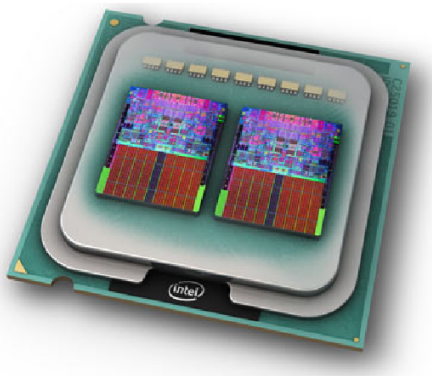
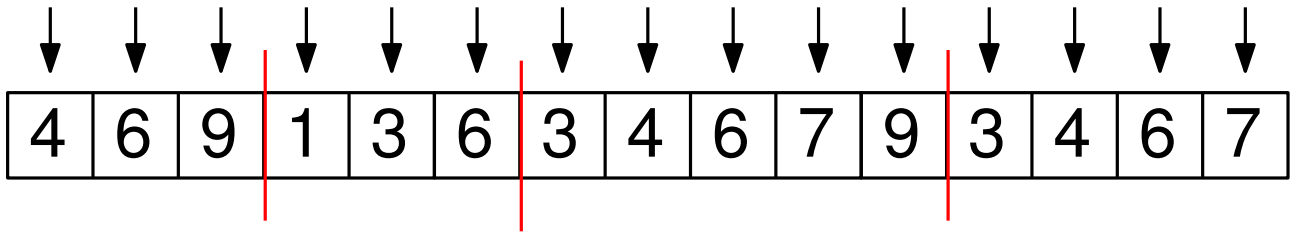
INFORMATION & COMPUTER SCIENCES
UNIVERSITY of HAWAI'I at MĀNOA

AlgoPARC

ICS 443: Parallel Algorithms

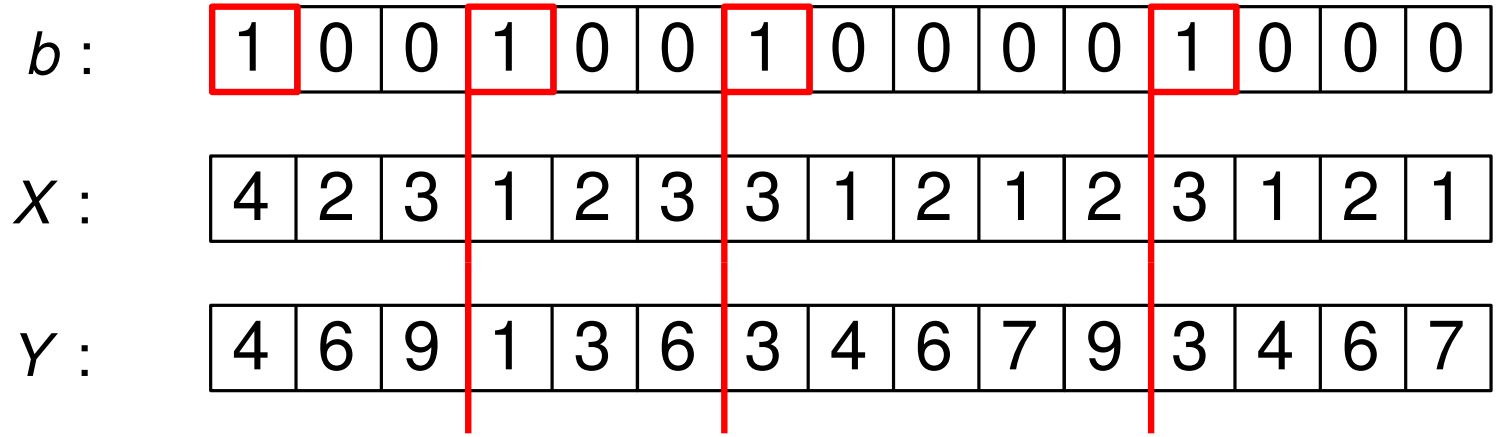
Prof. Nodari Sitchinava





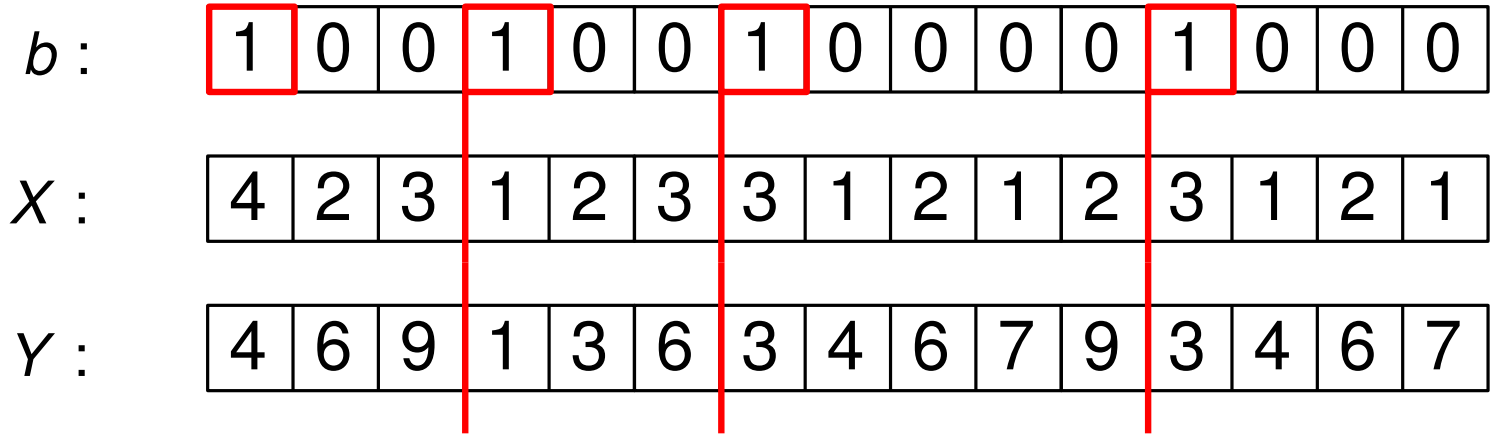
Lecture 7: Segmented Prefix Sums II

Reminder: Segmented Prefix Sums



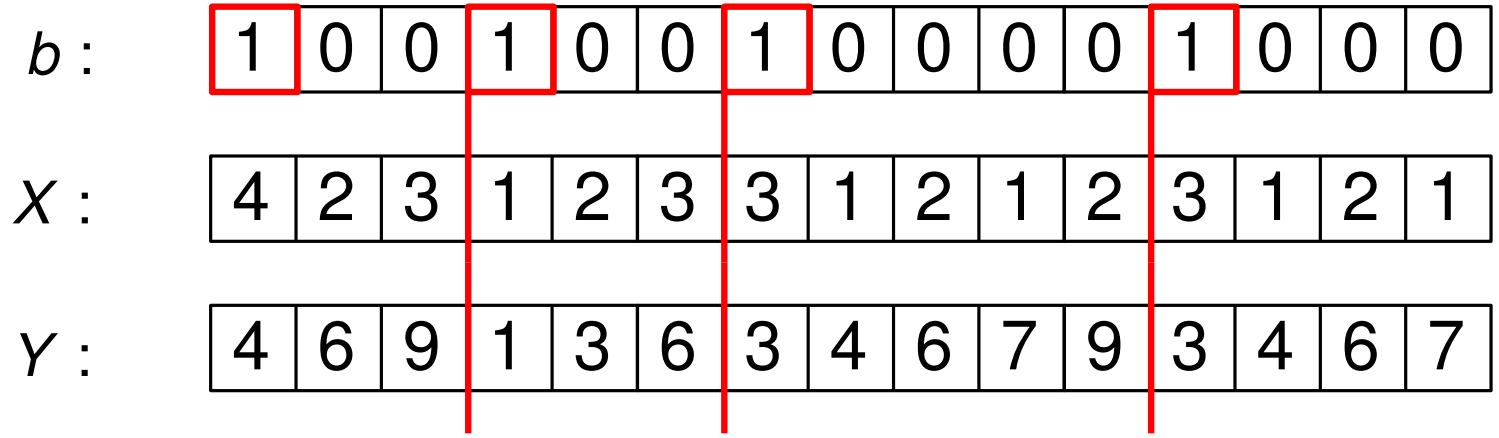
$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \cdot (1 - b_i)) + x_i & \text{if } i > 1 \end{cases}$$

Generalized Segmented Scan



$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

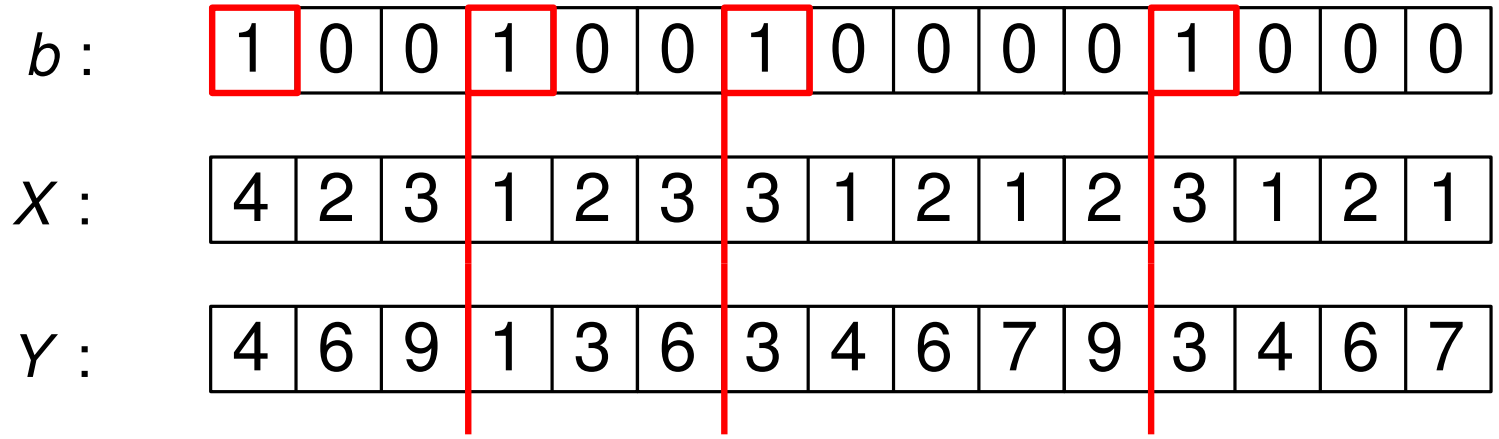
Generalized Segmented Scan



$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

Generalized Segmented Scan



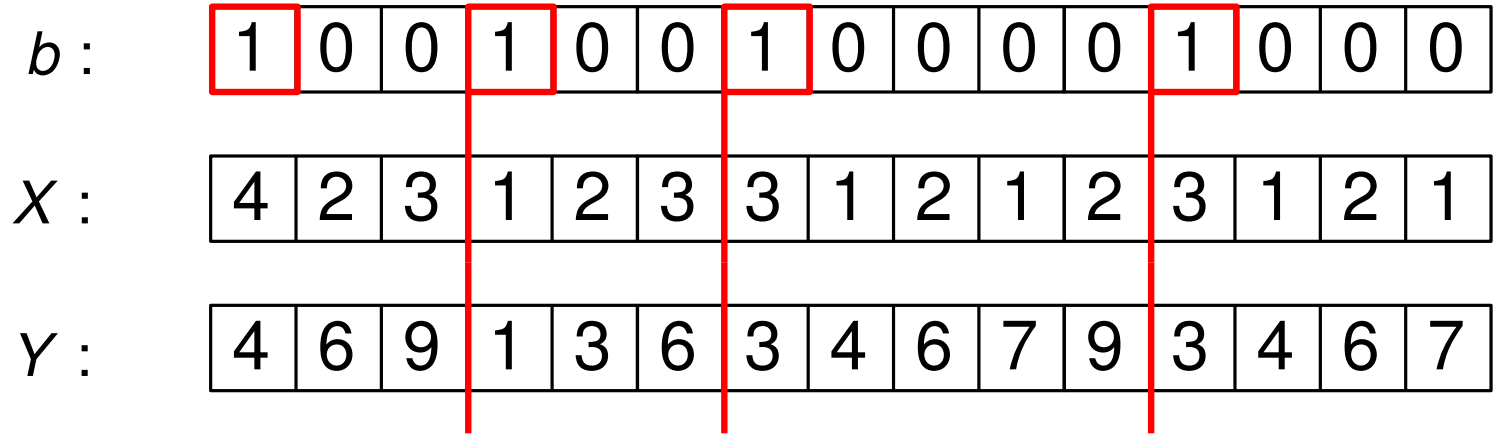
$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Generalized Segmented Scan



$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

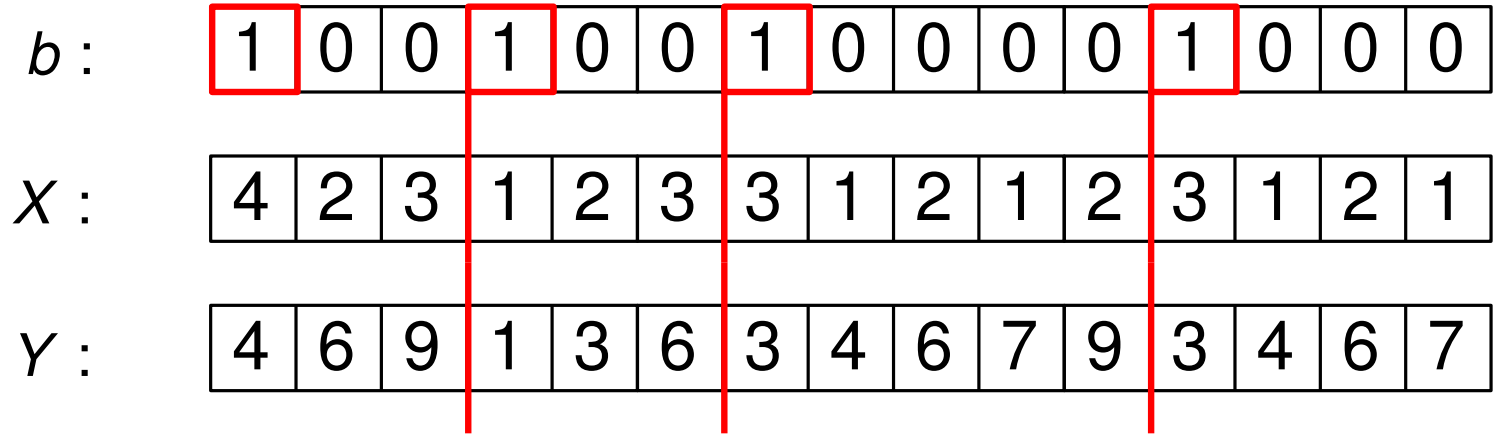
$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

Generalized Segmented Scan



$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

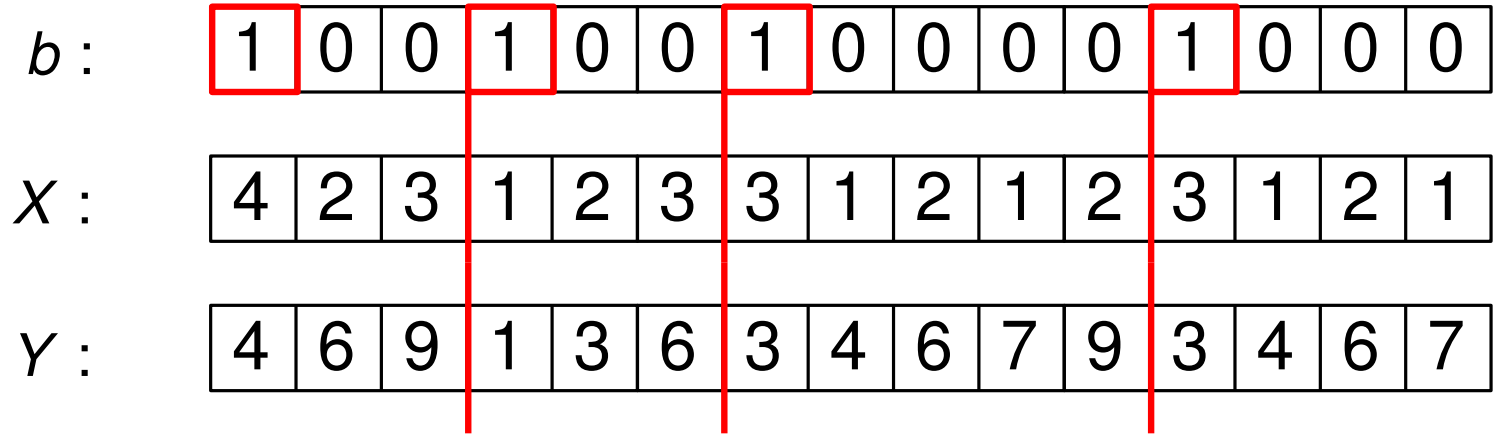
$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

Generalized Segmented Scan



$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

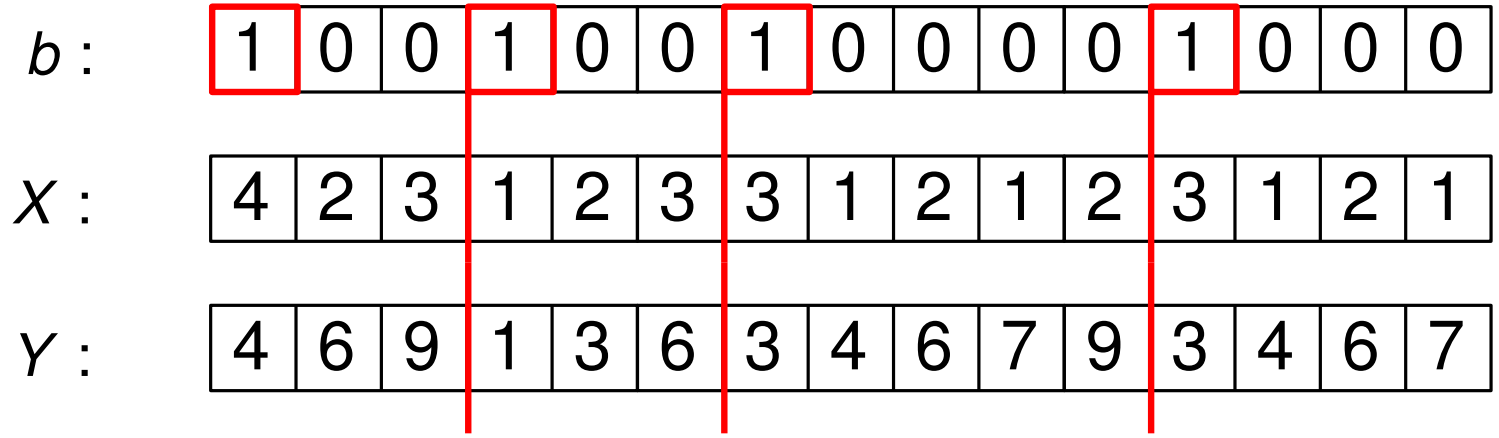
$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

