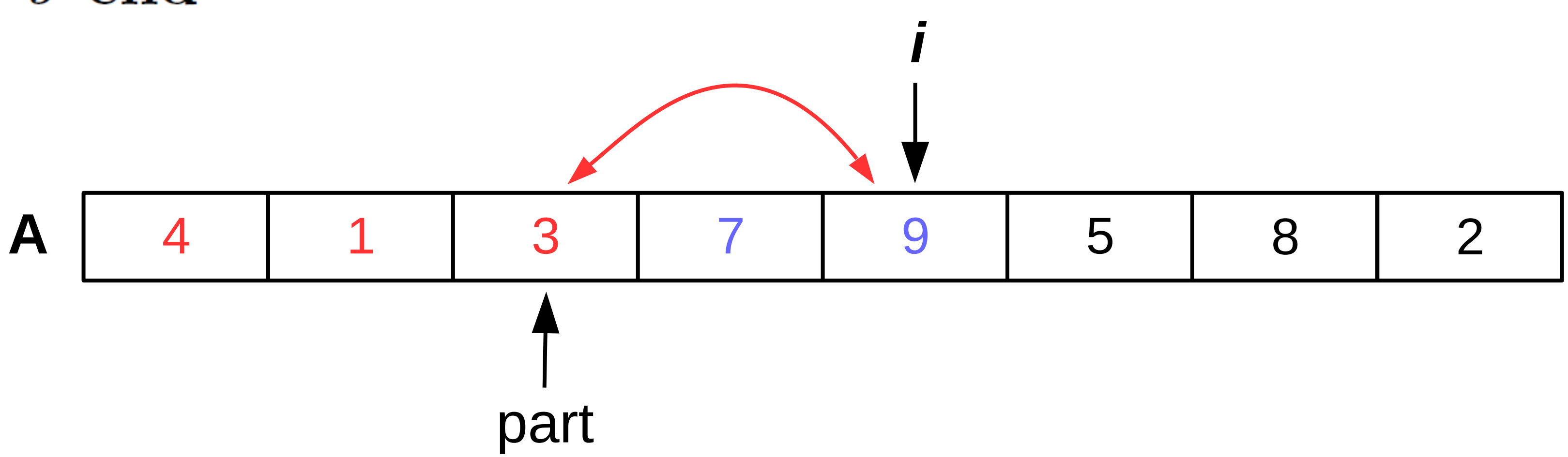
Partition elements

```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then    TRUE
        swap(A[i], A[part])
        part++
    end
end
```



