Unit 7, Part 2

Sorting II: Divide-and-Conquer Algorithms, Distributive Sorting

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

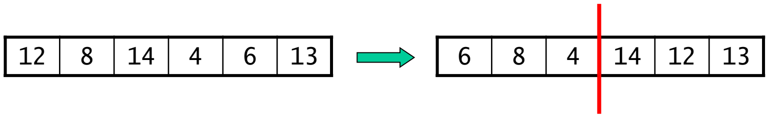
Quicksort

- Like bubble sort, quicksort uses an approach based on swapping out-of-order elements, but it's more efficient.

- 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 left array <= each element in right array

    example:



12	8	14	4	6	13

→

6	8	4	14	12	13

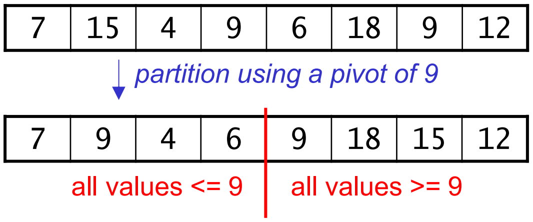
  - *conquer:* apply quicksort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* nothing needs to be done, because of the way we formed the subarrays

Partitioning an Array Using a Pivot

- The process that quicksort uses to rearrange the elements is known as *partitioning* the array.

- It uses one of the values in the array as a *pivot,* rearranging the elements to produce two subarrays:
  - left subarray: all values <= pivot
  - right subarray: all values >= pivot

equivalent to the criterion on the previous page.



7	15	4	9	6	18	9	12

partition using a pivot of 9

7	9	4	6	9	18	15	12

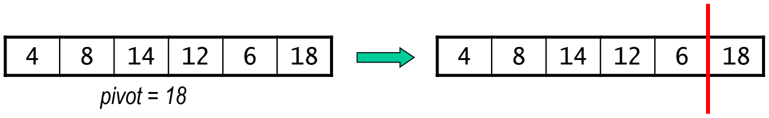
all values <= 9 | all values >= 9

- The subarrays will *not* always have the same length.

- This 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



4	8	14	12	6	18

pivot = 18

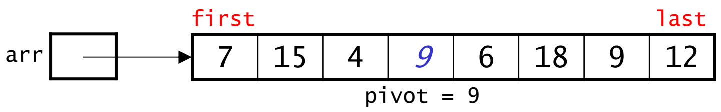
4	8	14	12	6	18

- 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



first							last
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

- *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`.

- Subarrays: left = from `first` to `j`, right = from `j+1` to `last`

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

Partitioning Example 2

- Start (pivot = 13):

i									j
	24	5	2	*13*	18	4	20	19	

- Find:

i					j		
24	5	2	13	18	4	20	19

- Swap:

i					j		
4	5	2	13	18	24	20	19

- Find:

			i j				
4	5	2	13	18	24	20	19

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

- Subarrays:

			i j				
4	5	2	13	18	24	20	19

Partitioning Example 3 (done together)

- Start (pivot = 5):

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

- Find:

4	14	7	5	2	19	26	6

Partitioning Example 4

- Start
  (pivot = 15):

i									j
	8	10	7	*15*	20	9	6	18	

- Find:

8	10	7	15	20	9	6	18

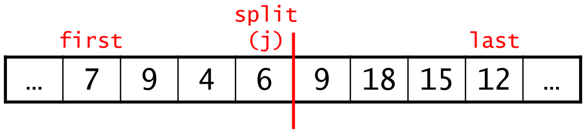
partition() Helper Method

```
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
    while (true) {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j) {
            swap(arr, i, j);
        } else {
            return j;   // arr[j] = end of left array
        }
    }
}
```

	first							last	
…	7	15	4	9	6	18	9	12	…

Implementation of Quicksort

```
public static void quickSort(int[] arr) { // "wrapper" method
    if (arr.length <= 1) {
        return;
    }
    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) {  // if left subarray has 2+ values
        qSort(arr, first, split);  // sort it recursively!
    }
    if (last > split + 1) {        // if right has 2+ values
        qSort(arr, split + 1, last);  // sort it!
    }
}   // note: base case is when neither call is made!
```



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 $\log_2 n$:
  - let's say that you repeatedly divide n by 2 (using integer division)
  - $\log_2 n$ 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 Logs (cont.)

