COSC 121:
Computer Programming II

Dr. Bowen Hui
University of British Columbia
Okanagan

Review: Improved Sorting

- Insertion and selection sort algorithms:
  - Relatively easy to understand
  - Order N^2, given N items to sort
  - Not efficient when N is large

- Idea: Take advantage of the “halfing” technique used in binary search

- Faster algorithms that run in worst case of order N log N
  - Merge Sort
  - Quick Sort

Review: General Idea

- Based on a recursive process: divide and conquer

- Merge sort: Given an array to sort:
  - Split in half
  - Each half being sorted separately
  - Results are merged together
- When to stop? (base case)

- To sort each half, the algorithm repeats the same process

Recall: MergeSort.java

```
public class MergeSort
{
  // attribute
  private int[] numbers;

  // methods
  public MergeSort( int[] list ) {...}
  public void sort() {...}
  // other methods as needed...
}
```

Recall: Define Parameters for Recursion

```
public void sort()
{
  helper( 0, numbers.length-1 );
}

private void helper( int min, int max )
{
  // if one elem left, stop: sorted
  // else:
  // sort on left half
  // sort on right half
  // combine two halves
}
```

```
// main method to call for sorting
public void sort()
{
  helper( 0, numbers.length-1 );
}

// recursive method for partitioning the array into two halves
private void helper( int min, int max )
{
  System.out.println( "inside helper:" );
  System.out.println( "\tmin=" + min + ", max=" + max );
  if( min < max )
  {
    int mid = min + (max-min)/2;

    // sort each half independently
    helper( min,   mid );
    helper( mid+1, max );

    // when done, combine two sorted arrays
    combine( min, mid, max );
  }
  // otherwise, array is sorted
}
```

recursive method

method to do all the hard work

Main Steps in `combine()`

```
private void combine( int low, int mid, int high)
{
  // 1. create extra array called temp
  int[] temp = . . .

  // 2. setup indices
  int left  = . . .     // left side starts at low
  int right = . . .     // right side starts at mid+1
  int pos   = . . .     // pos (to merge) starts at low

  // 3. compare each value in left + right
  // copy smaller value into numbers[pos]
  while( left <= mid && right <= high ) { ... }

  // 4. copy rest of left side into numbers
  while( left <= mid ) { ... }
}
```

```java
private void combine( int low, int mid, int high)
{
  // 1. create extra array called temp
  int[] temp = new int[numbers.length];
  for( int i=low; i<=high; i++ )
    temp[i] = numbers[i];

  // 2. setup indices
  int left  = low;      // left side starts at low
  int right = mid + 1;  // right side starts at mid+1
  int pos   = low;      // pos (to merge) starts at low

  // 3. compare each value in left + right
  // copy smaller value into numbers[pos]
  while( left <= mid && right <= high ) { ... }

  // 4. copy rest of left side into numbers
  while( left <= mid ) { ... }
}
```

Tracing through MergeSort

- numbers = [75 9 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7

- Method calls:

helper(0,7)

Tracing through MergeSort

- numbers = [75 9 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7

- Method calls:

helper(0,7)

  helper(0,3)

Tracing through MergeSort

- numbers = [75 9 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7

- Method calls:

helper(0,7)

  helper(0,3)

    helper(0,1)

Tracing through MergeSort

- numbers = [75 9 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7

- Method calls:

```
helper(0,7)
  helper(0,3)
    helper(0,1)
      helper(0,0)                // one elem left
```

Tracing through MergeSort

- numbers = [75 9 39 42 61 21 56 32]
  0 1 2 3 4 5 6 7
- Method calls:

```
helper(0,7)
  helper(0,3)
    helper(0,1)
      helper(0,0)            // one elem left
      helper(1,1)            // one elem left
```

Tracing through MergeSort

- numbers = [75 9 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7
- Method calls:

```
helper(0,7)
  helper(0,3)
    helper(0,1)
      helper(0,0)                    // one elem left
      helper(1,1)                    // one elem left
      combine(0,0,1)
        temp = [75 9]
                0  1
```

