Lecture 11
Sorting 2

EECS-214

Why is sorting hard?

- Have to **compare a lot** of elements
- Naïve algorithms involve **exhaustive comparisons**
  - **Every element** to every other element
  - $O(n^2)$

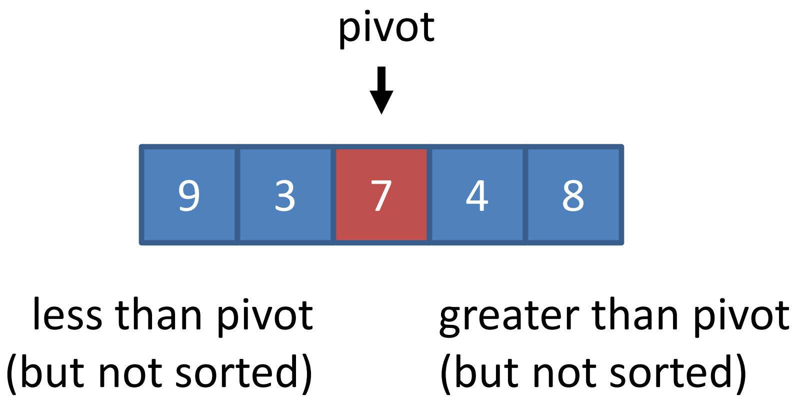
Question

- What can we do by comparing a **single element** to every other element?
  - $O(n)$

Partitioning

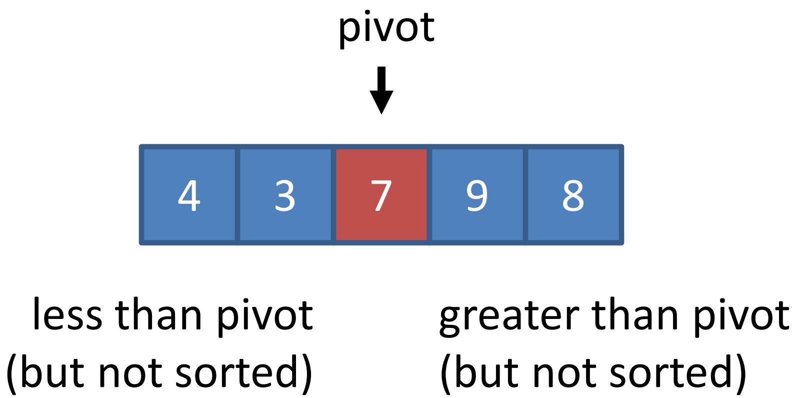
- Comparing one element to all the others lets us **divide the array** into the elements
  - **less than**,
  - **greater than**,
  - and **equal to** the original element

- This is called **partitioning** the array
  - And the element being compared to is called the **pivot** element



Partitioning

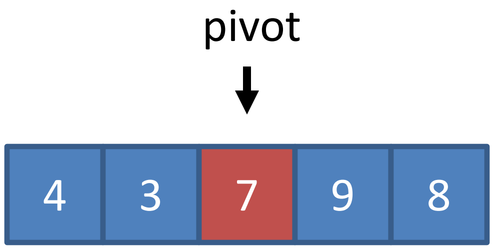
- Partitioning makes the array "**more sorted**"
- But we're not done yet
  - The left- and right- sides **aren't sorted internally**
- What can we do to finish sorting?



Partitioning

Recurse!

- Use partitioning to make the left and right sides more sorted too
- Keep going until everything is fully sorted



Quicksort (Hoare, 1960)

- One of the most famous algorithms in computer science
- **Approximately sorts** whole array
- **Recurses** to complete sorting of each half

Quicksort(array)

  select pivot element

  partition about pivot

  Quicksort(left side)

  Quicksort(right side)

Quicksort (Hoare, 1960)

- This code is pretty **hand-wavy**
  - For example what does it mean to call quicksort of the left side of the array?
  - Also, there's nothing to stop the recursion
- Making it more **rigorous** is **straightforward**
  - But there are more confusing details
  - So it's good to understand the basic idea first

```
Quicksort(array)
    select pivot element
    partition about pivot
    Quicksort(left side)
    Quicksort(right side)
```

Implementing partitioning

Basic idea

- **Move pivot** to the **right** (the end)
- **Sweep from left** to right
  - If you see an element that's **less than** the pivot
  - **Swap** it with something on the left that's bigger than the pivot
- **Move pivot between** the less-than and greater-than elements

```
Partition(a, pIndex)
  pValue = a[pIndex]
  // Move pivot to end
  swap a[pIndex]
        with last element of a
  // Move small values left
  nextLeft = 0
  for i = 0 to a.Length-1
     if a[i]<=pValue
       swap a[i]
             with a[nextLeft]
       nextLeft++
  // Move the pivot into place
  swap a[nextLeft]
        with last element of a
```

Implementing partitioning

This algorithm is a little inscrutable

- This is an algorithm that induces a kind of behavior in textbook writers that I hate: the **pseudoexplanation**

```
Partition(a, pIndex)
  pValue = a[pIndex]
  // Move pivot to end
  swap a[pIndex]
        with last element of a
  // Move small values left
  nextLeft = 0
  for i = 0 to a.Length-1
     if a[i]<=pValue
       swap a[i]
             with a[nextLeft]
       nextLeft++
  // Move the pivot into place
  swap a[nextLeft]
        with last element of a
```

Implementing partitioning

Wikipedia's pseudoexplanation:

- "This is the in-place partition algorithm. It partitions … the array … by moving all elements less than or equal to [the pivot value] to the beginning of the subarray, leaving all the greater elements following them"

