ENSF 593/594
Data Structures – Sorting

Mohammad Moshirpour

Outline

- Sorting Terminology
- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge sort
- Quick sort

Goal

- In this lecture we are going to know about three simple sorting algorithms how they operate and how efficient they are.
- Then, we will study some sophisticated sort algorithms that have complexities better than $O(n^2)$

It Is Important to Know About Sorting Algorithms, Even If You Are President of USA

https://www.youtube.com/watch?v=k4RRi_ntQc8

Terminology

- ***Internal sort:*** data is kept in primary memory
  - i.e. In RAM, using arrays
- ***External sort:*** data is kept in secondary storage
  - i.e. On disk or tape
  - Usually requires special sorting techniques

Terminology (Cont’d)

- ***In-place sort:*** a sort achieved by exchanging items “in place” within an array
  - i.e. Does not use extra memory
  - Some sorts use extra memory for speed reasons
- ***Stable sort:*** a sort that preserves the relative order of equal keys
  - E.g. Sort a list of people first by name, then by age
    - A stable sort ensures that people of the same age remain in alphabetic order

Analyzing Sorts

- Is usually done by considering the number of:
  - Comparisons
  - Data movements
    - i.e. exchanges or “swaps”
- Usually characterized with big-O notation

Analyzing Sorts (Cont’d)

- The efficiency of a sort may depend on the initial ordering of data
- We measure the number of comparisons and data movements for the:
  - Best case: data already sorted
  - Worst case: data in reverse order
  - Average case: data in random order

Analyzing Sorts (Cont’d)

- The number of comparisons and data movements may not coincide
  - If comparisons are expensive, we prefer sorts that minimize the number of comparisons
    - E.g. Comparing strings takes longer than comparing integers
  - If data items are large, we prefer sorts that minimize data movements
    - E.g. Moving large structs is expensive; moving external data even more so
- Sometimes simple, inefficient sorts are OK for small data sets
  - Are easy to implement and debug
  - Running times will not be much worse than for more elaborate sorts

Sorting Algorithms

- Simple sorting algorithms
  - Bubble sort (also known as an exchange sort)
  - Selection sort
  - Insertion sort
- Complex sorting algorithms
  - Merge sort
  - Quick sort

Bubble Sort

- Comparing successive pairs of items, swapping them if out of order
  - The smallest item “bubbles up” to the top (beginning) of the array on the first pass
  - The next smallest item bubbles up to its proper spot on the second pass
  - This is repeated until done
    - There are $n - 1$ passes

Bubble Sort (Cont'd)

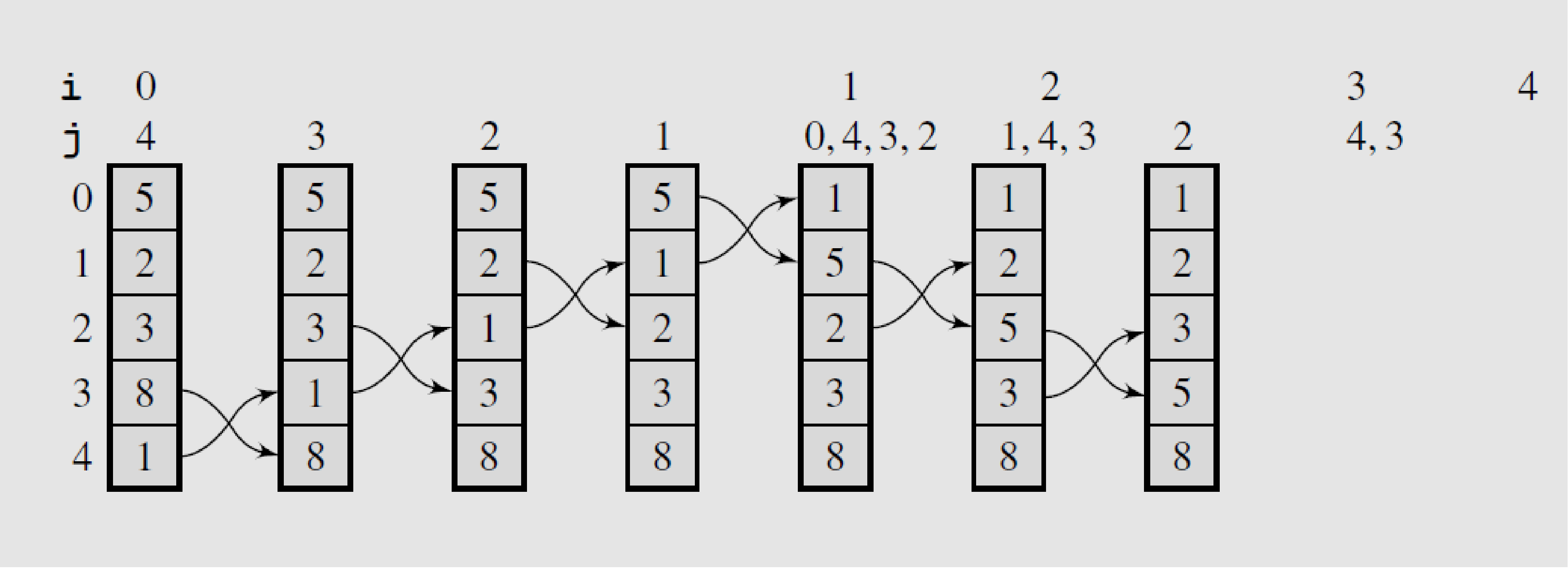
- Implementation:

```
void bubbleSort(int[] arr)
{
    for (int i = 0; i < arr.length-1; i++){
        for (int j = arr.length-1; j > i; j--){
            if (arr[j] < arr[j-1]){    //Compare items
                int temp= arr[j-1];    //Create a temporary integer to store value 1
                arr[j-1]= arr[j];      // Swap value 1 to value 2
                arr[j]=temp;           // Swap value 2 to value 1
            }
        }
    }
}
```

