QUICKSORT

David Kauchak
CS 62 – Spring 2021

1

Admin

Compression assignment

2

MergeSort

Divide the data in half

Call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

3

MergeSort

1 2 3 4 5 6 7 8
ms(7 1 4 2 6 5 3 8)

1 2 4 7
ms(7 1 4 2)

3 5 6 8
ms(6 5 3 8)

1 7
ms(7 1)

2 4
ms(4 2)

…

7 *ms*(7)

1 *ms*(1)

4 *ms*(4)

2 *ms*(2)

merge!

4

MergeSort: implementation 1

```
public static <E extends Comparable<E>> E[] mergeSort(E[] a) {
    if( a.length <= 1 ) {
        return a;
    } else {
        int mid = a.length/2;
        E[] left = Arrays.copyOfRange(a, 0, mid);
        E[] right = Arrays.copyOfRange(a, mid, a.length);
        E[] sortedLeft = mergeSort(left);
        E[] sortedRight = mergeSort(right);
        return merge(sortedLeft, sortedRight);
    }
}
```

5

MergeSort: implementation 1

```
public static <E extends Comparable<E>> E[] mergeSort(E[] a) {
    if( a.length <= 1 ) {
        return a;
    } else {
        int mid = a.length/2;
        E[] left = Arrays.copyOfRange(a, 0, mid);
        E[] right = Arrays.copyOfRange(a, mid, a.length);
        E[] sortedLeft = mergeSort(left);
        E[] sortedRight = mergeSort(right);
        return merge(sortedLeft, sortedRight);
    }
}
```

requires copying the data

6

MergeSort: implementation 2

```
mergeSortHelper(data, low, high)
   if high-low > 1
      midPoint = low + (high-low)/2

      mergeSortHelper(data, low, mid)
      mergeSortHelper(data, mid, high)

      merge(data, low, mid, high)
```

How is this different?

7

MergeSort: implementation 2

```
mergeSortHelper(data, low, high)
   if high-low > 1
      midPoint = low + (high-low)/2

      mergeSortHelper(data, low, mid)
      mergeSortHelper(data, mid, high)

      merge(data, low, mid, high)
```

- doesn't require the extra copy!
- low/high specify the range we're sorting
- merge = mergeSortHelper(data, 0, data.length)

8

Merge:

merge(data, low, mid, high)

Assume:
- data starting at low up to, but not including, mid is sorted
- data starting at mid up to, but not including, high is sorted

Goal:
- data from low up to, but not including, high is sorted

Note: merge still requires an extra helper array!

9

MergeSort runtime

Divide the data in half

Call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

What is the runtime of mergesort??

10

Merge runtime

```
public static <E extends Comparable<E>> E[] merge(E[] left, E[] right) {
    int size = left.length+right.length;
    E[] result = (E[])new Object[size];

    int lIndex = 0;
    int rIndex = 0;

    for( int i = 0; i < size; i++ ) {
        if( lIndex >= left.length ) { // done with left
            result[i] = right[rIndex];
            rIndex++;
        } else if( rIndex >= right.length ) { // done with right
            result[i] = left[lIndex];
            lIndex++;
        } else if( left[lIndex].compareTo(right[rIndex]) <= 0 ) {
            result[i] = left[lIndex];
            lIndex++;
        } else {
            result[i] = right[rIndex];
            rIndex++;
        }
    }

    return result;
}
```

What is the runtime of merge?

11

Merge runtime

```
public static <E extends Comparable<E>> E[] merge(E[] left, E[] right) {
    int size = left.length+right.length;
    E[] result = (E[])new Object[size];

    int lIndex = 0;
    int rIndex = 0;

    for( int i = 0; i < size; i++ ) {
        if( lIndex >= left.length ) { // done with left
            result[i] = right[rIndex];
            rIndex++;
        } else if( rIndex >= right.length ) { // done with right
            result[i] = left[lIndex];
            lIndex++;
        } else if( left[lIndex].compareTo(right[rIndex]) <= 0 ) {
            result[i] = left[lIndex];
            lIndex++;
        } else {
            result[i] = right[rIndex];
            rIndex++;
        }
    }

    return result;
}
```

O(left.length + right.length) = O(n)

12

MergeSort runtime

Divide the data in half

Call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

Ignoring the cost of the recursive call, how much work is done per call of MergeSort?