- This is simply a **restatement** of what the procedure **should do**
- It's not an explanation of **why it works**

```
Partition(a, pIndex)
  pValue = a[pIndex]
  // Move pivot to end
  swap a[pIndex]
        with last element of a
  // Move small values left
  nextLeft = 0
  for i = 0 to a.Length-1
     if a[i]<=pValue
       swap a[i]
             with a[nextLeft]
       nextLeft++
  // Move the pivot into place
  swap a[nextLeft]
        with last element of a
```

Implementing partitioning

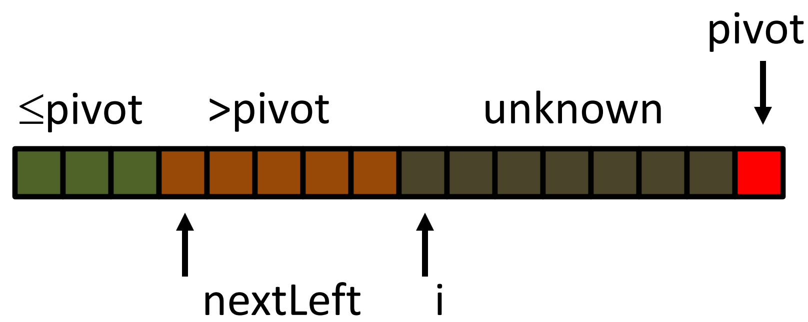
Okay, why does it actually work?

```
Partition(a, pIndex)
  pValue = a[pIndex]
  // Move pivot to end
  swap a[pIndex]
        with last element of a
  // Move small values left
  nextLeft = 0
  for i = 0 to a.Length-1
     if a[i]<=pValue
       swap a[i]
             with a[nextLeft]
       nextLeft++
  // Move the pivot into place
  swap a[nextLeft]
        with last element of a
```

Implementing partitioning

Claim: the loop maintains the **invariants** that:

- Elements **before nextLeft** are ≤ **the pivot**
- Elements **between nextLeft and i**, but not including i, **are > the pivot**

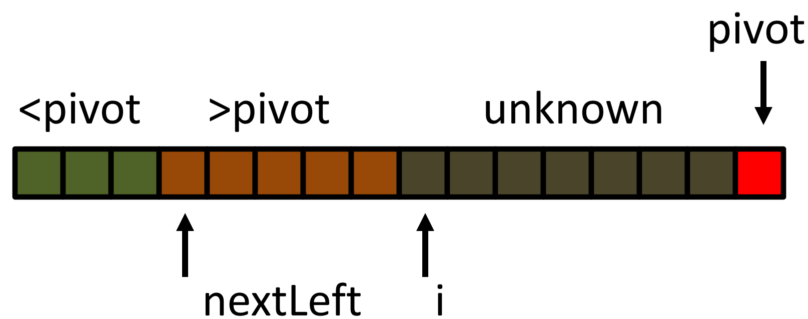


Implementing partitioning

Theorem: on each iteration of the for loop:

For any j:

- a[j] ≤ pivot, if j<nextLeft
- a[j] > pivot, if nextLeft≤j<i



Proof:

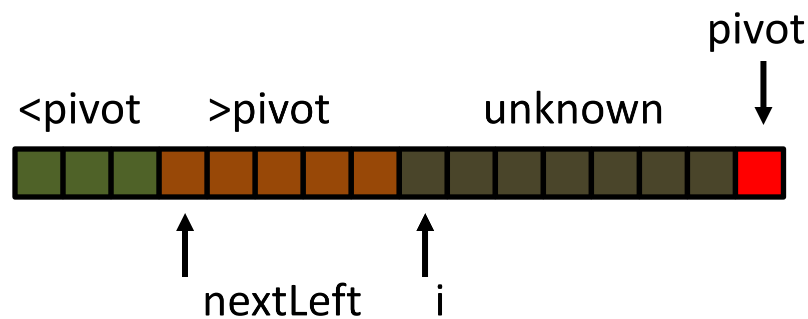
- For the first iteration, i=nextLeft=0
- So there are no j<nextLeft or j<i
- So it's trivially true

Implementing partitioning

Theorem: on each iteration of the for loop:

For any j:

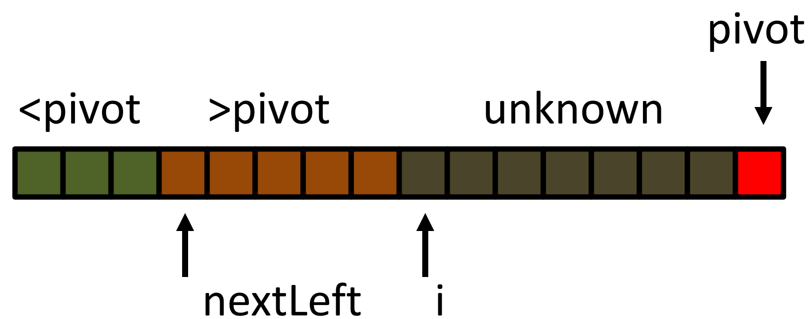
- a[j] ≤ pivot, if j<nextLeft
- a[j] > pivot, if nextLeft≤j<i



Proof:

- Now assume it's true at the end of iteration i-1
- Let's show it's true at the end of iteration i

Implementing partitioning



Theorem: on each iteration of the for loop:

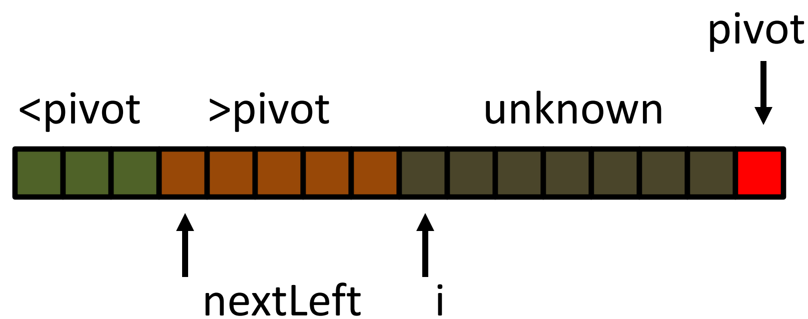
For any j:

- $a[j] \leq pivot$, if $j<nextLeft$
- $a[j] > pivot$, if $nextLeft \leq j < i$

Proof:

- So we know
  - $a[j] \leq pivot$, for $j<nextLeft$
  - $a[j] > pivot$, for $nextLeft \leq j < i-1$
- Case 1: $a[i] > pivot$
  - Don't do anything
  - So $a[j]$ still $\leq pivot$, for $j<nextLeft$
  - $a[j]$ still $> pivot$, for $nextLeft \leq j < i-1$
  - but $a[i]$ also $> pivot$, so
  - $a[j] > pivot$, for $nextLeft \leq j < i$

Implementing partitioning



Theorem: on each iteration of the for loop:

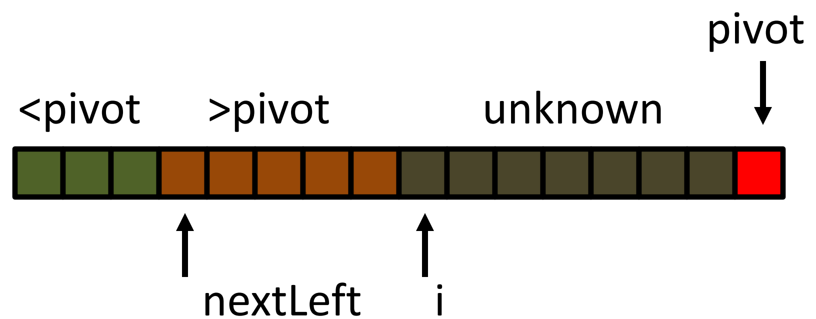
For any j:

– a[j] ≤ pivot, if j<nextLeft
– a[j] > pivot, if nextLeft≤j<i

Proof:

- So we know
  - a[j] ≤ pivot, for j<nextLeft
  - a[j] > pivot, for nextLeft≤j<i-1
- Case 2: a[i] ≤ pivot
  - We swap a[i] and a[nextLeft]
    - a[nextLeft] is now ≤ pivot
  - And we increment nextLeft
  - So once again, a[j] ≤ pivot, for j<nextLeft

Implementing partitioning



Theorem: on each iteration of the for loop:

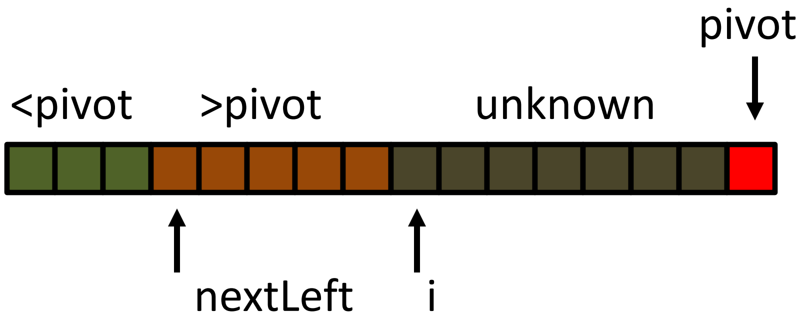
For any j:

- $a[j] \leq$ pivot, if j<nextLeft
- $a[j] >$ pivot, if nextLeft$\leq$j<i

Proof:

- So we know
  - $a[j] \leq$ pivot, for j<nextLeft
  - $a[j] >$ pivot, for nextLeft$\leq$j<i-1
- Case 2: $a[i] \leq$ pivot
  - We swap a[i] and a[nextLeft]
    - This also means a[i] is now > pivot
  - a[j] still > pivot, for nextLeft$\leq$j<i-1
  - but a[i] also > pivot, so
  - $a[j] >$ pivot, for nextLeft$\leq$j<i

Implementing partitioning



Theorem: on each iteration of the for loop:

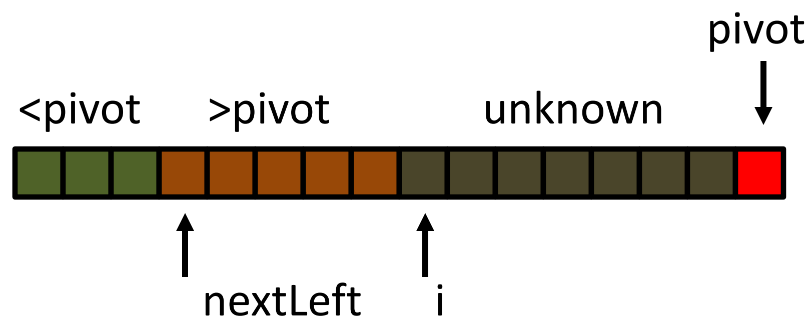
For any j:

- a[j] ≤ pivot,   if j<nextLeft
- a[j] > pivot,   if nextLeft≤j<i

Proof:

- So by induction, the theorem holds for each iteration

Implementing partitioning



Theorem: on each iteration of the for loop:

For any j:

- a[j] ≤ pivot,   if j<nextLeft
- a[j] > pivot,   if nextLeft≤j<i

Corollary: the stupid algorithm works

- We run until i gets up to the end (where the pivot element is)
- Everything before nextLeft is ≤ pivot
- Everything from nextLeft to pivot is > pivot
- Then we swap nextLeft (which is > pivot) with the pivot
- So we end up with:
  - Before nextLeft is ≤ pivot
  - nextLeft is the pivot
  - After nextLeft is > the pivot
- Yay!