- $O(\log n)$ algorithm – one in which the number of operations is proportional to $\log_b n$ for any base b
- $\log_b n$ grows much more slowly than n

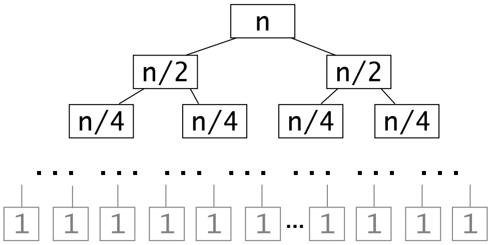
n	$\log_2 n$
2	1
1024 (1K)	10
1024*1024 (1M)	20
1024*1024*1024 (1G)	30

- Thus, for large values of n:
  - a $O(\log n)$ algorithm is much faster than a $O(n)$ algorithm
    - $\log n$ << n
  - a $O(n \log n)$ algorithm is much faster than a $O(n^2)$ algorithm
    - $n * \log n$ << $n * n$
      $n \log n$ << n^2

it's also faster than a $O(n^{1.5})$ algorithm like Shell sort

Time Analysis of Quicksort

- Partitioning an array of length n requires approx. n comparisons.
  - most elements are compared with the pivot once; a few twice
- *best case:* partitioning always divides the array in half
  - repeated recursive calls give:

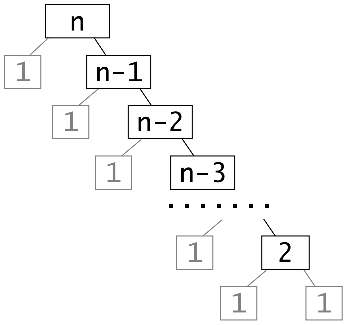


comparisons
n
2*(n/2) = n
4*(n/4) = n
...
0

  - at each "row" except the bottom, we perform n comparisons
  - there are ________ rows that include comparisons
  - C(n) = ?
  - Similarly, M(n) and running time are both ____________

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 $O(n^2)$.

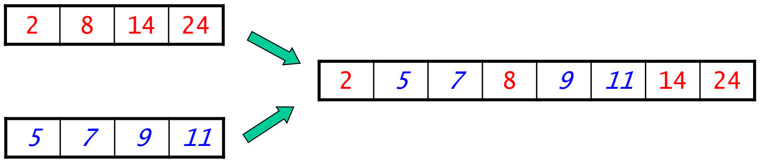
- *average case* is harder to analyze
  - $C(n) > n \log_2 n$, but it's still $O(n \log n)$

Mergesort

- The algorithms we've seen so far have sorted the array in place.
  - use only a small amount of additional memory

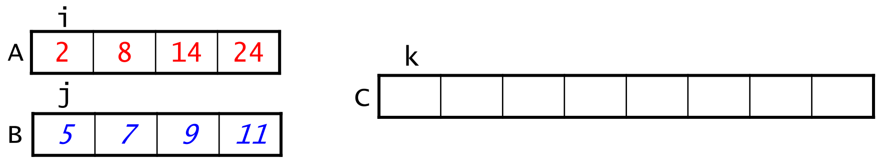
- Mergesort requires an additional temporary array of the same size as the original one.
  - it needs $O(n)$ additional space, where n is the array size

- It is based on the process of *merging* two sorted arrays.
  - example:

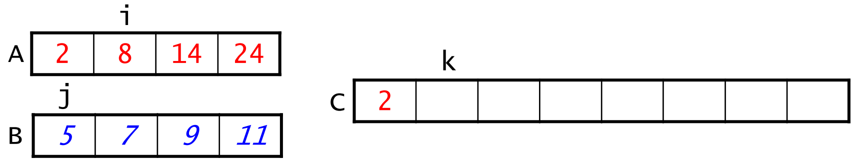


Merging Sorted Arrays

- 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:

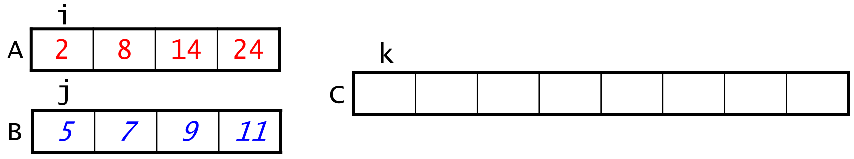


- We repeatedly do the following:
  - compare A[i] and B[j]
  - copy the smaller of the two to C[k]
  - increment the index of the array whose element was copied
  - increment k

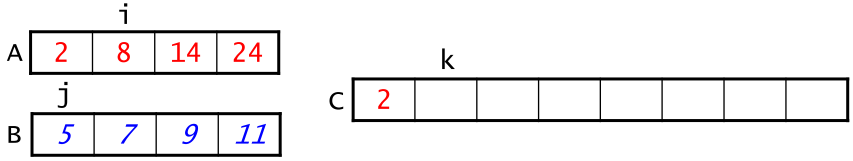


Merging Sorted Arrays (cont.)