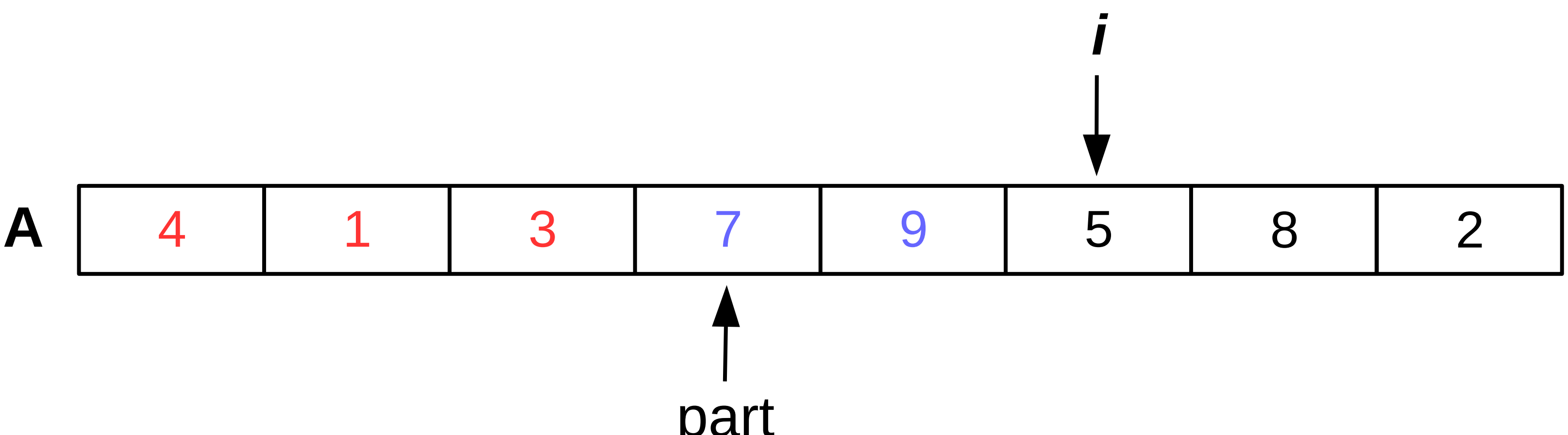
Partition elements

part =1
for i = *1 to n-1* **do**
    if $A[i] \leq A[0]$ **then**   TRUE
        swap(A[i], A[part])
        part++
    end
end



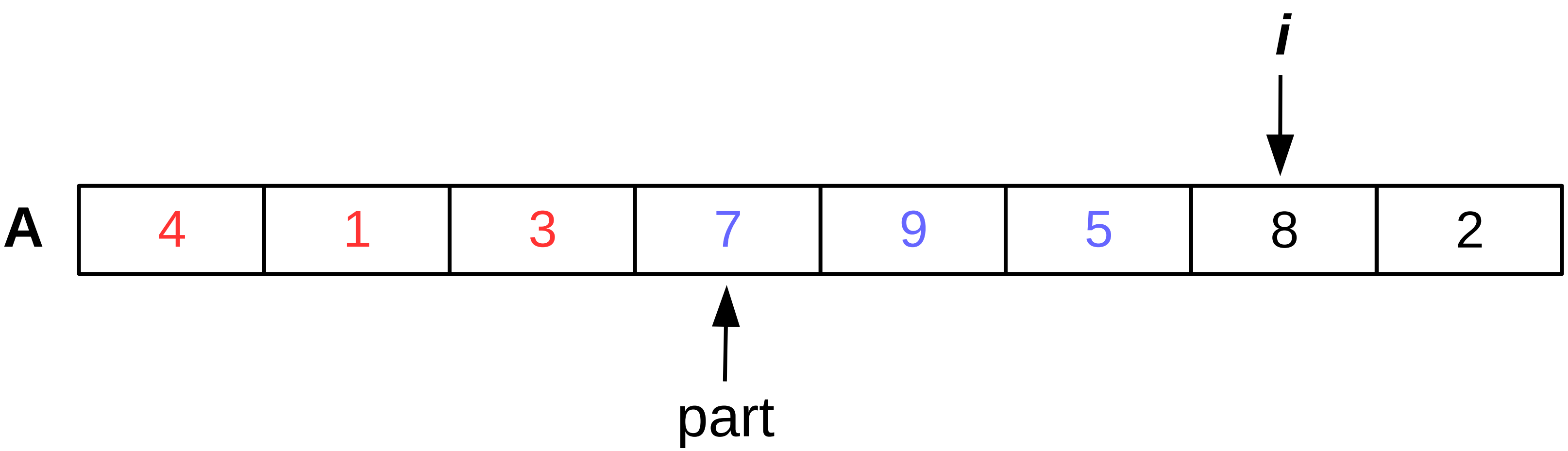
Partition elements

```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then      FALSE
        swap(A[i], A[part])
        part++
    end
end
```



