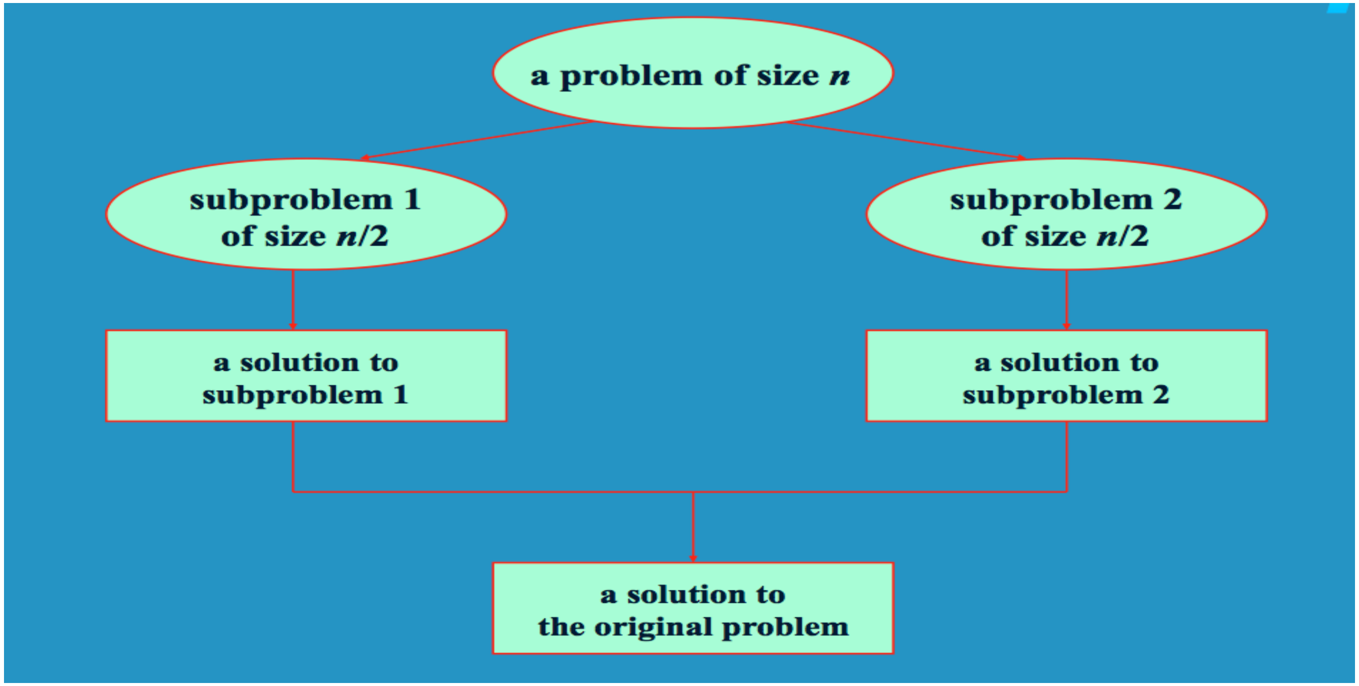
Exchange based sorting algorithms

Bubble Sort

Selection Sort

Quicksort

Sorting and Algorithm Analysis: **Divide and Conquer**



Computer Science 112
Boston University

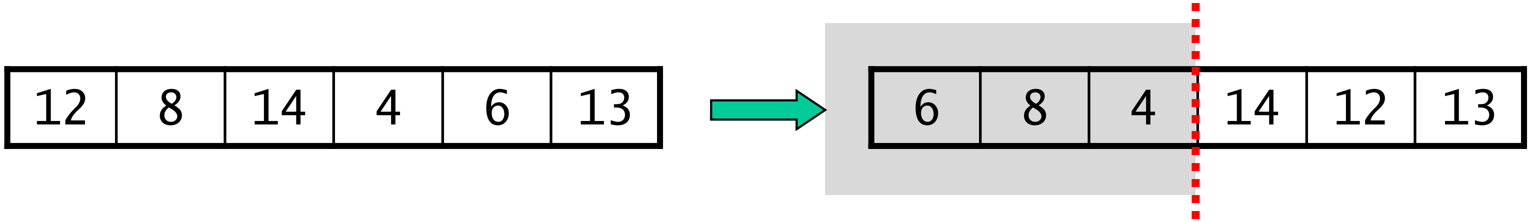
Christine Papadakis-Kanaris

Quicksort

- Quicksort uses an approach based on exchanging out-of-order elements.
- A recursive, divide-and-conquer algorithm:
  - ***divide***: rearrange the elements so that we end up with two subarrays that meet the following criterion:

    each element in the left array *<= each element in the right array*

    example:



12	8	14	4	6	13

6	8	4	14	12	13

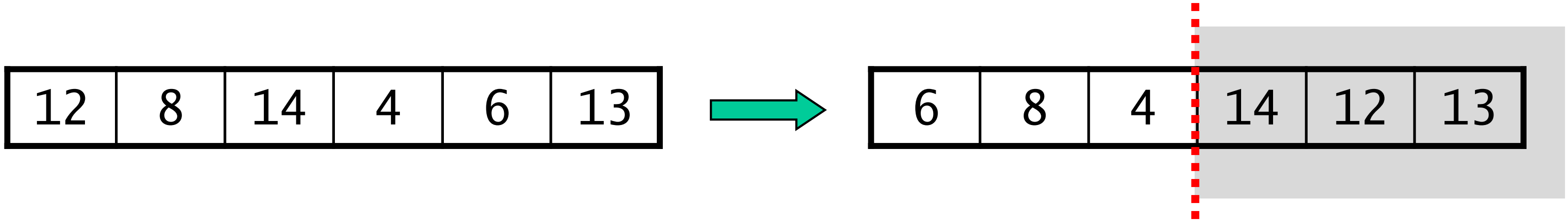
  - *conquer:*
  - *combine:*

Quicksort

- Quicksort uses an approach based on exchanging out-of-order elements.
- A recursive, divide-and-conquer algorithm:
  - ***divide***: rearrange the elements so that we end up with two subarrays that meet the following criterion:

    each element in the left array <= ***each element in the right array***

    example:



12	8	14	4	6	13

6	8	4	14	12	13

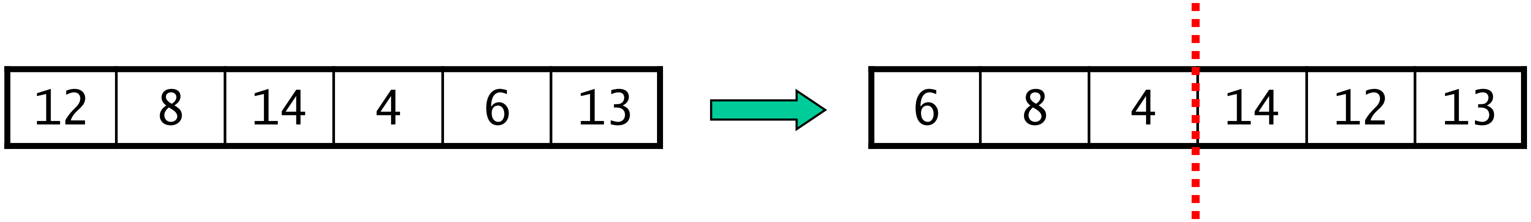
  - *conquer:*
  - *combine:*

Quicksort

- Quicksort uses an approach based on exchanging out-of-order elements.
- A recursive, divide-and-conquer algorithm:
  - *divide:* rearrange the elements so that we end up with two subarrays that meet the following criterion:

    each element in the left array <= each element in the right array

    example:



12	8	14	4	6	13

6	8	4	14	12	13

  - *conquer:* apply quicksort recursively to the subarrays, stopping when a subarray has a single element
  - ***combine***: *nothing needs to be done, because of the criterion used in forming the subarrays*

Partitioning an Array Using a Pivot

- The process that quicksort uses to rearrange the elements is known as *partitioning* the array.
- Partitioning is done using a value known as the *pivot.*
- We rearrange the elements to produce two subarrays:
  - left subarray: all values <= pivot
  - right subarray: all values >= pivot

equivalent to the criterion on the previous page.

Example:

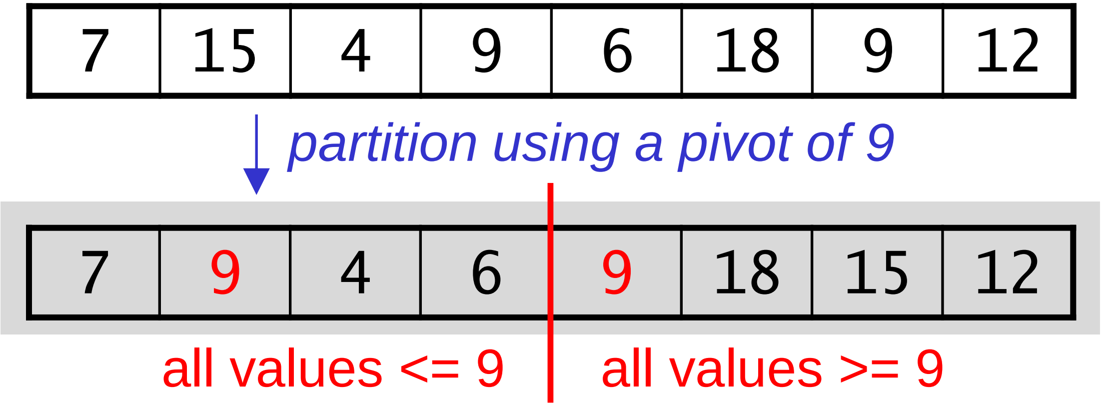
7	15	4	9	6	18	9	12

Partitioning an Array Using a Pivot

- The process that quicksort uses to rearrange the elements is known as *partitioning* the array.
- Partitioning is done using a value known as the *pivot.*
- We rearrange the elements to produce two subarrays:
  - left subarray: all values <= pivot
  - right subarray: all values >= pivot

equivalent to the criterion on the previous page.

Example:



7	15	4	9	6	18	9	12

partition using a pivot of 9

7	9	4	6	9	18	15	12

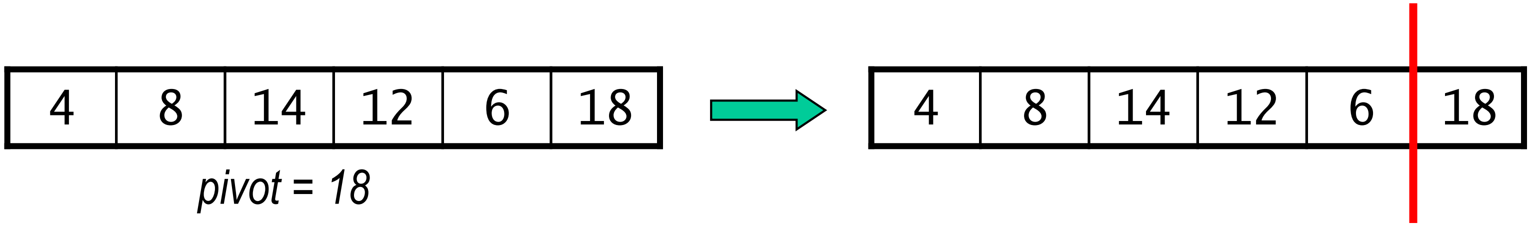
all values <= 9

all values >= 9

- Why did we choose 9 as the pivot?
- Our approach to partitioning is one of several variants.

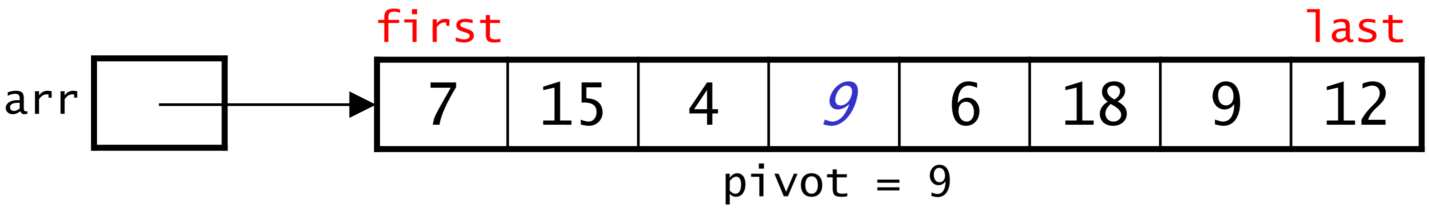
Possible Pivot Values

- First element or last element
  - risky, can lead to terrible worst-case behavior
  - especially poor if the array is almost sorted



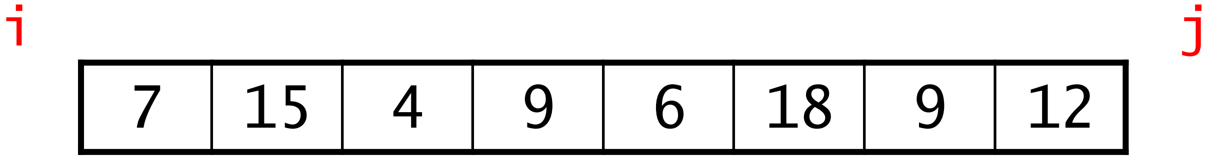
- Middle element (what we will use)
- Randomly chosen element
- Median of three elements
  - left, center, and right elements
  - three randomly selected elements
  - taking the median of three decreases the probability of getting a poor pivot

Partitioning an Array: An Example

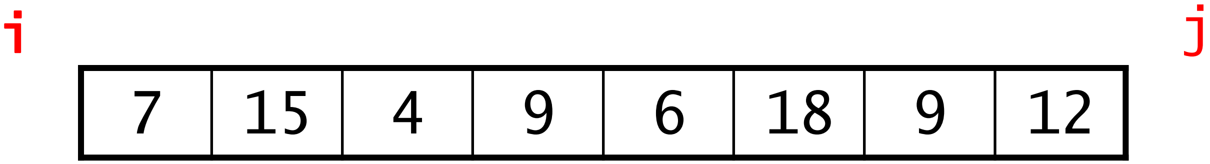


- Maintain indices `i` and `j`, starting them "outside" the array:

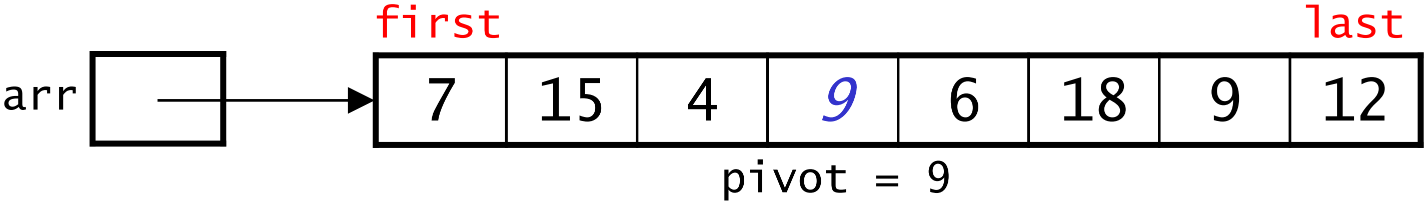
```
i = first - 1
j = last + 1
```



- *Find* "out of place" elements:
  - increment i until **`arr[i] >= pivot`**



Partitioning an Array: An Example



first last

`arr`

7	15	4	*9*	6	18	9	12

`pivot = 9`

- Maintain indices `i` and `j`, starting them “outside” the array:

  `i = first - 1`

  `j = last + 1`

i j

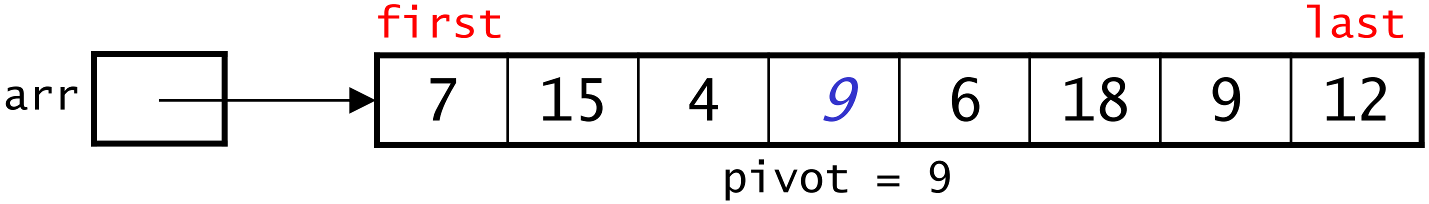
7	15	4	9	6	18	9	12

- *Find* “out of place” elements:
  - increment `i` until `arr[i] >= pivot`
  - decrement `j` until **`arr[j] <= pivot`**

i j

7	15	4	9	6	18	9	12

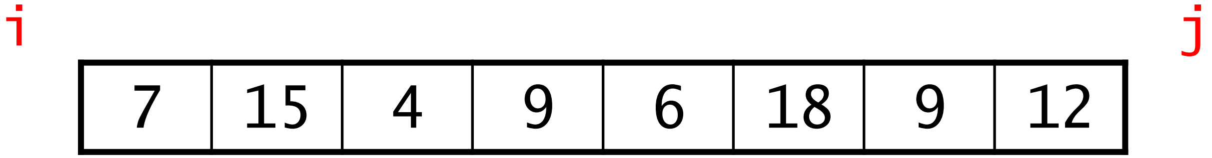
Partitioning an Array: An Example



7	15	4	9	6	18	9	12

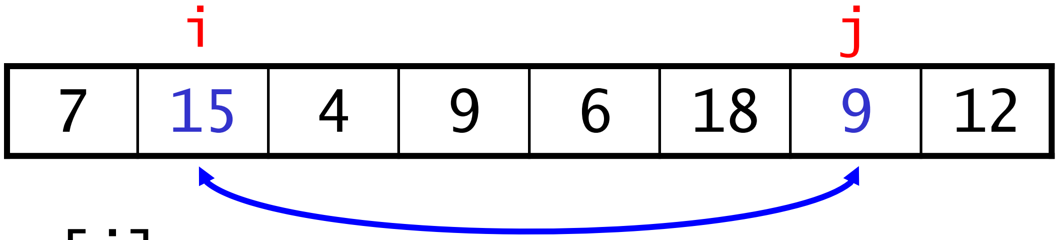
- Maintain indices `i` and `j`, starting them “outside” the array:

```
i = first - 1
j = last + 1
```



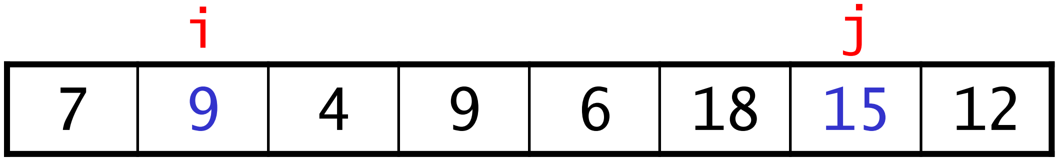
7	15	4	9	6	18	9	12

- *Find* “out of place” elements:
  - increment `i` until `arr[i] >= pivot`
  - decrement `j` until `arr[j] <= pivot`



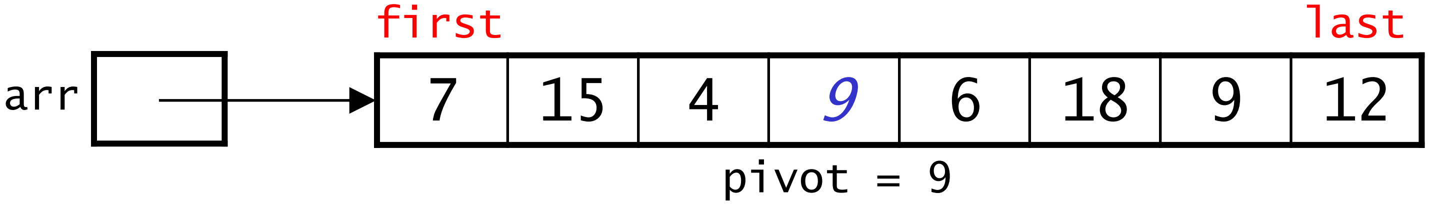
7	15	4	9	6	18	9	12

- *Swap* `arr[i]` and `arr[j]`:



7	9	4	9	6	18	15	12

Partitioning an Array: An Example



7	15	4	*9*	6	18	9	12

- Maintain indices `i` and `j`, starting them “outside” the array:

  `i = first – 1`
  `j = last + 1`

i j

7	15	4	9				12

Note by continuing until <= *pivot*, the pivot may actually swap into a different region!

- *Find* “out of place” elements:
  - increment `i` until `arr[i] >=`
  - decrement `j` until `arr[j] <=`

i j

7	15	4	9	6	18	9	12

- *Swap* `arr[i]` and `arr[j]`:

i j

7	9	4	9	6	18	15	12

Partitioning Example (cont.)

from prev. page:

	i					j	
7	9	4	9	6	18	15	12

- Find:

			i	j			
7	9	4	9	6	18	15	12

- Swap:

			i	j			
7	9	4	6	9	18	15	12

- Find:

			j	i			
7	9	4	6	9	18	15	12

and now the indices have crossed, so we return `j`.

Partitioning Example (cont.)

from prev. page:

	i					j	
7	9	4	9	6	18	15	12

- Find:

			i	j			
7	9	4	9	6	18	15	12

- Swap:

			i	j			
7	9	4	6	9	18	15	12

- Find:

			j	i			
7	9	4	6	9	18	15	12

and now the indices have crossed, so we return `j`.

Partitioning Example (cont.)

from prev. page:

	i					j	
7	9	4	9	6	18	15	12

- Find:

			i	j			
7	9	4	9	6	18	15	12

- Swap:

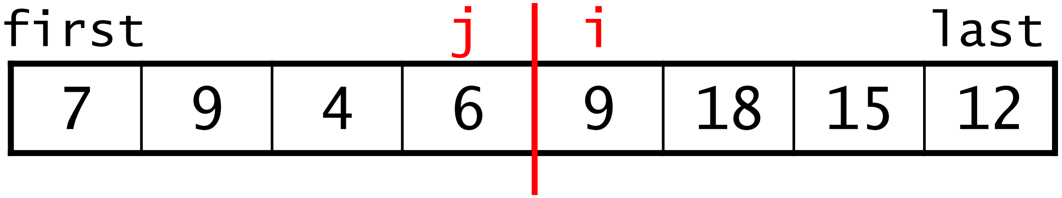
			i	j			
7	9	4	6	9	18	15	12

- Find:

			j	i			
7	9	4	6	9	18	15	12

and now the indices have crossed, so we return `j`.

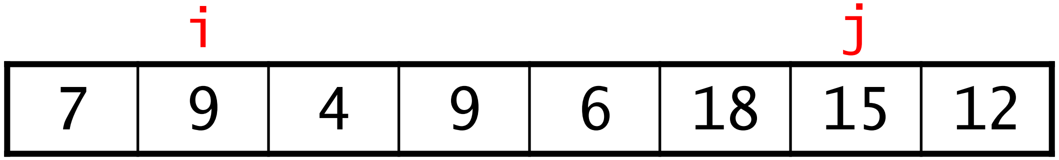
- Subarrays: left = `arr[first:j]`, right = `arr[j+1:last]`



first			j	i			last
7	9	4	6	9	18	15	12

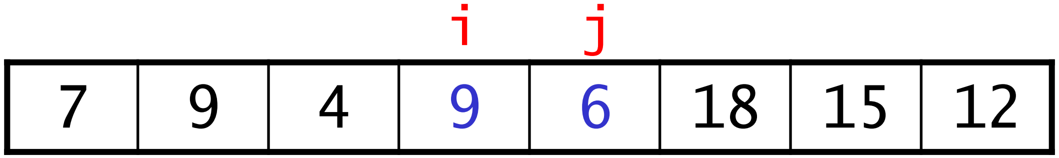
Partitioning Example (cont.)

from prev. page:



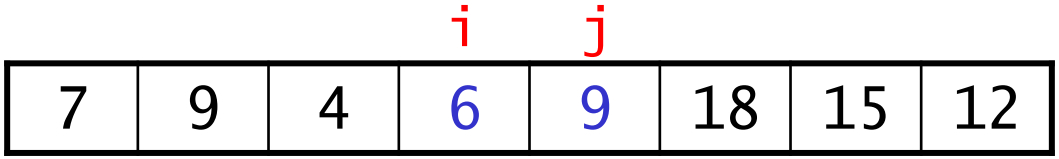
	i					j	
7	9	4	9	6	18	15	12

- Find:



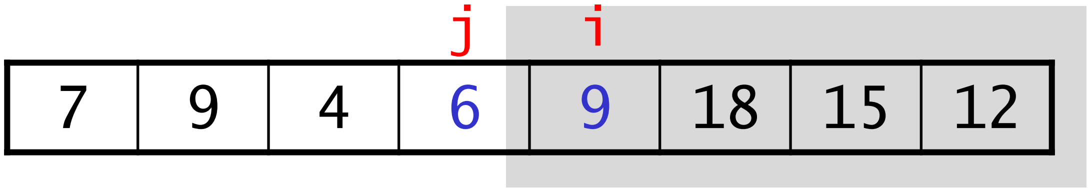
			i	j			
7	9	4	9	6	18	15	12

- Swap:



			i	j			
7	9	4	6	9	18	15	12

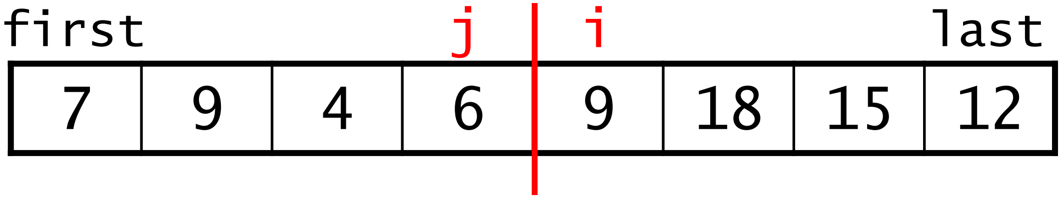
- Find:



			j	i			
7	9	4	6	9	18	15	12

and now the indices have crossed, so we return `j`.

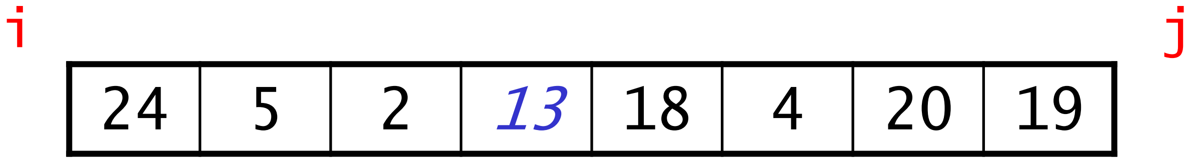
- Subarrays: left = `arr[first:j]`, right = `arr[j+1:last]`



first			j	i			last
7	9	4	6	9	18	15	12

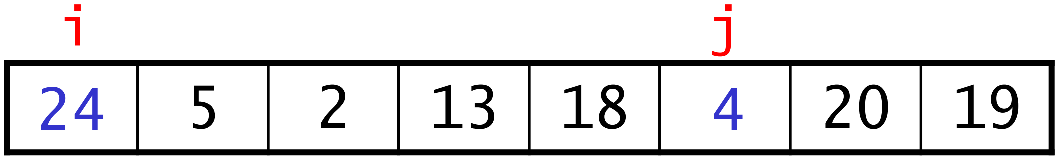
Partitioning Example 2

- Start (pivot = 13):



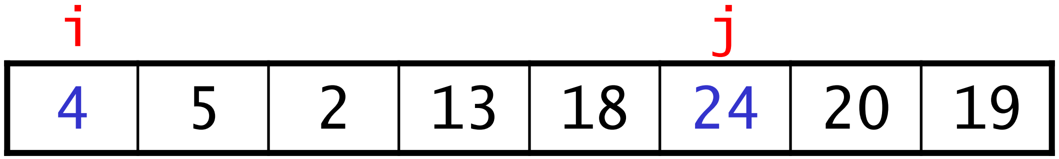
24	5	2	*13*	18	4	20	19

- Find:



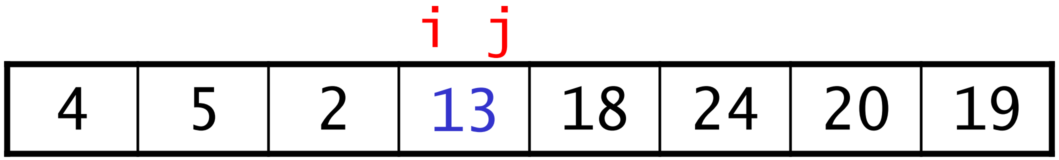
24	5	2	13	18	4	20	19

- Swap:



4	5	2	13	18	24	20	19

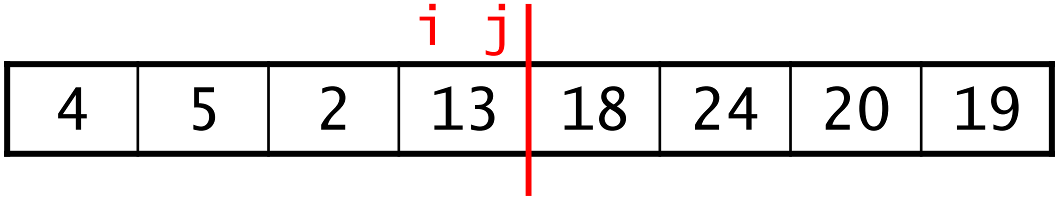
- Find:



4	5	2	13	18	24	20	19

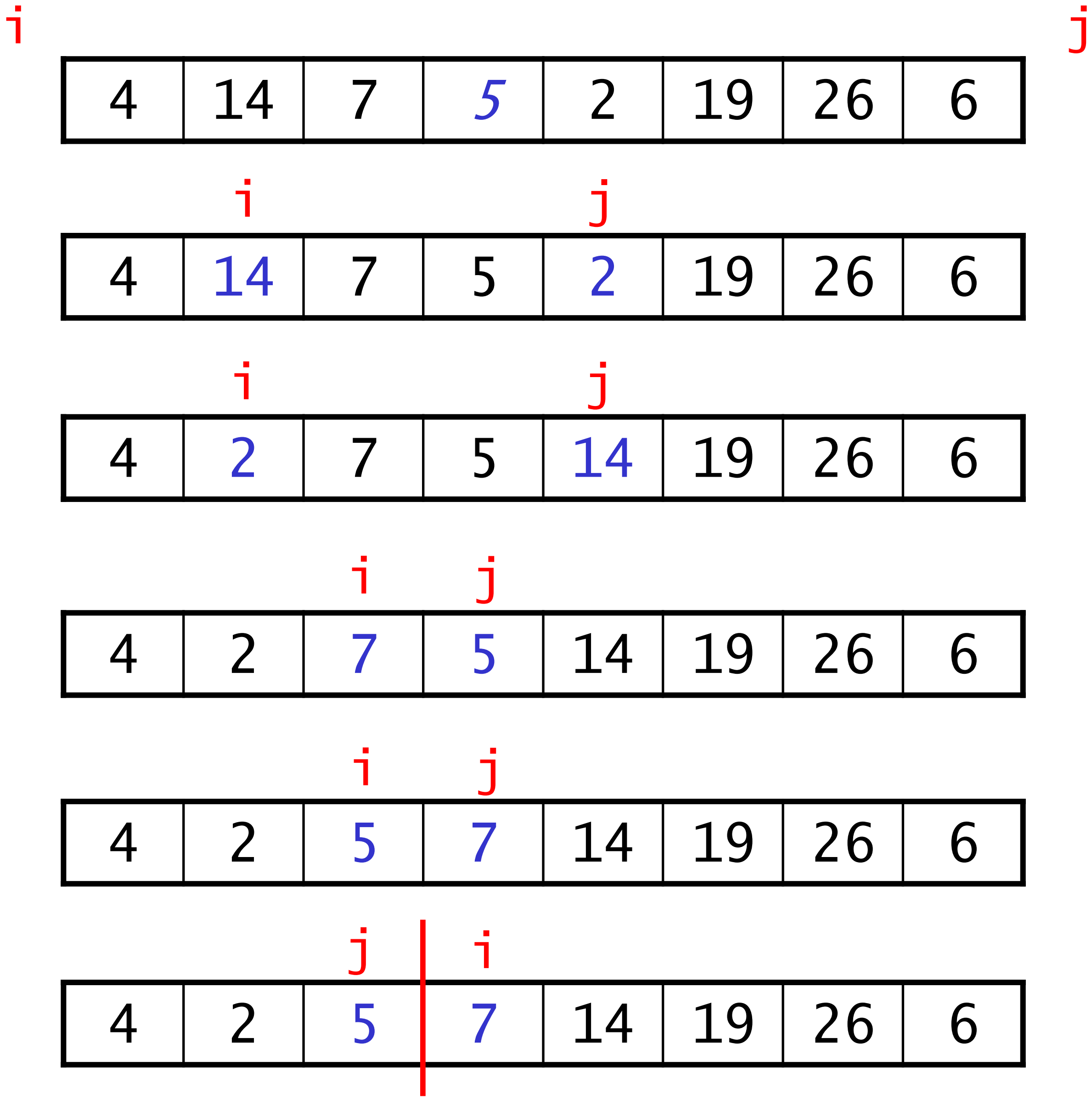
and now the indices are equal, so we return `j`.