Generalized Segmented Scan



$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ (y_{i-1} \otimes b_i) \oplus x_i & \text{if } i > 1 \end{cases}$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

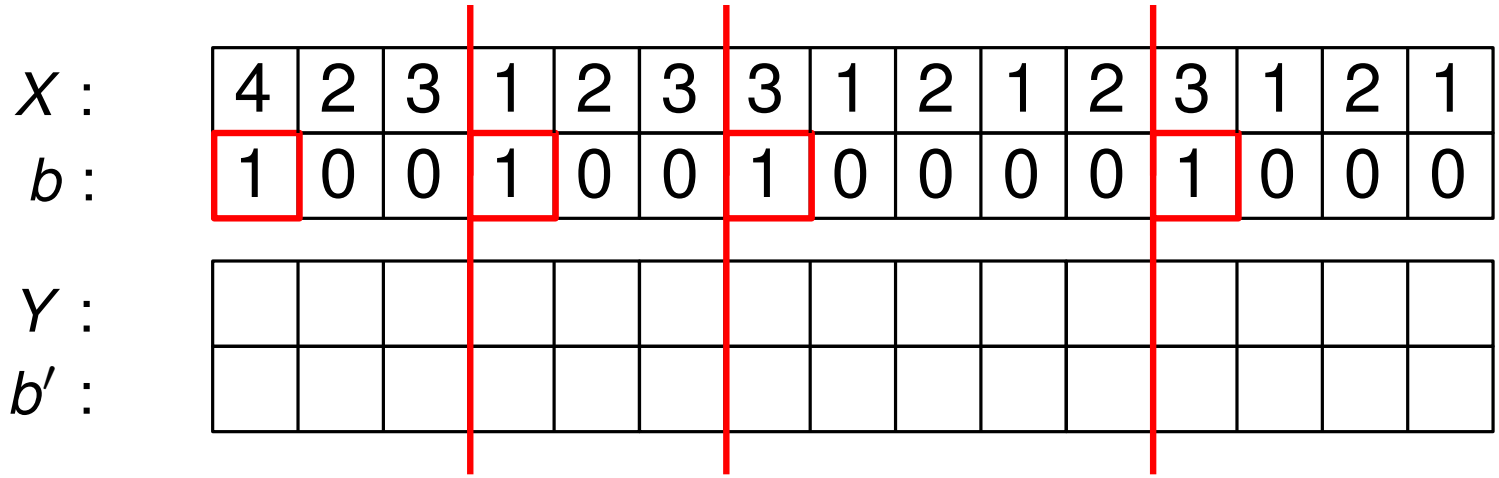
Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

Segmented Prefix Sums via $\bullet$



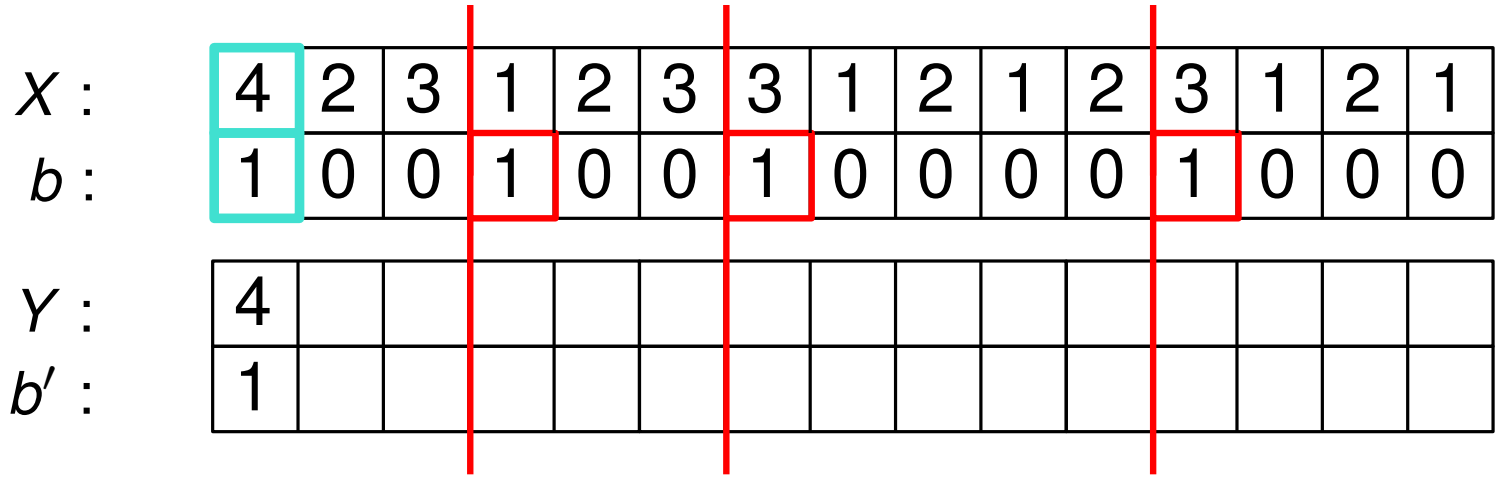
Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

$$\otimes : \mathbb{R} \times \{0, 1\} \to \mathbb{R}$$

$$z \otimes b = \begin{cases} z & \text{if } b = 0 \\ I_{\oplus} & \text{if } b = 1 \end{cases}$$

Segmented Prefix Sums via $\bullet$



$$\begin{pmatrix} y_1 \\ b'_1 \end{pmatrix} = \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \end{pmatrix}$$

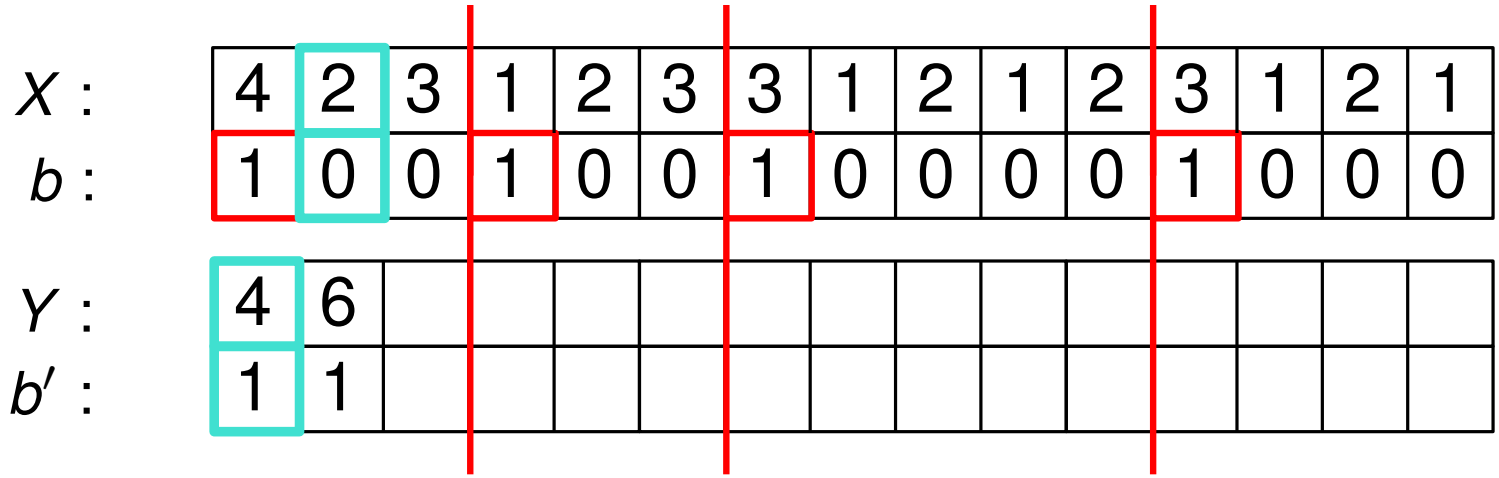
Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

$$\otimes : \mathbb{R} \times \{0, 1\} \to \mathbb{R}$$

$$z \otimes b = \begin{cases} z & \text{if } b = 0 \\ I_{\oplus} & \text{if } b = 1 \end{cases}$$

Segmented Prefix Sums via $\bullet$



$$\begin{pmatrix} y_2 \\ b'_2 \end{pmatrix} = \begin{pmatrix} (y_1 \otimes b_2) \oplus x_2 \\ b'_1 \vee b_2 \end{pmatrix} = \begin{pmatrix} (4 \otimes 0) \oplus 2 \\ 1 \vee 0 \end{pmatrix}$$

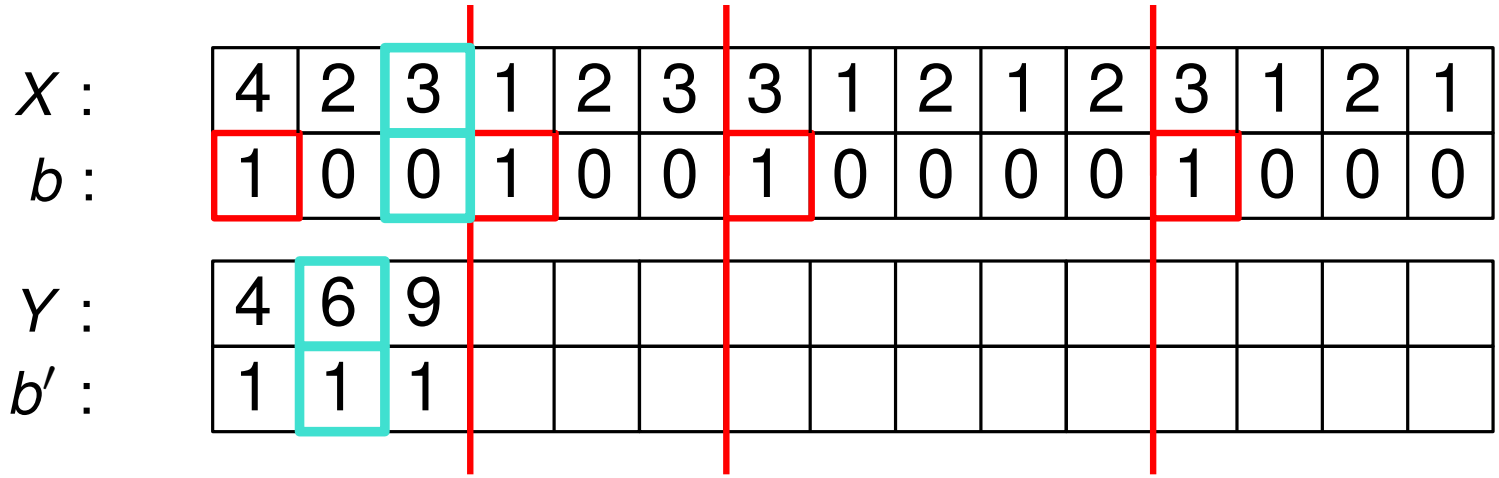
Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

$$\otimes : \mathbb{R} \times \{0, 1\} \to \mathbb{R}$$

$$z \otimes b = \begin{cases} z & \text{if } b = 0 \\ l_\oplus & \text{if } b = 1 \end{cases}$$

Segmented Prefix Sums via $\bullet$



$$\begin{pmatrix} y_3 \\ b'_3 \end{pmatrix} = \begin{pmatrix} (y_2 \otimes b_3) \oplus x_3 \\ b'_2 \vee b_3 \end{pmatrix} = \begin{pmatrix} (6 \otimes 0) \oplus 3 \\ 1 \vee 0 \end{pmatrix}$$

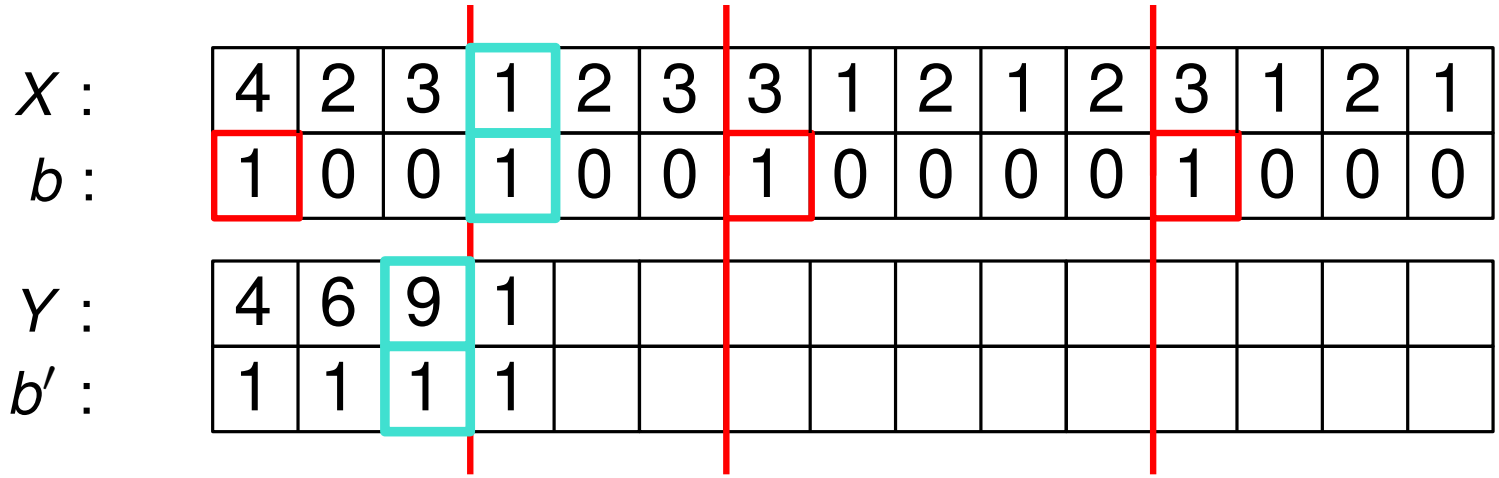
Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

$$\otimes : \mathbb{R} \times \{0, 1\} \to \mathbb{R}$$

$$z \otimes b = \begin{cases} z & \text{if } b = 0 \\ I_\oplus & \text{if } b = 1 \end{cases}$$

Segmented Prefix Sums via $\bullet$



$$\begin{pmatrix} y_4 \\ b'_4 \end{pmatrix} = \begin{pmatrix} (y_3 \otimes b_4) \oplus x_4 \\ b'_3 \vee b_4 \end{pmatrix} = \begin{pmatrix} (9 \otimes 1) \oplus 1 \\ 1 \vee 1 \end{pmatrix}$$

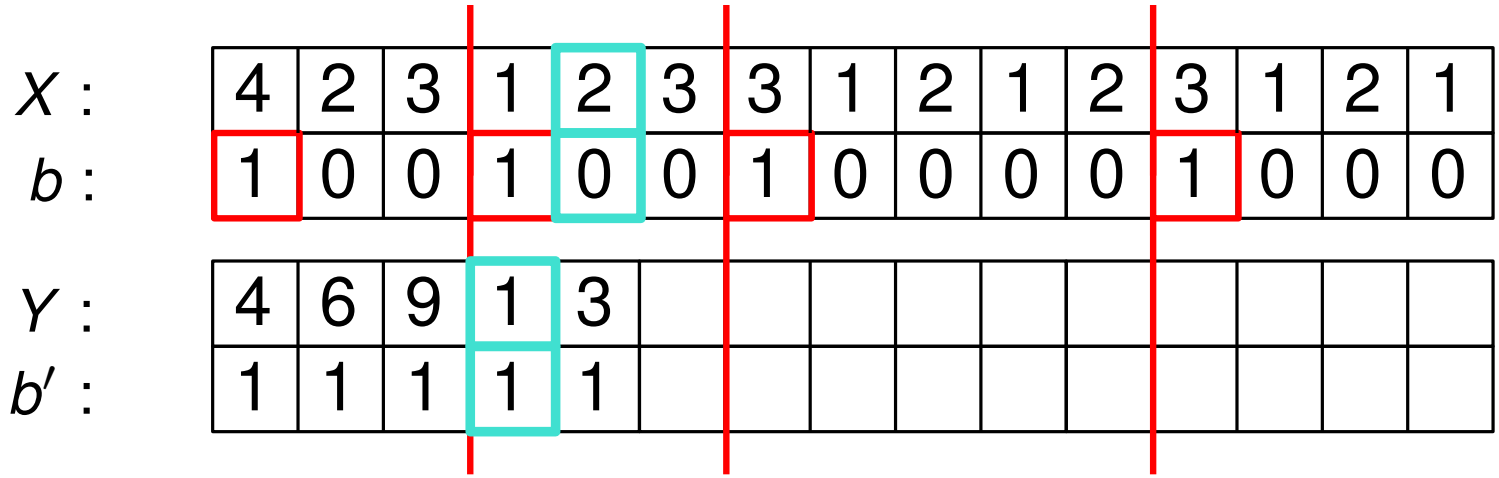
Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

$$\otimes : \mathbb{R} \times \{0, 1\} \to \mathbb{R}$$

$$z \otimes b = \begin{cases} z & \text{if } b = 0 \\ I_\oplus & \text{if } b = 1 \end{cases}$$

Segmented Prefix Sums via $\bullet$



$$\begin{pmatrix} y_5 \\ b'_5 \end{pmatrix} = \begin{pmatrix} (y_4 \otimes b_5) \oplus x_5 \\ b'_4 \vee b_5 \end{pmatrix} = \begin{pmatrix} (1 \otimes 0) \oplus 2 \\ 1 \vee 0 \end{pmatrix}$$

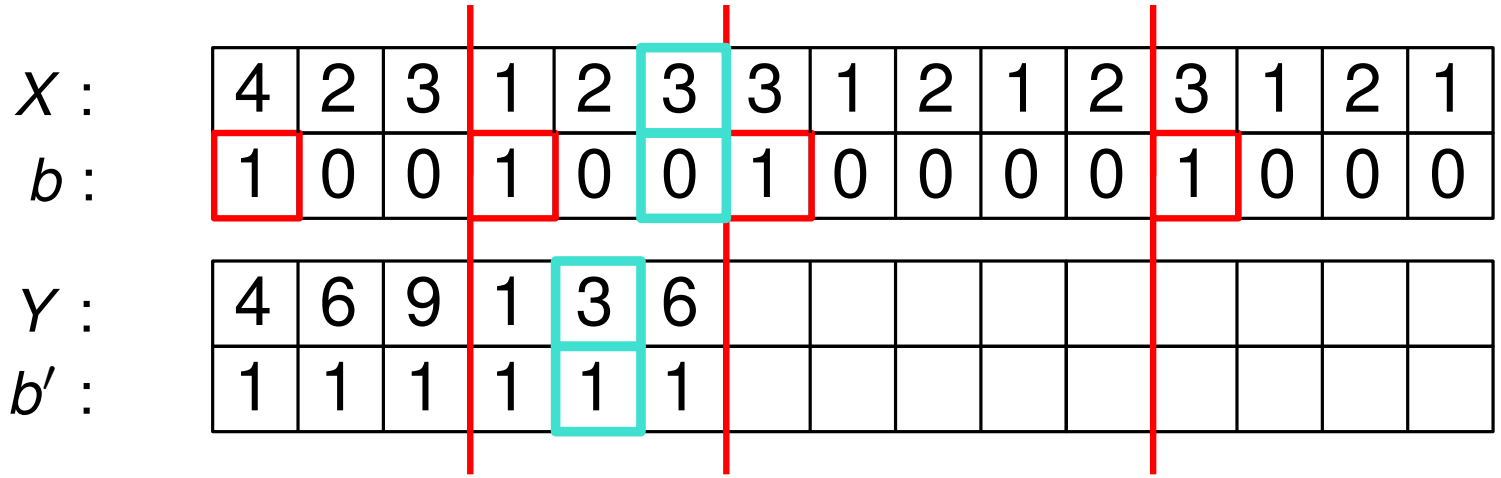
Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

$$\otimes : \mathbb{R} \times \{0, 1\} \to \mathbb{R}$$

$$z \otimes b = \begin{cases} z & \text{if } b = 0 \\ l_\oplus & \text{if } b = 1 \end{cases}$$