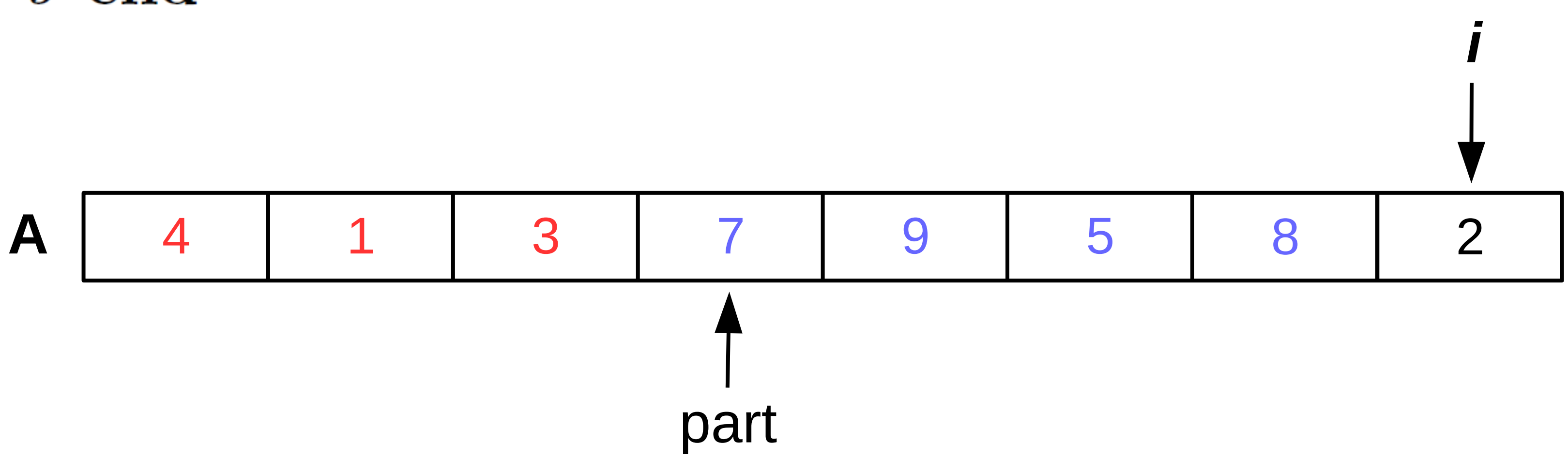
Partition elements

```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then      FALSE
        swap(A[i], A[part])
        part++
    end
end
```



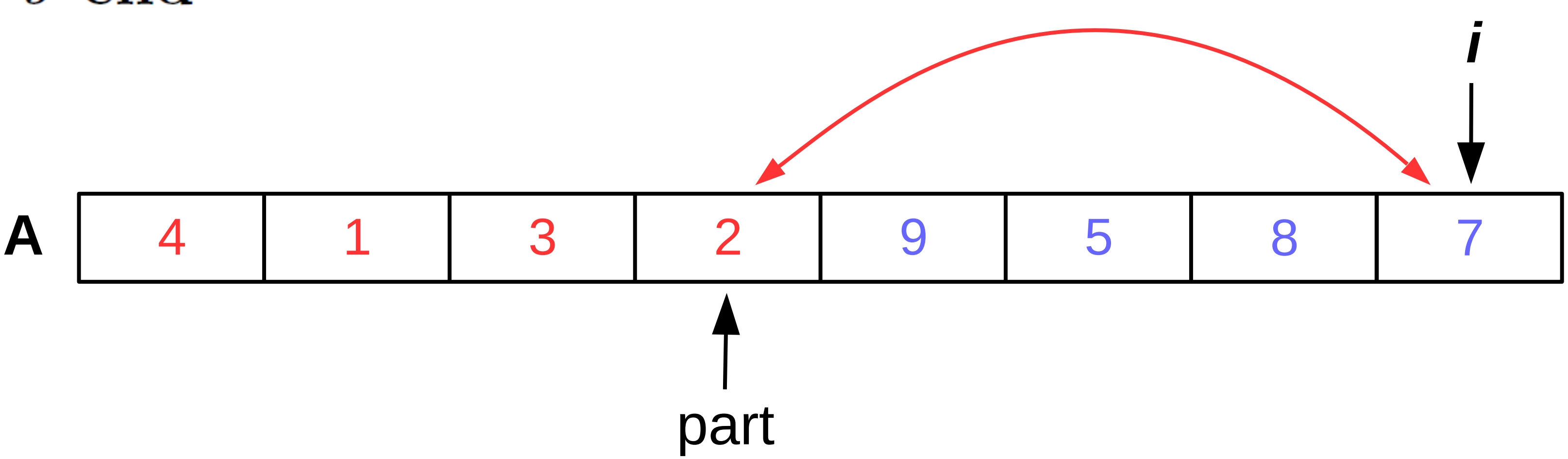
Partition elements

```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then        TRUE
        swap(A[i], A[part])
        part++
    end
end
```