Bubble Sort (Cont'd)

- Sample run:

The array [5 2 3 8 1] sorted by bubble sort.



Bubble Sort (Cont'd)

- The number of comparisons is the same for the best, average, and worst cases:

$$[n(n-1)] / 2 = O(n^2)$$

  - Does not depend on input order
- Number of swaps:
  - Worst case (reverse order): same as for number of comparisons
  - Best case (already sorted): no swaps
  - Average case (random order): $[n(n-1)]/4 = O(n^2)$

Selection Sort

- Works by selecting the smallest item above the current item in the array, then swapping them
  - This is repeated for each item of the array, up to the second-last item
  - After each pass, the low part of the array is sorted, and is no longer considered

Selection Sort (Cont'd)

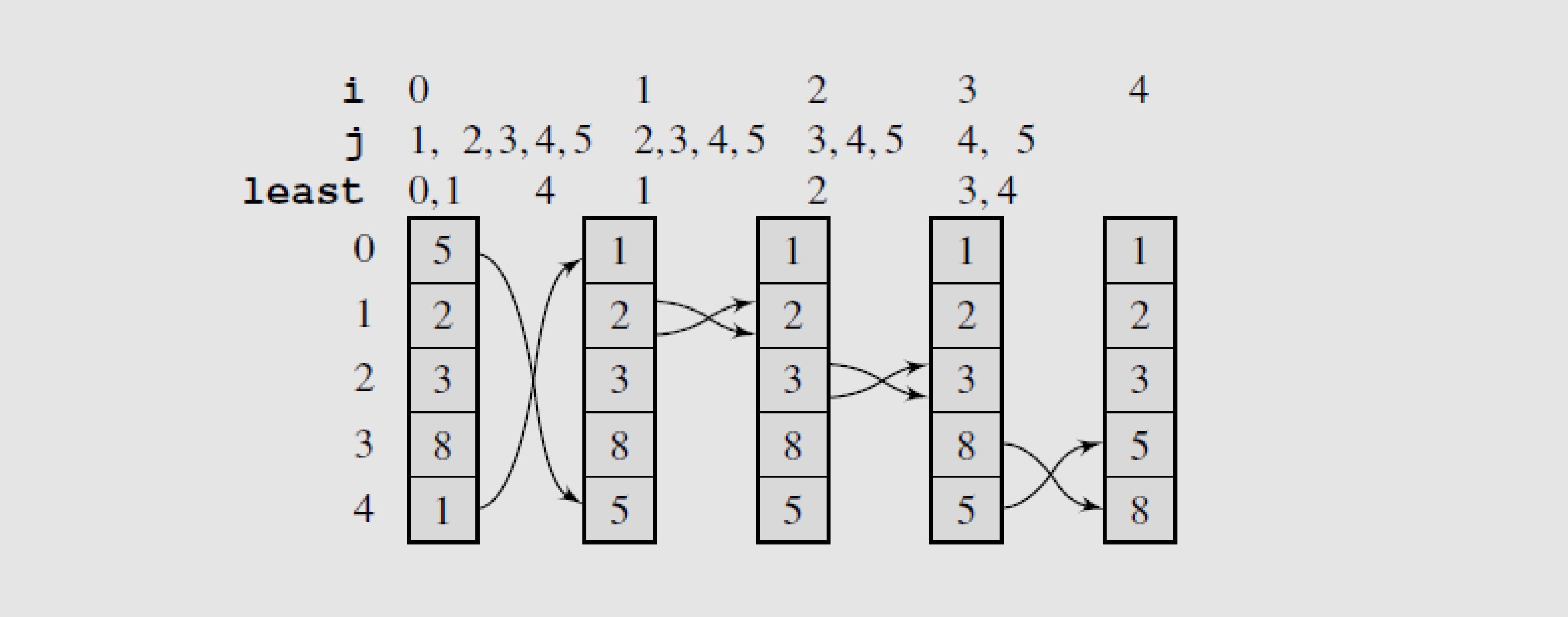
- Implementation:

```
void selectionSort(int[] arr)
{
    for (int i = 0; i < arr.length-1; i++){
        // Find the least element in right subarray
        int min  = i;
        for (int j = i + 1; j < arr.length; j++){
            if (arr[j] < arr[min])
                min  = j;
        }
        //Swap items
        int temp= arr[min];
        arr[min]= arr[i];
        arr[i]=temp;
    }
}
```

