Sorting

Chapter 8

Chapter Objectives

- To learn the use of the Java API standard sorting methods
- To learn various sorting algorithms:
  - Selection sort, bubble sort, insertion sort, Shell sort, merge sort, heapsort, and quicksort
- To understand the difference among the above sorting algorithms

The Java Sorting Methods

- Java API provides the class **Arrays** with several overloaded sort methods for different array types
- The **Collections** class provides similar sorting methods
- Sorting methods for arrays of primitive types are based on quicksort algorithm
- Method of sorting for arrays of objects and Lists based on mergesort

Generics Declaration

- ...public static <T> void foo(…something about E …)
  - The point here is that T is declared before the return class specification
- …<? super T>: any class that is a super class of T
- …<? extends T>: any class extends T
- It is possible to do something like:
  - ...public static <T> void foo(List<? extends T> lll, Comparator<? super T> ccc)
  - This means that the lll is a list of objects of class that extends T, including T itself, and that ccc is a comparator for the class that is a super class of T, including T itself.

Using Java Sorting Method (Arrays)

Method sort in Class Arrays	Behavior
public static void sort(int[] items)	Sorts items in ascending order
public static void sort(int[] items, int fromIndex, int toIndex)	Sorts items[fromIndex] to items[toIndex]
public static void sort(Object[] items)	Sorts the items in ascending order according to the **compareTo()** method. All objects in the array are expected to implement the **Comparable interface** and must be mutually comparable
public static void sort(Object[] item, int fromIndex, int toIndex)	An index-range-specific version of the above.
public static <T> void sort(T[] items, Comparator<? super T> comp)	Sorts the items in ascending order according to the **comp.compare** method, which is defined for a class extending T (a wildcard). All objects in the array must implement the **Comparable interface** and must be mutually comparable
public static <T> void sort(T[] items, int fromIndex, int toIndex, Comparator<? super T> comp)	An index-range-specific version of the above.

Using Java Sorting Method (Lists)

Method sort in Class Arrays	**Behavior**
public static <T extends Comparable<T>> void sort (List<T> list)	Sorts the items in list ascending order according to their natural ordering as defined in the **compareTo()** method. All objects in the list must implement the **Comparable interface** and must be mutually comparable
public static <T> void sort (List<T> list, Comparator<? super T> comp)	Sorts the items in list ascending order according to the **comp.compare** method, which is defined for a class extending T (a wildcard). All objects in the list must be comparable by the comparator.

Measuring the Complexity of Sorting Algorithms

- Use the number of comparisons of items in the array.

Selection Sort

- Selection sort
  - Keep finding the “next” smallest item
  - Place the item found in the appropriate position
- Efficiency is $O(n^2)$

Selection Sort Algorithm

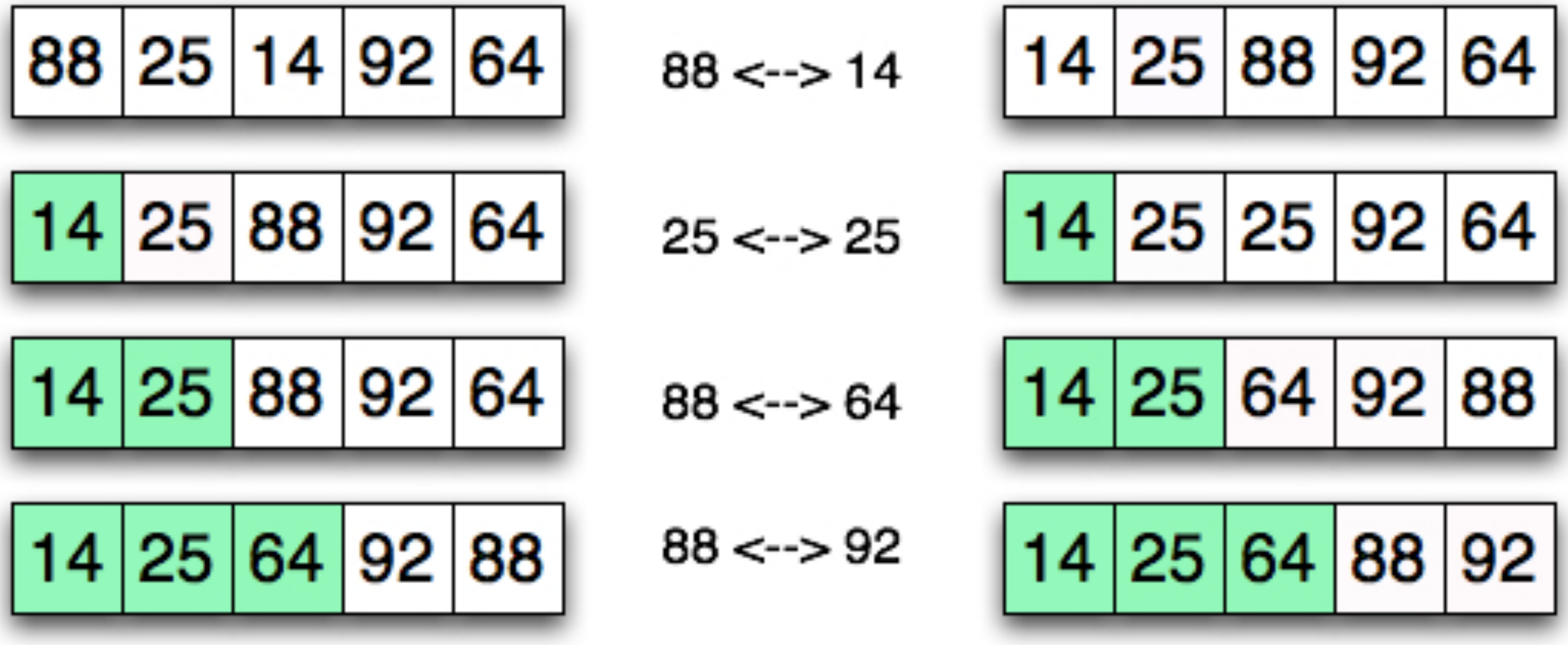
```
for fill = 0 to n-2 do
    pos = the index of the smallest item in the range [fill,n-1]
    exchange the element at index pos and the element at index fill
```