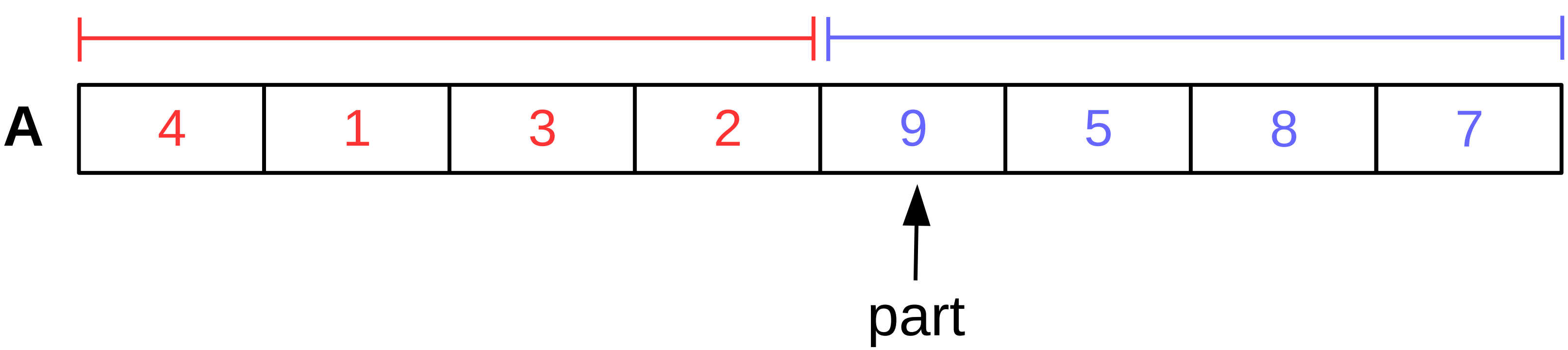
Partition elements

```
part =1
for i = 1 to n-1 do
    if A[i] ≤ A[0] then        TRUE
        swap(A[i], A[part])
        part++
    end
end
```



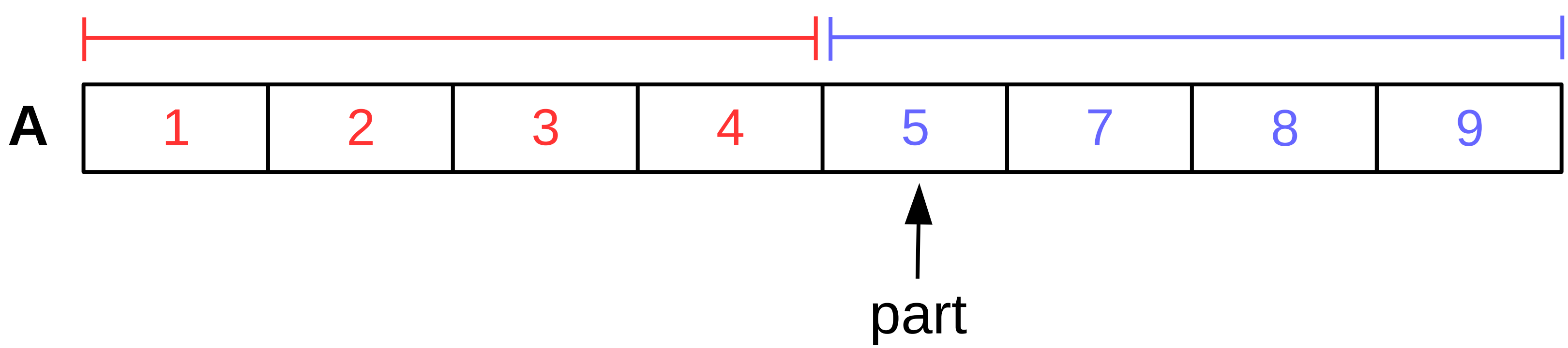
Recurse

```
if part > 2 then
    Quicksort(A[0,··· ,part-1])
end
if part < n-1 then
    Quicksort(A[part,··· ,n-1])
end
```



