

QUICKSORT

David Kauchak
CS 62 – Fall 2026

1

Admin

Wordle Assignment

No office hours Thursday (reach out, I can meet other times)

No mentor hours over fall break

2

swap

```
private static <E extends Comparable<E>> void swap(E[] vals, int first, int second){
    E temp = vals[first];
    vals[first] = vals[second];
    vals[second] = temp;
}
```

What does this method do?

3

Partition

```
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }

    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

what does this method do?

4

... 5 7 1 2 8 4 3 6 ...

↑ start ↑ end

```
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

5

... 5 7 1 2 8 4 3 6 ...

↑ start ↑ end

lessThanIndex

```
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

6

... 5 7 1 2 8 4 3 6 ...

↑ start ↑ end

lessThanIndex i

```
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

7

... 5 7 1 2 8 4 3 6 ...

↑ start ↑ end

lessThanIndex i

```
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

8

Diagram illustrating the partitioning process. The array contains elements: ... 5 7 1 2 8 4 3 6 The pivot is 6. The start index is 5 and the end index is 3. The lessThanIndex pointer is at index 0. The code block highlights the logic for finding elements less than the pivot.

```
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

9

Diagram illustrating the partitioning process. The array contains elements: ... 5 7 1 2 8 4 3 6 The pivot is 6. The start index is 5 and the end index is 3. The lessThanIndex pointer is at index 1. The code block highlights the logic for finding elements less than the pivot.

```
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

10

Diagram illustrating the partitioning process. The array contains elements: ... 5 7 1 2 8 4 3 6 The pivot is 6. The start index is 5 and the end index is 3. The lessThanIndex pointer is at index 2. The code block highlights the logic for finding elements less than the pivot.

```
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

11

Diagram illustrating the partitioning process. The array contains elements: ... 5 7 1 2 8 4 3 6 The pivot is 6. The start index is 5 and the end index is 3. The lessThanIndex pointer is at index 3. The code block highlights the logic for finding elements less than the pivot.

```
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

12

Diagram illustrating the partitioning process. The array contains elements: ... 5 7 1 2 8 4 3 6 The 'start' pointer is at index 5 (value 3) and the 'end' pointer is at index 3 (value 6). The 'lessThanIndex' pointer is at index 1 (value 1) and the current element 'i' is at index 1 (value 1). The element 6 is highlighted in a red box.

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

13

Diagram illustrating the partitioning process. The array contains elements: ... 5 1 7 2 8 4 3 6 The 'start' pointer is at index 5 (value 3) and the 'end' pointer is at index 3 (value 6). The 'lessThanIndex' pointer is at index 1 (value 1) and the current element 'i' is at index 2 (value 7). The element 6 is highlighted in a red box.

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

14

Diagram illustrating the partitioning process. The array contains elements: ... 5 1 7 2 8 4 3 6 The 'start' pointer is at index 5 (value 3) and the 'end' pointer is at index 3 (value 6). The 'lessThanIndex' pointer is at index 1 (value 1) and the current element 'i' is at index 3 (value 2). The element 6 is highlighted in a red box.

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

15

Diagram illustrating the partitioning process. The array contains elements: ... 5 1 7 2 8 4 3 6 The 'start' pointer is at index 5 (value 3) and the 'end' pointer is at index 3 (value 6). The 'lessThanIndex' pointer is at index 2 (value 7) and the current element 'i' is at index 4 (value 8). The element 6 is highlighted in a red box.

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

16

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

17

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;
    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

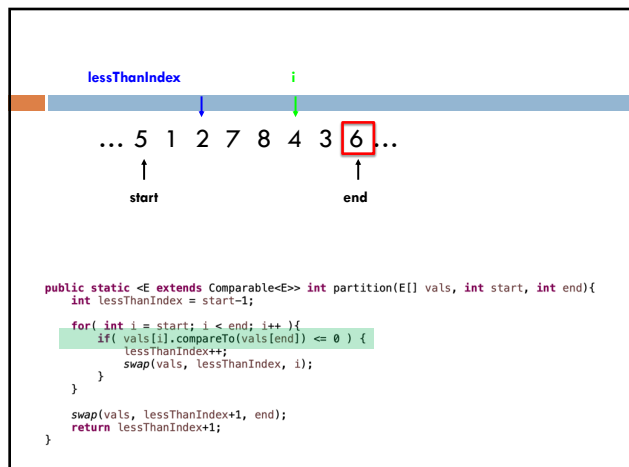
18

What's happening?

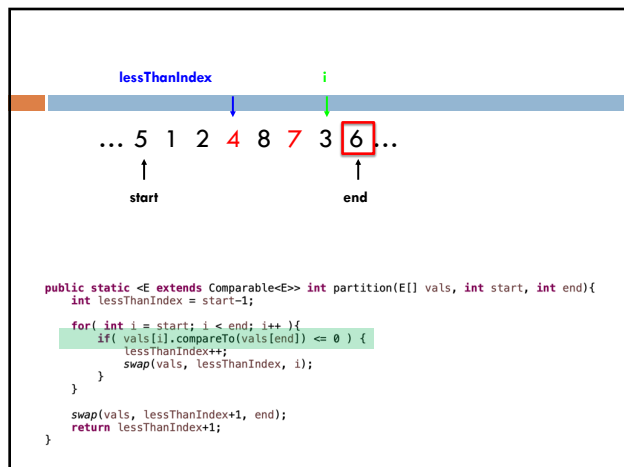
19

≤ pivot > pivot unprocessed

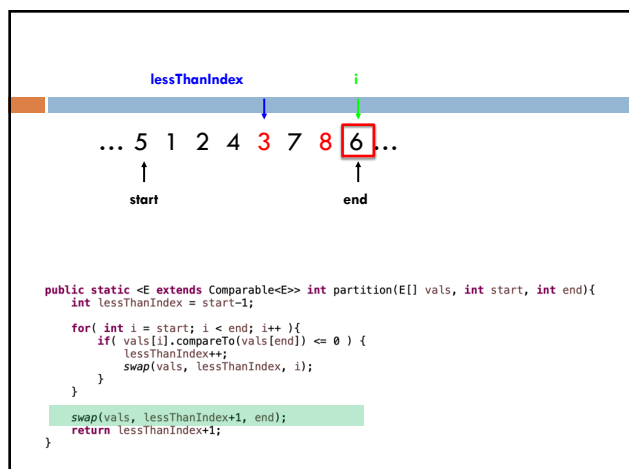
20



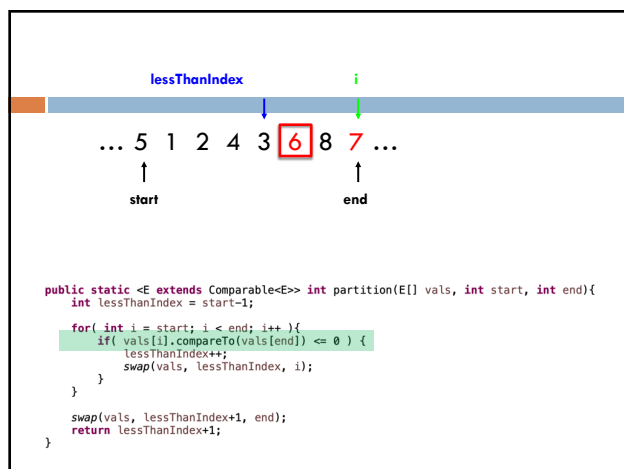
21



22



23



24

