CS11313 - **Spring** 2022

Design & Analysis *of* Algorithms

Selection

Ibrahim Albluwi

Warmup Quiz

How can we find the maximum m elements in an array of size n ?

Warmup Quiz

How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Warmup Quiz

How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 2. Sort the array using Merge Sort and take the last m elements.
Running time: $\Theta(n \log n)$.

Warmup Quiz

How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 2. Sort the array using Merge Sort and take the last m elements.
Running time: $\Theta(n \log n)$.

Answer 3. Insert all elements into a max-PQ and then remove m elements.
Running time: $\Theta(n \log n)$ to insert + $O(m \log n)$ to remove = $\Theta(n \log n)$

Warmup Quiz

How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 2. Sort the array using Merge Sort and take the last m elements.
Running time: $\Theta(n \log n)$.

Answer 3. Insert all elements into a max-PQ and then remove m elements.
Running time: $\Theta(n \log n)$ to insert + $O(m \log n)$ to remove = $\Theta(n \log n)$

Answer 4. Construct a heap and run m iterations of Heapsort. Take the last m elements in the array.
Running time: $\Theta(n)$ for heap construction + $O(m \log n)$ for the m iterations = $O(n + m \log n)$

Warmup Quiz

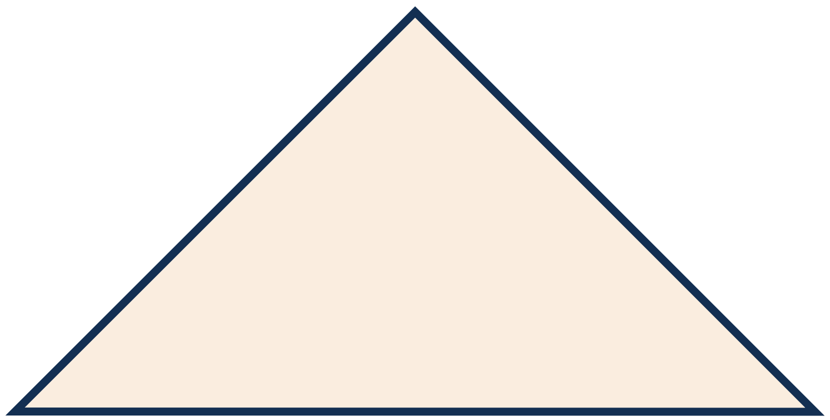
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

```
a[] = 1 5 9 8 4 11 3 0 7 8 6 10 2
```




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

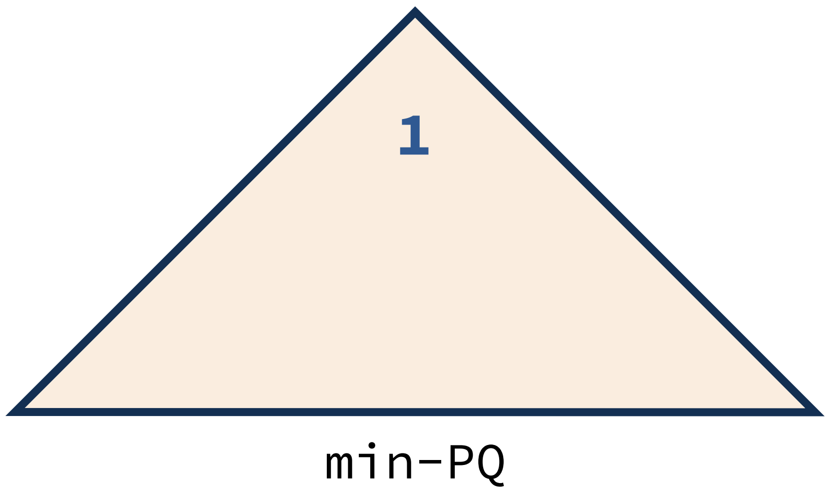
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

```
a[] =   5 9 8 4 11 3 0 7 8 6 10 2
```




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

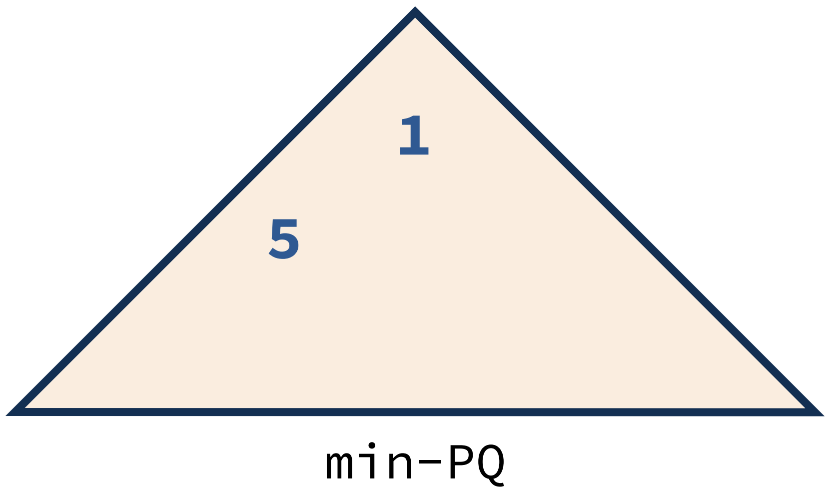
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

```
a[] =     9 8 4 11 3 0 7 8 6 10 2
```




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

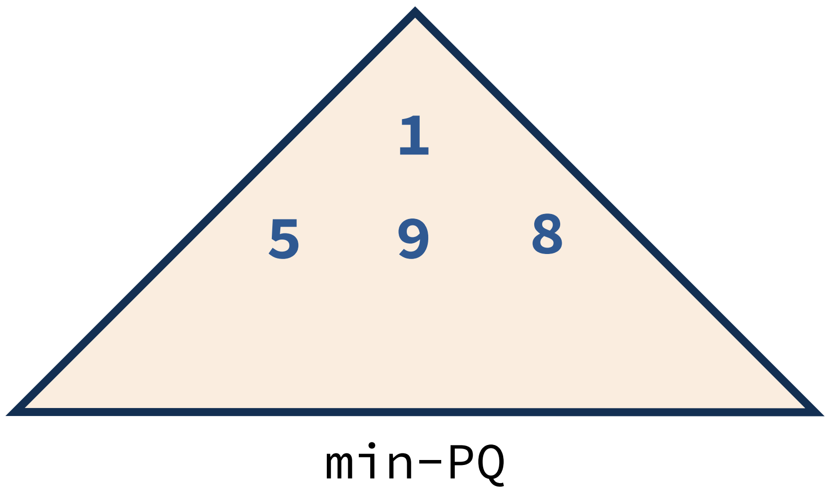
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

```
a[] =        4 11 3 0 7 8 6 10 2
```




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

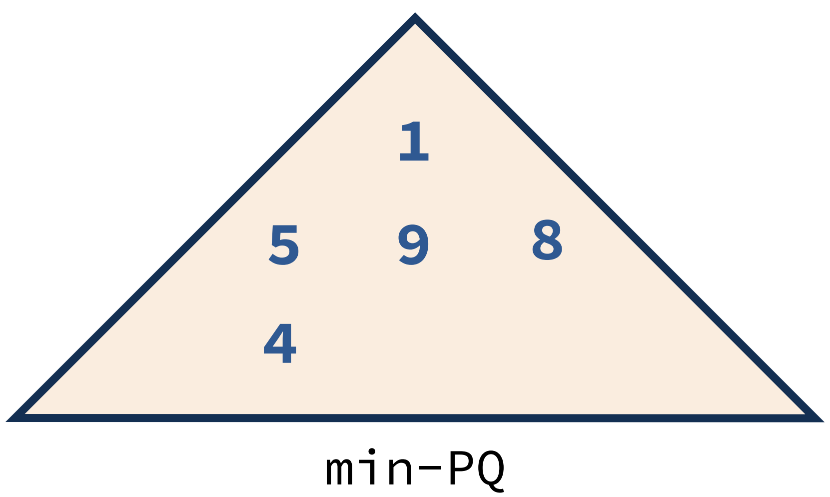
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

```
a[] =          11 3 0 7 8 6 10 2
```




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

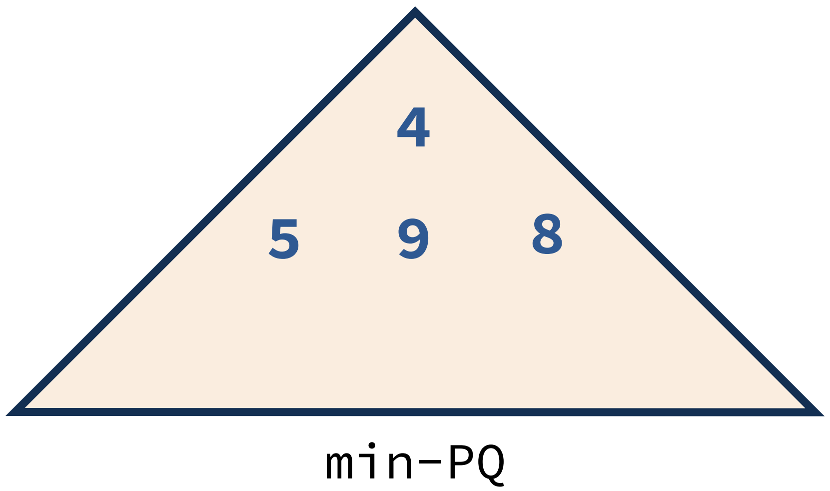
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

```
a[] =     11 3 0 7 8 6 10 2
```




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

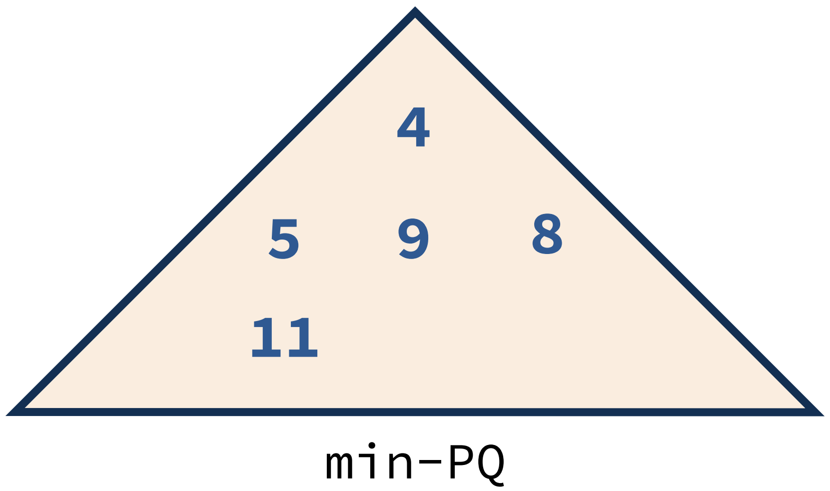
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

```
a[] =                3 0 7 8 6 10 2
```




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

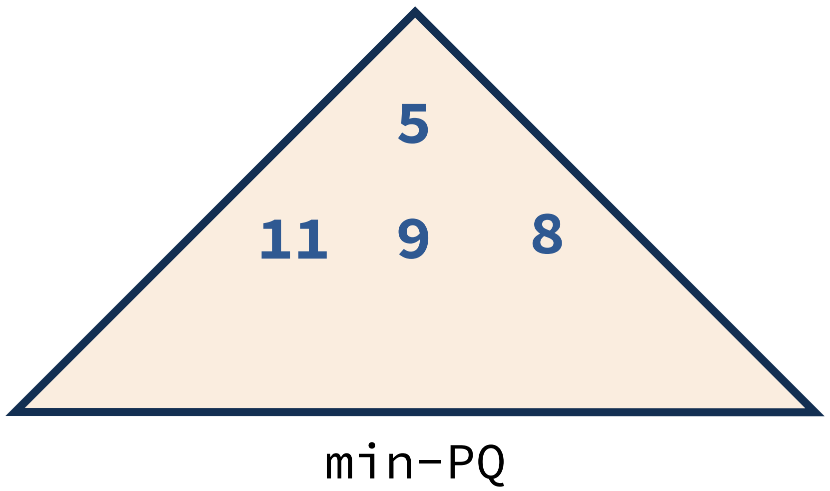
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

```
a[] =        3 0 7 8 6 10 2
```




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

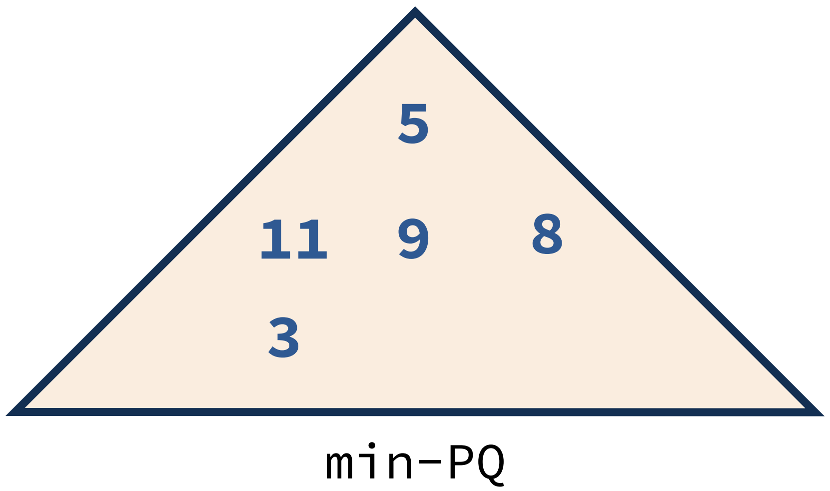
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

a[] = 0 7 8 6 10 2




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

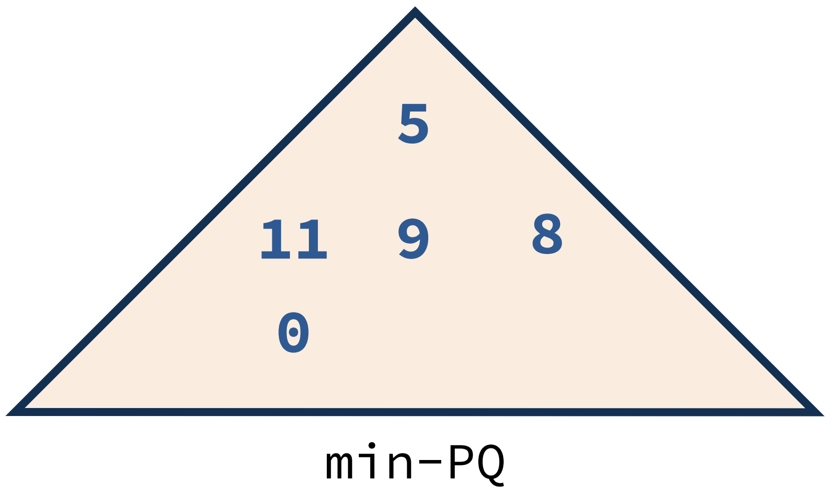
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

a[] = 7 8 6 10 2




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

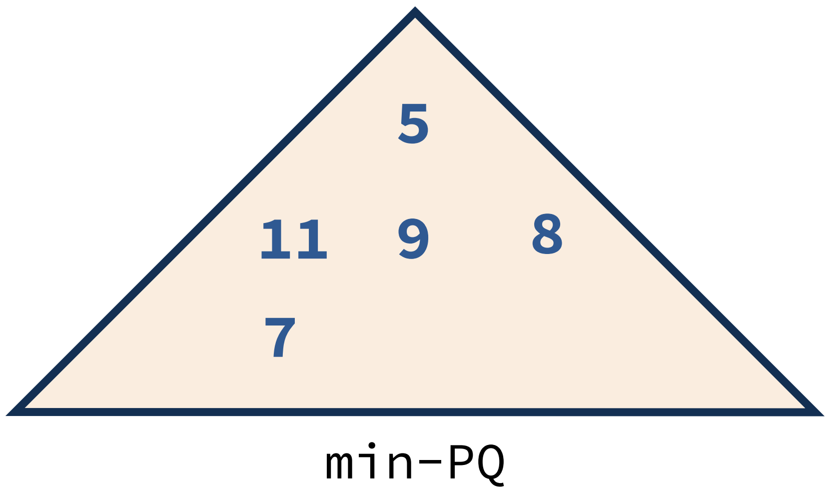
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

a[] = 8 6 10 2




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

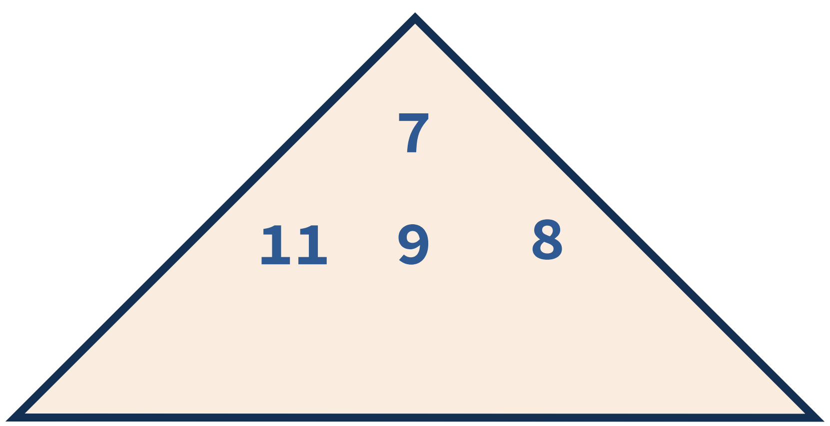
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

```
a[] =                    8 6 10 2
```



min-PQ

```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

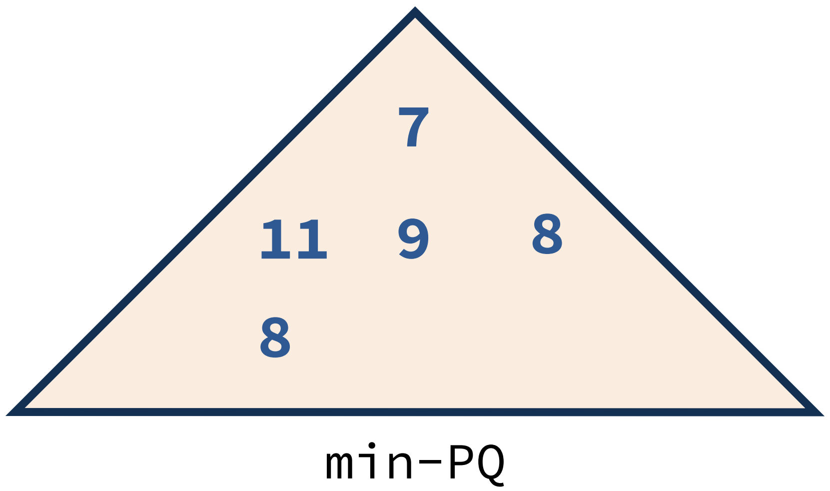
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

```
a[] =                6 10 2
```




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

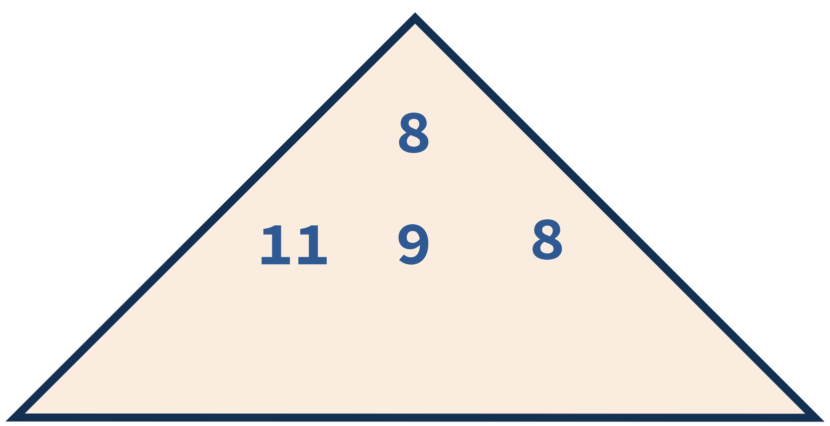
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

a[] = 6 10 2



min-PQ

```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

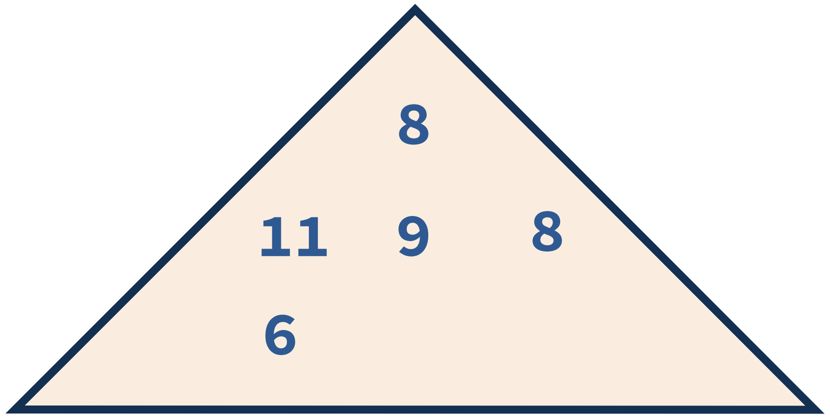
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

`a[] = 10 2`



min-PQ

```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

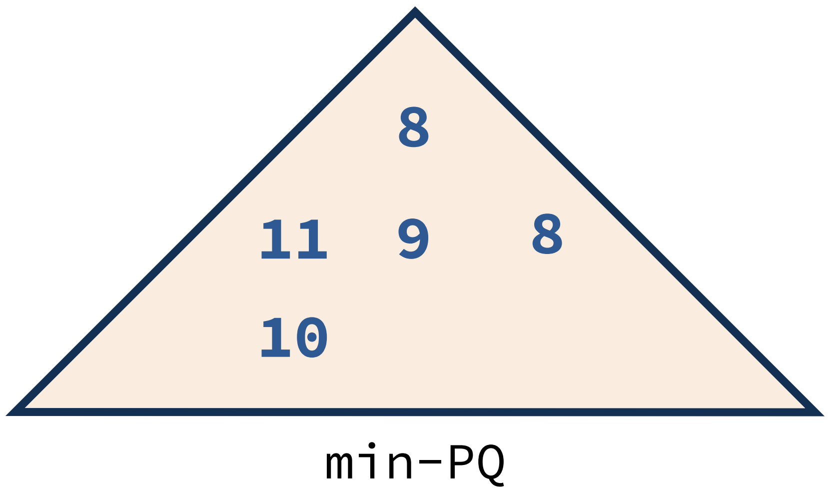
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

a[] = 2




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

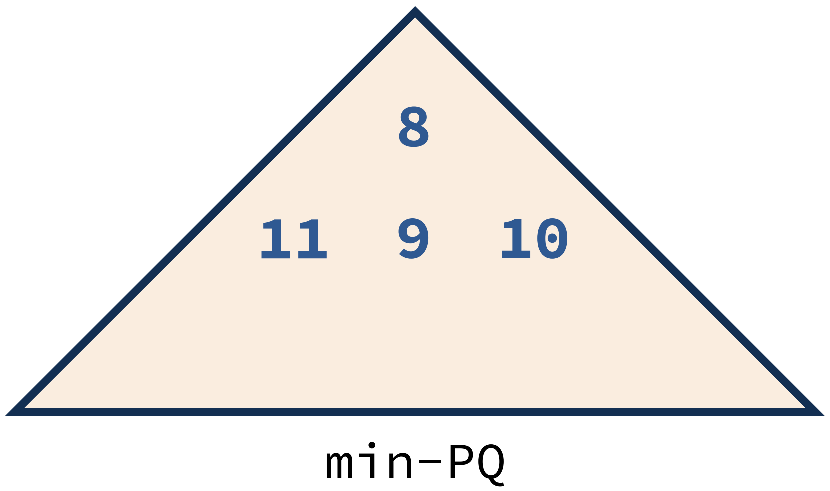
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

a[] = 2




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

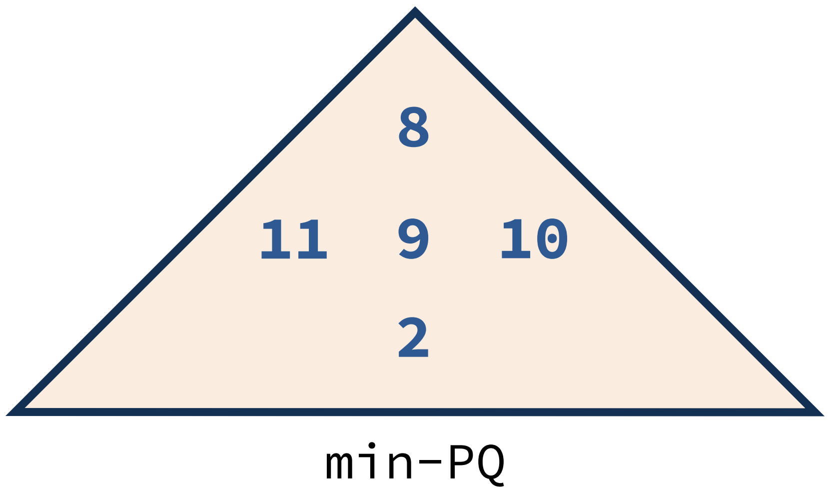
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

a[] =




```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

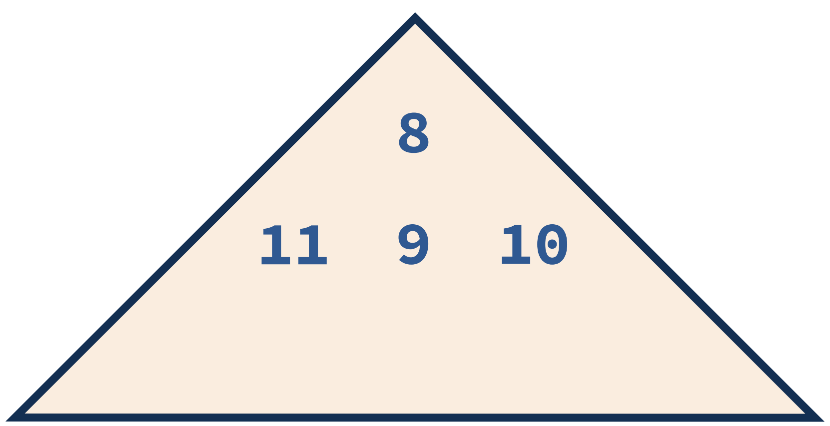
How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 5.