… the slot at fill can be used to maintain the smallest value
… this yields to

```
for fill = 0 to n-2 do
    for next = fill + 1 to n-1 do
        if the element at next is smaller than the element at pos
        then exchange the two
```

The number of comparisons is $O(n^2)$, so is the number of exchanges.

Selection Sort Example



Bubble Sort

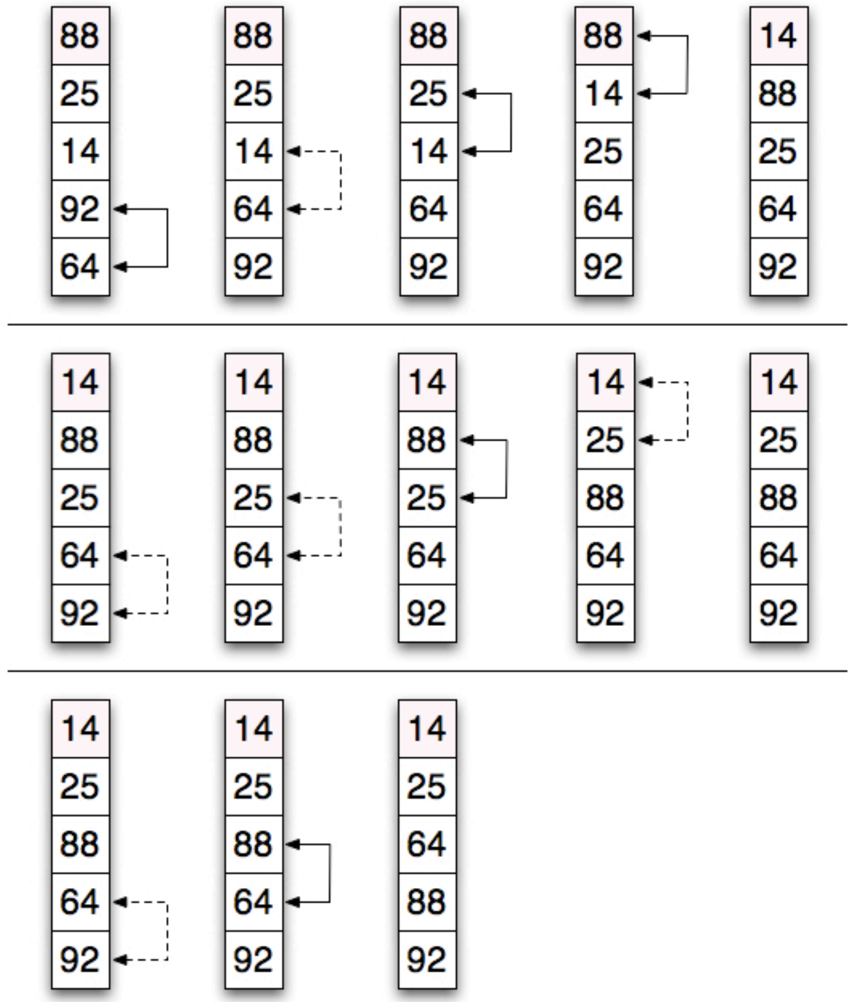
- Compares adjacent array elements and exchanges their values if they are out of order
- Smaller values bubble up to the top of the array and larger values sink to the bottom

```
do
    for each pair of adjacent array elements
        if the values in the pair are out of order
        then exchange the two
while the array is not sorted
```

The last condition can be checked by remembering whether any exchange has occurred during the execution of the for-loop.

The number of comparisons is $O(n^2)$.