13

MergeSort runtime

Divide the data in half

Call MergeSort on each half (resulting in two sorted halves)

Merge the two halves

Ignoring the cost of the recursive call, how much work is done per call of MergeSort? O(n)

14

Mergesort runtime

work done

ms(length=n) n

15

Mergesort runtime

work done

ms(length=n) n

ms(n/2) ms(n/2)

16

Mergesort runtime

work done

ms(length=n) n

ms(n/2) ms(n/2)

How much work is done for these calls?

17

Mergesort runtime

work done

ms(length=n) n

ms(n/2) ms(n/2) n/2+n/2 = n

How much work is done for these calls?

18

Mergesort runtime

work done

ms(length=n) n

ms(n/2) ms(n/2) n

ms(n/4) ms(n/4) ms(n/4) ms(n/4)

19

Mergesort runtime

work done

ms(length=n) n

ms(n/2) ms(n/2) n

ms(n/4) ms(n/4) ms(n/4) ms(n/4)

How much work is done for these calls?

20

Mergesort runtime

	work done
ms(length=n)	n
ms(n/2) ms(n/2)	n
ms(n/4) ms(n/4) ms(n/4) ms(n/4)	n

How much work is done for these calls?

21

Mergesort runtime

	work done
ms(length=n)	n
ms(n/2) ms(n/2)	n
ms(n/4) ms(n/4) ms(n/4) ms(n/4)	n
…	…

How many levels are there?

22

Mergesort runtime

	work done
ms(length=n)	n
ms(n/2) ms(n/2)	n
ms(n/4) ms(n/4) ms(n/4) ms(n/4)	n
…	…

How many levels are there?

23

We've seen this before…

Each level down we decrease the size by 2:

$$length = \frac{n}{2^{level}}$$

24

We've seen this before…

Each level down we decrease the size by 2:

$$length = \frac{n}{2^{level}}$$

We stop when the length is 1:

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

$$2^{level} = n$$

$$level = log_2 n$$

25

MergeSort running time

O(n) work at each level

log n levels

Overall runtime: O(n log n) (best, worst, average)

26

MergeSort properties

Stable?

In-place?

27

MergeSort properties

Stable: yes!

```
public static <E extends Comparable<E>> E[] merge(E[] left, E[] right) {
    int size = left.length+right.length;
    E[] result = (E[])new Object[size];

    int lIndex = 0;
    int rIndex = 0;

    for( int i = 0; i < size; i++ ) {
        if( lIndex >= left.length ) { // done with left
            result[i] = right[rIndex];
            rIndex++;
        } else if( rIndex >= right.length ) { // done with right
            result[i] = left[lIndex];
            lIndex++;
        } else if( left[lIndex].compareTo(right[rIndex]) <= 0 ) {
            result[i] = left[lIndex];
            lIndex++;
        } else {
            result[i] = right[rIndex];
            rIndex++;
        }
    }

    return result;
}
```

28

MergeSort properties

Stable: yes

In-place: sort of

- easy way to implement is not
- very hard to do (but possible)
- in practice, variants of merge sort that combine with insertion sort are in-place