```

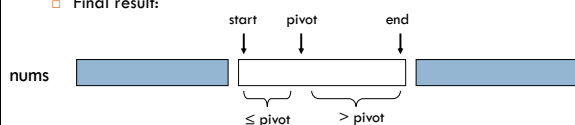
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }

    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

- `vals[end]` is called the *pivot*
- Partitions the elements `nums[start...end-1]` into two sets, those $\leq$ pivot and those $>$ pivot
- Operates in place
- Final result:



25

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }

    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

5	3	6	2	4	0	4
---	---	---	---	---	---	---

What would be the result if we called partition on this list?

26

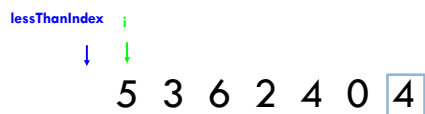
```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }

    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```



27

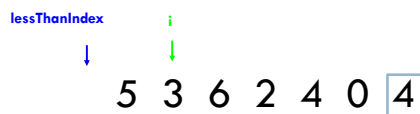
```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }

    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```



28

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

lessThanIndex

↓

3 5 6 2 4 0 4

29

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

lessThanIndex

↓

3 2 6 5 4 0 4

30

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

lessThanIndex

↓

3 2 4 5 6 0 4

31

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

lessThanIndex

↓

3 2 4 0 6 5 4

32

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

3 2 4 0 4 5 6

33

Partition running time?

$n = \text{end} - \text{start}$

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

34

Partition running time?