Bubble Sort Example



Analysis of Bubble Sort

- Provides excellent performance in some cases and very poor performances in some cases
- Works best when array is already nearly sorted
- Worst case number of comparisons is $O(n^2)$
- Worst case number of exchanges is $O(n^2)$
- Best case occurs when the array is already sorted
    - $O(n)$ comparisons
    - $O(1)$ exchanges

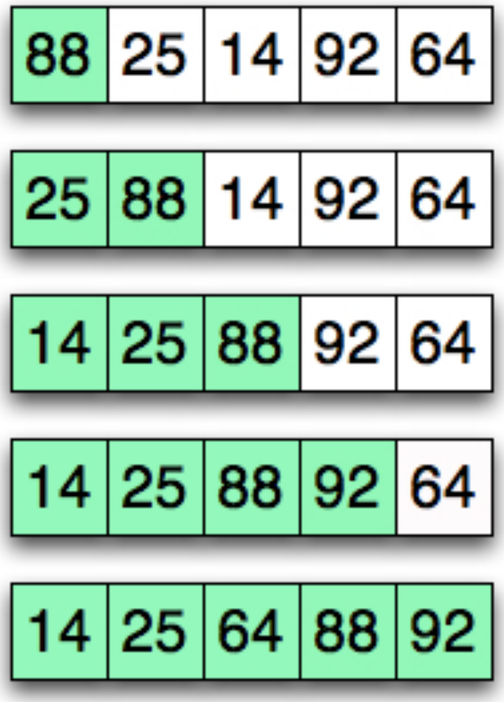
Insertion Sort

- Sort the elements in range [0,m] for m = 0,…, n−1
- No action need for m=0
- When going from m to m+1, insert the element in index m+1, to its appropriate location

```
For m = 1 to n−1 do
    // determine the position for the element at index m
    pos = m−1
    item = element at position m
    while (pos >= 0) and (the element at index pos is greater than item)
        place the element at index pos to index pos+1
        decrement pos
    place item at index pos+1
```

The number of comparisons is $O(n^2)$.