Selection Sort (Cont'd)

- Sample run:

The array [5 2 3 8 1] sorted by selection sort.



Selection Sort (Cont'd)

- The number of comparisons is the same for the best, average, and worst cases:

$$[n(n-1)] / 2 = O(n^2)$$

- Does not depend on input order
- The number of swaps is $n - 1$, which is $O(n)$
- Is an $O(n^2)$ sort regardless of input order

Insertion Sort

- Is like sorting a hand of playing cards
- Start with the 2nd item, and compare it with the first
  - If less, move the 1st to the right and insert the 2nd
- Repeat with each successive item, inserting it into its proper position in the left subarray
  - Must move all items greater than the item one position to the right

Insertion Sort (Cont'd)

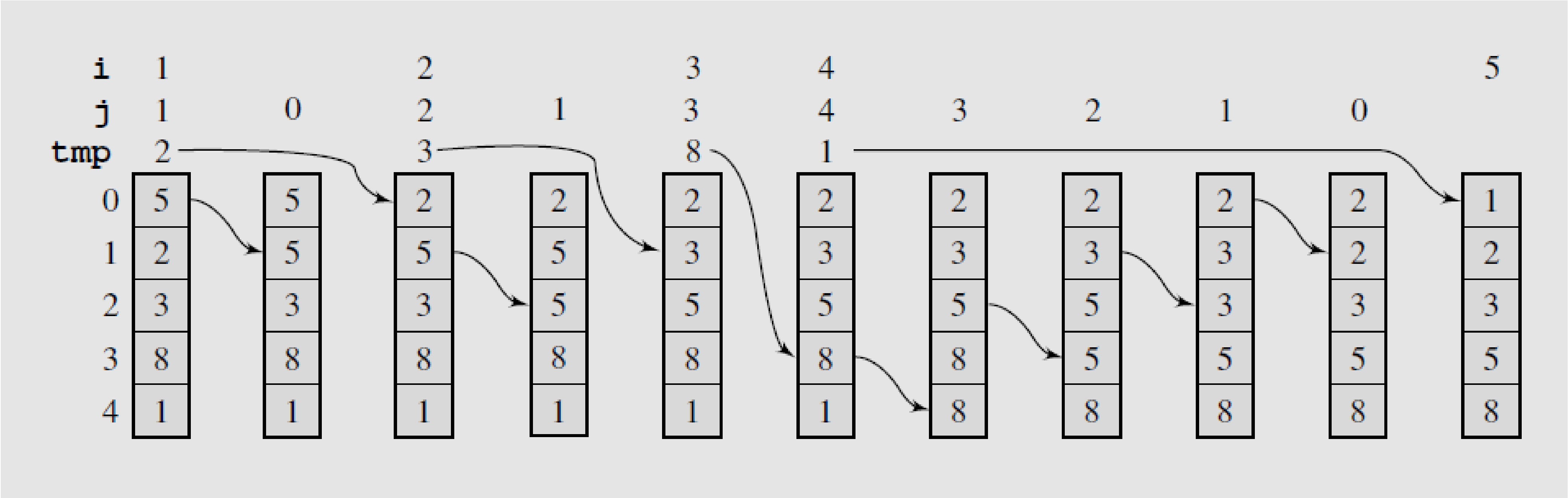
- Implementation:

```
void insertionSort(int[] arr)
{
    for (int i = 1, j; i < arr.length; i++){
        int temp = arr[i];
        for (j = i; j > 0 && temp < arr[j-1]; j--)
            arr[j] = arr[j-1];
        arr[j] = temp;
    }
}
```

Insertion Sort (Cont’d)

- Sample run:

The array [5 2 3 8 1] sorted by insertion sort.