$O(n)$

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

35

Quicksort

How can we use this method to sort?

```

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }
    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}

```

36

Quicksort

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}

public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    for( int i = start; i < end; i++ ){
        if( vals[i].compareTo(vals[end]) <= 0 ){
            lessThanIndex++;
            swap(vals, lessThanIndex, i);
        }
    }

    swap(vals, lessThanIndex+1, end);
    return lessThanIndex+1;
}
```

37

Quicksort

8 5 1 3 6 2 7 4

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

38

Quicksort

8 5 1 3 6 2 7 4

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

39

Quicksort

1 3 2 4 6 8 7 5

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

40

Quicksort

1 3 2 4 6 8 7 5

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

41

Quicksort

1 3 2 4 6 8 7 5

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

42

Quicksort

1 2 3 4 6 8 7 5

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

43

Quicksort

1 2 3 4 6 8 7 5

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

44

Quicksort

1 2 3 4 6 8 7 5

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

45

Quicksort

1 2 3 4 6 8 7 5

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

46

Quicksort

1 2 3 4 6 8 7 5

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

47

Quicksort

1 2 3 4 5 8 7 6

What happens here?

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

48

Quicksort

1 2 3 4 5 8 7 6

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

49

Quicksort

1 2 3 4 5 8 7 6

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

50

Quicksort

1 2 3 4 5 6 7 8

```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

51

Quicksort

1 2 3 4 5 6 7 8

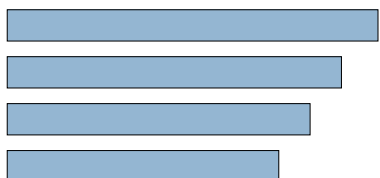
```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

52

Running time of Quicksort?

Worst case?

Each call to Partition splits the array into an empty array and n-1 array



53

Quicksort: Worst case running time

$$n-1 + n-2 + n-3 + \dots + 1 = O(n^2)$$

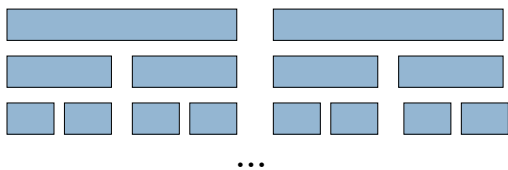
When does this happen?

- sorted
- reverse sorted
- near sorted/reverse sorted

54

Quicksort best case?

Each call to Partition splits the array into two equal parts



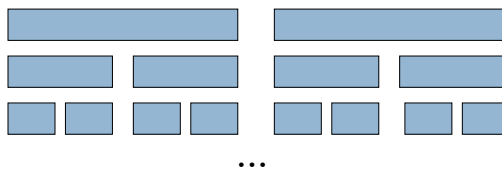
How much work is done at each "level", i.e. running time of a level?

$O(n)$

55

Quicksort best case?

Each call to Partition splits the array into two equal parts



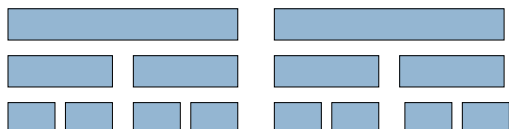
How many levels are there?

Similar to mergesort, each call to Partition will throw away half the data until we're down to one element: $\log_2 n$ levels

56

Quicksort best case?

Each call to Partition splits the array into two equal parts



...

Overall runtime?

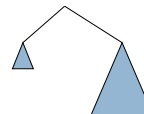
$O(n \log n)$

57

Quicksort Average case?

Two intuitions

- As long as the Partition procedure always splits the array into some constant ratio between the left and the right, say L-to-R, e.g. 9-to-1, then we maintain $O(n \log n)$



- As long as we only have a constant number of "bad" partitions intermixed with a "good partition" then we maintain $O(n \log n)$



58

How can we avoid the worst case?

Inject randomness into the data

```
void randomizedPartition(E [] nums, int start, int end){
    int i = randomInt(start, end);
    swap(nums, i, end);
    return partition = partition(nums, start, end);
}
```

Randomized quicksort is average case $O(n \log n)$

59

What is the worst case running time of randomized Quicksort?

$O(n^2)$

We could still get very unlucky and pick "bad" partitions at every step

60

Quicksort properties

Stable?

In-place?

61

Quicksort properties

Stable: possible, but not the way we've written it (and requires more storage!)

In-place: yes!

62

Sorting summarized

	in-place?	stable?	Best	Average	Worst	Notes
Selection	X		$O(n^2)$	$O(n^2)$	$O(n^2)$	n swaps
Insertion	X	X	$O(n)$	$O(n^2)$	$O(n^2)$	use for partially ordered
Merge		X	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	guaranteed
Quick	X		$O(n \log n)$	$O(n \log n)$	$O(n^2)$	fastest in practice

63

Comparable interface

<https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>

Interface Comparable<T>

int compareTo(T other)

- -1: this object is less than other (technically, any negative number)
- 0: this object is equal to other
- 1: this object is greater than other (technically, any positive number)

64

Built-in sorting

Arrays: (<https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>)