Insertion Sort Example



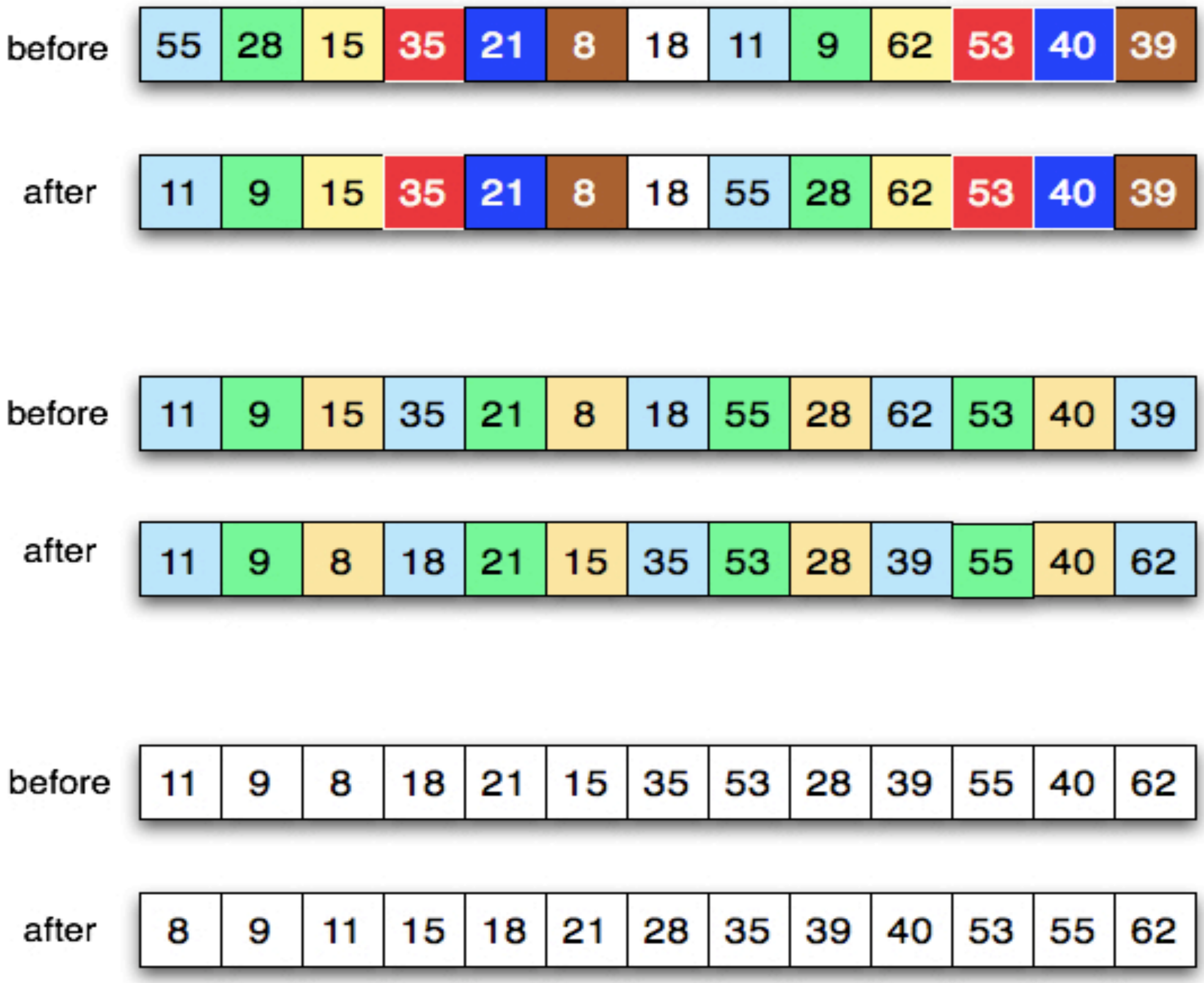
Analysis of Insertion Sort

- Maximum number of comparisons is $O(n^2)$
- In the best case, number of comparisons is $O(n)$
- The number of shifts performed during an insertion is one less than the number of comparisons or, when the new value is the smallest so far, the same as the number of comparisons
- A shift in an insertion sort requires the movement of only one item whereas in a bubble or selection sort an exchange involves a temporary item and requires the movement of three items

Shell Sort: A Divide-and-Conquer Insertion Sort

- Shell sort is a type of insertion sort but with $O(n^{1.5})$ or better performance.
- For p in a decreasing series $\{q_1, q_2, \ldots, q_k\}$ of gap values, execute Insertion Sort on the n/p subarrays consisting of the elements at every p-th position.
  - Choose q_k=1 to guarantee correctness.
  - Use gap values of the form 2^k-1, for $1 \leq k \leq ceil(\log_2(n+1))$

Shell Sort Example with Gaps of 7, 3, 1



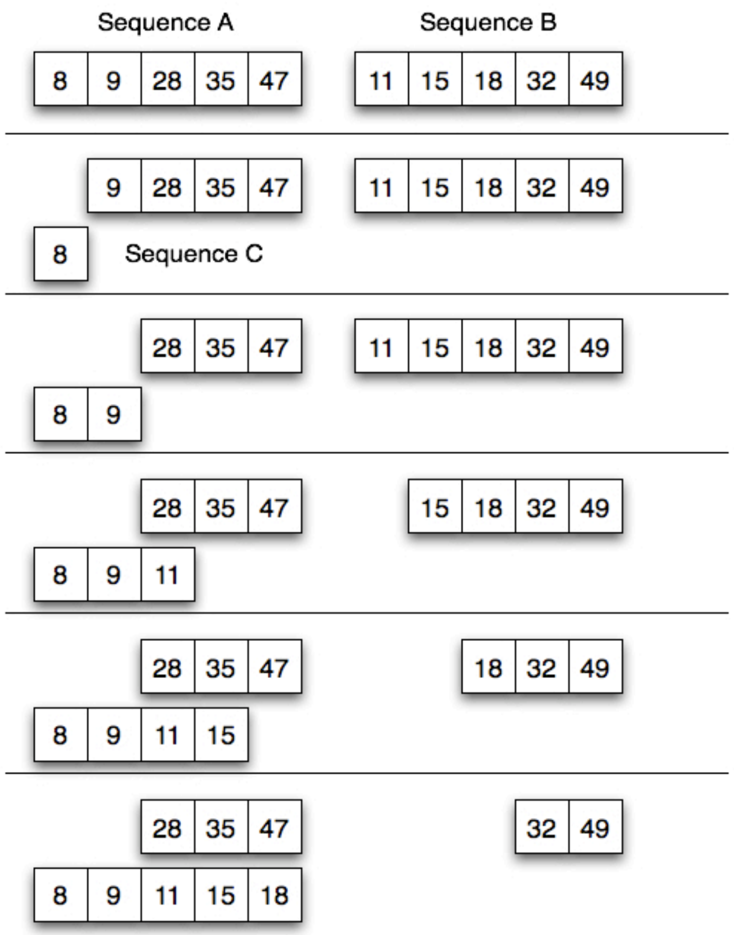
before	55	28	15	35	21	8	18	11	9	62	53	40	39
after	11	9	15	35	21	8	18	55	28	62	53	40	39
before	11	9	15	35	21	8	18	55	28	62	53	40	39
after	11	9	8	18	21	15	35	53	28	39	55	40	62
before	11	9	8	18	21	15	35	53	28	39	55	40	62
after	8	9	11	15	18	21	28	35	39	40	53	55	62