- Subarrays:



4	5	2	13	18	24	20	19

Partitioning Example 3



- Start (pivot = 5):

i									j
	4	14	7	*5*	2	19	26	6	

- Find:

	i			j			
4	14	7	5	2	19	26	6

- Swap:

	i			j			
4	2	7	5	14	19	26	6

- Find:

		i	j				
4	2	7	5	14	19	26	6

- Swap:

		i	j				
4	2	5	7	14	19	26	6

- Find/*done!*:

		j	i				
4	2	5	7	14	19	26	6

Implementation of Quicksort

```
public static void quickSort(int[] arr) {
    qSort(arr, 0, arr.length-1);
}

private static void qSort(int[] arr, int first, int last) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);        // left subarray
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);   // right subarray
    }
}
```

Implementation of Quicksort

```
public static void quickSort(int[] arr) {
    qSort(arr, 0, arr.length-1);
}
```

```
private static void qSort(int[] arr, int first, int last) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);        // left subarray
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);     // right subarray
    }
}
```

Implementation of Quicksort

```
public static void quickSort(int[] arr) {
    qSort(arr, 0, arr.length-1);
}
```

```
private static void qSort(int[] arr, int first, int last) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);      // left subarray
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);   // right subarray
    }
}
```

Implementation of Quicksort

```
public static void quickSort(int[] arr) {
    qSort(arr, 0, arr.length-1);
}
```

```
private static void qSort(int[] arr, int first, int last) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);       // left subarray
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);    // right subarray
    }
}
```

What is the first thing each recursive call of qSort does?

Implementation of Quicksort

```
public static void quickSort(int[] arr) {
    qSort(arr, 0, arr.length-1);
}
```

```
private static void qSort(int[] arr, int first, int last) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);      // left subarray
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);   // right subarray
    }
}
```

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;  // index going left to right
    int j = last + 1;   // index going right to left

    do {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j) {
            swap(arr, i, j);
        }
    } while ( i < j );

    return j;    // arr[j] = end of left array
}
```

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;  // index going left to right
    int j = last + 1;   // index going right to left

    do {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j) {
            swap(arr, i, j);
        }
    } while ( i < j );

    return j;    // arr[j] = end of left array
}
```

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;  // index going left to right
    int j = last + 1;   // index going right to left

    do {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j) {
            swap(arr, i, j);
        }
    } while ( i < j );

    return j;    // arr[j] = end of left array
}
```

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;  // index going left to right
    int j = last + 1;   // index going right to left

    do {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j) {
            swap(arr, i, j);
        }
    } while ( i < j );

    return j;    // arr[j] = end of left array
}
```

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;     // index going left to right
    int j = last + 1;      // index going right to left

    do {
        do { // move i until the first element >= pivot
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j) {
            swap(arr, i, j);
        }
    } while ( i < j );

    return j;       // arr[j] = end of left array
}
```

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;  // index going left to right
    int j = last + 1;   // index going right to left

    do {
        do {
            i++;
        } while (arr[i] < pivot);
        do { // move j until the first element <= pivot
            j--;
        } while (arr[j] > pivot);
        if (i < j) {
            swap(arr, i, j);
        }
    } while ( i < j );

    return j;     // arr[j] = end of left array
}
```

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;  // index going left to right
    int j = last + 1;   // index going right to left

    do {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j) { // If we have not crossed partitions
            swap(arr, i, j);
        }
    } while ( i < j );

    return j;    // arr[j] = end of left array
}
```

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;  // index going left to right
    int j = last + 1;   // index going right to left

    do {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j) {
            swap(arr, i, j);
        }
    } while ( i < j ); // while we have not crossed partitions

    return j;    // arr[j] = end of left array
}
```

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;  // index going left to right
    int j = last + 1;   // index going right to left

    do {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j) {
            swap(arr, i, j);
        }
    } while ( i < j );

    return j;    // arr[j] = end of left array
}
```

Implementation of Quicksort

```
public static void quickSort(int[] arr) {
    qSort(arr, 0, arr.length-1);
}

private static void qSort(int[] arr, int first, int last) {
    int split = partition(arr, first, last);

    if (first < split) {
        qSort(arr, first, split);      // left subarray
    }
    if (last > split + 1) {
        qSort(arr, split + 1, last);   // right subarray
    }
}
```

Implementation of Quicksort

```
public static void quickSort(int[] arr) {
    qSort(arr, 0, arr.length-1);
}
```

```
private static void qSort(int[] arr, int first, int last) {
    int split = partition(arr, first, last);