In-place partitioning

9	3	7	4	8

Pivot around 7

- pIndex = 2
- pValue = 7

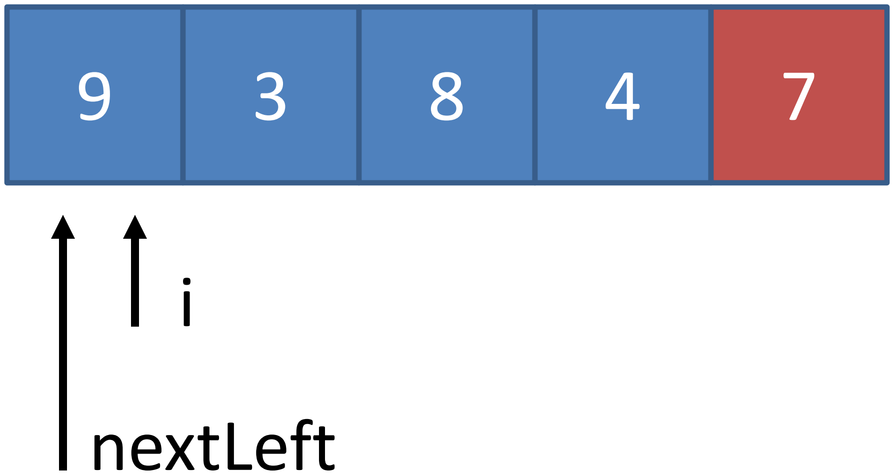
In-place partitioning

Swap pivot to end

Pivot around 7

- pIndex = 2
- pValue = 7

In-place partitioning



start i, nextLeft at the beginning

Pivot around 7

- pValue = 7

In-place partitioning

a[i]<=pValue?

Pivot around 7

- pValue = 7

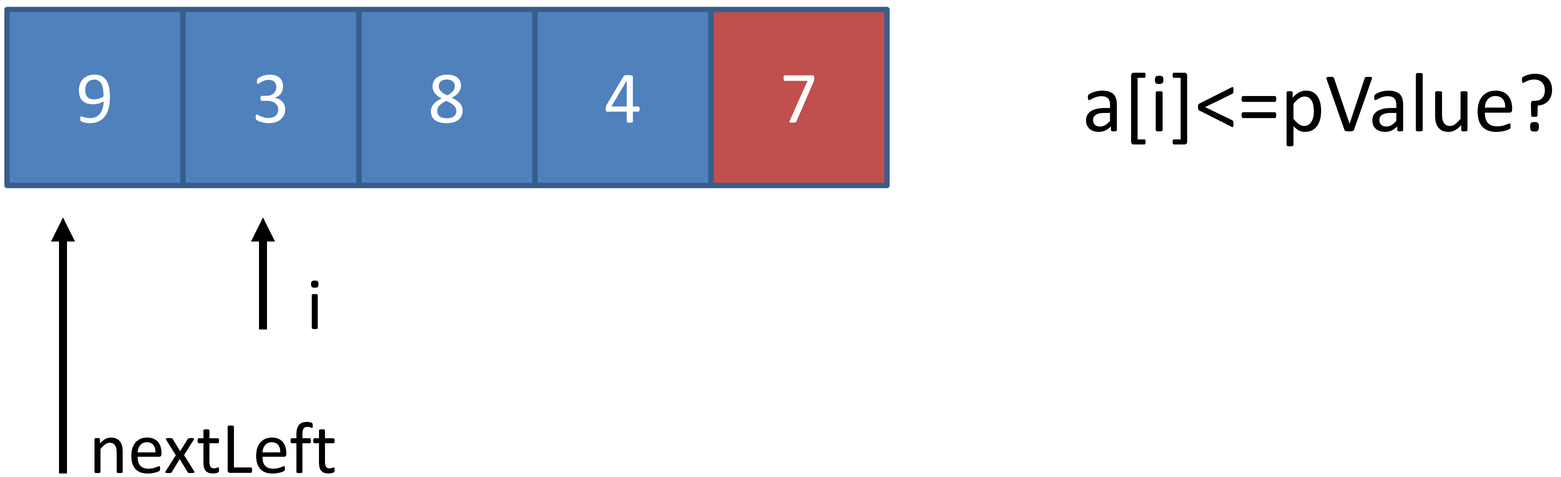
In-place partitioning

no – move on

Pivot around 7

- pValue = 7

In-place partitioning



Pivot around 7

- pValue = 7

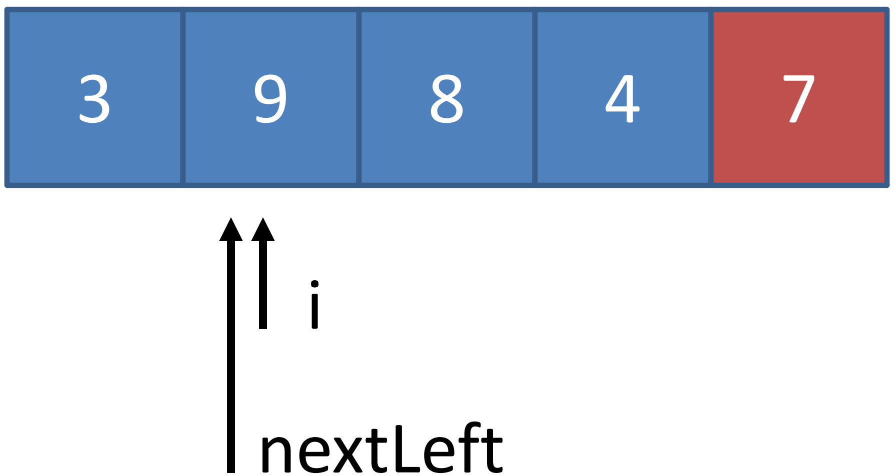
In-place partitioning

yes – swap with a[nextLeft]