Insertion Sort (Cont'd)

- **Best case (already sorted):**
  - Comparisons: $n-1 = O(n)$
  - Data moves: $2(n-1) = O(n)$
- **Worst case (reverse order):**
  - Comparisons: $[n(n-1)]/2 = O(n^2)$
  - Data moves: $[n(n-1)]/2 + 2(n-1) =$
    $[n^2 + 3n - 4]/2 =$
    $O(n^2)$

Insertion Sort (Cont'd)

- Average case (random order):
  - Comparisons: $[n^2 + n - 2]/4 = O(n^2)$
  - Data moves: $[n^2 + 5n - 6]/4 = O(n^2)$
- Is an $O(n^2)$ sort in the average and worst cases
  - But performs well if the input data is nearly-sorted order (approaches $O(n)$)

Ideal Performance of Sorts

- What is the theoretical best running time we can achieve for a comparison-based sort?
  - *O(n lg n)* for the worst and average cases
  - Justification: using a decision tree analysis, we find the required number of comparisons for an ideal sort is *lg(n!)* which is *O(n lg n)*
- Thus, we can devise better sorts than those already presented

Merge Sort

- **Basic idea:**
  - Divide the array into two equal-size sub-arrays
  - Sort each sub-array
    - Done by applying the merge sort recursively
  - Merge the sub-arrays into a temporary array
  - Copy the temporary array back into the original array

Merge Sort (Cont'd)

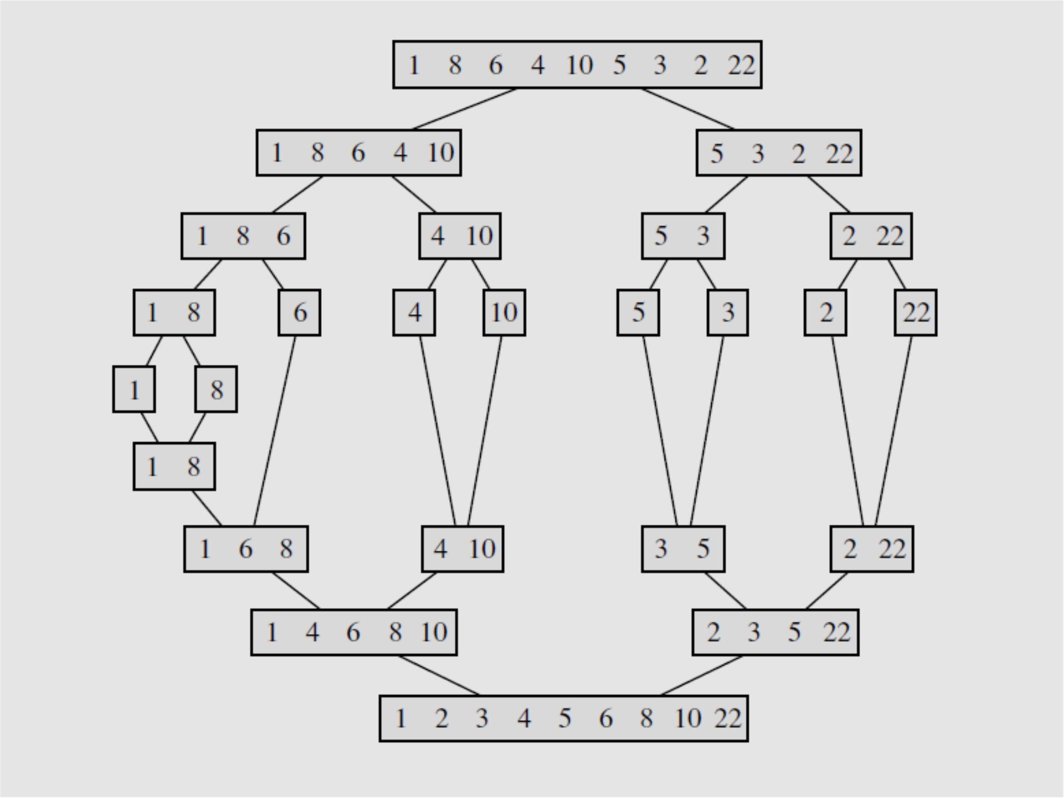
- Implementation:

```
void mergeSort(int[] arr, int first, int last){
     if(first < last) //Checks whether there is more than 1 element.
     {
        int mid = (first + last) / 2;
        mergeSort(arr, first, mid); //Recursively sort the first half of the array
        mergeSort(arr, mid+1, last); //Recursively sort the second half of the array
        merge(arr, first, mid, mid+1, last);
     }
 }
```

Merge Sort (Cont’d)

- Sample run:

The array [1 8 6 4 10 5 3 2 22] sorted by mergesort.