Merge Sort

Sort by merging two already-sorted sequences, itemsA and itemsB of length lenA and lenB, respectively, into a new sequence, itemsC

```
Set the indices posA, posB, posC to 0
while (posA<lenA) and (posB<lenB) do
    if (itemsA[posA]<itemsB[posB]) {
            itemsC[posC] = itemsA[posA];
            posA = posA + 1;
    }
    else {
            itemsC[posC] = itemsB[posB];
            posB = posB + 1;
    }
    posC = posC + 1;
if (posA<lenA-1) {
    append all the remaining elements of itemsA to itemsC;
}
else if (posB<lenB-1) {
    append all the remaining elements of itemsB to itemsC;
}
```

Merge Example



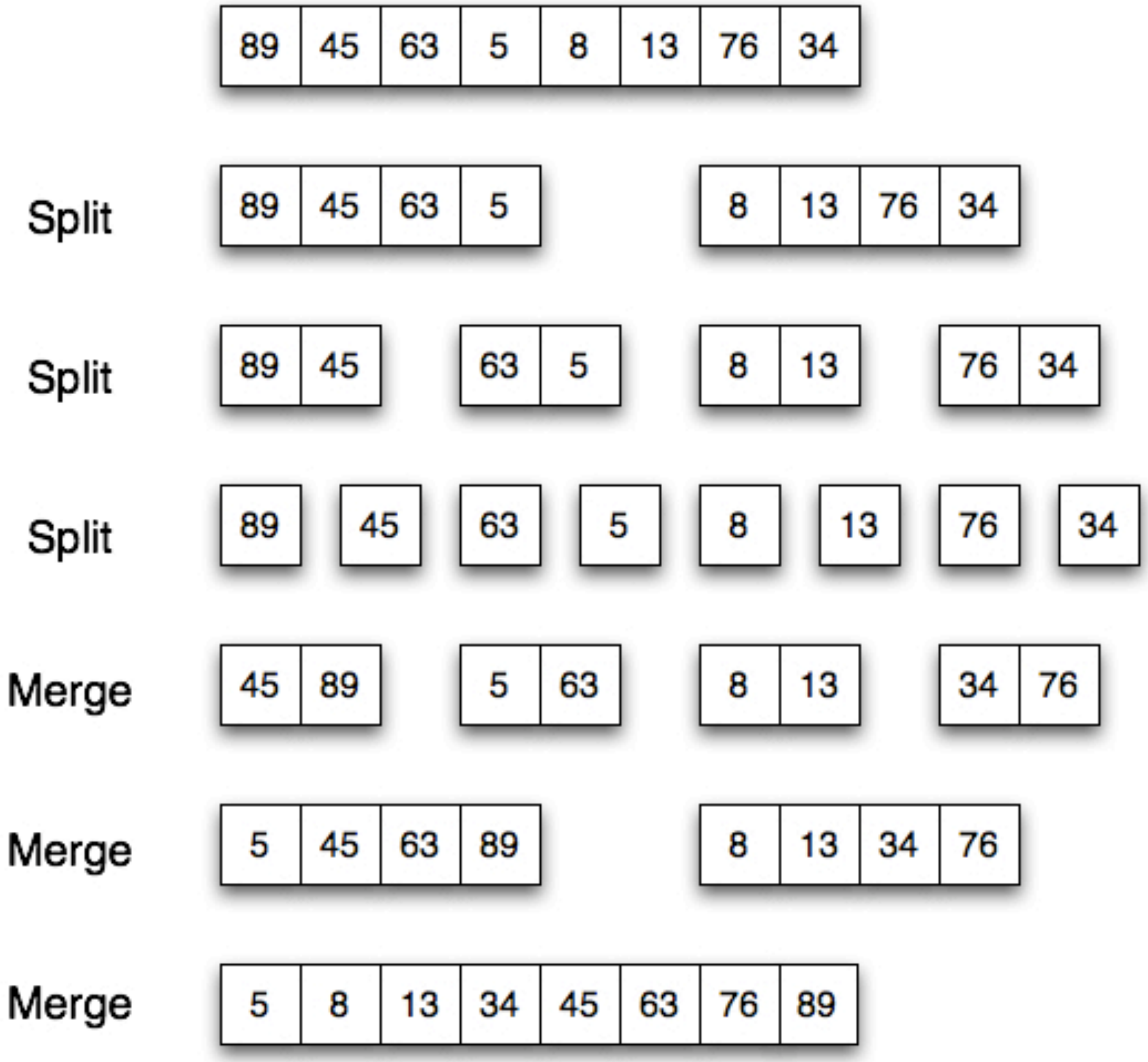
Analysis of Merge

- For two input sequences that contain a total of n elements, we need to move each element's input sequence to its output sequence
- The number of comparisons is not more than the total number of elements.
- So, the time for merging is O(n).

Merge Sort

- If an input sequence has size > 1, divide it into left and right halves
- Use Merge Sort to sort the halves
- Merge the two sorted halves.