Example. $m = 4$

a[] = 1 5 **9** **8** 4 **11** 3 0 7 8 6 **10** 2



min-PQ

```
for each element k in a[]:
    minPQ.INSERT(k)
    if (minPQ.size > m)
        minPQ.DEL-MIN()
```

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Warmup Quiz

How can we find the maximum m elements in an array of size n ?

Answer 1. Perform m iterations of selection sort.
Running time: $\Theta(mn)$.

Answer 2. Sort the array using Merge Sort and take the last m elements.
Running time: $\Theta(n \log n)$.

Answer 3. Insert all elements into a max-PQ and then remove m elements.
Running time: $\Theta(n \log n)$ to insert + $O(m \log n)$ to remove = $\Theta(n \log n)$

Answer 4. Construct a heap and run m iterations of Heapsort. Take the last m elements in the array.
Running time: $\Theta(n)$ for heap construction + $O(m \log n)$ for the m iterations = $O(n + m \log n)$

Answer 5. Insert all elements into a min-PQ. Remove the minimum element whenever the size of the min-PQ exceeds m.
Running time: $\Theta(n \log m)$

Selection

Problem. Find the element with rank k (k^{th} largest element) in an arbitrary array of size n.

Selection

Problem. Find the element with rank k (k^{th} largest element) in an arbitrary array of size n.

Examples. $k = 0$ (minimum), $k = n - 1$ (maximum), $k = \frac{n}{2}$ (median).

Selection

Problem. Find the element with rank k (k^{th} largest element) in an arbitrary array of size n.

Examples. $k = 0$ (minimum), $k = n - 1$ (maximum), $k = \frac{n}{2}$ (median).

Relation to Sorting.

- Repeated selection leads to sorting.
- If the array is sorted, selection is easy!

Selection

Problem. Find the element with rank k (k^{th} largest element) in an arbitrary array of size n.

Examples. $k = 0$ (minimum), $k = n - 1$ (maximum), $k = \frac{n}{2}$ (median).

Relation to Sorting.

- Repeated selection leads to sorting.
- If the array is sorted, selection is easy!

Candidate Solutions.

- Perform k iterations of selection sort. $\longleftarrow \Theta(kn)$
- Insert the elements into a binary heap data structure. $\longleftarrow O(n \log n)$
- Sort and then get the element at index k. $\longleftarrow O(n \log n)$ if heapsort is used.
- Heapify and then remove k elements from the heap. $\longleftarrow O(n + k \log n)$.

Selection

Problem. Find the element with rank k (k^{th} largest element) in an arbitrary array of size n.

Examples. $k = 0$ (minimum), $k = n - 1$ (maximum), $k = \frac{n}{2}$ (median).

Relation to Sorting.

- Repeated selection leads to sorting.
- If the array is sorted, selection is easy!

Candidate Solutions.

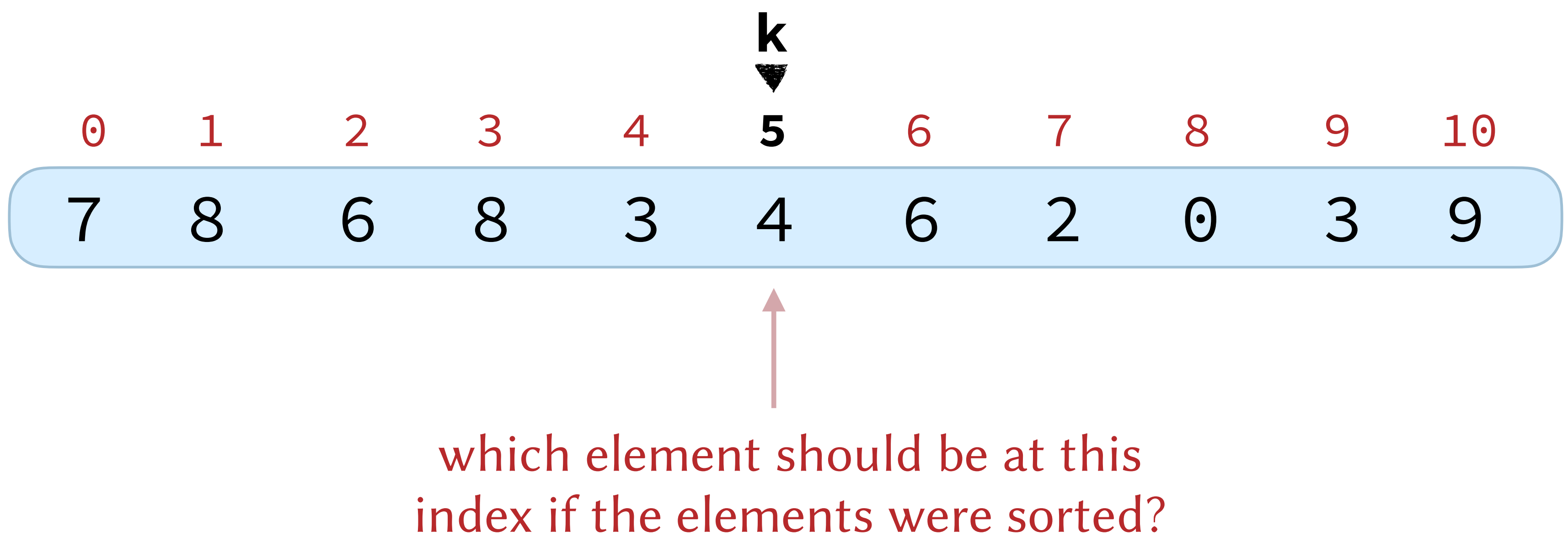
- Perform k iterations of selection sort. $\longleftarrow \Theta(kn)$
- Insert the elements into a binary heap data structure. $\longleftarrow O(n \log n)$
- Sort and then get the element at index k. $\longleftarrow O(n \log n)$ if heapsort is used.
- Heapify and then remove k elements from the heap. $\longleftarrow O(n + k \log n)$.

Can we do better?

Is selection as hard as sorting?
(requires $\sim n \log n$ compares
in the worst case if $k = \frac{n}{2}$)

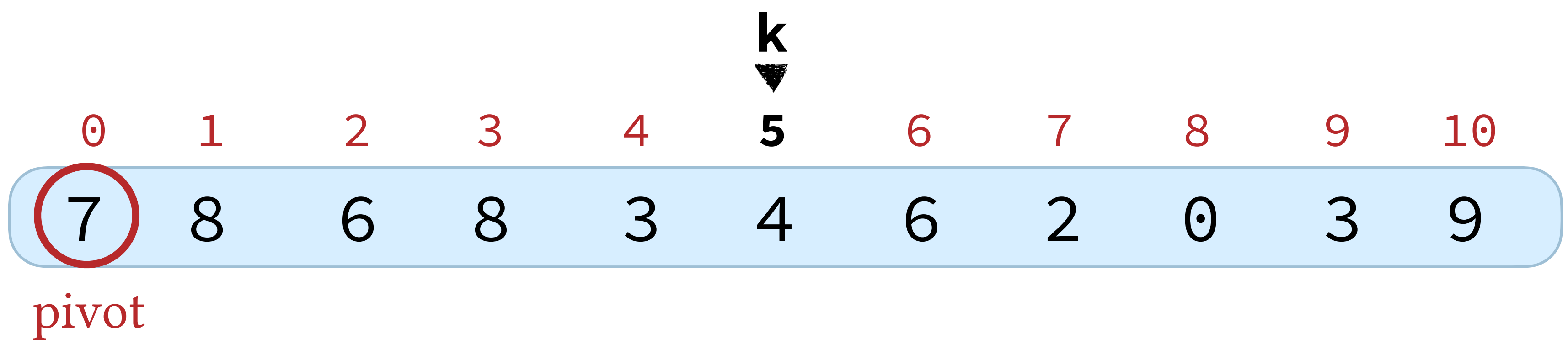
Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).



Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).

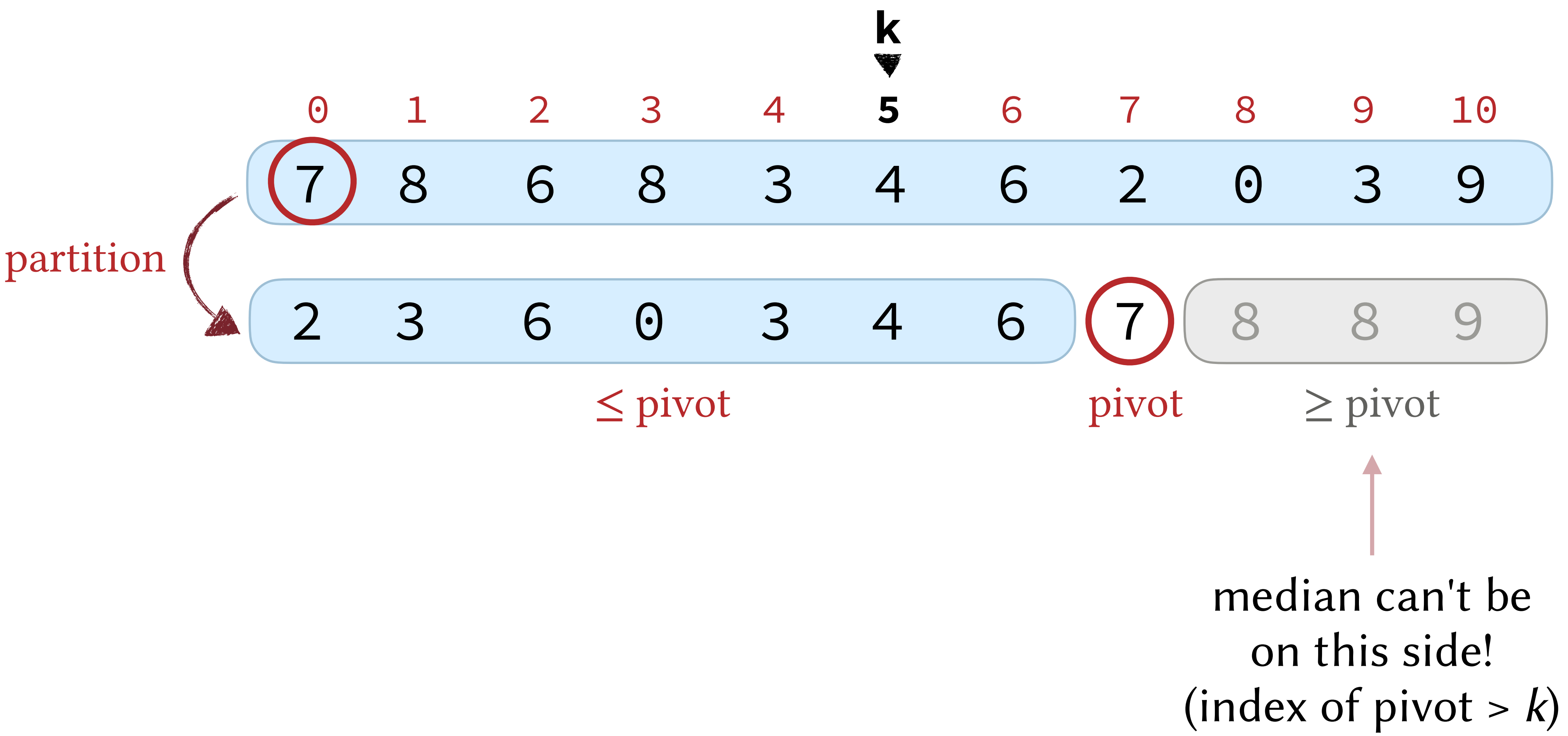


Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).

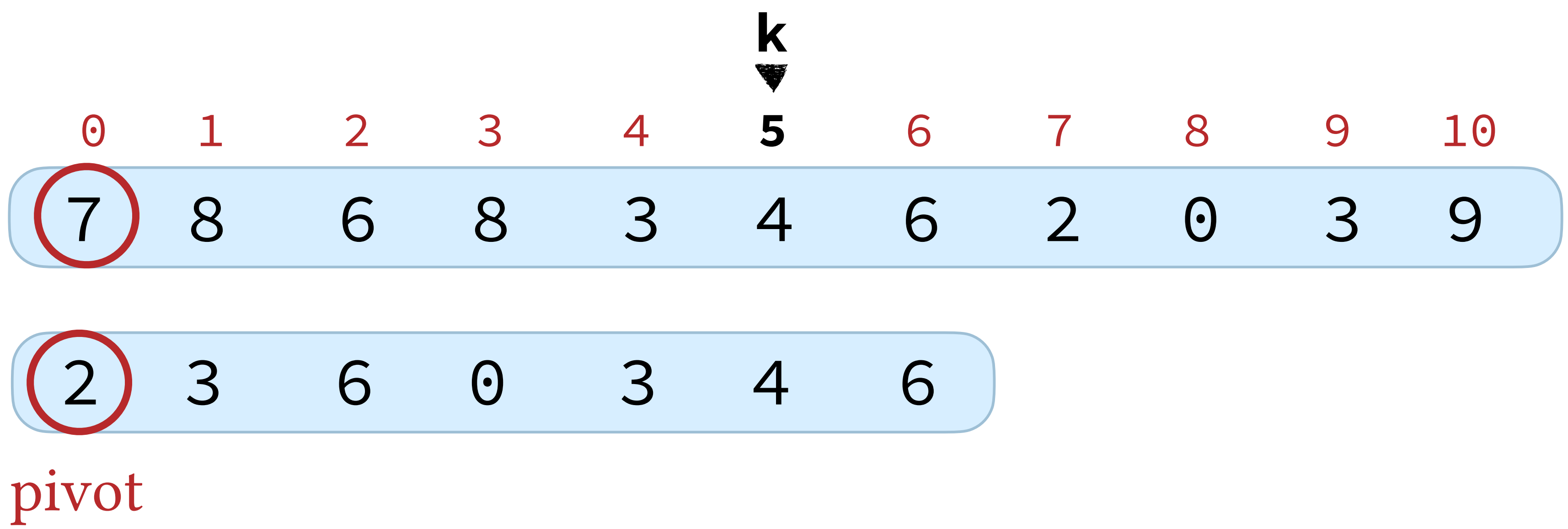
Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).



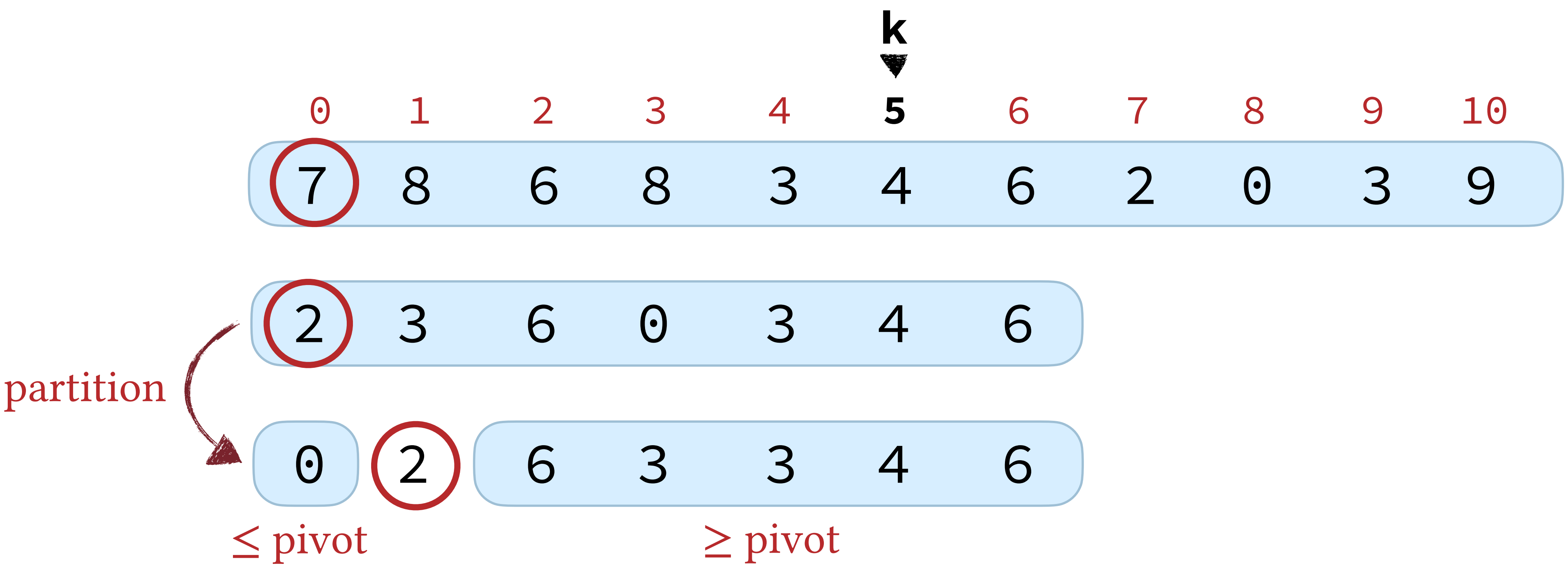
Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).



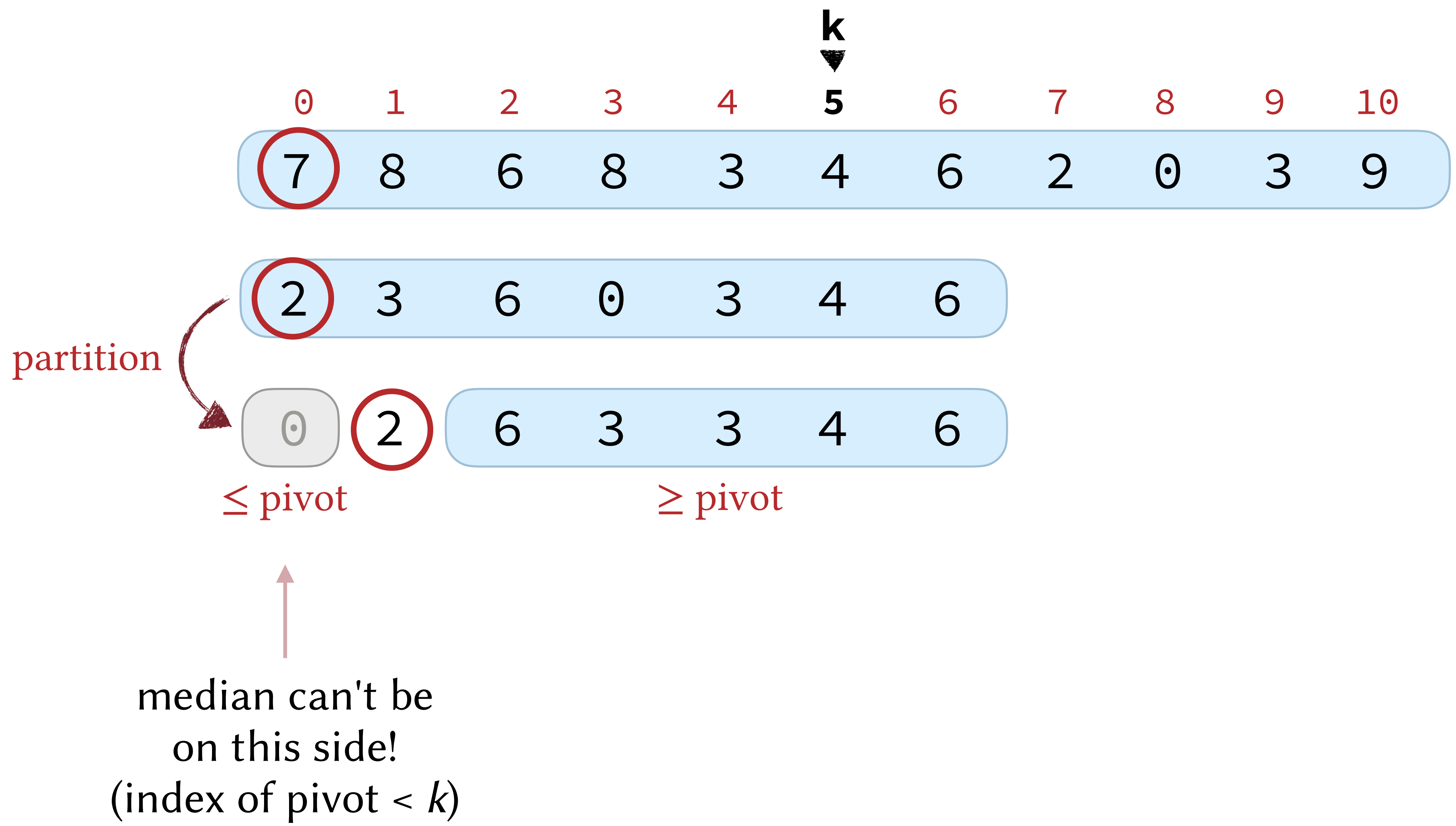
Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).



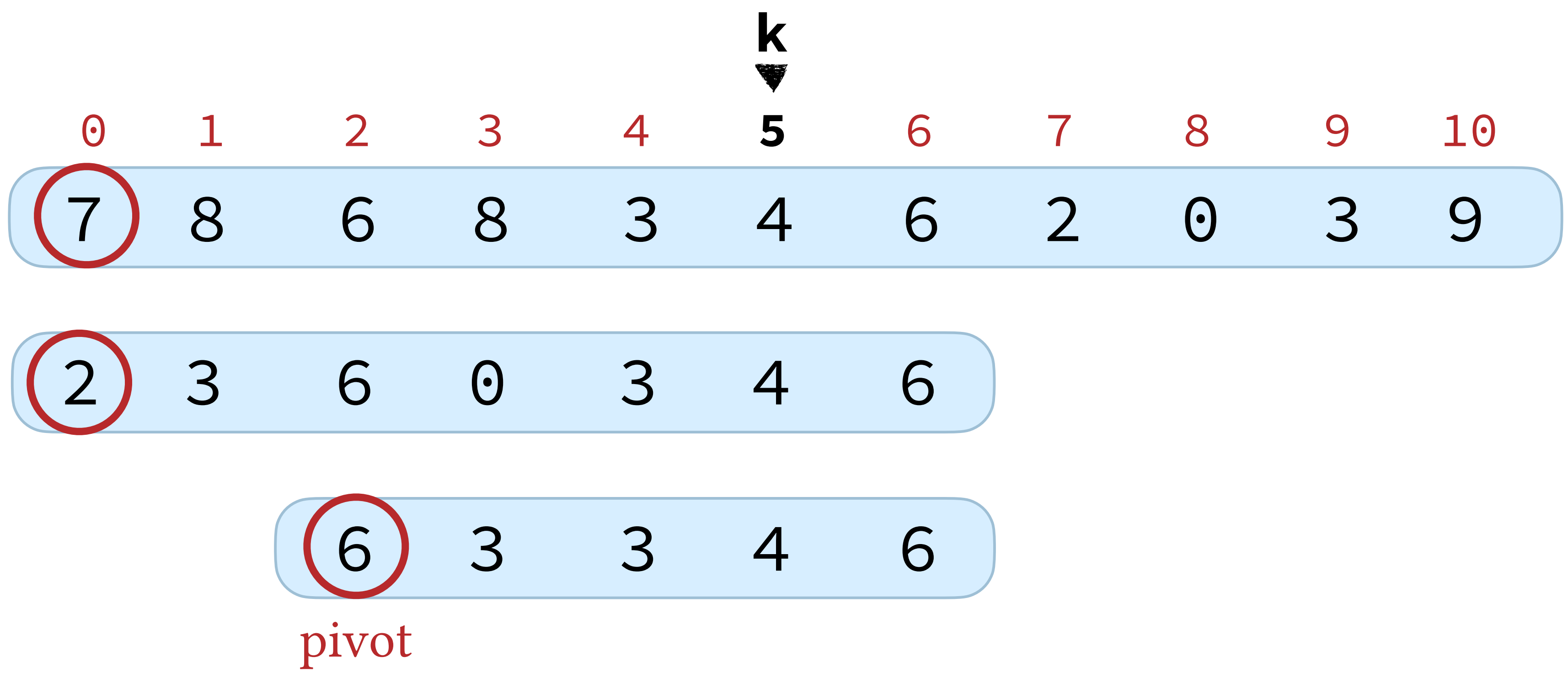
Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).