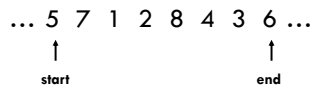
29

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;
}
```

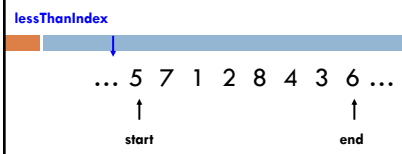
what does this method do?

30



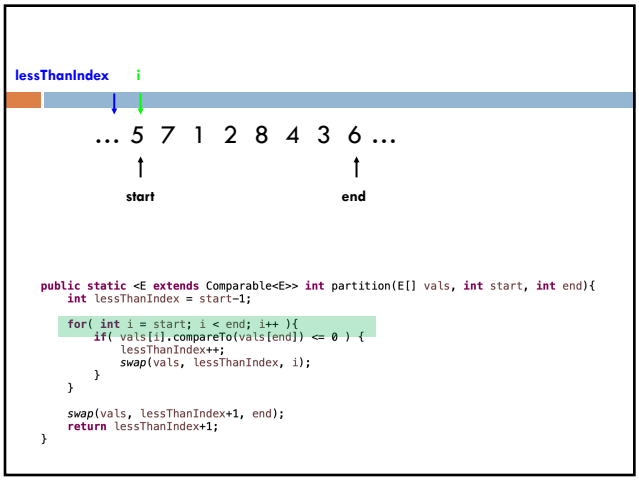

```
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    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;
}
```

31

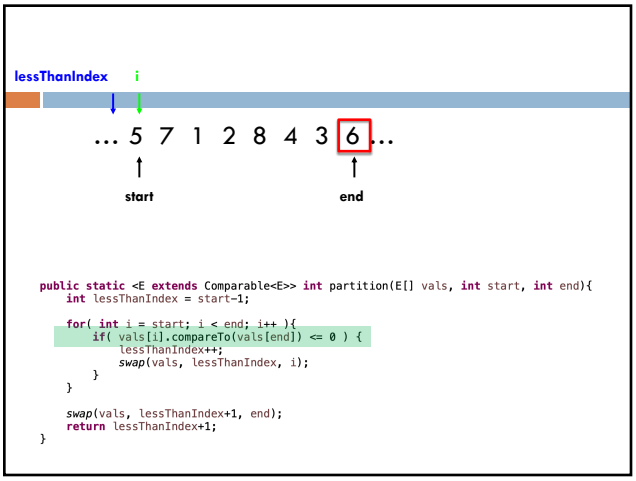



