Lecture 21
Sorting
CSE11 Fall '13

Sorting

Unsorted Array

indices

0	1	2	3	4	5	6	7	8	9
8	**3**	**14**	**23**	**11**	**9**	**-4**	**2**	**7**	**1**

values

Sorted Array

0	1	2	3	4	5	6	7	8	9
-4	**1**	**2**	**3**	**7**	**8**	**9**	**11**	**14**	**23**

Sorting

- You can sort an array of objects **if there is a comparison method.**
  - e.g. String.compareTo(another String)
- We'll describe a number of different algorithms to sort arrays.
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort
- Use integer arrays as examples.

compareTo()

- Many java objects properly define the compareTo function
  - obj1.compareTo(obj2)
    - Returns < 0 if obj1 “less than” obj2
    - Returns 0 if obj1 “equals” obj2
    - Returns > 0 if obj1 “greater than” obj2
  - The specific definition of “less than”, “equals”, “greater than” depends upon the class
    - e.g. Strings compare lexicographically
    - When you define your own “comparable” objects you define what this means. (e.g. You could compare areas of graphical objects as an non-obvious example)

BubbleSort

0	1	2	3	4	5	6	7	8	9
8	**3**	**14**	**23**	**11**	**9**	**-4**	**2**	**7**	**1**

- Basic Idea
  - compare A[i] and A[i+1]
    - If A[i] > A[i+1], **exchange** the array elements
  - First time through the array – largest element ends at highest index. It “bubbles” to the end of the array.
  - Next time through the array – the next largest element at highest – 1 index
  - Repeat N times to sort the array
    - Means you might do about N^2 comparisons

BubbleSort

i=0

0	1	2	3	4	5	6	7	8	9
8	3	14	23	11	9	-4	2	7	1

swap >

i=0

0	1	2	3	4	5	6	7	8	9
3	8	14	23	11	9	-4	2	7	1

i=1

0	1	2	3	4	5	6	7	8	9
3	8	14	23	11	9	-4	2	7	1

no swap <

i=2

0	1	2	3	4	5	6	7	8	9
3	8	14	23	11	9	-4	2	7	1

no swap <

i=3

0	1	2	3	4	5	6	7	8	9
3	8	14	23	11	9	-4	2	7	1

swap >

BubbleSort

	0	1	2	3	4	5	6	7	8	9
i=4	3	8	14	11	23	9	-4	2	7	1

swap >

	0	1	2	3	4	5	6	7	8	9
i=5	3	8	14	11	9	23	-4	2	7	1

swap >

	0	1	2	3	4	5	6	7	8	9
i=6	3	8	14	11	9	-4	23	2	7	1

swap >

	0	1	2	3	4	5	6	7	8	9
i=7	3	8	14	11	9	-4	2	23	7	1

swap >

	0	1	2	3	4	5	6	7	8	9
i=8	3	8	14	11	9	-4	2	7	23	1

swap >

BubbleSort

0	1	2	3	4	5	6	7	8	9
3	8	14	11	9	-4	2	7	1	23

```
Largest number (23) bubbled to the end
Rest of array not sorted
```

Do it again for one size smaller array

i=0

0	1	2	3	4	5	6	7	8	9
3	8	14	11	9	-4	2	7	1	23

no swap

>

i=1

0	1	2	3	4	5	6	7	8	9
3	8	14	11	9	-4	2	7	1	23

no swap

>

i=2

0	1	2	3	4	5	6	7	8	9
3	8	14	11	9	-4	2	7	1	23

swap

>

BubbleSort

0	1	2	3	4	5	6	7	8	9
3	8	11	9	-4	2	7	1	14	23

>

```
Second Largest number (14) bubbled to the
end of the shorter array
Rest of array not sorted
```

Do it again for one size smaller array
Keep doing this until you have the entire array sorted

0	1	2	3	4	5	6	7	8	9
-4	1	2	3	7	8	9	11	14	23

>

>

Analysis of Bubblesort

- Array of length N
  - # of comparisons =
    - (N-1) + (N-2) + (N-3) ... (1) = $N^2 – N$
- This is the simplest sorting algorithm to implement.
- It's also the *least* efficient
  - If asked in a job interview to do a sort, don't do this :-)