Segmented Prefix Sums via $\bullet$



$$\begin{pmatrix} y_6 \\ b'_6 \end{pmatrix} = \begin{pmatrix} (y_5 \otimes b_6) \oplus x_6 \\ b'_5 \vee b_6 \end{pmatrix} = \begin{pmatrix} (3 \otimes 0) \oplus 3 \\ 1 \vee 0 \end{pmatrix}$$

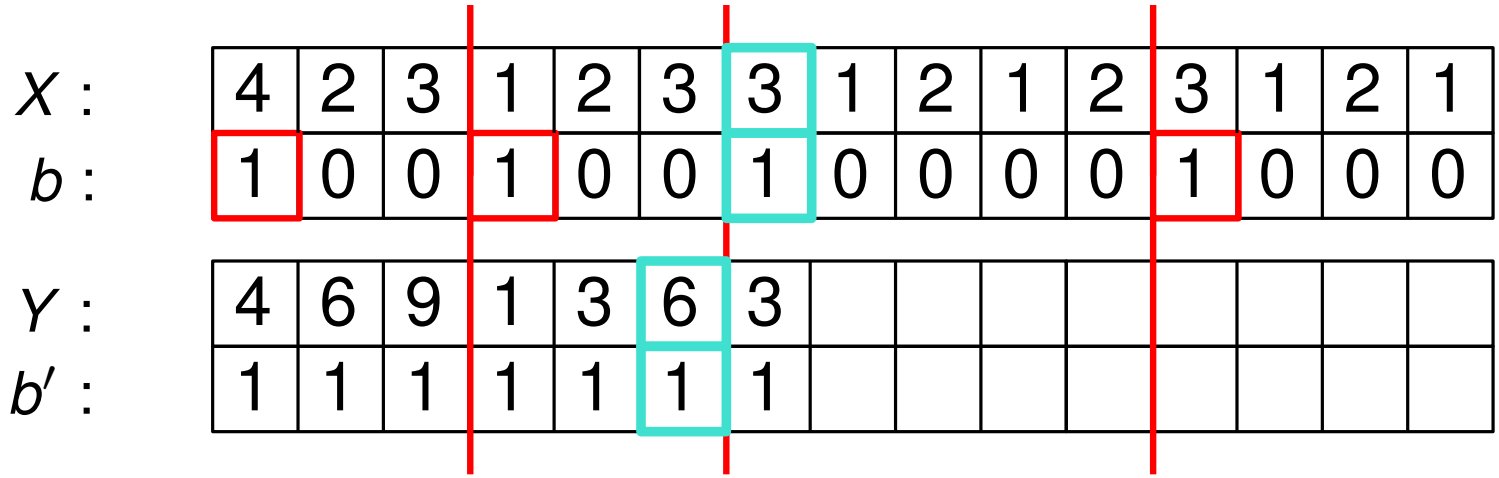
Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

$$\otimes : \mathbb{R} \times \{0, 1\} \to \mathbb{R}$$

$$z \otimes b = \begin{cases} z & \text{if } b = 0 \\ I_\oplus & \text{if } b = 1 \end{cases}$$

Segmented Prefix Sums via $\bullet$



$$\begin{pmatrix} y_7 \\ b'_7 \end{pmatrix} = \begin{pmatrix} (y_6 \otimes b_7) \oplus x_7 \\ b'_6 \vee b_7 \end{pmatrix} = \begin{pmatrix} (6 \otimes 1) \oplus 3 \\ 1 \vee 1 \end{pmatrix}$$

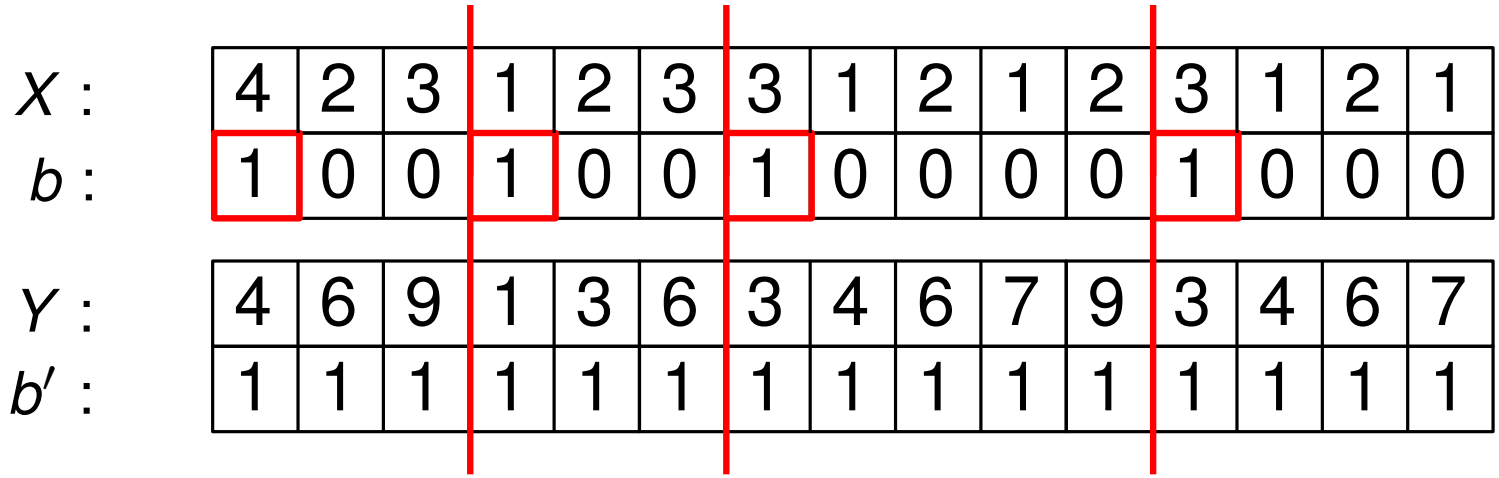
Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

$$\otimes : \mathbb{R} \times \{0, 1\} \to \mathbb{R}$$

$$z \otimes b = \begin{cases} z & \text{if } b = 0 \\ I_\oplus & \text{if } b = 1 \end{cases}$$

Segmented Prefix Sums via $\bullet$



Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

$$\otimes : \mathbb{R} \times \{0, 1\} \to \mathbb{R}$$

$$z \otimes b = \begin{cases} z & \text{if } b = 0 \\ l_\oplus & \text{if } b = 1 \end{cases}$$

Segmented Prefix Sums via $\bullet$

X :	4	2	3	1	2	3	3	1	2	1	2	3	1	2	1
b :	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0
Y :	4	6	9	1	3	6	3	4	6	7	9	3	4	6	7
b' :	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} = \begin{pmatrix} (y_{i-1} \otimes b_i) \oplus x_i \\ b'_{i-1} \vee b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

$$\otimes : \mathbb{R} \times \{0, 1\} \to \mathbb{R}$$

$$z \otimes b = \begin{cases} z & \text{if } b = 0 \\ l_\oplus & \text{if } b = 1 \end{cases}$$

Claim. $\bullet$ *is associative*

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

procedure PREFIX-SUMS($x[1..n]$)
  if $n \leq 1$ **then return**
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x'[i] = x[2i-1] + x[2i]$
  PREFIX-SUMS($x'[1..\frac{n}{2}]$)
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x[2i] = x'[i]$
    if $i \neq \frac{n}{2}$ **then**
      $x[2i+1] = x[2i+1] + x'[i]$

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

procedure SCAN($x[1..n], \oplus$)
  if $n \leq 1$ **then return**
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x'[i] = x[2i-1] \oplus x[2i]$
  SCAN($x'[1..\frac{n}{2}]$)
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x[2i] = x'[i]$
    if $i \neq \frac{n}{2}$ **then**
      $x[2i+1] = x[2i+1] \oplus x'[i]$

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

procedure SCAN($x[1..n], \oplus$)
  if $n \leq 1$ **then return**
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x'[i] = x[2i-1] \oplus x[2i]$
  SCAN($x'[1..\frac{n}{2}]$)
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x[2i] = x'[i]$
    if $i \neq \frac{n}{2}$ **then**
      $x[2i+1] = x[2i+1] \oplus x'[i]$

procedure SEG-SCAN($x[1..n], b[1..n], \oplus$)

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

class PAIR
  double first
  bit second

procedure SCAN($x[1..n], \oplus$)
  if $n \leq 1$ **then return**
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x'[i] = x[2i-1] \oplus x[2i]$
  SCAN($x'[1..\frac{n}{2}]$)
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x[2i] = x'[i]$
    if $i \neq \frac{n}{2}$ **then**
      $x[2i+1] = x[2i+1] \oplus x'[i]$

procedure SEG-SCAN($x[1..n], b[1..n], \oplus$)

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

class PAIR
  double first
  bit second

procedure SCAN($x[1..n]$, $\oplus$)
  if $n \leq 1$ **then return**
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x'[i] = x[2i-1] \oplus x[2i]$
  SCAN($x'[1..\frac{n}{2}]$)
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x[2i] = x'[i]$
    if $i \neq \frac{n}{2}$ **then**
      $x[2i+1] = x[2i+1] \oplus x'[i]$

procedure SEG-SCAN($x[1..n]$, $b[1..n]$, $\oplus$)
  $X[1..n]$ = new array of PAIRS
  for $i = 1$ to n **in parallel do**
    $X[i].first = x[i]$
    $X[i].second = b[i]$

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\binom{y_i}{b'_i} = \begin{cases} \binom{x_1}{b_1} & \text{if } i = 1 \\ \binom{y_{i-1}}{b'_{i-1}} \bullet \binom{x_i}{b_i} & \text{if } i > 1 \end{cases}$$

```
class PAIR
    double first
    bit second
```

```
procedure SCAN(x[1..n], ⊕)
    if n ≤ 1 then return
    for i = 1 to n/2 in parallel do
        x'[i] = x[2i − 1] ⊕ x[2i]
    SCAN(x'[1..n/2])
    for i = 1 to n/2 in parallel do
        x[2i] = x'[i]
        if i ≠ n/2 then
            x[2i + 1] = x[2i + 1] ⊕ x'[i]
```

```
procedure SEG-SCAN(x[1..n], b[1..n], ⊕)
    X[1..n] = new array of PAIRs
    for i = 1 to n in parallel do
        X[i].first = x[i]
        X[i].second = b[i]
    operator •(PAIR, PAIR, ⊕)
```

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\binom{y_i}{b'_i} = \begin{cases} \binom{x_1}{b_1} & \text{if } i = 1 \\ \binom{y_{i-1}}{b'_{i-1}} \bullet \binom{x_i}{b_i} & \text{if } i > 1 \end{cases}$$

class PAIR
  double first
  bit second

procedure SCAN($x[1..n]$, $\oplus$)
  if $n \leq 1$ **then return**
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x'[i] = x[2i-1] \oplus x[2i]$
  SCAN($x'[1..\frac{n}{2}]$)
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x[2i] = x'[i]$
    if $i \neq \frac{n}{2}$ **then**
      $x[2i+1] = x[2i+1] \oplus x'[i]$

procedure SEG-SCAN($x[1..n]$, $b[1..n]$, $\oplus$)
  $X[1..n]$ = new array of PAIRs
  for $i = 1$ to n **in parallel do**
    $X[i].first = x[i]$
    $X[i].second = b[i]$
  operator $\bullet$(PAIR, PAIR, $\oplus$)
  SCAN($X[1..n]$, $\bullet$)

Segmented scan via prefix sums

Scan

$$y_i = \begin{cases} x_1 & \text{if } i = 1 \\ y_{i-1} \oplus x_i & \text{if } i > 1 \end{cases}$$

Segmented scan

$$\begin{pmatrix} y_i \\ b'_i \end{pmatrix} = \begin{cases} \begin{pmatrix} x_1 \\ b_1 \end{pmatrix} & \text{if } i = 1 \\ \begin{pmatrix} y_{i-1} \\ b'_{i-1} \end{pmatrix} \bullet \begin{pmatrix} x_i \\ b_i \end{pmatrix} & \text{if } i > 1 \end{cases}$$

class PAIR
  double first
  bit second

procedure SCAN($x[1..n]$, $\oplus$)
  if $n \leq 1$ **then return**
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x'[i] = x[2i-1] \oplus x[2i]$
  SCAN($x'[1..\frac{n}{2}]$)
  for $i = 1$ to $\frac{n}{2}$ **in parallel do**
    $x[2i] = x'[i]$
    if $i \neq \frac{n}{2}$ **then**
      $x[2i+1] = x[2i+1] \oplus x'[i]$

procedure SEG-SCAN($x[1..n]$, $b[1..n]$, $\oplus$)
  $X[1..n]$ = new array of PAIRs
  for $i = 1$ to n **in parallel do**
    $X[i].first = x[i]$
    $X[i].second = b[i]$
  operator $\bullet$(PAIR, PAIR, $\oplus$)
  SCAN($X[1..n]$, $\bullet$)
  for $i = 1$ to n **in parallel do**
    $x[i] = X[i].first$

Defining $\bullet$

$$\bullet : (\mathbb{R} \times \{0,1\}) \times (\mathbb{R} \times \{0,1\}) \to (\mathbb{R} \times \{0,1\})$$

$$\binom{x}{b_1} \bullet \binom{y}{b_2} = \binom{(x \otimes b_2) \oplus y}{b_1 \vee b_2}$$

operator $\bullet$(PAIR x, PAIR y, **operator** $\oplus$)

$$\otimes : \mathbb{R} \times \{0,1\} \to \mathbb{R}$$

$$x \otimes b = \begin{cases} x & \text{if } b = 0 \\ I_{\oplus} & \text{if } b = 1 \end{cases}$$

Defining $\bullet$

$$\bullet : (\mathbb{R} \times \{0,1\}) \times (\mathbb{R} \times \{0,1\}) \to (\mathbb{R} \times \{0,1\})$$

$$\binom{x}{b_1} \bullet \binom{y}{b_2} = \binom{(x \otimes b_2) \oplus y}{b_1 \vee b_2}$$

operator $\bullet$(PAIR x, PAIR y, **operator** $\oplus$)
    z = **new** PAIR

$$\otimes : \mathbb{R} \times \{0,1\} \to \mathbb{R}$$

$$x \otimes b = \begin{cases} x & \text{if } b = 0 \\ I_{\oplus} & \text{if } b = 1 \end{cases}$$

Defining $\bullet$

$$\bullet : (\mathbb{R} \times \{0,1\}) \times (\mathbb{R} \times \{0,1\}) \to (\mathbb{R} \times \{0,1\})$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

operator $\bullet$(PAIR x, PAIR y, **operator** $\oplus$)
    z = **new** PAIR
    if $y.second = 0$ **then**
        $z.first = x.first \oplus y.first$
    else
        $z.first = I_{\oplus} \oplus y.first$

$$\otimes : \mathbb{R} \times \{0,1\} \to \mathbb{R}$$

$$x \otimes b = \begin{cases} x & \text{if } b = 0 \\ I_{\oplus} & \text{if } b = 1 \end{cases}$$

Defining $\bullet$

$$\bullet : (\mathbb{R} \times \{0,1\}) \times (\mathbb{R} \times \{0,1\}) \to (\mathbb{R} \times \{0,1\})$$

$$\begin{pmatrix} x \\ b_1 \end{pmatrix} \bullet \begin{pmatrix} y \\ b_2 \end{pmatrix} = \begin{pmatrix} (x \otimes b_2) \oplus y \\ b_1 \vee b_2 \end{pmatrix}$$

operator $\bullet$(PAIR x, PAIR y, **operator** $\oplus$)

z = **new** PAIR

if $y.second = 0$ **then**

$z.first = x.first \oplus y.first$

else

$z.first = I_{\oplus} \oplus y.first$

$z.second = x.second \vee y.second$

$$\otimes : \mathbb{R} \times \{0,1\} \to \mathbb{R}$$

$$x \otimes b = \begin{cases} x & \text{if } b = 0 \\ I_{\oplus} & \text{if } b = 1 \end{cases}$$

Defining $\bullet$

$$\bullet : (\mathbb{R} \times \{0,1\}) \times (\mathbb{R} \times \{0,1\}) \to (\mathbb{R} \times \{0,1\})$$

$$\binom{x}{b_1} \bullet \binom{y}{b_2} = \binom{(x \otimes b_2) \oplus y}{b_1 \vee b_2}$$

operator $\bullet$(PAIR x, PAIR y, **operator** $\oplus$)
    z = **new** PAIR
    if $y.second = 0$ **then**
        $z.first = x.first \oplus y.first$
    else
        $z.first = I_{\oplus} \oplus y.first$
    $z.second = x.second \vee y.second$
    return z

$$\otimes : \mathbb{R} \times \{0,1\} \to \mathbb{R}$$

$$x \otimes b = \begin{cases} x & \text{if } b = 0 \\ I_{\oplus} & \text{if } b = 1 \end{cases}$$

Defining Segmented Applications

- Define $\oplus : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$
  - Associative operator
  - Left identity $I_\oplus$

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$

$$I_\oplus \otimes x = x$$

Defining Segmented Applications

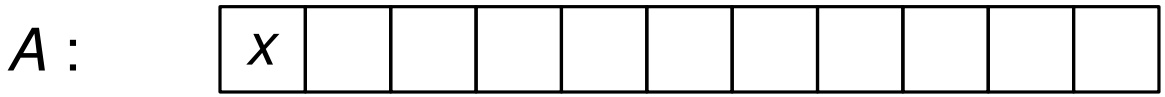
- Define $\oplus : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$
  - Associative operator
  - Left identity $I_\oplus$