```
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    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;
}
```

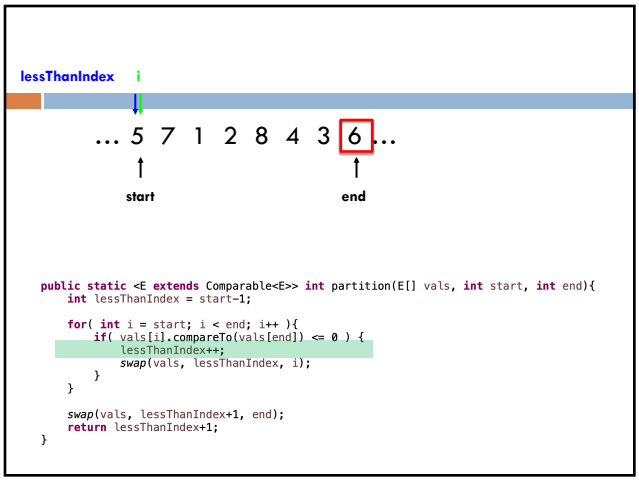
32


lessThanIndex
i
... 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;
}

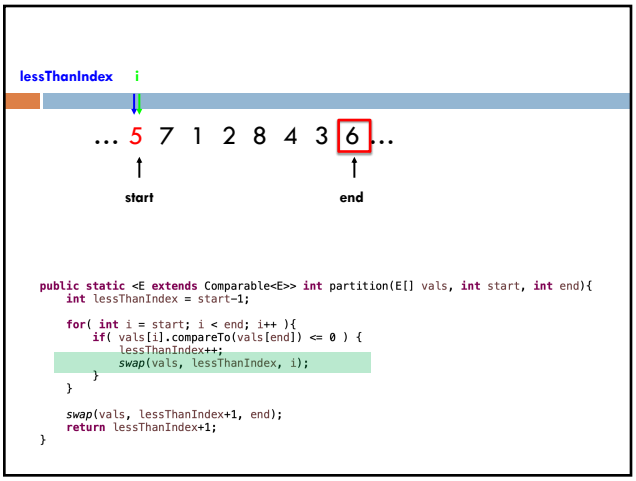
33


lessThanIndex
i
... 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;
}

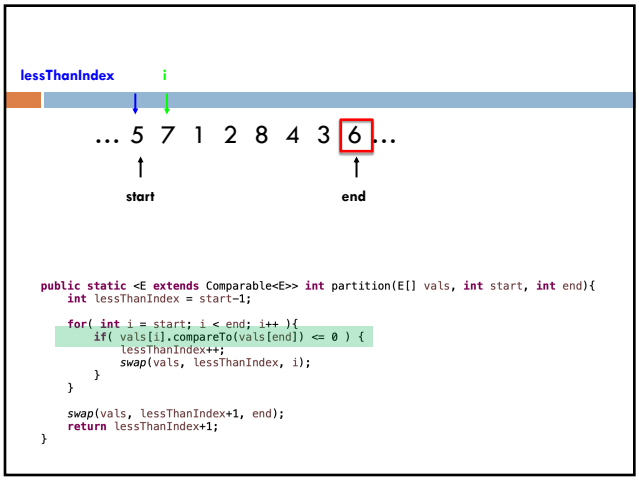
34


lessThanIndex
i
... 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;
}

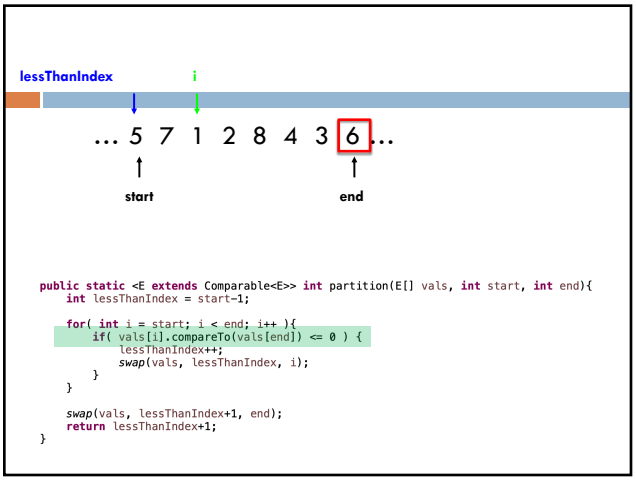
35


lessThanIndex
i
... 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;