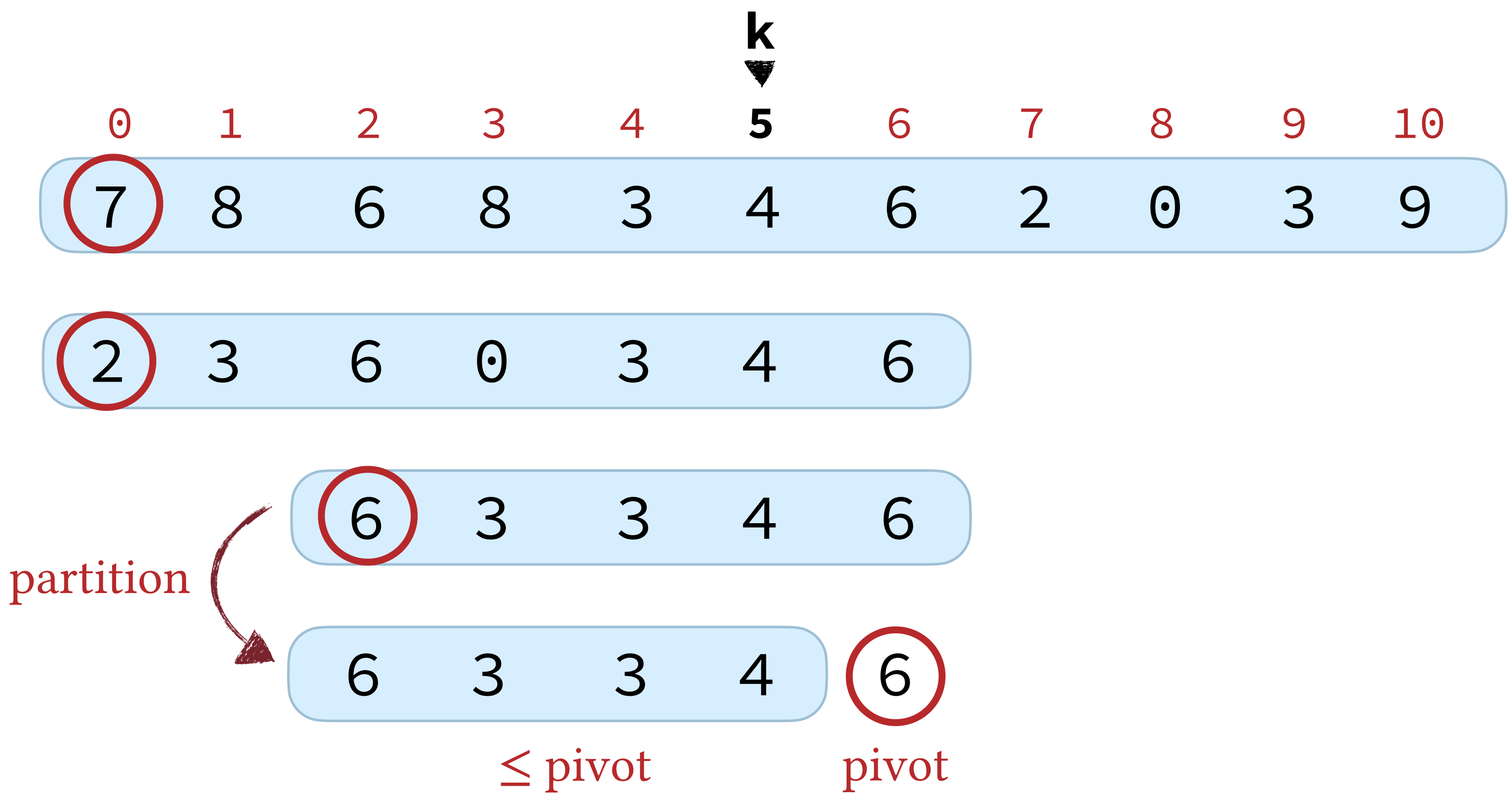
Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).



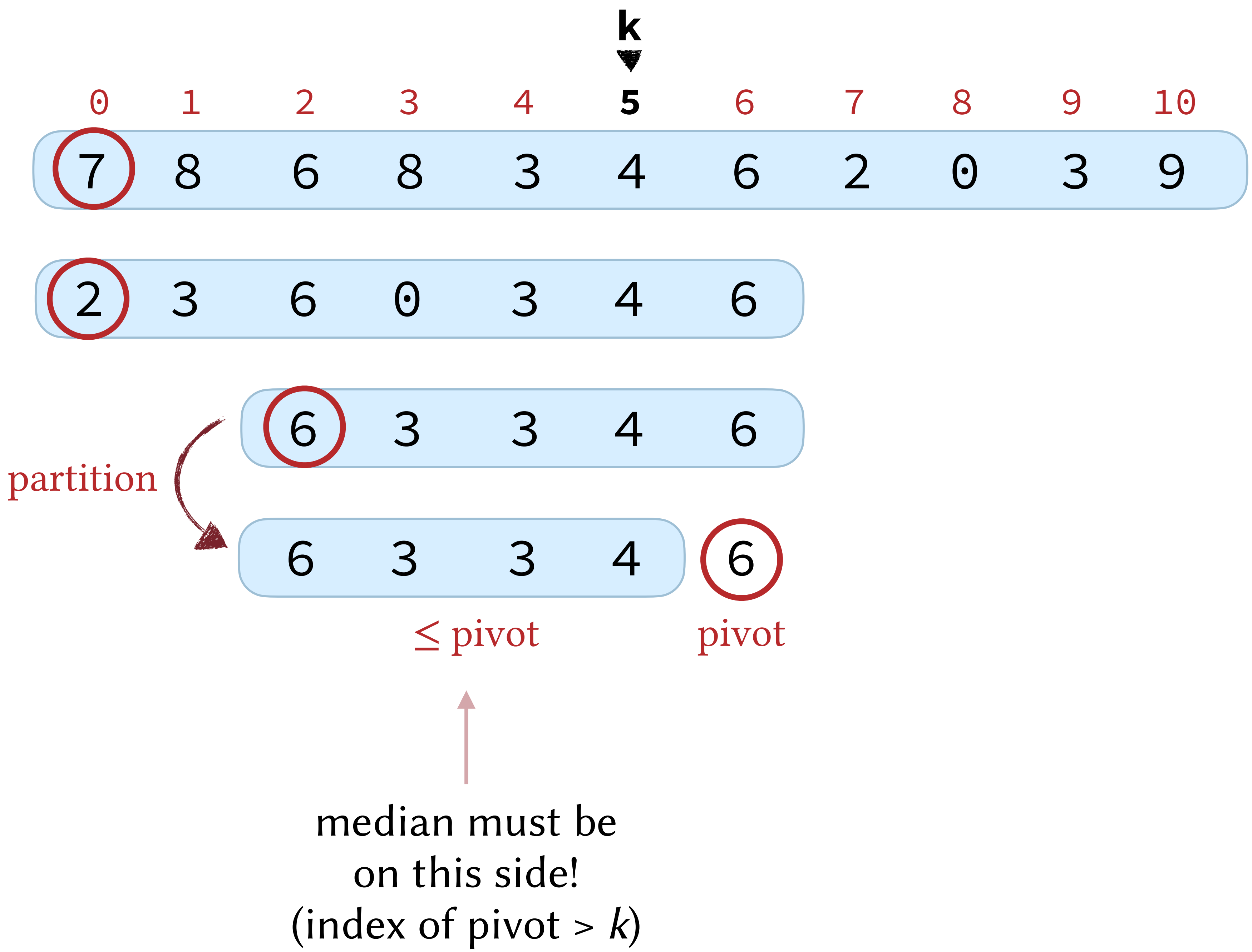
Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).



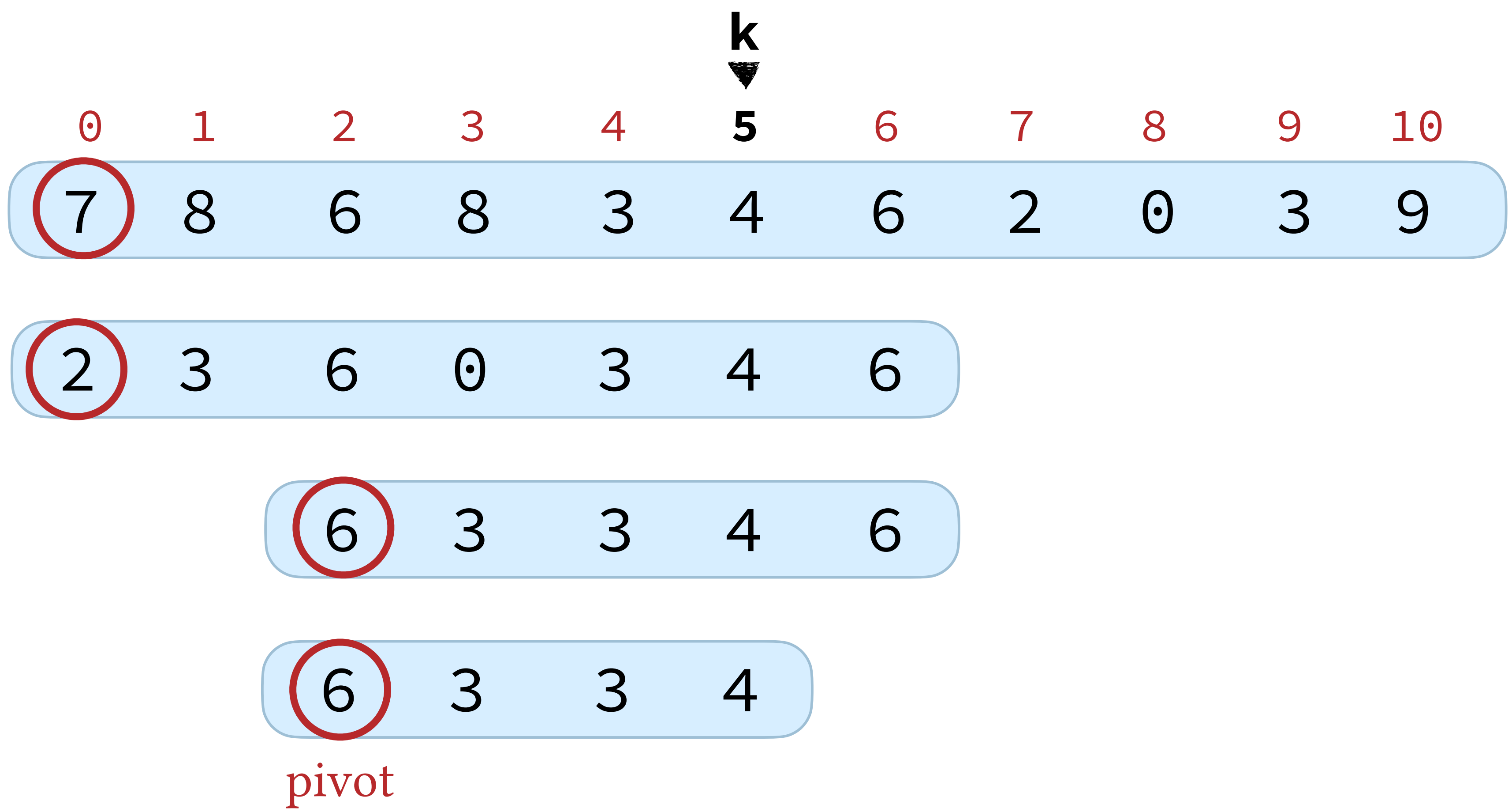
Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).



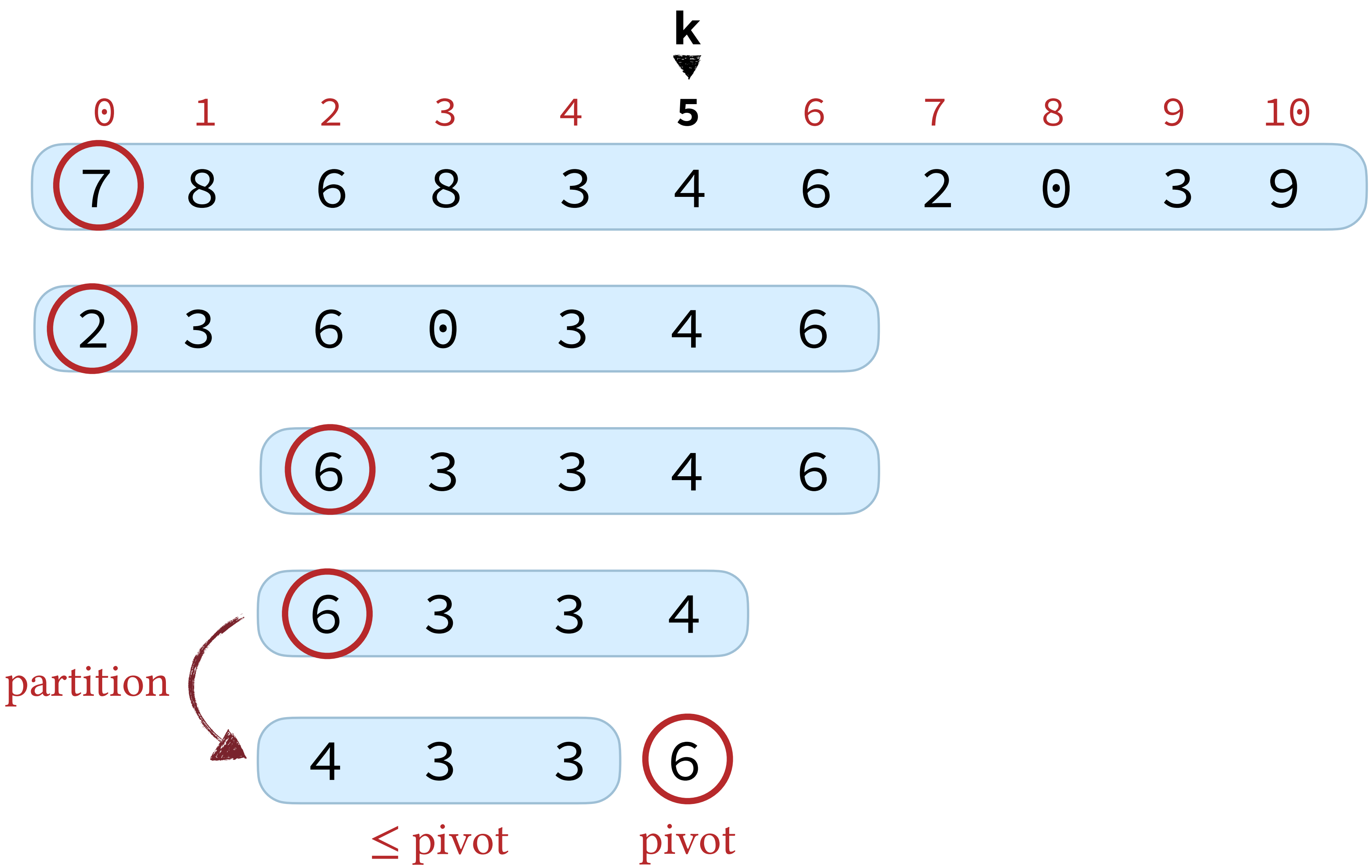
Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).



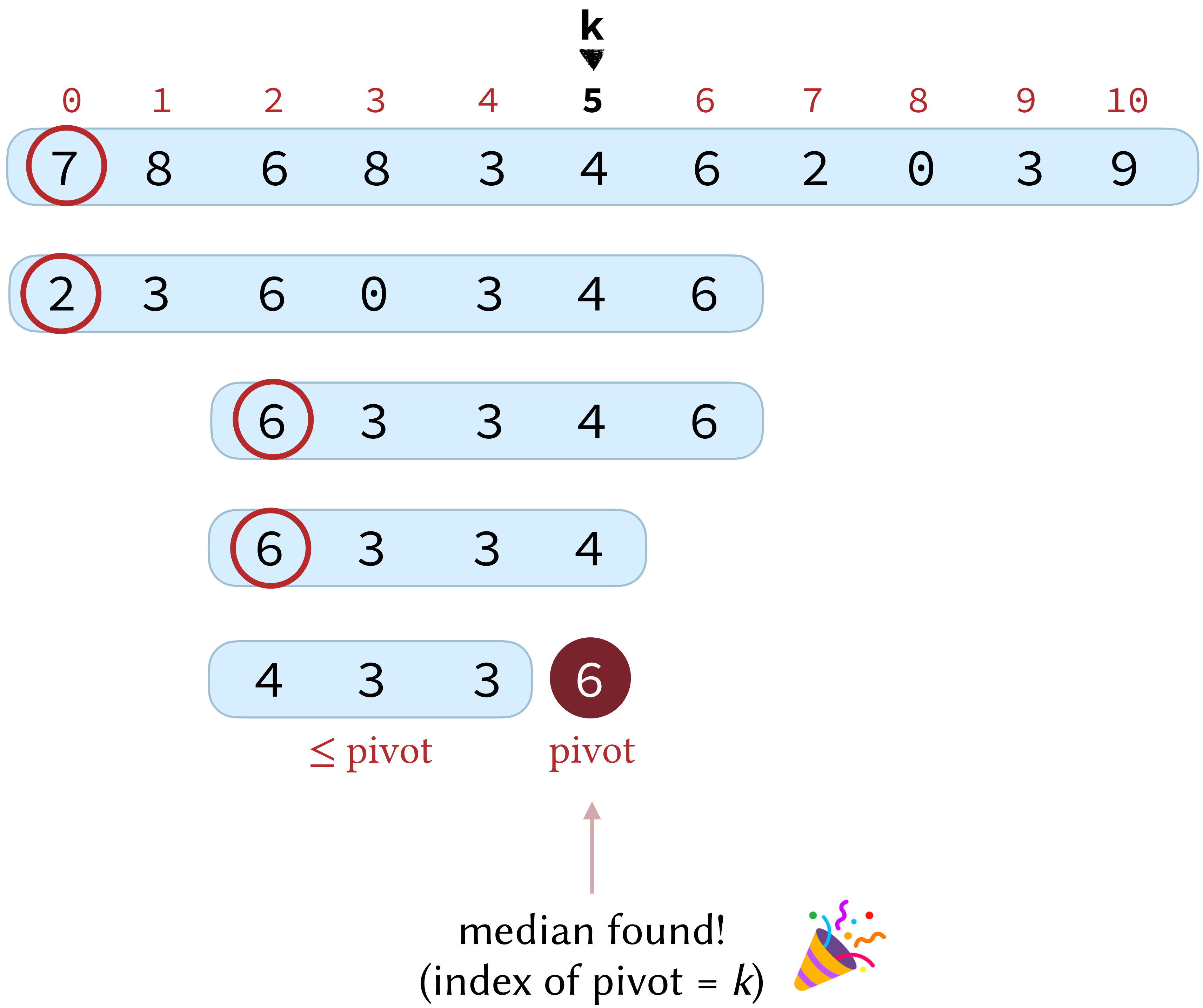
Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).



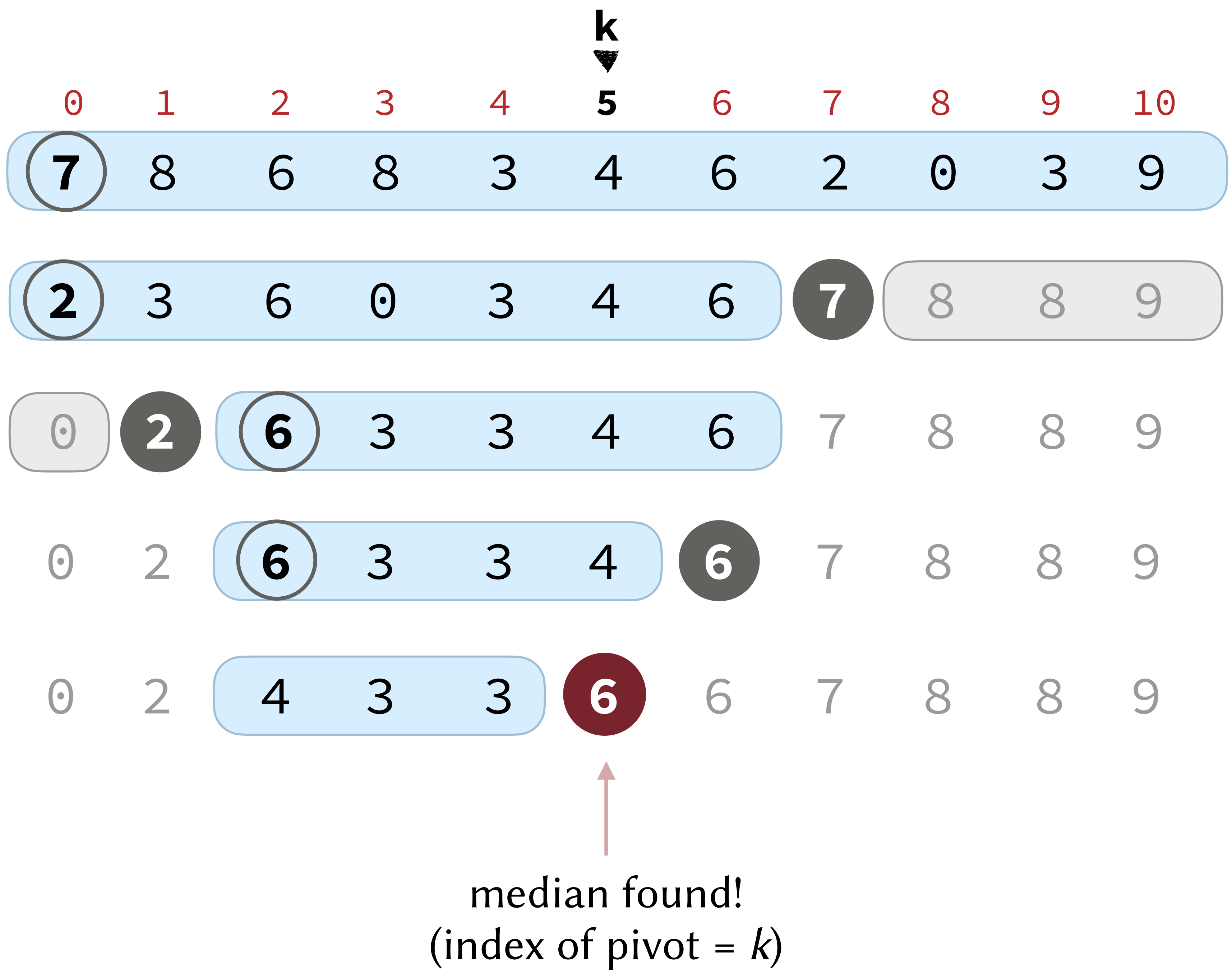
Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).



Quickselect Demo

Assume $k = \frac{n}{2}$ (5 in the example below).



Quickselect Algorithm

```
SELECT(a[], first, last, k)
    SHUFFLE(a, first, last)
    QUICK-SELECT(a, first, last, k)
```

to guard against the worst case (or pick pivot randomly)

assuming *k* is a valid index

Quickselect Algorithm

```
SELECT(a[], first, last, k)
  SHUFFLE(a, first, last)
  QUICK-SELECT(a, first, last, k)
```

```
QUICK-SELECT(a[], first, last, k)
  if (first >= last):
     return a[k]

  p = PARTITION(a, first, last)

  if p == k:
      return a[k]
  if k > p:
      return QUICK-SELECT(a, p+1, last, k)
  else:
      return QUICK-SELECT(a, first, p-1, k)
```

Quickselect Algorithm

```
SELECT(a[], first, last, k)
  SHUFFLE(a, first, last)
  QUICK-SELECT(a, first, last, k)
```

```
QUICK-SELECT(a[], first, last, k)
  if (first >= last):
     return a[k]

  p = PARTITION(a, first, last)

  if p == k:
      return a[k]
  if k > p:
      return QUICK-SELECT(a, p+1, last, k)
  else:
      return QUICK-SELECT(a, first, p-1, k)
```

Quickselect Algorithm

```
SELECT(a[], first, last, k)

  SHUFFLE(a, first, last)
  QUICK-SELECT(a, first, last, k)
```

```
QUICK-SELECT(a[], first, last, k)

  if (first >= last):
     return a[k]

  p = PARTITION(a, first, last)

  if p == k:
      return a[k]
  if k > p:
      return QUICK-SELECT(a, p+1, last, k)
  else:
      return QUICK-SELECT(a, first, p-1, k)
```

Quickselect Algorithm

```
SELECT(a[], first, last, k)
    SHUFFLE(a, first, last)
    QUICK-SELECT(a, first, last, k)
```

```
QUICK-SELECT(a[], first, last, k)
    if (first >= last):
        return a[k]

    p = PARTITION(a, first, last)

    if p == k:
        return a[k]
    if k > p:
        return QUICK-SELECT(a, p+1, last, k)
    else:
        return QUICK-SELECT(a, first, p-1, k)
```

Quickselect Algorithm

```
SELECT(a[], first, last, k)
  SHUFFLE(a, first, last)
  QUICK-SELECT(a, first, last, k)
```

```
QUICK-SELECT(a[], first, last, k)
  if (first >= last):
     return a[k]

  p = PARTITION(a, first, last)

  if p == k:
      return a[k]
  if k > p:
      return QUICK-SELECT(a, p+1, last, k)
  else:
      return QUICK-SELECT(a, first, p-1, k)
```

Quickselect Analysis

Best Case.

Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case.

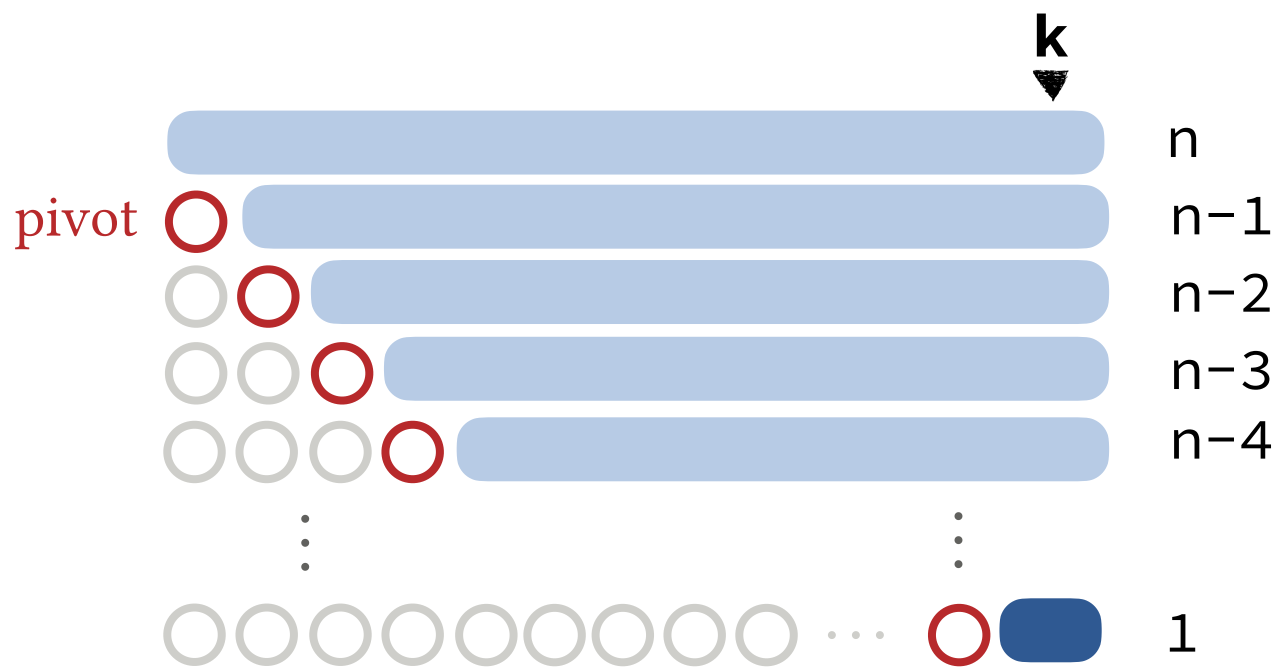
Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n-1$ partitioning steps:

$$n + (n-1) + (n-2) + \ \dots\ + 1 = \Theta(n^2)$$

Example 1. $k = n - 1$



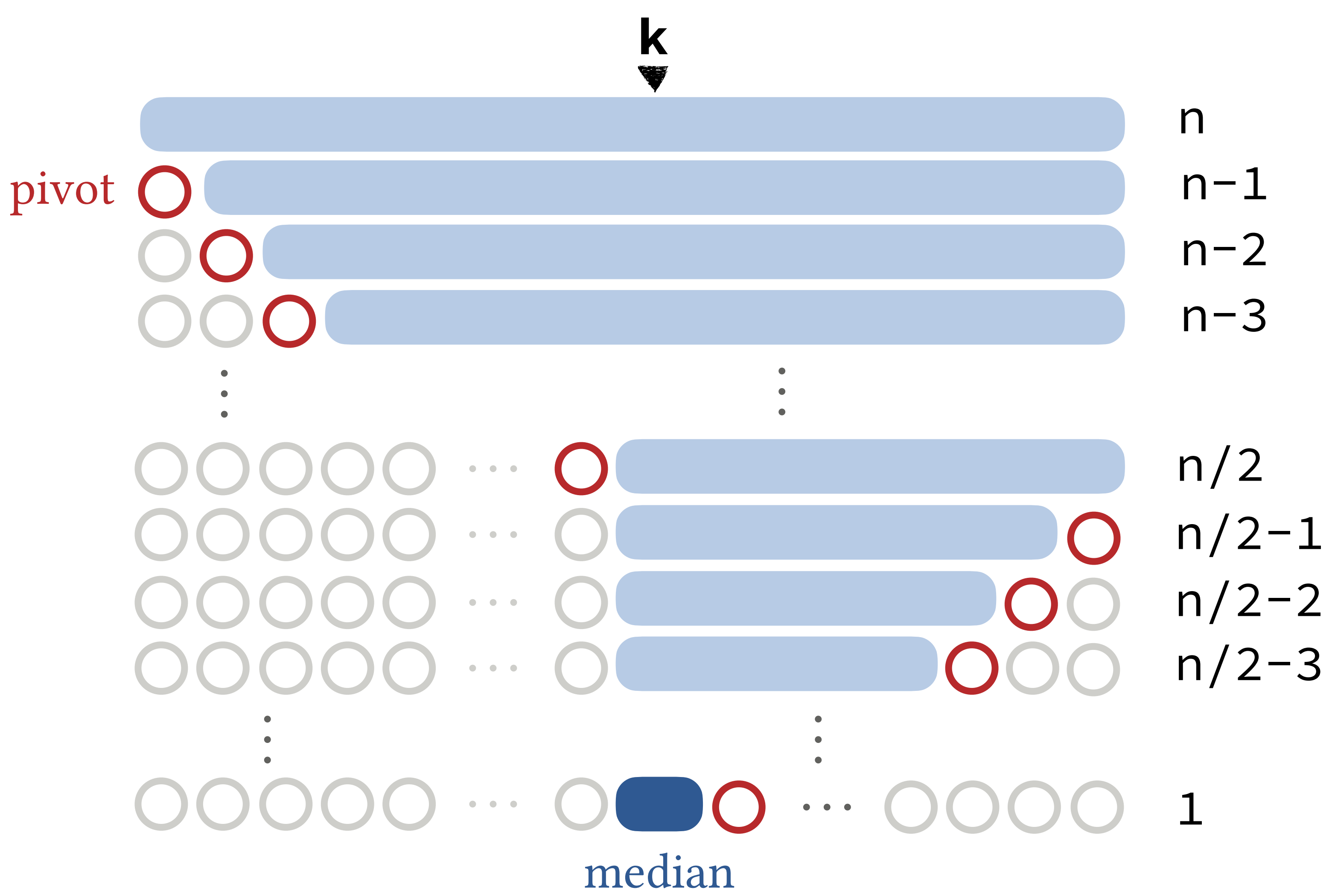
Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n-1$ partitioning steps:

$$n + (n-1) + (n-2) + \ \dots \ + 1 = \Theta(n^2)$$

Example 2. $k = \dfrac{n}{2}$



Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n-1$ partitioning steps:

$$n + (n-1) + (n-2) + \ \ldots \ + 1 = \Theta(n^2)$$

probabilistically almost-impossible if the array is shuffled!

Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n-1$ partitioning steps:

$$n + (n-1) + (n-2) + \ \dots \ + 1 = \Theta(n^2)$$

Expected Case. $\Theta(n)$

Intuition. Partitioning always gets rid of around half of the remaining elements.

$$T(n) = T(\tfrac{n}{2}) + \sim n$$

time to select from an array of size n

Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n-1$ partitioning steps:

$$n + (n-1) + (n-2) + \ \ldots\ + 1 = \Theta(n^2)$$

Expected Case. $\Theta(n)$

Intuition. Partitioning always gets rid of around half of the remaining elements.

$$T(n) = T(\tfrac{n}{2}) + \sim n$$

time to partition an array of size n

time to select from an array of size n

Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n - 1$ partitioning steps:

$$n + (n-1) + (n-2) + \ \dots \ + 1 = \Theta(n^2)$$

Expected Case. $\Theta(n)$

Intuition. Partitioning always gets rid of around half of the remaining elements.

$$T(n) = T(\tfrac{n}{2}) + \sim n$$

time to partition an array of size n

time to select from an array of size n

time to select from an array of size n/2

Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n-1$ partitioning steps:

$$n + (n-1) + (n-2) + \ \dots\ + 1 = \Theta(n^2)$$

Expected Case. $\Theta(n)$

Intuition. Partitioning always gets rid of around half of the remaining elements.

$$T(n) = T(\tfrac{n}{2}) + \sim n$$

n

time to partition
the whole array

Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n-1$ partitioning steps:

$$n + (n-1) + (n-2) + \ \ldots \ + 1 = \Theta(n^2)$$

Expected Case. $\Theta(n)$

Intuition. Partitioning always gets rid of around half of the remaining elements.

$$T(n) = T(\tfrac{n}{2}) + \sim n$$

$$n + \frac{n}{2}$$

↑

time to partition
half the array

Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n-1$ partitioning steps:

$$n + (n-1) + (n-2) + \ldots + 1 = \Theta(n^2)$$

Expected Case. $\Theta(n)$

Intuition. Partitioning always gets rid of around half of the remaining elements.

$$T(n) = T(\tfrac{n}{2}) + \sim n$$

$$n + \frac{n}{2} + \frac{n}{4}$$

time to partition
a quarter of the array

Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n - 1$ partitioning steps:

$$n + (n-1) + (n-2) + \ \dots\ + 1 = \Theta(n^2)$$

Expected Case. $\Theta(n)$

Intuition. Partitioning always gets rid of around half of the remaining elements.

$$T(n) = T(\tfrac{n}{2}) + \sim n$$

$$n + \frac{n}{2} + \frac{n}{4} + \dots + 1$$

Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n - 1$ partitioning steps:

$$n + (n-1) + (n-2) + \ \dots \ + 1 = \Theta(n^2)$$

Expected Case. $\Theta(n)$

Intuition. Partitioning always gets rid of around half of the remaining elements.

$$T(n) = T(\tfrac{n}{2}) + \sim n$$

$$n + \frac{n}{2} + \frac{n}{4} + \dots + 1 = n(1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{n})$$

Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n-1$ partitioning steps:

$$n + (n-1) + (n-2) + \ \ldots\ + 1 = \Theta(n^2)$$

Expected Case. $\Theta(n)$

Intuition. Partitioning always gets rid of around half of the remaining elements.

$$T(n) = T(\tfrac{n}{2}) + \sim n$$

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 = n(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{n}) = \Theta(n)$$

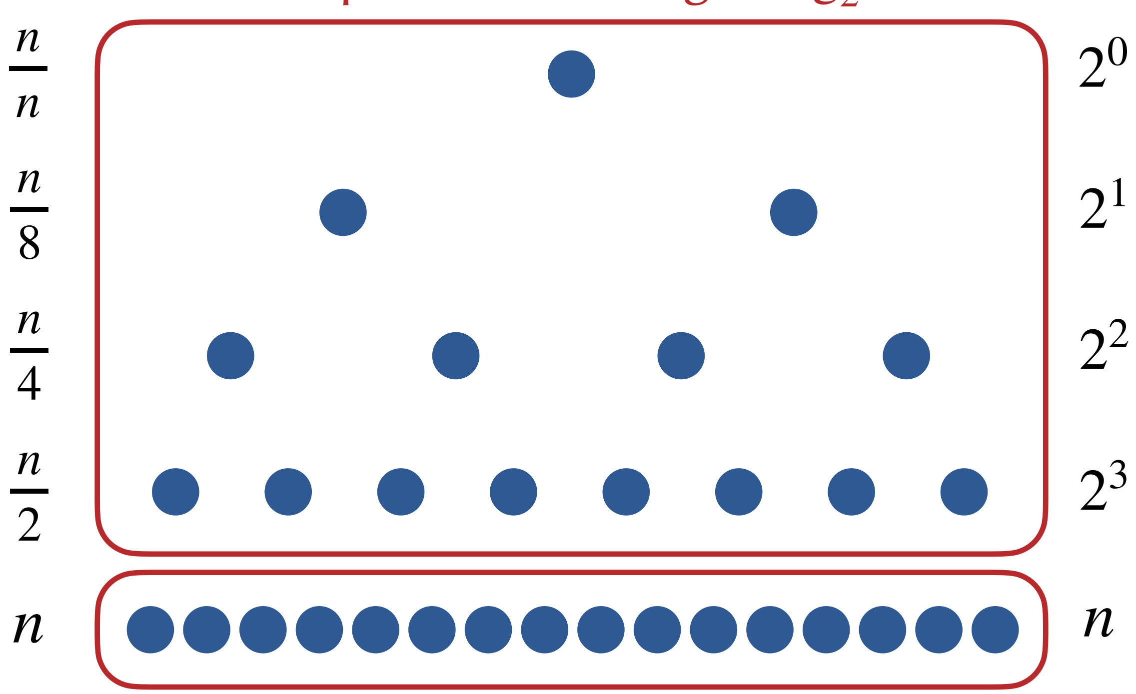
Remember!

$$\sum_{i=0}^{\log_2 n} 2^i = 2^{\log_2 n + 1} - 1 = 2n - 1$$

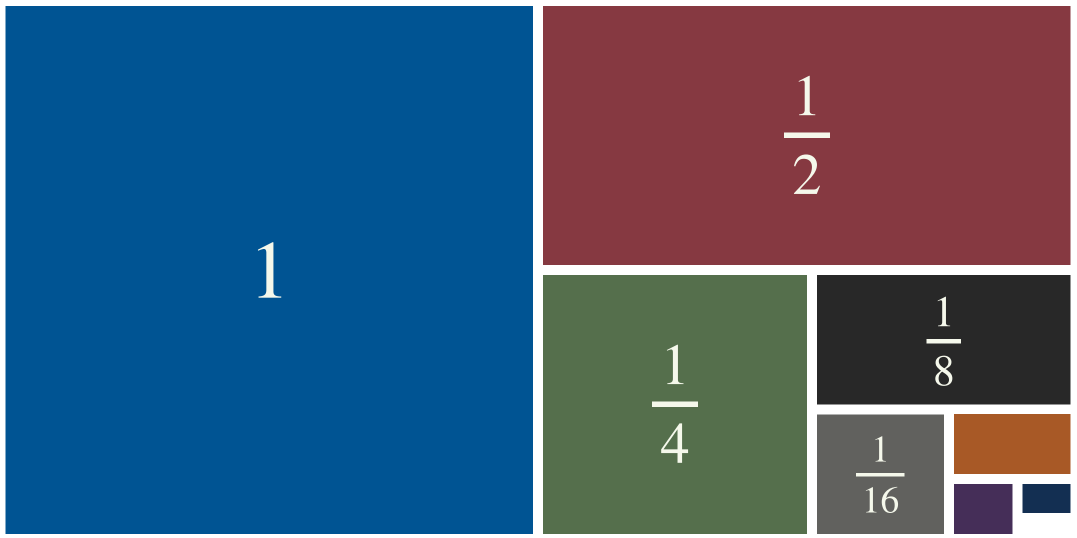
$$1 + 2 + 4 + 8 + \ldots + n$$
$$= 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{\log_2 n}$$

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 4 + 2 + 1$$
$$n \times (1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{n})$$

n -1 internal nodes in a complete tree of height $\log_2 n$



n leaves in a complete tree of height $\log_2 n$



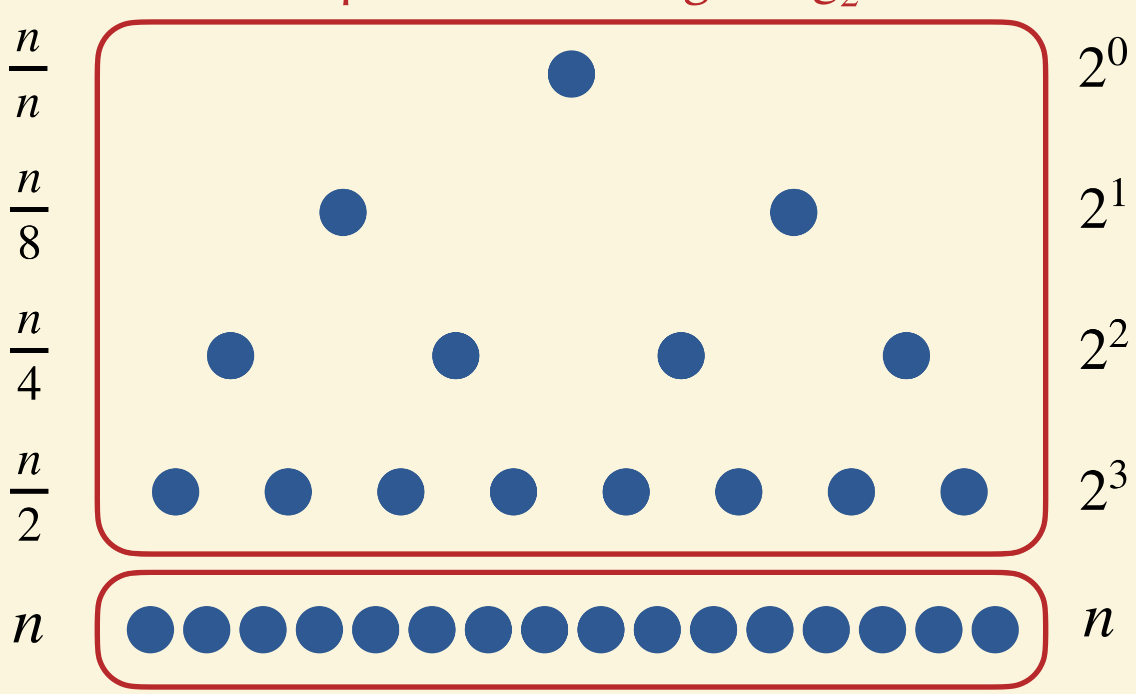
≤ 2

Remember!

$$\sum_{i=0}^{\log_2 n} 2^i = 2^{\log_2 n + 1} - 1 = 2n - 1$$

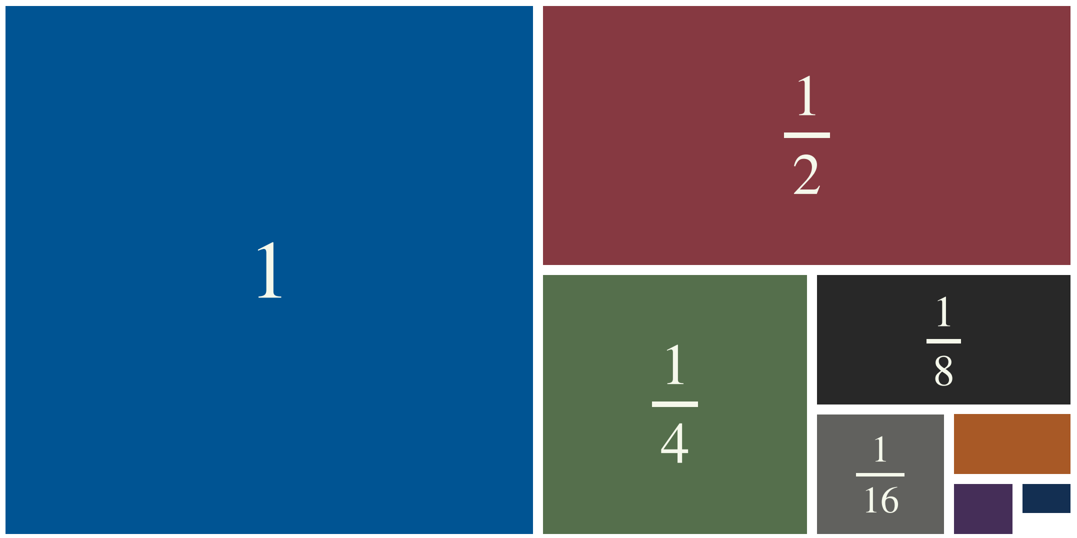
$$1 + 2 + 4 + 8 + \ldots + n$$
$$= 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{\log_2 n}$$

n -1 internal nodes in a complete tree of height $\log_2 n$



n leaves in a complete tree of height $\log_2 n$

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 4 + 2 + 1$$
$$n \times (1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{n})$$



≤ 2

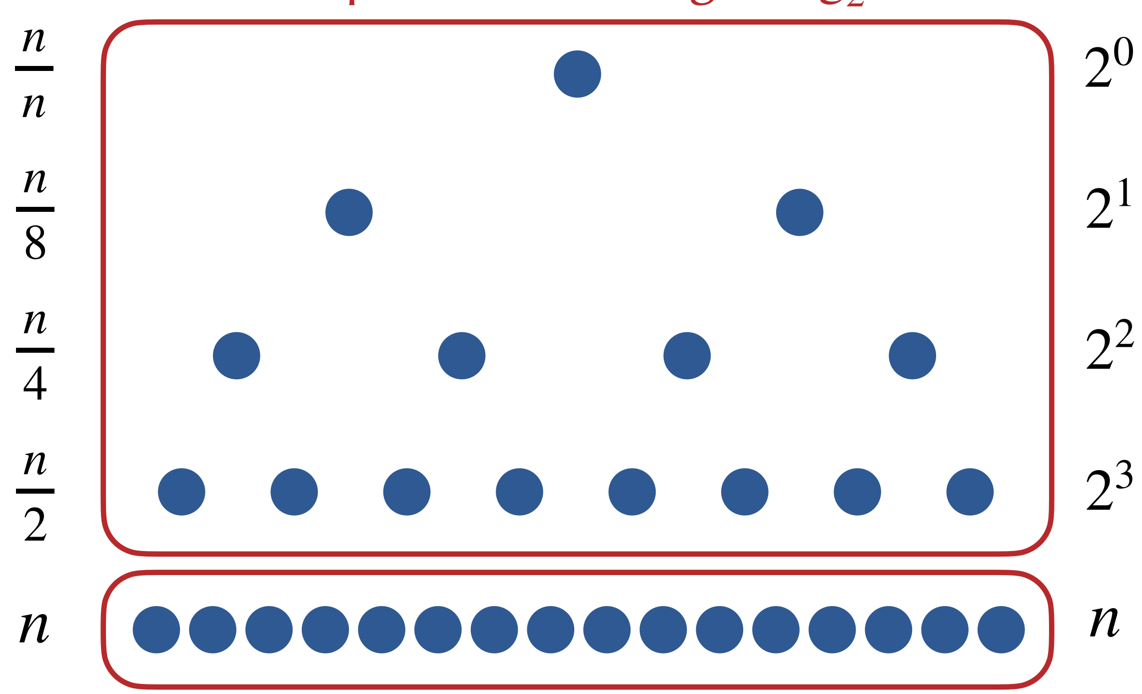
Remember!

$$\sum_{i=0}^{\log_2 n} 2^i = 2^{\log_2 n + 1} - 1 = 2n - 1$$

$$1 + 2 + 4 + 8 + \ldots + n$$

$$= 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{\log_2 n}$$

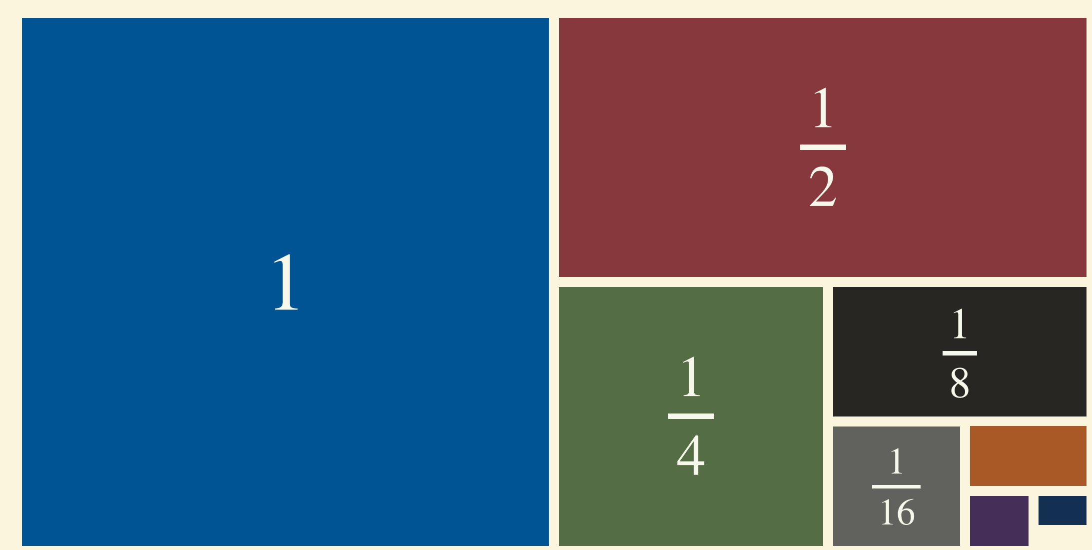
n -1 internal nodes in a complete tree of height $\log_2 n$



n leaves in a complete tree of height $\log_2 n$

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 4 + 2 + 1$$

$$n \times \left(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{n}\right)$$



$$\leq 2$$

Analysis Notes

Remember: Code that follows the pattern below has a running time of $\Theta(n \log n)$

```
foo(n)
  if (n == 0): return
  foo(n / 2)
  foo(n / 2)
  linear(n)
```

solve *two* subproblems of half the size.

do a *linear* amount of work.