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$

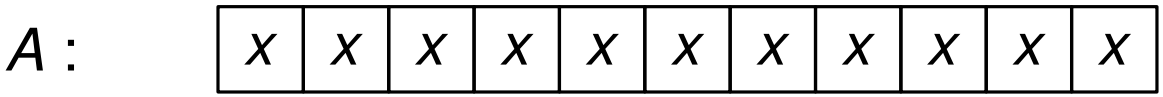
$$I_\oplus \otimes x = x$$

- Run SEG-SCAN with operator $\oplus$

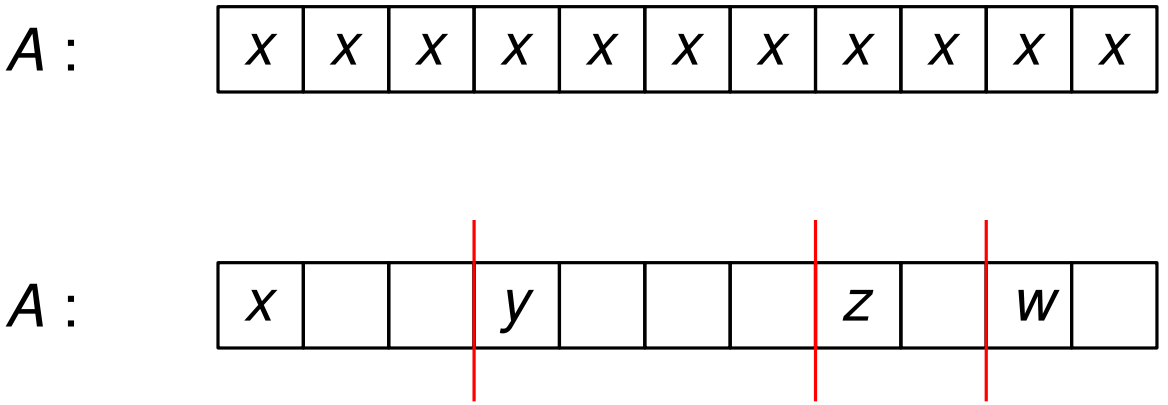
Segmented BROADCAST



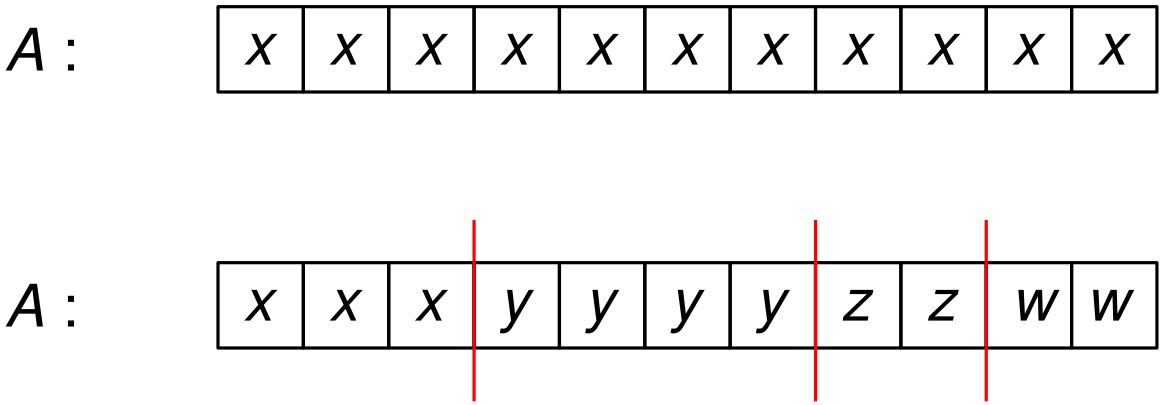
Segmented BROADCAST



Segmented BROADCAST



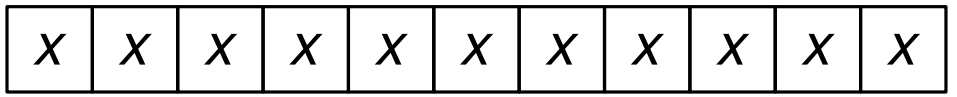
Segmented BROADCAST



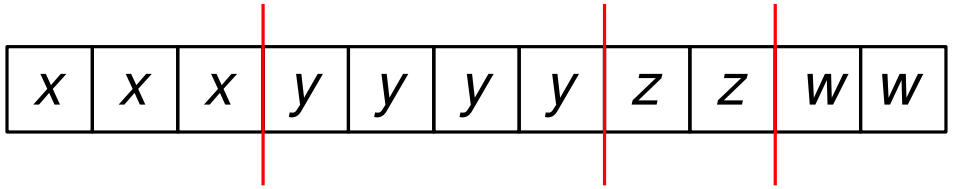
Segmented BROADCAST

BROADCAST($A[1..n]$)

A :



A :

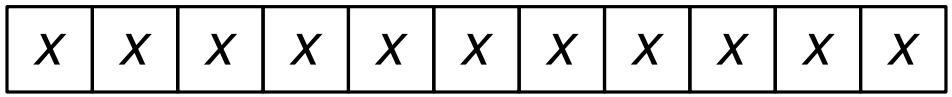


Segmented BROADCAST

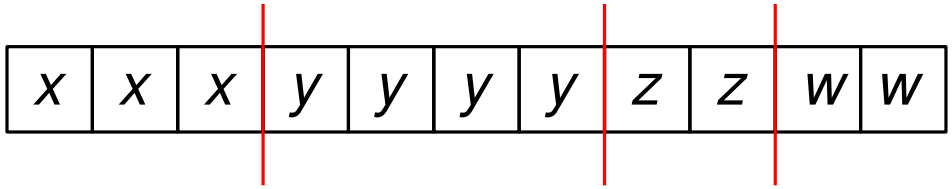
BROADCAST($A[1..n]$)
    SCAN($A[1..n]$, COPY)

operator COPY(x, y)
    return x

A :



A :

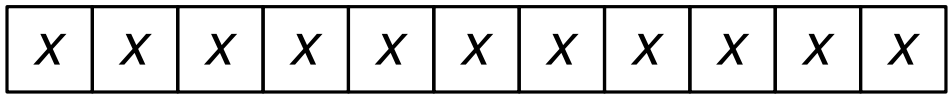


Segmented BROADCAST

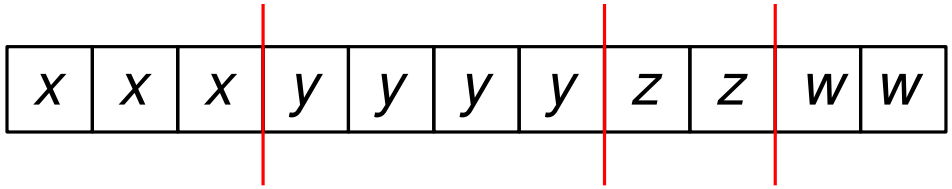
BROADCAST($A[1..n]$)
    SCAN($A[1..n]$, COPY)

operator COPY(x, y)
    return x

A :



A :

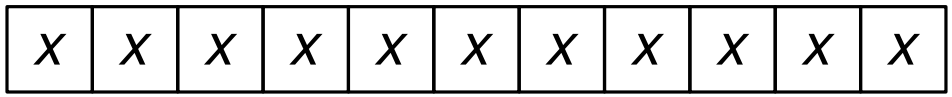


- Define $\oplus : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$
  - Associative operator
  - Left identity $I_\oplus$

Segmented BROADCAST

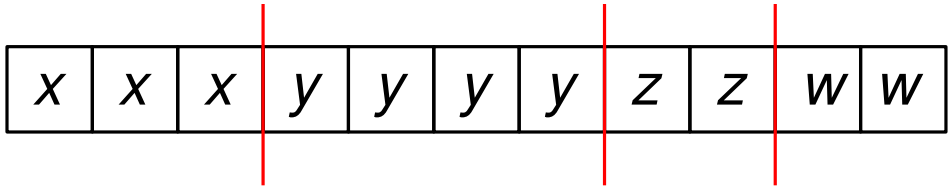
BROADCAST($A[1..n]$)
    SCAN($A[1..n]$, COPY)

A :



operator COPY(x, y)
    return x

A :

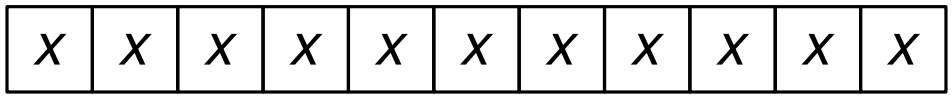


- Define COPY : $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
  - Associative operator ✓ COPY(a, COPY(b, c)) = COPY(COPY(a, b), c)
  - Left identity $I_{\oplus}$

Segmented BROADCAST

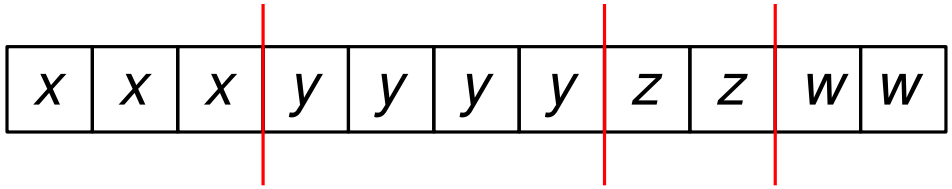
BROADCAST($A[1..n]$)
    SCAN($A[1..n]$, COPY)

A :



operator COPY(x, y)
    return x

A :



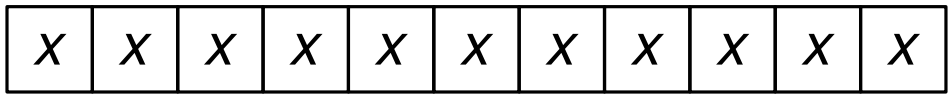
- Define COPY : $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
  - Associative operator ✓ COPY(a, COPY(b, c)) = COPY(COPY(a, b), c)
  - Left identity $I_{\oplus}$ ✗ COPY(??, x) = x

Segmented BROADCAST

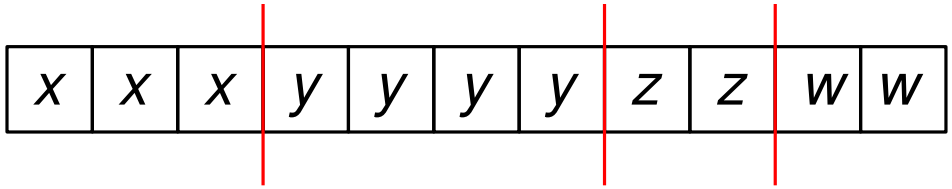
```
BROADCAST(A[1..n])
    SCAN(A[1..n], COPY)
```

```
operator COPY'(x, y)
    if x = ⊥ then
        return y
    else
        return x
```

A :



A :



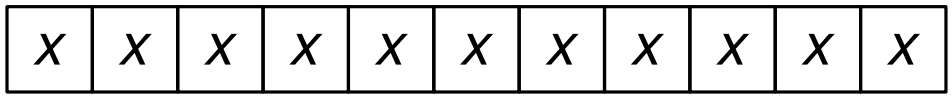
- Define COPY' : $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
  - Associative operator
  - Left identity $I_{\oplus}$

Segmented BROADCAST

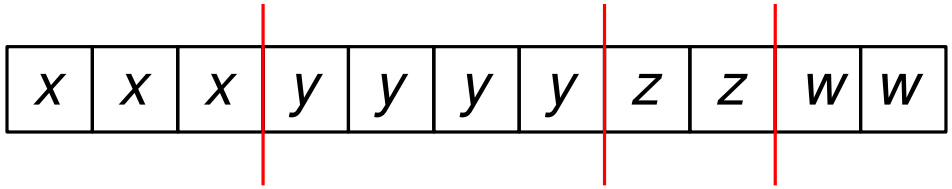
```
BROADCAST(A[1..n])
    SCAN(A[1..n], COPY)
```

```
operator COPY'(x, y)
    if x = ⊥ then
        return y
    else
        return x
```

A :



A :



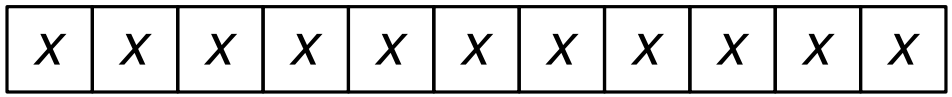
- Define COPY' : $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
  - Associative operator ✓ COPY(a, COPY(b, c)) = COPY(COPY(a, b), c)
  - Left identity $I_{\oplus}$

Segmented BROADCAST

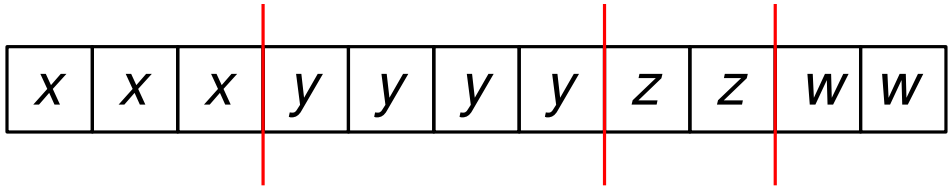
```
BROADCAST(A[1..n])
    SCAN(A[1..n], COPY)
```

```
operator COPY'(x, y)
    if x = ⊥ then
        return y
    else
        return x
```

A :



A :



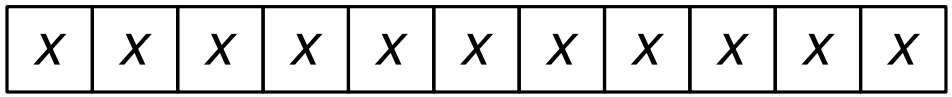
- Define COPY' : $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
  - Associative operator ✓ COPY(a, COPY(b, c)) = COPY(COPY(a, b), c)
  - Left identity $I_{\oplus}$ ✓ COPY($\perp$, x) = x

Segmented BROADCAST

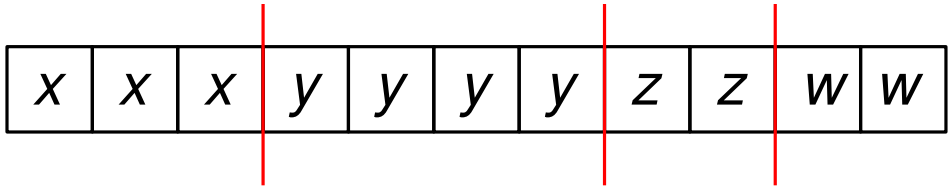
```
BROADCAST(A[1..n])
    SCAN(A[1..n], COPY)
```

```
operator COPY'(x, y)
    if x = ⊥ then
        return y
    else
        return x
```

A :



A :



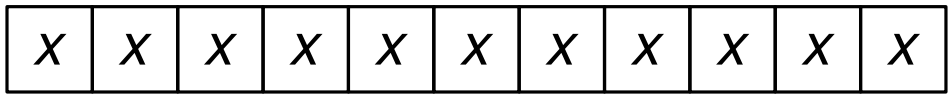
- Define COPY' : $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
  - Associative operator ✓ $\text{COPY}(a, \text{COPY}(b, c)) = \text{COPY}(\text{COPY}(a, b), c)$
  - Left identity $I_{\oplus} = \bot$ ✓ $\text{COPY}(\bot, x) = x$

Segmented BROADCAST

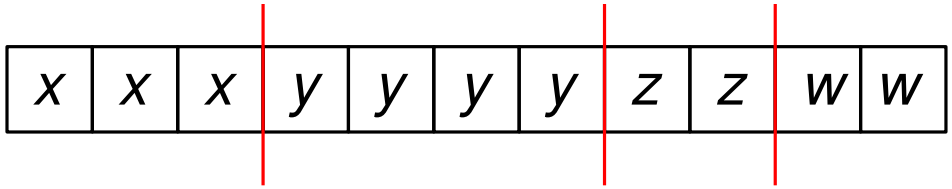
```
BROADCAST(A[1..n])
    SCAN(A[1..n], COPY)
```

```
operator COPY'(x, y)
    if x = ⊥ then
        return y
    else
        return x
```

A :



A :



- Define COPY' : $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
  - Associative operator ✓ COPY(a, COPY(b, c)) = COPY(COPY(a, b), c)
  - Left identity $I_{\oplus} = \perp$ ✓ COPY($\perp$, x) = x

```
SEG-BROADCAST(A[1..n], b[1..n])
    SEG-SCAN(A[1..n], b[1..n], COPY')
```

Segmented BROADCAST

```
BROADCAST(A[1..n])
    SCAN(A[1..n], COPY)
```

```
operator COPY'(x, y)
    if x = ⊥ then
        return y
    else
        return x
```

A :

x	x	x	x	x	x	x	x	x	x	x

A :

x	x	x	y	y	y	y	z	z	w	w

- Define COPY' : $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
  - Associative operator ✓ CO
  - Left identity $I_{\oplus} = \perp$ ✓ CO

```
SEG-BROADCAST(A[1..n], b[1..n])
    SEG-SCAN(A[1..n], b[1..n], COPY')
```

```
procedure SEG-SCAN(x[1..n], b[1..n], ⊕)
    X[1..n] = new array of PAIRs
    for i = 1 to n in parallel do
        X[i].first = x[i]
        X[i].second = b[i]
    operator •(PAIR, PAIR, ⊕)
    SCAN(X[1..n], •)
    for i = 1 to n in parallel do
        x[i] = X[i].first
```

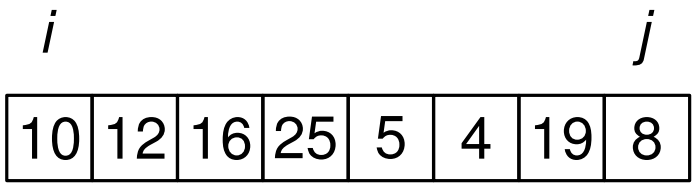
QUICKSORT($A[1..n]$)

QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[i..j])
  if i < j then
      pivot = RANDOM(i, j)
      SWAP(A[i], A[pivot])
      k = PARTITION(A[i..j])
      in parallel do
          QUICKSORT(A[i..k − 1])
          QUICKSORT(A[k + 1..j])
```

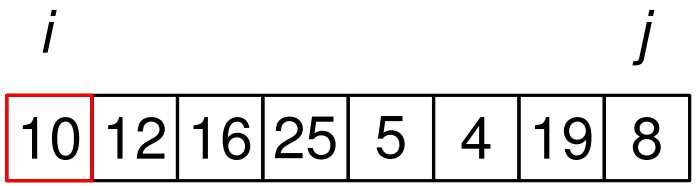
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[i..j])
  if i < j then
      pivot = RANDOM(i, j)
      SWAP(A[i], A[pivot])
      k = PARTITION(A[i..j])
      in parallel do
          QUICKSORT(A[i..k − 1])
          QUICKSORT(A[k + 1..j])
```