}

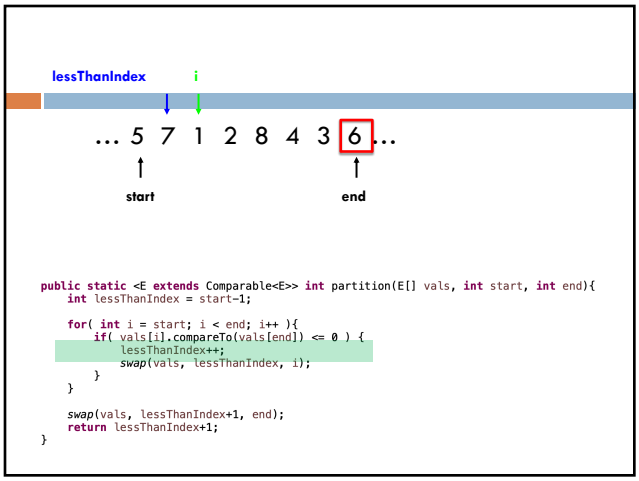
36


lessThanIndex
i
... 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;
}

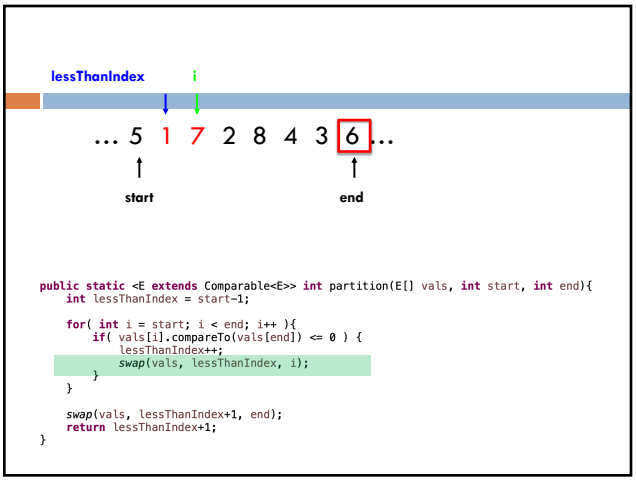
37


lessThanIndex
i
... 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;
}

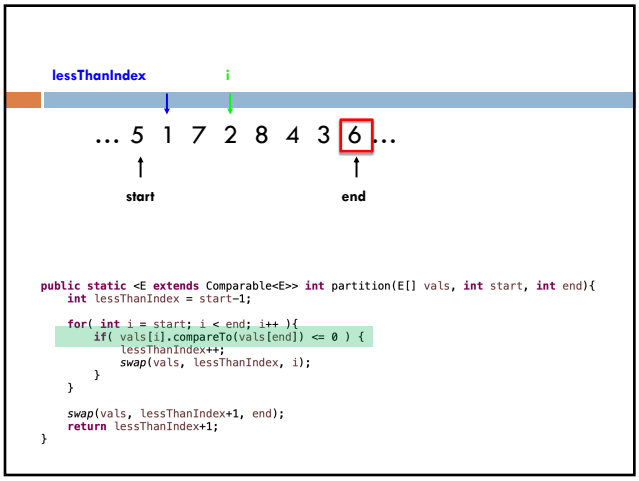
38


lessThanIndex
i
... 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;
}

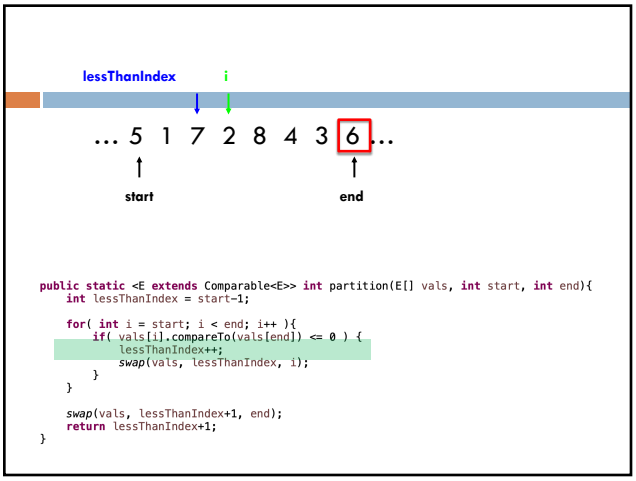
39


lessThanIndex
i
... 5 1 7 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;
}

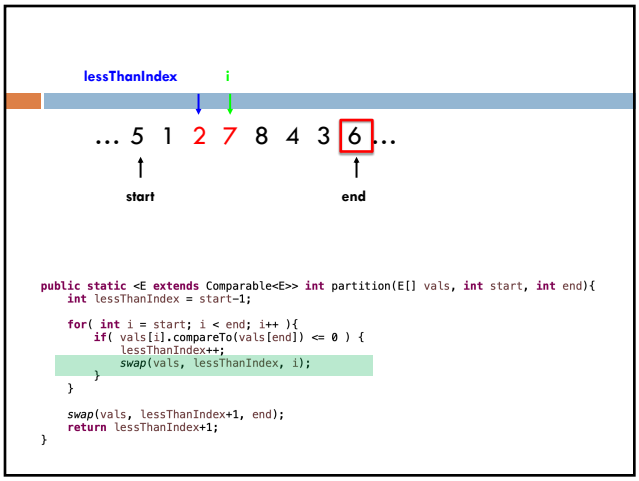
40


lessThanIndex
i