Tracing through MergeSort

- numbers = [75 9 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7
- Method calls:

helper(0,7)

helper(0,3)

helper(0,1)

helper(0,0) // one elem left

helper(1,1) // one elem left

combine(0,0,1)

temp = [75 9]

0 1 left=0, right=1, pos=0

- numbers = [75 9 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7

Tracing through MergeSort

- numbers = [75 9 39 42 61 21 56 32]
  0 1 2 3 4 5 6 7
- Method calls:

helper(0,7)
  helper(0,3)
    helper(0,1)
      helper(0,0) // one elem left
      helper(1,1) // one elem left
      combine(0,0,1)
        temp = [75 9]
          0 1 left=0, right=1, pos=0

- numbers = [75 9 39 42 61 21 56 32]
  0 1 2 3 4 5 6 7

Tracing through MergeSort

- numbers = [75 9 39 42 61 21 56 32]
  0 1 2 3 4 5 6 7
- Method calls:

helper(0,7)
  helper(0,3)
    helper(0,1)
      helper(0,0) // one elem left
      helper(1,1) // one elem left
      combine(0,0,1)
        temp = [75 9]
        0 1 left=0, right=1, pos=0

- numbers = [9 9 39 42 61 21 56 32] when exit comparison loop
  0 1 2 3 4 5 6 7

Tracing through MergeSort

- numbers = [75 9 39 42 61 21 56 32]
  0 1 2 3 4 5 6 7
- Method calls:

helper(0,7)
  helper(0,3)
    helper(0,1)
      helper(0,0) // one elem left
      helper(1,1) // one elem left
      combine(0,0,1)
        temp = [75 9]
        0 1 left=0, right=1, pos=0

- numbers = [9 9 39 42 61 21 56 32] when exit comparison loop
  0 1 2 3 4 5 6 7
- numbers = [9 75 39 42 61 21 56 32] when exit `combine()`
  0 1 2 3 4 5 6 7

merged

Tracing through MergeSort

- numbers = [9 75 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7
- Method calls:

helper(0,7)

helper(0,3)

helper(0,1) just completed this

helper(2,3) work on this half now

helper(2,2) // one elem left

helper(3,3) // one elem left

combine(2,2,3)

Tracing through MergeSort

- numbers = [9 75 39 42 61 21 56 32]
  0 1 2 3 4 5 6 7
- Method calls:

helper(0,7)
  helper(0,3)
    helper(0,1) just completed this
    helper(2,3) work on this half now
      helper(2,2) // one elem left
      helper(3,3) // one elem left
      combine(2,2,3)
        temp = [. . . 39 42]
        0 1 2 3 left=2, right=3, pos=2

- numbers = [9 75 39 42 61 21 56 32]when exit comparison loop
  0 1 2 3 4 5 6 7
- Nothing left in the left half

Tracing through MergeSort

- numbers = [9 75 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7

- Method calls:

helper(0,7)

helper(0,3)

helper(0,1)

helper(2,3) just completed this

Tracing through MergeSort

- numbers = [9 75 39 42 61 21 56 32]
  0 1 2 3 4 5 6 7
- Method calls:

helper(0,7)
  helper(0,3)
    helper(0,1)
    helper(2,3)
    combine(0,1,3)
      temp = [9 75 39 42]
      0 1 2 3

two halves are *internally* sorted only

left=0, right=2, pos=0

- numbers = [9 75 39 42 61 21 56 32]
  0 1 2 3 4 5 6 7

Tracing through MergeSort

- numbers = [9 75 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7
- Method calls:

helper(0,7)

helper(0,3)

helper(0,1)

helper(2,3)

combine(0,1,3)

temp = [9 75 39 42]

0 1 2 3

two halves are *internally* sorted only

left=1, right=2, pos=1

- numbers = [9 39 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7

Tracing through MergeSort

- numbers = [9 75 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7
- Method calls:

helper(0,7)

helper(0,3)

helper(0,1)

helper(2,3)

combine(0,1,3)

temp = [9 75 39 42]