Selection Sort

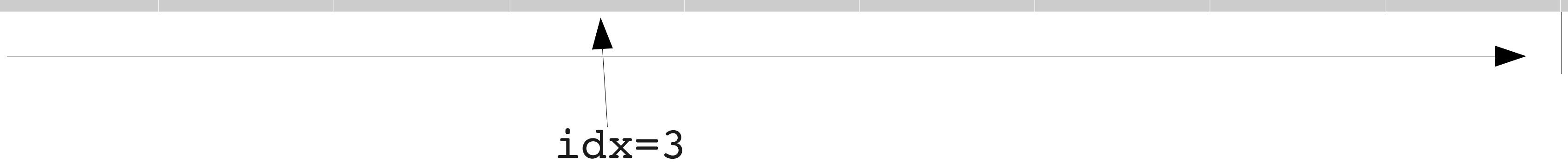
- Similar to Bubblesort
  - instead of doing pairwise comparisons, find the largest element in the array, **select it, and exchange it** with the largest index (A.length - 1)

i=9

0	1	2	3	4	5	6	7	8	9
8	3	14	23	11	9	-4	2	7	1

Maxvalue = 1; scan array and find the ***index*** of the largest value > 1

0	1	2	3	4	5	6	7	8	9
8	3	14	23	11	9	-4	2	7	1



Selection Sort

0	1	2	3	4	5	6	7	8	9
8	**3**	**14**	**1**	**11**	**9**	**-4**	**2**	**7**	**23**

```
Exchange A[i] and A[idx], Then
look at the index A.length-2
```

i=8

0	1	2	3	4	5	6	7	8	9
8	**3**	**14**	**1**	**11**	**9**	**-4**	**2**	**7**	**23**

Maxvalue =7; scan array and find the ***index*** of the largest value > 7

0	1	2	3	4	5	6	7	8	9
8	**3**	**14**	**1**	**11**	**9**	**-4**	**2**	**7**	**23**

`idx=2`

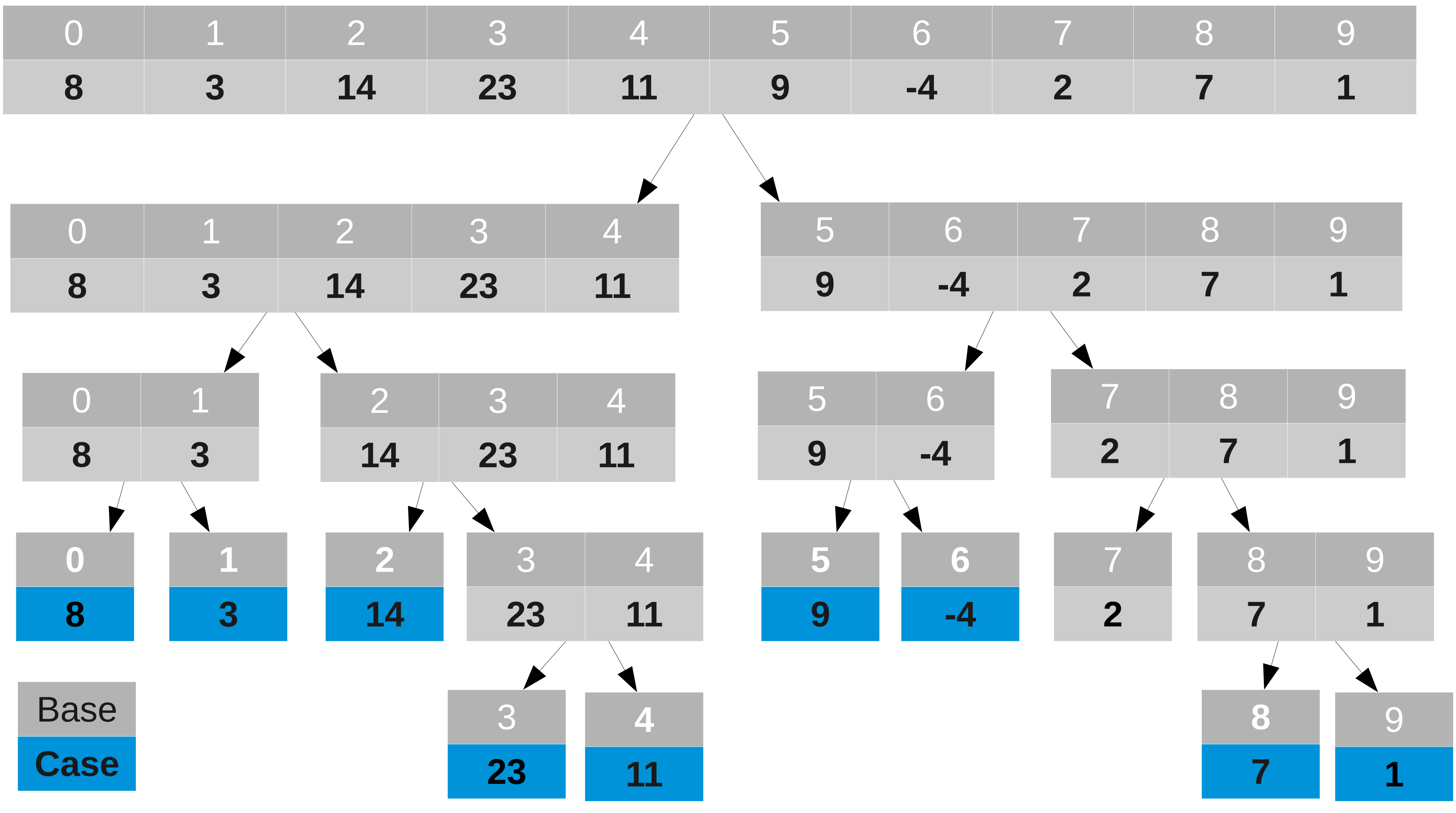
Keep Doing this until the entire array is sorted