Merge Sort (Cont'd)

- The number of comparisons and data moves is *O(n lg n)* in the best, worst, and average cases
  - Is insensitive to initial order of data
- The need for a temporary array (extra space) is a disadvantage

Quick Sort

- **Basic idea:**
- Choose one array element to be the *pivot* (or *bound*)
- Partition the array into 2 subarrays, such that:
  - Subarray1 contains only elements ≤ pivot
  - The pivot is in its final position in the array
  - Subarray2 contains only elements ≥ pivot

≤ pivot	pivot	≥ pivot
subarray1		subarray2

Quick Sort (Cont’d)

- Apply this recursively to each subarray
  - Stop when the subarray is ≤ 1 in length
- Try to choose pivots that divide the array into (nearly) equal halves
  - Some possible approaches:
    - Pick the first array element
      - Fares poorly when the array us in (nearly) sorted order
    - Pick the middle element
    - Pick the median of the first, middle, and last elements
    - Pick an element randomly

Quick Sort (Cont’d)

- To partition the array:
  - Scan the array inward from the edges, using two pointers (indices)
    - Stop the left pointer when it reaches an element > pivot
    - Stop the right pointer when it reaches an element < pivot
  - Exchange the two elements
  - Repeat until the pointers cross

Quick Sort (Cont’d)

- During partitioning, the pivot is sometimes moved out of the way by exchanging with the first element
  - Then is moved back to its final position with another exchange
- To avoid index bound checks, the largest element can be put into the last array position
  - Done before the actual sort

Quick Sort (Cont'd)

- Implementation:

```
void quickSort(Object[] data, int first, int last) {
          int lower = first + 1, upper = last;
          swap(data,first,(first+last)/2);
          Comparable bound = (Comparable)data[first];
          while (lower <= upper) {
              while bound.compareTo(data[upper])
                    lower++;
              while (bound.compareTo(data[upper])> 0)
                    upper--;
              if (lower < upper)
                   swap(data,lower++,upper--);
              else lower++;
          }
          swap(data,upper,first);
          if (first < upper-1)
              quickSort(data,first,upper-1);
          if (upper+1 < last)
              quickSort(data,upper+1,last);
}
```

Quick Sort (Cont'd)