Pivot around 7

- pValue = 7

In-place partitioning

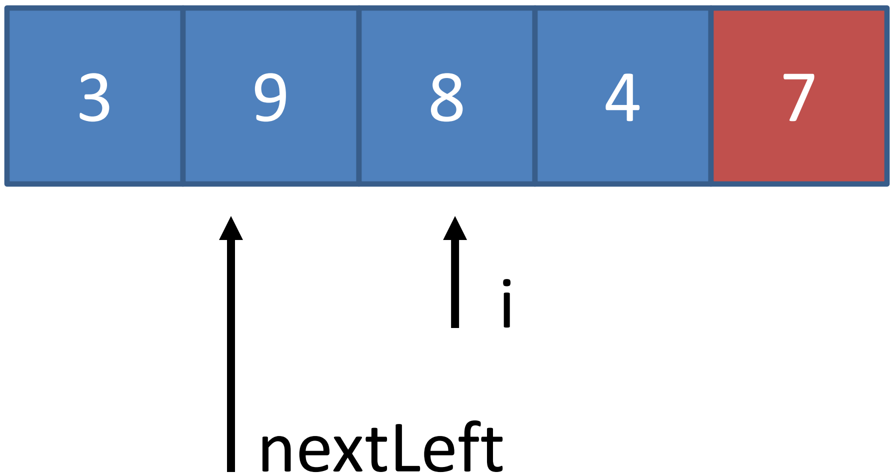


increment nextLeft

Pivot around 7

- pValue = 7

In-place partitioning

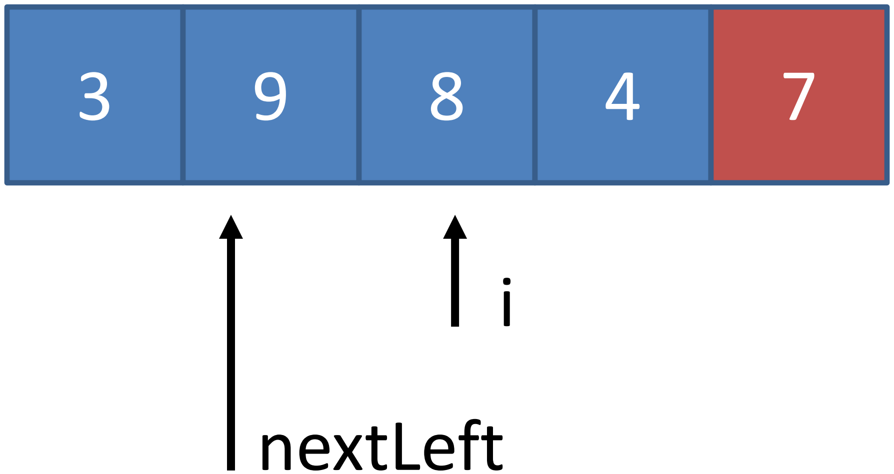


next i

Pivot around 7

- pValue = 7

In-place partitioning

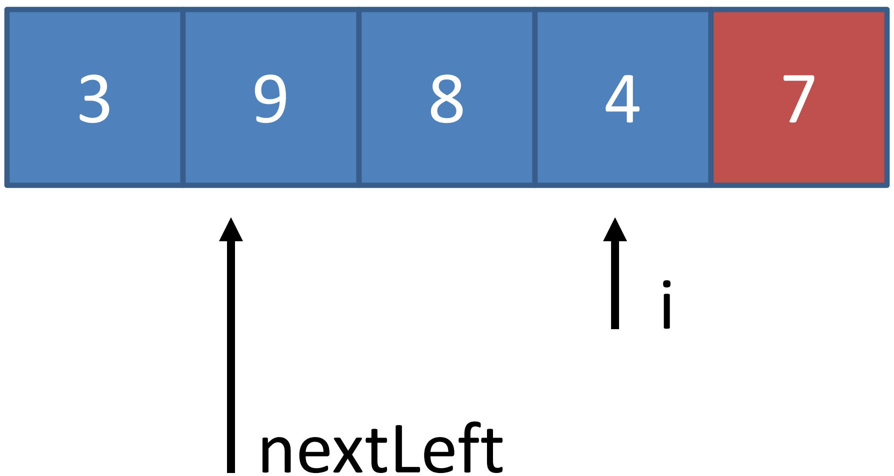


a[i] < pValue?