Remember: Code that follows the pattern below has a running time of $\Theta(n)$

```
foo(n)
  if (n == 0): return
  foo(n / 2)
  linear(n)
```

solve *one* subproblems of half the size.

do a *linear* amount of work.

Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n-1$ partitioning steps:

$$n + (n-1) + (n-2) + \ \dots \ + 1 = \Theta(n^2)$$

Expected Case. $\Theta(n)$

Intuition. Partitioning always gets rid of around half of the remaining elements.

$$T(n) = T(\tfrac{n}{2}) + \Theta(n)$$

$$n + \frac{n}{2} + \frac{n}{4} + \dots + 1 = \Theta(n)$$

Can we do better?

Is selection as hard as sorting?
(requires $\sim n \log n$ compares
in the worst case if $k = \frac{n}{2}$)

Quickselect Analysis

Best Case. Element at rank k found immediately after the first partitioning step: $\Theta(n)$.

Worst Case. Element at rank k found after $n-1$ partitioning steps:

$$n + (n-1) + (n-2) + \ \dots\ + 1 = \Theta(n^2)$$

Expected Case. $\Theta(n)$

Intuition. Partitioning always gets rid of around half of the remaining elements.

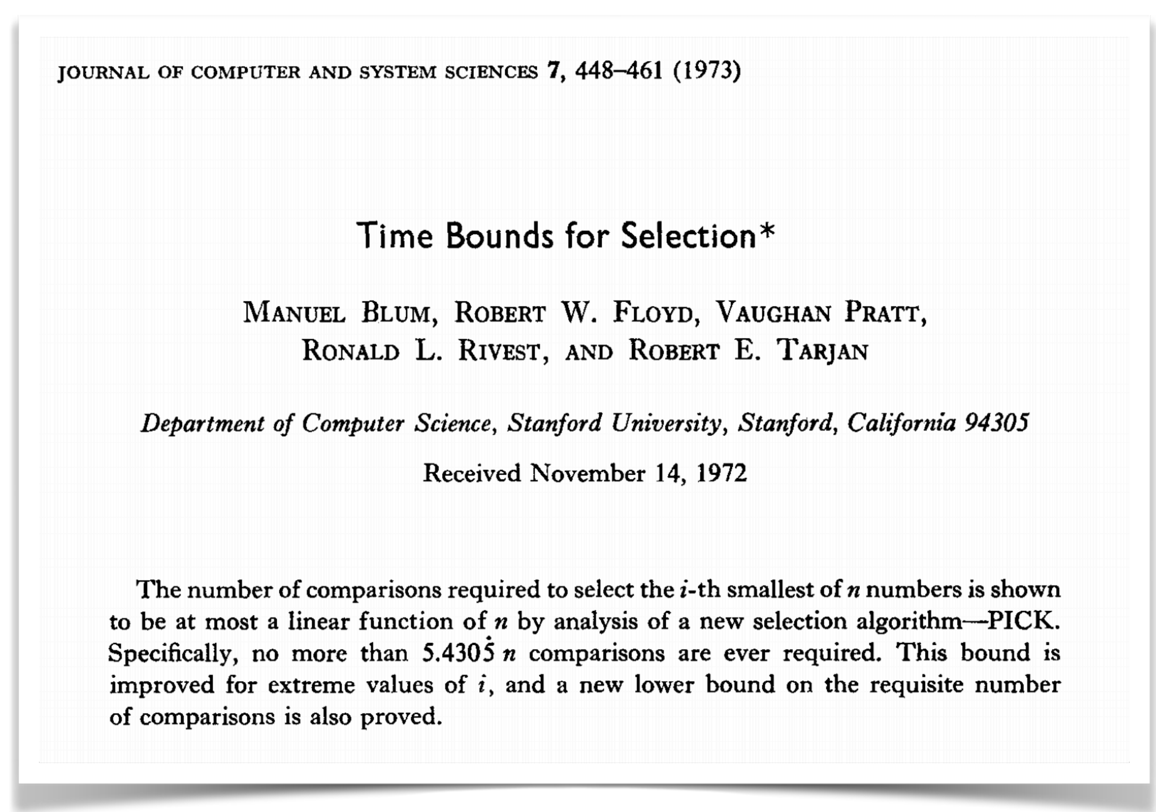
$$T(n) = T(\tfrac{n}{2}) + \Theta(n)$$

$$n + \frac{n}{2} + \frac{n}{4} + \dots + 1 = \Theta(n)$$

Can we do better?

Theoretically. Selection can be done in linear time in the worst case using the Median of Medians algorithm. (Blum, Floyd, Pratt, Rivest, and Tarjan 1973).

Practically. Quickselect is faster in practice.


JOURNAL OF COMPUTER AND SYSTEM SCIENCES **7**, 448–461 (1973)

Time Bounds for Selection*

MANUEL BLUM, ROBERT W. FLOYD, VAUGHAN PRATT, RONALD L. RIVEST, AND ROBERT E. TARJAN

Department of Computer Science, Stanford University, Stanford, California 94305

Received November 14, 1972

The number of comparisons required to select the i-th smallest of n numbers is shown to be at most a linear function of n by analysis of a new selection algorithm—PICK. Specifically, no more than $5.4305\, n$ comparisons are ever required. This bound is improved for extreme values of i, and a new lower bound on the requisite number of comparisons is also proved.

Article Talk Read Edit View history Search Wikipedia

Introselect

From Wikipedia, the free encyclopedia

In computer science, **introselect** (short for "introspective selection") is a selection algorithm that is a hybrid of quickselect and median of medians which has fast average performance and optimal worst-case performance. Introselect is related to the introsort sorting algorithm: these are analogous refinements of the basic quickselect and quicksort algorithms, in that they both start with the quick algorithm, which has good average performance and low overhead, but fall back to an optimal worst-case algorithm (with higher overhead) if the quick algorithm does not progress rapidly enough. Both algorithms were introduced by David Musser in (Musser 1997), with the purpose of providing generic algorithms for the C++ Standard Library that have both fast average performance and optimal worst-case performance, thus allowing the performance requirements to be tightened.[1] However, in most C++ Standard Library implementations that use introselect, another "introselect" algorithm is used, which combines quickselect and heapselect, and has a worst-case running time of $O(n \log n)$[2].

Introselect

Class	**Selection algorithm**
Data structure	Array
Worst-case performance	$O(n)$
Best-case performance	$O(n)$

Quicksort Improvement

What is the order of growth of the running time of quicksort is if a linear time median finding algorithm is used to pick the pivot?

Quicksort Improvement

What is the order of growth of the running time of quicksort is if a linear time median finding algorithm is used to pick the pivot?

Answer. If the pivot is always the median, the array is always split into almost equally-sized partitions. Therefore, the algorithm would run in $\Theta(n \log n)$.

Quicksort Improvement

What is the order of growth of the running time of quicksort is if a linear time median finding algorithm is used to pick the pivot?

Answer. If the pivot is always the median, the array is always split into almost equally-sized partitions. Therefore, the algorithm would run in $\Theta(n \log n)$.

However: the overhead for finding the pivot would be high and the algorithm would be slower in practice compared to just picking the pivot randomly.

Selection (special cases)

- Finding the largest (or smallest) element: $n - 1$ compares.
  Perform a linear scan in the array.

Selection (special cases)

- Finding the largest (or smallest) element: $n - 1$ compares.
  Perform a linear scan in the array.

- Finding the largest and smallest elements:
  - $2n - 3$ compares.
    Perform a linear scan to find the largest ($n - 1$) and then another scan on the elements (excluding the largest) to find the minimum ($n - 2$).

Selection (special cases)

- Finding the largest (or smallest) element: $n - 1$ compares.
  Perform a linear scan in the array.

- Finding the largest and smallest elements:
  - $2n - 3$ compares.
    Perform a linear scan to find the largest ($n - 1$) and then another scan on the elements (excluding the largest) to find the minimum ($n - 2$).
  - **Can we do better?**
    We can use some of the information gained while finding the maximum to find the minimum!

Selection (special cases)

- Finding the largest (or smallest) element: $n - 1$ compares.
  Perform a linear scan in the array.

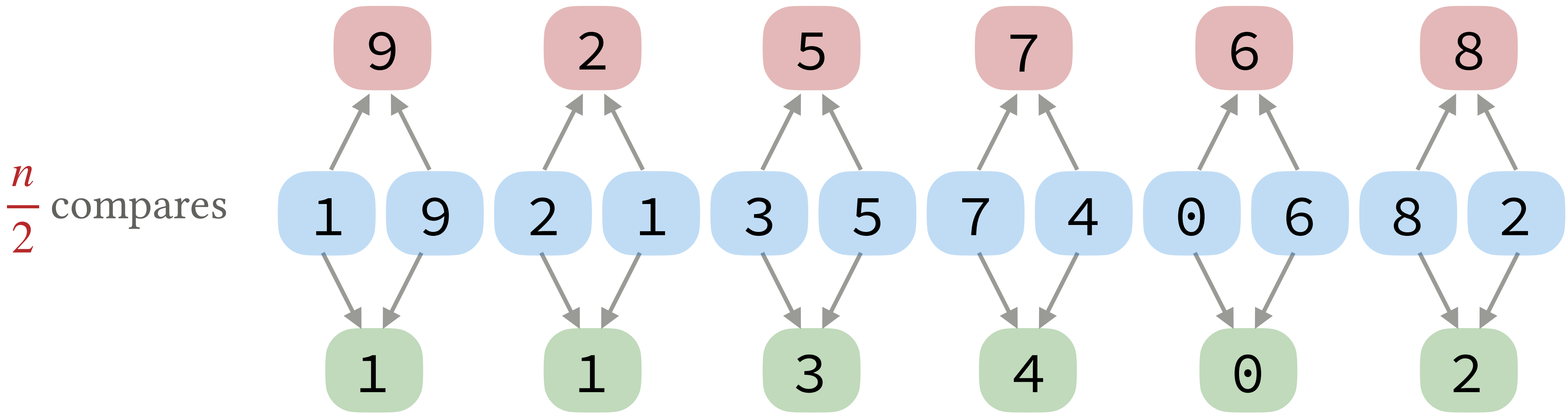
- Finding the largest and smallest elements:
  - $2n - 3$ compares.
    Perform a linear scan to find the largest ($n - 1$) and then another scan on the elements (excluding the largest) to find the minimum ($n - 2$).
  - **Can we do better?**
    We can use some of the information gained while finding the maximum to find the minimum!

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

Selection (special cases)

- Finding the largest (or smallest) element: $n-1$ compares.
  Perform a linear scan in the array.

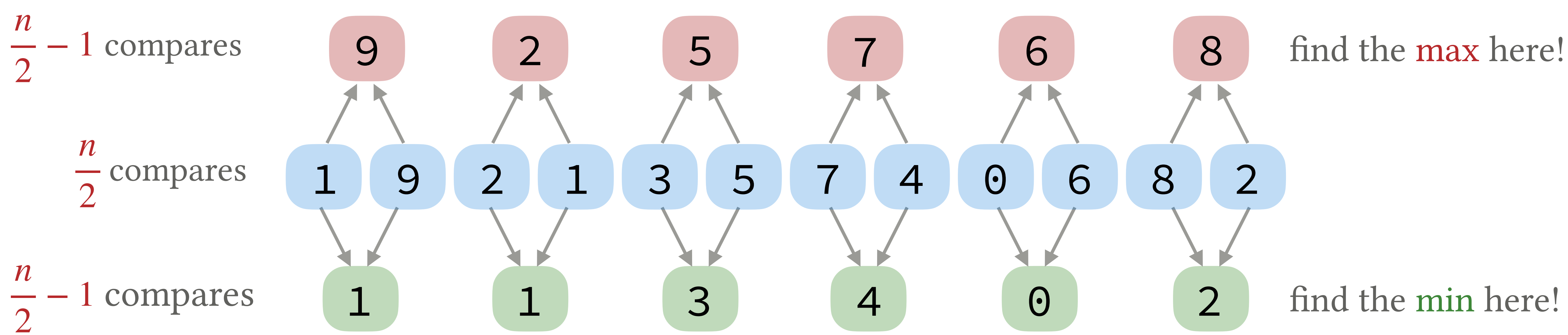
- Finding the largest and smallest elements:
  - $2n-3$ compares.
    Perform a linear scan to find the largest ($n-1$) and then another scan on the elements (excluding the largest) to find the minimum ($n-2$).
  - **Can we do better?**
    We can use some of the information gained while finding the maximum to find the minimum!



Selection (special cases)

- Finding the largest (or smallest) element: $n - 1$ compares.
  Perform a linear scan in the array.

- Finding the largest and smallest elements:
  - $2n - 3$ compares.
    Perform a linear scan to find the largest ($n - 1$) and then another scan on the elements (excluding the largest) to find the minimum ($n - 2$).
  - **Can we do better?**
    We can use some of the information gained while finding the maximum to find the minimum!



Selection (special cases)

- Finding the largest (or smallest) element: $n - 1$ compares.
  Perform a linear scan in the array.

- Finding the largest and smallest elements:
  - $2n - 3$ compares.
    Perform a linear scan to find the largest ($n - 1$) and then another scan on the elements (excluding the largest) to find the minimum ($n - 2$).
  - **Can we do better?**
    We can use some of the information gained while finding the maximum to find the minimum!

$\frac{n}{2} - 1$ compares

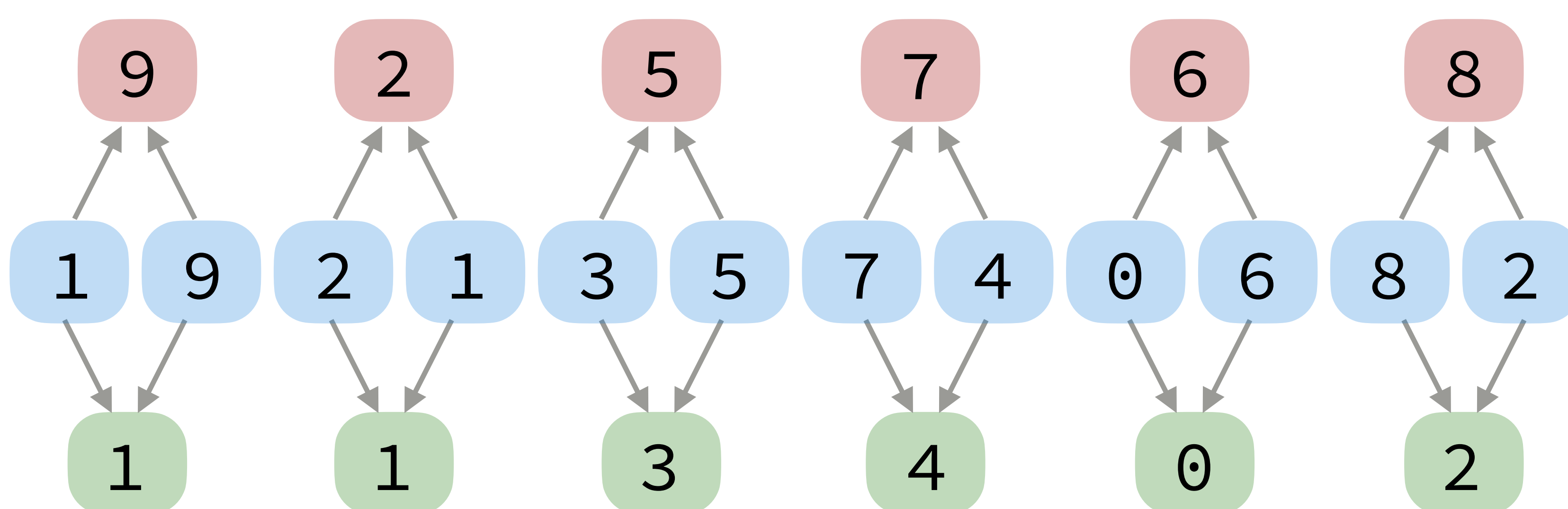
$+$

$\frac{n}{2}$ compares

$+$

$\frac{n}{2} - 1$ compares

$\frac{3}{2}n - 2$ compares



find the max here!

find the min here!

Selection (special cases)

- Finding the largest (or smallest) element: $n-1$ compares.
  Perform a linear scan in the array.

- Finding the largest and smallest elements:
  - $2n-3$ compares.
    Perform a linear scan to find the largest ($n-1$) and then another scan on the elements (excluding the largest) to find the minimum ($n-2$).
  - **Can we do better?**

  Theorem. Any comparison-based algorithm for finding both the minimum and maximum elements in an arbitrary array must perform at least $\frac{3}{2}n-2$ compares in the worst case*.

* for the **proof**, see Computer Algorithms: Introduction to Design and Analysis for Sara Baase and Allen Van Gelder

Selection (special cases)

- Finding the largest (or smallest) element: $n - 1$ compares.
  Perform a linear scan in the array.

- Finding the largest and smallest elements:
  - $2n - 3$ compares.
    Perform a linear scan to find the largest ($n - 1$) and then another scan on the elements (excluding the largest) to find the minimum ($n - 2$).
  - **Can we do better?**

> **Theorem.** Any comparison-based algorithm for finding both the minimum and maximum elements in an arbitrary array must perform at least $\frac{3}{2}n - 2$ compares in the worst case*.

- Finding the 2^{nd} largest element:
  - $2n - 3$ compares.
    Perform a linear scan to find the largest ($n - 1$) and then another scan on the elements (excluding the largest) to find the second largest ($n - 2$).

* for the **proof**, see Computer Algorithms: Introduction to Design and Analysis for Sara Baase and Allen Van Gelder

Selection (special cases)

- Finding the largest (or smallest) element: $n - 1$ compares.
  Perform a linear scan in the array.

- Finding the largest and smallest elements:
  - $2n - 3$ compares.
    Perform a linear scan to find the largest $(n - 1)$ and then another scan on the elements (excluding the largest) to find the minimum $(n - 2)$.
  - **Can we do better?**

  Theorem. Any comparison-based algorithm for finding both the minimum and maximum elements in an arbitrary array must perform at least $\frac{3}{2}n - 2$ compares in the worst case*.

- Finding the 2^{nd} largest element:
  - $2n - 3$ compares.
    Perform a linear scan to find the largest $(n - 1)$ and then another scan on the elements (excluding the largest) to find the second largest $(n - 2)$.
  - **Can we do better?**
    We can use some of the information gained while finding the largest to find the second largest!

* for the **proof**, see Computer Algorithms: Introduction to Design and Analysis for Sara Baase and Allen Van Gelder

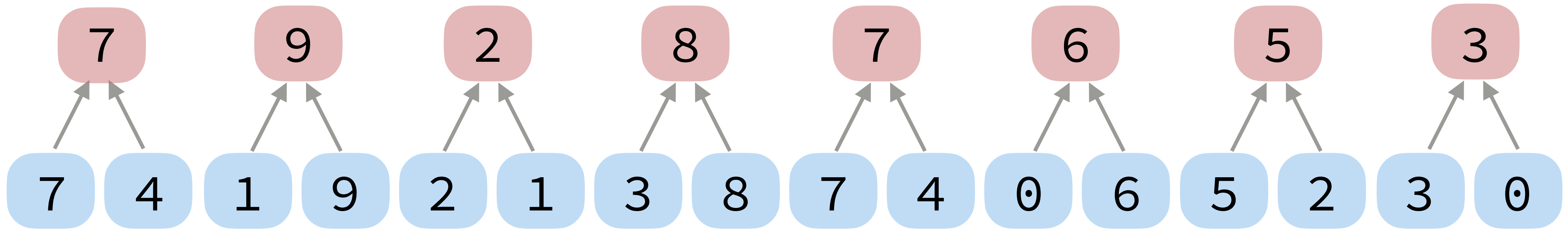
Selecting the 2nd Largest Element

A Tournament-based approach.

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

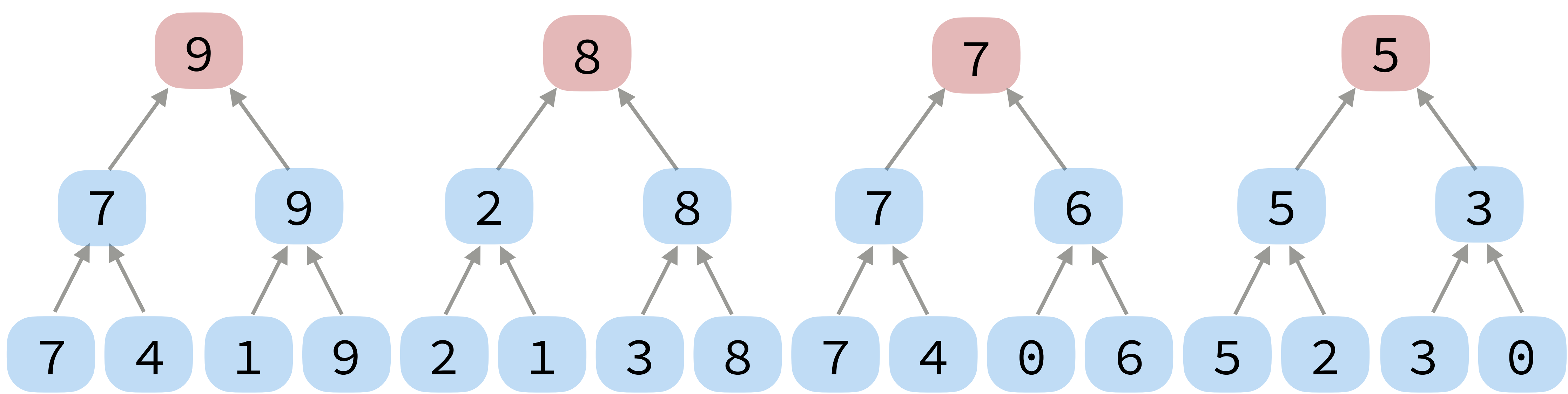
Selecting the 2nd Largest Element

A Tournament-based approach.



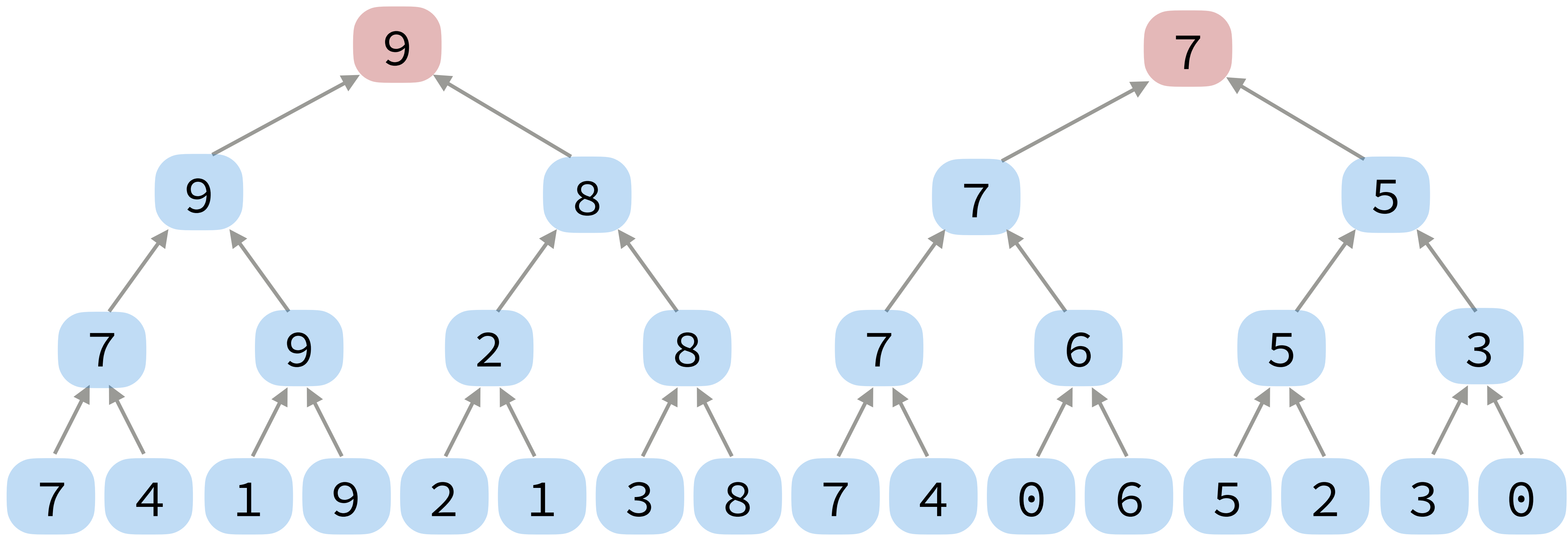
Selecting the 2nd Largest Element

A Tournament-based approach.