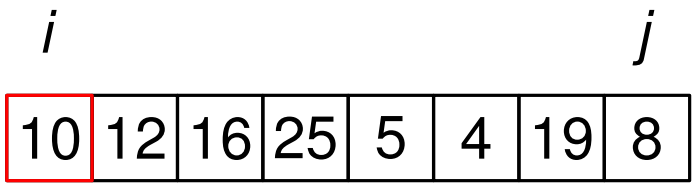
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[i..j])
  if i < j then
      pivot = RANDOM(i, j)
      SWAP(A[i], A[pivot])
      k = PARTITION(A[i..j])
      in parallel do
          QUICKSORT(A[i..k − 1])
          QUICKSORT(A[k + 1..j])
```



QUICKSORT($A[1..n]$)

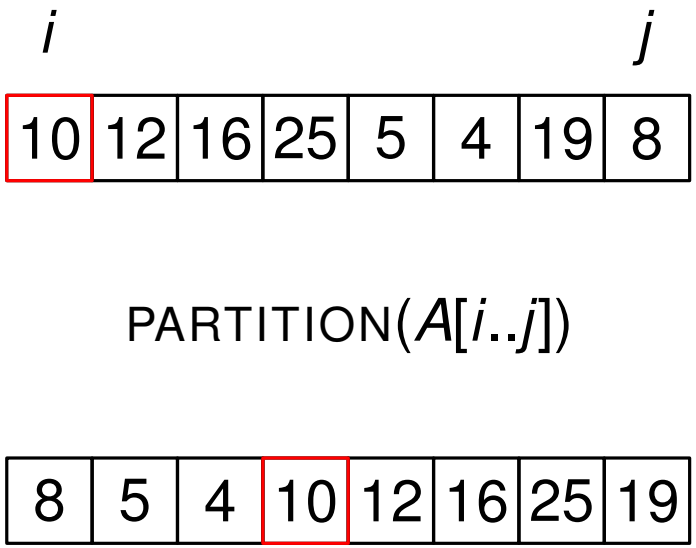
```
procedure QUICKSORT(A[i..j])
  if i < j then
      pivot = RANDOM(i, j)
      SWAP(A[i], A[pivot])
      k = PARTITION(A[i..j])
      in parallel do
          QUICKSORT(A[i..k − 1])
          QUICKSORT(A[k + 1..j])
```



PARTITION($A[i..j]$)

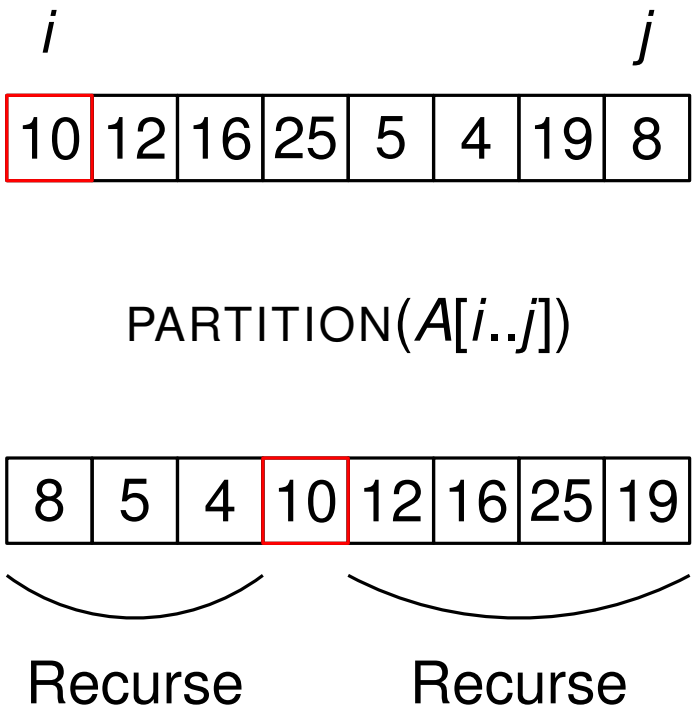
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[i..j])
  if i < j then
      pivot = RANDOM(i, j)
      SWAP(A[i], A[pivot])
      k = PARTITION(A[i..j])
      in parallel do
          QUICKSORT(A[i..k − 1])
          QUICKSORT(A[k + 1..j])
```



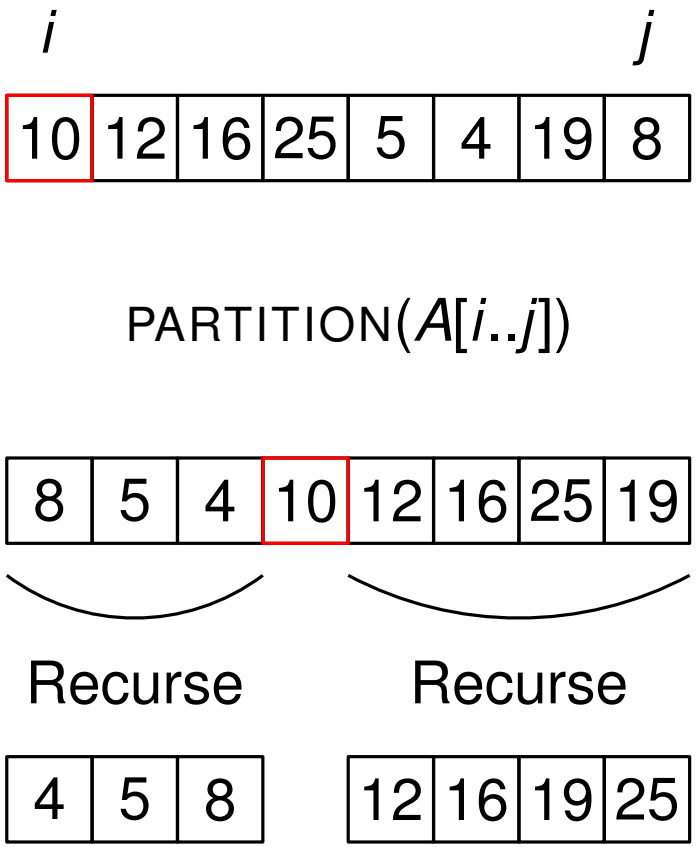
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[i..j])
  if i < j then
      pivot = RANDOM(i, j)
      SWAP(A[i], A[pivot])
      k = PARTITION(A[i..j])
      in parallel do
          QUICKSORT(A[i..k − 1])
          QUICKSORT(A[k + 1..j])
```



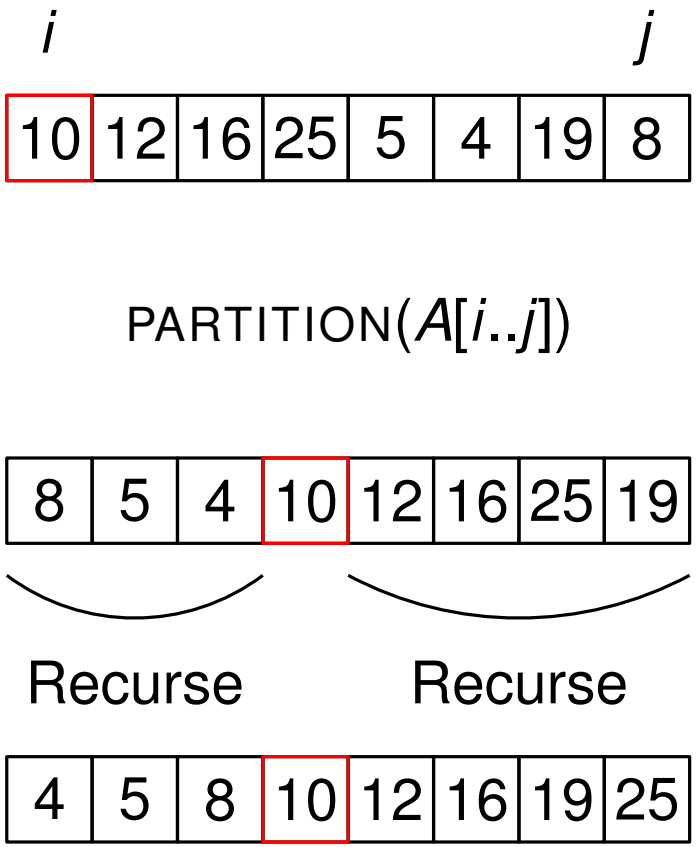
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[i..j])
  if i < j then
      pivot = RANDOM(i, j)
      SWAP(A[i], A[pivot])
      k = PARTITION(A[i..j])
      in parallel do
          QUICKSORT(A[i..k − 1])
          QUICKSORT(A[k + 1..j])
```



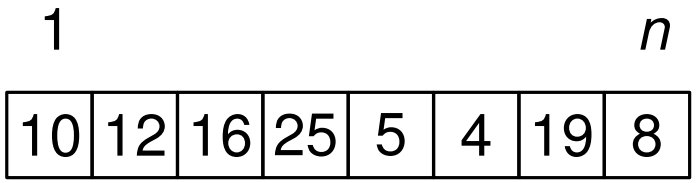
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[i..j])
  if i < j then
      pivot = RANDOM(i, j)
      SWAP(A[i], A[pivot])
      k = PARTITION(A[i..j])
      in parallel do
          QUICKSORT(A[i..k − 1])
          QUICKSORT(A[k + 1..j])
```



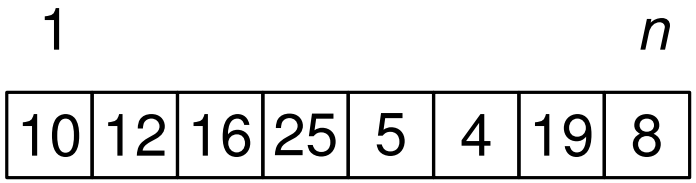
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```



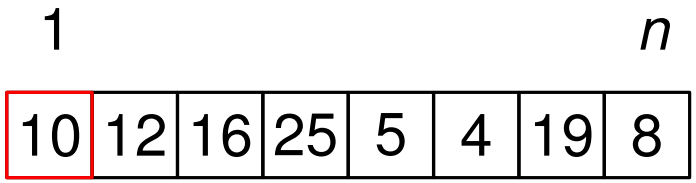
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[1..n])
    segs = new array of n bits
    for i = 2 to n in parallel do segs[i] = 0
    segs[1] = 1                          ▷ initialize segments
    while not SORTED(A[1..n]) do
        SEG-RANDOMIZE-PIVOTS(A, segs)
        p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
        UPDATE-SEGS(segs[1..n], p[1..n])
```



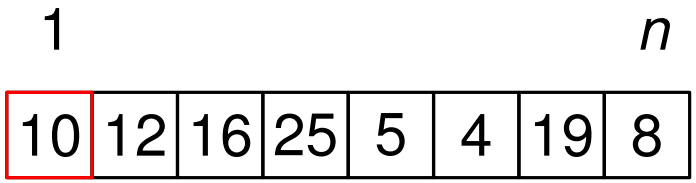
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[1..n])
    segs = new array of n bits
    for i = 2 to n in parallel do segs[i] = 0
    segs[1] = 1                          ▷ initialize segments
    while not SORTED(A[1..n]) do
        SEG-RANDOMIZE-PIVOTS(A, segs)
        p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
        UPDATE-SEGS(segs[1..n], p[1..n])
```



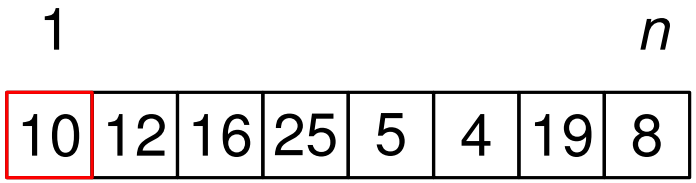
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```



QUICKSORT($A[1..n]$)

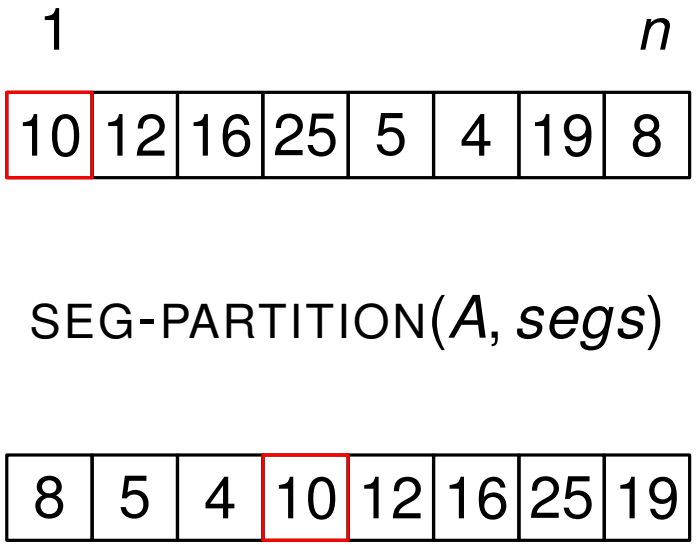
```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```



SEG-PARTITION(A, $segs$)

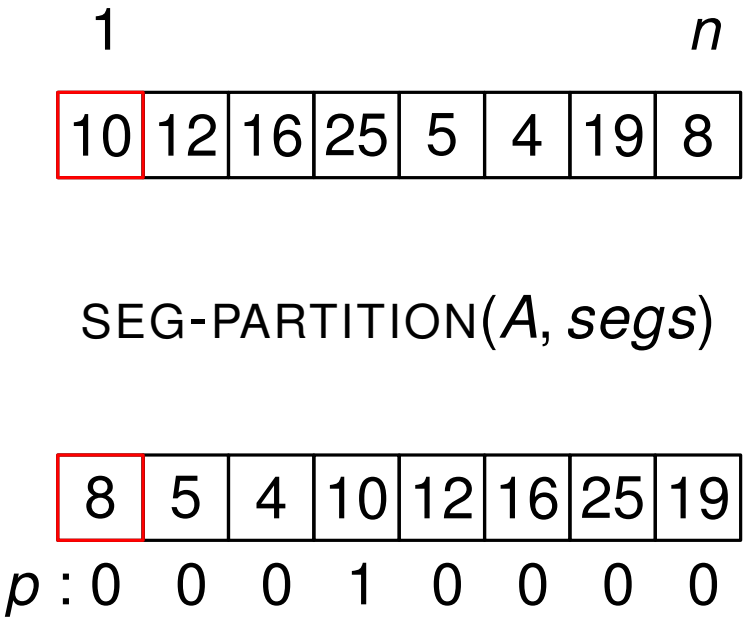
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```



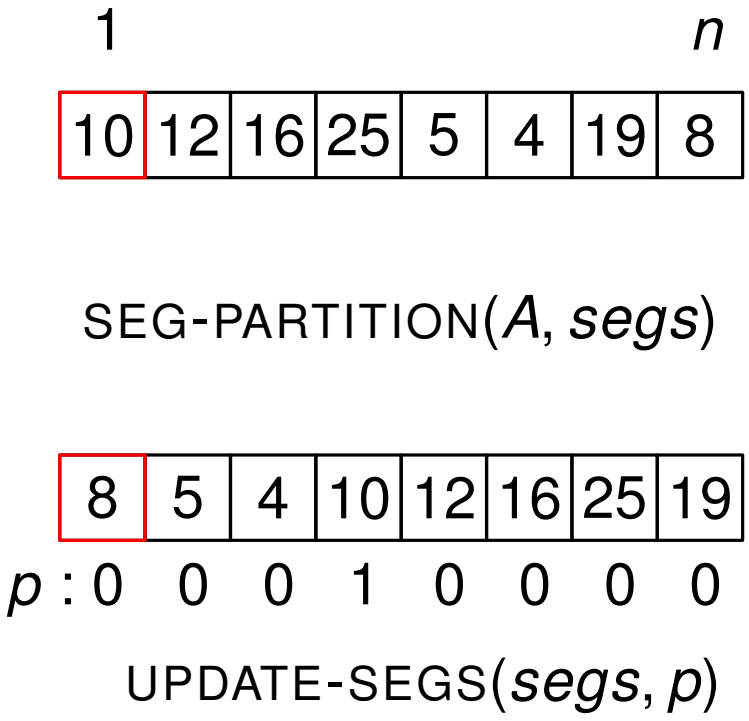
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```



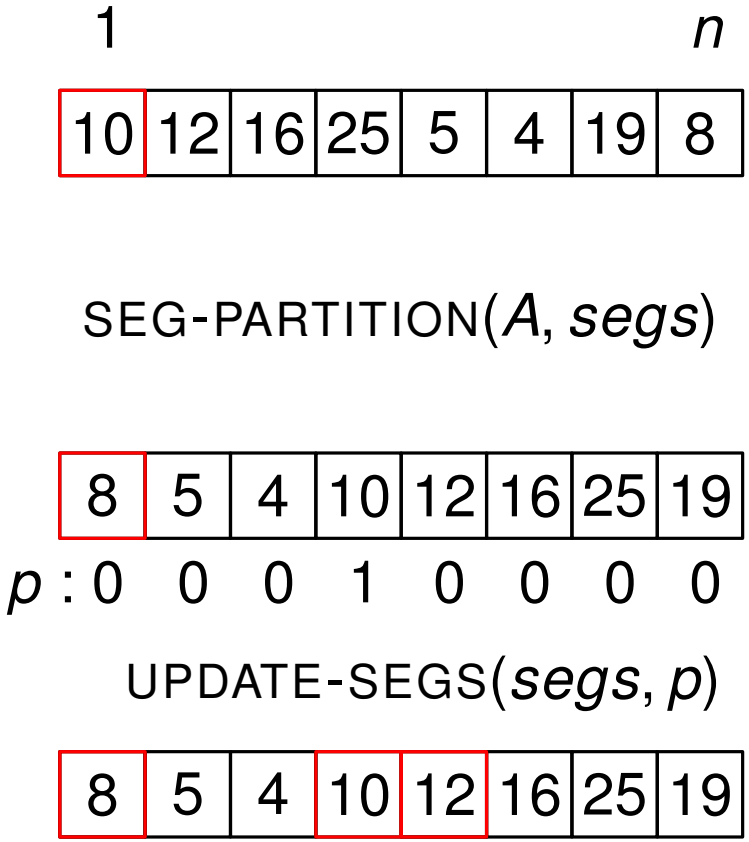
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```



QUICKSORT($A[1..n]$)