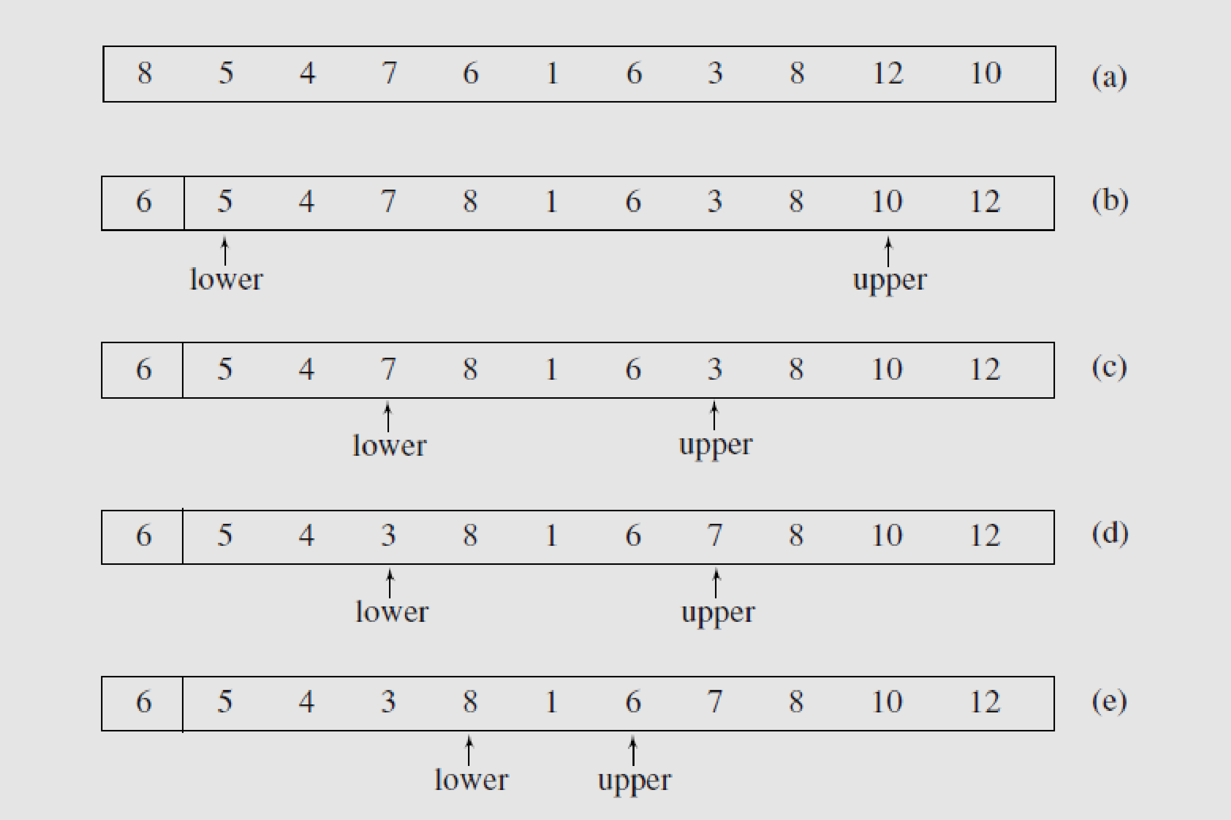
- Implementation:

```
void quickSort(Object[] data) {
        if (data.length < 2)
            return;
        int max = 0;
        // find the largest element and put it at the end of data;
        for (int i = 1; i < data.length; i++)
            if (((Comparable)data[max]).compareTo(data[i]) < 0)
                max = i;
        swap(data,data.length-1,max); // largest el is now in its
        quickSort(data,0,data.length-2); // final position;
}
```

Quick Sort (Cont'd)

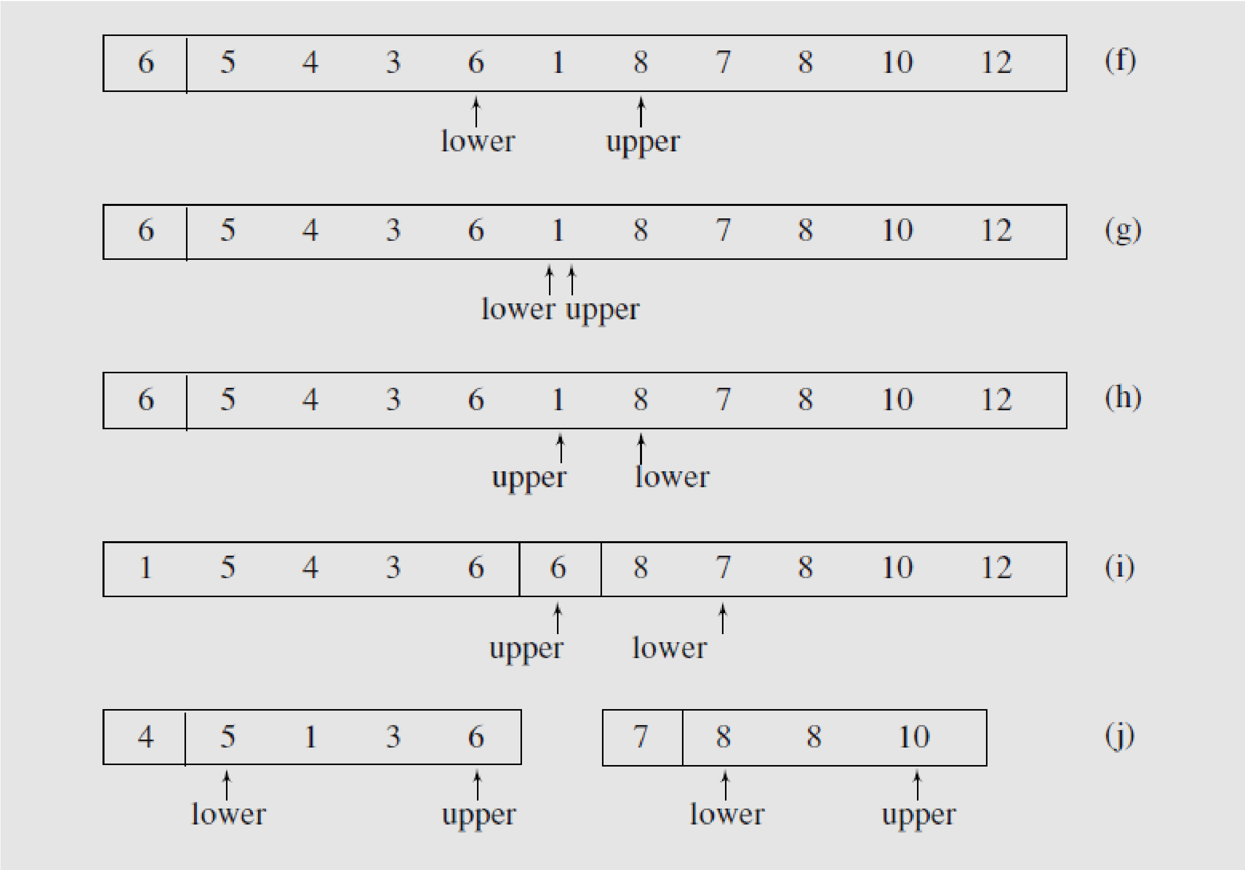
- Sample run:

Partitioning the array [8 5 4 7 6 1 6 3 8 12 10] with `quicksort()`.



Quick Sort (Cont'd)

- Sample run:



Quick Sort (Cont’d)

- The worst case occurs when the largest or smallest element is always chosen for the pivot
  - Results in a subarray of size 0, and another of size *n-1*
  - Is $O(n^2)$
- The best case occurs when the pivots always create equal-sized subarrays
  - Results in a tree with *lg n* levels
  - Is $O(n \lg n)$

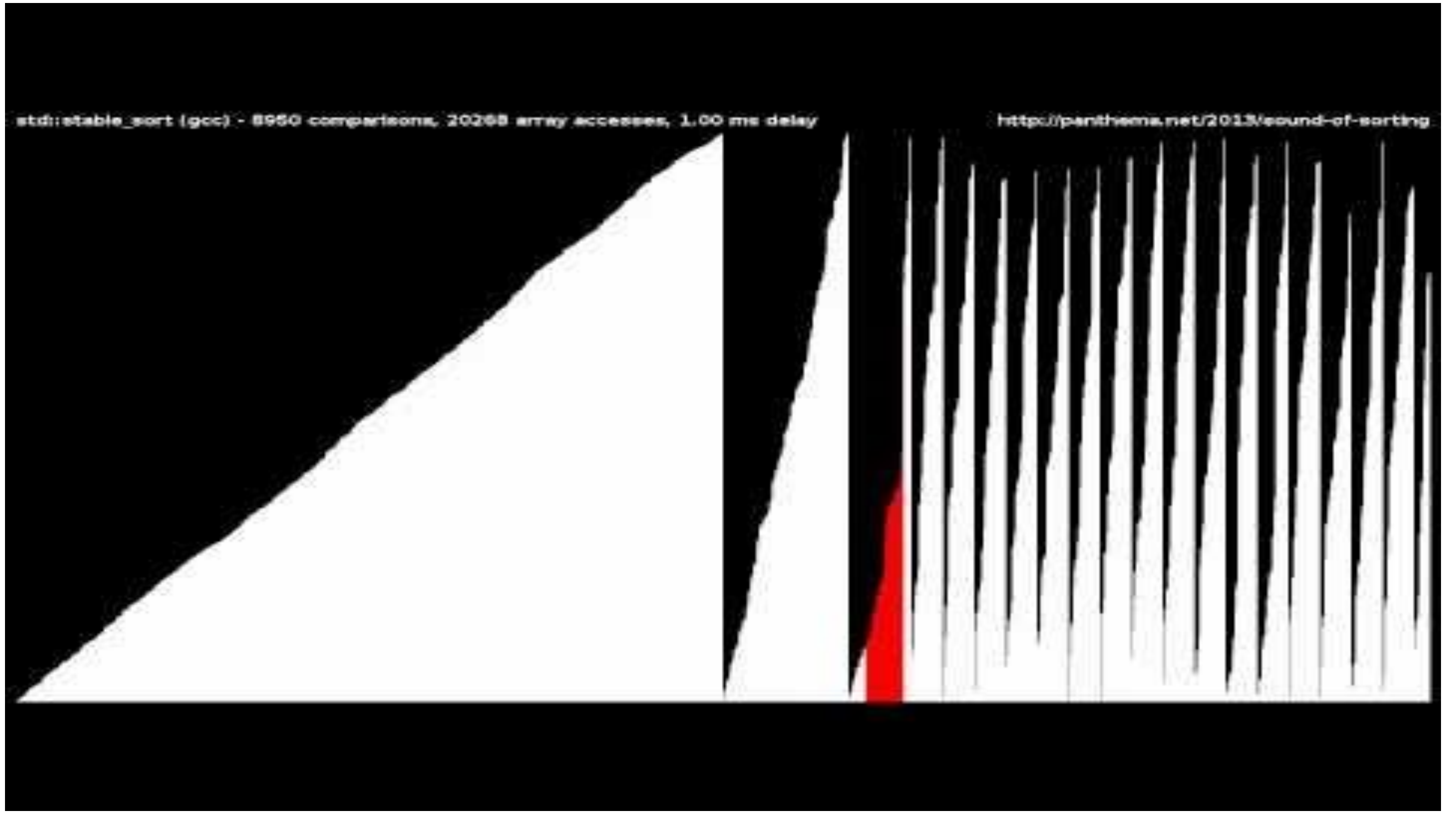
Quick Sort (Cont'd)