0 1 2 3

two halves are *internally* sorted only

left=1, right=3, pos=2

- numbers = [9 39 42 42 61 21 56 32]

  0 1 2 3 4 5 6 7

Tracing through MergeSort

- numbers = [9 75 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7
- Method calls:

helper(0,7)

helper(0,3)

helper(0,1)

helper(2,3)

combine(0,1,3)

temp = [9 75 39 42]

0 1 2 3

two halves are *internally* sorted only

copy remaining of left half

- numbers = [9 39 42 75 61 21 56 32]

  0 1 2 3 4 5 6 7

Tracing through MergeSort

- numbers = [9 75 39 42 61 21 56 32]

  0 1 2 3 4 5 6 7
- Method calls:

helper(0,7)

helper(0,3)

helper(0,1)

helper(2,3)

combine(0,1,3)

temp = [9 75 39 42]

0 1 2 3

two halves are *internally* sorted only

left=0, right=2, pos=0

left=1, right=2, pos=1

left=1, right=3, pos=2

left=1, right=4, pos=3

- numbers = [9 39 42 75 61 21 56 32] when exit `combine()`

  0 1 2 3 4 5 6 7

Tracing through MergeSort

- numbers = [9 39 42 75 61 21 56 32]

  0 1 2 3 4 5 6 7

- Method calls:

helper(0,7)

  helper(0,3) just completed this

  helper(4,7)

Tracing through MergeSort

- numbers = [9 75 39 42 61 21 56 32]
  0 1 2 3 4 5 6 7
- Method calls:

helper(0,7)
  helper(0,3)
  helper(4,7)
    helper(4,5)
      helper(4,4)
      helper(5,5)
      combine(4,4,5)
    helper(6,7)

- numbers = [9 39 42 75 21 61 56 32] when exit `combine()`
  0 1 2 3 4 5 6 7

Tracing through MergeSort

- numbers = [9 75 39 42 21 61 56 32]
  0 1 2 3 4 5 6 7
- Method calls:

helper(0,7)
  helper(0,3)
  helper(4,7)
    helper(4,5) just completed this
    helper(6,7)
      helper(6,6)
      helper(7,7)
      combine(6,6,7)

- numbers = [9 39 42 75 21 61 32 56] when exit `combine()`
  0 1 2 3 4 5 6 7

Tracing through MergeSort

- numbers = [9 75 39 42 21 61 32 56] two halves are *internally* sorted only

  0 1 2 3 4 5 6 7

- Method calls:

helper(0,7)

helper(0,3)

helper(4,7)

helper(4,5)

helper(6,7) just completed this

combine(4,5,7)

- numbers = [9 39 42 75 21 32 56 61] when exit `combine()`

  0 1 2 3 4 5 6 7

Tracing through MergeSort

- numbers = [9 39 42 75 21 32 56 61] two halves are *internally* sorted only

  0 1 2 3 4 5 6 7

- Method calls:

helper(0,7)

helper(0,3)

helper(4,7) just completed this

combine(0,3,7)

- numbers = [9 21 32 39 42 56 61 75] when exit `combine()`

  0 1 2 3 4 5 6 7

Printing Meaningful Output

- Not sure if your program works?
  - Now: use `println()` to show intermediate steps
  - Next year: unit testing
- Example:
  - Give test case to binary search
  - Expect to be found – actually found
  - Expect to not be found – actually didn't find it
  - Were we just lucky?
  - To know for sure: print out search indices at each step

Output from MergeDemo.java

```
75
9
39
42
61
21
56
32
inside helper:
        min=0, max=7
inside helper:
        min=0, max=3
inside helper:
        min=0, max=1
inside helper:
        min=0, max=0
inside helper:
        min=1, max=1
inside combine:
        min=0, mid=0, max=1
        temp has:                75 9
        numbers before combine: 75 9 39 42 61 21 56 32
        numbers after comparison: 9 9 39 42 61 21 56 32
        numbers after copy left half: 9 75 39 42 61 21 56 32
```