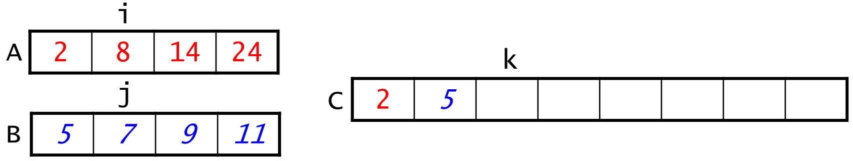
- Starting point:



- After the first copy:

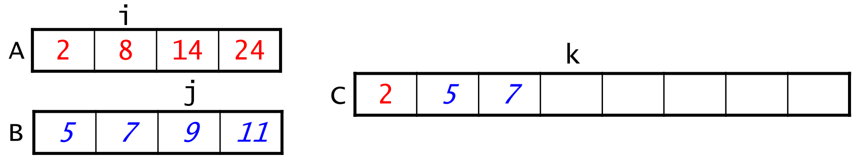


- After the second copy:

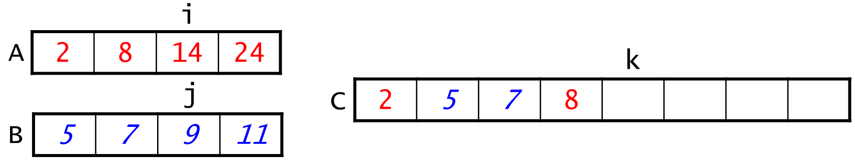


Merging Sorted Arrays (cont.)

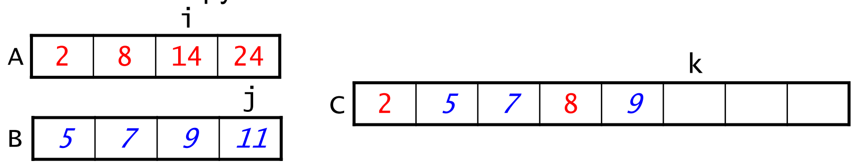
- After the third copy:



- After the fourth copy:

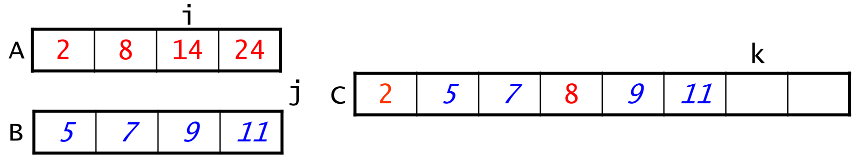


- After the fifth copy:

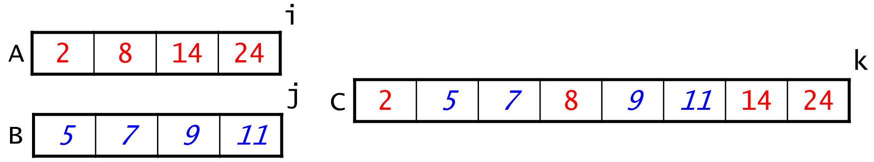


Merging Sorted Arrays (cont.)