    if (first <      lit) {
        qSort(arr,                                            harray
    }
    if (last > s
        qSort(a                                                array
    }
}
```

Note that the **`partition`** method determines the split of the two partitions.

Quicksort

Let's say that we're using quicksort to sort the following array:
{24, 8, 5, 7, 17, 2, 10}

What will the array look like after the first partitioning of the array, with a pivot of ?

A. {10, 2, 5, 7, 17, 8, 24}
B. {2, 5, 8, 7, 17, 24, 10}
C. {2, 7, 5, 8, 17, 24, 10}
D. {2, 5, 8, 24, 17, 10, 7}

Quicksort

Let's say that we're using quicksort to sort the following array:

{24, 8, 5, 7, 17, 2, 10}

What will the array look like after the first partitioning of the array, with a pivot of **7**.

A. {10, 2, 5, 7, 17, 8, 24}
B. {2, 5, 8, 7, 17, 24, 10}
C. {2, 7, 5, 8, 17, 24, 10}
D. {2, 5, 8, 24, 17, 10, 7}

Quicksort

Let's say that we're using quicksort to sort the following array:
{24, 8, 5, 7, 17, 2, 10}

What will the array look like after the first partitioning of the array, with a pivot of 7.

A. {10, 2, 5, 7, 17, 8, 24}
B. {2, 5, 8, 7, 17, 24, 10}
C. {2, 7, 5, 8, 17, 24, 10}
D. {2, 5, 8, 24, 17, 10, 7}

Quicksort

The next call to qsort will be with which elements?

A. {10, 2, 5, 7, 17, 8, 24}

B. {2, 5, 8, 7, 17, 24, 10}

C. {2, 7, 5, 8, 17, 24, 10}

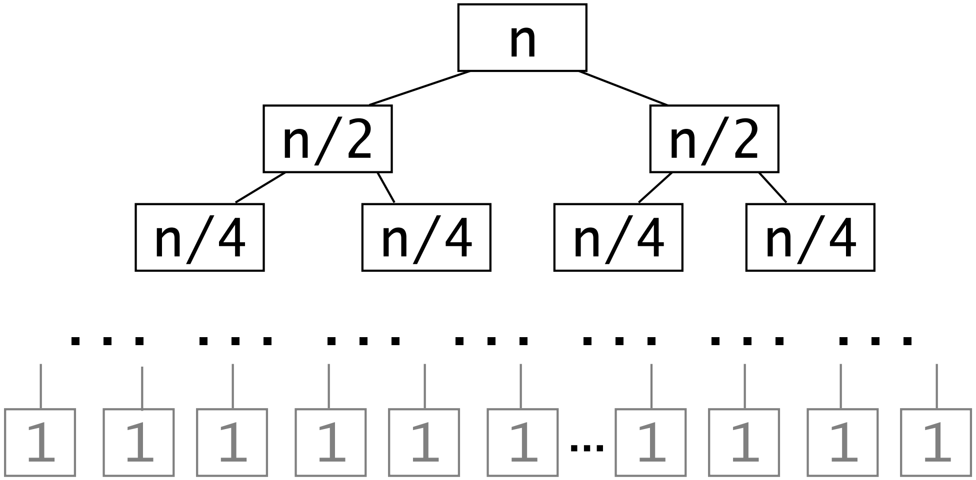
D. {2, 5, 8, 24, 17, 10, 7}

Time Analysis of Quicksort

- Partitioning an array requires **n** comparisons, because each element is compared with the pivot.
- *best case:* partitioning always divides the array in half

Time Analysis of Quicksort

- Partitioning an array requires n comparisons, because each element is compared with the pivot.
- *best case:* partitioning always divides the array in half
    - repeated recursive calls give:



comparisons

n

2*(n/2) = n

4*(n/4) = n

...

0

- at each "row" except the bottom, we perform n comparisons
- there are $\sim\log_2 n$ rows that include comparisons

A Quick Review of Logarithms

- $\log_b n$ = the exponent to which b must be raised to get n
  - $\log_b n = p$ if $b^p = n$
  - examples: $\log_2 8 = 3$ because $2^3 = 8$
    $\log_{10} 10000 = 4$ because $10^4 = 10000$

A Quick Review of Logarithms

- $\log_b n$ = the exponent to which b must be raised to get n
  - $\log_b n = p$ if $b^p = n$
  - examples: $\log_2 8 = 3$ because $2^3 = 8$
    $\log_{10} 10000 = 4$ because $10^4 = 10000$
- Another way of looking at logs:
  - let's say that you repeatedly divide n by b (using integer division)
  - $\log_b n$ is an upper bound on the number of divisions needed to reach 1
  - example: $\log_2 18$ is approx. 4.17

    18/2 = 9   9/2 = 4   4/2 = 2   2/2 = 1

A Quick Review of Logarithms:

summary

- $\log_b n$ = the exponent to which b must be raised to get n
  - $\log_b n = p$ if $b^p = n$
  - examples: $\log_2 8 = 3$ because $2^3 = 8$

8/2 | 4/2 | 2/2

We divide 8 by 2 three times to get to 1
or …..$\log_2 8$

How many times do we have to divide some n to get to 1?

$\log_2 n$

A Quick Review of Logs (cont.)

- If the number of operations performed by an algorithm is proportional to $\log_b n$ for any base b, we say it is a $O(\log n)$ algorithm – dropping the base.

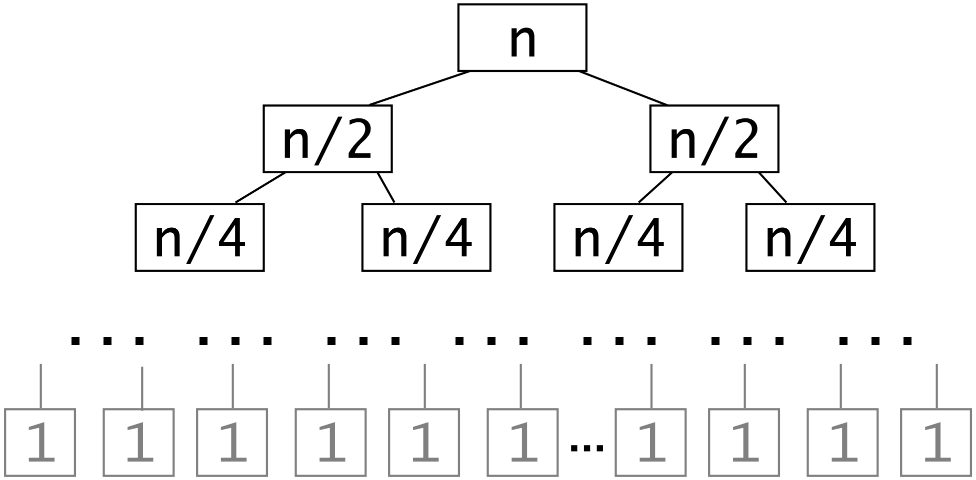
- $\log_b n$ grows much more slowly than `n`

n	**$\log_2 n$**
2	1
1024 (1K)	10
1024*1024 (1M)	20

- Thus, for large values of n:
  - a $O(\log n)$ algorithm is more efficient than a $O(n)$ algorithm
  - a $O(n \log n)$ algorithm is more efficient than a $O(n^2)$ algorithm

Time Analysis of Quicksort

- Partitioning an array requires n comparisons, because each element is compared with the pivot.
- *best case:* partitioning always divides the array in half
    - repeated recursive calls give:



comparisons

n

$2*(n/2) = n$

$4*(n/4) = n$

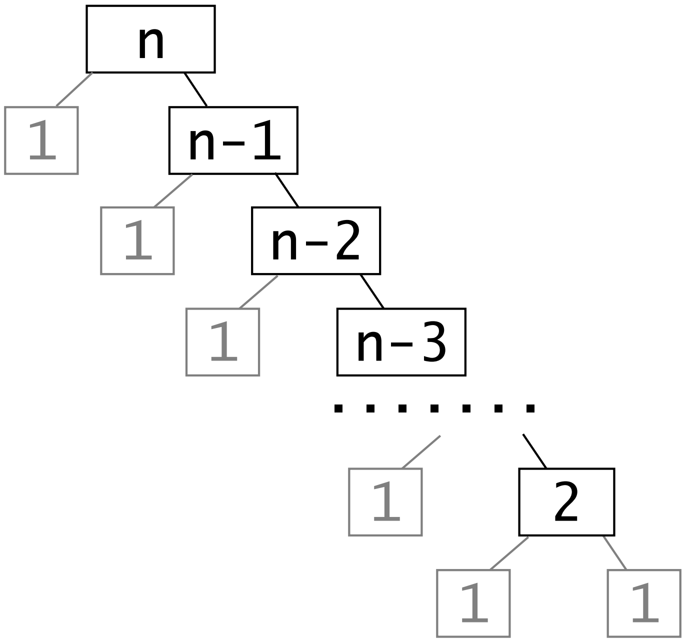
...

0

- at each "row" except the bottom, we perform n comparisons
- there are $\sim\log_2 n$ rows that include comparisons
- $C(n) = \sim n \log_2 n = O(n \log n)$
- Similarly, $M(n)$ and running time are both $O(n \log n)$

Time Analysis of Quicksort (cont.)

- *worst case:* pivot is always the smallest or largest element
  - one subarray has `1` element, the other has `n - 1`
  - repeated recursive calls give:



comparisons

n

n-1

n-2

n-3

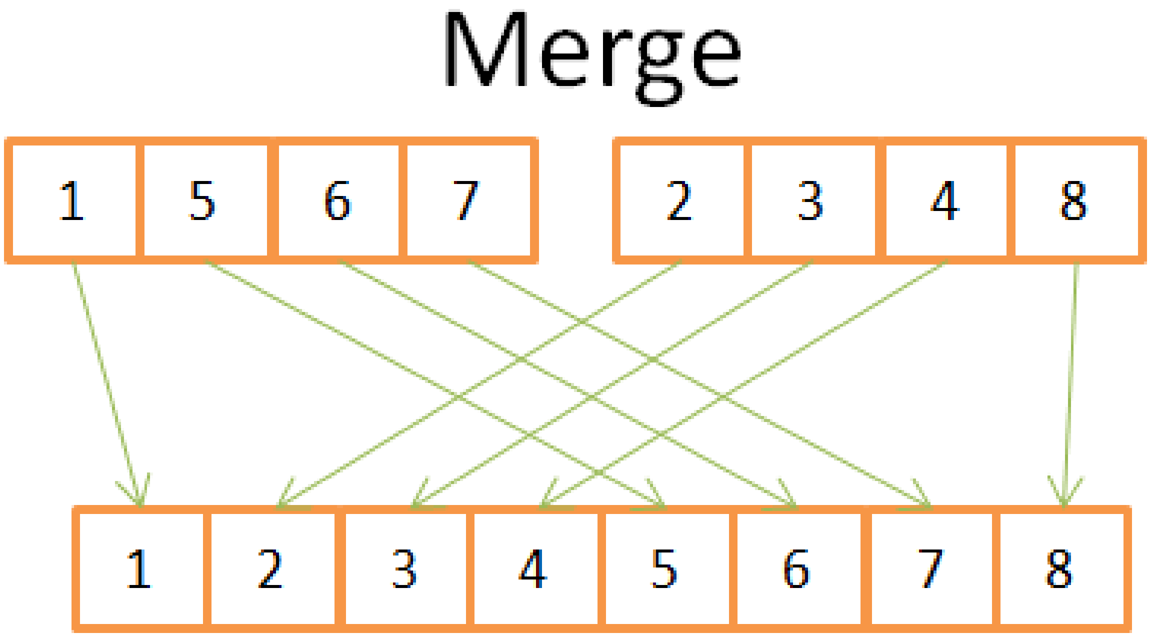
...

2