- The worst case occurs when the largest or smallest element is always chosen for the pivot
  - Results in a subarray of size 0, and another of size $n-1$
  - Is $O(n^2)$
- The best case occurs when the pivots always create equal-sized subarrays
  - Results in a tree with $lg\ n$ levels
  - Is $O(n\ lg\ n)$

Quick Sort (Cont'd)

- In the average case (random initial order), is $O(n \lg n)$
- Is normally the quickest algorithm
  - But may be $O(n^2)$ on some inputs (rare)
  - Unlike merge sort, does not need extra memory
    - i.e. is an in-place sort
  - Is not appropriate for array < about 30 items
    - Use a simple sort like insertion sort

15 Sorting Algorithms in 6 Minutes



https://www.youtube.com/watch?v=kPRA0W1kECg

Summary

- *Sequential search* is the simplest search to implement but it’s not efficient on big number of data.
- *Binary search* is a more efficient algorithm than sequential search but it needs that data be sorted.
- *Interpolation search* is a variant of binary search. It tries to find a better position to divide data and then search among them.

Summary (Cont'd)

- **Bubble sort:**
  - Works by comparing successive pairs of items, swapping them if out of order.
  - Is $O(n^2)$ in all the cases.
- **Selection sort:**
  - Works by selecting the smallest item above the current item in the array, then swapping them.
  - Is an $O(n^2)$ sort, regardless of input order.
- **Insertion sort:**
  - Is like sorting a hand of playing cards. It start with the 2nd item, and compare it with the first. If less, move the 1st to the right and insert the 2nd.
  - Is an $O(n^2)$ sort in the average and worst cases, but performs well if the input data is in nearly-sorted order (approaches $O(n)$).

Summary (Cont'd)

- Theoretical best running time we can achieve for a comparison-based sort is *O(n lg n)* for the worst and average cases.
- **Merge Sort:**
  - Divide the array into two equal-size sub-arrays
  - Sort each sub-array
  - Merge the sub-arrays into a temporary array
- **Quick Sort:**
  - Choose one array element to be the pivot (or bound)
  - Partition the array into 2 subarrays, such that:
    - Subarray1 contains only elements ≤ pivot
    - The pivot is in its final position in the array
    - Subarray2 contains only elements ≥ pivot
  - Apply this procedure recursively to each subarray
    - Stop when the subarray is ≤ 1 in length

Review Questions

- What is the difference between primary and secondary key?
- What are data, record and field?
- Explain sequential search algorithm.
- What are time complexities of sequential search in best, average, and worst case?
- Explain binary search algorithm.
- What are time complexities of binary search in best, average, and worst case?
- Explain interpolation search algorithm.
- What are time complexities of interpolation search in best, average, and worst case?

Review Questions (Cont’d)

- What is the difference between internal and external sorting?
- What is a stable sorting?
- Explain bubble sort algorithm.
- What are time complexities of bubble sort in best, average, and worst case?
- Explain selection sort algorithm.
- What are time complexities of selection sort in best, average, and worst case?
- Explain insertion sort algorithm.
- What are time complexities of insertion sort in best, average, and worst case?

Review Questions (Cont’d)

- Explain merge sort algorithm.
- What are time complexities of merge sort in best, average, and worst case?
- What is the disadvantage of merge sort?
- Explain quick sort algorithm.
- What are time complexities of quick sort in best, average, and worst case?
- What are some possible approaches to select pivot in quick sort?

Any questions?