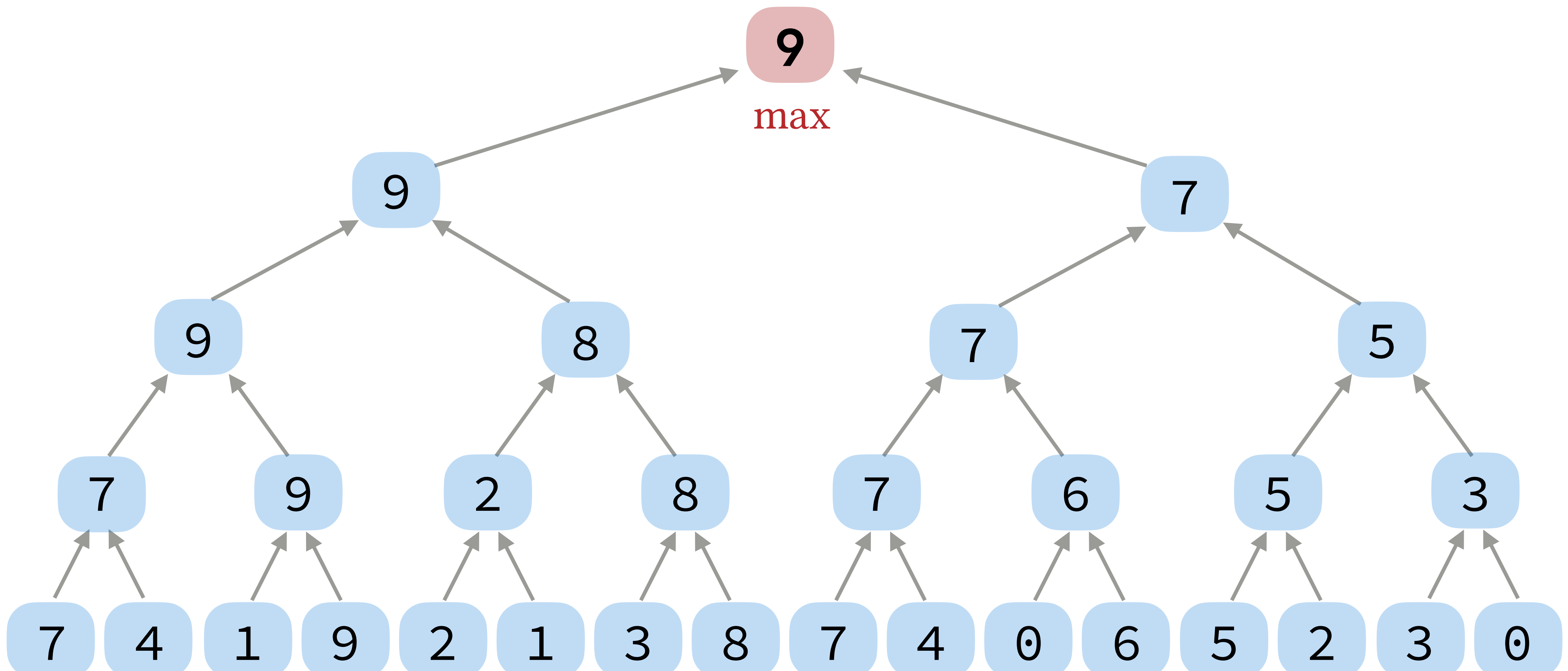
Selecting the 2nd Largest Element

A Tournament-based approach.



Selecting the 2nd Largest Element

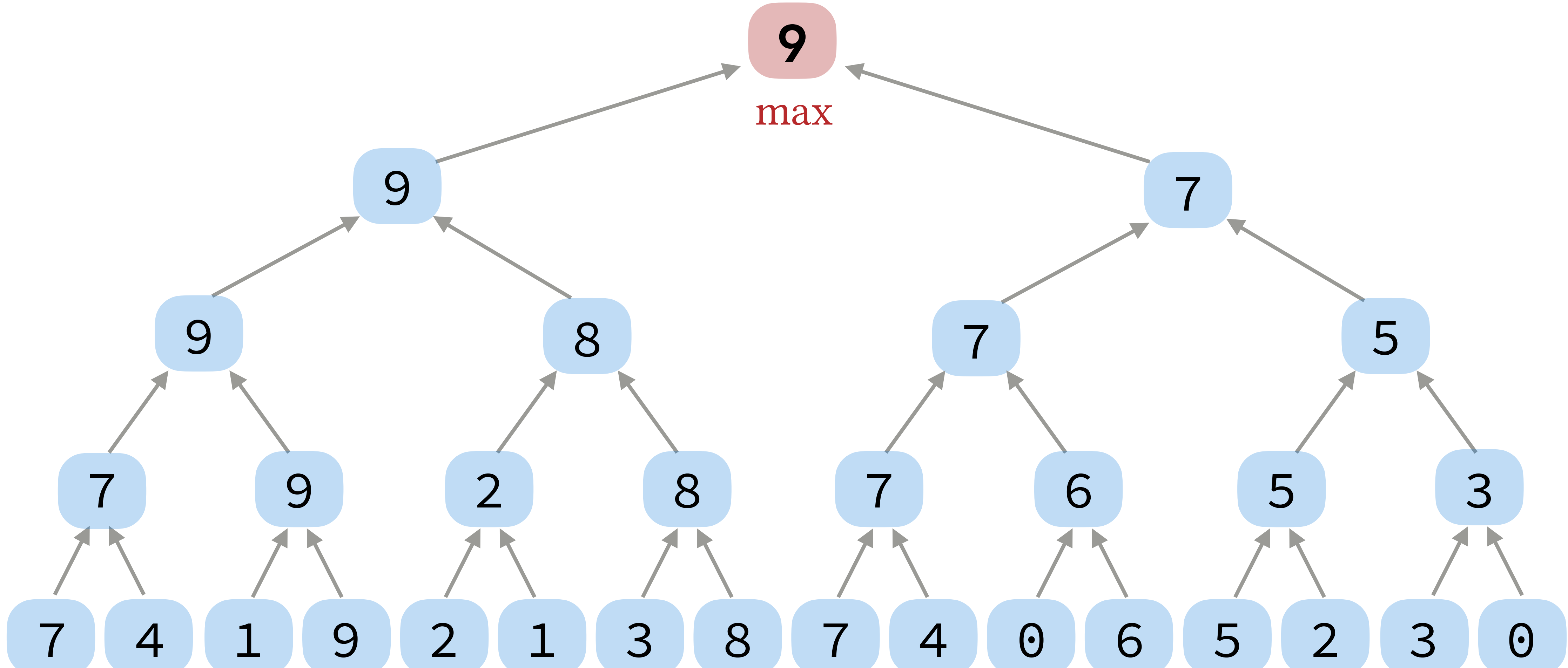
A Tournament-based approach.



Selecting the 2nd Largest Element

A Tournament-based approach.

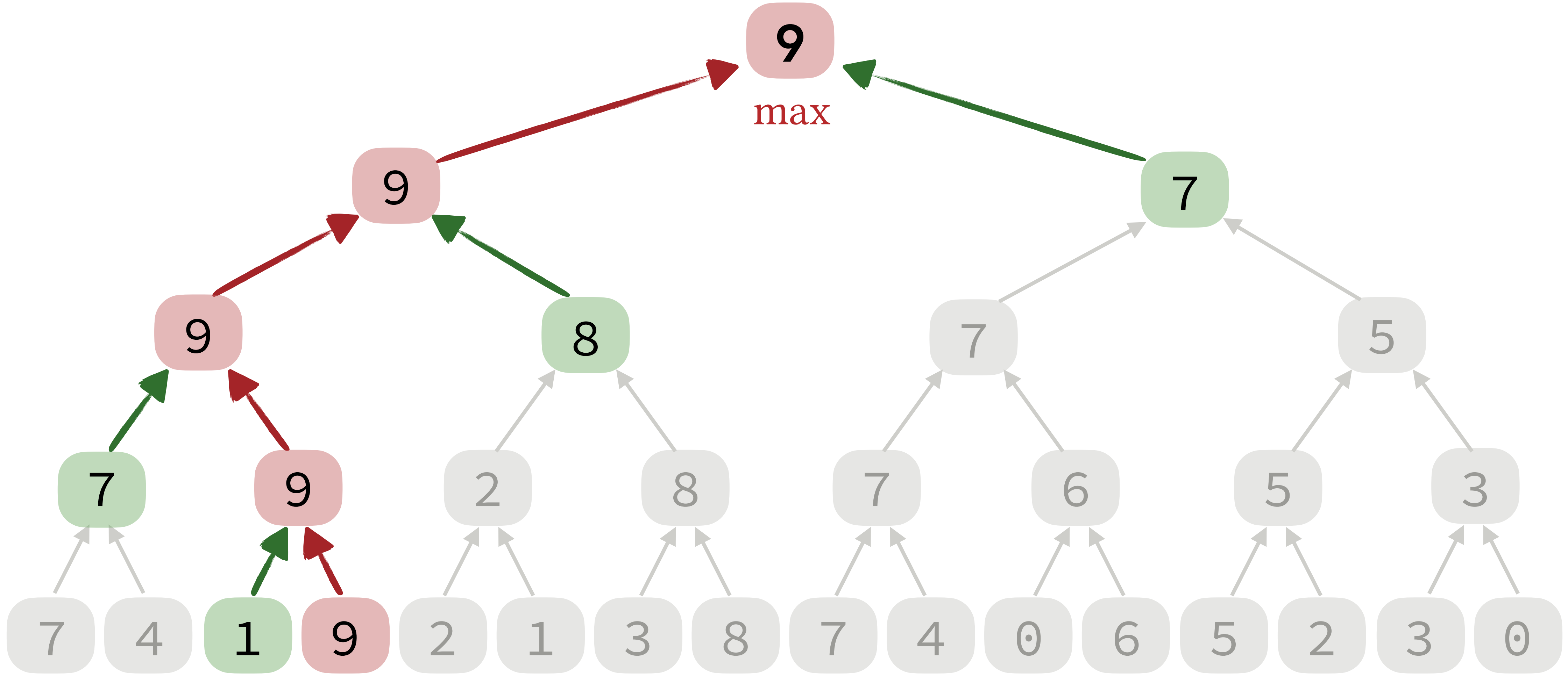
Observation. A key that loses to a key other than the max can't possibly be the second largest.
There are at least two keys larger than that key: the max and the key it lost to!



Selecting the 2nd Largest Element

A Tournament-based approach.

Observation. A key that loses to a key other than the max can't possibly be the second largest.
There are at least two keys larger than that key: the max and the key it lost to!



The second largest must be on this path!
(must have lost to the max)

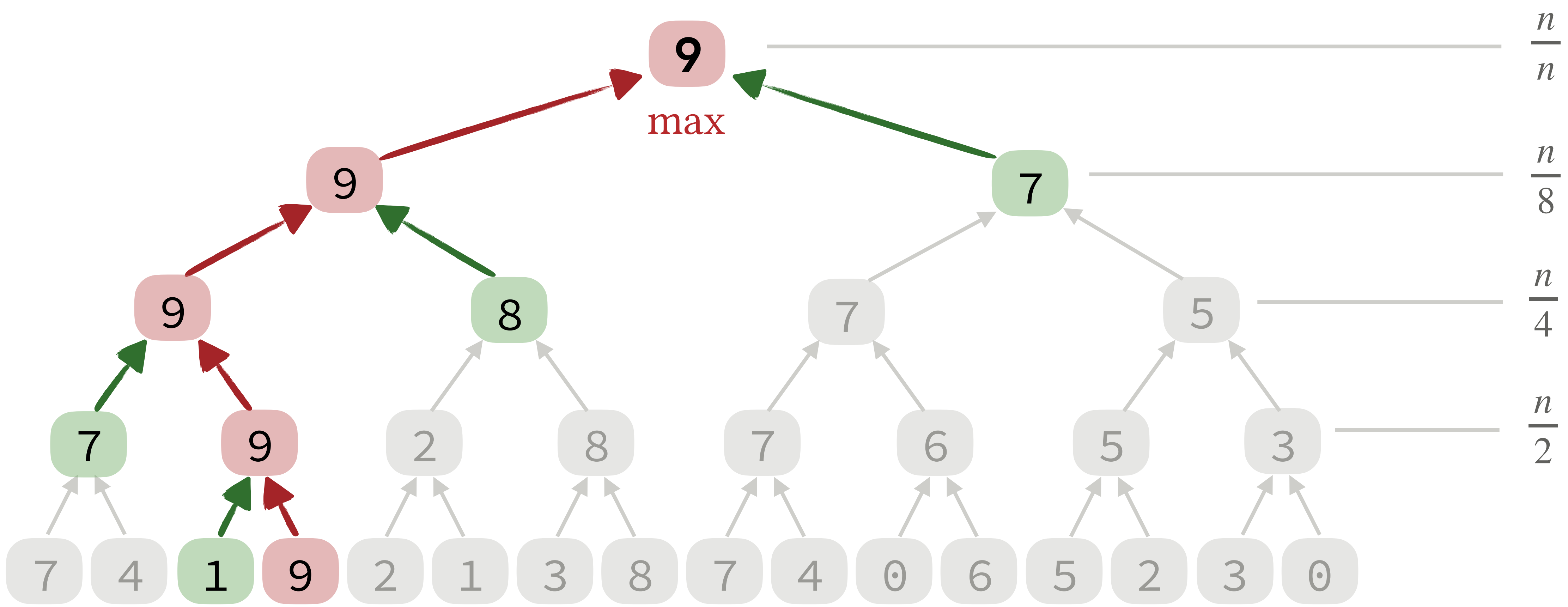
Selecting the 2nd Largest Element

A Tournament-based approach.

Observation. Keys that lose to keys other than the max can't possibly be the second largest.
Proof by contradiction.

Analysis. (assuming n is a power of 2)

- The algorithm performs $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ \dots\ + 1 = n - 1$ compares to find the max.



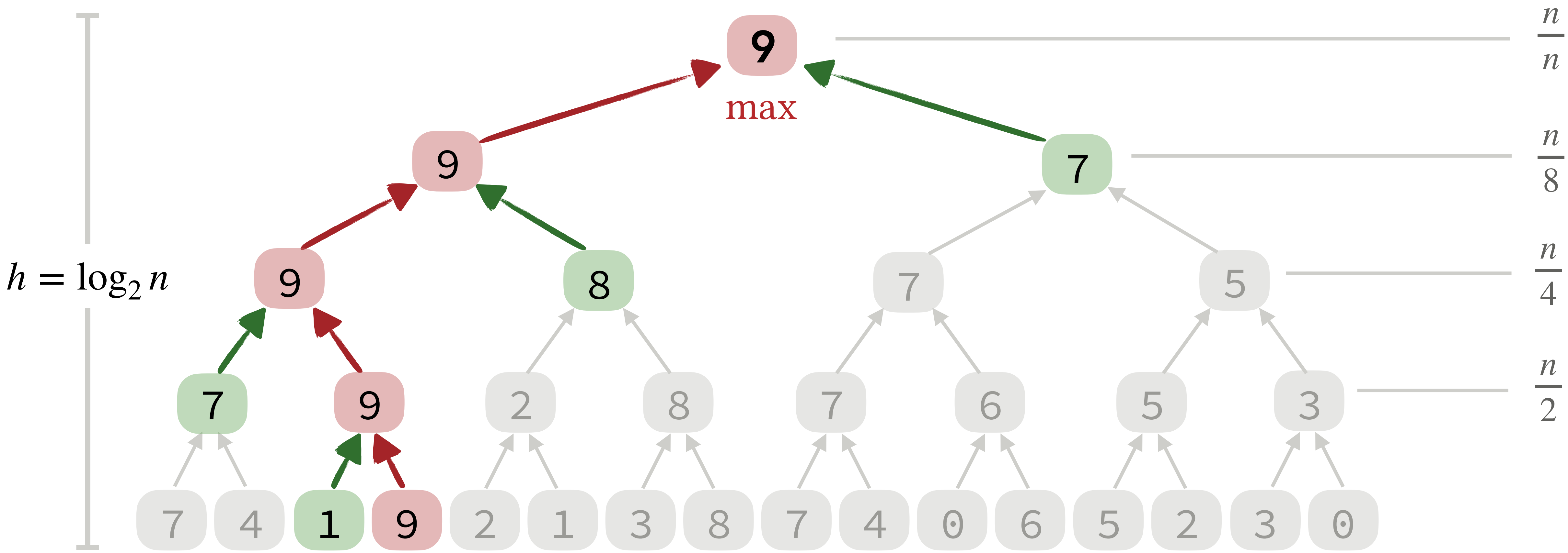
Selecting the 2nd Largest Element

A Tournament-based approach.

Observation. Keys that lose to keys other than the max can't possibly be the second largest.
Proof by contradiction.

Analysis. (assuming n is a power of 2)

- The algorithm performs $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ \ldots \ + 1 = n - 1$ compares to find the max.
- The algorithm needs an additional $\log_2 n - 1$ compares to find the second largest among the nodes on the path representing comparisons with the max.



Selecting the 2nd Largest Element

A Tournament-based approach.

Observation. Keys that lose to keys other than the max can't possibly be the second largest.
Proof by contradiction.

Analysis. (assuming n is a power of 2)

- The algorithm performs $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ \dots \ + 1 = n - 1$ compares to find the max.
- The algorithm needs an additional $\log_2 n - 1$ compares to find the second largest among the nodes on the path representing comparisons with the max.

  Theorem. Any comparison-based algorithm for finding the second largest element in an arbitrary array must make at least $n + \lceil \log_2 n \rceil - 2$ compares in the worst case*.

* for the **proof**, see Computer Algorithms: Introduction to Design and Analysis for Sara Baase and Allen Van Gelder

Selecting the 2nd Largest Element

A Tournament-based approach.

Observation. Keys that lose to keys other than the max can't possibly be the second largest.
Proof by contradiction.

Analysis. (assuming n is a power of 2)

- The algorithm performs $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ \ldots \ + 1 = n - 1$ compares to find the max.
- The algorithm needs an additional $\log_2 n - 1$ compares to find the second largest among the nodes on the path representing comparisons with the max.

Theorem. Any comparison-based algorithm for finding the second largest element in an arbitrary array must make at least $n + \lceil \log_2 n \rceil - 2$ compares in the worst case*.

Implementation. How can we keep track of which elements lost to the max?

* for the **proof**, see Computer Algorithms: Introduction to Design and Analysis for Sara Baase and Allen Van Gelder

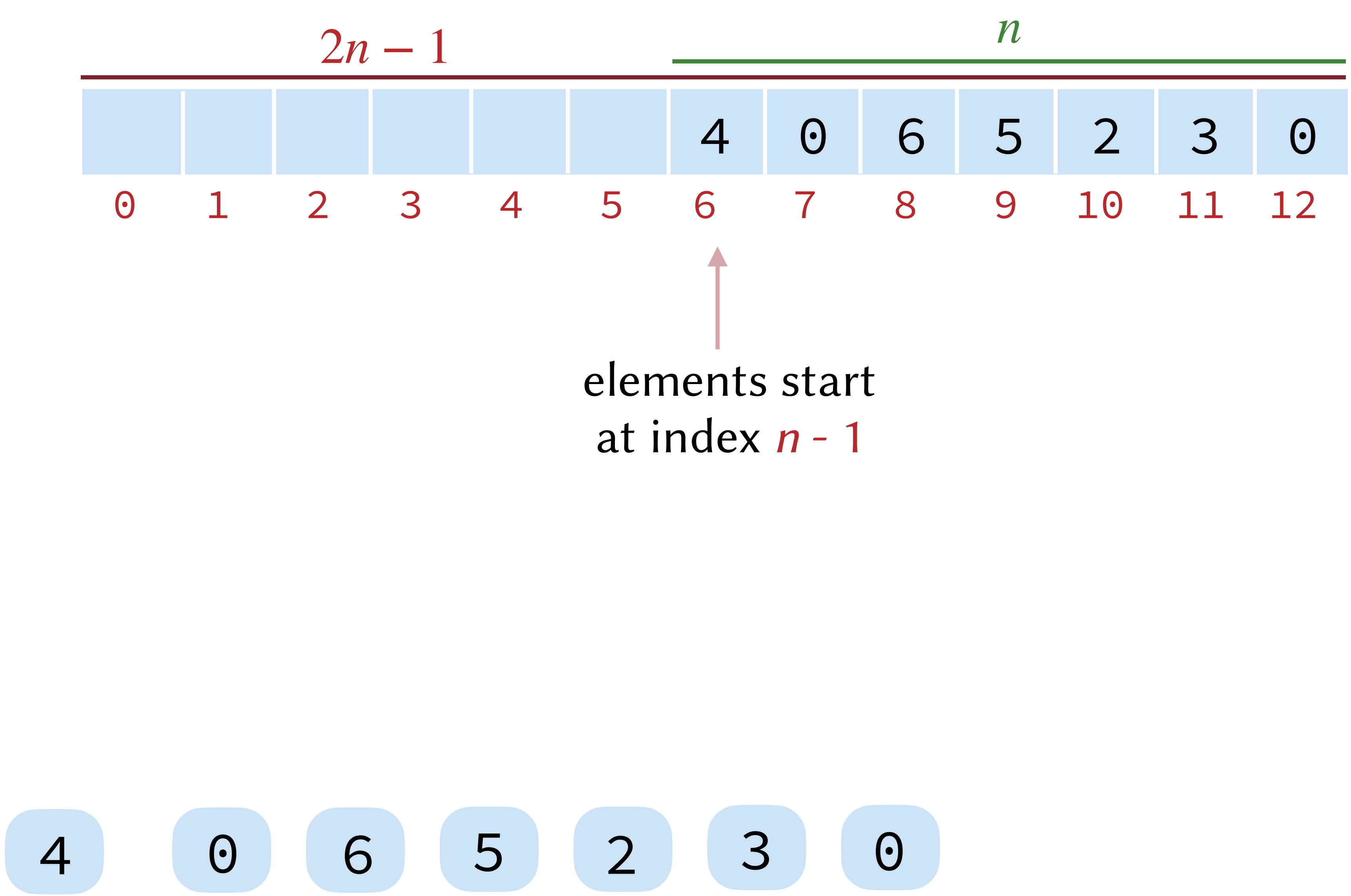
Implementation

Use a heap-like structure to keep track of the comparisons.

Implementation

Use a heap-like structure to keep track of the comparisons.

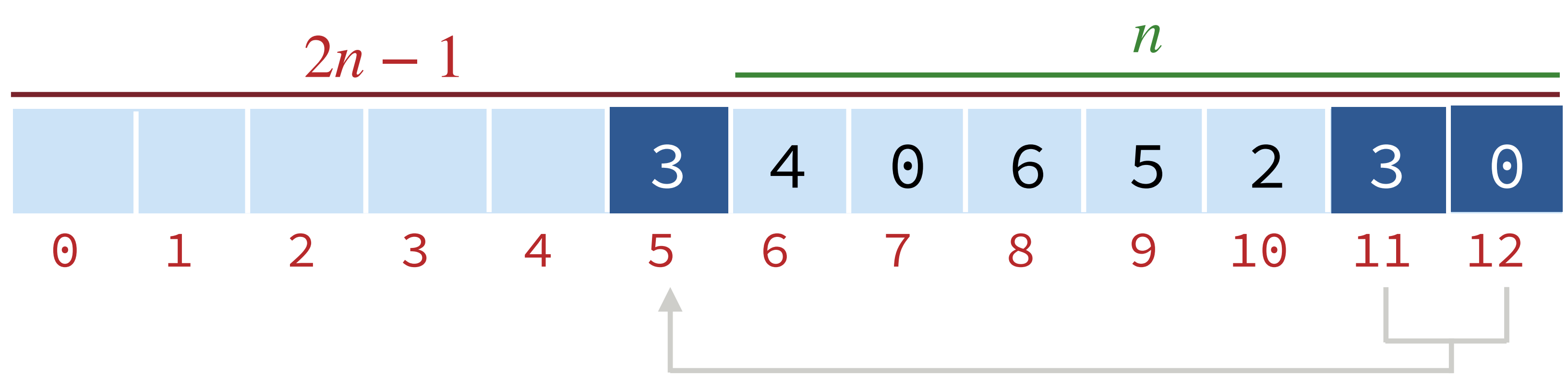
1. Load the n elements into the right half of an array of size $2n - 1$.