Cont.

```
inside helper:
        min=2, max=3
inside helper:
        min=2, max=2
inside helper:
        min=3, max=3
inside combine:
        min=2, mid=2, max=3
        temp has:               0 0 39 42
        numbers before combine: 9 75 39 42 61 21 56 32
        numbers after comparison: 9 75 39 42 61 21 56 32
        numbers after copy left half: 9 75 39 42 61 21 56 32
inside combine:
        min=0, mid=1, max=3
        temp has:               9 75 39 42
        numbers before combine: 9 75 39 42 61 21 56 32
        numbers after comparison: 9 39 42 42 61 21 56 32
        numbers after copy left half: 9 39 42 75 61 21 56 32
```

Cont.

```
inside helper:
        min=4, max=7
inside helper:
        min=4, max=5
inside helper:
        min=4, max=4
inside helper:
        min=5, max=5
inside combine:
        min=4, mid=4, max=5
        temp has:               0 0 0 0 61 21
        numbers before combine: 9 39 42 75 61 21 56 32
        numbers after comparison: 9 39 42 75 21 21 56 32
        numbers after copy left half: 9 39 42 75 21 61 56 32
inside helper:
        min=6, max=7
inside helper:
        min=6, max=6
inside helper:
        min=7, max=7
```

Cont.

```
inside combine:
        min=6, mid=6, max=7
        temp has:                   0 0 0 0 0 0 56 32
        numbers before combine: 9 39 42 75 21 61 56 32
        numbers after comparison: 9 39 42 75 21 61 32 32
        numbers after copy left half: 9 39 42 75 21 61 32 56
inside combine:
        min=4, mid=5, max=7
        temp has:                   0 0 0 0 21 61 32 56
        numbers before combine: 9 39 42 75 21 61 32 56
        numbers after comparison: 9 39 42 75 21 32 56 56
        numbers after copy left half: 9 39 42 75 21 32 56 61
inside combine:
        min=0, mid=3, max=7
        temp has:                   9 39 42 75 21 32 56 61
        numbers before combine: 9 39 42 75 21 32 56 61
        numbers after comparison: 9 21 32 39 42 56 61 61
        numbers after copy left half: 9 21 32 39 42 56 61 75
9
21
32
39
42
56
61
75
```

Merge Sort Summary

- Very intuitive and easy to remember
- New technique: recursion
- Sort method:
  - Manages recursive structure
- Combine method:
  - Keeping track of indices to sort in-place
- Next: Quick sort