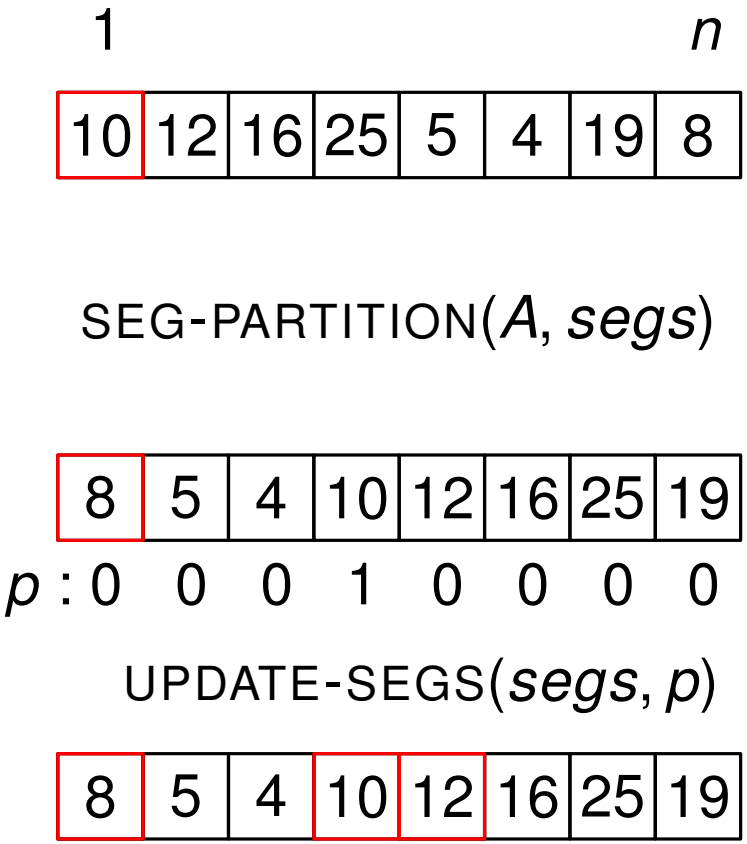
```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```



QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```

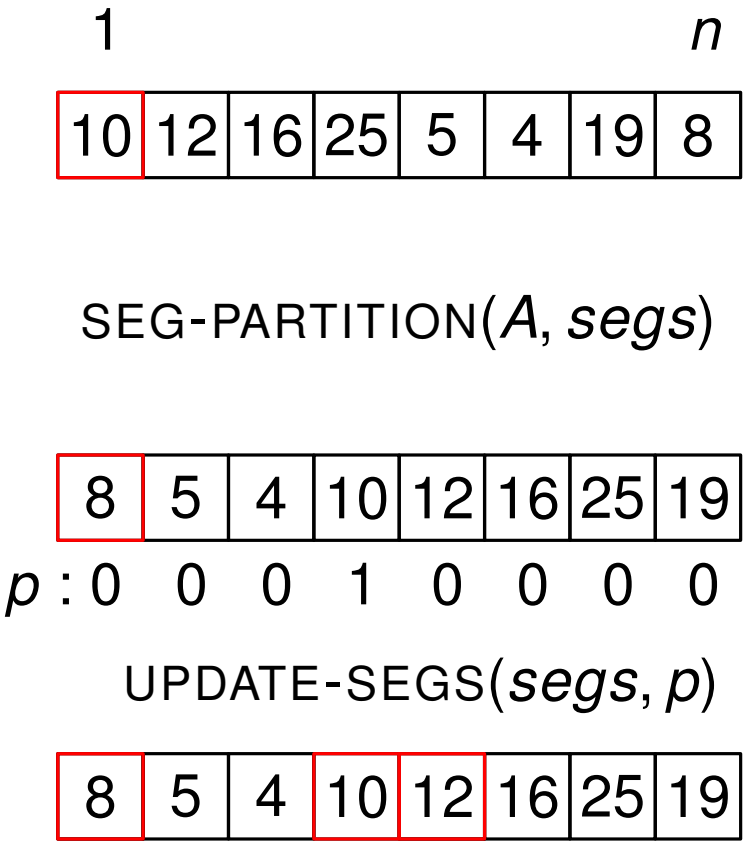
```
procedure UPDATE-SEGS(segs[1..n], p[1..n])
```



QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```

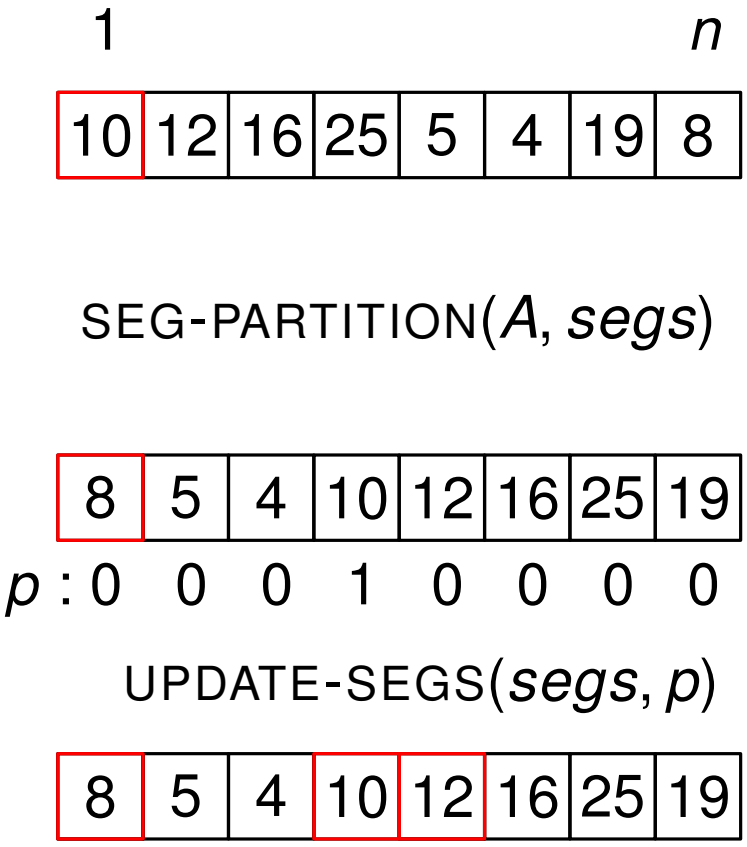
```
procedure UPDATE-SEGS(segs[1..n], p[1..n])
    for i = 1 to n in parallel do
        if p[i] = 1 then
            segs[i] = 1

                segs[i + 1] = 1
```



QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```

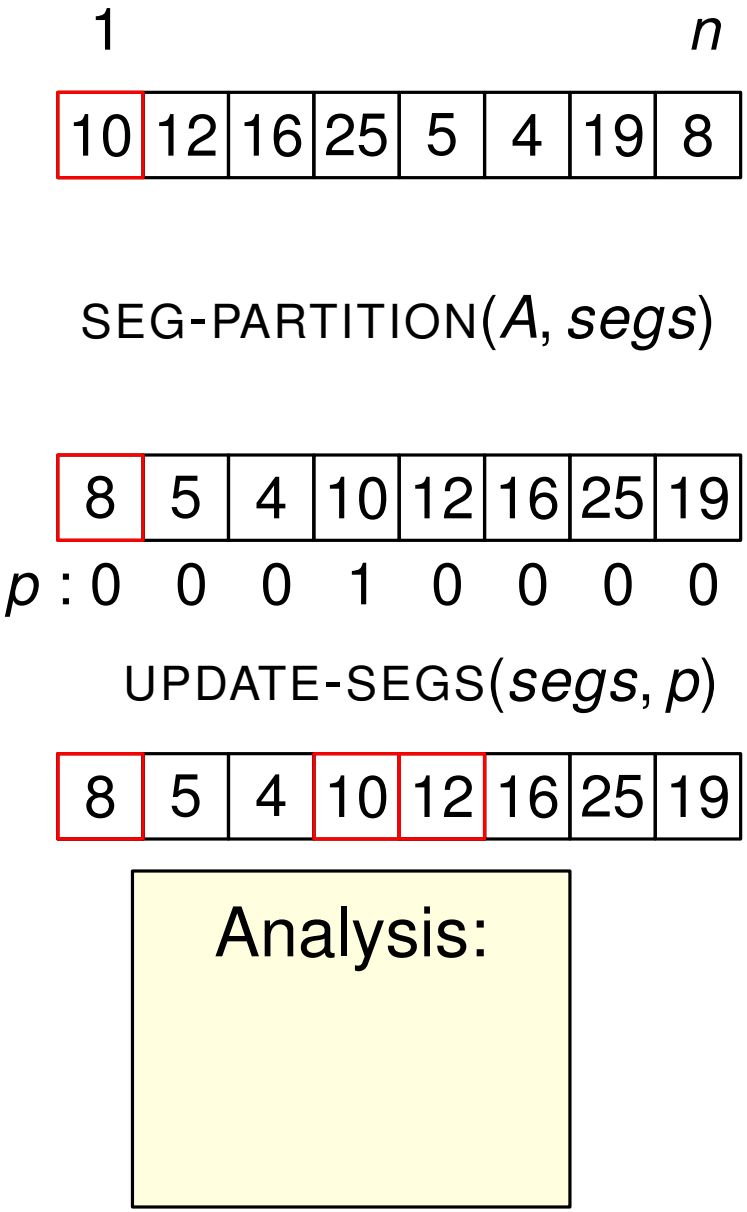
```
procedure UPDATE-SEGS(segs[1..n], p[1..n])
    for i = 1 to n in parallel do
        if p[i] = 1 then
            segs[i] = 1
            if i + 1 ≤ n then
                segs[i + 1] = 1
```



QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```

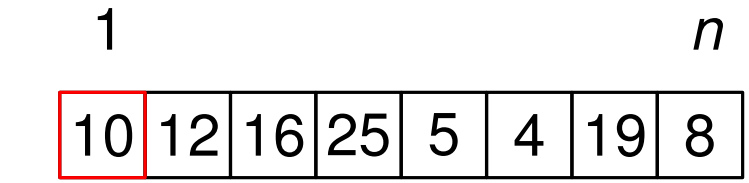
```
procedure UPDATE-SEGS(segs[1..n], p[1..n])
    for i = 1 to n in parallel do
        if p[i] = 1 then
            segs[i] = 1
            if i + 1 ≤ n then
                segs[i + 1] = 1
```



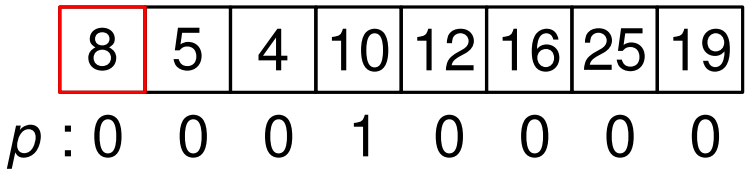
QUICKSORT($A[1..n]$)

```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```

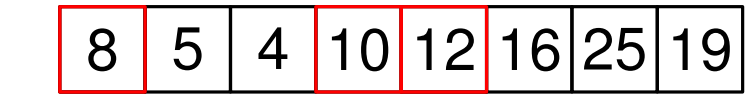
```
procedure UPDATE-SEGS(segs[1..n], p[1..n])
    for i = 1 to n in parallel do
        if p[i] = 1 then
            segs[i] = 1
            if i + 1 ≤ n then
                segs[i + 1] = 1
```



SEG-PARTITION(A, $segs$)



UPDATE-SEGS($segs$, p)



Analysis:

$T(n) = O(1)$

$W(n) = O(n)$

Checking Sortedness

procedure SORTED($A[1..n]$)

5	8	4	10	12	25	16	19

Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```

5	8	4	10	12	25	16	19

Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```

5	8	4	10	12	25	16	19

Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```

5	8	4	10	12	25	16	19

1

Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```

5	8	4	10	12	25	16	19

1 0

Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```

5	8	4	10	12	25	16	19

1 0 1

Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```

5	8	4	10	12	25	16	19

1 0 1 1

Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```

5	8	4	10	12	25	16	19
1	0	1	1	1	0	1	

Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```

5	8	4	10	12	25	16	19
1	0	1	1	1	0	1	

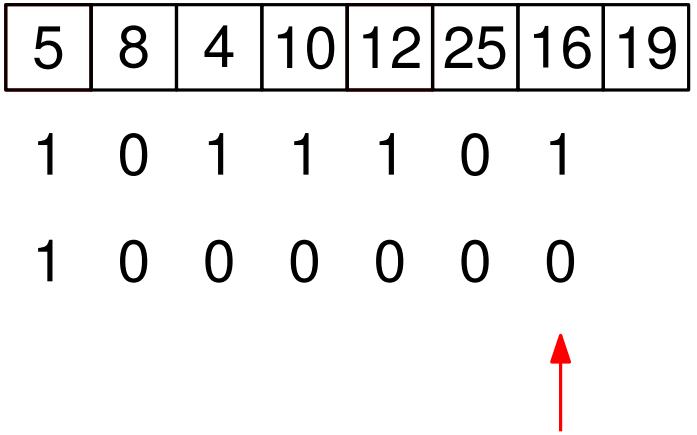
Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```

5	8	4	10	12	25	16	19
1	0	1	1	1	0	1	
1	0	0	0	0	0	0	

Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```



Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```

5	8	4	10	12	25	16	19
1	0	1	1	1	0	1	
1	0	0	0	0	0	0	

Analysis:

Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```

5	8	4	10	12	25	16	19
1	0	1	1	1	0	1	
				0	0	0	

$T(n) = O(1)$

$W(n) = O(n)$

Analysis:

Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```

5	8	4	10	12	25	16	19
1	0	1	1	1	0	1	
				0	0	0	

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

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

Analysis:

Checking Sortedness

```
procedure SORTED(A[1..n])
    sorted = new array of n − 1 booleans
    for i = 1 to n − 1 in parallel do
        if A[i] > A[i + 1] then
            sorted[i] = false
        else
            sorted[i] = true
    SCAN(sorted[1..n − 1], AND)
    return sorted[n − 1]
```

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

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

5	8	4	10	12	25	16	19
1	0	1	1	1	0	1	
				0	0	0	

Analysis:

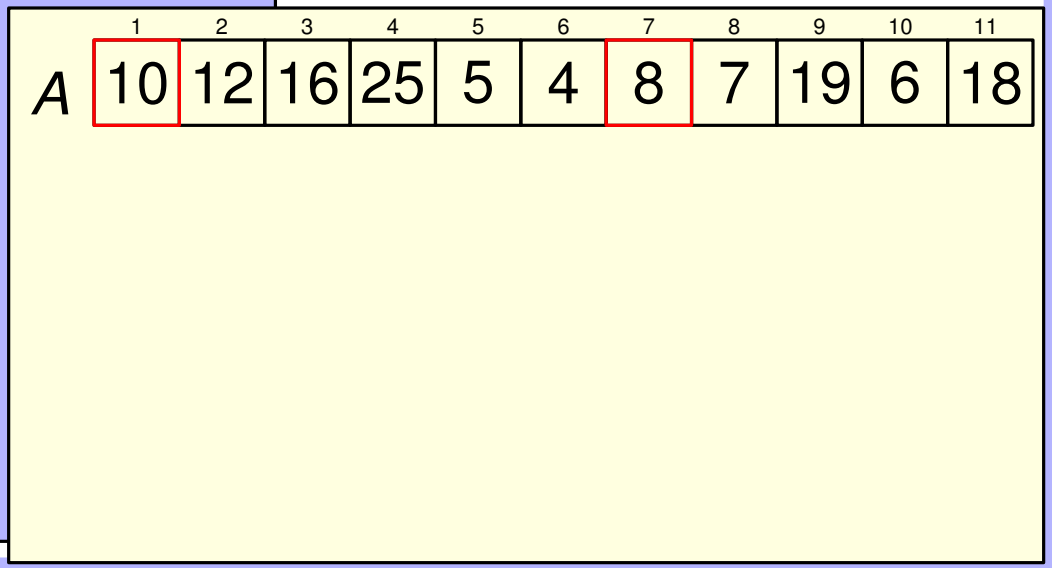
$T(n) = O(\log n)$
$W(n) = O(n)$

Segmented Partition

procedure SEG-PARTITION($A[1..n]$, $segs[1..n]$)

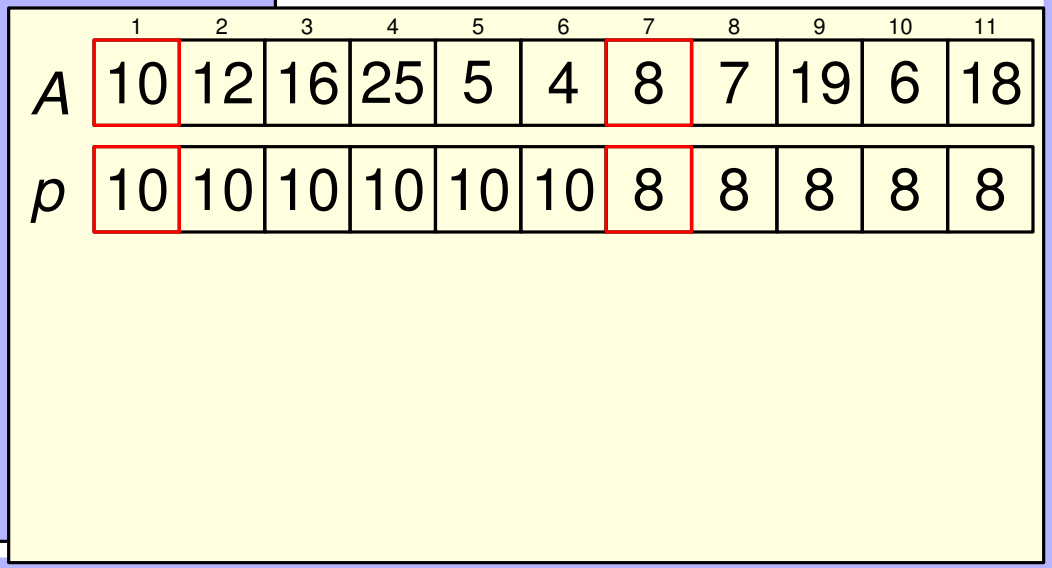
Segmented Partition

procedure SEG-PARTITION($A[1..n]$, $segs[1..n]$)



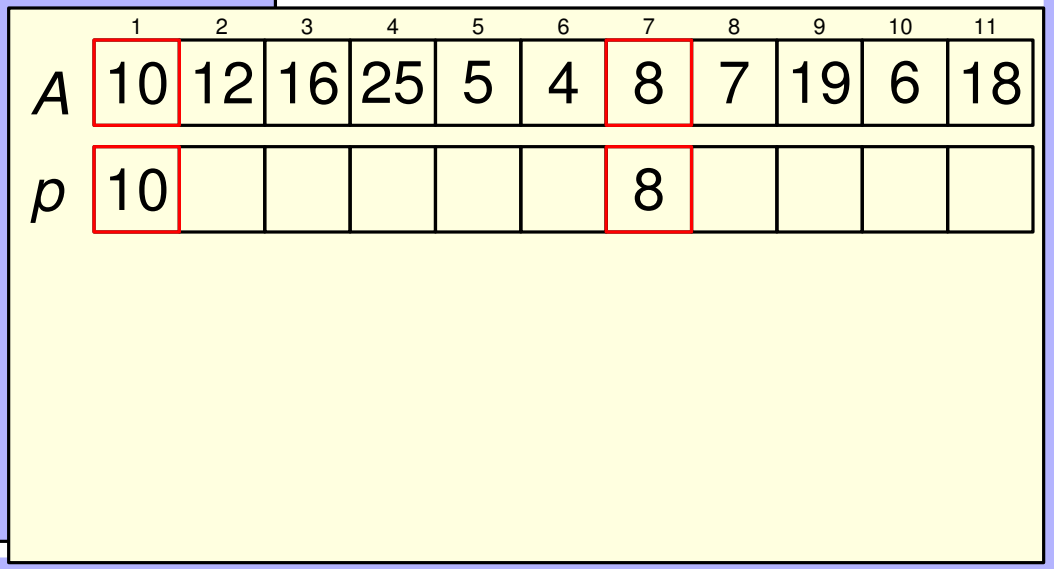
Segmented Partition

procedure SEG-PARTITION($A[1..n]$, $segs[1..n]$)