- To sort each half, create its copy and make a recursive call to Merge Sort.
- To merge, use the two half-size sequences returned by the recursive calls



89 45 63 5 8 13 76 34
Split
89 45 63 5
8 13 76 34
Split
89 45
63 5
8 13
76 34
Split
89
45
63
5
8
13
76
34
Merge
45 89
5 63
8 13
34 76
Merge
5 45 63 89
8 13 34 76
Merge
5 8 13 34 45 63 76 89

Analysis of Merge Sort

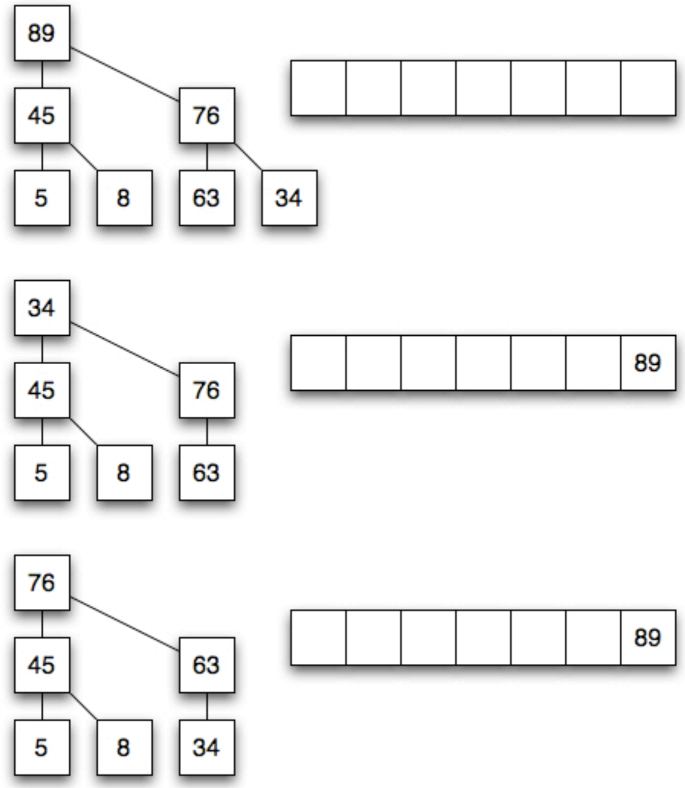
- Assume the length of the sequence is a power of 2, say 2^m.
  - We can assume that infinitely large elements exist beyond the sequence to make its length a power of 2; the length will then be at most 2 times the length of the original.
- Let f(m)=time for merge-sorting a sequence of size 2^m.
- We have $f(m) = 2*f(m-1) + c*2^m$ for some constant c.
- We then have:
  - $f(m) = 4*f(m-2)+2c*2^{m-1} = 8*f(m-3)+3c*2^{m-1} = \ldots$
    $= 2^m*f(0)+mc*2^m$. Thus, f(m) is $O(m*2^m)$.
- This implies that Merge Sort has efficient of O(nlog n).

Heapsort

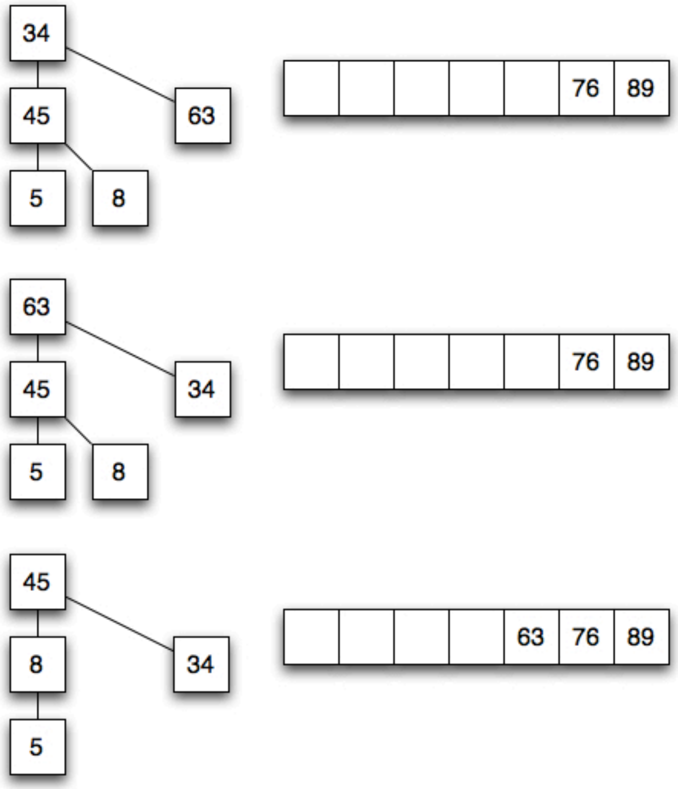
- Use a property of the heap that the largest element is at the root.
- Given n elements, turn them into a heap of size n (implementable using an array of size n).
- For p = n, …, 2 do the following:
  - Exchange the element at position 0 (the root) and the element at position p.
  - Enforce the heap property starting from position 0.
  - (In the next round p is decremented, so the above achieves the removal of the root.)