Quick Sort

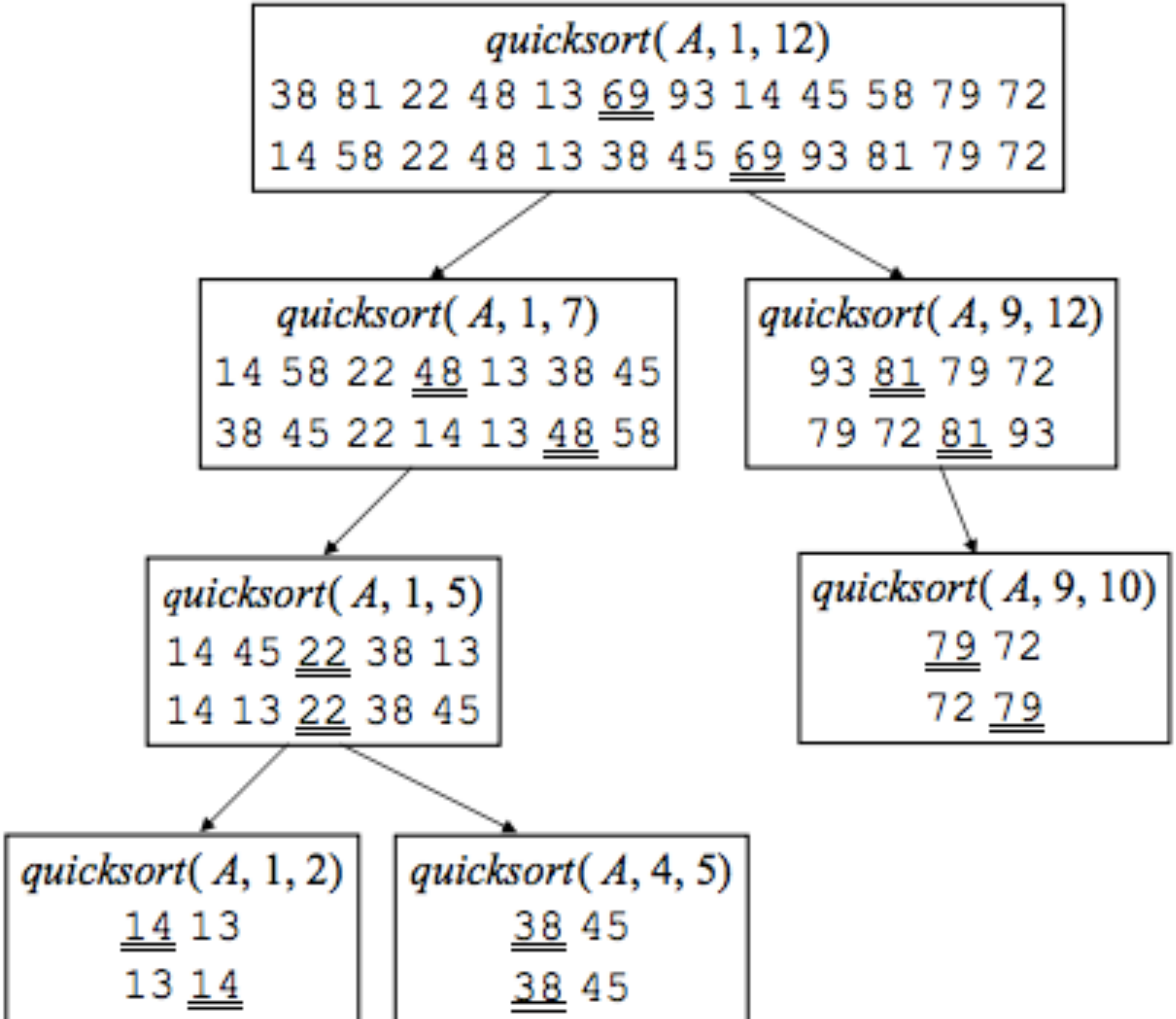
- The quick sort algorithm recursively sorts the array in-place
- Strategy:
  - Choose a pivot
  - Split array into two sub-arrays via pivot
  - Recursively sort the two sub-arrays
- Purpose: understand general concept
- Implementation not expected