Recursion sorts sublists

```
if part > 2 then
    Quicksort(A[0,··· ,part-1])
end
if part < n-1 then
    Quicksort(A[part,··· ,n-1])
end
```



How can we parallelize?

Quicksort$(A[0, \cdots, n-1])$

pivot = random$(1 \cdots n)$
swap(A[0], A[pivot])
part =1

O(1)

for i = *1 to n-1* **do**
    if $A[i] \leq A[0]$ **then**
        swap(A[i], A[part])
        part++
    end
end

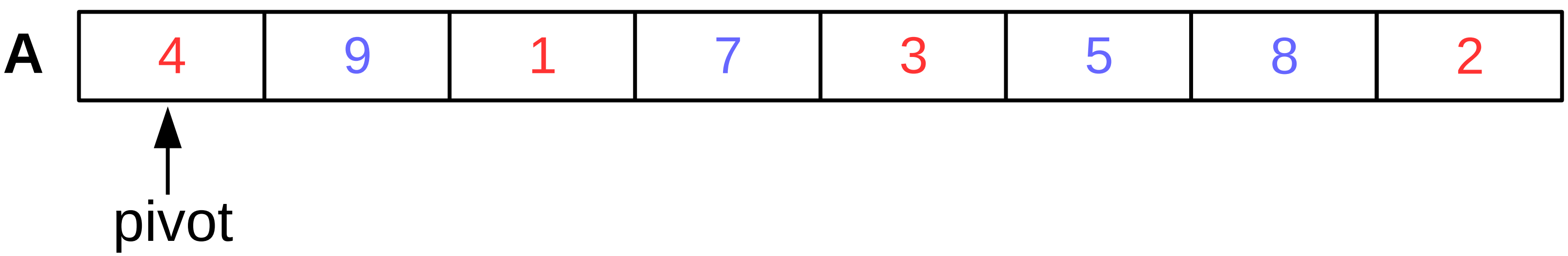
???

if *part* $>$ *2* **then**
    Quicksort(A[0,$\cdots$,part-1])
end
if *part* $<$ *n-1* **then**
    Quicksort(A[part,$\cdots$,n-1])
end

Parallel calls

Parallel partition

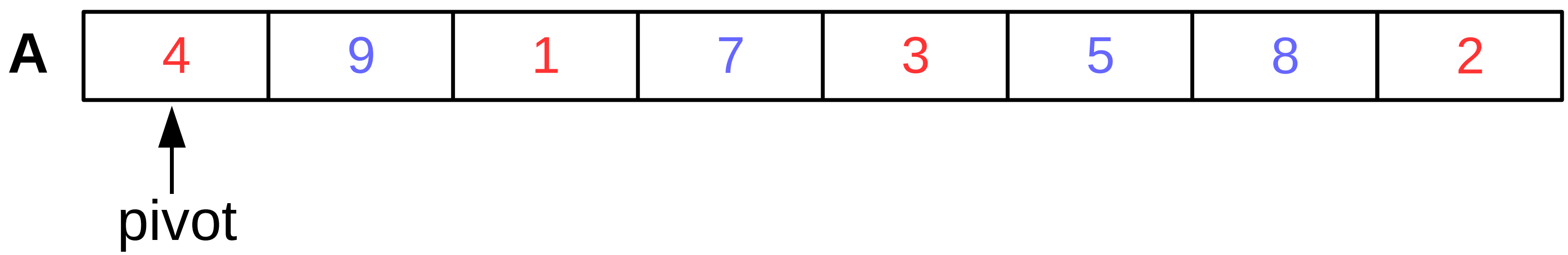
- **Separate** all elements ≤ pivot



- How can we do this in parallel?