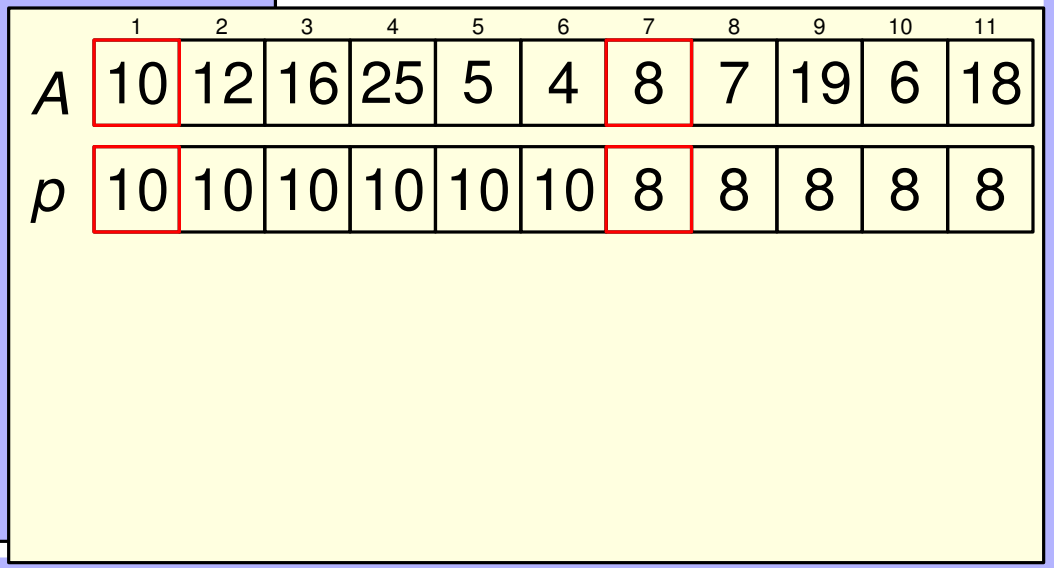
Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
```



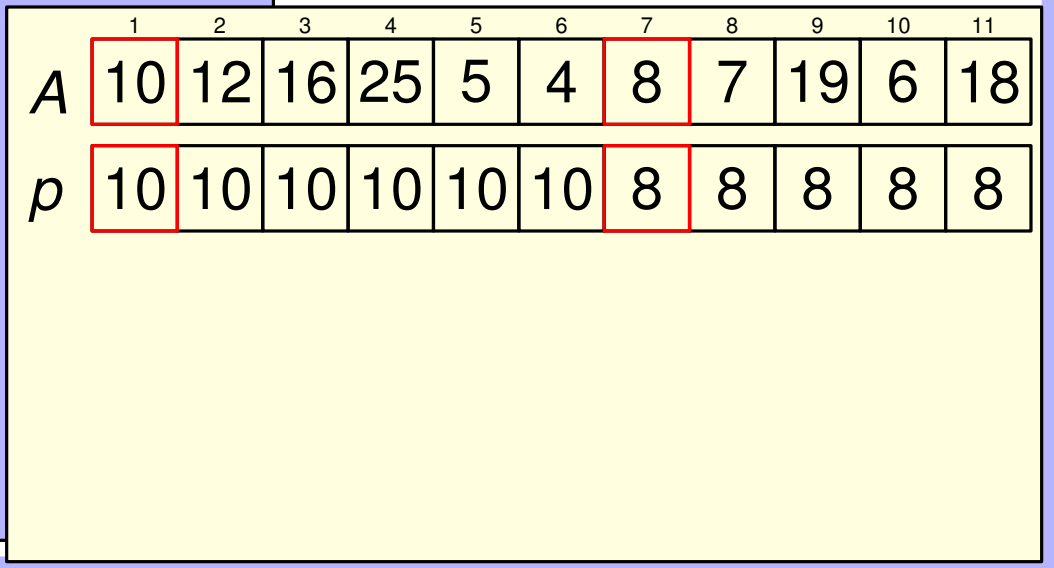
Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
```



Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
```



Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	0	0	0	1	1	1	1	0	1	0

Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
    for i = 1 to n in parallel do
        if A[i] ≤ pivots[i] then flags[i] = 1
        else flags[i] = 0
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	0	0	0	1	1	1	1	0	1	0

Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
    for i = 1 to n in parallel do
        if A[i] ≤ pivots[i] then flags[i] = 1
        else flags[i] = 0
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	0	0	0	1	1	1	1	0	1	0

Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
    for i = 1 to n in parallel do
        if A[i] ≤ pivots[i] then flags[i] = 1
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	0	0	0	1	1	1	1	0	1	0

Segmented Partition

procedure SEG-PARTITION($A[1..n]$, $segs[1..n]$)
  $pivots$, $flags$ = new arrays of size n
  for i = 1 to n **in parallel do**
    if $segs[i] = 1$ **then** $pivots[i] = A[i]$
  SEG-BROADCAST($pivots$, $segs$)
  for i = 1 to n **in parallel do**
    if $A[i] \leq pivots[i]$ **then** $flags[i] = 1$
    else $flags[i] = 0$
  $k[1..n]$ = SEG-FILTER($A[1..n]$, $flags[1..n]$)

	1	2	3	4	5	6	7	8	9	10	11
A	10	5	4	12	16	25	8	7	6	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0

Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
    for i = 1 to n in parallel do
        if A[i] ≤ pivots[i] then flags[i] = 1
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
```

Returns: sizes of new partitions

Details in Homework 3

	1	2	3	4	5	6	7	8	9	10	11
A	10	5	4	12	16	25	8	7	6	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0

Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
    for i = 1 to n in parallel do
        if A[i] ≤ pivots[i] then flags[i] = 1
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
```

Returns: sizes of new partitions

Details in Homework 3

	1	2	3	4	5	6	7	8	9	10	11
A	10	5	4	12	16	25	8	7	6	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				

Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
    for i = 1 to n in parallel do
        if A[i] ≤ pivots[i] then flags[i] = 1
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
```

Returns: sizes of new partitions

Details in Homework 3

	1	2	3	4	5	6	7	8	9	10	11
A	10	5	4	12	16	25	8	7	6	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				

Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
    for i = 1 to n in parallel do
        if A[i] ≤ pivots[i] then flags[i] = 1
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
```

Returns: sizes of new partitions

Details in Homework 3

pivots

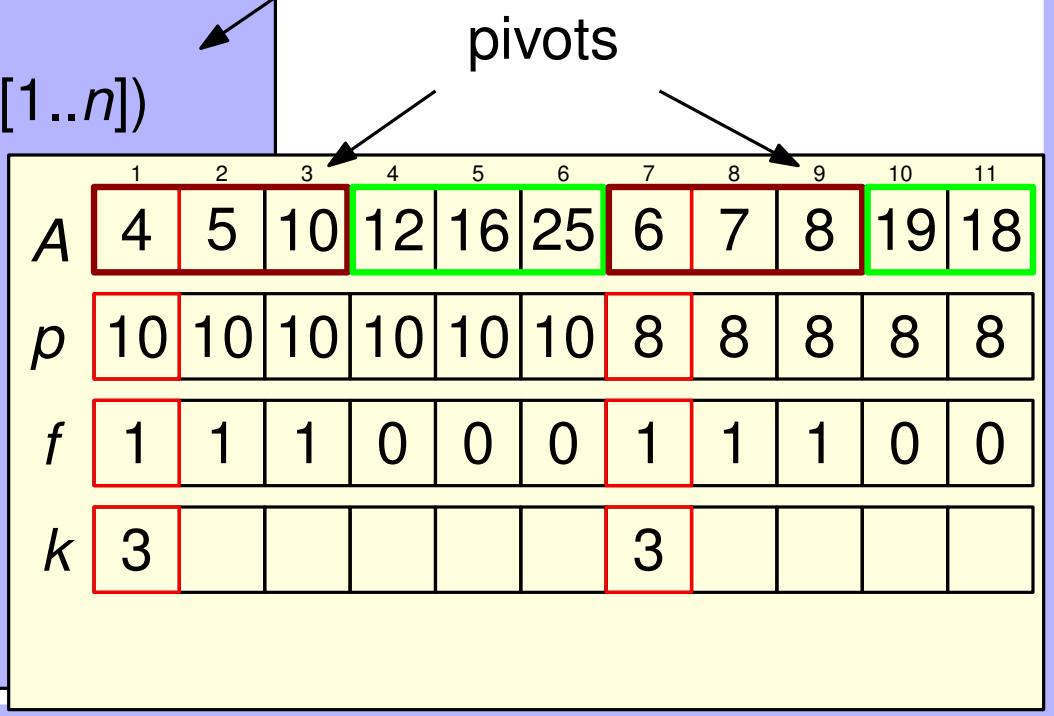
	1	2	3	4	5	6	7	8	9	10	11
A	10	5	4	12	16	25	8	7	6	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				

Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
    for i = 1 to n in parallel do
        if A[i] ≤ pivots[i] then flags[i] = 1
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
```

Returns: sizes of new partitions

Details in Homework 3



Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
    for i = 1 to n in parallel do
        if A[i] ≤ pivots[i] then flags[i] = 1
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
```

```
for i = 1 to n in parallel do

    if segs[i] = 1 then
        SWAP(A[i], A[i + k[i] − 1])
```

Returns: sizes of new partitions

Details in Homework 3

pivots

	1	2	3	4	5	6	7	8	9	10	11
A	4	5	10	12	16	25	6	7	8	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				

Segmented Partition

```
procedure QUICKSORT(A[1..n])
    segs = new array of n bits
    for i = 2 to n in parallel do segs[i] = 0
    segs[1] = 1                                  ▷ initialize segments
    while not SORTED(A[1..n]) do
        SEG-RANDOMIZE-PIVOTS(A, segs)
        p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
        UPDATE-SEGS(segs[1..n], p[1..n])
```

```
procedure SEG-PARTITION
    pivots, flags = new arr
    for i = 1 to n in paralle
        if segs[i] = 1 then p
    SEG-BROADCAST(pivot
    for i = 1 to n in paralle
        if A[i] ≤ pivots[i] th
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
```

```
for i = 1 to n in parallel do

    if segs[i] = 1 then
        SWAP(A[i], A[i + k[i] − 1])
```

pivots

	1	2	3	4	5	6	7	8	9	10	11
A	4	5	10	12	16	25	6	7	8	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				

Segmented Partition

```
procedure QUICKSORT(A[1..n])
    segs = new array of n bits
    for i = 2 to n in parallel do segs[i] = 0
    segs[1] = 1                                  ▷ initialize segments
    while not SORTED(A[1..n]) do
        SEG-RANDOMIZE-PIVOTS(A, segs)
        p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
        UPDATE-SEGS(segs[1..n], p[1..n])
```

```
procedure SEG-PARTITION
    pivots, flags = new arr
    for i = 1 to n in paralle
        if segs[i] = 1 then p
    SEG-BROADCAST(pivot
    for i = 1 to n in paralle
        if A[i] ≤ pivots[i] th
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
```

```
for i = 1 to n in parallel do

    if segs[i] = 1 then
        SWAP(A[i], A[i + k[i] − 1])
```

pivots

	1	2	3	4	5	6	7	8	9	10	11
A	4	5	10	12	16	25	6	7	8	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				
pF	0	0	1	0	0	0	0	0	1	0	0

Segmented Partition

```
procedure QUICKSORT(A[1..n])
    segs = new array of n bits
    for i = 2 to n in parallel do segs[i] = 0
    segs[1] = 1                                  ▷ initialize segments
    while not SORTED(A[1..n]) do
        SEG-RANDOMIZE-PIVOTS(A, segs)
        p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
        UPDATE-SEGS(segs[1..n], p[1..n])
```

```
procedure SEG-PARTITION
    pivots, flags = new arr
    for i = 1 to n in paralle
        if segs[i] = 1 then p
    SEG-BROADCAST(pivot
    for i = 1 to n in paralle
        if A[i] ≤ pivots[i] th
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
    pivotFlags = new array of size n
    for i = 1 to n in parallel do

        if segs[i] = 1 then
            SWAP(A[i], A[i + k[i] − 1])
```

pivots

	1	2	3	4	5	6	7	8	9	10	11
A	4	5	10	12	16	25	6	7	8	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				
pF	0	0	1	0	0	0	0	0	1	0	0

Segmented Partition

```
procedure QUICKSORT(A[1..n])
    segs = new array of n bits
    for i = 2 to n in parallel do segs[i] = 0
    segs[1] = 1                                  ▷ initialize segments
    while not SORTED(A[1..n]) do
        SEG-RANDOMIZE-PIVOTS(A, segs)
        p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
        UPDATE-SEGS(segs[1..n], p[1..n])
```

```
procedure SEG-PARTITION
    pivots, flags = new arr
    for i = 1 to n in paralle
        if segs[i] = 1 then p
    SEG-BROADCAST(pivot
    for i = 1 to n in paralle
        if A[i] ≤ pivots[i] th
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
    pivotFlags = new array of size n
    for i = 1 to n in parallel do
        pivotFlags[i] = 0
        if segs[i] = 1 then
            SWAP(A[i], A[i + k[i] − 1])
            pivotFlags[i + k[i] − 1] = 1
```

pivots

	1	2	3	4	5	6	7	8	9	10	11
A	4	5	10	12	16	25	6	7	8	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				
pF	0	0	1	0	0	0	0	0	1	0	0

Segmented Partition

```
procedure QUICKSORT(A[1..n])
    segs = new array of n bits
    for i = 2 to n in parallel do segs[i] = 0
    segs[1] = 1                                  ▷ initialize segments
    while not SORTED(A[1..n]) do
        SEG-RANDOMIZE-PIVOTS(A, segs)
        p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
        UPDATE-SEGS(segs[1..n], p[1..n])
```

```
procedure SEG-PARTITION
    pivots, flags = new arr
    for i = 1 to n in paralle
        if segs[i] = 1 then p
    SEG-BROADCAST(pivot
    for i = 1 to n in paralle
        if A[i] ≤ pivots[i] th
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
    pivotFlags = new array of size n
    for i = 1 to n in parallel do
        pivotFlags[i] = 0
        if segs[i] = 1 then
            SWAP(A[i], A[i + k[i] − 1])
            pivotFlags[i + k[i] − 1] = 1
    return pivotFlags[1..n]
```

pivots

	1	2	3	4	5	6	7	8	9	10	11
A	4	5	10	12	16	25	6	7	8	19	18
p	10	10	10	10	10	10	8	8	8	8	8
f	1	1	1	0	0	0	1	1	1	0	0
k	3						3				
pF	0	0	1	0	0	0	0	0	1	0	0

Segmented Partition

Analysis:

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
    for i = 1 to n in parallel do
        if A[i] ≤ pivots[i] then flags[i] = 1
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
    pivotFlags = new array of size n
    for i = 1 to n in parallel do
        pivotFlags[i] = 0
        if segs[i] = 1 then
            SWAP(A[i], A[i + k[i] − 1])
            pivotFlags[i + k[i] − 1] = 1
    return pivotFlags[1..n]
```

Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
    for i = 1 to n in parallel do
        if A[i] ≤ pivots[i] then flags[i] = 1
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
    pivotFlags = new array of size n
    for i = 1 to n in parallel do
        pivotFlags[i] = 0
        if segs[i] = 1 then
            SWAP(A[i], A[i + k[i] − 1])
            pivotFlags[i + k[i] − 1] = 1
    return pivotFlags[1..n]
```

Analysis:

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

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

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

Segmented Partition

Analysis:

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
    for i = 1 to n in parallel do
        if A[i] ≤ pivots[i] then flags[i] = 1
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
    pivotFlags = new array of size n
    for i = 1 to n in parallel do
        pivotFlags[i] = 0
        if segs[i] = 1 then
            SWAP(A[i], A[i + k[i] − 1])
            pivotFlags[i + k[i] − 1] = 1
    return pivotFlags[1..n]
```

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

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

Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then pivots[i] = A[i]
    SEG-BROADCAST(pivots, segs)
    for i = 1 to n in parallel do
        if A[i] ≤ pivots[i] then flags[i] = 1
        else flags[i] = 0
    k[1..n] = SEG-FILTER(A[1..n], flags[1..n])
    pivotFlags = new array of size n
    for i = 1 to n in parallel do
        pivotFlags[i] = 0
        if segs[i] = 1 then
            SWAP(A[i], A[i + k[i] − 1])
            pivotFlags[i + k[i] − 1] = 1
    return pivotFlags[1..n]
```

Analysis:

Total:

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

$W(n) = O(n)$

Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i]
    SEG-BROADC
    for i = 1 to n
        if A[i] ≤ p
        else flags
    k[1..n] = SEG
    pivotFlags =
    for i = 1 to n
        pivotFlags[i] = 0
        if segs[i] = 1 then
            SWAP(A[i], A[i + k[i] − 1])
            pivotFlags[i + k[i] − 1] = 1
    return pivotFlags[1..n]
```

```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```

Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i]
    SEG-BROADC
    for i = 1 to n
        if A[i] ≤ p
        else flags
    k[1..n] = SEG
    pivotFlags =
    for i = 1 to n
        pivotFlags[i] = 0
        if segs[i] = 1 then
            SWAP(A[i], A[i + k[i] − 1])
            pivotFlags[i + k[i] − 1] = 1
    return pivotFlags[1..n]
```

```
procedure QUICKSORT(A[1..n])
    segs = new array of n bits
    for i = 2 to n in parallel do segs[i] = 0
    segs[1] = 1                          ▷ initialize segments
✓   while not SORTED(A[1..n]) do
        SEG-RANDOMIZE-PIVOTS(A, segs)
✓       p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
✓       UPDATE-SEGS(segs[1..n], p[1..n])
```

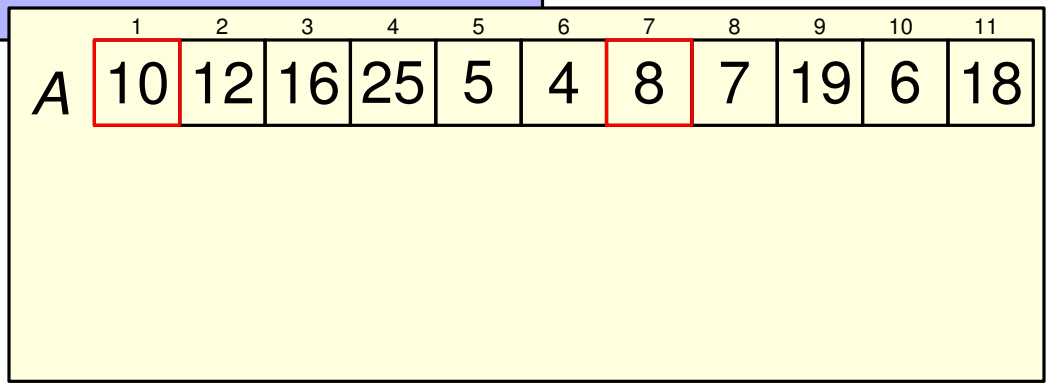
Segmented Partition

```
procedure SEG-PARTITION(A[1..n], segs[1..n])
    pivots, flags = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i]
    SEG-BROADC
    for i = 1 to n
        if A[i] ≤ p
        else flags
    k[1..n] = SEG
    pivotFlags =
    for i = 1 to n
        pivotFlags[i] = 0
        if segs[i] = 1 then
            SWAP(A[i], A[i + k[i] − 1])
            pivotFlags[i + k[i] − 1] = 1
    return pivotFlags[1..n]
```

```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
✓ while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
✓     p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
✓     UPDATE-SEGS(segs[1..n], p[1..n])
```