Quick Sort Pseudocode

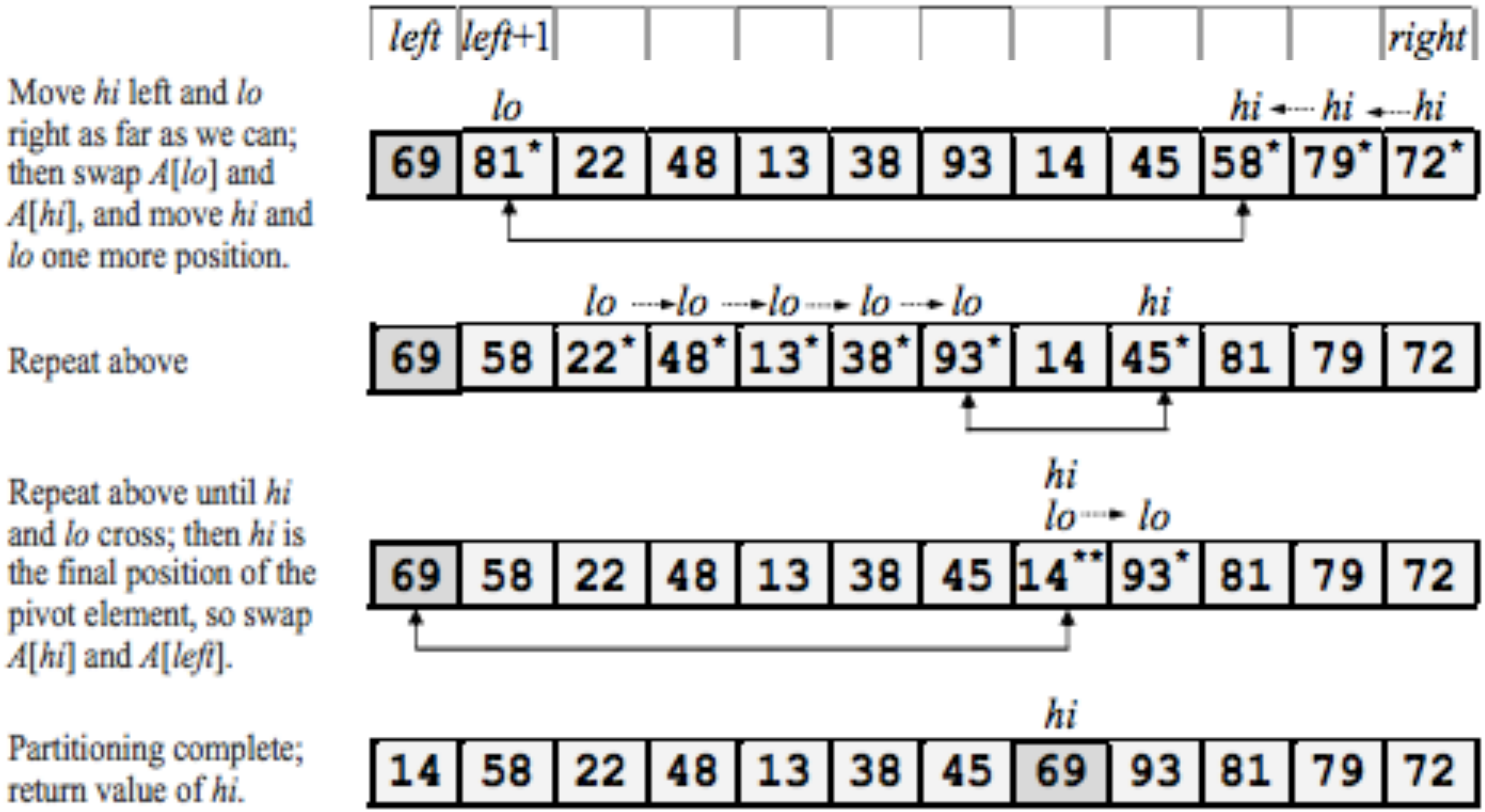
- Pseudocode for algorithm:

```
quicksort( A, left, right )
  if( left < right )
  {
    pivot = partition( A, left, right );
    quicksort( A, left, pivot-1 );
    quicksort( A, pivot+1, right );
  }
```

- Recall Merge sort: split helper method, combine
- Here: rearrange (via partition) then split


quicksort(A, 1, 12)
38 81 22 48 13 69 93 14 45 58 79 72
14 58 22 48 13 38 45 69 93 81 79 72
quicksort(A, 1, 7)
14 58 22 48 13 38 45
38 45 22 14 13 48 58
quicksort(A, 9, 12)
93 81 79 72
79 72 81 93
quicksort(A, 1, 5)
14 45 22 38 13
14 13 22 38 45
quicksort(A, 9, 10)
79 72
72 79
quicksort(A, 1, 2)
14 13
13 14
quicksort(A, 4, 5)
38 45
38 45

Inside One Partition Call



Exercise

56	13	1	99	65	12	27	8	44	83	2

Sort the following array using:

1. Quick sort

Let's start with pivot=56

Exercise

- Input:

56	13	1	99	65	12	27	8	44	83	2

- Quick Sort:

pivot=56:

56 | 13* 1* 99* 65 12 27 8 44 83 2*

56 | 13 1 2 65* 12 27 8 44* 83* 99

56 | 13 1 2 44 12* 27* 8** 65 83 99

8 13 1 2 44 12 27 **56** 65 83 99

pivot=8?