- $C(n) = \sum_{i=2}^{n} i = O(n^2)$. $M(n)$ and run time are also $O(n^2)$.

- *average case* is harder to analyze
  - $C(n) > n \log_2 n$, but it's still $O(n \log n)$

Mergesort

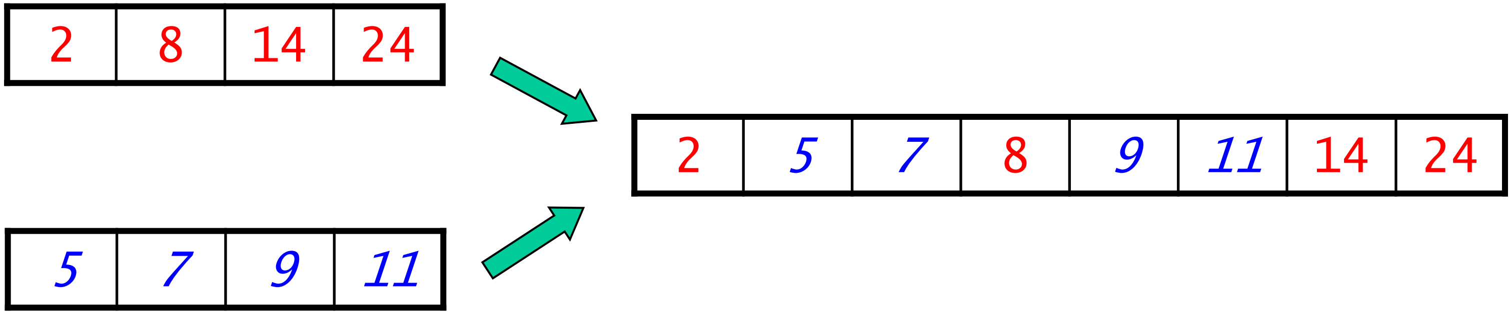


Mergesort

- All of the comparison-based sorting algorithms that we've seen thus far have sorted the array in place.
    - used only a small amount of additional memory

- Mergesort is a sorting algorithm that requires an additional temporary array of the same size as the original one.
    - it needs *O*(n) additional space, where n is the array size

- It is based on the process of *merging* two sorted arrays into a single sorted array.
    - Example:



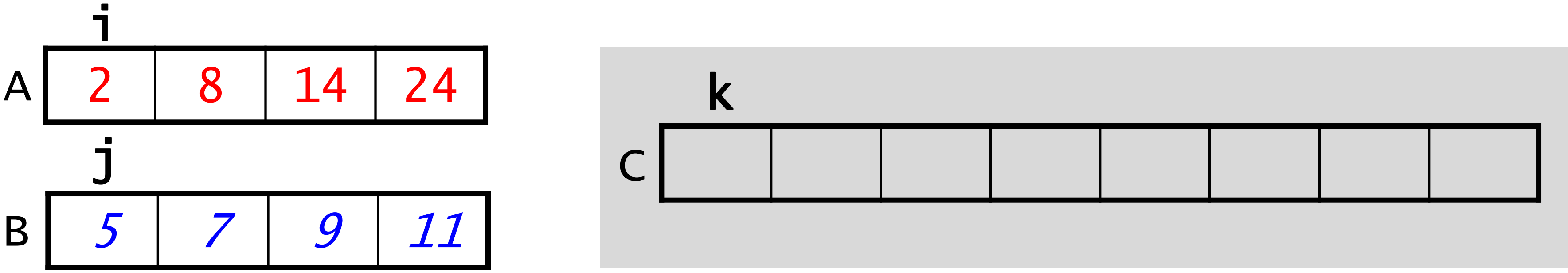
2	8	14	24

5	*7*	*9*	*11*

2	*5*	*7*	8	*9*	*11*	14	24

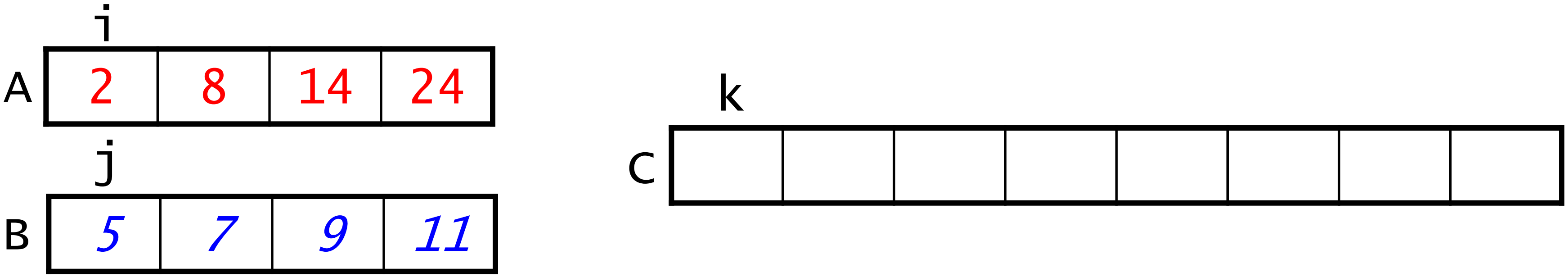
Merging Sorted Arrays

- To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:

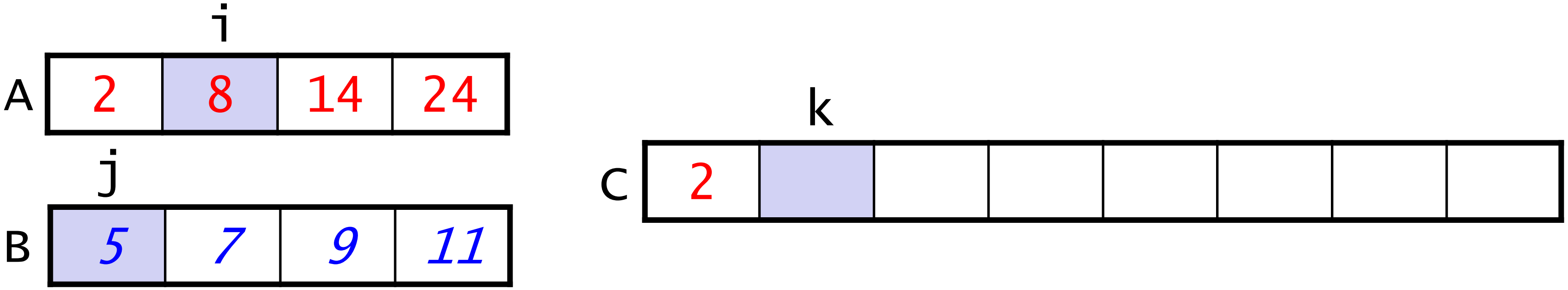


Merging Sorted Arrays

- To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:

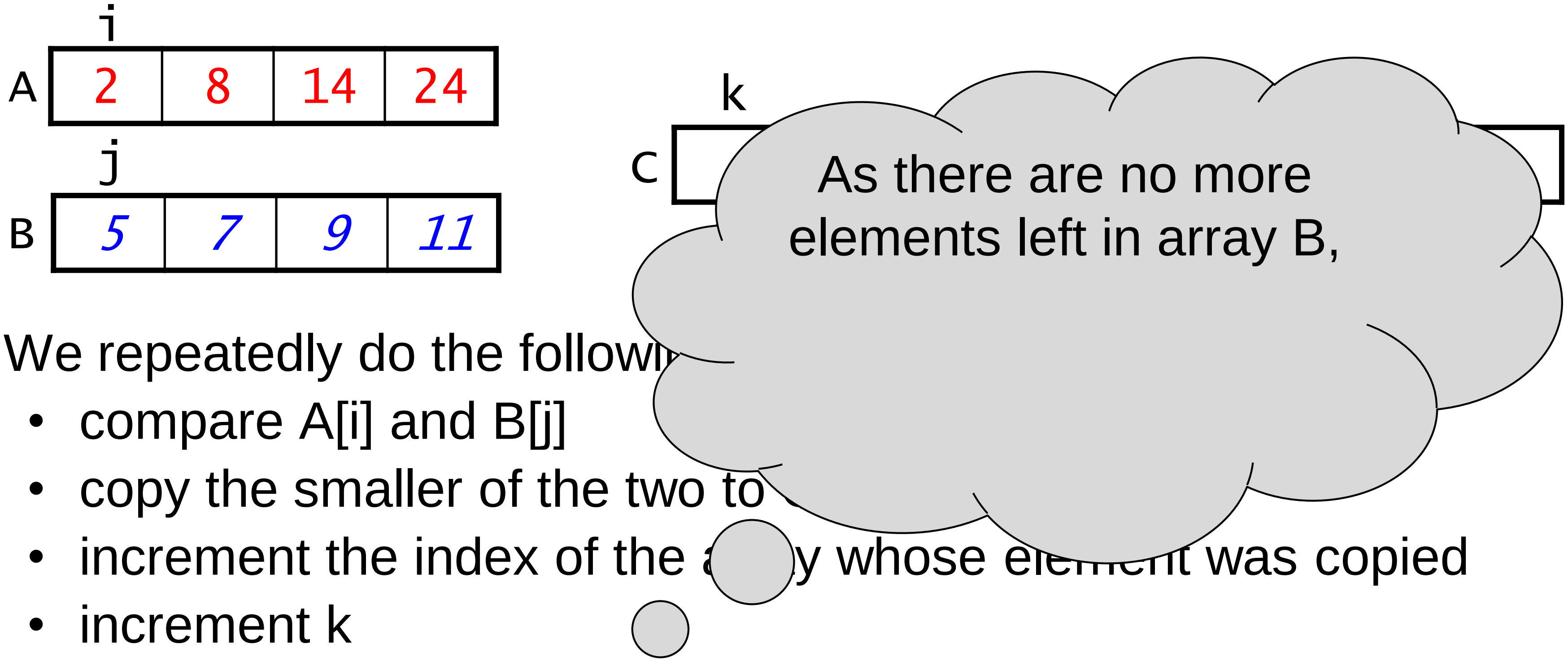


- We repeatedly do the following:
  - compare A[i] and B[j]
  - copy the smaller of the two to C[k]
  - increment the index of the array whose element was copied
  - increment k



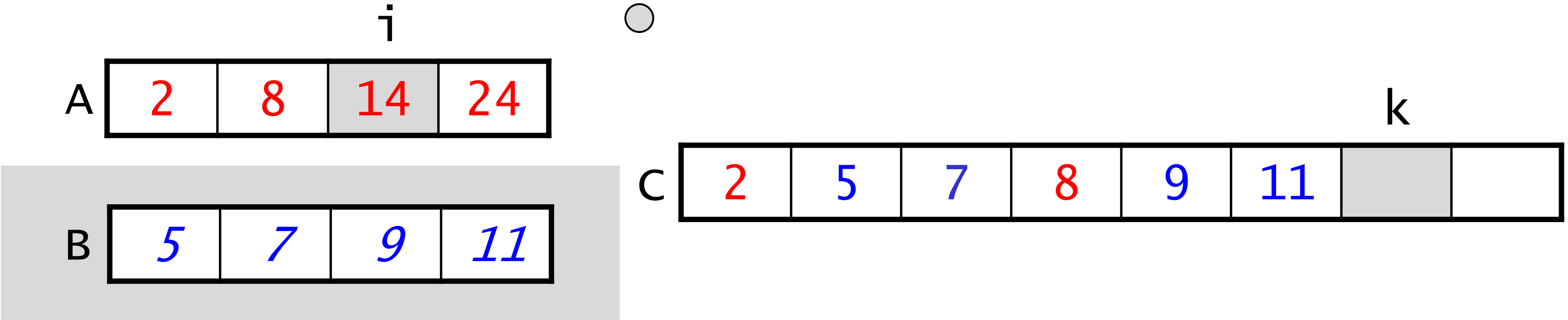
Merging Sorted Arrays

- To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:



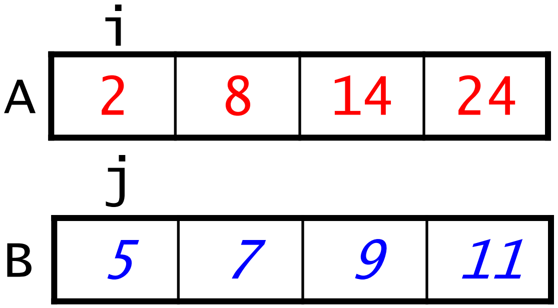
As there are no more elements left in array B,

- We repeatedly do the followi
  - compare A[i] and B[j]
  - copy the smaller of the two to
  - increment the index of the a y whose element was copied
  - increment k



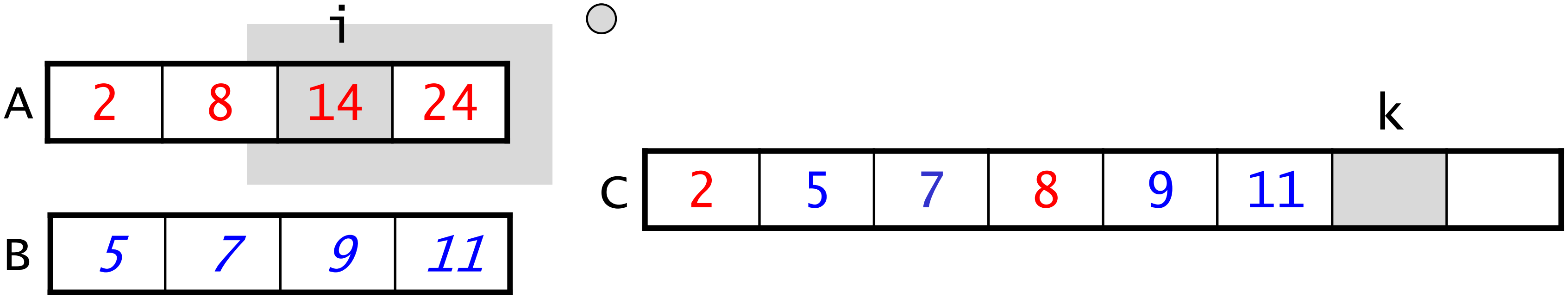
Merging Sorted Arrays

- To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:



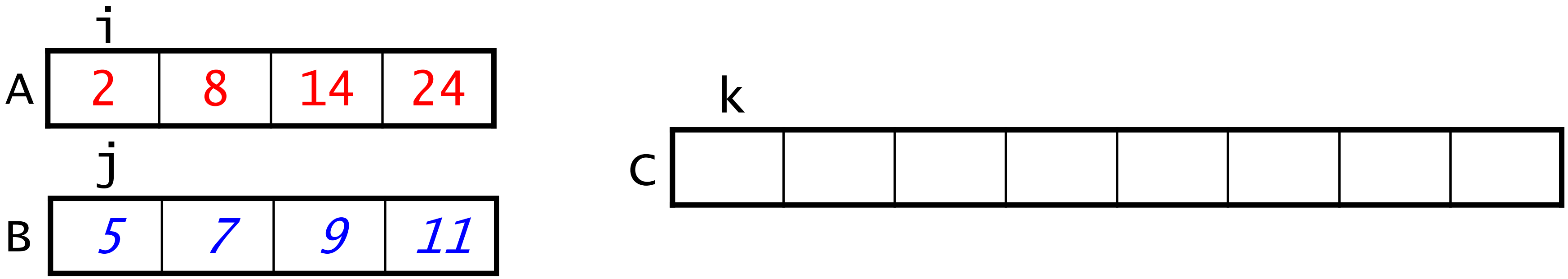
- We repeatedly do the followi
  - compare A[i] and B[j]
  - copy the smaller of the two to
  - increment the index of the a y whose ele nt was copied
  - increment k

As there are no more elements left in array B, we simply copy over the remaining elements from array A!



Merging Sorted Arrays

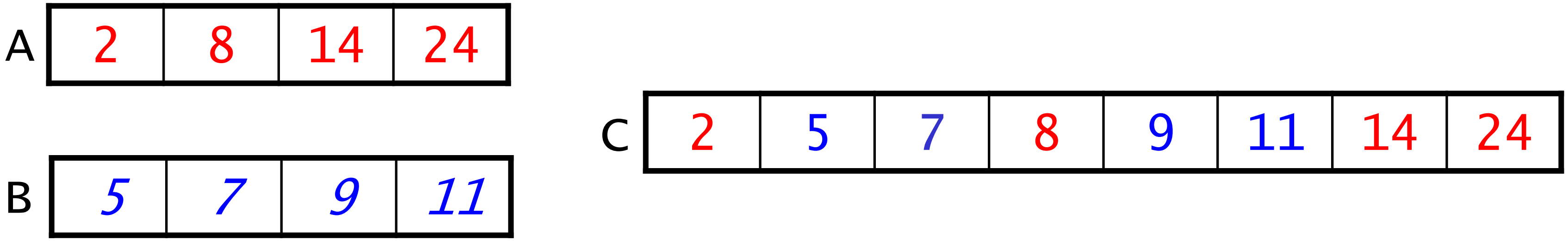
- To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:



	i			
A	2	8	14	24
	j			
B	5	7	9	11

	k							
C								

- We repeatedly do the following:
  - compare A[i] and B[j]
  - copy the smaller of the two to C[k]
  - increment the index of the array whose element was copied
  - increment k

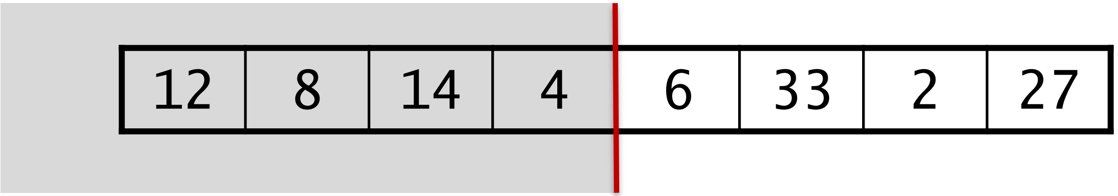


A	2	8	14	24
B	5	7	9	11

C	2	5	7	8	9	11	14	24

Divide and Conquer

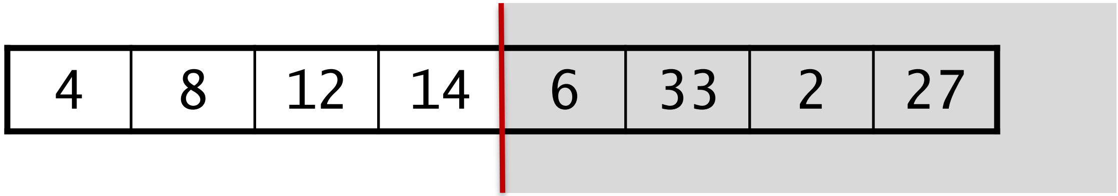
- Like quicksort, mergesort is a divide-and-conquer algorithm.
  - *divide:* split the array in half, forming two subarrays
  - ***conquer**:* apply mergesort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* merge the sorted subarrays



12	8	14	4	6	33	2	27

Divide and Conquer

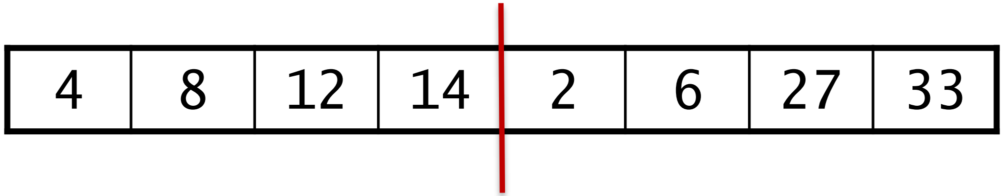
- Like quicksort, mergesort is a divide-and-conquer algorithm.
  - *divide:* split the array in half, forming two subarrays
  - ***conquer**:* apply mergesort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* merge the sorted subarrays



4	8	12	14	6	33	2	27

Divide and Conquer

- Like quicksort, mergesort is a divide-and-conquer algorithm.
  - *divide:* split the array in half, forming two subarrays
  - ***conquer**:* apply mergesort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* merge the sorted subarrays



4	8	12	14	2	6	27	33

Divide and Conquer

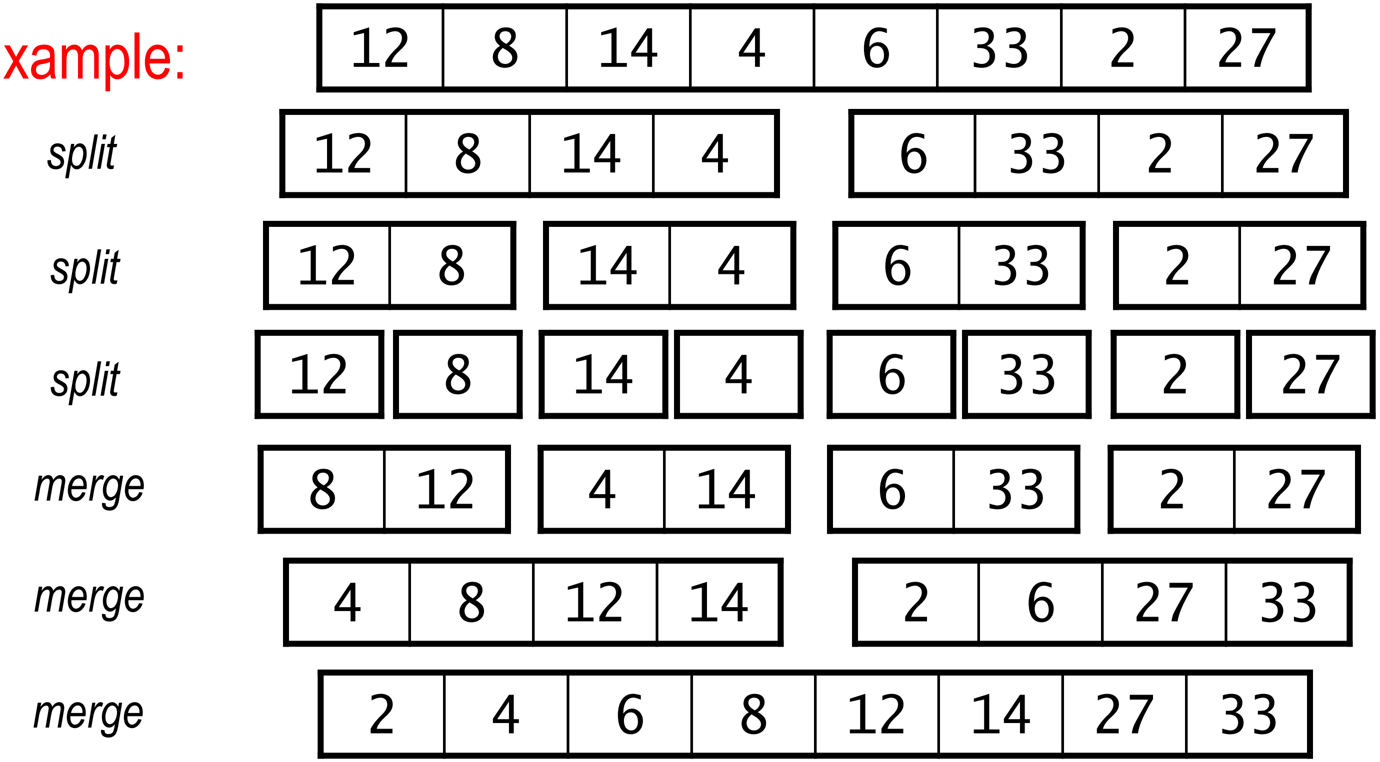
- Like quicksort, mergesort is a divide-and-conquer algorithm.
  - *divide:* split the array in half, forming two subarrays
  - *conquer:* apply mergesort recursively to the subarrays, stopping when a subarray has a single element
  - ***combine**:* merge the sorted subarrays

2	4	6	8	12	14	27	33

Divide and Conquer

- Like quicksort, mergesort is a divide-and-conquer algorithm.
  - *divide:* split the array in half, forming two subarrays
  - *conquer:* apply mergesort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* merge the sorted subarrays

Example:



12 8 14 4 6 33 2 27

split 12 8 14 4 6 33 2 27

split 12 8 14 4 6 33 2 27

split 12 8 14 4 6 33 2 27

merge 8 12 4 14 6 33 2 27

merge 4 8 12 14 2 6 27 33

merge 2 4 6 8 12 14 27 33

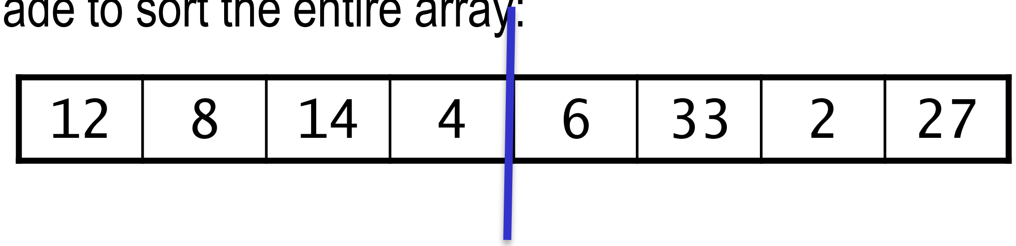
Mergesort Algorithm

- Let's design the recursive method together:

```
private static void mSort(int[] arr, int start, int end) {
    if (start < end) {
        // find the middle of the array
        // recursively sort the left half (start – middle)
        // recursively sort the right half (middle+1 – end)

        // merge the two sorted halves
    }
}
```

Tracing the Calls to Mergesort

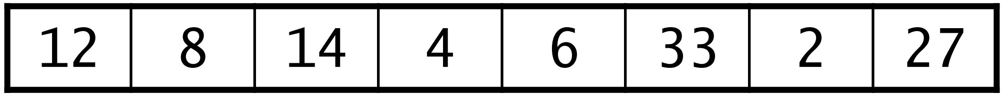
the initial call is made to sort the entire array:



12	8	14	4	6	33	2	27

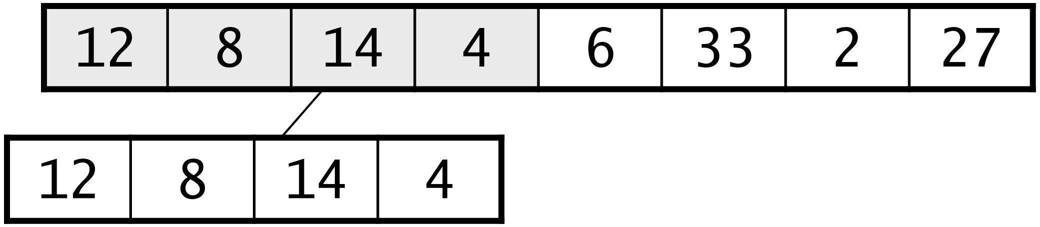
Tracing the Calls to Mergesort

the initial call is made to sort the entire array:



12	8	14	4	6	33	2	27

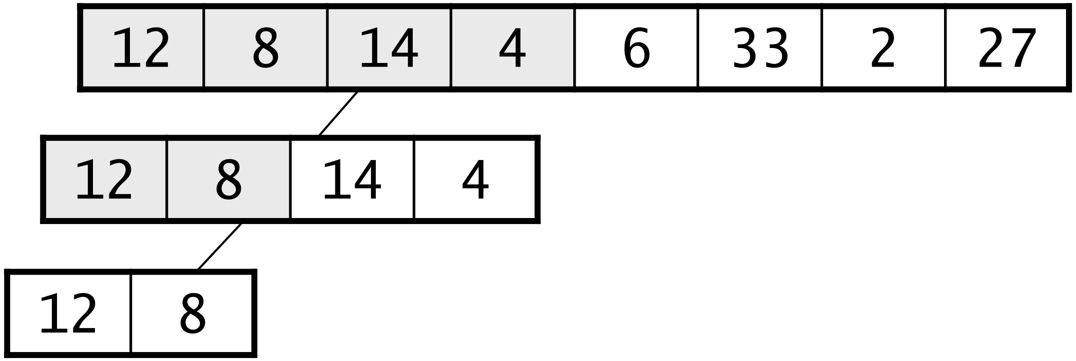
split into two 4-element subarrays, and make a recursive call to sort the left subarray:



12	8	14	4	6	33	2	27

12	8	14	4

split into two 2-element subarrays, and make a recursive call to sort the left subarray:



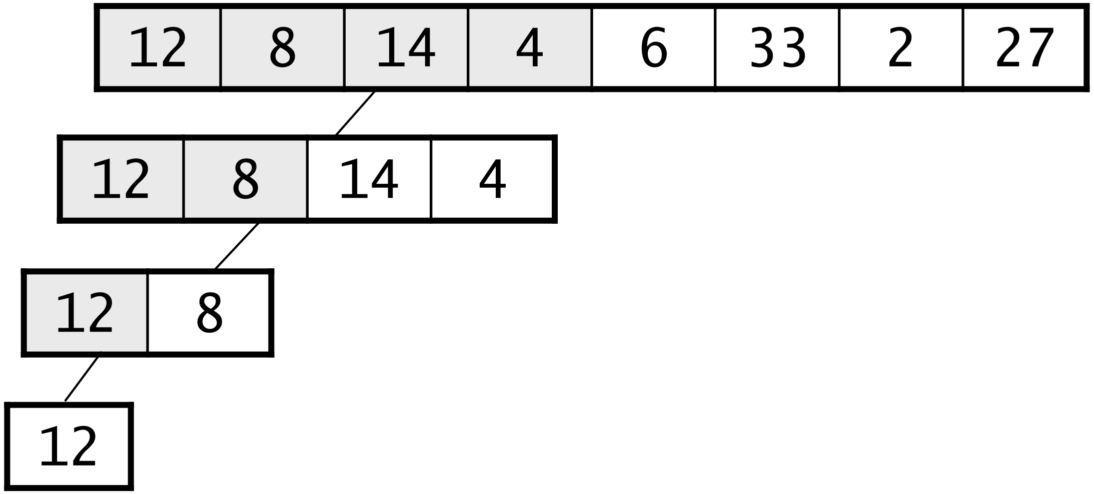
12	8	14	4	6	33	2	27

12	8	14	4

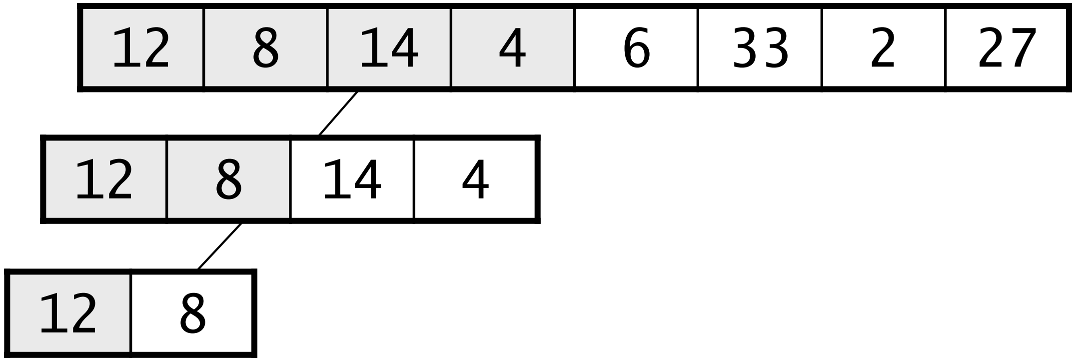
12	8

Tracing the Calls to Mergesort

split into two 1-element subarrays, and make a recursive call to sort the left subarray:

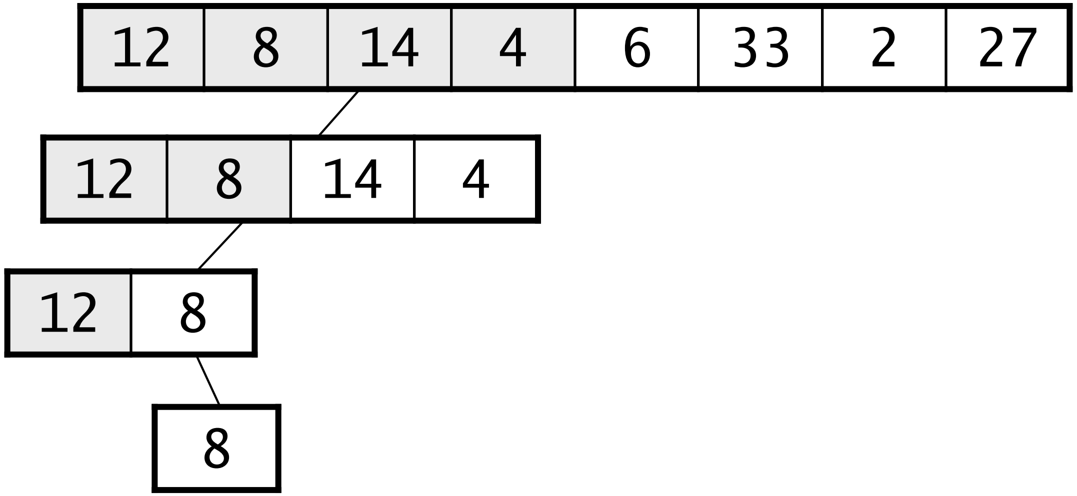


base case, so return to the call for the subarray {12, 8}:

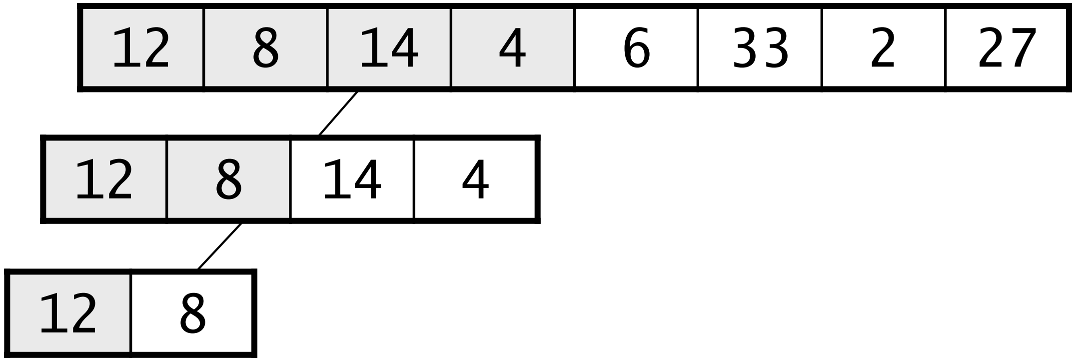


Tracing the Calls to Mergesort

make a recursive call to sort its right subarray:

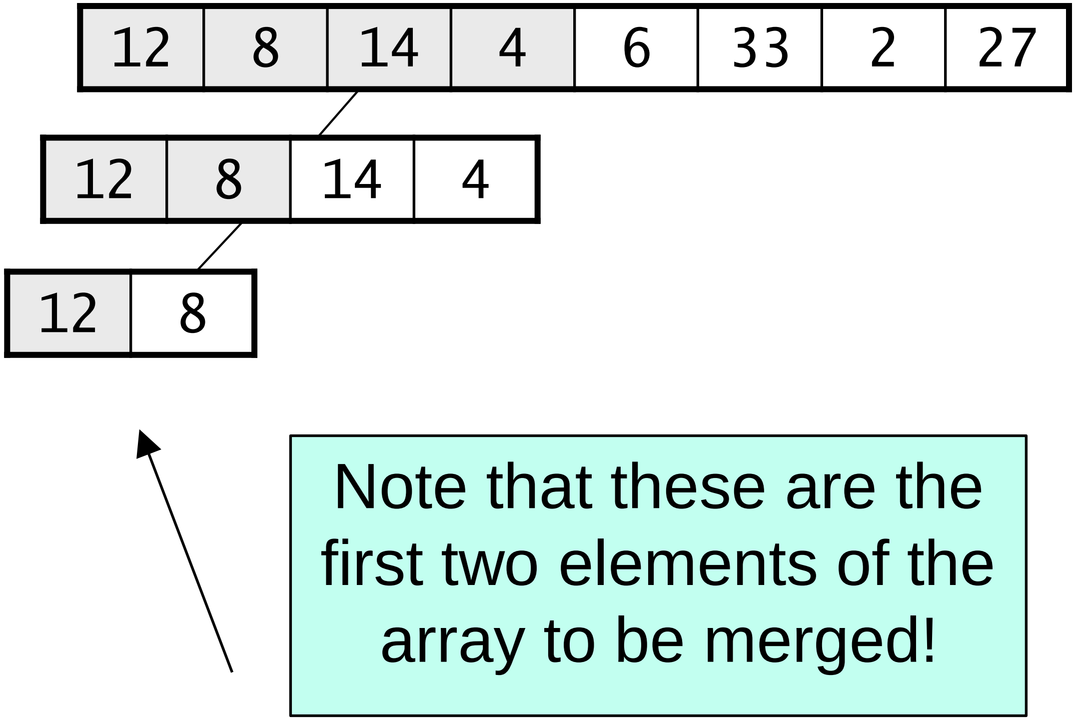


base case, so return to the call for the subarray {12, 8}:



Tracing the Calls to Mergesort

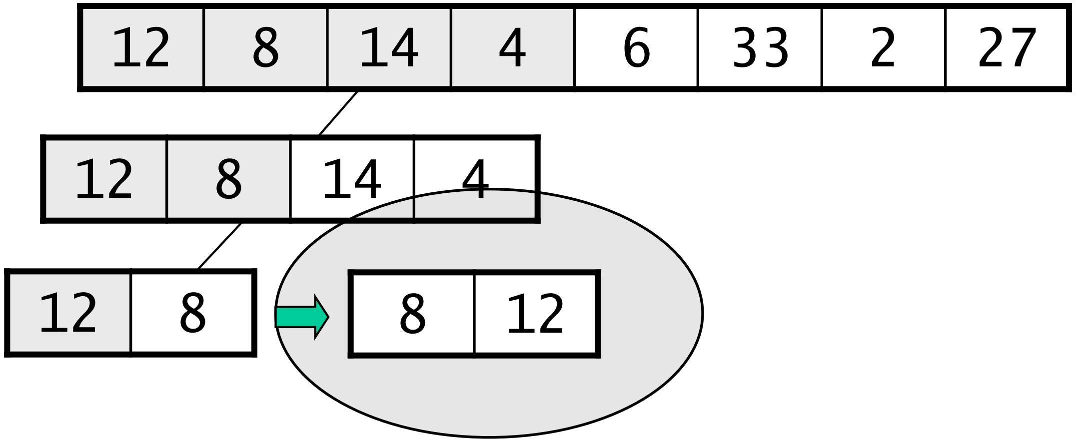
merge the sorted halves of {12, 8}:



Note that these are the first two elements of the array to be merged!

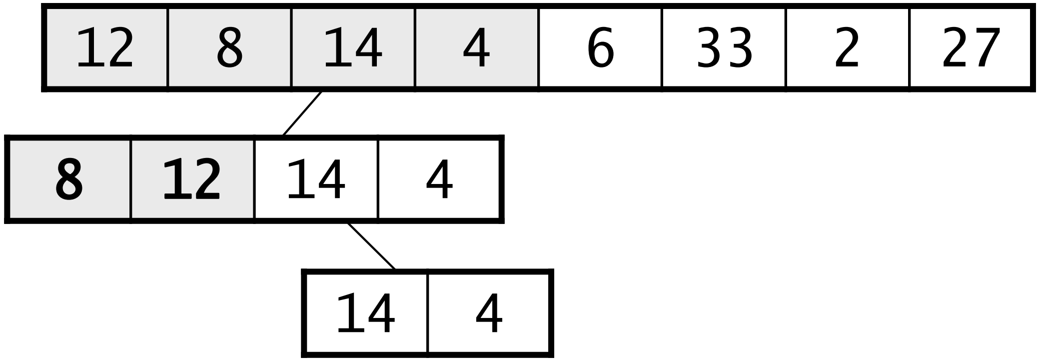
Tracing the Calls to Mergesort

merge the sorted halves of {12, 8}:

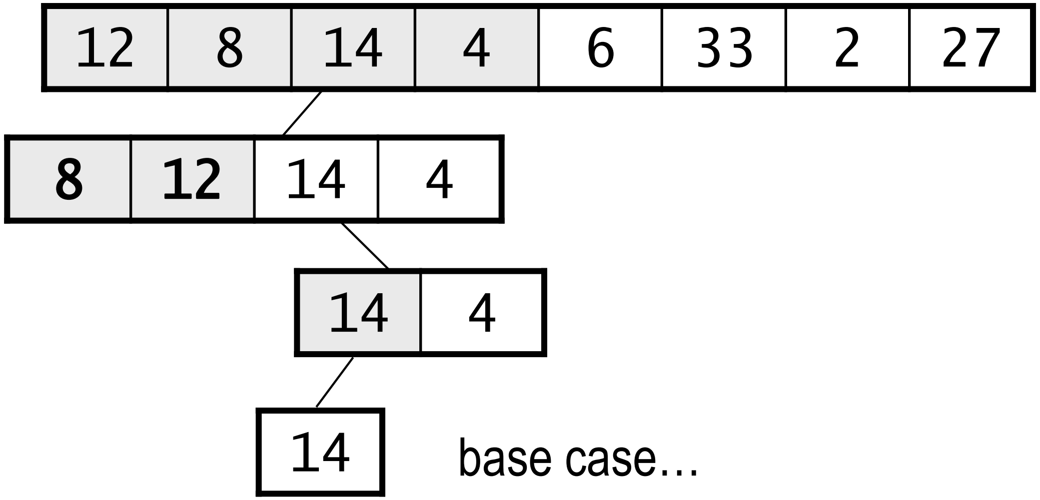


Tracing the Calls to Mergesort

make a recursive call to sort the right subarray of the 4-element subarray



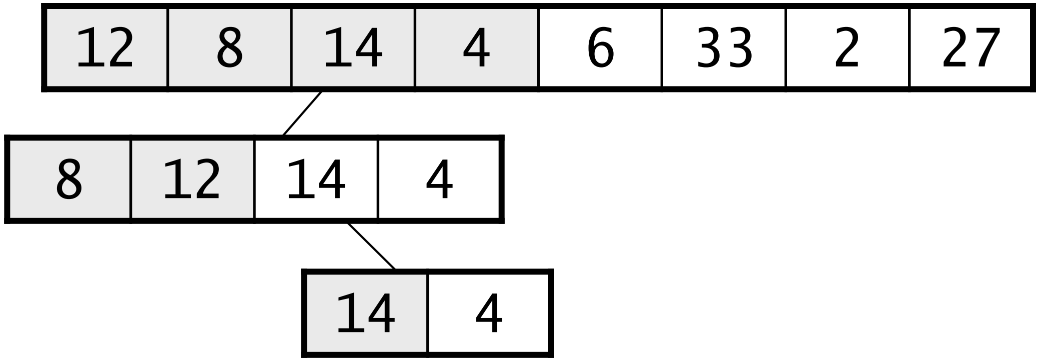
split it into two 1-element subarrays, and make a recursive call to sort the left subarray:



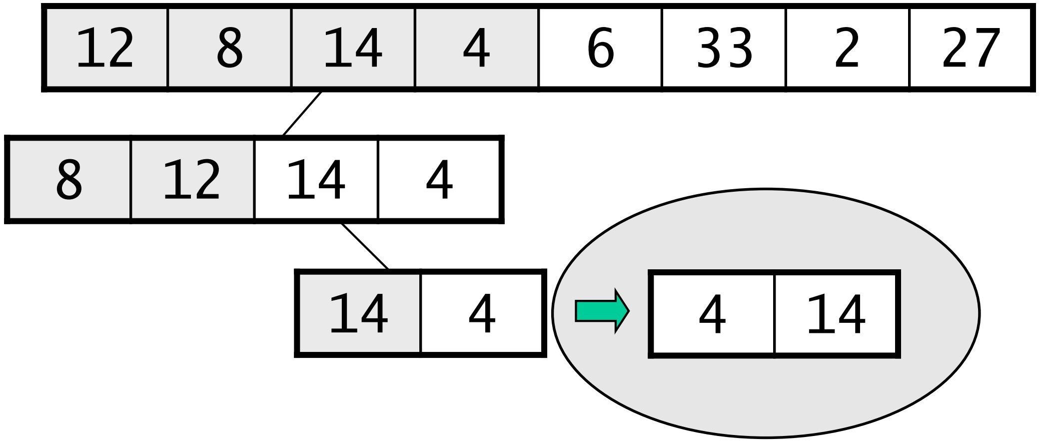
base case…

Tracing the Calls to Mergesort

return to the call for the subarray {14, 4}:

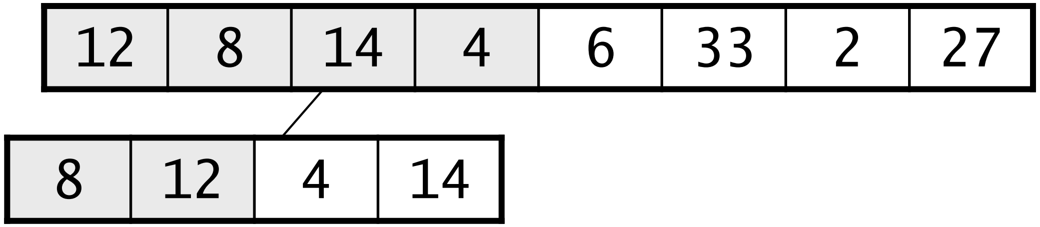


merge the sorted halves of {14, 4}:

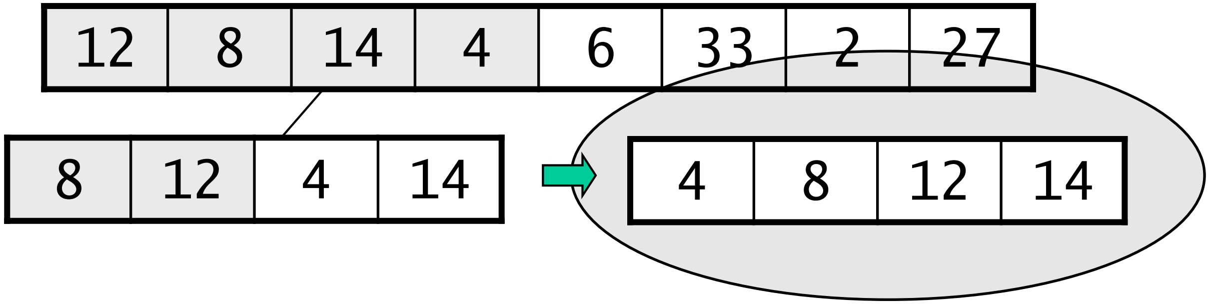


Tracing the Calls to Mergesort

end of the method, so return to the call for the 4-element subarray, which now has two sorted 2-element subarrays:



merge the 2-element subarrays:



Tracing the Calls to Mergesort

end of the method, so return to the call for the original array, which now has a sorted left subarray:

4	**8**	**12**	**14**	6	33	2	27

Tracing the Calls to Mergesort

end of the method, so return to the call for the original array, which now has a sorted left subarray:

4	**8**	**12**	**14**	6	33	2	27

perform a similar set of recursive calls to sort the right subarray. here's the result:

4	8	12	14	**2**	**6**	**27**	**33**

Tracing the Calls to Mergesort

end of the method, so return to the call for the original array, which now has a sorted left subarray:

4	**8**	**12**	**14**	6	33	2	27

perform a similar set of recursive calls to sort the right subarray. here's the result:

4	8	12	14	**2**	**6**	**27**	**33**

finally, merge the sorted 4-element subarrays to get a fully sorted 8-element array:

4	8	12	14	2	6	27	33

2	4	6	8	12	14	27	33

Implementing the Mergesort Algorithm

- The recursive merge sort method:

```
private static void mSort(int[] arr, int start, int end) {

    if (start < end) {
        int middle = (start + end)/2;
        mSort(arr, start, middle);
        mSort(arr, middle + 1, end);

        merge(arr, start, middle, middle + 1, end);
    }
}
```

When we return from ***this*** call to mSort the elements in the array from start to middle are sorted!

Implementing the Mergesort Algorithm

- The recursive merge sort method:

```
private static void mSort(int[] arr, int start, int end) {

    if (start < end) {
       int middle = (start + end)/2;
       mSort(arr, start, middle);
       mSort(arr, middle + 1, end);

       merge(arr, start, middle, middle + 1, end);
    }
}
```

When we return from ***this*** call to mSort the elements in the array from middle+1 to end are sorted!

Implementing the Mergesort Algorithm

- The recursive merge sort method:

```
private static void mSort(int[] arr, int start, int end) {

    if (start < end) {
        int middle = (start + end)/2;
        mSort(arr, start, middle);
        mSort(arr, middle + 1, end);

        merge(arr, start, middle, middle + 1, end);
    }
}
```

Now we can merge these two sorted halves!

Implementing the Mergesort Algorithm

- The recursive merge sort method:

```
private static void mSort(int[] arr, int start, int end) {

    if (start < end) {
       int middle = (start + end)/2;
       mSort(arr, start, middle);
       mSort(arr, middle + 1, end);

       merge(arr, start, middle, middle + 1, end);
    }
}
```

Note how the two subarrays to be merged are defined.

Implementing the Mergesort Algorithm

- The recursive merge sort method:

```
private static void mSort(int[] arr, int start, int end) {

    if (start < end) {
       int middle = (start + end)/2;
       mSort(arr, start, middle);
       mSort(arr, middle + 1, end);

       merge(arr, start, middle, middle + 1, end);
    }
}
```

Note how the two subarrays to be merged are defined.

Implementing the Mergesort Algorithm

- The recursive merge sort method:

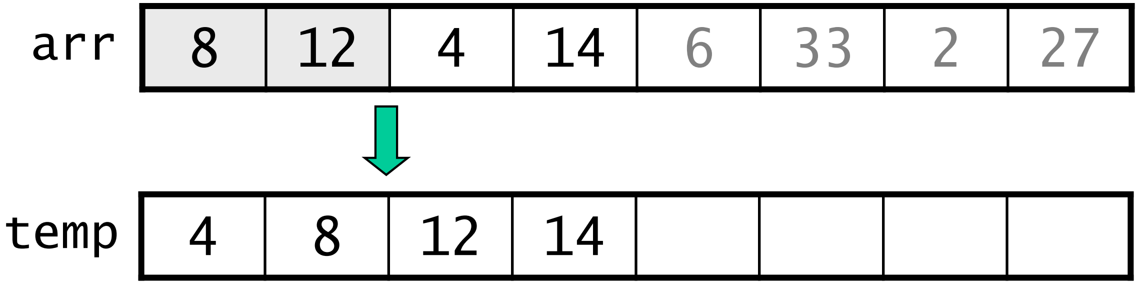
```
private static void mSort(int[] arr, int start, int end) {

    if (start < end) {
       int middle = (start + end)/2;
       mSort(arr, start, middle);
       mSort(arr, middle + 1, end);

       merge(arr, start, middle, middle + 1, end);
    }
}
```

But what will the function merge use as a temporary array to perform the merge?

Implementing Mergesort

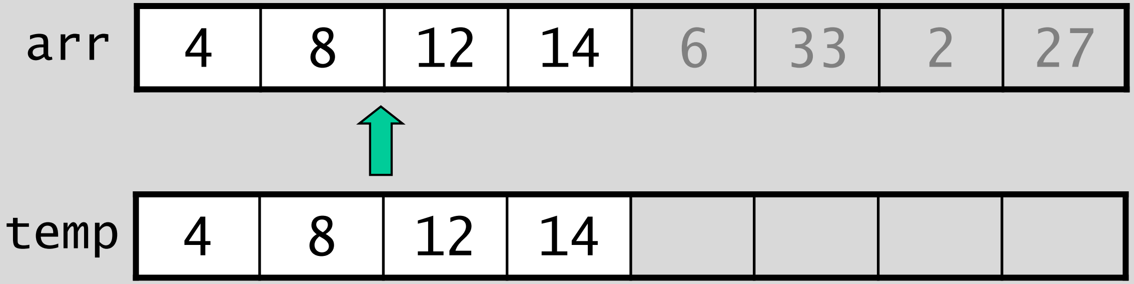
- One approach is to create new arrays for each new set of subarrays, and to merge them back into the array that was split.

- Another is to create a temporary array of the same size and
  - pass it to each call of the recursive mergesort method
  - use it when merging subarrays of the original array:



arr	8	12	4	14	6	33	2	27
temp	4	8	12	14				

  - after each merge, copy the result back into the original array:



arr	4	8	12	14	6	33	2	27
temp	4	8	12	14				

A Method for Merging Subarrays

```
private static void merge(int[] arr, int[] temp,
  int leftStart, int leftEnd, int rightStart, int rightEnd) {
    int i = leftStart;    // index into left subarray
    int j = rightStart;   // index into right subarray
    int k = leftStart;    // index into temp

    while (i <= leftEnd && j <= rightEnd) {
        if (arr[i] < arr[j]) {
            temp[k] = arr[i];
            i++; k++;
        } else {
            temp[k] = arr[j];
            j++; k++;
        }
    }

    while (i <= leftEnd) {
        temp[k] = arr[i];
        i++; k++;
    }
    while (j <= rightEnd) {
        temp[k] = arr[j];
        j++; k++;
    }

    for (i = leftStart; i <= rightEnd; i++) {
        arr[i] = temp[i];
    }
}
```

A Method for Merging Subarrays

```
private static void merge(int[] arr, int[] temp,
  int leftStart, int leftEnd, int rightStart, int rightEnd) {
     int i = leftStart;     // index into left subarray
     int j = rightStart;    // index into right subarray
     int k = leftStart;     // index into temp

     while (i <= leftEnd && j <= rightEnd) {
         if (arr[i] < arr[j]) {
             temp[k] = arr[i];
             i++; k++;
         } else {
             temp[k] = arr[j];
             j++; k++;
         }
     }

     while (i <= leftEnd) {
         temp[k] = arr[i];
         i++; k++;
     }
     while (j <= rightEnd) {
         temp[k] = arr[j];
         j++; k++;
     }

     for (i = leftStart; i <= rightEnd; i++) {
         arr[i] = temp[i];
     }
}
```

Merge the elements into the temporary array.

A Method for Merging Subarrays

```
private static void merge(int[] arr, int[] temp,
  int leftStart, int leftEnd, int rightStart, int rightEnd) {
     int i = leftStart;    // index into left subarray
     int j = rightStart;   // index into right subarray
     int k = leftStart;    // index into temp

     while (i <= leftEnd && j <= rightEnd) {
         if (arr[i] < arr[j]) {
             temp[k] = arr[i];
             i++; k++;
         } else {
             temp[k] = arr[j];
             j++; k++;
         }
     }

     while (i <= leftEnd) {
         temp[k] = arr[i];
         i++; k++;
     }
     while (j <= rightEnd) {
         temp[k] = arr[j];
         j++; k++;
     }

     for (i = leftStart; i <= rightEnd; i++) {
         arr[i] = temp[i];
     }
}
```

Once we hit the end of one of the two subarrays, copy the remaining elements of the larger subarray to temp.

A Method for Merging Subarrays

```
private static void merge(int[] arr, int[] temp,
  int leftStart, int leftEnd, int rightStart, int rightEnd) {
     int i = leftStart;    // index into left subarray
     int j = rightStart;   // index into right subarray
     int k = leftStart;    // index into temp

     while (i <= leftEnd && j <= rightEnd) {
         if (arr[i] < arr[j]) {
             temp[k] = arr[i];
             i++; k++;
         } else {
             temp[k] = arr[j];
             j++; k++;
         }
     }

     while (i <= leftEnd) {
         temp[k] = arr[i];
         i++; k++;
     }
     while (j <= rightEnd) {
         temp[k] = arr[j];
         j++; k++;
     }

     for (i = leftStart; i <= rightEnd; i++) {
         arr[i] = temp[i];
     }
}
```

Copy all the elements from temp back to the array using the correct starting and ending point.

Methods for Mergesort

- The mergeSort method creates the temporary array and makes the initial call to a separate *recursive* method:

```
public static void mergeSort(int[] arr) {
    int[] temp = new int[arr.length];
    mSort(arr, temp, 0, arr.length - 1);
}
```

- The recursive merge sort method:

```
private static void mSort(int[] arr, int[] temp,
   int start, int end) {

    if (start < end) {
        int middle = (start + end)/2;
        mSort(arr, temp, start, middle);
        mSort(arr, temp, middle + 1, end);

       merge(arr, temp, start, middle, middle + 1, end);
    } // if
}
```

Methods for Mergesort

- The mergeSort method creates the temporary array and makes the initial call to a separate *recursive* method:

```
public static void mergeSort(int[] arr) {
    int[] temp = new int[arr.length];
    mSort(arr, temp, 0, arr.length - 1);
}
```

- The recursive merge sort method:

```
private static void mSort(int[] arr, int[] temp,
   int start, int end) {

    if (start < end) {
       int middle = (start + end)/2;
       mSort(arr, temp, start, middle);
       mSort(arr, temp, middle + 1, end);

       merge(arr, temp, start, middle, middle + 1, end);
    } // if
}
```

Methods for Mergesort

- The mergeSort method creates the temporary array and makes the initial call to a separate *recursive* method :

```
public static void mergeSort(int[] arr) {
    int[] temp = new int[arr.length];
    mSort(arr, temp, 0, arr.length - 1);
}
```

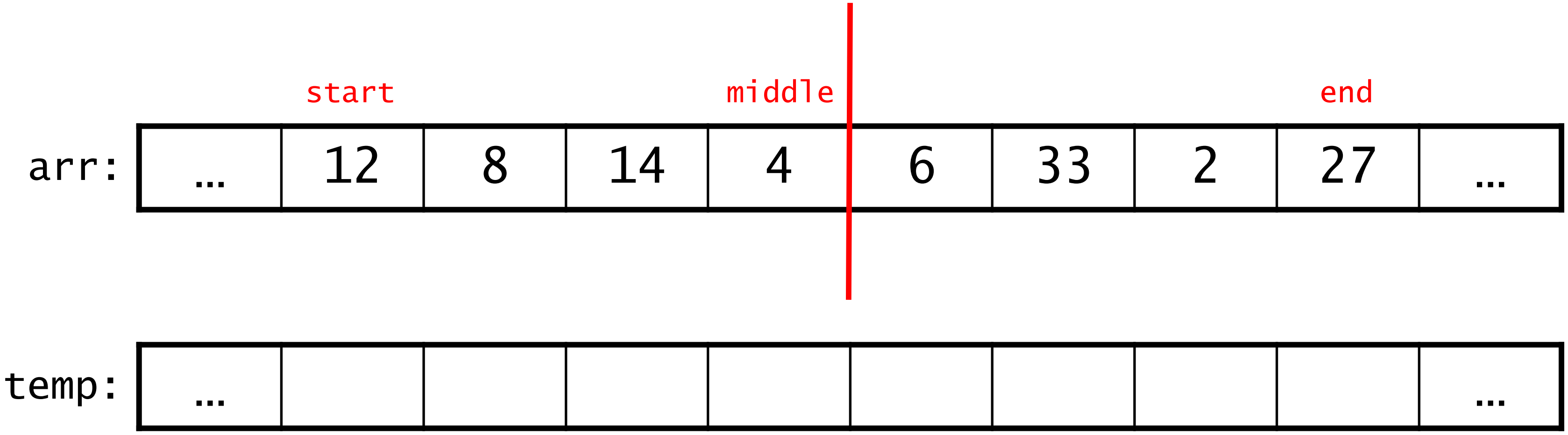
- The recursive merge sort method:

```
private static void mSort(int[] arr, int[] temp,
  int start, int end) {

    if (start < end) {
       int middle = (start + end)/2;
       mSort(arr, temp, start, middle);
       mSort(arr, temp, middle + 1, end);

       merge(arr, temp, start, middle, middle + 1, end);
    } // if
}
```

Methods for Mergesort

- Here's the key recursive method:

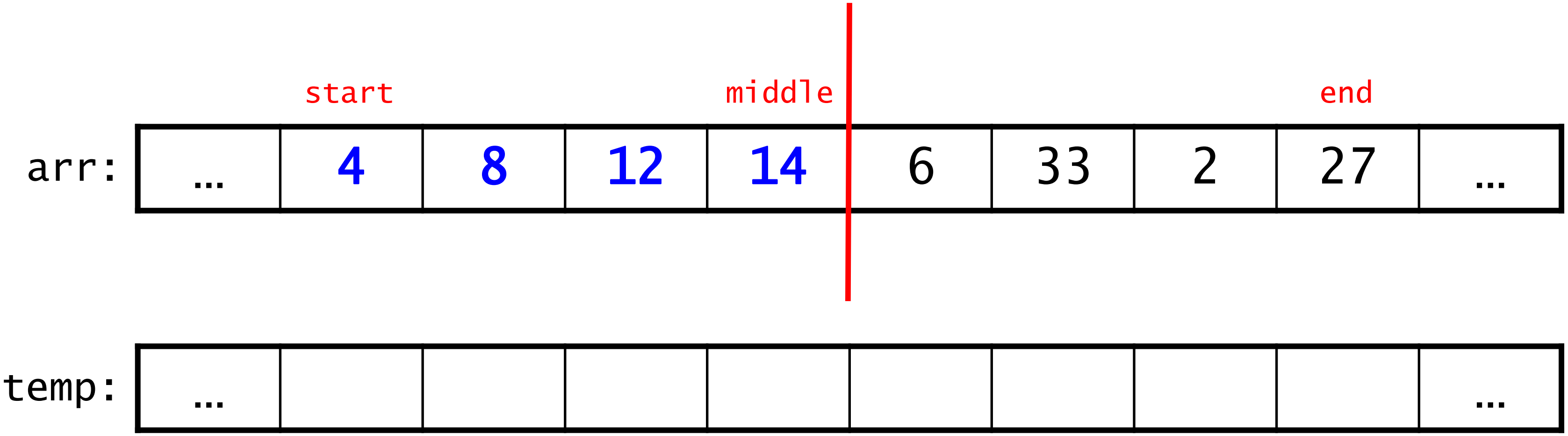
```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) {  // explicit base case: subarray of length 0 or 1
        return;
    } else {
        int middle = (start + end)/2;

        mSort(arr, temp, start, middle);
        mSort(arr, temp, middle + 1, end);

        merge(arr, temp, start, middle, middle + 1, end);
    }
}
```



Methods for Mergesort

- Here's the key recursive method:

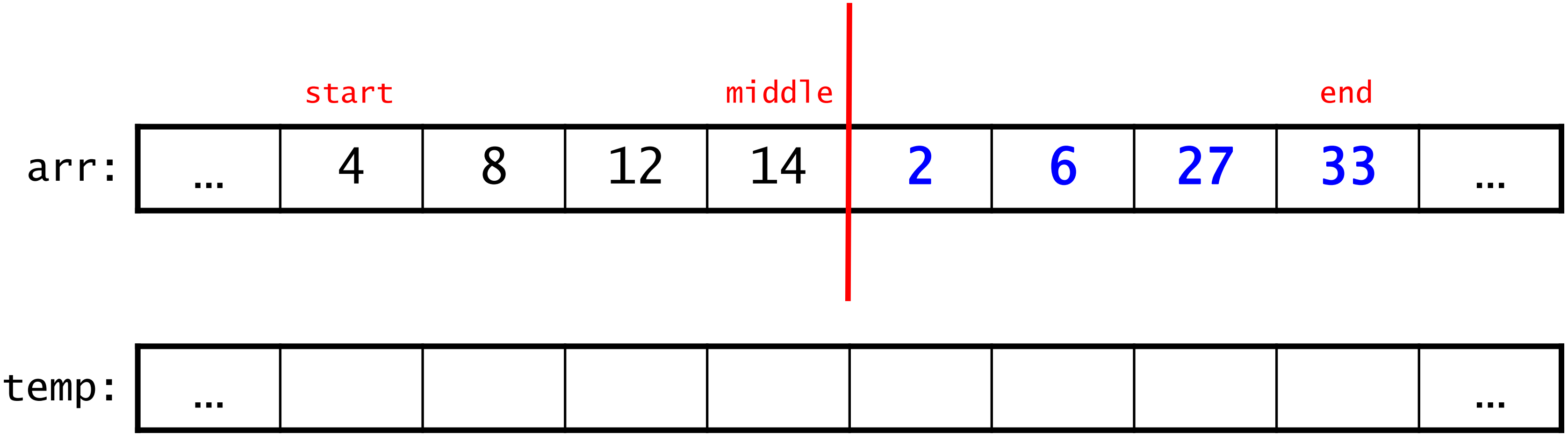
```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) {  // explicit base case: subarray of length 0 or 1
        return;
    } else {
        int middle = (start + end)/2;

        mSort(arr, temp, start, middle);
        mSort(arr, temp, middle + 1, end);

        merge(arr, temp, start, middle, middle + 1, end);
    }
}
```



Methods for Mergesort

- Here's the key recursive method:

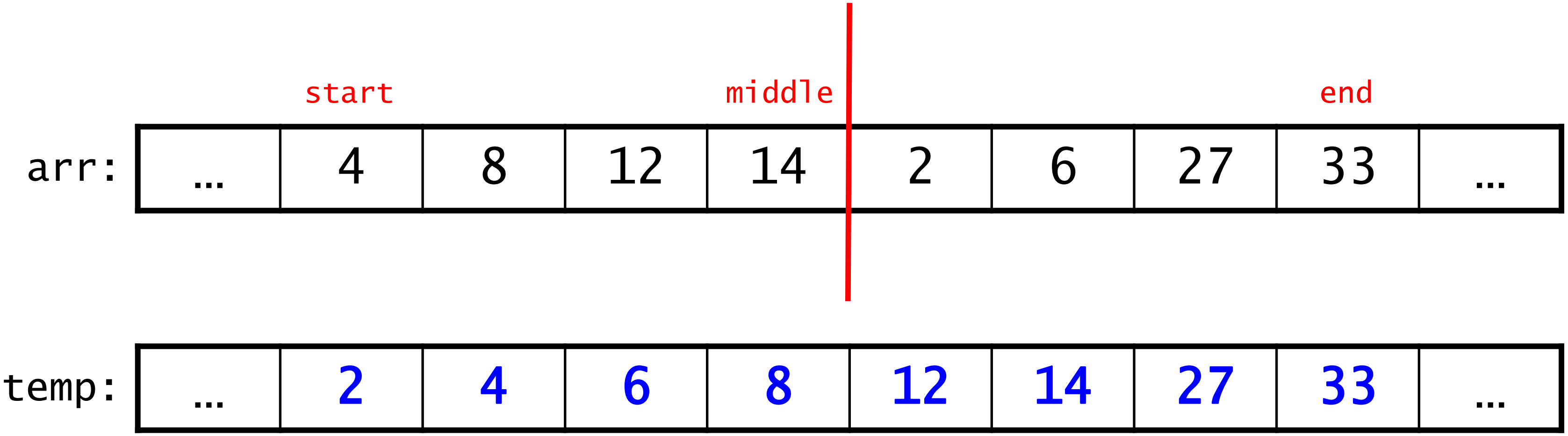
```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) {  // explicit base case: subarray of length 0 or 1
        return;
    } else {
        int middle = (start + end)/2;

        mSort(arr, temp, start, middle);
        mSort(arr, temp, middle + 1, end);

        merge(arr, temp, start, middle, middle + 1, end);
    }
}
```



Methods for Mergesort

- Here's the key recursive method:

```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) {  // explicit base case: subarray of length 0 or 1
        return;
    } else {
        int middle = (start + end)/2;

        mSort(arr, temp, start, middle);
        mSort(arr, temp, middle + 1, end);

        merge(arr, temp, start, middle, middle + 1, end);
    }
}
```

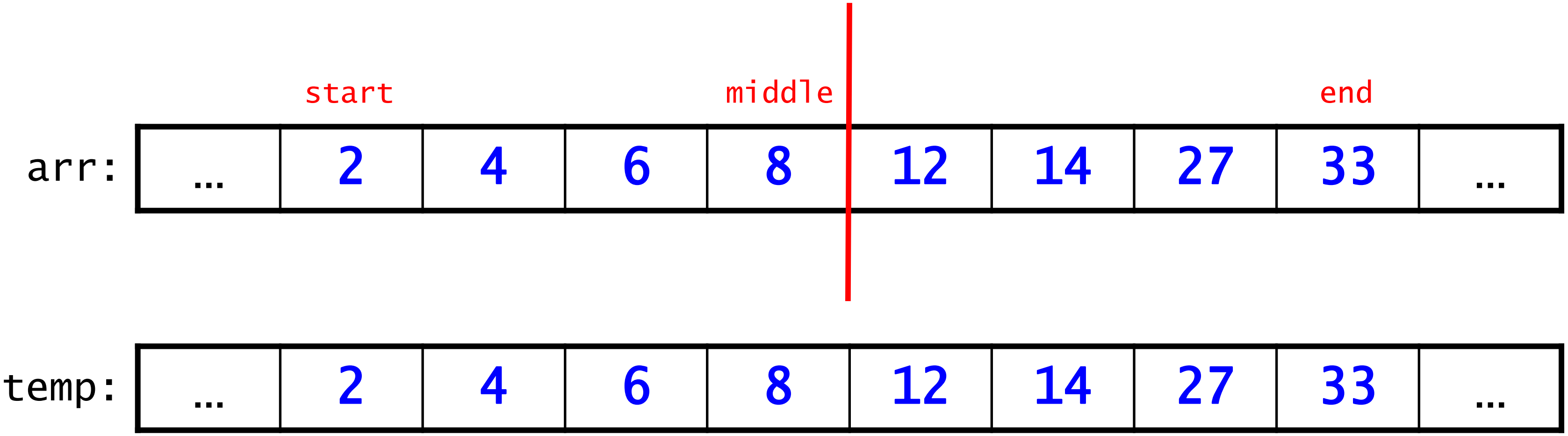


arr:	…	4	8	12	14	2	6	27	33	…
temp:	…	2	4	6	8	12	14	27	33	…

Methods for Mergesort

- Here's the key recursive method:

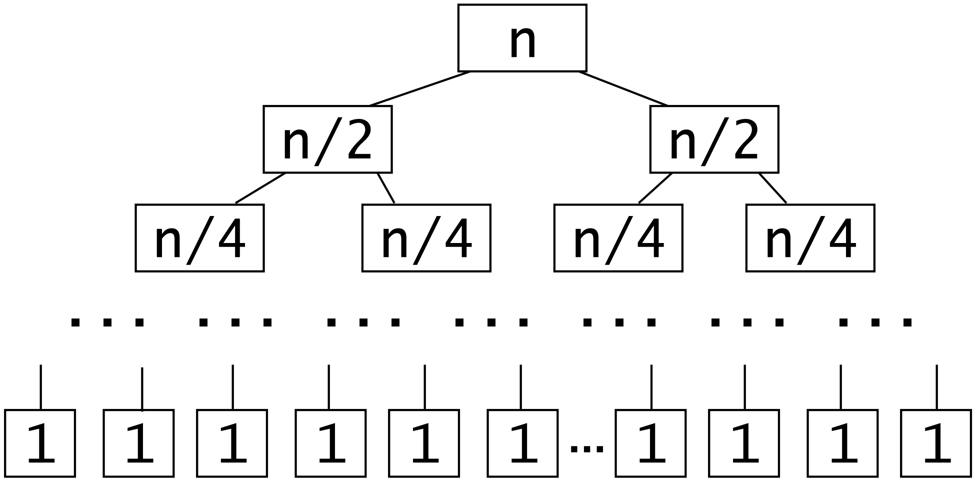
```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) {  // explicit base case: subarray of length 0 or 1
        return;
    } else {
        int middle = (start + end)/2;

        mSort(arr, temp, start, middle);
        mSort(arr, temp, middle + 1, end);

        merge(arr, temp, start, middle, middle + 1, end);
    }
}
```



arr:	...	2	4	6	8	12	14	27	33	...
temp:	...	2	4	6	8	12	14	27	33	...

Time Analysis of Mergesort

- Merging two halves of an array of size n requires 2n moves. Why?
- Mergesort repeatedly divides the array in half, so we have the following call tree (showing the sizes of the arrays):



moves

$2n$

$2*2*(n/2) = 2n$

$4*2*(n/4) = 2n$

...

  - at all but the last level of the call tree, there are 2n moves
  - how many levels are there? $\sim\log_2 n$
  - $M(n) = \sim 2n \log_2 n = O(n \log n)$
  - $C(n) = O(n \log n)$

Summary: Comparison-Based Sorting Algorithms

algorithm	best case	avg case	worst case	extra memory
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell sort	$O(n \log n)$	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(1)$
mergesort	**$O(n \log n)$**	**$O(n \log n)$**	**$O(n \log n)$**	**$O(n)$**

- Insertion sort is best for nearly sorted arrays.
- Mergesort has the best worst-case complexity, but requires extra memory – and moves to and from the temp array.
- Quicksort is comparable to mergesort in the average case. With a reasonable pivot choice, its worst case is seldom seen.
- Use `SortCount.java` to experiment.

Big-O

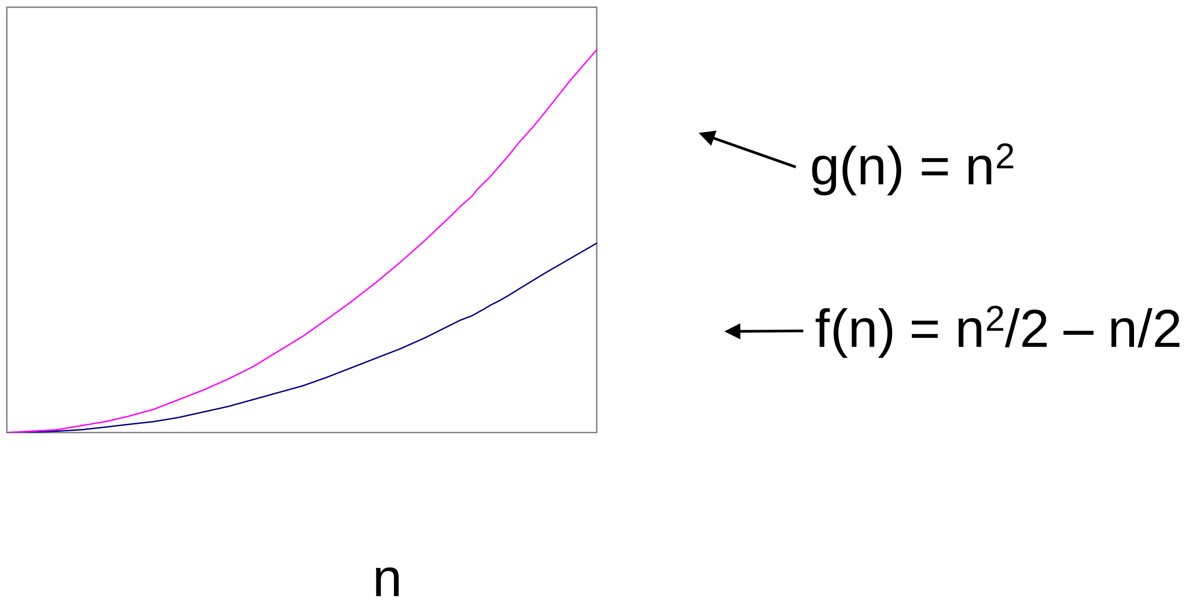
A little extra for those of who want to know the math behind the numbers!

Mathematical Definition of Big-*O* Notation

- $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \;<=\; c\,g(n)$ for all $n >= n_0$
- Example: $f(n) = n^2/2 - n/2$ is $O(n^2)$, because

$$n^2/2 - n/2 \;<=\; n^2 \;\text{ for all } n >= 0.$$

$c = 1$ $n_0 = 0$



- Big-*O* notation specifies an *upper bound* on a function f(n) as n grows large.

Big-*O* Notation and Tight Bounds

- Big-O notation provides an upper bound, *not* a tight bound (upper and lower).
- Example:
  - 3n – 3 is *O*(n^2) because 3n – 3 <= n^2 for all n >= 1
  - 3n – 3 is also *O*(2^n) because 3n – 3 <= 2^n for all n >= 1

- However, we generally try to use big-O notation to characterize a function as closely as possible – i.e., as if we were using it to specify a tight bound.
  - for our example, we would say that 3n – 3 is *O*(n)

Big-Theta Notation

- In theoretical computer science, *big-theta* notation (Θ) is used to specify a tight bound.
- $f(n) = \Theta(g(n))$ if there exist constants c_1, c_2, and n_0 such that $c_1 g(n) \;<=\; f(n) \;<=\; c_2 g(n)$ for all $n > n_0$
- Example: $f(n) = n^2/2 – n/2$ is $\Theta(n^2)$, because

$$(1/4)*n^2 <= n^2/2 – n/2 <= n^2 \quad \text{for all } n >= 2$$

$c_1 = 1/4$

$c_2 = 1$

$n_0 = 2$