Segmented Randomize Pivots

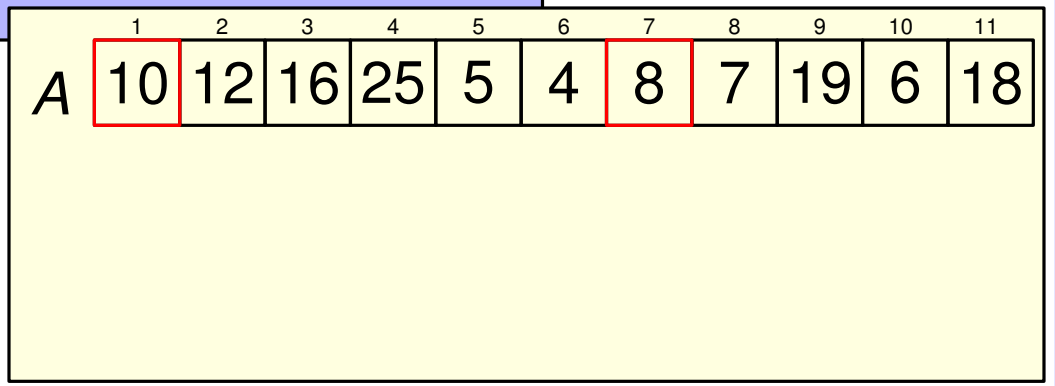
procedure SEG-RANDOMIZE-PIVOTS($A[1..n]$, $segs[1..n]$)



Segmented Randomize Pivots

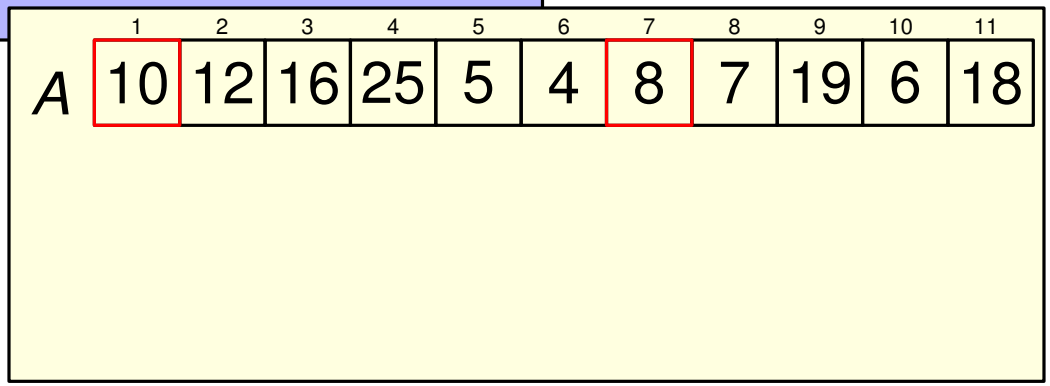
procedure SEG-RANDOMIZE-PIVOTS($A[1..n]$, $segs[1..n]$)

for $i = 1$ to n **in parallel do**
  if $seg[i] = 1$ **then**
    rnd = RANDOM($start[i]$, $end[i]$)
    SWAP($A[i]$, $A[rnd]$)



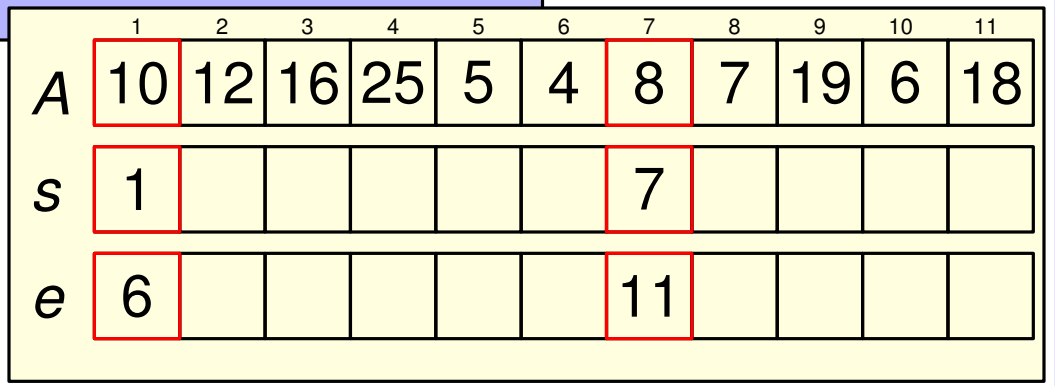
Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS(A[1..n], segs[1..n])
    start, end = new arrays of size n




    for i = 1 to n in parallel do
        if seg[i] = 1 then
            rnd = RANDOM(start[i], end[i])
            SWAP(A[i], A[rnd])
```



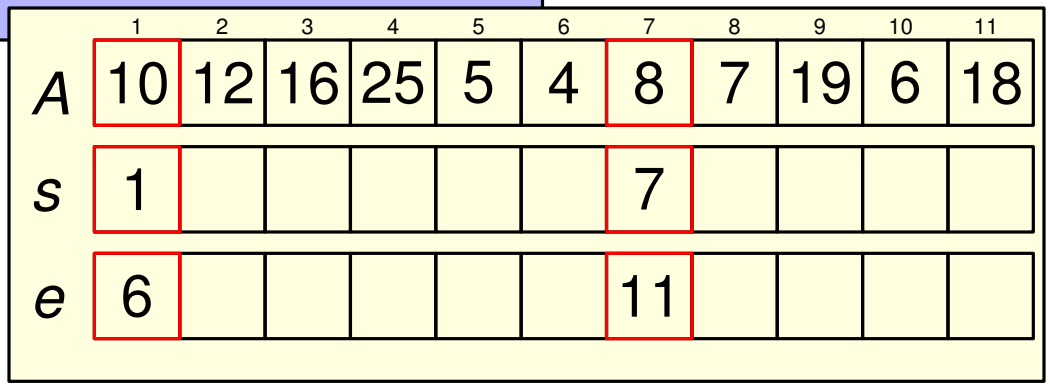
Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS(A[1..n], segs[1..n])
    start, end = new arrays of size n




    for i = 1 to n in parallel do
        if seg[i] = 1 then
            rnd = RANDOM(start[i], end[i])
            SWAP(A[i], A[rnd])
```



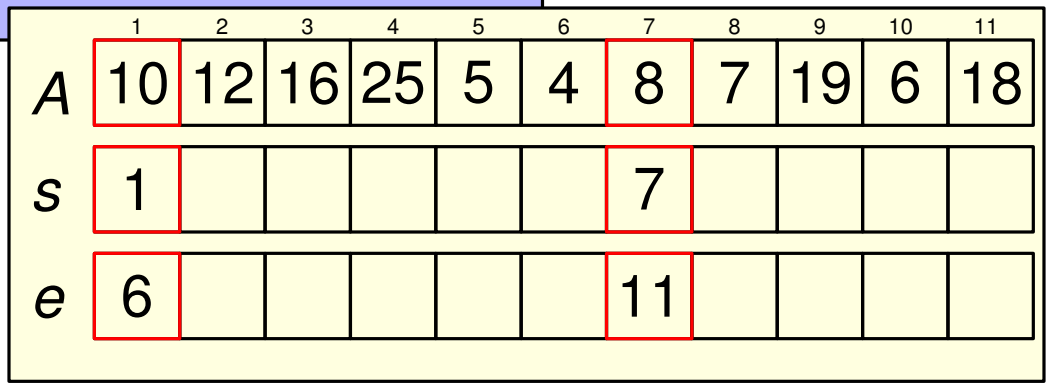
Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS(A[1..n], segs[1..n])
    start, end = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then start[i] = i

    for i = 1 to n in parallel do
        if seg[i] = 1 then
            rnd = RANDOM(start[i], end[i])
            SWAP(A[i], A[rnd])
```



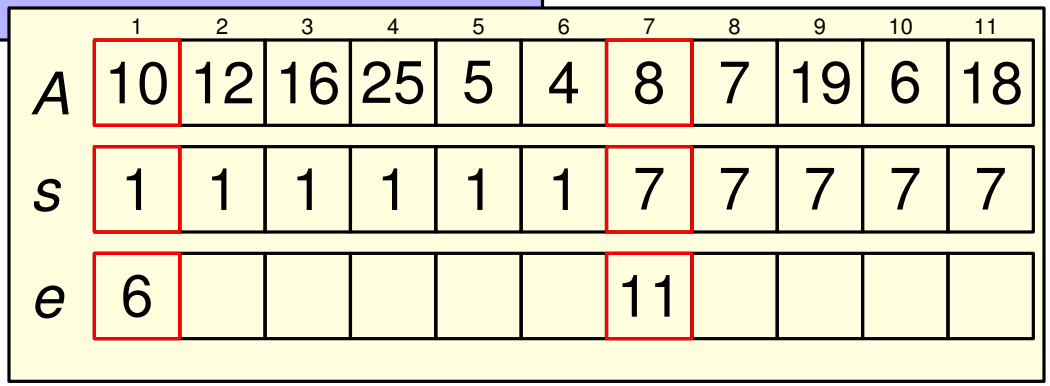
Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS(A[1..n], segs[1..n])
    start, end = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then start[i] = i
    SEG-BROADCAST(start[1..n], segs[1..n])


    for i = 1 to n in parallel do
        if seg[i] = 1 then
            rnd = RANDOM(start[i], end[i])
            SWAP(A[i], A[rnd])
```



Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS(A[1..n], segs[1..n])
    start, end = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then start[i] = i
    SEG-BROADCAST(start[1..n], segs[1..n])


    for i = 1 to n in parallel do
        if seg[i] = 1 then
            rnd = RANDOM(start[i], end[i])
            SWAP(A[i], A[rnd])
```



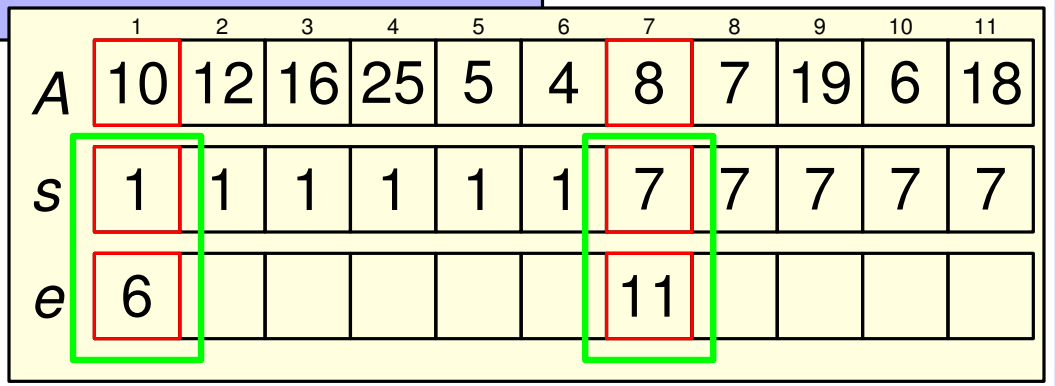
Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS(A[1..n], segs[1..n])
    start, end = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then start[i] = i
    SEG-BROADCAST(start[1..n], segs[1..n])
    for i = 1 to n in parallel do
        if i = n  OR  segs[i + 1] = 1 then end[start[i]] = i
    for i = 1 to n in parallel do
        if seg[i] = 1 then
            rnd = RANDOM(start[i], end[i])
            SWAP(A[i], A[rnd])
```

	1	2	3	4	5	6	7	8	9	10	11
A	10	12	16	25	5	4	8	7	19	6	18
s	1	1	1	1	1	1	7	7	7	7	7
e	6						11				

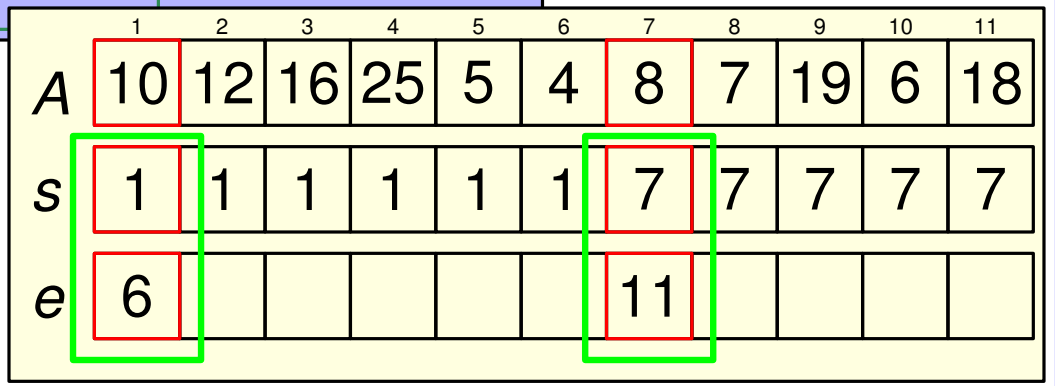
Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS(A[1..n], segs[1..n])
    start, end = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then start[i] = i
    SEG-BROADCAST(start[1..n], segs[1..n])
    for i = 1 to n in parallel do
        if i = n  OR  segs[i + 1] = 1 then end[start[i]] = i
    for i = 1 to n in parallel do
        if seg[i] = 1 then
            rnd = RANDOM(start[i], end[i])
            SWAP(A[i], A[rnd])
```



Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS(A[1..n], segs[1..n])
    start, end = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then start[i] = i
    SEG-BROADCAST(start[1..n], segs[1..n])
    for i = 1 to n in parallel do
        if i = n  OR  segs[i + 1] = 1 then end[start[i]] = i
    for i = 1 to n in parallel do
        if seg[i] = 1 then
            rnd = RANDOM(start[i], end[i])
            SWAP(A[i], A[rnd])
```



Segmented Randomize Pivots

Analysis:

```
procedure SEG-RANDOMIZE-PIVOTS(A[1..n], segs[1..n])
    start, end = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then start[i] = i
    SEG-BROADCAST(start[1..n], segs[1..n])
    for i = 1 to n in parallel do
        if i = n  OR  segs[i + 1] = 1 then end[start[i]] = i
    for i = 1 to n in parallel do
        if seg[i] = 1 then
            rnd = RANDOM(start[i], end[i])
            SWAP(A[i], A[rnd])
```

Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS(A[1..n], segs[1..n])
    start, end = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then start[i] = i
    SEG-BROADCAST(start[1..n], segs[1..n])
    for i = 1 to n in parallel do
        if i = n  OR  segs[i + 1] = 1 then end[start[i]] = i
    for i = 1 to n in parallel do
        if seg[i] = 1 then
            rnd = RANDOM(start[i], end[i])
            SWAP(A[i], A[rnd])
```

Analysis:

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

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

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

Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS(A[1..n], segs[1..n])
    start, end = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then start[i] = i
    SEG-BROADCAST(start[1..n], segs[1..n])
    for i = 1 to n in parallel do
        if i = n  OR  segs[i + 1] = 1 then end[start[i]] = i
    for i = 1 to n in parallel do
        if seg[i] = 1 then
            rnd = RANDOM(start[i], end[i])
            SWAP(A[i], A[rnd])
```

Analysis:

Total:

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

$W(n) = O(n)$

Segmented Randomize Pivots

```
procedure SEG-RANDOMIZE-PIVOTS(A[1..n], segs[1..n])
    start, end = new arrays of size n
    for i = 1 to n in parallel do
        if segs[i] = 1 then start[i] = i
    SEG-BROADCAST(start[1..n], segs[1..n])
    for i = 1 to n in parallel do
        if i = n OR segs[i + 1] = 1 then end[start[i]] = i
    for i = 1 to n in parallel do
        if seg[i] = 1 then
            rnd = RANDOM(start[i], end[i])
            SWAP(A[i], A[rnd])
```

Analysis:

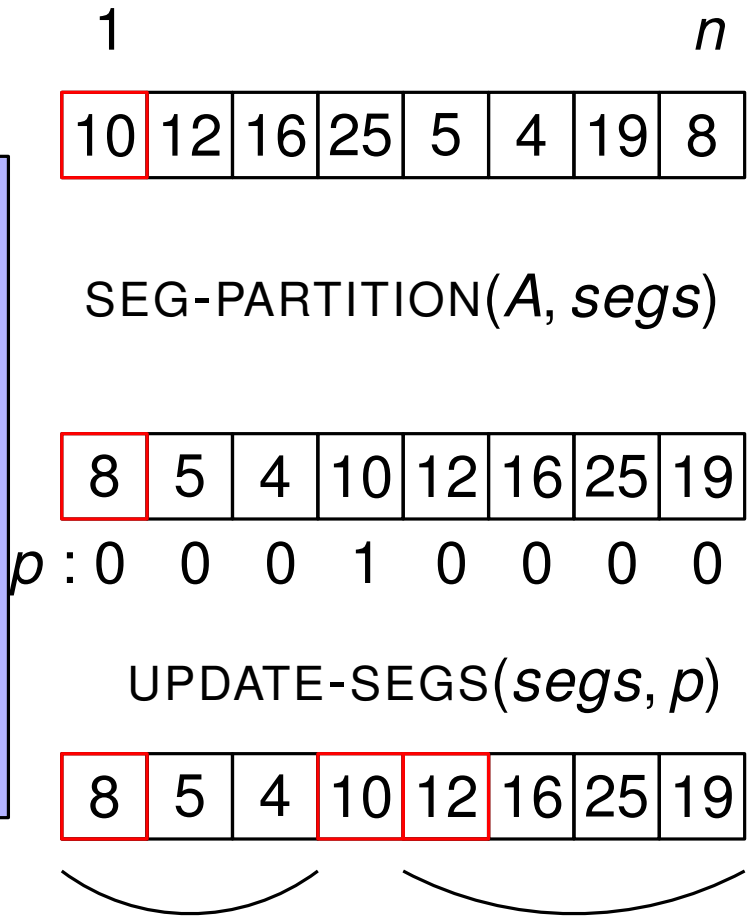
Total:

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

$W(n) = O(n)$

QUICKSORT($A[1..n]$) Analysis

```
procedure QUICKSORT(A[1..n])
  segs = new array of n bits
  for i = 2 to n in parallel do segs[i] = 0
  segs[1] = 1                          ▷ initialize segments
  while not SORTED(A[1..n]) do
      SEG-RANDOMIZE-PIVOTS(A, segs)
      p[1..n] = SEG-PARTITION(A[1..n], segs[1..n])
      UPDATE-SEGS(segs[1..n], p[1..n])
```



Analysis:

$O(\log n)$ iterations of the **while** loop in expectation

Each iteration: $T(n) = O(\log n)$, $W(n) = O(n)$

Total (Expected): $T(n) = O(\log^2 n)$, $W(n) = O(n \log n)$