... 5 1 7 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;
}

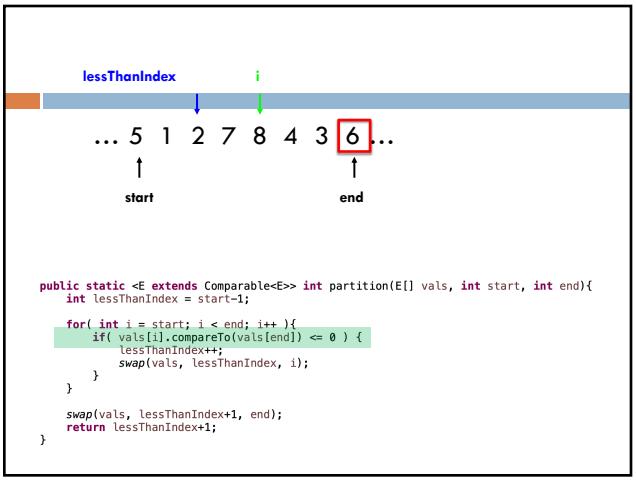
41


lessThanIndex
i
... 5 1 7 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;
}

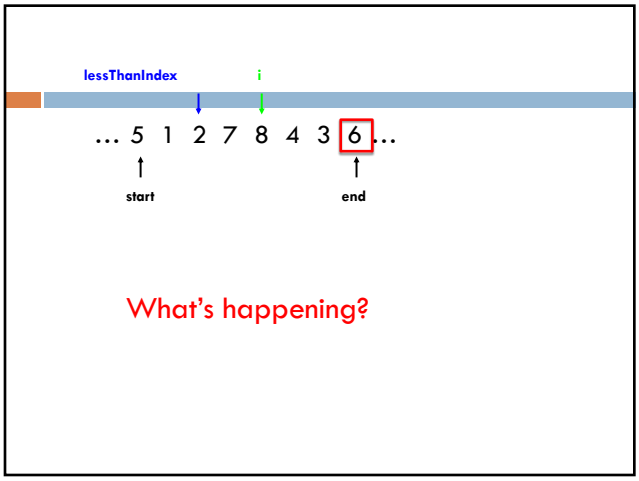
42


lessThanIndex
i
... 5 1 2 7 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;
}

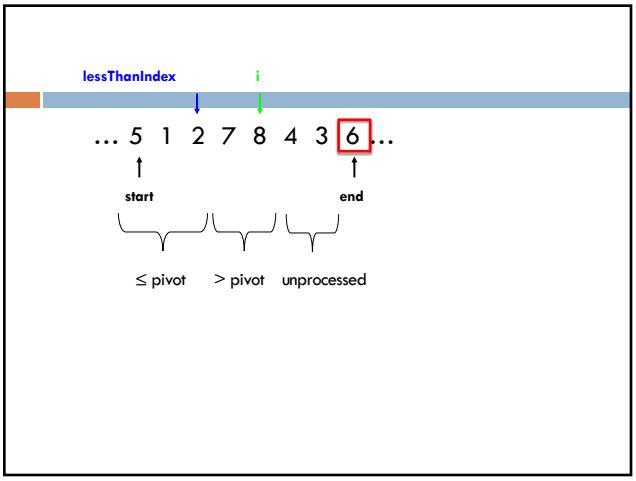
43


lessThanIndex
i
... 5 1 2 7 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;
}

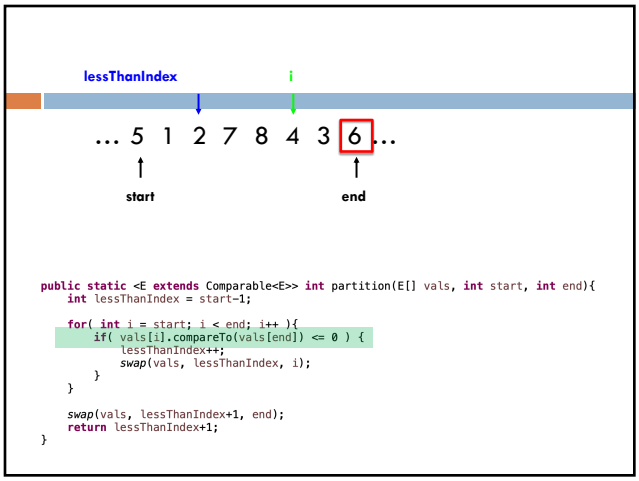
44


lessThanIndex
i
... 5 1 2 7 8 4 3 6 ...
start
end
What's happening?

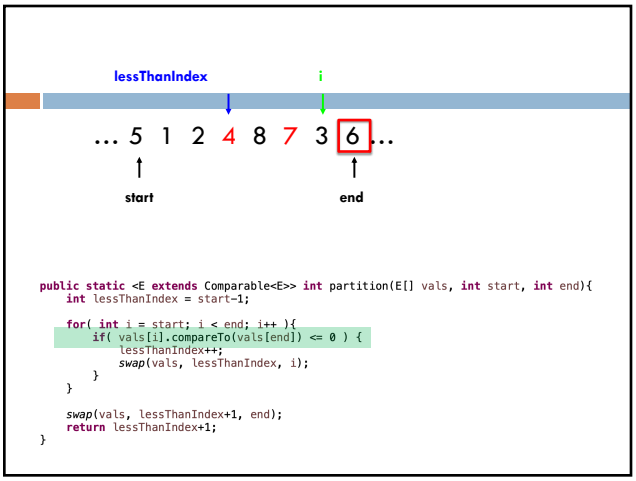