```
static void sort(Object[] a)
    Sorts the specified array of objects into ascending order, according to the natural ordering of its elements.

static void sort(Object[] a, int fromIndex, int toIndex)
    Sorts the specified range of the specified array of objects into ascending order, according to the natural ordering of its elements.
```

Collections: (<https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>)

```
static <T extends Comparable? super T> void sort(List<T> list)
    Sorts the specified list into ascending order, according to the natural ordering of its elements.
```

65

Naturally sorting cards

<https://github.com/pomonaCS62-fa2026/LectureCode/tree/main/SortingCards>

SortableCard:

- implements Comparable<SortableCard>
- Utilizes String.compareTo and Integer.compare
- Foreach loop!

naturalSort()

67

Comparator: unnatural sorting

<https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html>

Create a different ordering **without having to modify the class!**

Interface Comparator<T>

```
int compare(T o1, T o2)
-1: o1 is less than o2 (technically, any negative number)
0: o1 is equal to o2
1: o1 is greater than o2 (technically, any positive number)
```

68

Unnatural sorting

Arrays: (<https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>)

```
static <T> void sort(T[] a, Comparator? super T> c)
    Sorts the specified array of objects according to the order induced by the specified comparator.

static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator? super T> c)
    Sorts the specified range of the specified array of objects according to the order induced by the specified comparator.
```

Collections: (<https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>)

```
static <T> void sort(List<T> list, Comparator? super T> c)
    Sorts the specified list according to the order induced by the specified comparator.
```

69

Unnaturally sorting cards

<https://github.com/pomonaCS62-fa2026/LectureCode/blob/main/SortingCards/BridgeCardSort.java>

- Add 20 to aces
- Reverse the suit ordering
- Reverse the number ordering

bridgeOrderingSort

70

Iterator

<https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html>

A way to move through all of the data in a collection

Interface `Iterator<E>`:

- `boolean hasNext()`
- `E next()`

Have we seen this before? How can we iterate through the data?

71

Iterator

```
while( dealer.hasNext() ) {
    Card c = dealer.next();
    System.out.println(c);
    cards.add(new SortableCard(c.getNumber(), c.getSuit()));
}
```

72

Iterator example

```
public static void temp(){
    List<String> list = new ArrayList<String>();

    list.add("bananas");
    list.add("taste");
    list.add("good");

    Iterator<String> iterator = list.iterator();

    while( iterator.hasNext() ) {
        System.out.println(iterator.next());
    }
}
```

What would we see printed?

73

Iterator example

```
public static void temp(){
    List<String> list = new ArrayList<String>();

    list.add("bananas");
    list.add("taste");
    list.add("good");

    Iterator<String> iterator = list.iterator();

    while( iterator.hasNext() ) {
        System.out.println(iterator.next());
    }

    bananas
    taste
    good
```

74

Iterator example

```
public static void temp2(){
    List<String> list = new ArrayList<String>();

    list.add("bananas");
    list.add("taste");
    list.add("good");

    Iterator<String> iterator = list.iterator();
    Iterator<String> iterator2 = list.iterator();

    System.out.println(iterator2.next());

    while( iterator.hasNext() ) {
        System.out.println(iterator.next());
    }

    System.out.println(iterator2.next());
}
```

What would we see printed?

75

Iterator example

```
public static void temp2(){
    List<String> list = new ArrayList<String>();

    list.add("bananas");
    list.add("taste");
    list.add("good");

    Iterator<String> iterator = list.iterator();
    Iterator<String> iterator2 = list.iterator();

    System.out.println(iterator2.next());

    while( iterator.hasNext() ) {
        System.out.println(iterator.next());
    }

    System.out.println(iterator2.next());

    bananas
    bananas
    taste
    good
    taste
```

Each iterator has its own state!

76

Iterable

<https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html>

interface Iterable<E>:

Iterator<E> iterator()

Just a single method that returns an Iterator.

77

Why Iterable??

```
for( SortableCard c: cards ) {
    System.out.println(c);
}
```

Any class that implements the Iterable class can be used in a foreach loop!

78

How to make a class Iterable

- Implement Iterable interface
- Make a private inner class that implements the Iterator interface
- Have the iterator method return an instance of the private inner class

79

An example

<https://github.com/pomonaCS62-fa2026/LectureCode/tree/main/Iterable>

Each instance of the inner class will have its own *count* instance variable

80

Iterator vs. Iterable

Iterators are a useful mechanism for iterating over almost any type of data

Iterators are the thing that do most of the work (and require most of the coding!)

Iterable allows us to use it in a foreach loop and is often just creating an instance of an Iterator

81