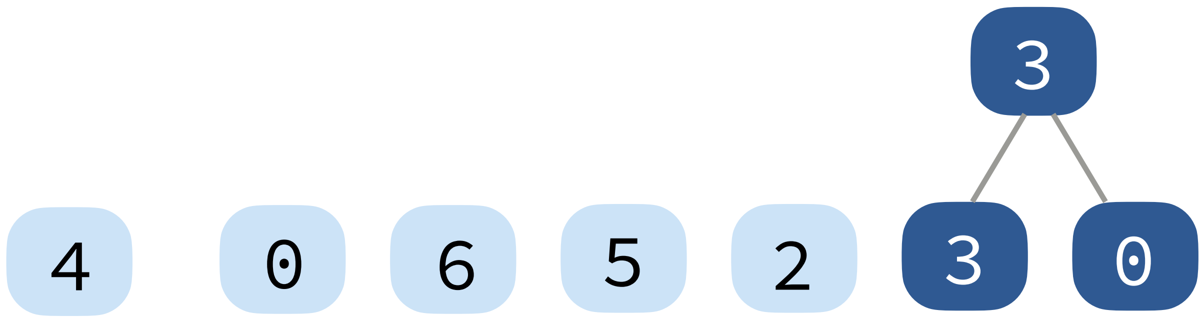


Implementation

Use a heap-like structure to keep track of the comparisons.

1. Load the n elements into the right half of an array of size $2n$.
2. Compute the max of each pair and store it at the location of the parent.

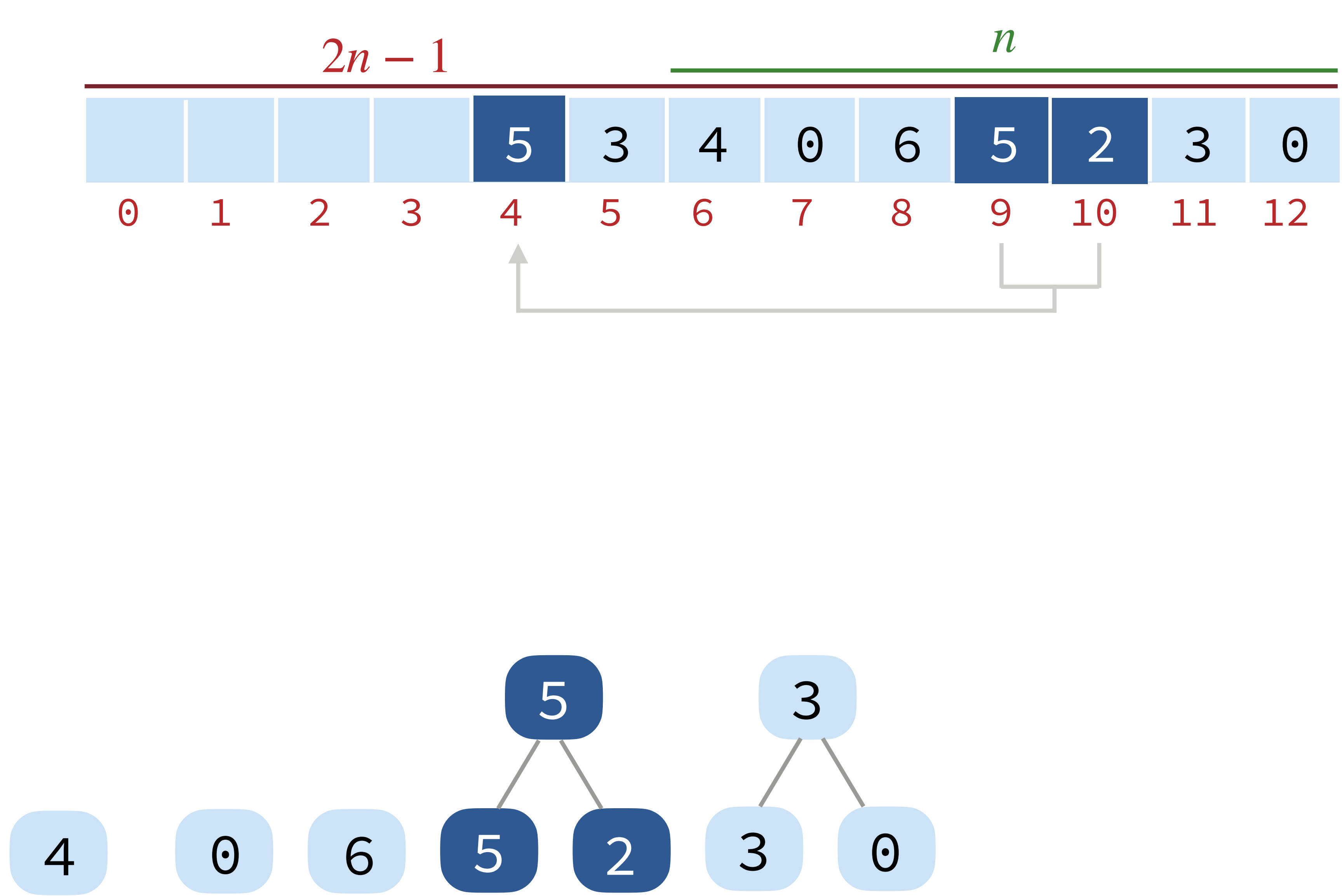




Implementation

Use a heap-like structure to keep track of the comparisons.

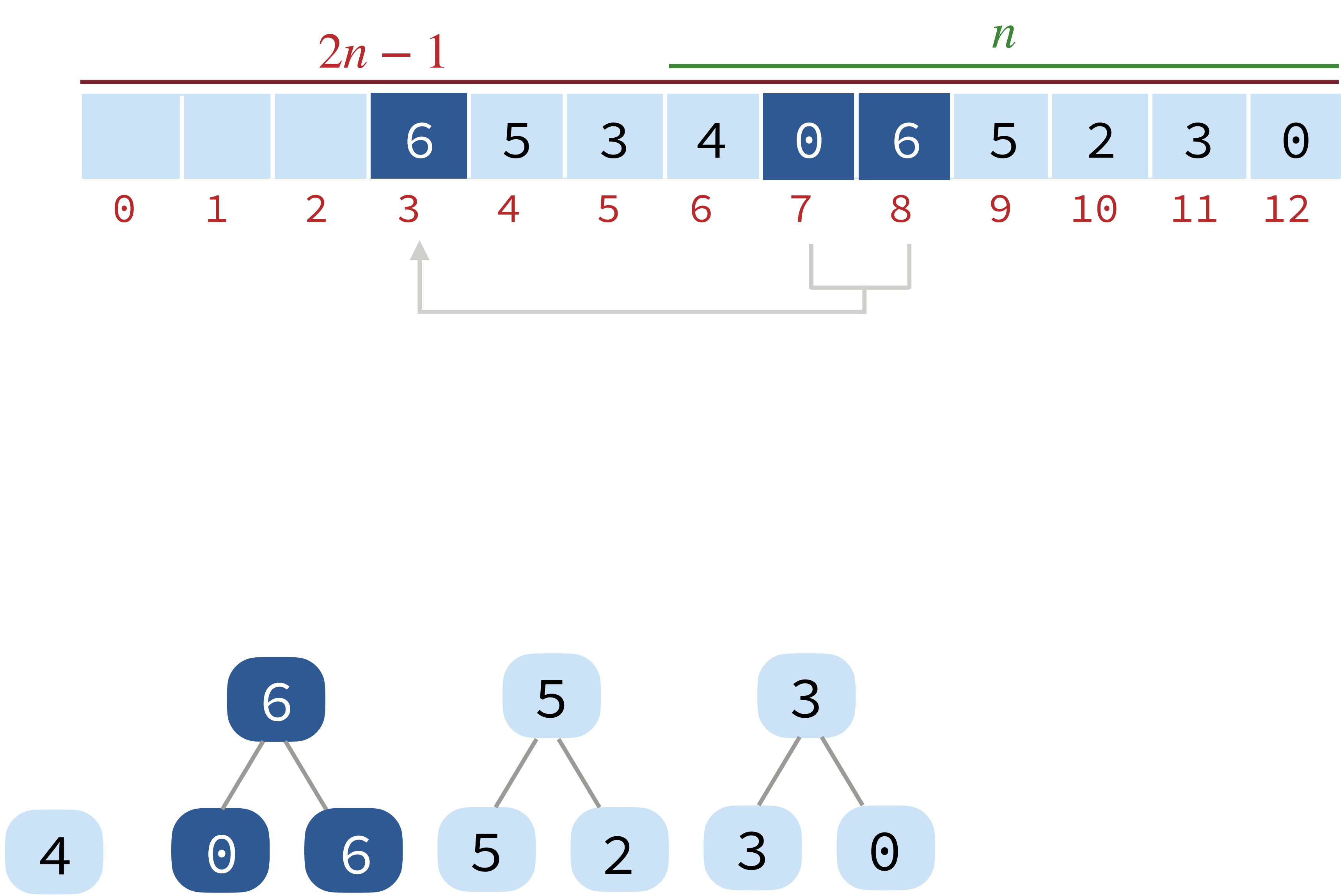
1. Load the n elements into the right half of an array of size $2n$.
2. Compute the max of each pair and store it at the location of the parent.



Implementation

Use a heap-like structure to keep track of the comparisons.

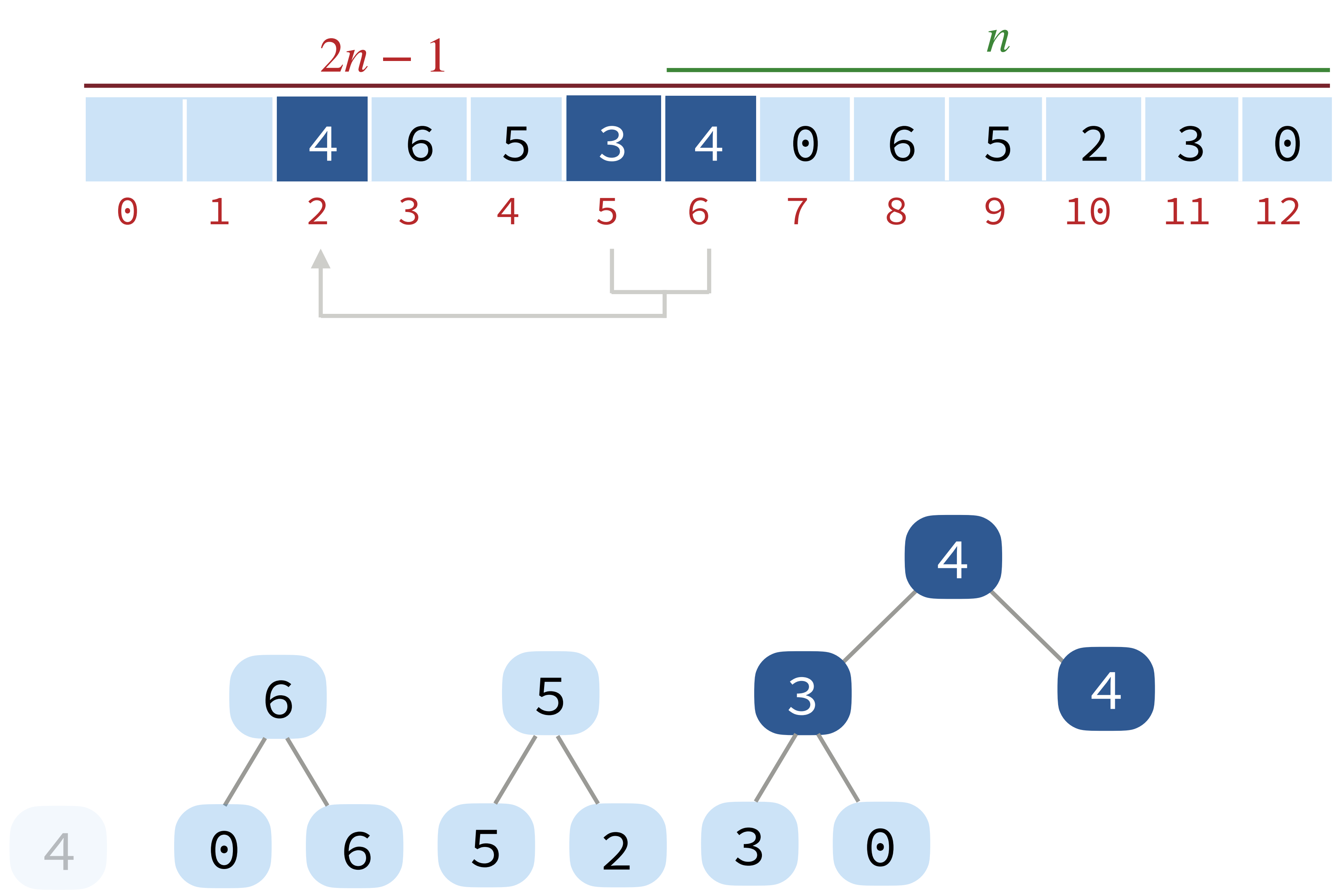
1. Load the n elements into the right half of an array of size $2n$.
2. Compute the max of each pair and store it at the location of the parent.



Implementation

Use a heap-like structure to keep track of the comparisons.

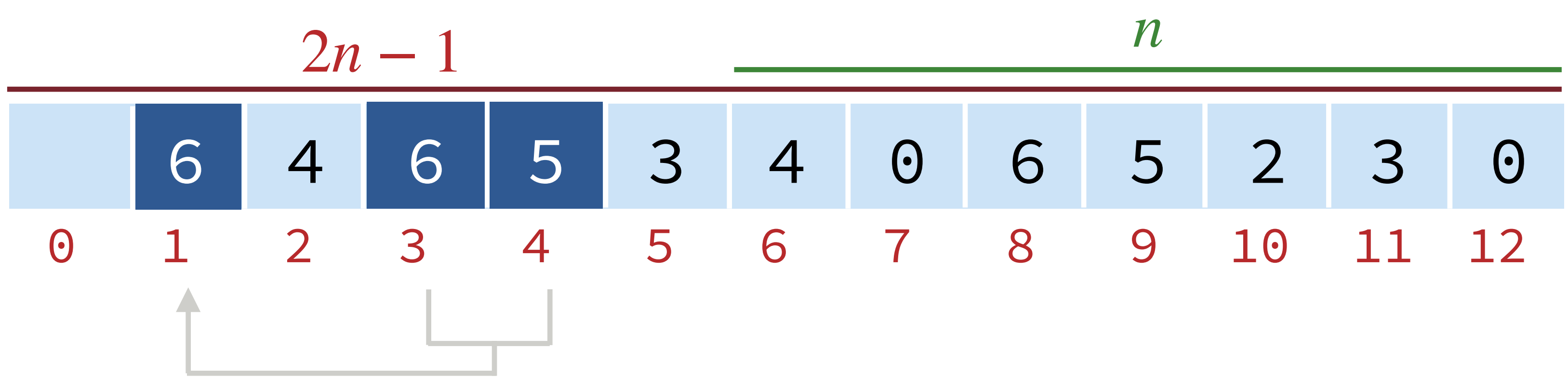
1. Load the n elements into the right half of an array of size $2n$.
2. Compute the max of each pair and store it at the location of the parent.

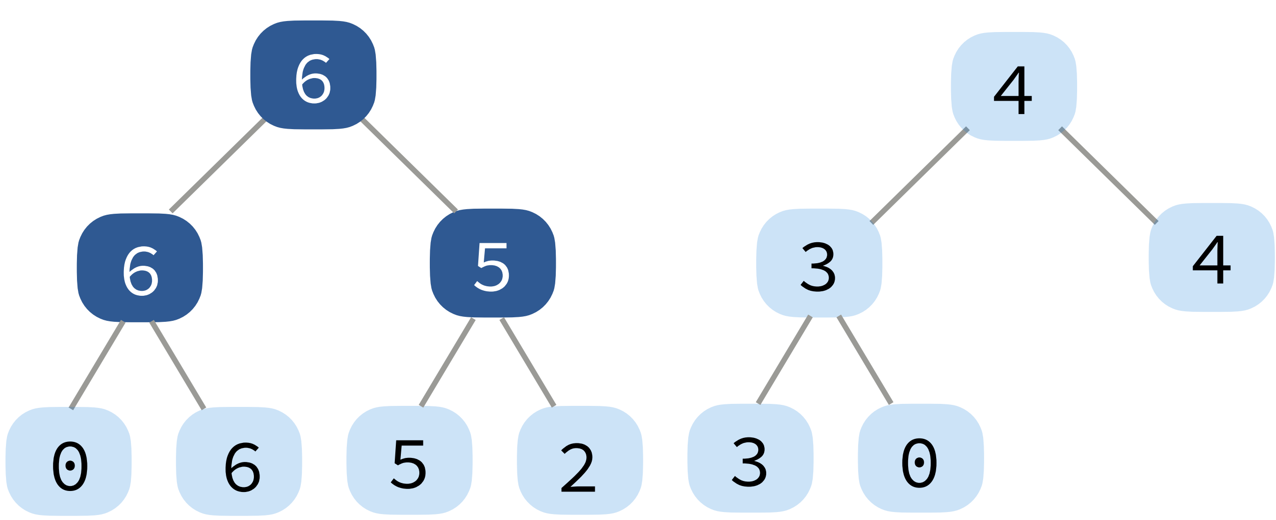


Implementation

Use a heap-like structure to keep track of the comparisons.

1. Load the n elements into the right half of an array of size $2n$.
2. Compute the max of each pair and store it at the location of the parent.

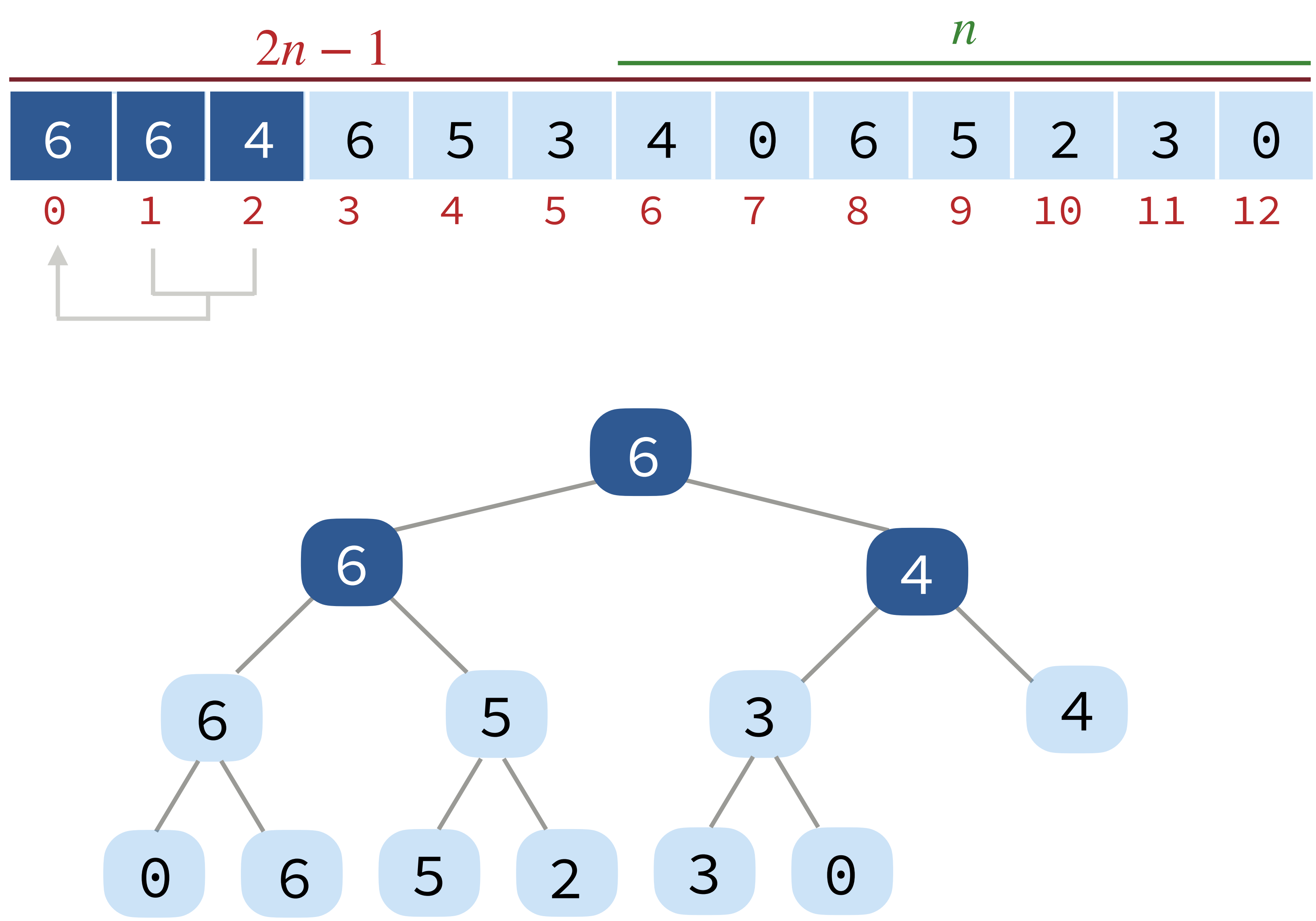




Implementation

Use a heap-like structure to keep track of the comparisons.

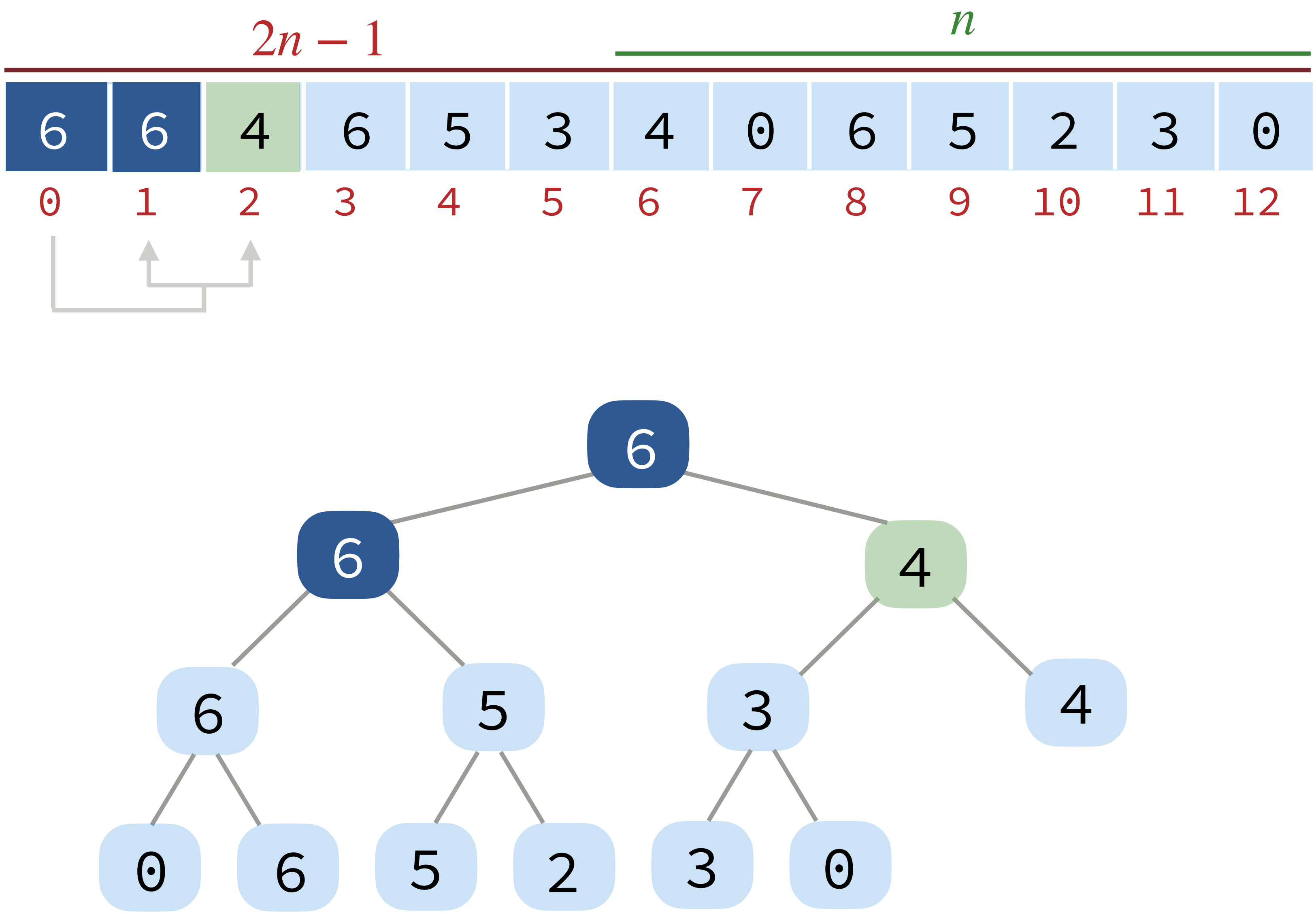
1. Load the n elements into the right half of an array of size $2n$.
2. Compute the max of each pair and store it at the location of the parent.



Implementation

Use a heap-like structure to keep track of the comparisons.