45


lessThanIndex
i
... 5 1 2 7 8 4 3 6 ...
start
end
≤ pivot
> pivot
unprocessed

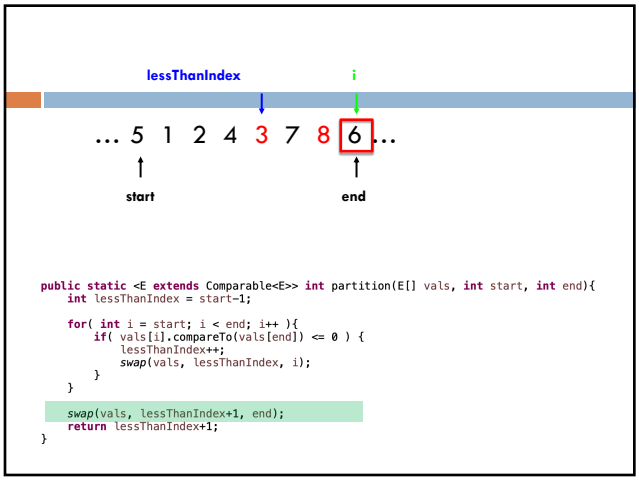
46


lessThanIndex
i
... 5 1 2 7 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;
}

47

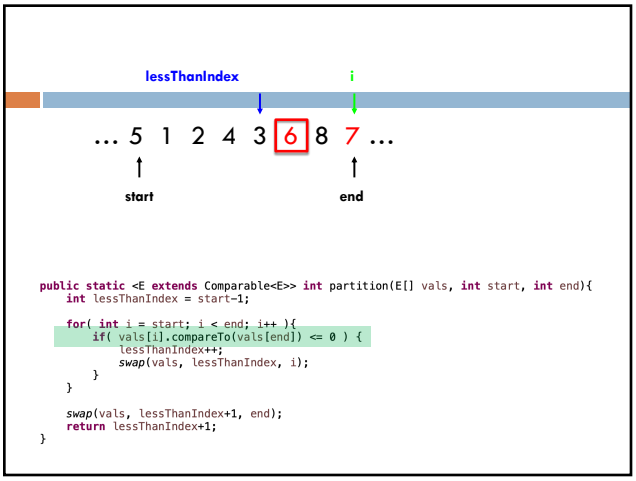

lessThanIndex
i
... 5 1 2 4 8 7 3 6 ...
start
end
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
int lessThanIndex = start-1;
for(int i = start; i < end; i++){
if(vals[i].compareTo(vals[end]) <= 0) {
lessThanIndex++;
swap(vals, lessThanIndex, i);
}
}
swap(vals, lessThanIndex+1, end);
return lessThanIndex+1;
}

48