- After the sixth copy:

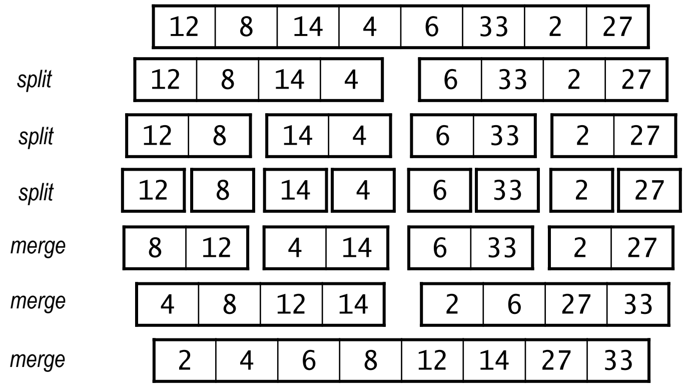


- There's nothing left in B, so we simply copy the remaining elements from A:



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

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

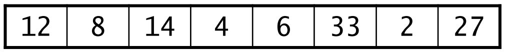
merge 8 12 | 4 14 | 6 33 | 2 27

merge 4 8 12 14 | 2 6 27 33

merge 2 4 6 8 12 14 27 33

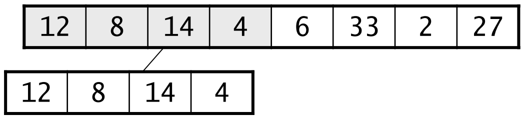
Tracing the Calls to Mergesort

the initial call is made to sort the entire array:



12 8 14 4 6 33 2 27

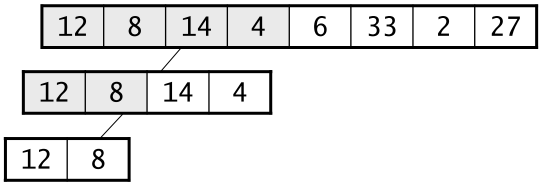
split into two 4-element subarrays, and make a recursive call to sort the left subarray:



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:



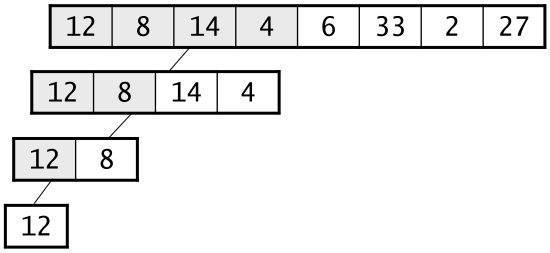
12 8 14 4 6 33 2 27

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:



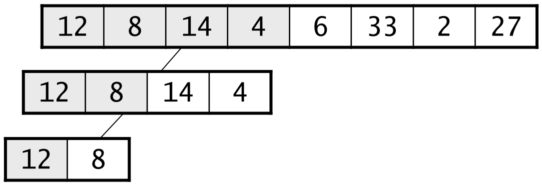
12 | 8 | 14 | 4 | 6 | 33 | 2 | 27

12 | 8 | 14 | 4

12 | 8

12

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



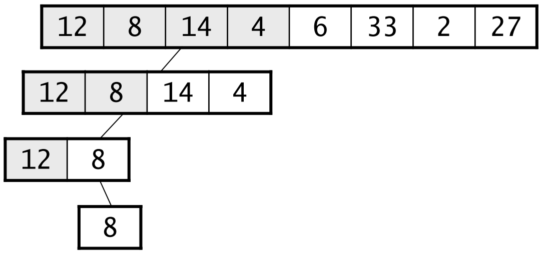
12 | 8 | 14 | 4 | 6 | 33 | 2 | 27

12 | 8 | 14 | 4

12 | 8

Tracing the Calls to Mergesort

make a recursive call to sort its right subarray:



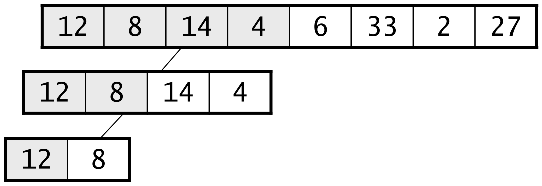
12 | 8 | 14 | 4 | 6 | 33 | 2 | 27

12 | 8 | 14 | 4

12 | 8

8

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



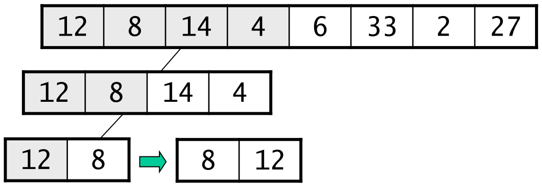
12 | 8 | 14 | 4 | 6 | 33 | 2 | 27

12 | 8 | 14 | 4

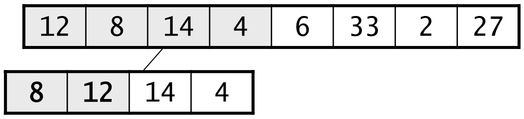
12 | 8

Tracing the Calls to Mergesort

merge the sorted halves of {12, 8}:

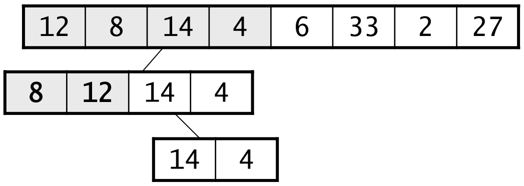


end of the method, so return to the call for the 4-element subarray, which now has a sorted left subarray:

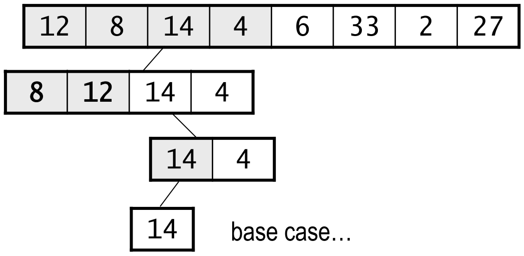


Tracing the Calls to Mergesort

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

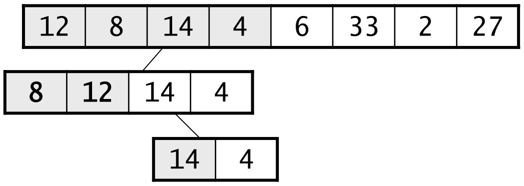


split it into two 1-element subarrays, and make a recursive call to sort the left subarray:



Tracing the Calls to Mergesort

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

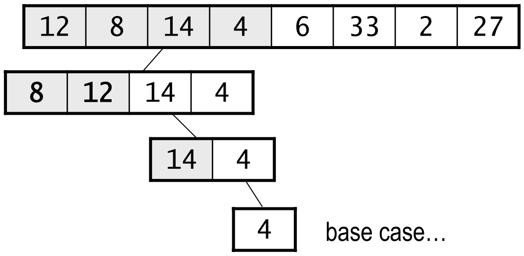


12	8	14	4	6	33	2	27

8	12	14	4

14	4

make a recursive call to sort its right subarray:



12	8	14	4	6	33	2	27

8	12	14	4

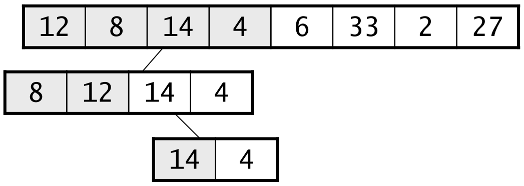
14	4

4

base case…

Tracing the Calls to Mergesort

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

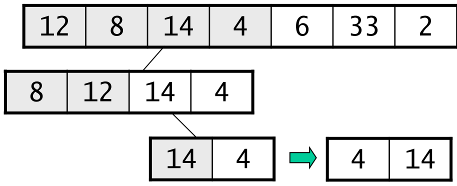


12	8	14	4	6	33	2	27

8	12	14	4

14	4

merge the sorted halves of {14, 4}:



12	8	14	4	6	33	2	27

8	12	14	4

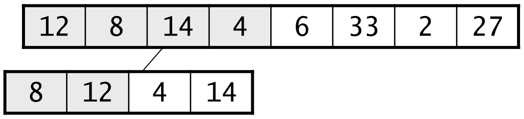
14	4

➡

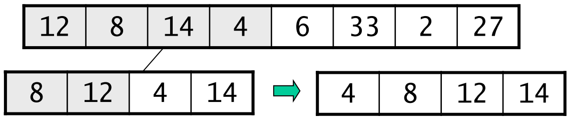
4	14

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:

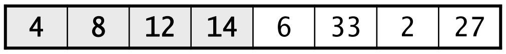


merge the 2-element subarrays:

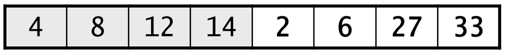


Tracing the Calls to Mergesort

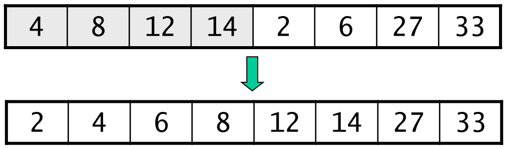
end of the method, so return to the call for the original array, which now has a sorted left subarray:



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

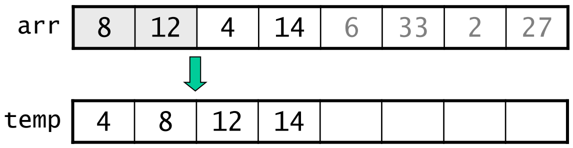


finally, merge the sorted 4-element subarrays to get a fully sorted 8-element array:



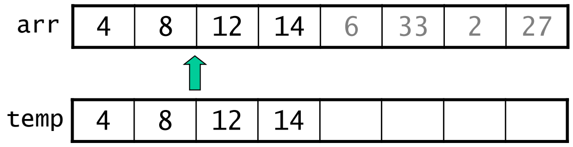
Implementing Mergesort

- In theory, we could create new arrays for each new pair of subarrays, and merge them back into the array that was split.
- Instead, we'll create a temp. array of the same size as the original.
  - 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:



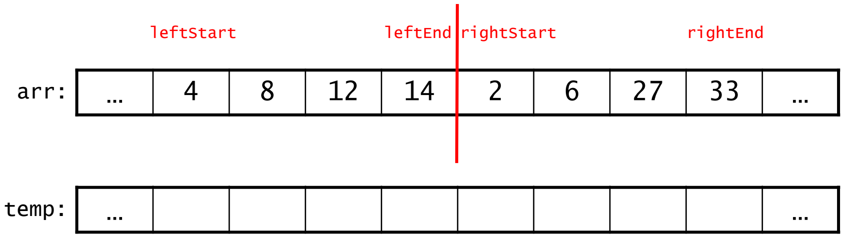
arr	4	8	12	14	6	33	2	27
temp	4	8	12	14				

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) { // both subarrays still have values
         if (arr[i] < arr[j]) {
             temp[k] = arr[i];
             i++; k++;
         } else {
             temp[k] = arr[j];
             j++; k++;
         }
     }
     ...
}
```



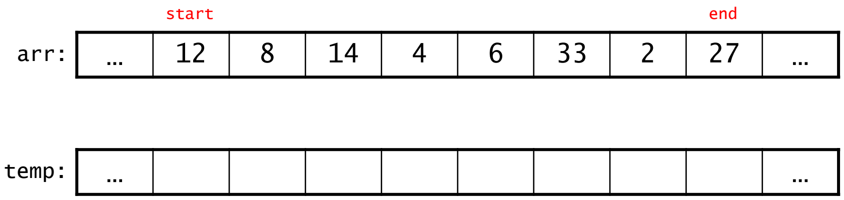
leftStart leftEnd rightStart rightEnd

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

Methods for Mergesort

- Here's the key recursive method:

```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) {  // 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);
    }
}
```



start end

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

Methods for Mergesort

- Here's the key recursive method:

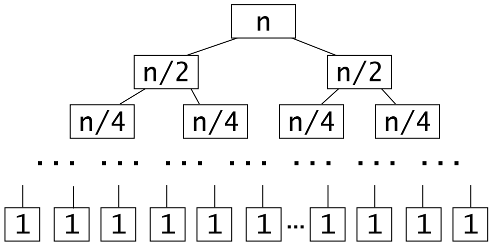
```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) {  // base case: subarray of length 0 or 1
        return;
    } else {
        int middle = (start + end)/2;

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

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

- We use a "wrapper" method to create the `temp` array, and to make the initial call to the recursive method:

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

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?
  - `M(n)` = ?
  - `C(n)` = ?

Summary: 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	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	*best/avg:* $O(\log n)$ *worst:* $O(n)$
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 O(n) extra memory – and moves to and from the temp. array.
- Quicksort is comparable to mergesort in the best/average case.
  - efficiency is also $O(n \log n)$, but less memory and fewer moves
  - its extra memory is from…
  - with a reasonable pivot choice, its worst case is seldom seen

Comparison-Based vs. Distributive Sorting

- All of the sorting algorithms we've considered have been *comparison-based:*
  - treat the values being sorted as wholes (comparing them)
  - don't "take them apart" in any way
  - all that matters is the relative order of the values
- No comparison-based sorting algorithm can do better than $O(n \log_2 n)$ on an array of length n.
  - $O(n \log_2 n)$ is a *lower bound* for such algorithms
- *Distributive* sorting algorithms do more than compare values; they perform calculations on the values being sorted.
- Moving beyond comparisons allows us to overcome the lower bound.
  - tradeoff: use more memory.