Pivot around 7

- pValue = 7

In-place partitioning

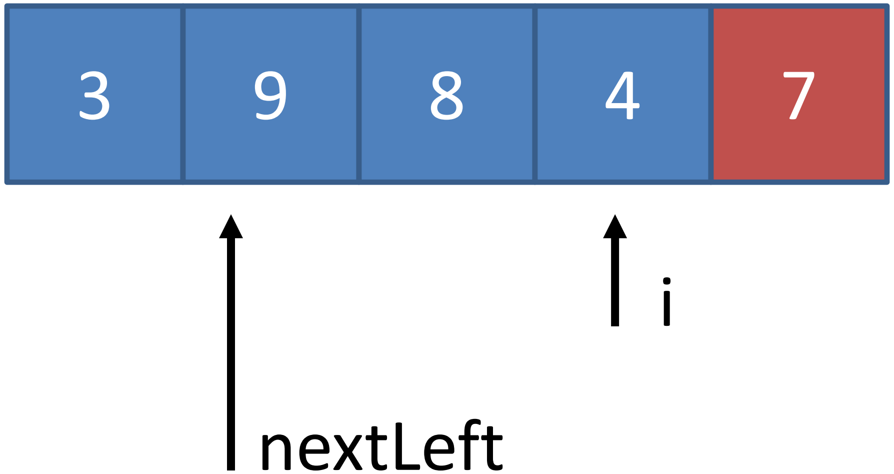


no – move on

Pivot around 7

- pValue = 7

In-place partitioning

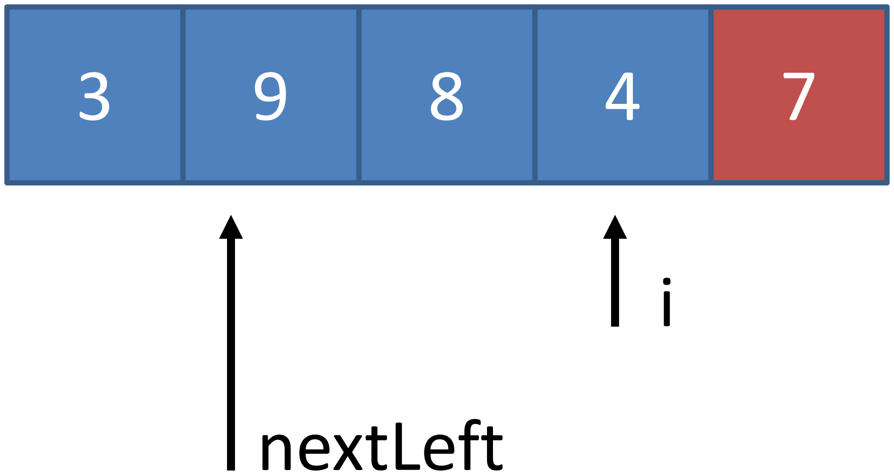


a[i]<=pValue?