```
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;
}
```

49

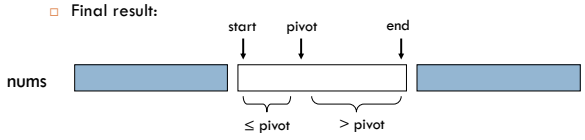



```
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end){
    int lessThanIndex = start-1;

    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;
}
```

50

```
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] in to two sets, those ≤ pivot and those > pivot
- Operates in place
- Final result:



51

Partition running time?

```
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;
}
```

52

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;
}
```

53

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;
}
```

54

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;
}
```

55

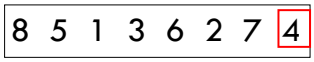
Quicksort

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);
    }
}
```

56

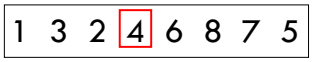
Quicksort




```
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);
    }
}
```

57

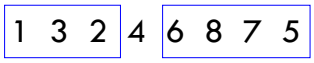
Quicksort




```
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);
    }
}
```

58

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);
    }
}
```

59

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);
    }
}
```

60

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);
    }
}
```

61

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);
    }
}
```

62

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);
    }
}
```

63

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);
    }
}
```

64

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);
    }
}
```

65

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);
    }
}
```

66

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);
    }
}
```

67

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);
    }
}
```

68

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);
    }
}
```

69

Quicksort

1 2 3 4 5 6 7 8

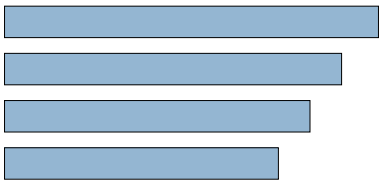
```
void quicksortHelper(E[] vals, int start, int end){
    if( start < end ){
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

70

Running time of Quicksort?

Worst case?

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



71

Quicksort: Worst case running time

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

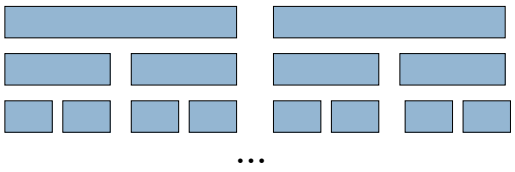
When does this happen?

- sorted
- reverse sorted
- near sorted/reverse sorted

72

Quicksort best case?

Each call to Partition splits the array into two equal parts



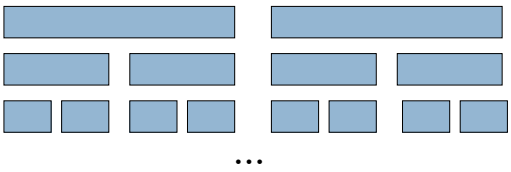
How much work is done at each "level", i.e. running time of a level?

O(n)

73

Quicksort best case?

Each call to Partition splits the array into two equal parts



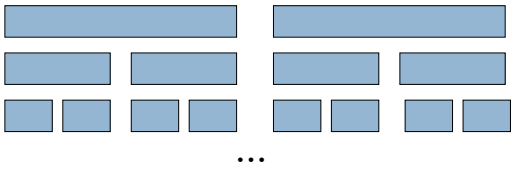
How many levels are there?

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

74

Quicksort best case?

Each call to Partition splits the array into two equal parts



Overall runtime?

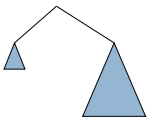
O(n log n)

75

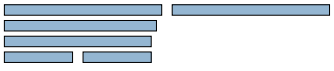
Quicksort Average case?

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)



76

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)

77

What is the wost case running time of randomized Quicksort?

$O(n^2)$

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

78

Quicksort properties

Stable?

In-place?

79

Quicksort properties

Stable: possible, but not the way we've written it (and requires more storage!0

In-place: yes!

80