Exercise

- Quick Sort:

pivot=56: . . .

8 13 1 2 44 12 27 **<u>56</u>** 65 83 99

pivot=8:

8 | 13 1 2 44 12 27 **<u>56</u>** 65 83 99

8 | 13* 1 2* 44* 12* 27* (other half still the same)

8 | 2 1** 13 44 12 27

1 2 **<u>8</u>** 13 44 12 27

pivot=1: 1 | 2 (other half still the same)

Exercise

- Quick Sort:

pivot=56: . . .

8 13 1 2 44 12 27 **<u>56</u>** 65 83 99

pivot=8: . . .

1 2 **<u>8</u>** 13 44 12 27

pivot=1:

1 | 2 (other half still the same)

1 | 2**

1 2

pivot=13: 13 | 44 12 27 (first half still the same)

Exercise

- Quick Sort:

pivot=56: . . .

8 13 1 2 44 12 27 **<u>56</u>** 65 83 99

pivot=8: . . .

1 2 **<u>8</u>** 13 44 12 27

pivot=1: . . .

1 2

pivot=13:

13 | 44 12 27 (first half still the same)

13 | 44* 12* 27*

13 | 12** 44 27

12 **<u>13</u>** 44 27

pivot=44: 44 | 27** (first half still the same)

Exercise

- Quick Sort:

pivot=56: . . .

8 13 1 2 44 12 27 **<u>56</u>** 65 83 99

pivot=8: . . .

1 2 **<u>8</u>** 13 44 12 27

pivot=1: . . .

1 2

pivot=13: . . .

12 **<u>13</u>** 44 27

pivot=44: 44 | 27**

27 **<u>44</u>**

Exercise

- Quick Sort:

pivot=56: . . .

8 13 1 2 44 12 27 **<u>56</u>** 65 83 99

pivot=8: . . .

1 2 **<u>8</u>** 13 44 12 27

pivot=1: . . .

1 2

pivot=13: . . .

12 **<u>13</u>** 44 27

pivot=44: . . .

27 **<u>44</u>**

Exercise

- Array so far (left half of 56 sorted):

<u>1</u>	**<u>2</u>**	**<u>8</u>**	**<u>12</u>**	**<u>13</u>**	**<u>27</u>**	**<u>44</u>**	**<u>56</u>**	65	83	99

- Quick Sort:

pivot=56: . . .

8 13 1 2 44 12 27 **<u>56</u>** 65 83 99

pivot=65: 65 | 83* 99* (first half still the same)

65 83 99

- Output:

<u>1</u>	**<u>2</u>**	**<u>8</u>**	**<u>12</u>**	**<u>13</u>**	**<u>27</u>**	**<u>44</u>**	**<u>56</u>**	**<u>65</u>**	**<u>83</u>**	**<u>99</u>**

How to Choose the Pivot?

- First item?
- Middle item?
- Randomly select one each time the algorithm is run?
- Use the median of three points (the first, middle and last elements in the array)?

How to Choose the Pivot?

- First item?
- Middle item?
- Randomly select one each time the algorithm is run?
- Use the median of three points (the first, middle and last elements in the array)?
- For *any* given pivot choice, we can always come up with an input list of numbers that makes the algorithm perform poorly!

Comparison of Sorting Algorithms

- Merge sort algorithm
  - Uses a recursive call to split and merge, halving each time
  - Average and worst case of order N logN
- Quick Sort algorithm
  - Average of order N logN, worst case of order N^2
  - Can suffer if there's a bad combination of pivot and input list
  - Implementation gives faster performance in practice

Summary

- Slowing sorting algorithms
  - Selection sort
  - Insertion sort
- Classic search algorithms
  - Linear search
  - Binary search
- Improved sorting algorithms
  - Merge sort
  - Quick sort
- Additional concepts:
  - Time and space performance issues
  - In-place sorting
  - Time complexity analysis