Pivot around 7

- pValue = 7

In-place partitioning

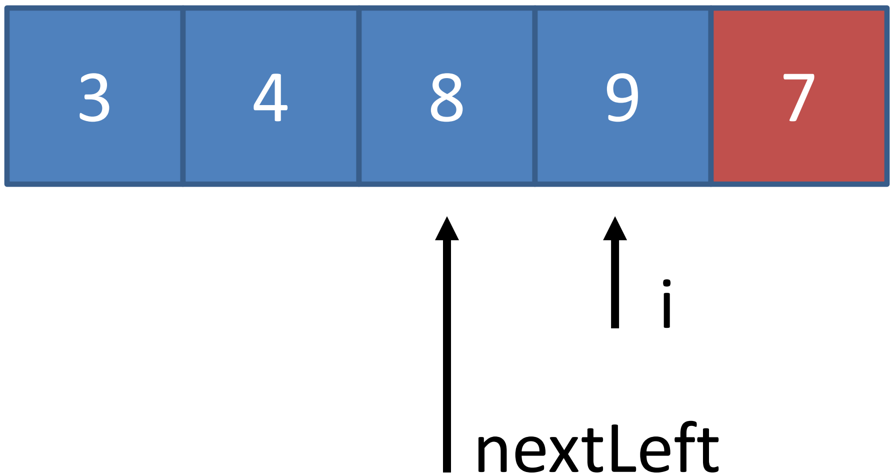


yes - swap

Pivot around 7

- pValue = 7

In-place partitioning

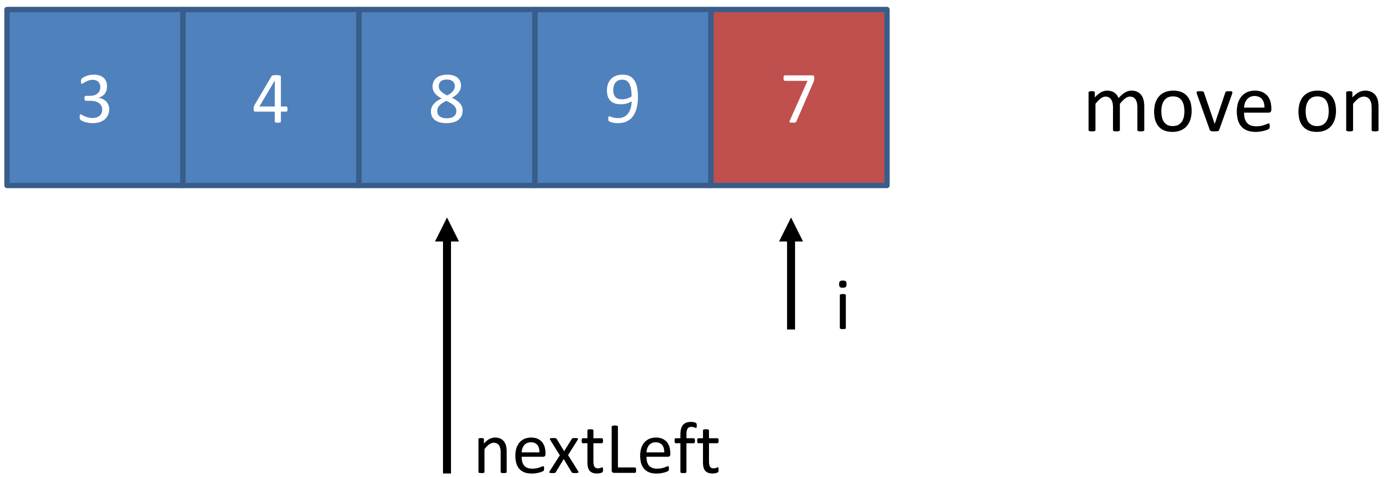


increment nextLeft

Pivot around 7

- pValue = 7

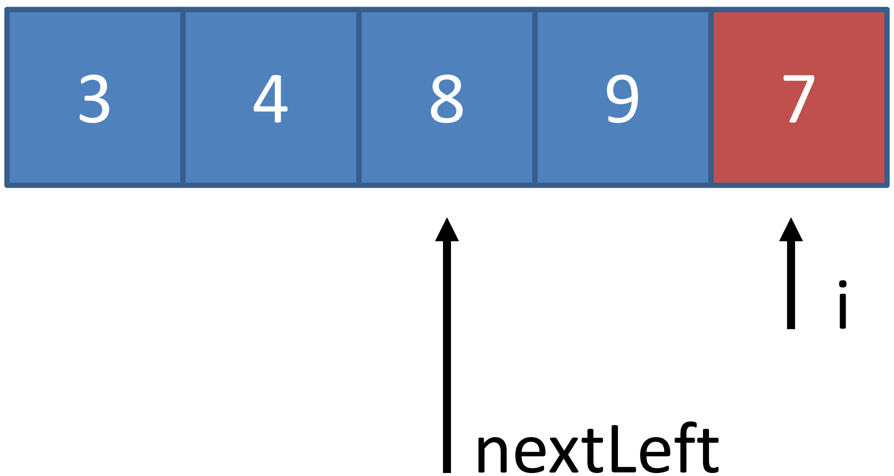
In-place partitioning



Pivot around 7

- pValue = 7

In-place partitioning

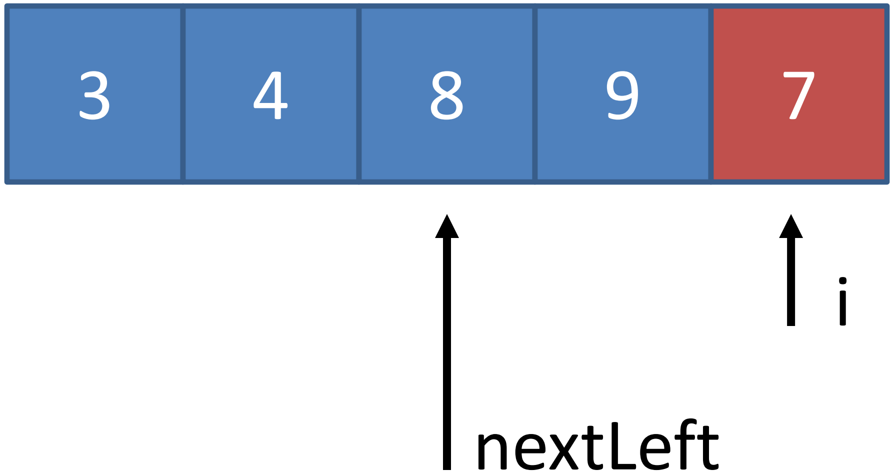


oops –
we’re at the pivot

Pivot around 7

- pValue = 7

In-place partitioning



swap it into nextLeft

Pivot around 7

- pValue = 7

In-place partitioning

3	4	7	9	8

done

Pivot around 7

- pValue = 7

Partitioning subranges of the array

- When quicksort recurses, it will need to partition **just the left and right halves** of the array
- So we need to change partition to take the **region of the array** to be partitioned as an **argument**
  - Specified by its starting and ending indices
- Quicksort also needs Partition to tell it where the split is, so it will return nextLeft

```
Partition(a, pIndex, start, end)
  pValue = a[pIndex]
  // Move pivot to end
  swap a[pIndex]
        with a[end]
  // Move small values left
  nextLeft = start
  for i = start to end-1
     if a[i]<=pValue
       swap a[i]
             with a[nextLeft]
       nextLeft++
  // Move the pivot into place
  swap a[nextLeft]
        with a[end]
  return nextLeft
```

The real quicksort algorithm

```
Quicksort(a)
  QuicksortRecur(a, 0, a.Length-1)

QuicksortRecur(a, start, end)
  if end>start
    pIndex = … select pivot index …
    newIndex = Partition(a, pIndex, start, end)
    QuicksortRecur(a, start, newIndex-1)
    QuicksortRecur(a, newIndex+1, end)
```