Merge Sort

- This is “divide and conquer” algorithm
- It can be coded as an iteration or recursively
- We'll describe it graphically, then in code.

Merge Sort

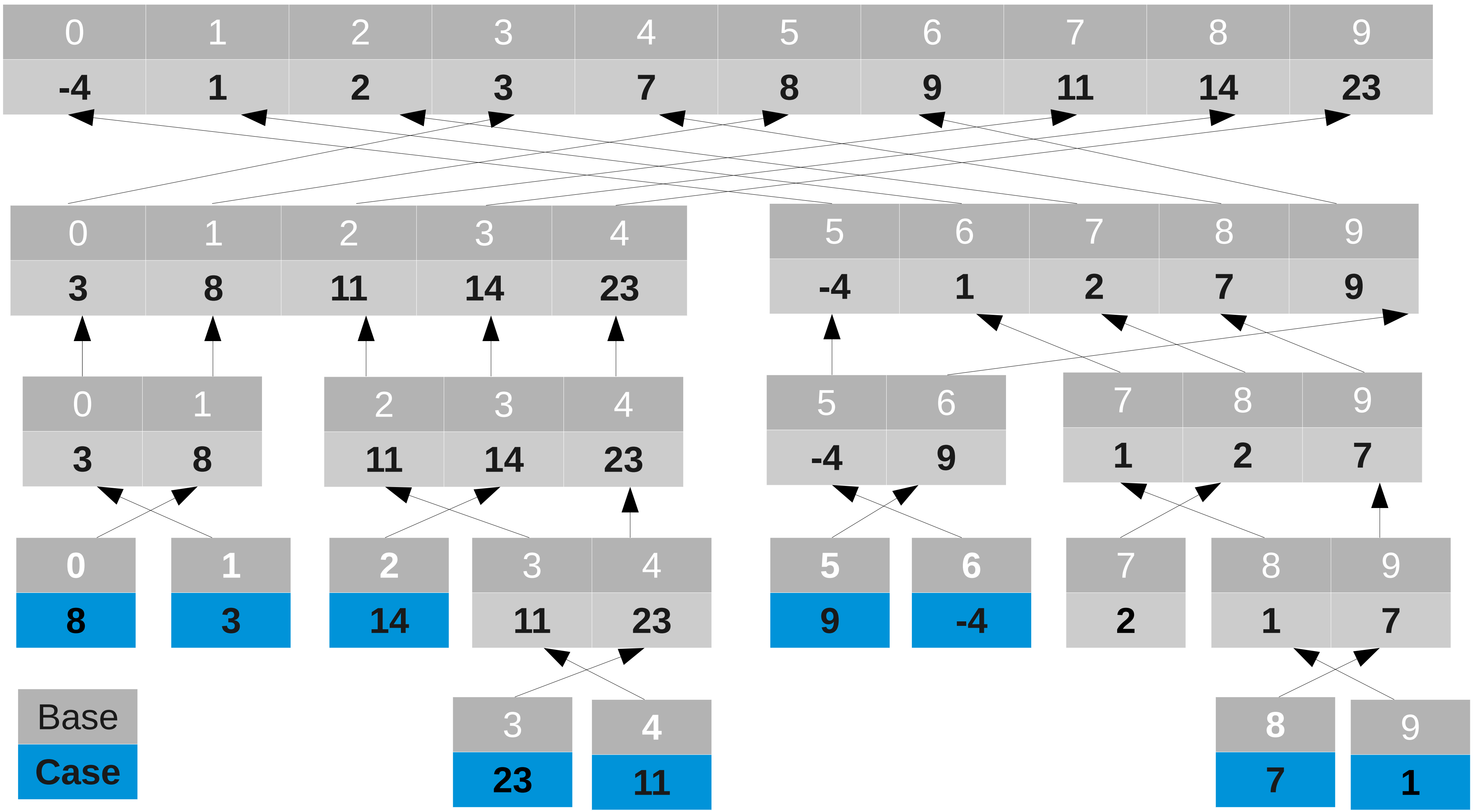
Divide



Merge

Each Level is sorted

Merge the sorted arrays from the level below



How does merging work?

left =0

0	1	2	3	4
3	**8**	**11**	**14**	**23**

right = 5

5	6	7	8	9
-4	**1**	**2**	**7**	**9**

@i=3

left=0

right=8

0	1	2	3	4	5	6	7	8	9
-4	**1**	**2**	?						

Dest

```
for (i = 0; i < Dest.length; i++) {
    if (left > 4 || right > 9) break;
    if (A[left] < A[right])
        Dest[i] = A[left++];
    else
        Dest[i] = A[right++];
}
// Copy whatever is left over
while (left < 4) Dest[i++] A[left++];
while (right < 9) Dest[i++] A[right++];
```

How many comparisons in Merge Sort

- (N/2) + 2 (N/4) + 4(N/8) + .. 2^LogN (N/N)
- 2^0(N/2) + 2^1(N/4) + 2^2(N/8) + + 2^LogN
- ~N * LogN

- BubbleSort and Selection Sort ~ N^2
- MergeSort ~ N LogN

- N = 1000
  - (bubble/selection) ~ 1,000,000 comparisons
  - Merge sort ~ 12000 comparisons

Searching

- Linear search --
  - Start at index 0, go through entire array until you find what you are looking for
    - Very inefficient for large arrays
- Binary Search
  - Must have a sorted array to start with
  - Look at middle element of array, if element is < search, look at the right half, > then search in left half (if find a match you are done)