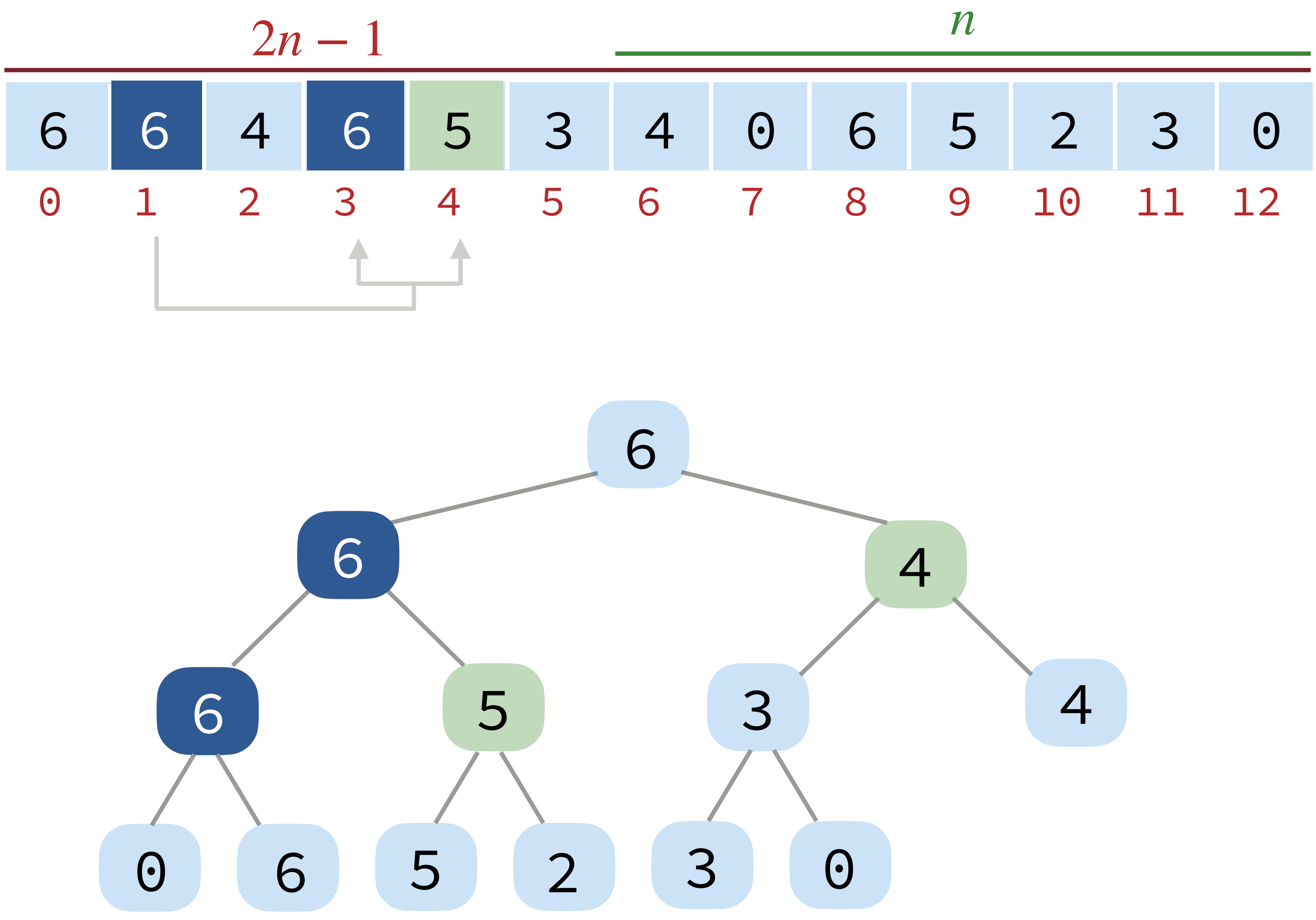
1. Load the n elements into the right half of an array of size $2n$.
2. Compute the max of each pair and store it at the location of the parent.
3. Trace back from the root and compare to elements that lost to the max.



Implementation

Use a heap-like structure to keep track of the comparisons.

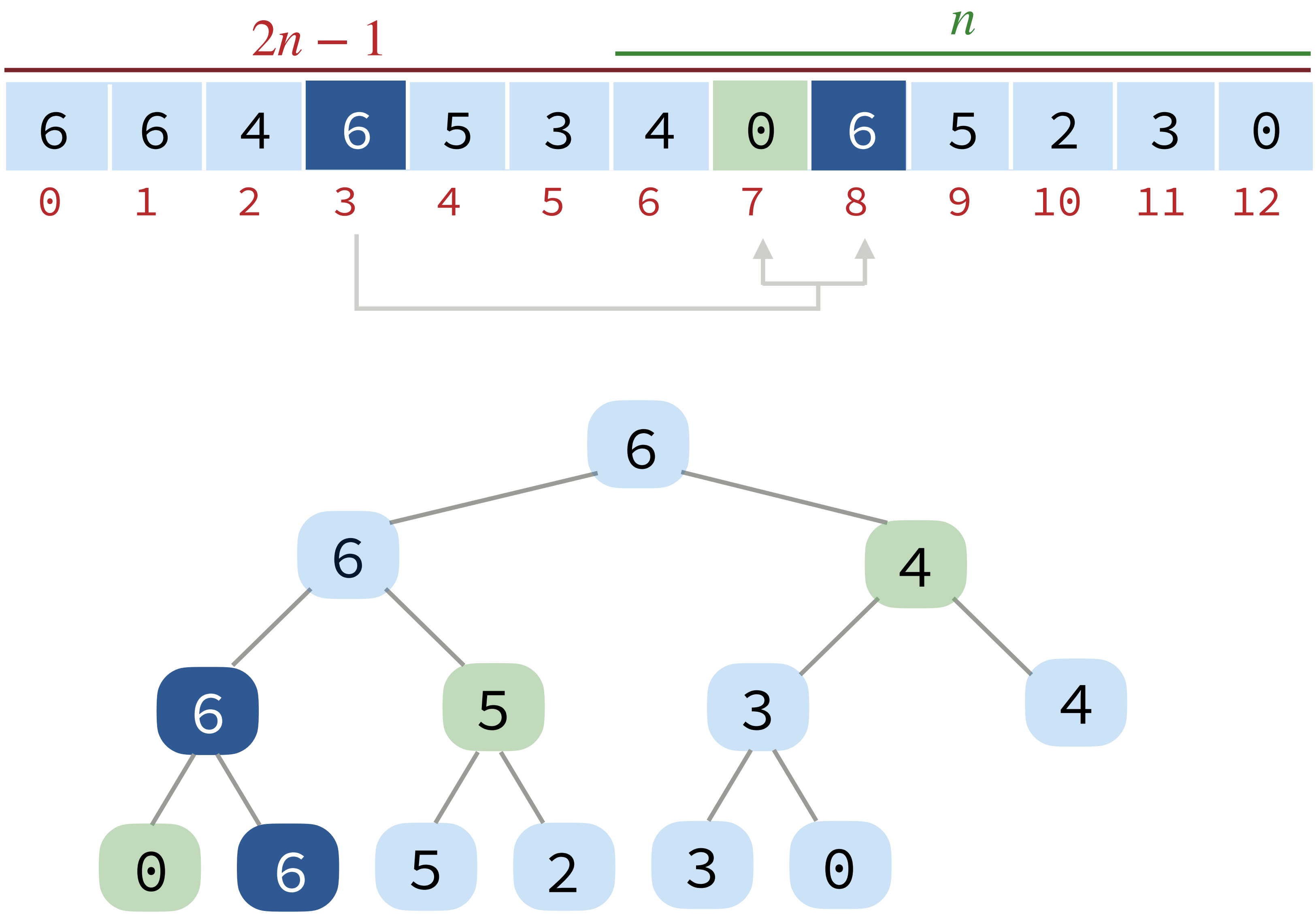
1. Load the n elements into the right half of an array of size $2n$.
2. Compute the max of each pair and store it at the location of the parent.
3. Trace back from the root and compare to elements that lost to the max.



Implementation

Use a heap-like structure to keep track of the comparisons.

1. Load the n elements into the right half of an array of size $2n$.
2. Compute the max of each pair and store it at the location of the parent.
3. Trace back from the root and compare to elements that lost to the max.



Implementation

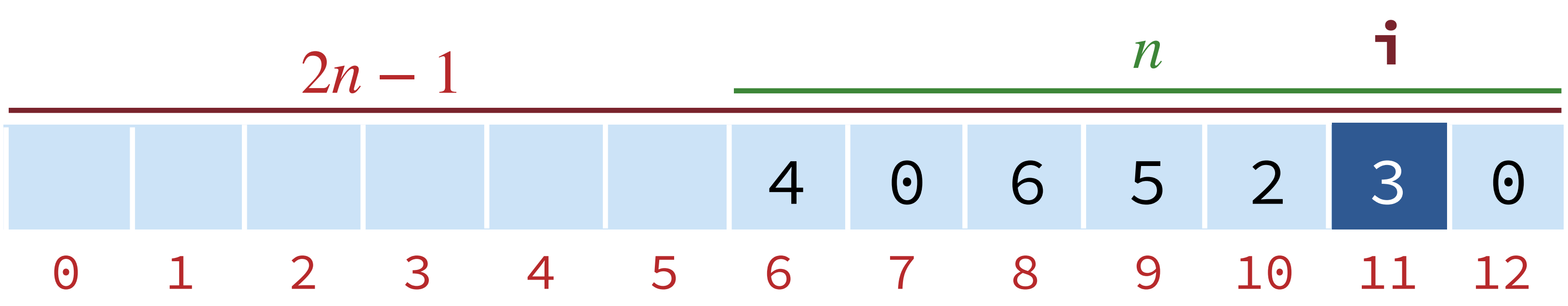
```
BUILD-HEAP(a[], n)
    Create an array named h[] of size 2n-1
    Copy a[0 → n-1] to h[n-1 → 2n-2]
```

Implementation

BUILD-HEAP(a[], n)

```
Create an array named h[] of size 2n-1
Copy a[0 → n-1] to h[n-1 → 2n-2]
i = 2n-3
```

start at the 2nd to last element in the array

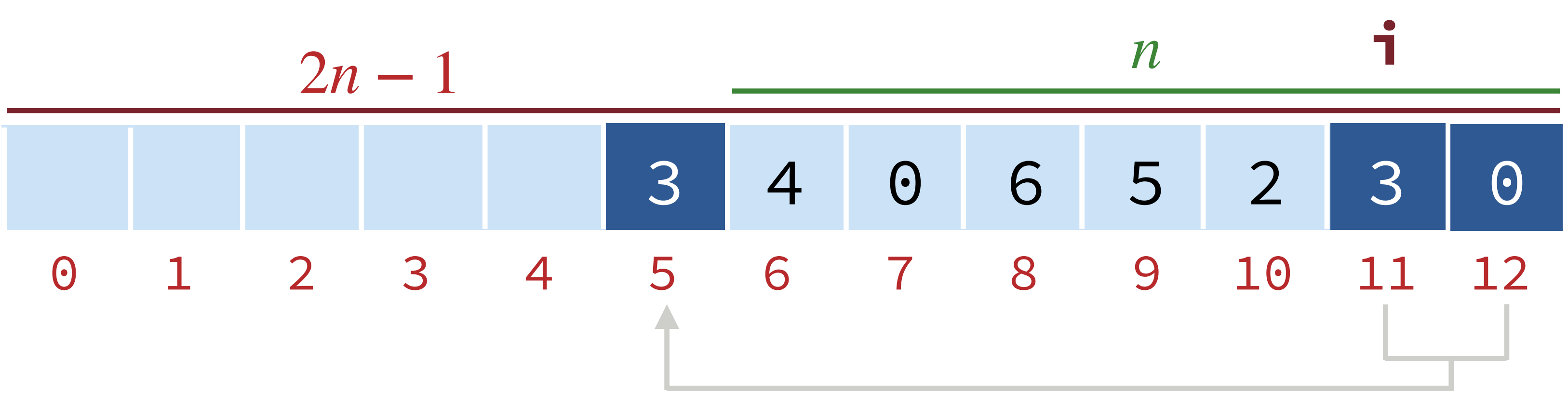


Implementation

```
BUILD-HEAP(a[], n)
  Create an array named h[] of size 2n-1
  Copy a[0 → n-1] to h[n-1 → 2n-2]

  i = 2n-3
  while (i > 0):
      h[PARENT(i)] = max(h[i], h[i+1])
```

store the max at the index of the parent node (i-1)/2

compare every pair of nodes

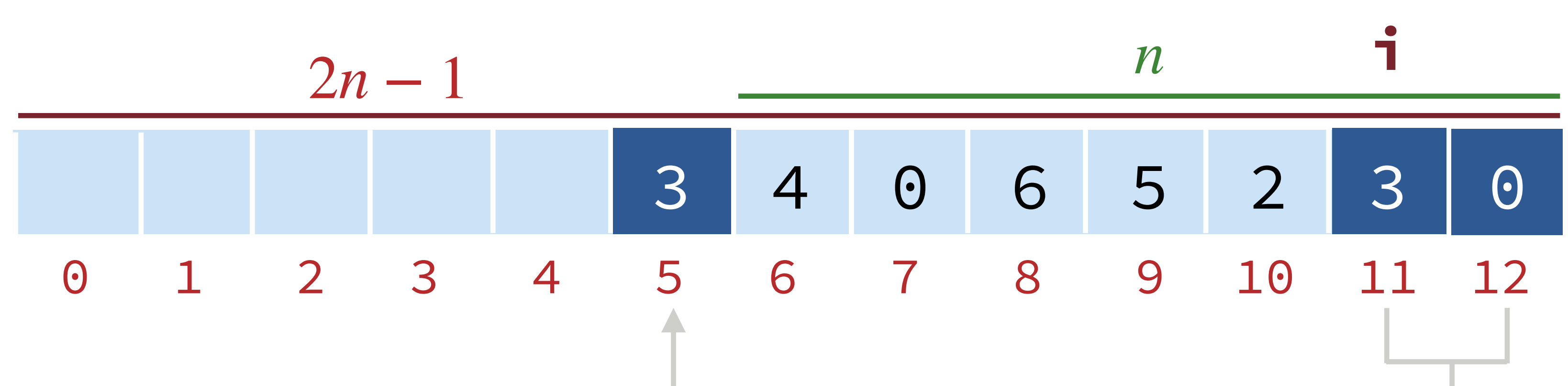


Implementation

BUILD-HEAP(a[], n)

```
Create an array named h[] of size 2n-1
Copy a[0 → n-1] to h[n-1 → 2n-2]

i = 2n-3

while (i > 0):

    h[PARENT(i)] = max(h[i], h[i+1])
    i = i-2
```

move to the next pair

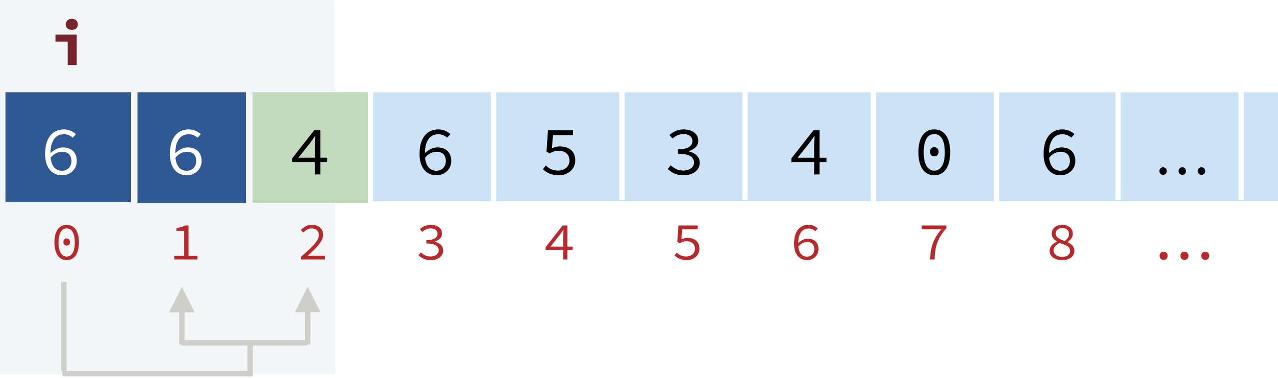


Implementation

```
BUILD-HEAP(a[], n)

  Create an array named h[] of size 2n-1
  Copy a[0 → n-1] to h[n-1 → 2n-2]

  i = 2n-3

  while (i > 0):

      h[PARENT(i)] = max(h[i], h[i+1])
      i = i-2
```

```
HEAP-FIND-MAX(h[], n)
  max2 = a very small number, i=0
  while i < n - 1:
```

elements after this are leaves



Implementation

BUILD-HEAP(a[], n)

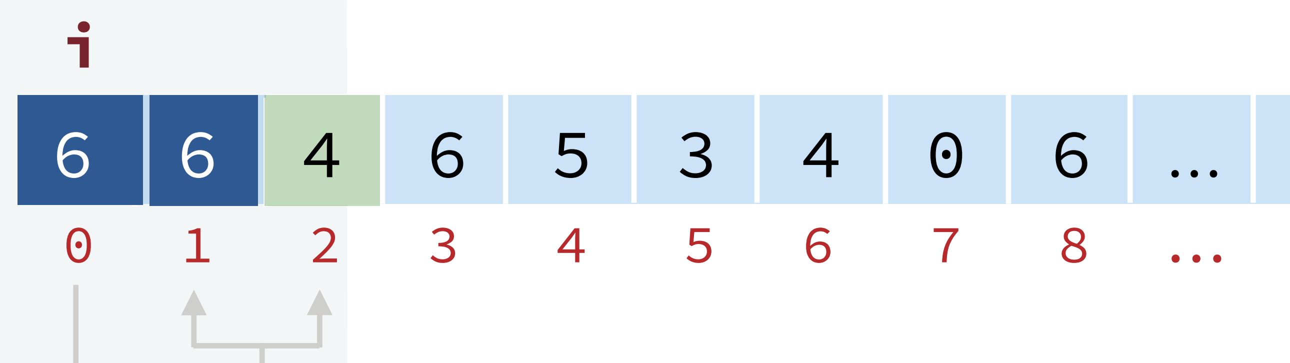
```
Create an array named h[] of size 2n-1
Copy a[0 → n-1] to h[n-1 → 2n-2]

i = 2n-3

while (i > 0):

    h[PARENT(i)] = max(h[i], h[i+1])
    i = i-2
```

HEAP-FIND-MAX(h[], n)

```
max2 = a very small number, i=0

while i < n - 1:

    if (h[i] == h[LEFT(i)]):
        max2 = MAX(max2, h[RIGHT(i)])
        i    = LEFT(i)
```

Remember:

```
LEFT(i):  2*i + 1
RIGHT(i): 2*i + 2
```



if the max is in the left child,
compare the current 2nd largest to the
right child and then move to the left child

Implementation

```
BUILD-HEAP(a[], n)

Create an array named h[] of size 2n-1
Copy a[0 → n-1] to h[n-1 → 2n-2]

i = 2n-3

while (i > 0):

    h[PARENT(i)] = max(h[i], h[i+1])
    i = i-2
```

```
HEAP-FIND-MAX(h[], n)

max2 = a very small number, i=0

while i < n - 1:

    if (h[i] == h[LEFT(i)]):
        max2 = MAX(max2, h[RIGHT(i)])
        i    = LEFT(i)
    else:
        max2 = MAX(max2, h[LEFT(i)])
        i    = RIGHT(i)
```

otherwise, compare the current 2nd largest to the left child and then move to the right child

Implementation

```
BUILD-HEAP(a[], n)

Create an array named h[] of size 2n-1
Copy a[0 → n-1] to h[n-1 → 2n-2]

i = 2n-3

while (i > 0):

    h[PARENT(i)] = max(h[i], h[i+1])
    i = i-2
```

```
HEAP-FIND-MAX(h[], n)

max2 = a very small number, i=0

while i < n - 1:

    if (h[i] == h[LEFT(i)]):
        max2 = MAX(max2, h[RIGHT(i)])
        i    = LEFT(i)
    else:
        max2 = MAX(max2, h[LEFT(i)])
        i    = RIGHT(i)

return max2
```

optional

Streaming Median

Assume that you receive an arbitrary stream of numbers. How can we efficiently report the median at any point in time?

Streaming Median

Assume that you receive an arbitrary stream of numbers. How can we efficiently report the median at any point in time?

Solution 1. Maintain a max-heap:

`insert()`: $O(\log n)$

`median()`: $O(n \log n)$
Remove from the heap the first $\frac{n}{2}$ elements to reach the median and then insert them back.

Streaming Median

Assume that you receive an arbitrary stream of numbers. How can we efficiently report the median at any point in time?

Solution 1. Maintain a max-heap:

`insert()`: $O(\log n)$

`median()`: $O(n \log n)$
Remove from the heap the first $\frac{n}{2}$ elements to reach the median and then insert them back.

Solution 2. Maintain an unordered array:

`insert()`: $\Theta(1)$
Add to the end of the list. Note that the array might resize, so the running time is amortized.

`median()`: $\Theta(n)$
Use `Quickselect` to find the median. Note that this is the expected case if the array is shuffled.

Streaming Median

Assume that you receive an arbitrary stream of numbers. How can we efficiently report the median at any point in time?

Solution 1. Maintain a max-heap:

`insert()`: $O(\log n)$

`median()`: $O(n \log n)$
Remove from the heap the first $\frac{n}{2}$ elements to reach the median and then insert them back.

Solution 2. Maintain an unordered array:

`insert()`: $\Theta(1)$
Add to the end of the list. Note that the array might resize, so the running time is amortized.

`median()`: $\Theta(n)$
Use `Quickselect` to find the median. Note that this is the expected case if the array is shuffled.

Solution 3. Maintain a sorted array:

`insert()`: $O(n)$
Search for the right position and then shift any elements that come after.

`median()`: $\Theta(1)$
The median is always at index $\frac{n}{2}$.

Streaming Median

Solution 4. Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.

Streaming Median

Solution 4. Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Streaming Median

Solution 4. Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size()` - **`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

`left` `right`

Example: `[1 2` **`3`** `• 4 5 6]`

↑ `median`

`left` `right`

Example: `[1 2 3` **`4`** `• 5 6 7]`

↑ `median`

Streaming Median

Solution 4. Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size()` - **`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

Example: `[1 2` **`3`** `• 4 5 6]` **Example**: `[1 2 3` **`4`** `• 5 6 7]`

General Idea:

- **Insert** the new element into the left heap if it is less than or equal to the current median and to the right if it is greater than the current median.

- **Rebalance** the heaps by moving an element from the larger heap to the smaller heap if the size invariant is violated.

Streaming Median

Solution 4. Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

Example: `[1 2 3 • 4 5 6]` **Example**: `[1 2 3 4 • 5 6 7]`

```
insert(k):

    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)
```

Streaming Median

Solution 4. Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

Example: `[1 2 `**`3`**` • 4 5 6]` **Example**: `[1 2 3 `**`4`**` • 5 6 7]`

```
insert(k):

    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)
```

insert k in **`left`** if $k \leq$ median

insert k in **`right`** if $k >$ median

median (`left.max()`)

Streaming Median

Solution 4. Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

Example: `[1 2 `**`3`**` • 4 5 6]` **Example**: `[1 2 3 `**`4`**` • 5 6 7]`

```
insert(k):

    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)

    If left.size()  > right.size()+1: right.insert(left.delMax()).
```

more than $\lceil \frac{n}{2} \rceil$ elements are in **`left`**

Streaming Median

Solution 4. Use a max-heap to store the lower half of the elements ($\leq$ median) and a min-heap to store the upper half of the elements ($>$ median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

Example: `[1 2 3 • 4 5 6]` **Example**: `[1 2 3 4 • 5 6 7]`

```
insert(k):

    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)

    If left.size()  > right.size()+1: right.insert(left.delMax()).
```

remove the max from left and insert it into right

Streaming Median

Solution 4. Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

Example: `[1 2 3 • 4 5 6]` **Example**: `[1 2 3 4 • 5 6 7]`

```
insert(k):

    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)

    If left.size()  > right.size()+1: right.insert(left.delMax()).
    If right.size() > left.size():    left.insert(right.delMin()).
```

If **`right`** is larger than **`left`** — remove the min from **`right`** and insert it into **`left`**.

Streaming Median

Solution 4. Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

Example: `[1 2` **`3`** `• 4 5 6]` **Example**: `[1 2 3` **`4`** `• 5 6 7]`

```
insert(k):
    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)

    If left.size()  > right.size()+1: right.insert(left.delMax()).
    If right.size() > left.size():    left.insert(right.delMin()).
```

insert into the correct heap

rebalance the heaps if necessary

Streaming Median

Solution 4. Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

Example: `[1 2 `**`3`**` • 4 5 6]`   **Example**: `[1 2 3 `**`4`**` • 5 6 7]`

```
insert(k):
    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)

    If left.size()  > right.size()+1: right.insert(left.delMax()).
    If right.size() > left.size():    left.insert(right.delMin()).
```

insert into the correct heap — $O(\log n)$

rebalance the heaps if necessary — $O(\log n)$

Streaming Median

Solution 4. Use a max-heap to store the lower half of the elements ($\leq$ median) and a min-heap to store the upper half of the elements ($>$ median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

Example: `[1 2 `**`3`**` • 4 5 6]`   **Example**: `[1 2 3 `**`4`**` • 5 6 7]`

Running Time:

`insert()`: $O(\log n)$
Inserting into the left or the right heaps is $O(\log n)$ and rebalancing is $O(\log n)$.

`median()`: $\Theta(1)$
The median is always `left.max()`.