Heap Sort

Initialization of heap and extraction of an element



Extraction of the next two elements



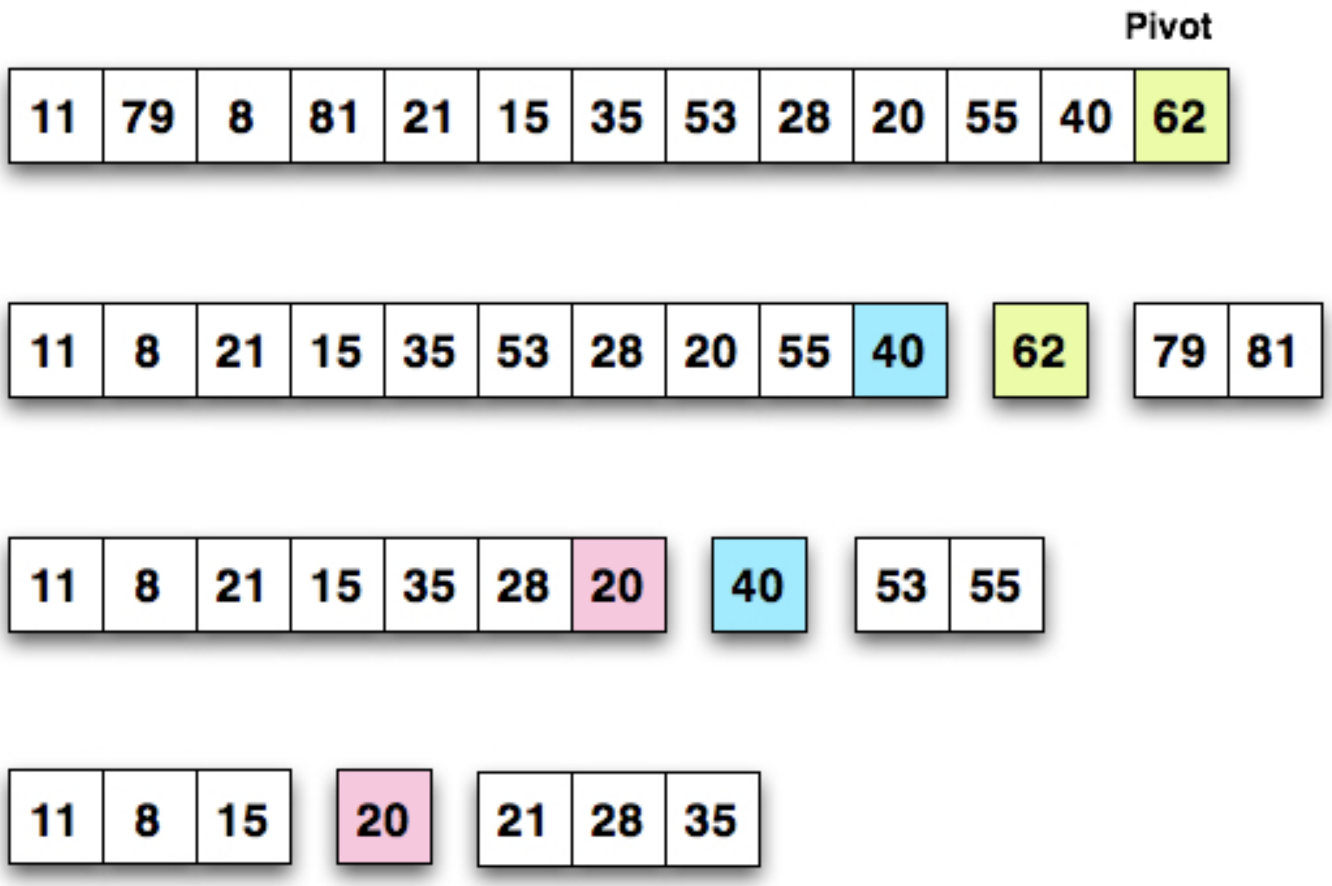
Heapsort Analysis

- Both insertion in the heap and the restructuring after removal for maintaining the heap property require time O (log n).
- Since both occur n times, the total time required for these is O(n long).
- The time required for other operations is O(n).
- Thus, Heapsort has the running time of O(nlog n).