Parallel partition

- **Separate** all elements ≤ pivot



- How can we do this in parallel?
  - Prefix sums!

Parallel partition

- Create **B[i]** by comparing A[i] to pivot
  - 1 if A[i] ≤ A[0]
  - 0 otherwise

A	4	9	1	7	3	5	8	2
B	1	0	1	0	1	0	0	1

Parallel partition

- Prefix sums on B

A	4	9	1	7	3	5	8	2
B	1	1	2	2	3	3	3	4

Parallel partition

- Write each A[i] ≤ A[0] to array **C**
  - C[B[i]] = A[i]

A	4	9	1	7	3	5	8	2
B	1	1	2	2	3	3	3	4
C	4	1	3	2				

Parallel partition

- Create **B‘** as opposite of **B**
  - B‘[i] = 1 if A[i] > A[0]
  - B‘[i] = 0 otherwise

A	4	9	1	7	3	5	8	2

B‘	0	1	0	1	0	1	1	0

C	4	1	3	2				

Parallel partition

- Prefix sums on **B‘**

A	4	9	1	7	3	5	8	2
B‘	0	1	1	2	2	3	4	4
C	4	1	3	2				

Parallel partition

- Write remaining elements to **C**
  - C[B[n-1] + B‘[i]] = A[i]

A	4	9	1	7	3	5	8	2
B‘	0	1	1	2	2	3	4	4
C	4	1	3	2	9	7	5	8

Parallel quicksort analysis

- Each recursive call performs prefix sum
- Worst-case, pivot is always min or max:

$$W(n) = W(n-1) + O(n) = O(n^2)$$

- If we assume “good“ pivot is chosen:

$$W(n) = W(\frac{n}{2}) + O(n) = O(n \log n)$$

Parallel quicksort analysis

- Assuming a “good“ pivot choice:

$$T(n) = T(\frac{n}{2}) + O(\log n)$$

$$\begin{aligned} &= \log n + \log \frac{n}{2} + \cdots + \frac{n}{n} \\ &= \log n) + (\log n - 1) + (\log n - 2) + \cdots + 1 \\ &= \frac{(\log n)(\log n + 1)}{2} = O(\log^2 n) \end{aligned}$$

Issues with parallel quicksort

- Have to copy **A** to **C** => **not** in-place
  - O(n) extra space needed
- $O(\log^2 n)$ “average“ parallel runtime
- Recursive definition
  - Difficult to make iterative
  - Perform **many** small prefix-sums
    - Performance overhead

Iterative solution

- What if we can combine recursive calls
  - One iteration for each level

- Separate recursive calls on partitions:

	P_0				P_1			P_2					P_3			
A	1	3	5	2	7	6	9	12	16	19	17	14	22	25	20	23

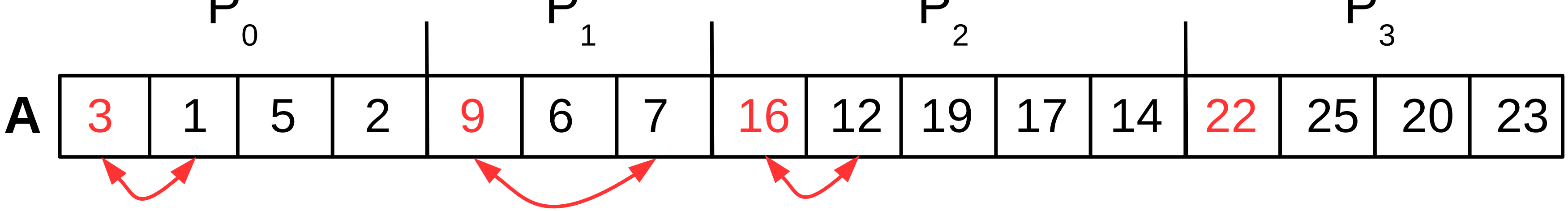
Iterative solution

- Know size of partition i = $|P_i|$
- Find a pivot for each partition

	P_0				P_1			P_2					P_3			
A	1	3	5	2	7	6	9	12	16	19	17	14	22	25	20	23

Iterative solution

- Know size of partition i = $|P_i|$
- Find a pivot for each partition
  - Move pivots to front