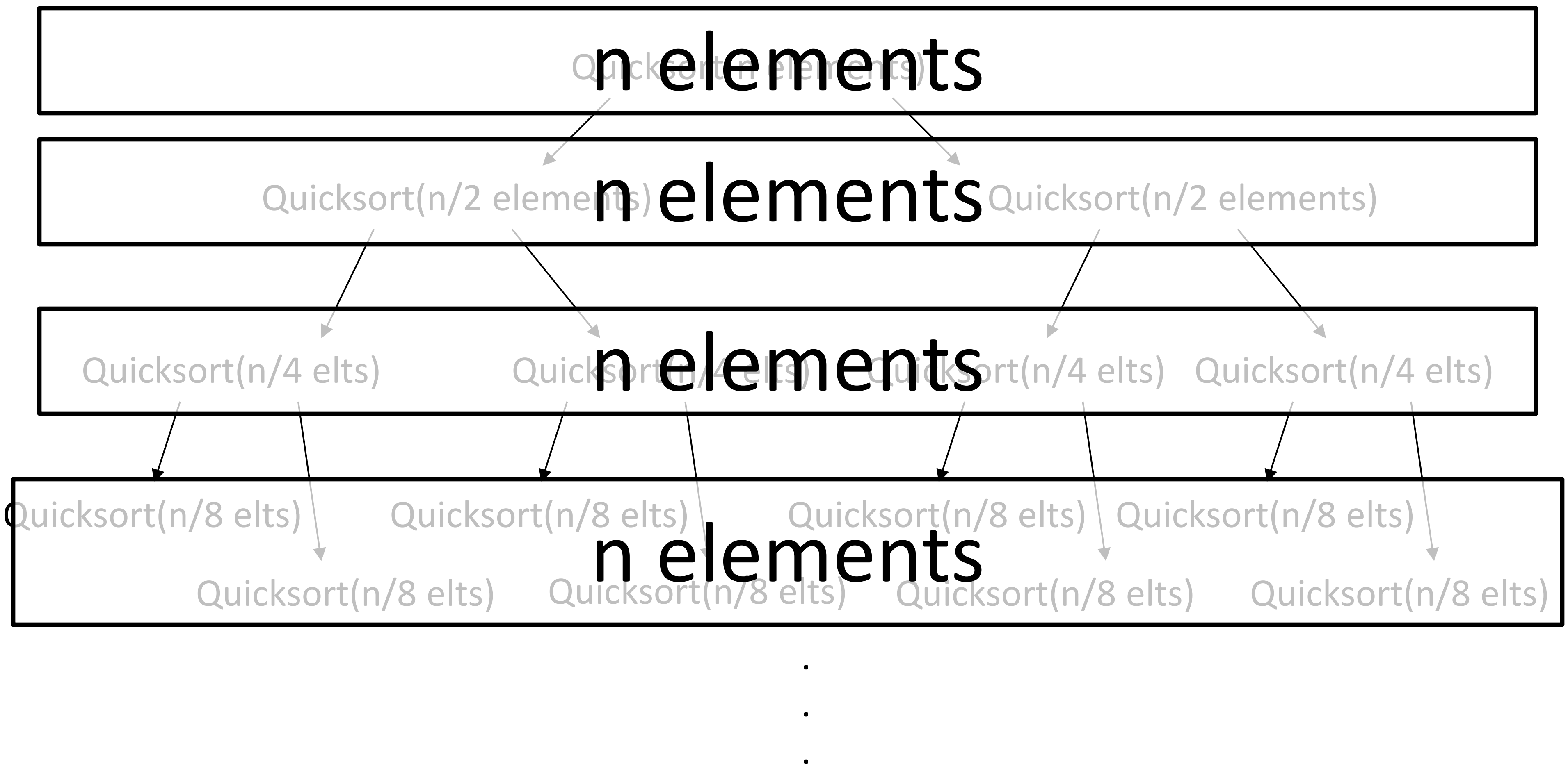
We'll talk about this part shortly …

Analysis (best case)

- The best case is where we **always pick** a pivot that splits the array **exactly in half**

- Then we recurse $\lceil \log_2 n \rceil$ times
  - $\lceil x \rceil$ means "x rounded up to the nearest integer
  - Pronounced "**ceiling** x"
  - $\lfloor x \rfloor$ means round down (read: "**floor** x")

Call tree



Note: this is somewhat hand-wavy, since each level of recursion removes one element (the pivot) before dividing by two. But this still works as an upper bound

Analysis (best case)

- $O(\log n)$ levels of recursion
- n items total partitioned in each level
- $\boldsymbol{O(n\, log\, n)}$ total time

Analysis (worst case)

- The worst case is where we always **pick largest or smallest** element as the pivot
  - All remaining elements on one side
- Then we recurse $\boldsymbol{n-1}$ times!
  - i.e. n total calls

Analysis (worst case)

- $O(n)$ levels of recursion
- $O(n)$ work per level of recursion
- $\boldsymbol{O(n^2)}$ total time

Analysis (average case)

Hand-wavy argument:

- You **have to pick really badly** to get worst-case behavior
- Even if you only split it so that each time:
  - 99% of the elements are on one side
  - 1% are on the the other side
  - You still get $O(\log n)$ levels of recursion
  - It's just closer to $7 \log n$ than $\log n$, but that's still $O(\log n)$

- So even if most of the time you only get the split to **within 1%, you still get $O(n\, log\, n)$** total execution time

Picking the pivot element

- So the choice of the pivot **determines performance**
- How do we choose the pivot element?

Technique 1: fixed choice

- E.g. always pick the **first or last** element in the array
- Usually works fine, but can give you worst-case behavior if the array is already sorted/reverse-sorted
  - Example: first or last element gives you worst-case behavior on **both sorted**, and **reverse-sorted** arrays

Technique 2: median of 3

- Grab **three elements** (e.g. first, last, and middle of the array)
- Find the **median** (the one that's between the other two
- Use that as the pivot
- Works well, but a **sneaky adversary** could cook up an array to force your sort to run slowly
  – Mostly of theoretical interest

Technique 3: Median of the whole array

- Compute the median (middle element) of the whole array
  - Unfortunately, simple methods for computing the media are $O(n^2)$ in the worst case
  - There are $O(n)$ **average case** methods, but they have **large constant factors**
- This is called **median sort**

Technique 4: choose randomly

- This is what people generally use in practice
- $O(n \log n)$ behavior on average
- Doesn't depend on the input
  - So there's **no worst-case input**
  - Just **worst-case luck**

Reading

- Read CLR chapter on Quicksort (chapter 8/9, depending on edition)