Quicksort

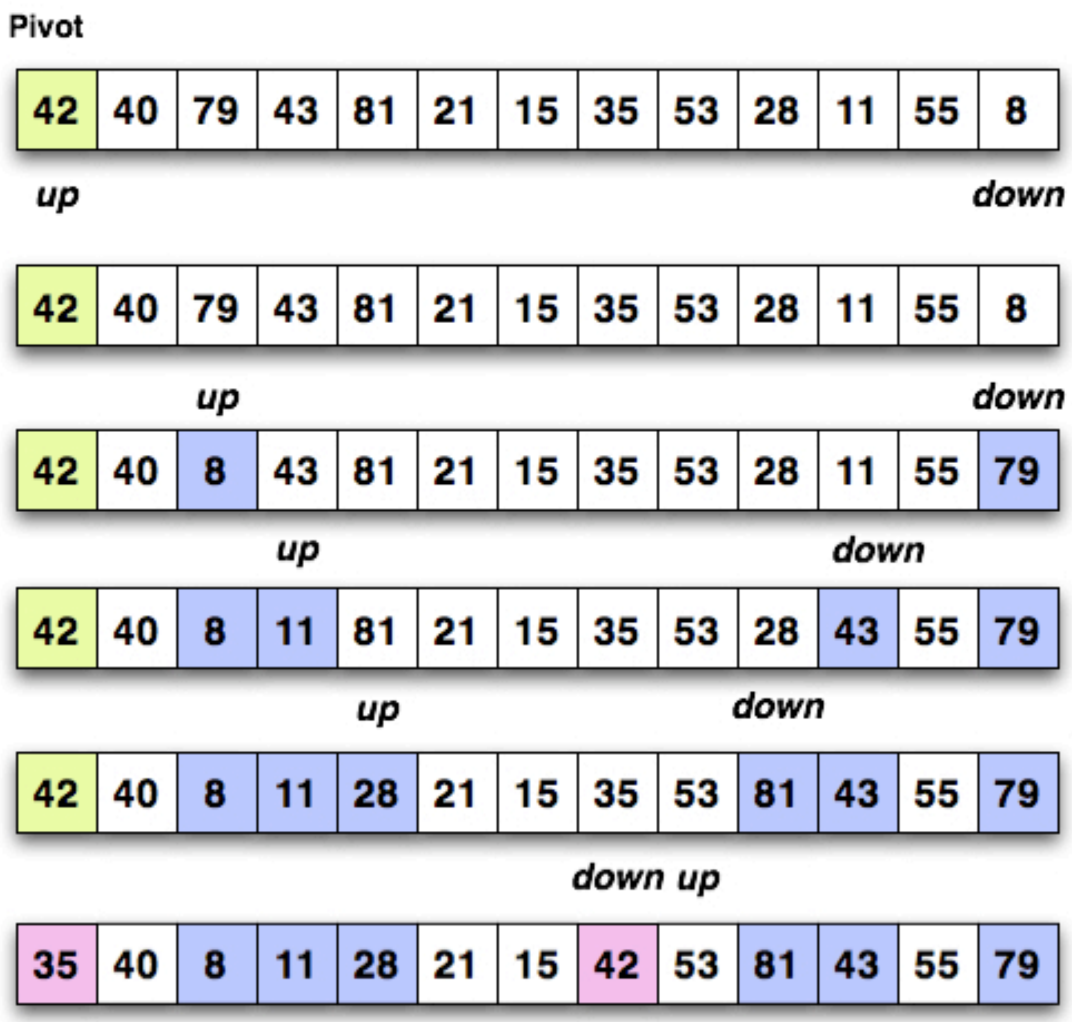
- We’ll assume that the elements are in an array.
- Select an element, called pivot, from the array and reorder the array so that:
  - Those smaller than the pivot come first (in any order)
  - Those larger than the pivot comes last
- Recursively sort the two subparts using Quicksort and then join them.
- Average case for Quicksort is O(n log n)

Idealized Split in Quicksort (the ordering is preserved here for the sake of clarity in presentation)



Algorithm for Partitioning Array *items* Between Index *P* and Index Q

```
Set pivot to items[P]
Set up to P and down to Q
do
    while (items[up] ≤ pivot) and (up < Q) { up++ }
    while (items[down] ≥ pivot) and (down > P) { down−− }
    if (up < down) { swap items[up] and items[down] }
while (up < down);
swap items[P] and items[down]
```


Pivot
42 40 79 43 81 21 15 35 53 28 11 55 8
up
down
42 40 79 43 81 21 15 35 53 28 11 55 8
up
down
42 40 8 43 81 21 15 35 53 28 11 55 79
up
down
42 40 8 11 81 21 15 35 53 28 43 55 79
up
down
42 40 8 11 28 21 15 35 53 81 43 55 79
down up
35 40 8 11 28 21 15 42 53 81 43 55 79

Practical Partition Algorithm

- Quicksort is $O(n^2)$ when each split yields one empty subarray, which is the case when the array is presorted
- This can be avoided by selecting as the pivot value that is “less likely to lead to a bad split”
  - One such a choice is the median of the first, middle, and last elements
- Another solution is to randomly select a position between *P* and *Q* and use the element at the position as the pivot
  - *On average, the time is O(n log(n))*

Testing the Sort Algorithms

- Need to use a variety of test cases
  - Small and large arrays
  - Arrays in random order
  - Arrays that are already sorted
  - Arrays with duplicate values
- Compare performance on each type of array