    - Continue dividing until either found or nothing is left to divide

Binary Search

- **Array must be sorted!**
- binsearch (14) find the index where “14” is the value, -1 if not found
  - leftidx = 0, rightidx = A.length - 1
  - Search index = (lftidx + rightidx)/2

0	1	2	3	4	5	6	7	8	9
-4	**1**	**2**	**3**	**7**	**8**	**9**	**11**	**14**	**23**

A[4] < 14, Look in right half of array

5	6	7	8	9
8	**9**	**11**	**14**	**23**

A[7] < 14, Look in right half of array

8	9
14	**23**

A[8] == 14, return 8

LogN comparisons to find an element

Insertion Sort

0	1	2	3	4	5	6	7	8	9
8	**3**	**14**	**23**	**11**	**9**	**-4**	**2**	**7**	**1**

Empty Sorted Array

0	1	2	3	4	5	6	7	8	9

i=0

0	1	2	3	4	5	6	7	8	9
8									

i=1. Search for index where 3 should go (a modification of binary search).
This is the **insertion** index. Put 3 in that place (0), make space if needed

0	1	2	3	4	5	6	7	8	9
3	**8**								

Insertion Sort

0	1	2	3	4	5	6	7	8	9
8	**3**	**14**	**23**	**11**	**9**	**-4**	**2**	**7**	**1**

i=2. Search for index where 14 should go. Put 14 in that place (2),

0	1	2	3	4	5	6	7	8	9
3	**8**	**14**							

0	1	2	3	4	5	6	7	8	9
3	**8**	**14**	**23**						

i=4. Search for index where 11 should go. Insert 11 in that place (2),

0	1	2	3	4	5	6	7	8	9
3	**8**	**11**	**14**	**23**					

N insertions, NLogN comparisons for search