Distributive Sorting Example: Radix Sort

- Breaks each value into a sequence of **m** components, each of which has **k** possible values.
- Examples:

	m	**k**
• integer in range 0 ... 999	3	10
• string of 15 upper-case letters	15	26
• 32-bit integer	32	2 (in binary)
	4	256 (as bytes)

- Strategy: Distribute the values into "bins" according to their last component, then concatenate the results:

33 41 12 24 31 14 13 42 34

get: 41 31 | 12 42 | 33 13 | 24 14 34

- Repeat, moving back one component each time:

get: | | |

Analysis of Radix Sort

- m = number of components
  k = number of possible values for each component
  n = length of the array
- Time efficiency: $O(m*n)$
  - perform m distributions, each of which processes all n values
  - $O(m*n) < O(n \log n)$ when $m < \log n$
    so we want m to be small
- However, there is a tradeoff:
  - as m decreases, k increases
    - fewer components ➔ more possible values per component
  - as k increases, so does memory usage
    - need more bins for the results of each distribution
  - increased speed requires increased memory usage

Big-*O* Notation Revisited

- We've seen that we can group functions into classes by focusing on the fastest-growing term in the expression for the number of operations that they perform.
  - e.g., an algorithm that performs $n^2/2 - n/2$ operations is a $O(n^2)$-time or quadratic-time algorithm
- Common classes of algorithms:

slower

name	example expressions	big-O notation
constant time	$1, 7, 10$	$O(1)$
logarithmic time	$3\log_{10}n$, $\log_2 n + 5$	$O(\log n)$
linear time	$5n$, $10n - 2\log_2 n$	$O(n)$
$n\log n$ time	$4n\log_2 n$, $n\log_2 n + n$	$O(n\log n)$
quadratic time	$2n^2 + 3n$, $n^2 - 1$	$O(n^2)$
cubic time	$n^2 + 3n^3$, $5n^3 - 5$	$O(n^3)$
exponential time	2^n, $5e^n + 2n^2$	$O(c^n)$
factorial time	$3n!$, $5n + n!$	$O(n!)$

How Does the Number of Operations Scale?

- Let's say that we have a problem size of 1000, and we measure the number of operations performed by a given algorithm.
- If we double the problem size to 2000, how would the number of operations performed by an algorithm increase if it is:
  - $O(n)$-time
  - $O(n^2)$-time
  - $O(n^3)$-time
  - $O(\log_2 n)$-time
  - $O(2^n)$-time

How Does the Actual Running Time Scale?

- How much time is required to solve a problem of size n?
  - assume that each operation requires 1 μsec (1 x 10^{-6} sec)

time function	problem size (n)					
	10	20	30	40	50	60
n	.00001 s	.00002 s	.00003 s	.00004 s	.00005 s	.00006 s
n^2	.0001 s	.0004 s	.0009 s	.0016 s	.0025 s	.0036 s
n^5	.1 s	3.2 s	24.3 s	1.7 min	5.2 min	13.0 min
2^n	.001 s	1.0 s	17.9 min	12.7 days	35.7 yrs	36,600 yrs

- sample computations:
  - when n = 10, an n^2 algorithm performs 10^2 operations.
    10^2 * (1 x 10^{-6} sec) = .0001 sec
  - when n = 30, a 2^n algorithm performs 2^{30} operations.
    2^{30} * (1 x 10^{-6} sec) = 1073 sec = 17.9 min

What's the Largest Problem That Can Be Solved?

- What's the largest problem size n that can be solved in a given time T? (again assume 1 μsec per operation)

time function	time available (T)			
	1 min	1 hour	1 week	1 year
n	60,000,000	3.6 x 10^9	6.0 x 10^{11}	3.1 x 10^{13}
n^2	7745	60,000	777,688	5,615,692
n^5	35	81	227	500
2^n	25	31	39	44

- sample computations:
  - 1 hour = 3600 sec
    that's enough time for 3600/(1 x 10^{-6}) = 3.6 x 10^9 operations
    - n^2 algorithm:
      $n^2 = 3.6 \times 10^9$ → $n = (3.6 \times 10^9)^{1/2} = 60{,}000$
    - 2^n algorithm:
      $2^n = 3.6 \times 10^9$ → $n = \log_2(3.6 \times 10^9)$ ~= 31