Iterative solution

- Know size of partition i = $|P_i|$
- Find a pivot for each partition
  - Move pivots to front
- Compute **B**
  - Compare each to the pivot in its partition

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B	1	1	0	1	1	1	1	1	1	0	0	1	1	0	1	0

Iterative solution

- Want prefix sum **within** each partition:
- ***Segmented prefix sums***
  - Each partition is a separate **segment**

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B	1	1	0	1	1	1	1	1	1	0	0	1	1	0	1	0

Iterative solution

- Want prefix sum **within** each partition:
- ***Segmented prefix sums***
  - Each partition is a separate **segment**
  - Can combine into 1 operation...

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B	1	2	2	3	1	2	3	1	2	2	2	3	1	1	2	2

Segmented prefix sums

- Input array **A** and *flag bits* **F**
  - 1 if start of new segment
  - 0 otherwise
- Prefix sums, except sum **resets** when F[i]=1

A	3	1	4	1	5	2	1	3	4	0	2	6	1	0	3	4
F	0	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0

Segmented prefix sums

- Input array **A** and *flag bits* **F**
  - 1 if start of new segment
  - 0 otherwise
- Prefix sums, except sum **resets** when F[i]=1

A	3	1	4	1	5	2	1	3	4	0	2	6	1	0	3	4
F	0	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0
C	3	4	8	1	6	8	9	12	16	0	2	6	1	1	4	8

Partition with segments

- Create **F** with partition boundaries

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B	1	1	0	1	1	1	1	1	1	0	0	1	1	0	1	0
F	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0

Partition with segments

- Create **F** with partition boundaries
- Perform ***segmented prefix sums*** on **B** and **F**

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B	1	2	2	3	1	2	3	1	2	2	2	3	1	1	2	2
F	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0

Partition with segments

- Create **F** with partition boundaries
- Perform ***segmented prefix sums*** on **B** and **F**
- Copy **A[i]** into **C[B[i]]** (plus partition offsets)

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B	1	2	2	3	1	2	3	1	2	2	2	3	1	1	2	2
C	3	1	2		9	6	7	16	12	14			22	20		

Partition with segments

- Repeat for > pivots:
  - Build **B‘**

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B‘	0	0	1	0	0	0	0	0	0	1	1	0	0	1	0	1
C	3	1	2		9	6	7	16	12	14			22	20		

Partition with segments

- Repeat for > pivots:
  - ***Segmented prefix sums*** on **B‘**

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B‘	0	0	1	1	0	0	0	0	0	1	2	2	0	1	1	2
C	3	1	2		9	6	7	16	12	14			22	20		

Partition with segments

- Repeat for > pivots:
  - Copy remaining **A** values into **C**

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B‘	0	0	1	1	0	0	0	0	0	1	2	2	0	1	1	2
C	3	1	2	5	9	6	7	16	12	14	19	17	22	20	25	23

Partition with segments

- Ready for next iteration...

	P_0				P_1			P_2					P_3			
A	3	1	5	2	9	6	7	16	12	19	17	14	22	25	20	23
B‘	0	0	1	1	0	0	0	0	0	1	2	2	0	1	1	2
C	3	1	2	5	9	6	7	16	12	14	19	17	22	20	25	23

Notes about Iterative quicksort

- Need to keep track of partition offsets, etc.
- Still need to pick good pivots
- Same runtime as recursive

- Easier to optimize
  - Unroll loops, etc.
- Less overhead (on most architectures)